



# ENOVIA DesignSync User's Guide

3DEXPERIENCE 2022

©2022 Dassault Systèmes. All rights reserved. 3DEXPERIENCE®, the Compass icon, the 3DS logo, CATIA, SOLIDWORKS, ENOVIA, DELMIA, SIMULIA, GEOVIA, EXALEAD, 3D VIA, BIOVIA, NETVIBES, IRWE and 3DEXCITE are commercial trademarks or registered trademarks of Dassault Systèmes, a French "société européenne" (Versailles Commercial Register # B 322 300 440), or its subsidiaries in the United States and/or other countries. All other trademarks are owned by their respective owners. Use of any Dassault Systèmes or its subsidiaries trademarks is subject to their express written approval.



# Table of Contents

Overview .....	1
Overview.....	1
ENOVIA Synchronicity DesignSync® Data Manager Capability .....	1
Using ENOVIA Synchronicity DesignSync Data Manager User's Guide Documentation.....	1
Before Reading this Guide .....	2
Getting Started with DesignSync .....	2
Setting Up Your Work Area .....	2
Modifying Data .....	4
Staying Informed of Project Changes.....	4
What is Revision Control? .....	4
Contacting ENOVIA.....	6
Introduction to Merging .....	6
What Is Merging? .....	6
Merge Conflicts .....	7
Two-Way Merge.....	9
Three-Way Merge .....	9
Merge Edges.....	11
Merge Conflict Editor.....	13
Setting up a Project or Module Workspace .....	19
Accessing a SyncServer with User Authentication .....	19
Setting Up a Work Area for a Project.....	21
Joining a Project Using a Wizard.....	21

- What Is a Project? ..... 21
- Defining a Public Project ..... 23
- Displaying Project Properties ..... 23
- Browsing a Project Vault ..... 24
- Workspace Wizard ..... 24
- Joining a Project Step-by-Step ..... 45
  - Specifying the Vault Location for a Design Hierarchy ..... 45
  - Adding a Vault to Bookmarks ..... 51
  - Verifying That a Vault Has Been Set on a Folder ..... 51
  - Browse the Vault for a File or Project ..... 51
  - Viewing the Contents of a Vault ..... 51
- Populating Your Work Area ..... 53
  - Using a Mirror ..... 69
  - Setting Permissions for the Mirror ..... 72
  - Changing the Vault for a Design Hierarchy ..... 73
- Setting a Workspace Root ..... 73
- Setting Persistent Populate Views and Filters ..... 74
- Using DesignSync ..... 79
  - Populating Your Work Area ..... 79
    - Populate Field Descriptions ..... 84
  - Changing the State of Objects in Your Work Area ..... 95
  - Specifying Module Objects for Operations ..... 97
- Checking Out Design Data ..... 100

DesignSync Data Manager User's Guide

- Check Out Field Descriptions..... 101
- Canceling a Checkout..... 105
  - Cancel Checkout Field Descriptions ..... 107
- Checking In Design Data ..... 108
  - Check In Field Descriptions ..... 113
- Adding a Member to a Module..... 121
- Creating Branches ..... 124
  - Make Branch Field Descriptions..... 125
- Tagging Versions and Branches..... 127
  - Tag Naming Conventions..... 127
  - Tagging Module Snapshots ..... 128
  - DesignSync Objects for Tag ..... 129
  - Tag Field Descriptions ..... 131
- Unlocking Server Data..... 136
  - Unlock Field Descriptions..... 137
- Working with Exclude Files..... 139
  - Exclude File Processing..... 139
  - Exclude File Formatting ..... 140
  - Related Topics ..... 141
- Adding/Removing Exclusions ..... 141
  - Creating and Maintaining Exclusion Files ..... 141
  - Add Exclusion Using DesignSync Commands..... 142
  - Remove Exclusion Using DesignSync Commands ..... 142

- Related Topics ..... 142
- Viewing Exclusions ..... 142
  - Related Topics ..... 143
- Using Revision Control Keywords..... 143
  - Revision Control Keywords Overview ..... 143
  - Using Revision Control Keywords ..... 144
- Working with Modules ..... 149
  - Specifying Module Objects for Operations..... 149
  - Creating a Module ..... 152
  - Creating a New Version of a Module ..... 154
  - Adding a Member to a Module..... 155
  - Creating a Hierarchical Reference..... 158
  - Removing a Member from a Module..... 163
  - Deleting a Hierarchical Reference ..... 165
  - Locking Module Data ..... 166
  - Setting a Workspace Root ..... 167
  - Rolling Back a Module..... 168
  - Deleting a Module..... 170
  - Deleting a Module Cache Link..... 174
    - Delete Field Descriptions ..... 175
  - Resolving Module Structure Conflicts..... 176
    - Related Topics ..... 177
  - Overlaying Module Data ..... 177

Synchronizing Enterprise Developments .....	180
Running the Synchronize command .....	180
See Also.....	181
Reference .....	181
What Is a Module? .....	181
Data Management of Modules .....	185
Operating on Module Data .....	186
Auto-Merging.....	189
Understanding Module Views .....	192
Filtering Module Data .....	196
Module Recursion .....	200
Module Locking.....	204
Module Hierarchy .....	206
Folder Versioning .....	215
Module Branching .....	216
Merging Module Data.....	219
Module Merging .....	224
External Modules .....	228
Module Member Tags .....	230
Edit-In-Place Methodology .....	233
Understanding Smart Module Detection .....	235
Conflict Handling .....	236
Module Version Updating.....	237

- Using a Module Cache ..... 238
- Working with Files and Directories ..... 241
  - Creating Files ..... 241
    - New File Field Descriptions..... 241
  - Moving and Renaming Files ..... 242
  - Adding a Member to a Module..... 242
  - Moving a module member ..... 245
    - Some notes on moving folders..... 245
  - Using the Moving Modules Members dialog box ..... 247
    - Renaming a module member ..... 248
  - Creating Folders ..... 250
    - New Folder Field Descriptions ..... 251
  - Moving and Renaming Folders ..... 252
- Removing Objects ..... 252
  - Deleting Design Data ..... 252
  - Deleting Files ..... 253
  - Deleting Folders ..... 255
  - Deleting Server Folders ..... 258
  - Deleting Vaults ..... 261
  - Deleting Versions from a Vault..... 264
  - Retiring Design Data ..... 267
  - Removing a Member from a Module ..... 270
- Comparing Files ..... 273

## DesignSync Data Manager User's Guide

Common Diff Operations .....	273
Advanced Diff Options .....	274
Field Descriptions.....	276
Reading Diff Results .....	280
Display diff-annotated file (revised format).....	281
Display only the diffs (standard format).....	281
Display only the diffs (unified format) .....	282
Display only the diffs (syncdiff format).....	283
Display diff-annotated file .....	284
Display Output in GUI .....	285
Revised Diff Format .....	285
Using Revised Diff Format .....	286
Revised Diff Format Actions.....	287
Related Topics .....	288
Graphical Diff Utility .....	288
Using Graphical Diff format .....	288
Graphical Diff Format Tools .....	289
Displaying Information.....	293
Showing Potential Checkouts .....	293
Identifying Changed Objects.....	293
Displaying Contents of Vault Data .....	295
Contents Field Descriptions .....	296
Displaying Contents of Vault Data .....	298



Contents Field Descriptions .....	299
Displaying a Module Cache .....	301
Displaying Module Hierarchy .....	302
To display module hierarchy .....	304
Displaying Module Status .....	306
Displaying Module Views .....	309
Displaying Module Where Used .....	309
Running the Where Used command .....	310
Understanding the Where Used command output .....	311
Where Used Actions .....	312
Vault browser object context menu .....	313
Displaying Enterprise Objects.....	313
Compare the Contents of Two Areas.....	314
Compare Workspaces/Selectors Field Descriptions .....	316
Compare the Contents of Two Areas.....	319
Compare Workspaces/Selectors Field Descriptions .....	322
Displaying Version History.....	325
Version History Field Descriptions .....	326
Controlling the Display of Module Information .....	331
Displaying module versions .....	331
Using Display Filters .....	331
Exploring Modules .....	332
Annotate Tool .....	333

DesignSync Data Manager User's Guide

- Using Annotate..... 333
- Annotate Actions ..... 335
- Highlighting the Annotate results..... 337
- Vault Browser Tool ..... 340
- Vault Browser Overview ..... 340
- Vault Browser Actions ..... 344
- Vault Browser Tools ..... 346
- Filter Interesting Dialog ..... 349
- Finding Objects in the Vault Browser ..... 350
- Working in SITaR ..... 355
- Using SITaR as a SITaR Designer ..... 355
- The Designer Role ..... 355
- Creating a Workspace..... 355
- Editing a Sub-Module ..... 356
- Synchronizing a Module with the Baseline ..... 356
- Submitting a Sub-Module for Integration ..... 357
- Synchronizing all Sub-Modules with the Baseline ..... 357
- Branching a Sub-Module..... 358
- Using SITaR as a SITaR Integrator ..... 359
- The Integrator Role ..... 359
- The Integration Workspace ..... 359
- Creating an Integration Workspace ..... 360
- Workflow for Updating the Container Module..... 360

Locating Submitted Modules for Integration .....	361
Selecting Sub-Modules for Integration .....	361
Integrating Selected Changes into the Container Module .....	362
Testing the Integration Version of the Container Module .....	362
Locating a Context Module Version .....	363
Recreating the Developer's Workspace .....	363
Releasing a New Baseline .....	364
Branching a Container or Sub-Module .....	364
Configuring SITaR .....	365
SITaR Environment Variables .....	365
Sample SITaR Environment Variable File .....	368
Creating a SITaR Container Module .....	369
Defining and Enabling Module Context .....	370
Creating a SITaR Sub-Module .....	371
Creating an Initial Baseline Release .....	372
Branching a Container or Sub-Module .....	374
Reference .....	374
Overview of SITaR Workflow .....	374
SITaR Module Structure .....	375
Branching in SITaR .....	377
Techniques .....	379
Getting Started with the GUI .....	379
Using the DesignSync GUI .....	379

## DesignSync Data Manager User's Guide

Assumed Environment: .....	379
Creating a Work Area.....	380
Creating a Work Area.....	382
Creating File Versions.....	383
Configuration/Release Management.....	386
Working with Files in Your DesignSync Work Area.....	391
Getting Started with the Command Shell.....	391
Using the DesignSync Command Shell .....	392
Assumed Environment.....	392
Creating a Work Area - Putting Files Under Revision Control.....	393
Creating a Work Area - Joining a Project Already Under Revision Control.....	395
Creating File Versions.....	397
Configuration/Release Management.....	400
Working with Files in Your DesignSync Work Area.....	405
Tutorials .....	407
Creating Modules and Module Data .....	407
Module Hierarchy: Module Structure.....	407
Creating Module Hierarchy: Overview.....	407
Creating Module Hierarchy: Create the Module .....	408
Creating Module Hierarchy: Add Files and Check In.....	408
Creating Module Hierarchy: Add an HREF to a Module in the Workspace .....	409
Creating Module Hierarchy: Populate with Dynamic HREF Mode .....	410
Creating Module Hierarchy: Add an HREF to a Module not in the Workspace ....	411

Creating a Peer Structure Module Hierarchy .....	412
Updating Module Hierarchy .....	413
Modifying Module Hierarchy: Overview .....	413
Modifying Module Hierarchy: New "Gold" Version of ALU Created .....	413
Modifying Module Hierarchy: Chip Team Uses New ALU Version .....	414
Modifying Module Hierarchy: CPU Team Reverts to Earlier ALU Version .....	415
Moving a File .....	416
Moving a Folder .....	417
Operating with Module Data .....	418
Operating on a Module .....	418
Operating on a Module's Contents .....	419
Filtering .....	420
Persistent Populate Filter .....	421
Folder-Centric Operations .....	422
Module-Centric Operations on a Module .....	423
Module-Centric Operations on a Subfolder .....	424
Module-Centric Operations on an HREF .....	425
Locking a Module Branch .....	426
Locking Module Content .....	427
Branching a Module .....	428
Merging and Modules .....	429
Auto-Merging Locally Added Files .....	429
Auto-Merging Locally Modified Files .....	430

## DesignSync Data Manager User's Guide

Auto-Merging Locally Modified Files Removed from the Module .....	431
Auto-Merging Non-Latest Locally Modified Files .....	432
Auto-Merging Locally Modified Files Renamed in the Module .....	433
Auto-Merging Locally Modified Files with Other Files Renamed in the Module....	434
In-Branch Merging of Locally Added Files.....	435
In-Branch Merging of Locally Modified Files .....	436
Step-by-Step Use Cases .....	437
Creating Modules and Module Data.....	437
Updating Module Hierarchy.....	484
Operating with Module Data.....	516
Merging and Modules.....	593
Reference.....	633
Understanding the DesignSync Architecture .....	633
DesignSync Architecture.....	633
What Is a SyncServer? .....	634
Object States.....	634
Object Types.....	636
Object Properties .....	637
URL Syntax.....	645
DesignSync URLs .....	648
Revision Control Status Values.....	649
Vaults, Versions, and Branches .....	651
Introduction to Data Replication .....	652

Metadata Overview .....	653
Mirrors .....	655
Understanding the GUI Interface .....	662
Using the Classic DesignSync GUI .....	662
Using the Workspace Structure Browser .....	663
DesignSync Symbols and Icons.....	664
Toolbars and Menus .....	668
Classic Windows and Panes .....	692
Workspace Structure Browser Windows and Views.....	702
DesignSync Shells.....	710
DesignSync Command Line Shells .....	710
Comparing the DesignSync Shells.....	711
Invoking a DesignSync Shell.....	712
Command Line Editing.....	714
Working with Command Aliases.....	719
Configuring the DesignSync Interface .....	720
Configuring DesignSync.....	720
Controlling Access to Your Local Work Area .....	721
Setting Up a Shared Work Area .....	721
Moving a Work Area.....	723
UNIX Permissions of Work Areas and Vaults .....	726
Command Line Defaults System.....	728
Working with Scripts .....	728

## DesignSync Data Manager User's Guide

DesignSync Scripts .....	728
Using DesignSync Commands in OS Shell Scripts.....	728
Creating DesignSync Scripts .....	730
Running Scripts.....	731
Running a Script at Startup .....	732
Improving Efficiency Using Caches and Mirrors .....	733
Mirrors Versus Caches.....	733
What is a File Cache? .....	733
Why Use a File Cache? .....	735
What is a Module Cache? .....	735
Mirroring Overview .....	736
Locking, Branching, and Merging .....	738
What Is Merging? .....	738
Locking and Merging Work Styles.....	739
Selecting Versions and Branches .....	744
Parallel (Multi-Branch) Development.....	763
Working with Legacy Modules .....	784
How DesignSync Handles Legacy Modules.....	784
Upgrading Legacy Modules .....	785
Upgrading DesignSync Vaults .....	791
Managing Legacy Configurations and REFERENCES .....	795
Collections .....	808
Collections Overview.....	808



Displaying Collections .....	809
Cadence Collections.....	810
Cadence Design Objects Overview.....	810
Enabling Cadence Object Recognition.....	812
How DesignSync Recognizes Cadence Data .....	812
How DesignSync Manages Cadence Objects.....	814
Managing Non-Collection Objects.....	815
Custom Type Package Collections .....	816
Custom Type Package Collections Overview .....	816
How DesignSync Recognizes CTP Data .....	817
Integration with ENOVIA Program Central.....	818
Using the ENOVIA Semiconductor Accelerator for DesignSync Central.....	818
Using the ENOVIA Semiconductor Accelerator for IP Management .....	818
User Interface.....	819
Performing GUI operations .....	819
Selecting Objects .....	819
Going to a Location .....	819
Navigating the Tree View .....	820
Adding, Editing, and Organizing Bookmarks.....	821
Defining and Modifying Bookmark Properties .....	823
Searching for Text.....	823
Reviewing History .....	824
Using Data Sheets .....	825

## DesignSync Data Manager User's Guide

Setting the Verbosity of the Output Window.....	826
Viewing the Results of an Operation.....	826
Common Interface Topics.....	827
Comment Field.....	827
Exclude Field.....	829
Filter Field .....	830
Force Overwrite of Local Modifications Option.....	831
Href Filter Field.....	831
Keys Field .....	832
Local Versions Field.....	832
Module Context Field .....	833
Module Views Field .....	834
Populate Log.....	835
Recursion Option .....	837
Retain Timestamp Field .....	837
Suggested Branches, Versions, and Tags.....	837
Trigger arguments.....	838
Command Invocation .....	838
Command Buttons .....	838
Get Tags/Versions .....	839
Select a path.....	840
Select Module Context.....	841
Select Module Instance .....	842

- Select Parent Module ..... 842
- Select Vault URL Browser ..... 843
- Filter Interesting Dialog..... 843
  - Related Topics ..... 844
- Select a Member Descendant ..... 845
  - Related Topics ..... 845
- Index ..... 847

# Overview

## Overview

ENOVIA Synchronicity DesignSync® Data Manager provides a graphical interface to the revision control operations that DesignSync provides. This helps documents both the graphical user interface and many of the concepts and working models that DesignSync supports.

## ENOVIA Synchronicity DesignSync® Data Manager Capability

DesignSync provides the major revision control functionality available in DesignSync, including, among others, the following concepts:

- What is Revision Control?
- DesignSync Architecture
- What Is a Design Configuration?
- What Is a Module?
- Overview of SITaR Workflow
- What Is a Project?
- Parallel (Multi-Branch) Development
- What Is a SyncServer?
- What is Merging?
- Introduction to Data Replication
- Mirroring Overview

and the following major revision control operations:

- Creating a Workspace
- Populating Your Work Area
- Checking out Design Data
- Checking in Design Data
- Adding a Member to a Module
- Creating Branches
- Tagging Versions and Branches
- Retiring Design Data/Removing a Member from a Module
- Creating a Hierarchical Reference

## Using ENOVIA Synchronicity DesignSync Data Manager User's Guide Documentation

This guide is single-sourced in HTML and generated to multiple locations.

- Integrated help - When you press F1 within the DesignSync (DesSync on UNIX) application or click on the Help button on DesignSync dialog boxes, the

DesignSync Data Manager's User's Guide help appropriate for the location or dialog opens in your default Web browser.

- DesignSync Documentation - available from the **Dassault Systems** product group in the Windows **Start** menu or on UNIX, by pointing your web browser to `$SYNC_DIR/share/content/doc/index.html`

**Note:** References from the *ENOVIA Synchronicity DesignSync Data Manager User's Guide* to the *ENOVIA Synchronicity Command Reference* guide always link to the ALL version of the guide, which contain information about all working methodologies for DesignSync. For more information about the available working methodologies, see *ENOVIA Synchronicity Command Reference*.

## Before Reading this Guide

This guide has no prerequisite guides because it contains all the introductory material to get started with DesignSync. If there is no client installed on your system, you may need to install the client. The client installation documentation is located in the Program Directory.

## Getting Started with DesignSync

This topic provides instructions to help you set up a work area and get started using ENOVIA Synchronicity DesignSync® Data Manager to manage your design data.

In the following scenario, assume that your project lead has checked in data for the project you are working on, creating vaults on a SyncServer. Your project lead provides you with the DesignSync URL for the project's vault data or the name of the project on the SyncServer. Your first step is to set up a work area for the project.

## Setting Up Your Work Area

You can set up a work area using any of these four methods:

**Method 1: Use the DesignSync GUI to manually set up your work area.**

1. Using the DesignSync GUI, specify the vault location to associate with your workspace.
2. If your project lead provided a mirror directory path for the project, use the `setmirror` command to associate your workspace with the mirror directory. Use the Command Shell Window to run commands in the GUI.
3. Fetch data from the vault. Your project lead will have set a default object state, with that value pre-selected in the Populate dialog box. For example, if the project has an associated mirror directory, the default setting in the Populate dialog box will be to populate your workspace with "Links to mirror".

### **Method 2: Use a DesignSync shell to manually set up your work area.**

1. Using a DesignSync Shell, use the `setvault` command to associate your workspace with the vault location.
2. If your project lead provided a mirror directory path for the project, use the `setmirror` command to associate your workspace with the mirror directory.
3. Use the `populate` command to fetch data from the vault. Your project lead will have set a default object state. For example, if the project has an associated mirror directory, `populate` will create symbolic links from your workspace to files in the mirror, as if the `-mirror` option had been specified on the `populate` command line.

### **Method 3: Use the Workspace Wizard to set up your work area.**

1. Invoke the Workspace Wizard.
2. From the Workspace Wizard, you can select, or browse to, the project to join. Or you can manually type in its DesignSync URL, in the Wizard page from which you specify a project vault.
3. In a subsequent Wizard page you will specify a workspace.
4. Your project lead will have set a default object state for the kind of data to fetch, with that value pre-selected in the Wizard page from which you specify an object state. For example, if the project has an associated mirror directory, the default setting will be to populate your workspace with "Links to mirror".
5. If "Links to mirror" will be populated, the next Wizard page prompts you to specify the mirror directory. When you finish setting up with the Wizard, DesignSync populates your workspace with data from the vault.

### **Method 4: Set up your work area by joining a project.**

1. Select a project.
2. Launch the Workspace Wizard, using **Revision Control | Workspace Wizard**. The Wizard launches positioned to the Wizard page in which you specify a workspace. You will not be prompted to specify the DesignSync URL of the project. That information is already known, from your having invoked the Wizard from the project you are joining.
3. Your project lead will have set a default object state for the kind of data to fetch, with that value pre-selected in the Wizard page from which you specify an object state. For example, if the project has an associated mirror directory, the default setting will be to populate your workspace with "Links to mirror".
4. If "Links to mirror" will be populated, the next Wizard page prompts you to specify the mirror directory. When you finish setting up with the Wizard, DesignSync populates your workspace with data from the vault.

You now have project data in your workspace. You might want to add a bookmark, to easily revisit this workspace. You should populate your workspace periodically, to update your workspace with the project's current data.

## Modifying Data

If your team is using the lock model, before modifying a file, you can check out the file with a lock. Checking out with a lock fetches a local, writable copy of the file and prevents anyone else from checking in their modifications to the file. You can check out files using the GUI's Checkout dialog box, or using the `co` command. You perform a Checkout based on the files that you already have in your workspace. To see what files have been added to the project since you last populated your workspace, you can Show Potential Checkouts using the GUI.

Double-click on a file to open it in your default editor. After you have edited some files, you can identify modified files in your workspace by running the GUI's Modified Objects report. From the report's results, you can select objects on which to operate. Right-click on a file in the report listing to display a context menu. You can show your local modifications to the file, or compare your locally modified file to the Latest version.

If you decide not to modify a file, you can cancel your checkout either by using the GUI's Cancel Checkouts dialog box, or by using the `cancel` command. You can also discard your local modifications by fetching a new copy of the file into your workspace, as part of the cancel operation.

When you are ready to check in your changes to the project vault, use the GUI's Check In dialog box, or the `ci` command to create a new version of each file that you modified. The check-in operation also releases the locks on those files that you previously obtained when you checked out the files.

## Staying Informed of Project Changes

To see the revision control history of a managed object, you can view its Data Sheet or show its Version History.

To receive e-mail notification of design changes to the project resulting from DesignSync operations or notification of bugs entered against the project, subscribe for e-mail notification of activity against the project.

## What is Revision Control?

Revision control is the ability to freeze the state of an object, such as a file, at various stages in its development. Each time a file is frozen, a new and distinct version of the file is created. The existence of the version lets you continue to change the file with the full confidence that if you make an error, you can always revert to the previous version. It is also possible to compare the current state of the file to an earlier state, and maintain a log of what changed with each version. You can also merge changes when more than one person makes changes to the same design object.

You create a version with the checkin operation and DesignSync stores the version in a vault. Once in the vault, any person who has access to that vault can retrieve the version. When using the locking model, only one person at a time should be "officially" changing the data. When you want to retrieve an object in order to change it, you can perform a checkout operation with a lock; this gives you an exclusive lock on that object. An alternative model, called the non-locking or merge model, does not require a lock on checkout; this checkout is often referred to as a fetch operation. When you complete your changes, you check the object back in, creating another new version of the data.

At any point in time it might be useful to take a specific version and create a branch emanating from it. The branch can then have a sequence of versions checked into and out of it, creating another thread of development for the stored object. A version that is the root of a new branch is referred to as a branch point version. As an example, consider a Verilog file that is part of an ASIC design. The ASIC is in verification and a new revision of the ASIC is in the works. During verification, the team finds it needs to modify the Verilog file for the original ASIC but not for the new revision of the ASIC; the team branches the Verilog file. The team makes the changes associated with the first ASIC on the branch, while the new work occurs on the original thread of development. The concept of "one person at a time" editing really means "one person at a time, per branch".

### **Related Topics**

Vaults, Versions, and Branches

Viewing the Contents of a Vault

Object Types

Object States

Specifying the Vault Location

Checking in Design Files

Checking out Design Files

Undoing a Check Out

Populating Your Work Area

Tagging Versions and Branches

Retiring Branches

Deleting Files or Versions from a Design Project



Parallel (Multi-Branch) Development

## Contacting ENOVIA

For solutions to technical problems, please use the 3ds web-based support system:

<http://media.3ds.com/support/>

From the 3ds support website, you can access the Knowledge Base, General Issues, Closed Issues, New Product Features and Enhancements, and Q&A's. If you are not able to solve your problem using this information, you can submit a Service Request (SR) that will be answered by an ENOVIA Synchronicity Support Engineer.

If you are not a registered user of the 3ds support site, send email to ENOVIA Customer Support requesting an account for product support:

[enovia.matrixone.help@3ds.com](mailto:enovia.matrixone.help@3ds.com)

### Related Topics

Using Help

## Introduction to Merging

### What Is Merging?

Merging combines changes made on two branches, or within a branch. Branching without merging has limited usability – two developments streams diverge forever and can never be reconciled.

There are several important notions related to the general concept of merging that are equally applicable to file merging and module merging:

Merge Conflicts

Two-Way Merge

Three-Way Merge

Merge Edges

Module merging has additional factors to consider because instead of merging specific files, you are merging a set of changes. For more information on module merging, see [Merging Module Data](#).

### Related Topics

[Merging Module Data](#)

[Using the Merging Work Style](#)

[Parallel \(Multi-Branch\) Development](#)

[Merge Conflicts](#)

[Two-Way Merge](#)

[Three-Way Merge](#)

[Merge Edges](#)

## Merge Conflicts

Whenever you merge, whether the merge is between two versions on the same branch (see [Using the Merging Work Style](#)) or between branches (see [Parallel \(Multi-Branch\) Development](#)), there may be merge conflicts. Merge conflicts can also arise when changes in the two branches being merged are incompatible, such as when a file is renamed to different locations on the two branches. Merge conflicts occur when different changes were made to the same region of the two versions that are being merged. DesignSync cannot automatically determine which changes are the correct ones; you must resolve the conflicts manually.

When working with modules, it is not necessary to resolve all conflicts before creating the next module version. You can check in, or perform other operations that create new module versions, as long as the objects being operated on are not the ones that are in conflict.

### Note:

DesignSync records "merge edges" – information about what versions participated in the merge -- with the new version resulting from the merge. DesignSync uses merge edges in future calculations of closest common ancestors instead of always going back to the original ancestor. This capability relieves you from having to resolve the same merge conflicts during future merges. See [Merge Edges](#) for more information.

## Resolving Merge Conflicts

A conflict is presented in a textual or graphical format as two options. You resolve the conflicts either one or at a time, or collectively by selecting the appropriate version.

DesignSync alerts you to conflicts during the merge. Conflicts are identified in the Changed Object Browser, by the Status field of the List View and from the **Is** command. You can also use the **url inconflict** command.

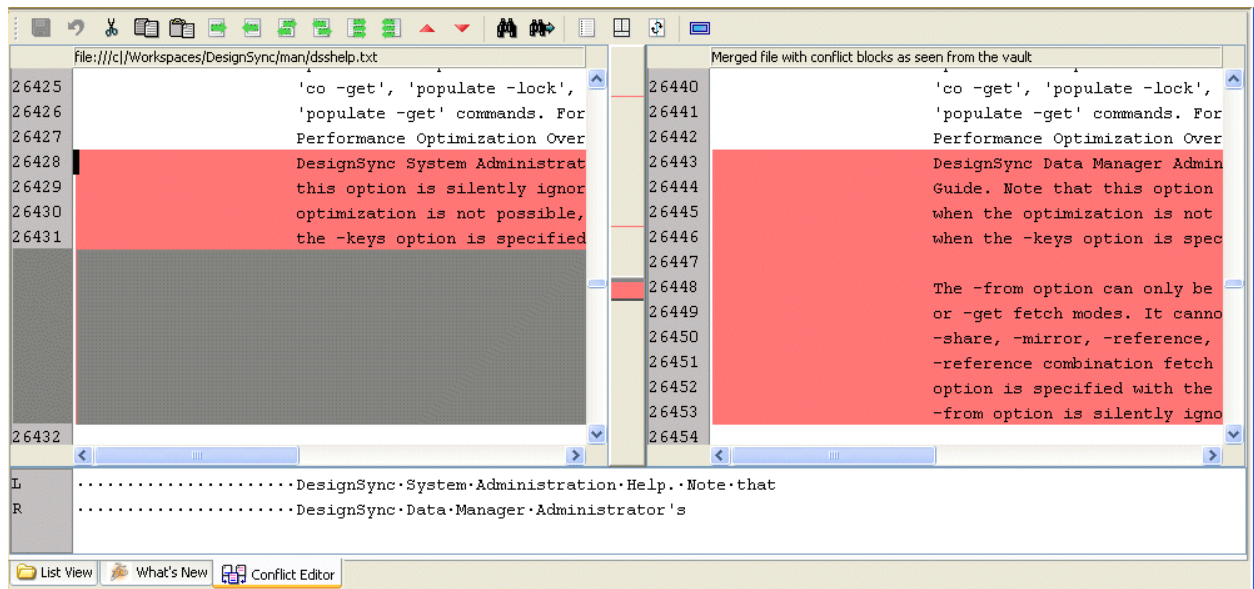
When you merge a file, the conflicts are indicated in the file text with a conflict delimiter (exactly 7 less-than, greater-than, or equal signs starting in column 1) and the version number to indicate what text is present in each version:

```
<<<<<<< versionID
Lines from Latest version (same-branch merge) or overlaid
version
Lines from locally modified version
>>>>>>>
```

DesignSync considers the conflicts resolved when the file no longer contains any of the conflict delimiters.

To invoke the Conflict Editor, select the Resolve Conflicts action from the context menu of an object that's identified as **In Conflict** by the Changed Object Browser. Or select an in conflict file and run **Tools => Resolve Conflicts**.

In the Merge Conflict Editor, the conflicts are indicated by highlighted text.



DesignSync considers the conflicts resolved when the file no longer contains any of the conflict delimiters. When you resolve a conflict, the Merge Conflict Editor removes the

conflict indicators for you and changes the highlight color to the resolved conflict color, light pink by default.

**Note:** The highlighted text uses the color specified for conflict resolution with SyncAdmin, in the **For multi-window Diff viewers/editors** section.

### Related Topics

Two-Way Merge

Three-Way Merge

Conflict Handling

Identifying Changed Objects

Locking and Merging Work Styles

Merge Conflict Editor

Using the Merging Work Style

Parallel (Multi-Branch) Development

ENOVIA Synchronicity Command Reference: ls

ENOVIA Synchronicity Command Reference: url inconflict

ENOVIA Synchronicity Command Reference: populate

## Two-Way Merge

In a two-way merge, differences between two objects are present as merge conflicts. This is the simplest merge approach, but it can lead to a significant number of conflicts.

For example, suppose one programmer modified one function in a source file while another programmer modified an unrelated function on a different branch. Two-way merge will present both modifications as merge conflicts.

### Related Topics

Merge Conflicts

Three-Way Merge

## Three-Way Merge

A three-way merge is more complicated than a two-way merge. The benefit of a three-way merge is that it minimizes the number of conflicts requiring resolution, by automatically resolving some differences.

Let's take the example described in the Two-Way Merge:

Suppose one programmer modified one function in a source file while another programmer modified an unrelated function on a different branch.

A three-way merge assumes that both objects participating in the merge were derived from a common earlier version of the object. That common earlier version of the object is referred to as the *base version*. There are therefore three versions of the object participating in the merge; thus the merge is a 3-way merge.

A three-way merge calculates two sets of differences:

- The difference between the base version and the first version
- The difference between the base version and the second version

Differences from these two sets are called *intersecting* if they involve a common part of the base object.

In the example above, changes made by the programmers to the two different functions are *non-intersecting* changes. Non-intersecting changes are presumed to result from unrelated improvements made to the base object. They are accepted automatically, resulting in a merged object with non-intersecting changes from both versions. Intersecting changes are shown as merge conflicts.

#### Choosing the Right Base Version

The base version is the version from which both objects participating in the merge were derived. When merging end points of two branches, a good candidate for a base version is the closest common branch-point version.

You can derive the closest common branch-point from the dot-numeric representations of branch names. Find a longest common prefix of two names ending with a dot, and remove the last dot.

If the resulting string designates a version, then that version is the closest common branch-point. This is also known as the closest common ancestor.

For example, the longest common prefix of 1.2.2.1.1.8 and 1.2.2.1.3.1 is 1.2.2.1. Removing the last dot leaves 1.2.2.1, which is a version, and therefore the closest common branch-point.

If the resulting string designates a branch, then the closest common branch-point will be the smaller endpoint branch-point version on this branch.

For example, the longest common prefix of 1.2.2.1.1.8 and 1.2.2.3 is 1.2.2. Removing the last dot leaves 1.2.2, which is a branch. 1.2.2.1 (in 1.2.2.1.1.8) is smaller than 1.2.2.3 (in 1.2.2.3), so the closest common branch-point version is 1.2.2.1.

### Related Topics

Merge Conflicts

Two-Way Merge

Merging Module Data

## Merge Edges

While the closest common branch-point version is usually a good candidate for the base version of a 3-way merge, in certain cases it is possible to find a better candidate.

For example, let's say the main development of a product takes place on the Trunk branch. Versions on the Trunk are periodically tagged (for example, bp-rel33, bp-rel40, bp-rel41, bp-rel42), and released to customers as major releases. To accommodate bug fixes, service pack branches are created for each release (rel33sp, rel40sp, rel41sp).

Some versions on service pack branches are tagged as service pack releases (rel41sp1, rel41sp2, rel42sp1).

How should bug fixes from service pack branches be incorporated into the main development branch? An appealing approach might be to merge a service pack branch into Trunk each time a service pack is released. This method generates a set of merges rel41sp->Trunk for every service pack of rel41. What would be the base versions for the merges? The closest common branch-point for all these merges will be the same, bp-rel41. But always using bp-rel41 as a base version results in a repetitive resolution of the same conflicts.

Suppose the fix of bug #12345 was released in rel41sp1, and merging of this fix into the Trunk resulted in a conflict. In other words, diff(bp-rel41, rel41sp1) and diff(bp-rel41, Trunk:Latest) intersected. This happened because the Trunk developer stumbled onto the same bug and fixed it in a different way than the rel41sp1 developer. The resolution of the merge conflict was to use the fix from rel41sp1.

The conflict reappears when rel41sp2 is later merged onto Trunk. Although the section of code pertaining to the fix of bug #12345 was not modified on either branch since the merge, the code fix for bug #12345 still shows as a conflict, because corresponding differences diff(bp-rel41, rel41sp2) and (bp-rel41, Trunk:Latest) still exist and intersect.

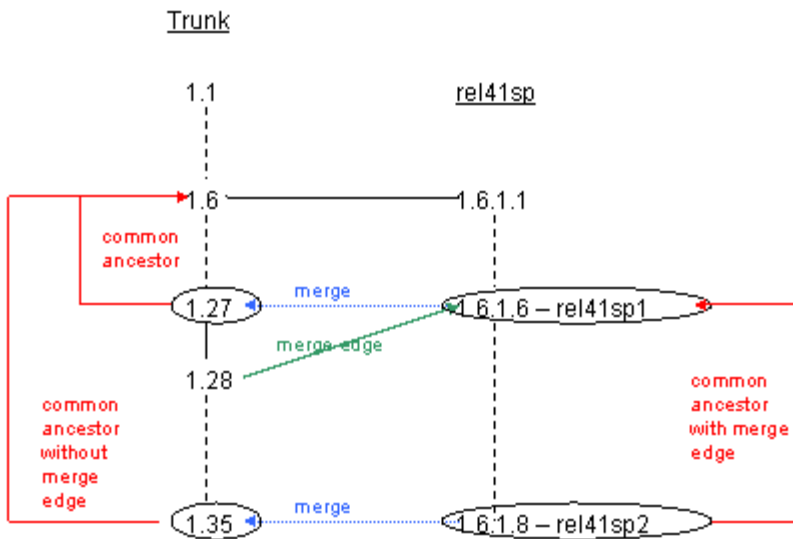
In this case, it is better to choose rel41sp1 as the base version, because then there would be no differences related to the code fix for bug #12345 between rel41sp1 and

rel41sp2. The conflict would then have been resolved automatically, as in the previous merge.

As more development is done on two branches, the branches continue to diverge from each other, making merges more and more difficult. By choosing appropriate base versions we can completely eliminate this issue. To select the right base version we need to discuss *merge edges*.

A merge edge is an edge in a history graph of an object from one participant of a merge to the merge result. The other participant has a direct edge (it is the parent of the new version).

For example:



Without merge edges the history graph is simply a tree. With merge edges the history graph becomes a directed acyclic graph.

Based on merge edges, we can choose the base version for a new merge as the closest common predecessor of the two versions participating in the merge. The advantage of this approach is that the base version chosen is much “closer” to these versions than the closest common branch-point, resulting in fewer differences and conflicts.

In the above diagram, we merged Trunk:27 and rel41sp1, resolved the conflicts and checked in the result as Trunk:28. The merge edge is version rel41sp1 (version 1.6.1.6). For the next merge (Trunk:35 to rel41sp2) the previous merge edge (version 1.6.1.6) is used to find the base version rel41sp1, so that none of the conflicts resolved in the previous merge reappear.

The **vhistory** command and the Version History report both indicate merge edges.

### Related Topics

Merge Conflicts

Two-Way Merge

Three-Way Merge

Merging Module Data

ENOVIA Synchronicity Command Reference: vhistory

## Merge Conflict Editor

DesignSync contains a built-in merge conflict editor to assist with conflict resolution. This utility has the ability to highlight and merge conflicts between the two versions of a selected file.

### There are two ways to open a file in the Merge Conflict Editor:

1. In the Changed Objects Browser, select a file designated as In Conflict. From the file's context menu, select **Resolve Conflicts**.
2. Select a text file in the View Pane. The file must be a writeable, merged file containing conflicts in the local workspace. Select **Tools => Resolve Conflicts** to open the Conflict Editor.

The result of the Resolve Conflicts operation is displayed in a new tab labeled **Conflict Editor**, in the View Pane.

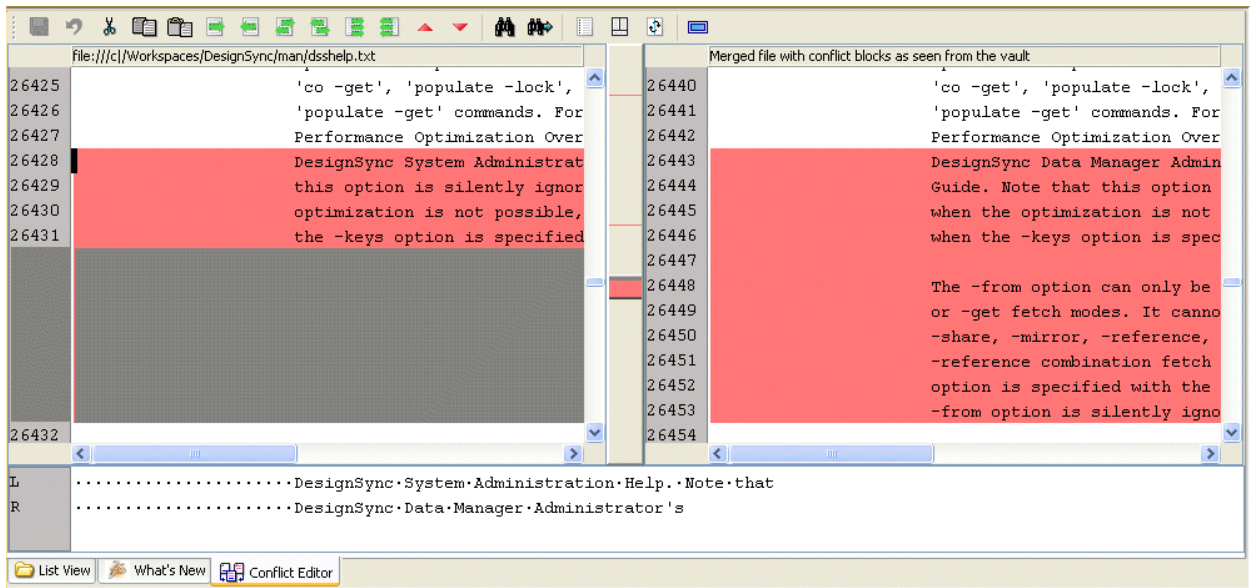
### Using the Merge Conflict Editor

The Resolve Conflicts command results in a new tab labeled **Conflict Editor**, in the View Pane. The Conflict Editor shows two files, side-by-side. The file on the left is the local workspace file that is in conflict. The file on the right side is the vault version with conflict blocks merged into it. Between the files is a scroll-bar which highlights the location of the differences in the file. This central bar shows symbolically all differences in the file as well as the current vertical position of the left and right windows in respect to the file. The knob on the central bar represents the current left and right visible windows in relation to the whole file. The colored bands on the knob represent currently visible diff and conflict blocks. Clicking on the left mouse button with the cursor on the central bar moves the left and right window content to the corresponding vertical position.

The left window is the editable local version of the file. You can copy and paste into this version. When you resolve a conflict in the file, the resolved text, regardless of which



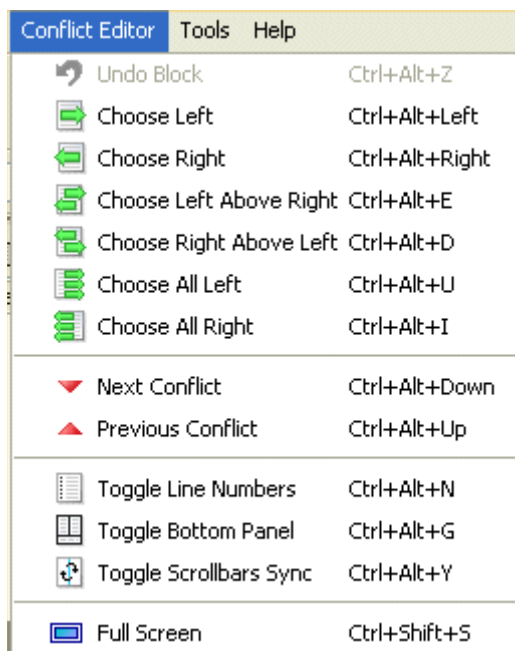
version the resolution came from, displays in the version in the left window. The resolved text displays in light pink by default.



You can customize the color used for conflicts and resolved conflicts in SyncAdmin, in the **For multi-window Diff viewers/editors** section.

## Merge Conflict Editor Tools

The Conflict Editor provides a set of tools to help you navigate the information provided and resolve the merge conflicts. The tools are available both from a **Conflict Editor** menu and a set of controls on the top of the Conflict Editor window.





### **Save the View**

Saves the changes made in the left (editable) window to the local workspace file. This option is only active when you've made a change to the window.

### **Undo Block**

Undo Block undoes all work on a block, restoring the selected block to its initial state.

### **Cut**

Cuts the selected text.

### **Copy**

Copies the selected text.

### **Paste**

Pastes text from the clipboard into the left (editable) window.

### **Choose Left**

The left part of the conflict (possibly modified) is kept; the conflict is marked as resolved.

### **Choose Right**

The right part of the conflict replaces the left part; the conflict is marked as resolved..

### **Choose Left Above Right**

The right part of the conflict is placed below the left part; the conflict is marked as resolved.

### **Choose Right Above Left**

The right part of the conflict is placed above the left part; the conflict is marked as resolved.

### **Choose All Left**

Left parts of all unresolved conflicts are used; all conflicts are marked as resolved.

### **Choose All Right**

Right parts of all unresolved conflicts are used; all conflicts are marked as resolved.

**Next Conflict**

Moves the marked window focus to the next unresolved conflict in the file. If you're at the bottom of the file, it may be necessary to get back quickly to the top of the file and start reviewing conflicts again. To do, use the knob in the central bar. Click at the top of the central bar to move the knob to the top and scroll the left and right windows to the top. Then, click in the left or right window to move the conflict "caret" marker. Next and Previous Conflict now start from that caret point.

**Previous Conflict**

Moves the marked window focus to the previous unresolved conflict in the file.

**Find**

See Searching for Text.

**Find Next**

Advances to the next instance of the text specified in the **Find** window.

**Toggle Line Numbers**

By default, the line numbers display on the left side of the each panel. You can select this option to toggle whether line numbers display.

**Toggle Bottom Panel**

By default, there is a bottom panel below the line compare windows containing the file text that shows exactly what the change is, indicating which change is part of which version. You can select this option to toggle whether this window displays. Even if this window isn't displayed, the file text window still shows the conflict information.

**Toggle Scrollbars Sync**

By default, the scrollbar between the two windows keeps the window synchronized with each other. When you move forward in one window, DesignSync rolls the other window forward. You can select this option to enable or disable this feature. When the scrollbar sync is disabled, scrolling in one window does not advance the other window to match the content.

**Full Screen**

By default, the Conflict Editor opens in a tab in the View panel. This option allows you to toggle between that view and a Conflict Editor that takes up the entire screen.

**Note:** The Full Screen display can be placed in the background to allow you to work in the DesignSync GUI before returning to the display.

## DesignSync Data Manager User's Guide

### **Related Topics**

What Is Merging?

Identifying Changed Objects

Merge Conflicts



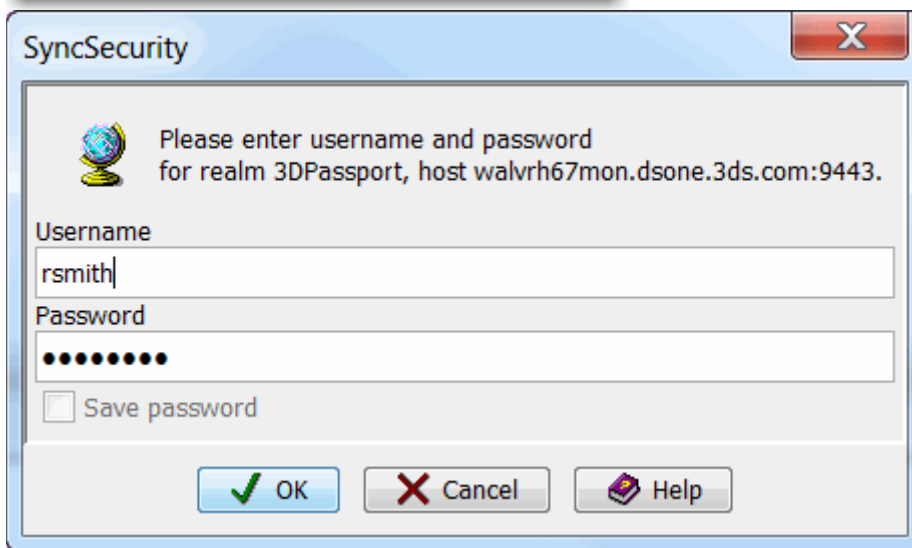
# Setting up a Project or Module Workspace

## Accessing a SyncServer with User Authentication

Your project leader may have configured your enterprise system or DesignSync server to require username and password authentication when accessing vault data on a SyncServer (client vaults never require authentication).

If the 3DPassport Central Authentication Server (CAS) is enabled at your site, then DesignSync supports a single-sign on model that allows you to sign in once and use all the Enterprise tools available to you including all the DesignSync graphical and command-line clients, the DesignSync Web Interface, ProjectSync, and all your other 3DEXPERIENCE tools.

The first time you log into the server, DesignSync requests your username and password. The DesignSync GUI presents a dialog box where you must supply your username and password.



The username and password must correspond to the desired login. Options include:

- 3DPassport Central Authentication Service username and password
- DesignSync user profile
- DesignSync-compatible LDAP profile, if the SyncServer is configured to use LDAP (Lightweight Directory Access Protocol),

See *DesignSync Administrator's Guide: What Are User Profiles?* for more details. If you do not know your username or password, contact your project leader.

**Note:** When connecting to servers using DesignSync user profiles and LDAP profiles, you will see the realm "Synchronicity." When connecting to the 3DPassport server, the realm is "3DPassport."

For 3DPassport login, authentication takes place the first time you log in to any server using 3DPassport from any client on your system and persists until it expires as defined by the 3DPassport system administrator.

For other types of logins, the authentication takes place each time you start a new DesignSync session or access a different server. If you select **Save password**, DesignSync stores your password, so in subsequent accesses to the server, you will not have to enter your username and password. Note that the username and password must correspond to your DesignSync user profile for that server. If you do not have a DesignSync user profile, contact your project leader.

### Related Topics

ENOVIA Synchronicity Access Control Guide: User Authentication Access Controls

## Setting Up a Work Area for a Project

If you are part of a project that your project leader has defined as a Public Project, you will want to populate a work area with that project's files. Public Projects are displayed in your Tree View under **Projects =>Public Projects**.

1. In the Tree View, expand the Projects item, then expand Public Projects.

The Tree view lists the defined projects (if any), and the List View displays each project name and its description.

2. Select the project for which you want to create a work area.

There may be a delay as DesignSync verifies that the vault for the selected project is accessible.

3. Launch the workspace wizard, by specifying **Revision Control | Workspace Wizard**. The Workspace Wizard is invoked. When launched by the **File =>Workspace Wizard** command, the Workspace Wizard asks you a series of questions to either create a new project or access an existing project. Launching the Workspace Wizard from the "Setup a work area for this project" command skips the initial questions and brings you directly to the Select Working Area dialog box. You continue from this point responding to the questions that the Workspace Wizard asks you.

### Related Topics

What Is a Project?

Workspace Wizard Overview

## Joining a Project Using a Wizard

### What Is a Project?



A DesignSync **project** is the DesignSync and ProjectSync infrastructure required to support a real-world design project, such as:

- Who are the team members, and what access rights do they have?
- Where is the vault that stores the project files located, and what cache directory is used?
- What design methodologies (such as locking versus non-locking file sharing, or required check-in comments) are enforced?
- What are the project-wide defaults for options such as the default ASCII editor and HTML browser, whether files are checked out writeable or read-only, and the default fetch state?

When fully implemented, all of these attributes would be defined by a project leader and inherited by all team members. Using the SyncAdmin tool, the DesignSync project leader or LAN administrator can do the following:

- Set LAN-wide preferences such as the default editor, default fetch state, recognition of collection objects, and so on. You inherit all of these settings, but can override some of them by using SyncAdmin yourself or through **Tools => Options** within DesignSync.
- Define **Public Projects**, which consist of the project name, the vault where the project files are stored, the project's cache directory, and an optional description.

Public Projects provide two benefits:

- They allow different vaults to use different caches. For example, the ASIC1 design project could use a cache called `asic1_cache`, while the FPGA project could use a cache called `fpga_cache`.
- They simplify how team members interact with a project vault. You can do the following:
  - View the properties (definition) of a Public Project.
  - Browse the vault of a Public Project.
  - Populate a work area with the files of a Public Project.

### Notes:

- The default cache directory used for vaults that have not been defined as part of a Public Project is the cache that was specified during installation (typically `<SYNC_DIR>/../sync_cache`) or by the LAN administrator using SyncAdmin.
- Although caching as a means for sharing DesignSync files is not implemented on Windows platforms (the **share** option to the check-in, check-out, cancel, and populate commands are not available), caches are used by DesignSync for other housekeeping purposes. Therefore, you can define and use Public Projects, including specifying the cache directory, on Windows.

### Related Topics

## DesignSync Data Manager User's Guide

Defining a Public Project

Displaying Project Properties

Browsing a Project Vault

Setting Up a Work Area for a Project

*DesignSync Data Manager Administrator's Guide: The SyncAdmin Tool*

### Defining a Public Project

Your project leader or LAN administrator uses the SyncAdmin tool to define Public Projects. See the SyncAdmin help for more information.

Note: You must restart DesignSync to see new Public Projects or changes to existing Public Projects.

#### Related Topics

What Is a Project?

*DesignSync Data Manager Administrator's Guide: The SyncAdmin Tool*

### Displaying Project Properties

You can display the definitions of projects that appear under **Projects =>Public Projects** in your Tree View as follows:

1. In the Tree View, expand the Projects item, then expand Public Projects.

The Tree view lists the defined projects (if any), and the List View displays each project name and its description.

2. Select the project whose properties you want to view.

There may be a delay as DesignSync verifies that the vault for the selected project is accessible.

3. Click the right mouse button.

A popup menu appears. If DesignSync was unable to access the project's vault, the commands in the popup menu will be grayed out and you cannot continue.

4. Select the Properties command.

The Properties dialog box appears, which displays the name, vault, cache, and description of the project. Note that the fields are not editable. Project leaders can modify a Public Project definition using the SyncAdmin tool.

### Related Topics

[What Is a Project?](#)

[Viewing and Setting Properties](#)

## Browsing a Project Vault

You can browse the vault of a Public Project:

1. In the Tree View, expand the Projects item, then expand Public Projects.

The Tree view lists the defined projects (if any), and the List View displays each project name and its description.

2. Click the project whose vault you want to browse.

There may be a delay as DesignSync attempts to communicate with the vault. Assuming the vault is accessible, a green check mark appears next to the project name and the top-level contents of the vault are displayed in the List View. You can then expand the project in the Tree View or the folders in the List View to navigate farther into the vault.

### Related Topics

[What Is a Project?](#)

## Workspace Wizard

### Workspace Wizard Overview

The **Workspace Wizard** helps you join an existing project, create a new project, work with an existing module, or create a new module.

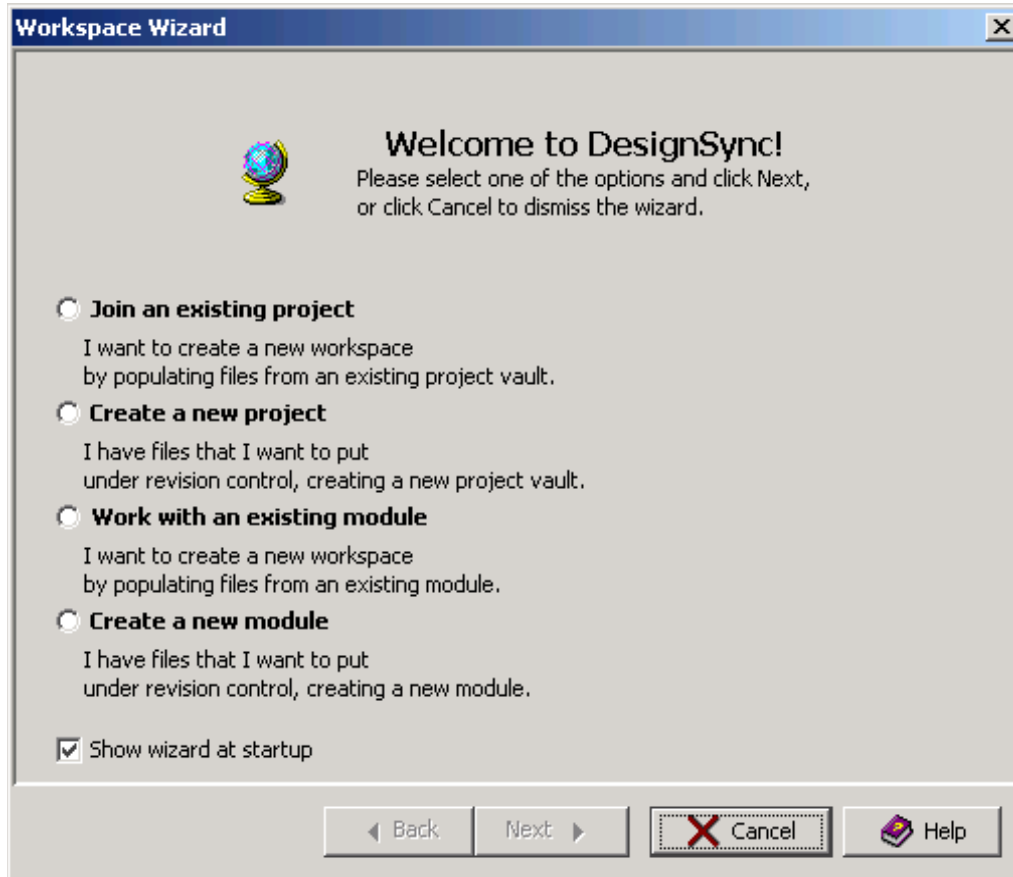
**Note:** The Workspace Wizard has a default fetch state of Unlocked copies.

To help you with a task, the Workspace Wizard guides you through a series of dialogs, gathering information it needs to perform the task; then it performs the task for you.

For example, when creating a new project, the Workspace Wizard creates a new vault, or data repository, and checks in initial versions of your design files so they can be accessed by other team members. When joining an existing project, the Workspace

Wizard associates an existing vault with your work area and populates your work area with the requested versions, typically the latest versions, of files from that vault.

By default, the Workspace Wizard runs when you invoke DesignSync. To disable this feature, de-select the **Show Wizard at startup** check box in the Workspace Wizard dialog box.



You can also invoke the Workspace Wizard from your DesignSync session in any of these three ways:

- Select **Revision Control =>Workspace Wizard** from the DesignSync menu.
- On the server, select a vault folder and right-click on the folder to display a popup menu. From the menu, select **Workspace Wizard**.

When DesignSync displays the **Workspace Wizard** dialog box, you select the option you want and click **Next** to continue to the next dialog box. Each options leads through the Workspace Wizard steps.

### Using the Workspace Wizard

Before you use the Workspace Wizard, you should review the Workspace Wizard Overview topic.

The **Workspace Wizard** dialog box (**Welcome to DesignSync!**) asks you to choose one of the four following options.

**Note:** The Workspace Wizard has a default fetch state of Unlocked copies. The fetch state is set through the SyncAdmin application.

<b>Option and Steps</b>	<b>Choose this option when</b>
<p><b>Join an existing project:</b></p> <ol style="list-style-type: none"> <li>1. Specify the vault of the existing project</li> <li>2. Select workspace for an existing project</li> <li>3. Specify selector or configuration</li> <li>4. Specify object state</li> <li>5. Specify mirror (optional)</li> <li>6. Final confirmation</li> </ol>	<p>The vault structure for a project is already in place and you have just joined the team.</p> <p><b>Note:</b> Create the desired folder structure in your workspace before running the wizard.</p>
<p><b>Create a new project:</b></p> <ol style="list-style-type: none"> <li>1. Specify a Vault for a New Project</li> <li>2. Select Workspace</li> <li>3. Specify Files to Exclude</li> <li>4. Specify the Object State</li> <li>5. Specify a Check-In Comment</li> <li>6. Final Confirmation</li> </ol>	<p>You want to create a vault on the server from files in your workspace.</p> <p>You want to put this design hierarchy, including any files it contains, into a vault accessible by the team members.</p>
<p><b>Work with an existing module:</b></p> <ol style="list-style-type: none"> <li>1. Select a Module</li> <li>2. Select a Workspace for the Module</li> <li>3. Fetch the Module Hierarchy</li> <li>4. Final Confirmation</li> </ol>	<p>A module already exists and you select this option to fetch the module to your workspace.</p> <p>For example, you join a design team that uses modules to contain design blocks. To fetch the module for a design block to your workspace, select this option.</p>
<p><b>Create a new module:</b></p> <ol style="list-style-type: none"> <li>1. Specify Information about the New Module</li> <li>2. Select Workspace Files for the New Module</li> <li>3. Final Confirmation</li> </ol>	<p>You want to create a module on the server from files in your workspace.</p> <p>For example, you are the design team leader and your team is using modules for its design blocks. Your work area already has the directory containing the files for the block. You want to put those</p>

	files under revision control and create a module from them. Then team members can fetch the module to their workspaces and work with it.
--	--

By default, the Workspace Wizard runs when you invoke DesignSync. To disable this feature, de-select the **Show Wizard at startup** check box in the Workspace Wizard dialog box.

**Click on the option in the following illustration to see the dialog box for the next step for that option.**

### Related Topics

Object States

SyncAdmin Help: Default Fetch States

## Joining an existing project

### Specify the Vault of the Existing Project

The **Select Vault** dialog box asks you specify the URL of the vault of the existing project. You can choose between remote and local vaults.

A remote vault is used when more than one person or an entire development team (often from diverse geographical locations) needs to share files. In this way any team member located anywhere can access your vault using the URL of your vault.

A local vault is used when you will not be sharing your design files with other users. DesignSync then uses the default client-vault location as specified when DesignSync was installed. Since you typically want to share files with other users, choosing a local vault is not often a logical choice when joining an existing project.

### To specify the existing project you want to join:

1. In the field of **Select Vault** dialog box, enter the URL of the vault that contains the project files for which you want to populate your work area. You can enter the URL in one of three ways:
  - Type the URL in the text field.
  - Click on the arrow and select from the URLs (if any) that you entered in previous Wizard invocations. DesignSync stores the 10 most recently entered URLs.
  - Click the Browse Servers button to browse the available public projects (as defined by your project leader) and SyncServers (as defined in your site and local SyncServer lists). Click the Browse Local button to select a local vault on your machine.

For example, you might specify:

```
sync://apollo.spaceco.com:2647/Projects/Sportster
```

where **sync:** is the DesignSync protocol for specifying a remote server, **apollo** is the server (machine) name, **2647** is the port number that was specified when the server was configured, **Projects** is the top-level folder on the server under which project vaults are located, and **Sportster** is the top-level vault folder for the Sportster project. If **apollo** is on your LAN, you can simplify the URL to:

```
>sync://apollo:2647/Projects/Sportster
```

2. Click **Next** button to continue to the Select Workspace dialog box.
  - If DesignSync alerts you if the specified vault location does not exist, you need to verify that the URL you entered is correct.
  - You may be asked for account information (username/password) to access the SyncServer.

### Notes:

- The default SyncServer port is 2647. You can omit the port specification when accessing a SyncServer that uses the default port number.
- On Windows machines, there may be a short delay before the next dialog box appears depending on your drive configuration.
- If you are working in a multi-branch environment, you can optionally specify the persistent selector list (see the What Are Persistent Selector Lists topic for details) for the hierarchy by following the vault URL with **@<selectorList>**. This URL syntax is equivalent to executing a **setselector** command following the **setvault** command. For example, you could define the persistent selector list to be "main" as follows:

```
sync://apollo:2647/Projects/Sportster@main
```

- DesignSync also supports a **syncs** protocol for communicating with secure (SSL) SyncServer ports. In most cases, DesignSync automatically redirects requests to a cleartext (non-secure) port using the **sync** protocol to the secure port, if one is defined. The default DesignSync secure port number is 2679. Your DesignSync administrator defines what SyncServer ports are available and whether secure communications are required.

### Select Workspace for an Existing Project

The **Select Workspace** dialog box asks you to choose the work area into which you want the project files to be put.

**To select a workspace for an existing project:**

1. In the Workspace field, enter the path to the existing project. You can click **Browse** to browse to the correct path.
2. Optionally, in the Bookmark field, enter the bookmark name that will display for this project from your DesignSync Bookmarks menu.
3. Click **Next** button to continue to the Specify Selector or Configuration dialog box.

**Note:** The Workspace Wizard will not let you proceed if you select any of the following as the directory you want to populate with project files:

- My Computer
- A drive, such as C: (Windows only)
- Your \$HOME directory (UNIX only)

**Specify Selector or Configuration**

The **Specify selector or configuration** dialog box asks you to specify what versions of design files you want to be put in your work area. You can specify:

- A branch selector, typically Trunk, which populates the Latest versions from the specified branch.
- Any other selector, such as a version tag. However, the selector you specify becomes the persistent selector list for the work area, meaning that the workspace will always be populated with the matching selector list. In the case of a version tag, it means it will always be populated with the same tagged version. If a module is populated with a static version tag, changes in the workspace cannot be checked in.
- A selector list, in a files-based workspace populates a single selector; the first valid selector on the list.
- A selector list, in a modules-based workspace uses the last selector as the base selector for the workspace and overlays all other selectors in the selector list as described in Module Member Tags.
- A project configuration, as created using ProjectSync, that has been applied to the project vault you are accessing. See the ProjectSync documentation for additional information about project configurations.

For information on selector formats and how to specify them, see Selector Formats.

**Note:** When working with a module selector list, creating a blended environment containing a main module and one or more Module Snapshots, the selector is applied recursively to the entire hierarchy, including submodules with the following exceptions:

- External modules
- Legacy modules
- Modules defined with a "static" hierarchical reference



- Referenced file vaults.

See Module Member Tags for more information.

**To select a workspace for an existing project:**

1. Click the arrow to the right of the text field to display a drop-down list of choices. The list contains:
  - **Trunk**, which is the default branch tag for branch 1.
  - The persistent selector for the work area you specified if it is different from the default of Trunk.
  - The selector, if any, that you specified with the **@<selectorList>** syntax when you specified the URL of the project vault.
2. Click **Next** button to continue to Specify the Object State Dialog box.

**Specify the Object State of Populated Items**

The **Specify Object State** dialog box asks you what should be placed in your work area as a result of the check out. The options are:

Option	Result
References to versions	Leaves DesignSync references to the files in your work area.
Unlocked copies	Leaves copies ( replicas) of the files in your work area, not locked. This is the default selection unless your project leader has defined a default fetch state.
Locked copies:	Leave files in your work area, locked. Other users can check the files out from the vault but cannot check in new versions as long as you have the files locked.
Links to caches	Links to a shared copy of the design object in a cache directory. This option is available only on UNIX platforms.
Links to mirror:	Links to a shared copy of the design object in a mirror directory. This option is available only on UNIX platforms.

**To select a specify the object state of populated items:**

1. Select one of the object state options.
2. If you chose the **Links to mirror** option, when you click **Next**, the Workspace Wizard takes you to the Specify Mirror Dialog box.

If you chose any other option, when you click **Next**, the Workspace Wizard takes you to Finished dialog box. This is where you can confirm or cancel the commands you have entered the previous dialog boxes.

### Specify Mirror

The **Specify Mirror** dialog box asks you to specify the mirror directory to associate with this project.

### Notes:

- Mirror capability is related to symbolic links which exist only on UNIX systems.
- A mirror directory must have already been set up for the project. See ENOVIA Synchronicity Command Reference: setmirror for more information.

### To specify a mirror:

1. Specify the mirror directory. Enter the directory in the field or You can click Browse to select the directory. If you have not associated a mirror directory with your work area, an alert box warns you that you cannot proceed.
2. Click **Next**, the Workspace Wizard takes you to Finished dialog box. This is where you can confirm or cancel the commands you have entered the previous dialog boxes.

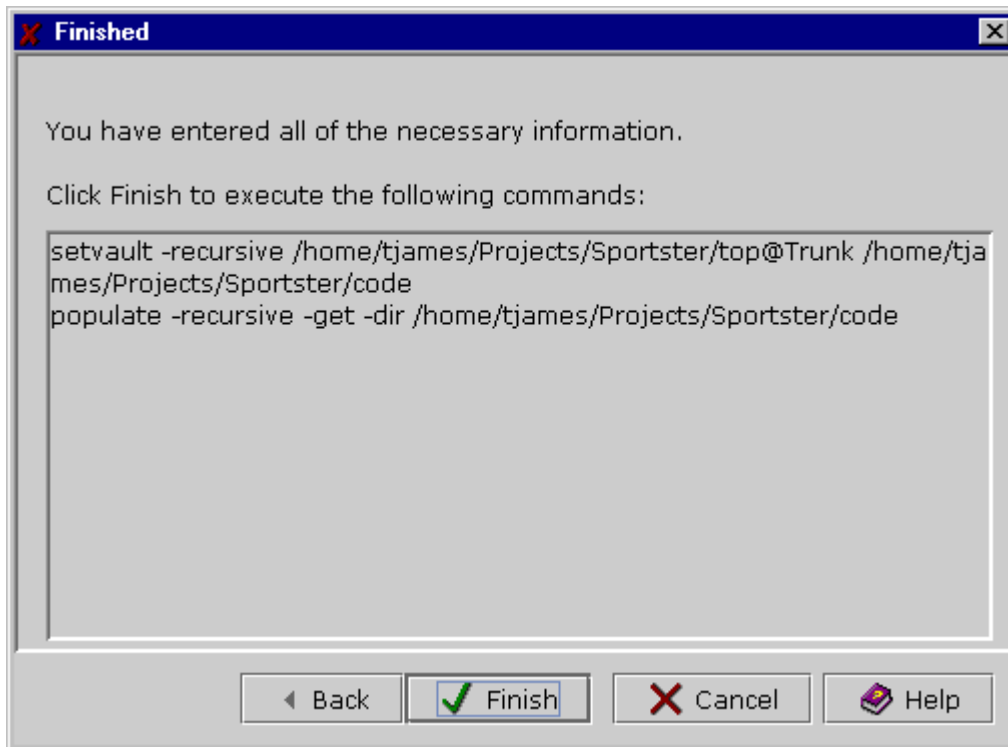
### Workspace Wizard Confirmation

After gathering information the previous dialog boxes, the Workspace Wizard displays the **Finished** dialog box. The **Finished** dialog displays the commands DesignSync uses to complete the operation. At this point, you must confirm or cancel the commands you have entered the previous dialog boxes.

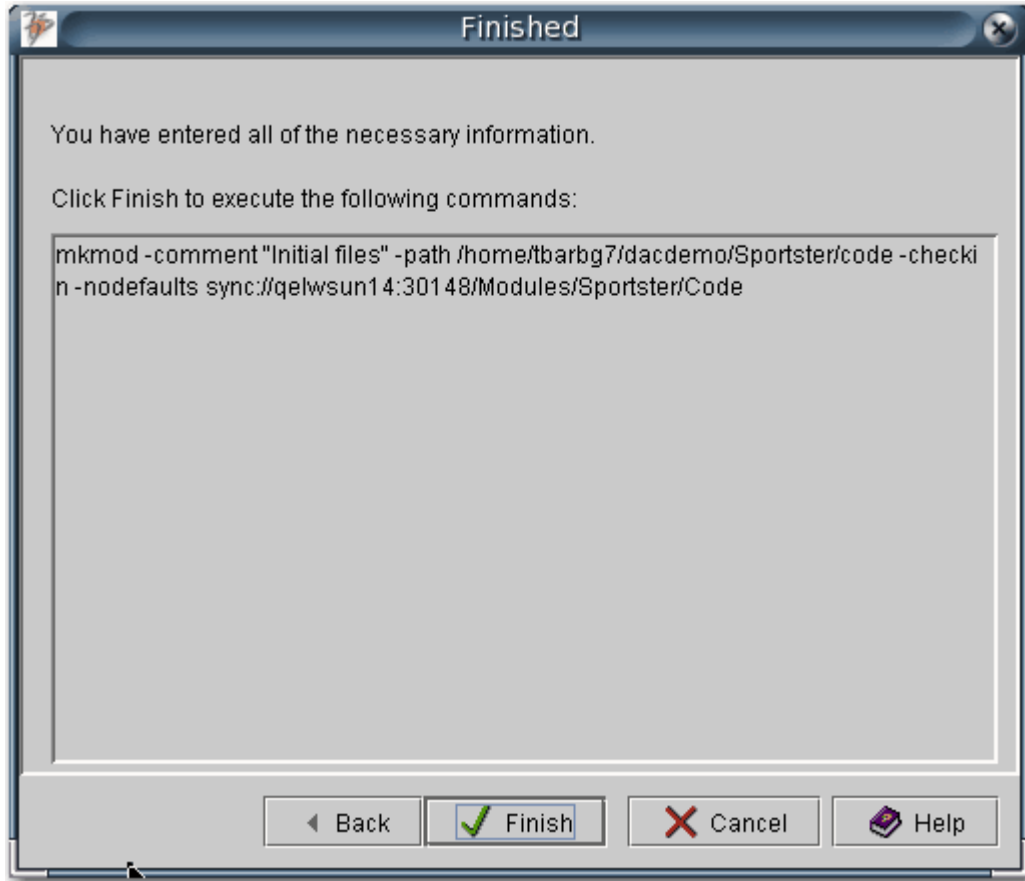
- Click **Finish** to have DesignSync perform the commands listed in the dialog box.
- Click **Cancel** to cancel these commands.

### Examples of commands you could see in the Finish dialog box:

If you selected **Join an existing project** from the **Workspace Wizard** dialog, the **Finished** dialog would display commands similar to these:



If you selected **Create a New Module** from the **Workspace Wizard** dialog, the **Finished** dialog would display commands similar to these:



## Related Topics

[Workspace Wizard Overview](#)

[Workspace Wizard Paths](#)

## Creating a new project

### Specify a Vault for a New Project

The **Select Vault** dialog box asks you specify the URL of the vault of the new project. You can choose between remote and local vaults:

A remote vault is used when more than one person or an entire development team (often from diverse geographical locations) needs to share files. In this way any team member located anywhere can access your vault using the URL of your vault.

A local vault is used when you will not be sharing your design files with other users. DesignSync then uses the default client-vault location as specified when DesignSync was installed. Since you typically want to share files with other users, choosing a local vault is not often a logical choice when creating a new project.

When you specify a vault for a workspace folder, if a workspace root has not already been set on a folder above the folder location where the vault is being declared, then DesignSync will automatically designate a workspace root folder according to the workspace root definition rules. By default this is one level above hierarchically above the specified folder. The workspace root stores metadata information about the design objects contained within the folder.

### To specify the new project you want to create:

1. In the field of **Select Vault** dialog box, enter the URL of the vault that contains the project files for which you want to populate your work area. You can enter the URL in one of three ways:
  - Type the URL in the text field.
  - Click on the arrow and select from the URLs (if any) that you entered in previous Wizard invocations. DesignSync stores the 10 most recently entered URLs.
  - Click the Browse Servers button to browse the available public projects (as defined by your project leader) and SyncServers (as defined in your site and local SyncServer lists). Click the Browse Local button to select a local vault on your machine.

For example, you might specify:

```
sync://apollo.spaceco.com:2647/Projects/Sportster
```

where **sync:** is the DesignSync protocol for specifying a remote server, **apollo** is the server (machine) name, **2647** is the port number that was specified when the server was configured, **Projects** is the top-level folder on the server under which project vaults are located, and **Sportster** is the top-level vault folder for the Sportster project. If **apollo** is on your LAN, you can simplify the URL to:

```
>sync://apollo:2647/Projects/Sportster
```

2. Click **Next** button to continue to the Select Workspace dialog box. Since the location you specified does not exist yet, an alert box tells you that DesignSync will create the vault for you.
  - If the location you specify does exist, you must confirm that you want to check in new files into an existing vault.
  - You may be asked for account information (username/password) to access the SyncServer.

### Select Workspace for a New Project

The **Select Workspace** dialog box asks you to choose the work area into which you want the project files to be put.

Your work area refers to the directory (folder) that represents your project -- that is, the directory that contains the files you wish to check in so that they will be under revision control.

### To select a workspace for an new project:

1. In the Workspace field, enter the path to the new project. You can click **Browse** to browse to the directory that represents your work area.
2. Optionally, in the Bookmark field, enter the bookmark name that will display for this project from your DesignSync Bookmarks menu.
3. Click **Next** button to continue to the Specify Exclude Files dialog box.

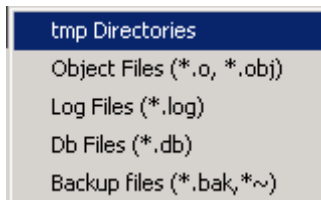
**Note:** The Workspace Wizard will not let you proceed if you select any of the following as the directory you want to populate with project files:

- My Computer
- A drive, such as C: (Windows only)
- Your \$HOME directory (UNIX only)

### Specify Files to Exclude

The **Specify Exclude Files** dialog box asks you to specify what files you want to exclude from being checked into the vault. You can choose items from the display of drop-down list of choices or add your own wildcards.

The drop-down list contains the categories of files that are often excluded:



These are the wildcards you can use in the **Exclude List** text box:

Wildcard	Representing
?	A single character
*	A string
A[1-9]	A1 through A9

### To create a list of excluded from check in files types:

1. Select from the drop-down list each file type you want excluded from check-in. As you choose, the system adds commas between the items.

2. If desired, add your own wildcard exclusions. When you choose to create your own wildcard file exclusions from the list above, remember that you will have to add your own commas to separate items on the file exclusions list.
3. Click **Next** button to continue to Specify the Object State Dialog box.

**Specify the Object State of Items Checked In**

The **Specify Object State** dialog box asks you what should be left behind in your work area when the check in operation completes. The options are:

Option	Result
References to versions	Leaves DesignSync references to the files in your work area.
Unlocked copies	<p>Checks file into the vault and a leaves a replica in your work area.</p> <p>You can use the file as if you had not checked it in. However, depending on your work methodology, you may want to keep a locked copy if you plan on editing the file.</p> <p>This is the default selection unless your project leader has defined a default fetch state.</p>
Locked copies	<p>Checks in file and immediately checks the file back out into your work area with a lock.</p> <p>Others can check the file out from the vault but cannot check in a new version as long as you have the lock on the files.</p>
Links to caches	Links to a shared copy of the design object in a cache directory. This option is available only on UNIX platforms.
Links to mirror	Links to a shared copy of the design object in a mirror directory. This option is available only on UNIX platforms.

**To select a specify the object state of checked in items:**

1. Select one of the object state options.
2. Click **Next** to continue to the Specify a Check-In Comment dialog box.

**Specify a Check-In Comment**

The **Specify Check-In Comment** dialog box asks you to specify a comment for the project you are creating. This comment serves as the initial check-in comment or all your files when you first place your project files under revision control.

**To specify a check in comment for the initial check in of a new project:**

1. Click in the comment field.

You can also use your default editor to prepare your comments. Right click in the client field and select **Use Editor** from the context menu. When you save and exit from your editor, your comments will be put in the message box. Quitting from your editor without saving leaves the message box empty.

2. Create your comment for this project. Note that projects can be defined with a minimum comment-length requirement and you will need to create a comment is long enough to meet this requirement.
3. Click **Next** button to continue to Finished dialog box. This is where you can confirm or cancel the commands you have entered the previous dialog boxes.

### **Workspace Wizard Confirmation**

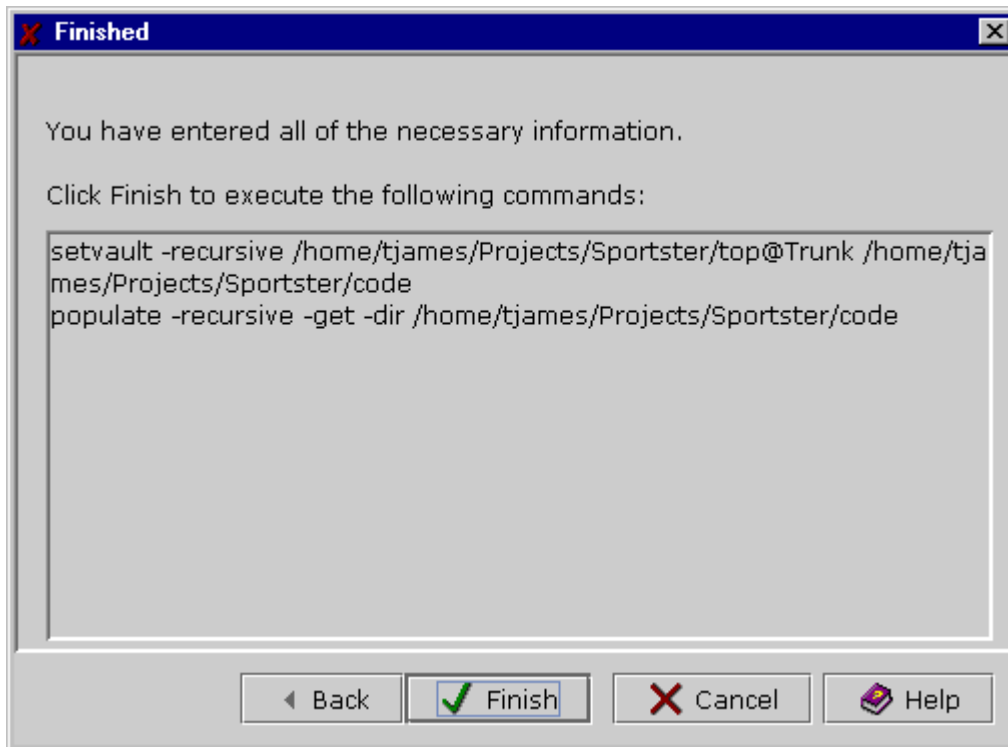
After gathering information the previous dialog boxes, the Workspace Wizard displays the **Finished** dialog box. The **Finished** dialog displays the commands DesignSync uses to complete the operation. At this point, you must confirm or cancel the commands you have entered the previous dialog boxes.

- Click **Finish** to have DesignSync perform the commands listed in the dialog box.
- Click **Cancel** to cancel these commands.

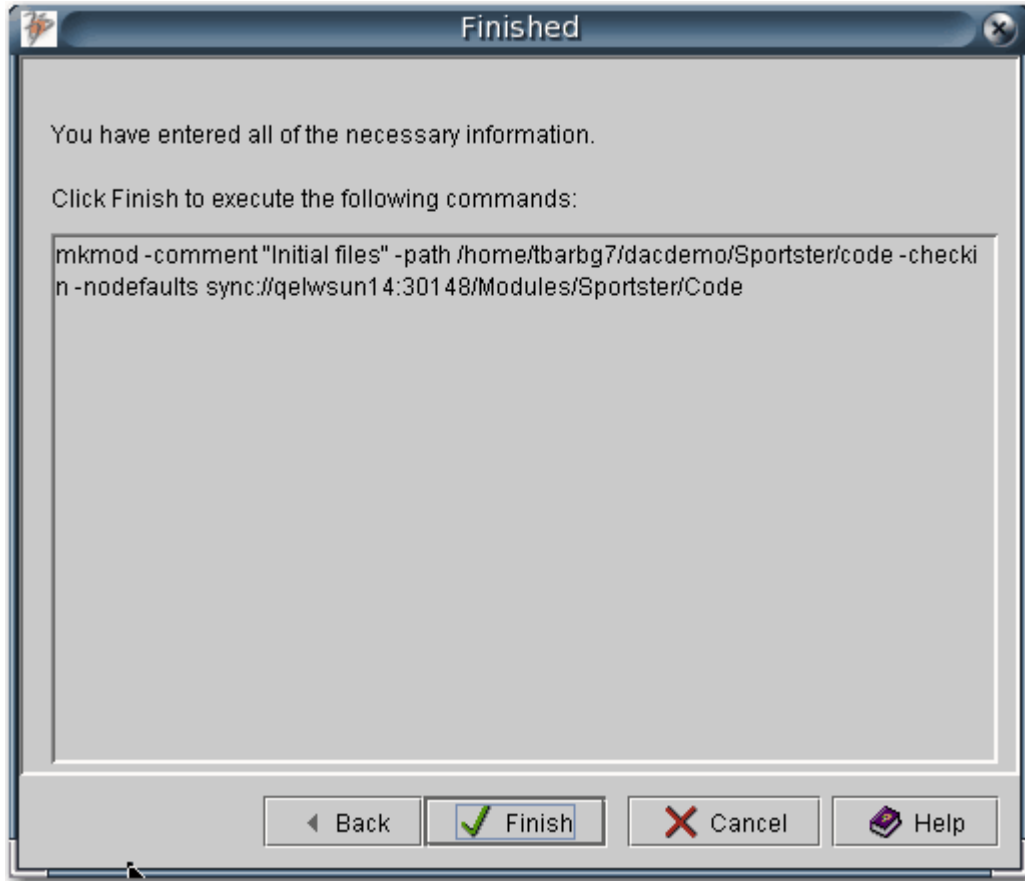
### **Examples of commands you could see in the Finish dialog box:**

If you selected **Join an existing project** from the **Workspace Wizard** dialog, the **Finished** dialog would display commands similar to these:





If you selected **Create a New Module** from the **Workspace Wizard** dialog, the **Finished** dialog would display commands similar to these:



## Related Topics

[Workspace Wizard Overview](#)

[Workspace Wizard Paths](#)

## Working with an existing module

### Select a Module

The **Select Module** dialog box asks you to specify the vault URL and the configuration/selector for the you want to fetch to your work area.

### To select the existing module:

1. In the **Specify the URL of the module's vault field**, enter the URL of the vault that contains the existing module. You can enter the URL in one of three ways:
  - Type the URL in the text field.
  - Click on the arrow and select from the URLs (if any) that you entered in previous Wizard invocations. DesignSync stores the 10 most recently entered URLs.

- Click the Browse Servers button to browse the available public projects (as defined by your project leader) and SyncServers (as defined in your site and local SyncServer lists). Click the Browse Local button to select a local vault on your machine.
2. If you selected a legacy module in step 1, then second field that is displayed is named **Configuration**.

If you selected a current module in step 1, then the second field that is displayed is named **Selector**.

You can type name of the selector for the module or the configuration for a legacy module. You can also accept the pre-filled default.

3. If your project manager has defined module views for your use, you can select a module view to set as the persistent view for your workspace. Use the down-arrow on the field to get a list of available views. You can select multiple views by separating the view names with a comma.
4. Click **Next** button to continue to the Select a Workspace for the Module dialog box.

### Select a Workspace for the Module

The **Select Workspace** dialog box asks you to choose the workspace where you want the module to reside.

#### To select a workspace for an existing module:

1. In the Workspace field, enter the path to the existing project. You can click **Browse** to browse to the correct path.
2. Optionally, in the Bookmark field, enter the bookmark name that will display for this module from your DesignSync Bookmarks menu.
3. Click **Next** button to continue to the Fetch the Module Hierarchy dialog box.

### Fetch the Module Hierarchy

The **Recurse** dialog box asks you to specify whether DesignSync should fetch just the selected module (the default) or fetch the module and each submodule in its hierarchy.

#### To select a workspace for an existing module:

1. If you want to fetch the files of the entire module hierarchy, check the **Get the module recursively** checkbox. By default, the checkbox is unchecked, indicating that only the files of the selected module are fetched.
2. Click **Next** button to continue to Finished dialog box. This is where you can confirm or cancel the commands you have entered the previous dialog boxes.

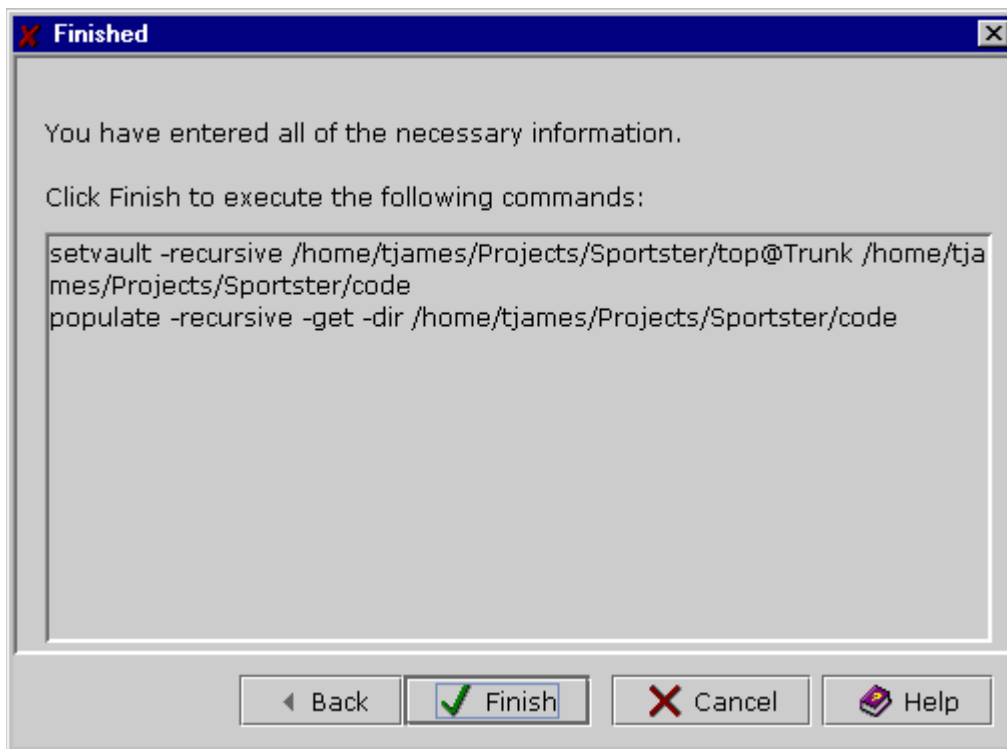
### Workspace Wizard Confirmation

After gathering information the previous dialog boxes, the Workspace Wizard displays the **Finished** dialog box. The **Finished** dialog displays the commands DesignSync uses to complete the operation. At this point, you must confirm or cancel the commands you have entered the previous dialog boxes.

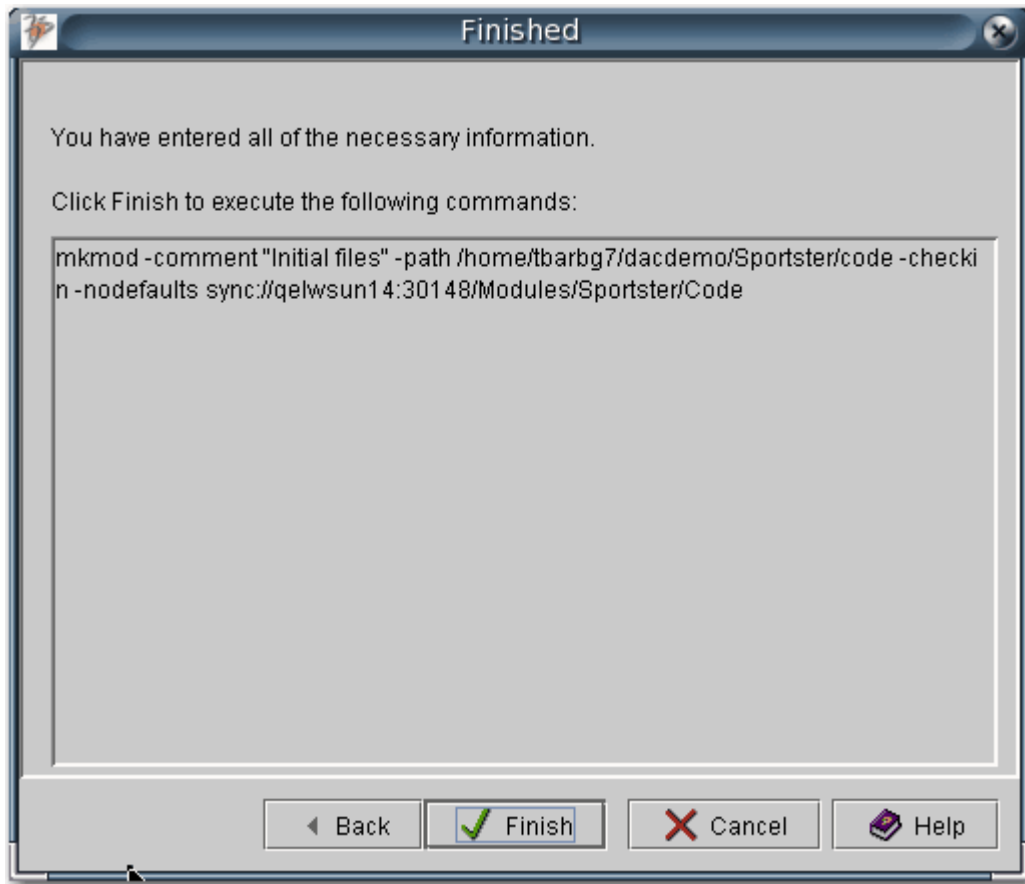
- Click **Finish** to have DesignSync perform the commands listed in the dialog box.
- Click **Cancel** to cancel these commands.

### Examples of commands you could see in the Finish dialog box:

If you selected **Join an existing project** from the **Workspace Wizard** dialog, the **Finished** dialog would display commands similar to these:



If you selected **Create a New Module** from the **Workspace Wizard** dialog, the **Finished** dialog would display commands similar to these:



## Related Topics

[Workspace Wizard Overview](#)

[Workspace Wizard Paths](#)

## Creating a new module

### Specify Information about the New Module

The **New Module** dialog box asks you to provide information about the new you want to create.

### To specify the new module you want to create:

1. In the **Server URL** field, enter the URL of the SyncServer where the module will reside. You only need to specify the server name and the port. Use this format:

```
sync://<host>:<port>
```

2. In the **Module name** field, enter the name for the module you want to create. A module name can not have these characters as part of its name:

~ ! @ # \$ % ^ & \* ( ) , ; : | ` ' " = [ ] /

3. In the **Comment** field, enter the first branch comment.

4. Click **Next** button to continue to the Select Workspace Files for the New Module.

### Select Workspace Files for the New Module

The **Select Workspace** dialog box asks you to select files that you want to include in the new module.

#### To select a workspace for an new module:

1. In the Workspace field, enter the path to the new module. You can click **Browse** to select the workspace directory containing the files you want to include in the module.
2. Optionally, in the Bookmark field, enter the bookmark name that will display for this module from your DesignSync Bookmarks menu.
3. Optionally, you can choose to include empty directories in the initial creation. This allows you to create a framework for the module, even if the data for the empty directories doesn't exist yet.

**Note:** Any directories explicitly added to the module must be explicitly removed if no longer needed. Any directories not explicitly added to the module are automatically removed from the module when they become empty.

4. Click **Next** button to continue to the Finished dialog box. This is where you can confirm or cancel the commands you have entered the previous dialog boxes.

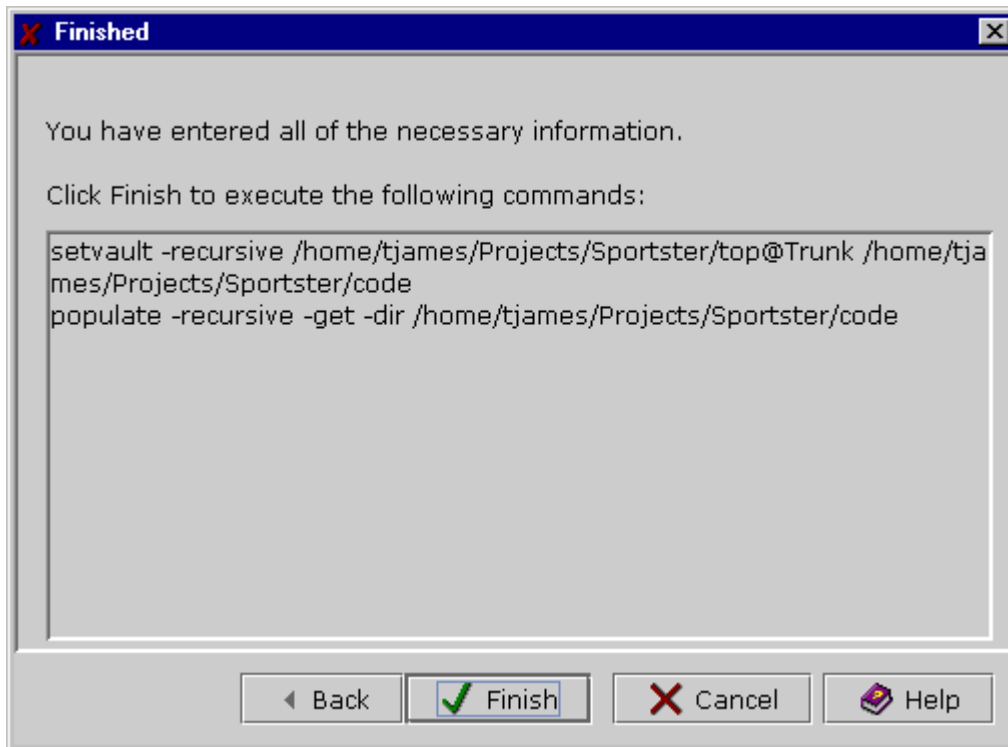
### Workspace Wizard Confirmation

After gathering information the previous dialog boxes, the Workspace Wizard displays the **Finished** dialog box. The **Finished** dialog displays the commands DesignSync uses to complete the operation. At this point, you must confirm or cancel the commands you have entered the previous dialog boxes.

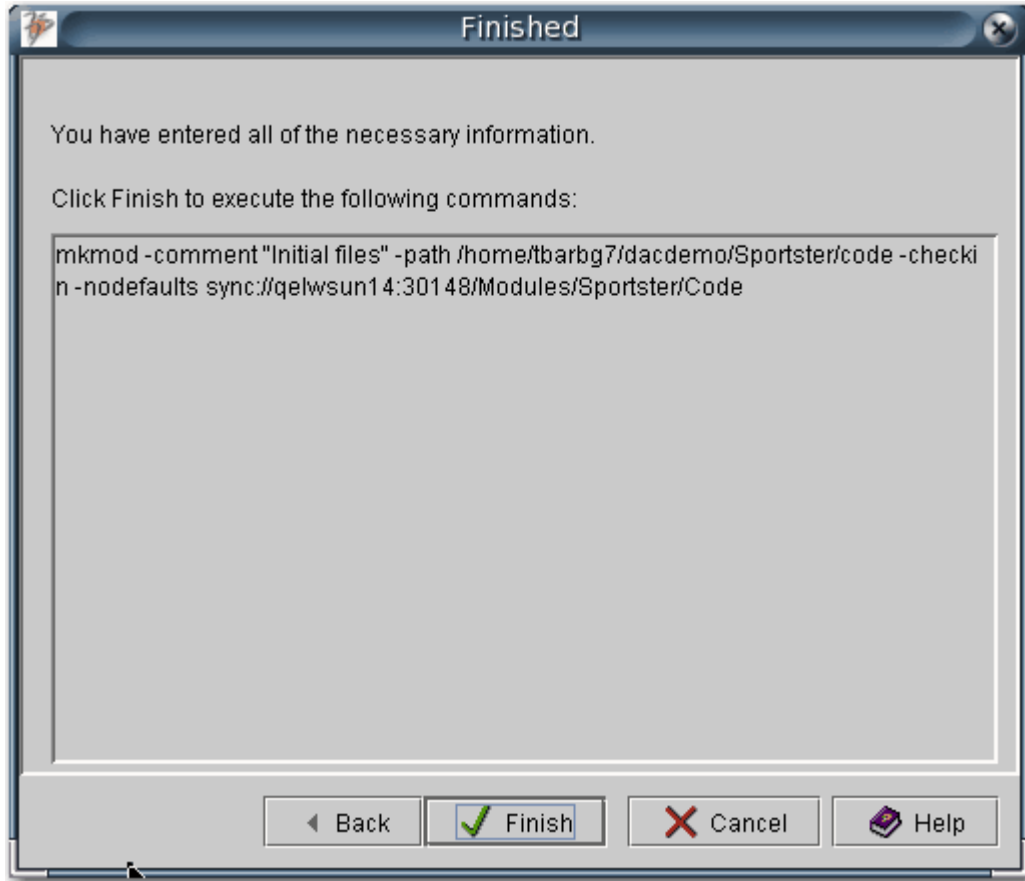
- Click **Finish** to have DesignSync perform the commands listed in the dialog box.
- Click **Cancel** to cancel these commands.

### Examples of commands you could see in the Finish dialog box:

If you selected **Join an existing project** from the **Workspace Wizard** dialog, the **Finished** dialog would display commands similar to these:



If you selected **Create a New Module** from the **Workspace Wizard** dialog, the **Finished** dialog would display commands similar to these:



## Related Topics

[Workspace Wizard Overview](#)

[Workspace Wizard Paths](#)

## Joining a Project Step-by-Step

### Specifying the Vault Location for a Design Hierarchy

Before you can check in files for the first time or check out files from an existing vault, you must associate a vault location with your local work area. Typically, an entire hierarchy of files and folders that represent a design project would be revision-controlled under the same top-level vault.

When you specify a vault for a workspace folder, if a workspace root has not already been set on a folder above the folder location where the vault is being declared, then DesignSync automatically designates a workspace root folder according to the workspace root definition rules. By default this is one level above hierarchically above



the specified folder. The workspace root stores metadata information about the design objects contained within the folder.

The Set Vault Association dialog box associates a vault location with a work area. This command maps a local folder (directory) to a revision-control vault folder (repository).

### **When working with modules and module objects**

The only time you would need to use the Set Vault Association dialog box on a module workspace is if you have moved a module, such as moving it to a different or server location. These module objects cannot have a vault association:

- Any subdirectory of a module
- A module root directory
- A module member object
- A module that is in a workspace as a result of a hierarchical reference from another module (such as a sub-module).

### **Client Vaults and Server Vaults**

Setting the vault association is the first step in placing design data under revision control or checking out (populating) data that is already managed. Every local folder and file has a default client vault even if you have not explicitly set the vault.

#### **Client vaults:**

- Reside in the location determined during installation of your client.
- Cannot be accessed by other users. Should only used to manage private data.
- Are always identified using a file URL.
- Do not require a workspace root folder.

You must explicitly set the vault for a folder before you can check in objects contained in the folder. Typically, you use server (remote) vaults, which are managed by SyncServers, instead of client vaults.

#### **Server vaults:**

- Can reside on your local host, but often reside on another host.
- Can be accessed by any user who is authorized to do so.
- Are always identified using a sync: URL.
- Use workspace root folders to store metadata to reduce the amount of information passed to and from the server during operations and allow for intelligent data processing.

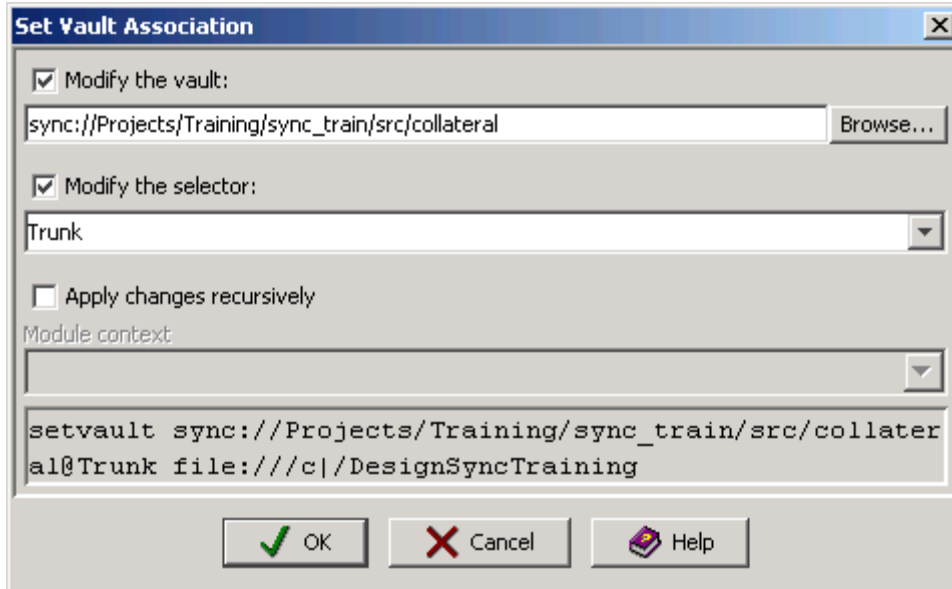
#### **Notes**

- When trying to resolve a selector, DesignSync does not search above the root of a workspace where a set vault association has been applied. Therefore, when a folder that has no selector or persistent selector set, DesignSync searches up the hierarchy stopping at the first folder that has a vault association.
- When resolving multiple selectors specified for a module, DesignSync overlays the module snapshots in order of appearance within the selector, so the first snapshot on the list has precedence over any module members with the same natural path or uuid previously specified.
- Vault settings on subfolders in a work area are affected by the recursive option when they are populated. Consider using REFERENCES in sync\_project.txt files to redirect a subfolder to a different vault. See Vault REFERENCES for Design Reuse for more information.
- When you specify a new vault in the Modify Vault field, the vault folder is not created until you check in design data.
- The local folder on which you are setting the vault must already exist.
- You must have write permission for the parent folder of the folder for which you are setting the vault.
- You need write permission in order for DesignSync to create local metadata stored in .SYNC directories for the parent folder.
- To facilitate the interaction between DesignSync and ProjectSync, it is recommended that you specify Projects as the top-level folder.
- Your DesignSync administrator defines what SyncServer ports are available and whether secure communications are required. See *DesignSync Data Manager Administrator's Guide: Using Secure Communications* for more information.
- DesignSync also supports a syncs protocol for communicating with secure (SSL) SyncServer ports. In most cases, DesignSync automatically redirects requests to a cleartext (non-secure) port using the sync protocol to the secure port, if one is defined. The default SyncServer ports are 2679 for the secure port or 2647 for the cleartext port.
- You can omit the port specification when accessing a SyncServer that uses the default port number.

### To associate a vault with your design hierarchy:

1. Highlight the top-level folder of the hierarchy that you want to place under revision control. To modify the vault location for a module, highlight the module's base directory.
2. From the main menu, select **Revision Control =>Set Vault Association**. You can also right-click and select **Set Vault Association** from the context menu.
3. Enter information or select options as needed from the Set Vault Association dialog box. See Field and Option Descriptions below for more details. If you will not be sharing your design data with other users, you can accept the default local vaults instead of specifying a remote vault.
4. Click **OK** to confirm.

**Click on the fields in the following illustration for information on each field.**



### Field and Option Descriptions

#### Modify the vault

**Note:** When the number of characters in the path to the vault exceeds 1024, the command will fail.

When checked, you can enter the location of new vault for the top-level Module or DesignSync object or folder. For a client vault, the path is the full, absolute path on your local machine. For a server vault, the path is relative to the server root as specified during the SyncServer installation.

By default, if there is no workspace root set, the folder above the specified folder is set as the workspace root. For more information on defining workspace roots, see *DesignSync Data Manager Administrator's Guide: Workspaces*

Click **Browse** to find known projects and servers. Typically, you will specify a remote vault location.

A module path should be specified in the following form:

```
<protocol>://<host>:<port>/[Modules|Projects]/ <path>
```

An example of a location of a new vault location is:

```
sync://apollo.spaceco.com:2647/Projects/Sportster
```

In this example:

Path item	Description
sync:	DesignSync protocol for specifying a remote server. For a standard connection use sync as the protocol. For an SSL connection, use syncs as the protocol.
apollo:	Server (machine) name. Specify a full domain name, such as myhost.myco.com. You can specify just the machine name, which in this case is apollo) if you are on the same LAN as the SyncServer host machine, for example:  <code>sync://apollo:2647/Projects/Sportster</code>
2647	Port number that was specified when the server was configured. You can omit the port specification if the SyncServer is using the default port of 2647.
Projects	Top-level folder on the server under which project vaults are located
Sportster	Top-level vault folder for the Sportster project

By default, this option is unchecked.

**Note:** All Modules must be in the top-level folders in order to be recognized by DesignSync. It is recommended legacy modules use the top-level folder Projects.

### Modify the selector

When checked, you can specify the persistent selector list for the hierarchy by following the vault URL with `;<selectorList>`. For example, to define the persistent selector list as main you would enter the path something like this:

```
sync://apollo:2647/Projects/Sportster;main
```

For non module object, you also can select from a list of available selectors, by highlighting **Get selectors** from the pull down. The system retrieves the available selectors for the project entered in the Modify the vault field.

For module objects, you can select either **Get Branches** or **Get Tags/Versions**.

When unchecked, no selector is set and the default is set to Trunk. If you need to remove a selector, deselect the Modify the vault option, select the Modify the selector option, and then delete the selector information.

### Apply changes recursively

When checked, recursively clears any vault information that may have been set in subfolders. When you are setting the vault for the first time, checking this option was no effect. Check this option if you are changing the vault location for a hierarchy.

When not checked, any previously set vault information in subfolders remains.

When you set the vault on a folder, that vault association is stored in the local metadata for that folder. Each subfolder inherits its vault association from its parent folder; the vault association is not stored in metadata unless the subfolder has an explicit set vault association applied to it.

Every versionable object (file or collection) in the hierarchy inherits its vault from the parent folder, although once a revision-control operation has been performed on the object, the vault association is stored in that object's metadata. Therefore, anytime you want to change a vault setting (as opposed to setting the vault for the first time), check the apply change recursively option. This removes all vault associations that are stored in metadata, which causes all objects in the hierarchy to inherit their vault associations from the folder on which the set vault association has been applied.

**Note:** The -recursive option is not valid for modules and is ignored.

**Caution:** If a subfolder had an explicit set vault association applied to it such that the vault information is stored in the folder's local metadata, that vault association is removed and the subfolder reverts to inheriting its vault association from the parent folder.

### Module context

The field is only enabled for module base folders and contains the module instance names of the modules based at that folder. These module instance names are listed alphabetically in the pull-down.

### Related Topics

Changing the Vault for a Design Hierarchy

*ENOVIA Synchronicity Command Reference:* setvault command

*ENOVIA Synchronicity Command Reference:* setselector

Using Vault REFERENCES for Design Reuse

*ENOVIA Synchronicity Command Reference:* populate

*ENOVIA Synchronicity Command Reference:* unlock

*ENOVIA Synchronicity Command Reference:* url vault

*ENOVIA Synchronicity Command Reference:* setselector

## Adding a Vault to Bookmarks

In the **List View** (right pane) of the DesignSync window, right-click an object. Then select **Go =>Go to Vault** from the menu.

Click to select the vault and select **Bookmark =>Add Bookmark** from the menu, or right-click on the vault and select **Add Bookmark**.

## Verifying That a Vault Has Been Set on a Folder

To verify that a vault has been set on a folder in the working directory, do either of the following:

- From DesignSync, select the folder of interest, then select **File =>Properties**. The **Properties** dialog box contains the vault information.
- From the command line, use the **url vault** command.

### Related Topics

Revision Control Properties

ENOVIA Synchronicity Command Reference: url vault

## Browse the Vault for a File or Project

To browse the vault for a file or project, enter a search path in the Location bar of the DesignSync window followed by a carriage return. The objects searched for will be displayed in the Tree View (left pane) of the DesignSync window.

The search path does not need to be complete. You can enter only the name of a server and port. For example, if you enter the following:

```
sync://myserver.myco.com:2647
```

You can then expand the search from that point.

## Viewing the Contents of a Vault

You can view the vault folder for your project by selecting the top-level local folder and clicking the **Go to Vault** button. You can navigate the vault hierarchy as you navigate other items in the **Tree View** to display the folders and vaults that make up the project.

You can also go directly to the vault of specific files or vault folders of specific folders. Select the local file or folder of interest and do one of the following:

- Click the **Go to Vault** button.
- Select **Go =>Go to Vault**.
- Right-click the folder or file and select **Go to Vault** from the pop-up menu.

### Notes:

- The folder or file must be part of a project that is under revision control. Attempting to view the vault of a file or folder that is not under revision control results in an error.
- A folder will not have a display in the Branch column if the folder belongs to a 5.0 module.



When viewing the vault of a file, you can see the versions of a file, as well as any branches that may emanate from a version. A version that has branches off it is called a branch-point version. Note that every managed object has at least one branch: branch 1, or Trunk. When you view the vault, you are viewing branch 1.

The illustration shows the DesignSync window after viewing the top-level vault folder for the project and then navigating down to **top.v;**, the vault for the **top.v** file. Note that vault names end in a semicolon ( ; ). The **List View** shows that there are currently three versions of **top.v** on the main (Trunk) branch in the vault.




You can double click on a version to display it in your editor. DesignSync fetches a copy of the version into the local DesignSync cache directory. You may be able to edit the cached version of the file, but your changes do not affect the version in the vault.

To see the check-in comments for a version, right-click on the version and select **Properties => Version**. The Revision Log window appears and displays the check-in comment. Use the **vhistory** command to see the entire version history of an object, including all version logs.

Version 1.2 is branch-point version. Double-clicking on the branch point version displays the branches emanating from version 1.2. The following illustration shows that two branches (1.2.1, tagged DevBM, and 1.2.2, tagged Rel2.1) emanate from version 1.2

sync://localhost/Projects/Sportster/top/top.v;1.2									
Name	Type	Modified	Size	Status	Locker	Version	Branch	Ver	
 top.v;1.2.1	Branch	06/08/2000 09:43			-		DevBM		
 top.v;1.2.2	Branch	06/08/2000 09:43			-		Rel2.1		

And double-clicking the **1.2.1** branch displays the versions on that branch:

sync://localhost/Projects/Sportster/top/top.v;1.2.1								
Name	Type	Modified	Size	Status	Locker	Version	Branch	Version
 top.v;1.2.1.1	Version	06/08/2000 10:54		-		1.2.1.1	DevBM	
 top.v;1.2.1.2	Version	06/08/2000 10:54		-		1.2.1.2		
 top.v;1.2.1.3	Version	06/08/2000 10:54		-		1.2.1.3		Latest

## Related Topics

Vaults, Versions, and Branches

Displaying Version History

ENOVIA Synchronicity Command Reference: vhistory

## Populating Your Work Area

If you have just joined a project, you need to use **Populate** to check out all the objects from that project into your local working folder. You typically want to populate recursively, (using the **Recursive** option) which traverses a vault folder and recreates the vault folder hierarchy in your work area or traverses a module hierarchy and recreates the hierarchy in your work area. You can also use the Workspace Wizard to help you join a project.

The default mode for populate fetches **Unlocked copies** of objects, unless you have saved a different object state setting with the **Save Settings** button, or your project leader has defined a default fetch state. See Saving the Setting of an Object's State and SyncAdmin Help: Default Fetch State for information on these two conditions.

After you have initially populated your work area, you need to populate periodically to update your local files, because the vault might have changed since your previous populate. For example, another user might have checked in new design objects or new versions of objects that are already in your work area. Generally, you can perform an incremental populate for these periodic updates. If you need to change the states of files in your work area (for example, if you are changing from locked to unlocked files, or unlocked files to links to the cache), you should perform a full populate. An incremental populate updates only those local folders whose corresponding vault folders have changed, which is faster than a full populate operation. DesignSync performs an incremental populate by default whenever possible, although it automatically reverts to a full populate when necessary. See the ENOVIA Synchronicity Command Reference:populate command description for the circumstances where DesignSync automatically performs a full populate and for the circumstances where you might choose to specify a full populate.

You can select these types of data for the populate operation:



- A DesignSync folder
- A managed DesignSync (non-module) object
- A module or external module that is already in your workspace (for an update of your workspace)
- A server module branch or server module version (for an initial populate of a workspace)
- An href that is already in your workspace

**Note:** You cannot select an href on the server; to select an href, you must populate the entire module from the server, or filter the data that is fetched.

Specifying an href for a populate is equivalent to specifying the relevant submodule directly. By specifying an href, the submodule (and the version to be fetched) is identified via the parent module. Specifying an href enables you to fetch a submodule of a module, without necessarily knowing where that module is sourced from, or what version is appropriate.

You can only specify the hrefs directly in the module being addressed, not in its submodules. Selecting an href in your workspace as the object to populate sets the module context accordingly, disabling the Module context field.

- A module folder that is already in your workspace

**Note:** You cannot select a module folder on the server; to select a module folder, you must populate the entire module from the server, or filter the data that is fetched.

- A module member that is already in your workspace

**Note:** You cannot select a module member on the server; to select a module member, you must populate the entire module from the server, or filter the data that is fetched.

- A legacy module already in your workspace
- A legacy module configuration on the server
- A server module branch and one or more server module snapshots. For information on creating a blending environment including a main server module with one or more module snapshots, see Module Snapshots.

### Additional Notes:

- You can only select one server object at a time.
- You cannot use the **Populate** dialog box to populate a specific version of a module member. The version specification is always associated with the module

and not its members. See Specifying Module Objects for Operations for details.  
 To populate a specific version of a module member, use the **populate** command.

Some of the fields in the Populate dialog box are applicable to DesignSync (non-module) data, and some to module data:

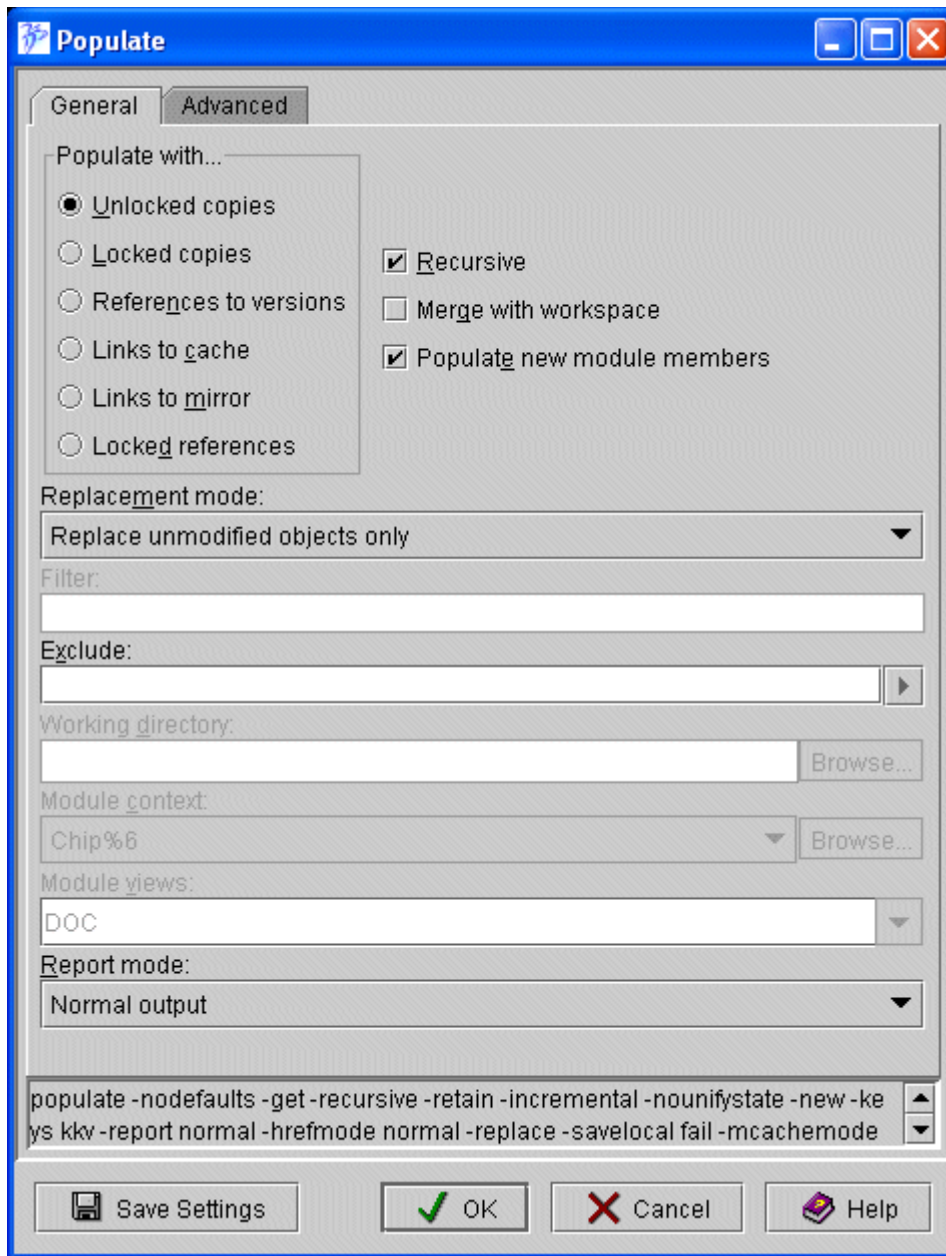
<b>Field in the Populate dialog box</b>	<b>Applicable to DesignSync (non-module) data</b>	<b>Applicable to module data</b>
Populate with Unlocked copies	yes	yes
Populate with Locked copies	yes	yes
Populate with References to versions	yes	yes
Populate with Links to cache	yes	yes
Populate with Links to mirror	yes	no
Populate with Locked references	yes	yes
Recursive	yes	yes
Merge with workspace	yes	yes
Populate new module members	no	yes
Replacement mode	yes	yes
Filter	no	yes
Exclude	yes	yes
Working directory	no	yes
Module context	no	yes
Module Views	no	yes
Report mode	yes	yes
Retain timestamp	yes	yes

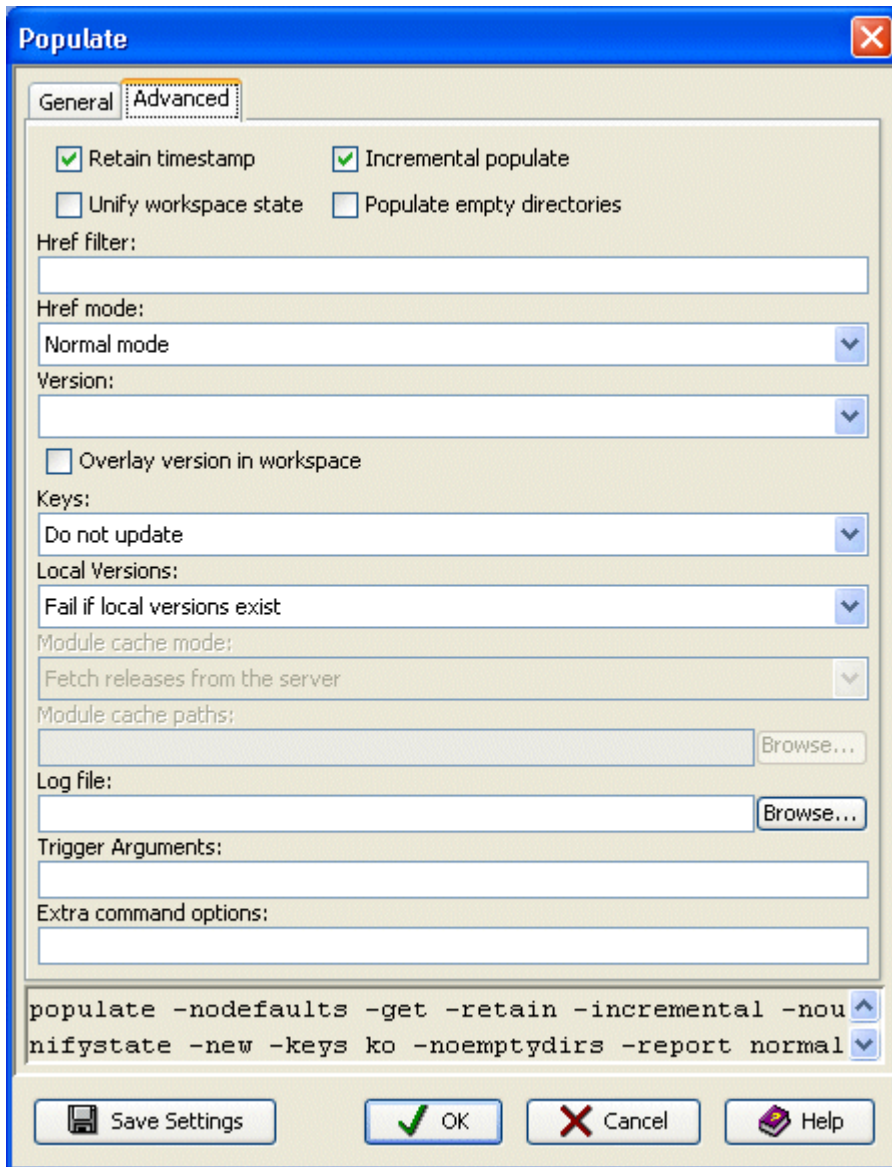
## Setting up a Project or Module Workspace

Unify workspace state	yes	yes
Incremental populate	yes	not typically
Populate empty directories	yes	no
Href filter	no	yes
Href mode	no	yes
Version	yes	yes
Overlay version in workspace	yes	yes
Keys	yes	yes
Local Versions	no	yes
Module Cache Mode	no	yes
Module Cache Paths	no	yes
Trigger Arguments	yes	yes

To interrupt a populate operation, Click the **Stop** button. DesignSync completes the processing of any objects being populated, before stopping the command. See the ENOVIA Synchronicity Command Reference: interrupt command line topic for more information.

**Click on the fields in the following illustration for information.**





## Populate Field Descriptions

### Populate with Unlocked copies

After the operation is over, keep an unlocked copy in your work area. This is the default unless your project leader has defined a default fetch state.

Because you have relinquished any lock you may have had on the object, someone else can check the object out from the vault with a lock in order to modify it.

The **Unlocked files are checked out in Read Only mode** option from the **Tools => Options => General** dialog box determines whether these files are read/write or read-only.

### Populate with Locked copies

After the operation is over, keep a locked copy in your work area. You can continue to work on the object. Others cannot check in new versions of the object as long as you have the branch locked. (The copy in your work area is known as an original.) This option is not available for legacy modules or legacy module configurations. When operating on 5.0 modules, it is module members that are locked; not whole modules. Thus, this option is mutually exclusive with the **Recursive** option, or with a **Module context** specified. To lock a module, use the **Modules => Lock Branch** dialog box, or the **lock** command. See Locking Module Data for details.

### Populate with References to versions

This option lets you acquire references to the versions you have selected.

**Tip:** Do not use this option when operating on a collection object. If you use this option, DesignSync creates a reference in the metadata for the collection object, but member files are not processed and are not included in the metadata.

### Populate with Links to cache

Use this option to link to a shared copy of the design object in a cache directory. This option is available only on UNIX platforms.

### Populate with Links to mirror

Use this option to link to a shared copy of the design object in a mirror directory. This option is available only on UNIX platforms. This option is not available for module data.

### Populate with Locked references

This option lets you acquire a locked reference. If you intend to regenerate the object, create a locked reference to avoid fetching a copy of the object from the vault. This option is not available for legacy modules or legacy module configurations. For other module data, this option is mutually exclusive with the **Recursive** option, or with a **Module context** specified.

### Merge with workspace

Select this option if you want to merge the Latest version of an object in the vault with a locally modified version. This option supports the merging work style where multiple team members can check out the Latest version of an object for editing. The first team

member to check his changes in creates the next version; other team members *merge* their local changes with the new Latest version, and then check the merged version in. To learn how the **Merge with workspace** option operates on module data, see Merging Module Data.

If there are no conflicts, then the merge succeeds, leaving you the merged file in your work area. If there are conflicts, a warning message results. You must edit the merged file to resolve the conflicts before DesignSync allows you to check in the merged version. Conflicts are shown as follows:

```
<<<<<<< local
Lines from locally modified version
=====
Lines from Latest version
>>>>>>> versionID
```

The conflicts are considered resolved when the file no longer contains any of the conflict delimiters (7 less-than, greater-than, or equal signs in the first column). The **Status** column of the List View and the **Is** command indicates if a file has unresolved conflicts.

### Populate new module members

This option only applies to module data. If selected, the populate operation fetches objects that are in the vault, but not currently in the workspace (subject to the **Filter** and **Exclude** fields). This is the default behavior. If unselected, the populate operation updates only objects already in the workspace.

### Replacement mode

The **Replacement mode** determines how the populate operation updates your work area with the data you are fetching. Specify one of the following update methods:

- **Do not replace any objects.** For DesignSync data, do not overwrite locally modified files. Do not remove files that do match the requested version.

For module data, preserve local modifications you made to module members, and leave intact any module members that are in your work area that are not in the requested module version.

- This mode causes the least disruption to your work area; however, it may require you to clean up resulting work area data.

- **Replace unmodified objects only.** For DesignSync data update unmodified objects with the current server version and remove any unmodified objects that are no longer present in the vault, for example, if a file was retired.

**Note:** Objects that have been retired, but remain in the workspace or were re-created in the workspace are considered locally modified.

For module data, update module members that have not been locally modified and that are part of the requested module version. Also remove any unmodified module members that are not part of the requested module version.

- This mode, which is the default behavior for module data, leaves intact any module members you have modified in your workspace.
- **Force overwrite of modified objects.** For DesignSync data, overwrite locally modified files. Also remove managed objects that do not match the requested selector.

For module data, replace or remove module members, regardless of whether you have modified them locally or whether they are part of the requested module version.

- The intent of this mode is to make the workspace match the data being requested (subject to the **Filter** and **Exclude** fields), as closely as possible. Unmanaged data is never removed.

This mode removes items that have been added to a module but have not yet been checked in; their "added" indicator is removed. The data, which is unmanaged, is not removed. See Adding a Member to a Module for information adding members to modules.

If there is a conflict with data fetched from another module, that other data is not removed. See Conflict Handling for details.

If populating a module overlaps with another module already in your workspace, data from that other module is not removed.

This mode is mutually exclusive with the **Merge with workspace** option.

The **Replacement mode** only applies to items that are not filtered out by the **Href filter**, **Filter** or **Exclude** options.

### Filter

See Filter Field.

### Exclude



See Exclude Field.

### Working directory

This field is used when initially populating a workspace with a module from a server. The **Working directory** field only applies when a server module is selected as the object to populate. Specify the path to the directory in your work area where you want the fetched module to reside. Click **Browse** to select the work area where you want to place the module. You can also type the absolute path to the directory.

**Note:** If you are populating a directory with links to a module cache, the Working directory must be new (uncreated).

### Module context

See Module Context Field.

### Module Views

See Module Views Field.

### Report mode

For the **Report mode**, choose the level of information to be reported:

- **Brief output:** Brief output mode reports the following information:
  - Failure messages.
  - Warning messages.
  - Version of each module processed.
  - Creation message for any new hierarchical reference populated as a result of a recursive module populate.
  - Removal message for any hierarchical reference. removed as part of a recursive module populate.
  - Success/failure status.
- **Normal output:** In addition to the information reported in Brief:
  - Informational messages for objects that are successfully updated by the populate operation.
  - Messages for objects excluded from the operation (due to exclusion filters or explicit exclusions).
  - Information about all fetched objects.
- **Verbose output:** In addition to the information reported in **Normal** mode:

- Informational message for every object even if it is not updated, for example objects that are skipped because the version in the workspace is the current version.
- For module data, also outputs information about all objects that are filtered.
- **Errors and Warnings only:** Errors and Warnings output mode reports the following information:
  - Failure messages.
  - Warning messages.
  - Success/failure status messages.

### Incremental populate

Perform a fast populate operation by updating only those folders whose corresponding vault folders have been modified. DesignSync performs an incremental populate by default whenever possible, although it automatically reverts to a full populate when necessary.

To change the default populate mode, your DesignSync administrator can use the SyncAdmin tool, or you can use the **Save Settings** button to override the default mode.

**Note:** This option by itself does not cause state changes of objects in your work area (for example, changing from locked to unlocked objects or unlocked objects to links to the cache). DesignSync changes the states of updated objects only. Furthermore, for an incremental populate, DesignSync only processes folders that contain updated objects, so state changes are not guaranteed. Select **Unify workspace state** to change the state of objects in your work area.

When populating a module, if you use the Version field to specify a different version, DesignSync silently ignores the Incremental option and performs a full populate of the module.

**For incremental populate operations:** If you exclude objects during a populate, a subsequent incremental populate will not necessarily process the folders of the previously excluded objects. DesignSync does not automatically perform a full populate in this case. To guarantee that previously excluded objects are fetched, turn off the **Incremental populate** check box for the subsequent populate operation.

This option usually is not relevant for module data. However, there are two circumstances in which you should deselect this option for a full populate of module data:

- To re-fetch data that was manually removed.

The **Unify workspace state** option also re-fetches such data, but for missing

objects to be considered for the operation, you must deselect the **Incremental populate** option.

- To re-fetch objects that are unchanged from the current module version to the new one, but for which the version in the workspace is incorrect.

For example, let's say there is a module `Chip`, containing the member `alu.v`. You have version 1.4 of the module `Chip`. The Latest version of the module is 1.5. There is no change to the file `alu.v` between module `Chip` versions 1.4 and 1.5; `alu.v` is version 1.3 in both module versions. However, you actually have version 1.2 of `alu.v`, having specifically fetched that file from a previous `Chip` module version 1.3. In this case, an incremental populate of the module would not re-fetch the 1.3 version of `alu.v`. A full populate is required to re-fetch the 1.3 version of `alu.v`.

### Unify workspace state

Sets the state of all objects processed, even up-to-date objects, to the specified state (**Unlocked copies**, **Locked copies**, **References to versions**, **Links to cache**, **Links to mirror**, **Locked references**) or to the default fetch state if no state option is specified. See SyncAdmin Help: Defining a Default Fetch State for more information. If turned off, DesignSync changes the state of only those objects that are not up-to-date. If checked, DesignSync changes the state of the up-to-date objects, as well.

The **Unify workspace state** check box:

- Does not change the state of locally modified objects; select the **Replacement Mode** setting **Force overwrite of modified objects** to force a state change and overwrite the local changes.
- Does not change the state of objects not in the configuration; use the **Replacement Mode** setting **Force overwrite of modified objects** to remove objects not in the configuration.
- Does not cancel locks. To cancel locks, you can check in the locked files, select **Revision Control =>Cancel Checkout** to cancel locks you have acquired, or select **Revision Control =>Unlock** to cancel team members' locks.

### Note:

The **Unify workspace state** option is ignored when you lock design objects. If you check out locked copies or locked references, DesignSync leaves all processed objects in the requested state.

### Populate empty directories

Determines whether empty directories are removed or retained when populating a directory. Select this option to retain empty directories.

If you do not select this option, the populate operation follows the DesignSync registry setting for "Populate empty directories". This registry setting is by your DesignSync administrator using the SyncAdmin tool. By default, this setting is not enabled; therefore, the populate operation removes empty directories.

This option is not applicable to module data. See Directory Versioning for background. To prevent specific empty directories from being created by the populate, use the **Filter** field.

### **Href filter**

See Href Filter Field

### **Href mode**

The **Href mode** option lets you specify how hierarchical references should be evaluated in order to identify the versions of submodules to reference when populating a module recursively. This field is only available when operating on module data.

**Normal mode:** Evaluates the selector and fetches the referenced submodules. If the selector resolves to a static version, by default, the hrefmode is set to **'Static'** for the next level of submodules to be populated. If the selector resolves to a dynamic version, the selector is resolved dynamically and the hrefmode remains Normal. For more information, on understanding this behavior, see Module Hierarchy. For more information on understanding dynamic and static version selectors, see Selector Formats. This is the default Href mode.

**Static mode:** Populates the static version of the submodule that was recorded with the href at the time the parent module's version was created.

**Dynamic mode:** Evaluates the selector associated with the href to identify the version of the submodule to populate.

The **Href mode** option is mutually exclusive with the option to fetch **Locked copies**.

### **Version**

Specify the version number or tag (or any selector or selector list) of the objects on which to operate.

For DesignSync folders, this field is set to the folder's persistent selector list by default. The **Version** field has a pull-down list containing suggested selectors; see Suggested Branches, Versions, and Tags for details.

**Note:** Modules do not support auto-branching, so for module data, the Version field can not contain the auto() construct.

When populating top-level module data, specifying a **Module Version** value changes the workspace selector. When populating sub-modules or files-based versions, the workspace selector does not change, even if the objects are being populated for the first time. When populating a module with a version selector list, the persistent selector is set to an environment that can contain module members from different module version. For more information on blended module member versions, see Module Member Tags. To set or change the workspace selector for a submodule, use the swap commands. For more information on swapping submodules, see Edit-In-Place Methodology. To set or change the workspace selector for files-based objects, see Specifying the Vault Location for a Design Hierarchy.

**Note:** If the selected module version is a static selector, such as a specific version number or tag, any changes to the workspace cannot be checked in.

If the only items selected to populate are module member files in the workspace, and all of the selected members are members of the same module, the **Version** field is replaced by a **Module Version** field. And, the **Module context** field will be disabled, because you are populating the selected member files as they existed in a specific version of the module.

If a legacy module on a server is selected as the object to populate, the **Version** field is replaced by a **Configuration** field, with a default value of "<Default>" (the default configuration). You can select a different configuration of the module from the pull-down list.

If a legacy module configuration on a server is selected as the object to populate, the **Version** field is also replaced by a **Configuration** field. The field's value is set to the configuration that was selected as the object to populate.

If a legacy module already in your workspace is selected as the object to populate, the **Version** field is replaced by a **Configuration** field. The value of the field defaults to the currently fetched configuration. You can select a different configuration of the same module from the pull-down list. If you change the configuration to be populated, the **Recursive** option and the **Replace unmodified objects only** mode are both selected.

### Overlay version in workspace

For DesignSync data, if this option is selected, the version specified in the **Version** field is used to indicate the version to be overlaid on the work area. Overlaying a version does not change the current version status as stored in the workspace metadata. The **Overlay version in workspace** option is often used in conjunction with the **Merge with workspace** option, to merge one branch onto another.

To find out how the **Overlay version in workspace** option operates on module data, see Overlaying Module Data.

## Keys

See Keys Field.

## Module cache mode

This field only applies to module data. A populate operation can link to data in a module cache instead of fetching data from the server, to help decrease fetch time and save disk space.

- **Link to module cache.** (UNIX only) Creates a symbolic link from your workspace to the base directory of a module in the module cache. This is the default mode on UNIX platforms.

**Note:** To use this option, the Working directory must be empty. In order to guarantee that the workspace is empty, the GUI client requires that the Working directory not already exist for the initial mcache populate. The command line does not enforce this, but overwrites any duplicate files in the workspace.

- **Copy from the module cache.** Copies a module from the module cache to your work area.

### Notes:

- This mode is the default mode on Windows platforms.
- This mode only applies to legacy modules.
- **Fetch from the server.** Fetches modules from the server.
- This option overrides the default module cache mode registry setting. If the registry value does not exist, the **Module Cache Mode** selection defaults to **Link to module cache** (UNIX platforms) or to **Copy from the module cache** (Windows platforms).

**Note:** When a link to an mcache is created, the fetch mode, specified by the "Populate with" option, is ignored and the module is fetched according to Module cache mode settings.

## Module Cache Paths

This field only applies to module data. Use the **Module Cache Paths** field to specify paths to the module caches that the populate operation searches, when using a **Module Cache Mode**. If your project leader defined a default module cache path (or paths), the **Module Cache Paths** field will be preset with that default path (or paths).

Click **Browse...** to select one or more paths. When a path is selected with the **Browse...** pop-up dialog box, the selected path is added to the end of the **Module Cache Paths** field.

You can also type paths into the **Module Cache Paths** field. You must specify the absolute path to each module cache. To specify multiple paths, separate paths with a comma (.). The paths must exist.

If no **Module Cache Paths** are specified, the populate operation fetches modules from the server.

### Log populate messages in populate log file

Use this option to log populate messages to a populate log file. The log file provides easy access to the populate messages to allow for later review. This is particularly useful for complex populate operations such as merging a module version from another branch.

Note: If the specified log file already exists, DesignSync will append the results of this populate operation to the file. If the file cannot be created for any reason, such as the directory specified does not exist, or you do not have write permissions to the directory, the populate operation fails.

### Trigger Arguments

See Trigger Arguments Field.

#### Extra command options

List of command line options to pass to the external module change management system. Any options specified with the Extra Command options field are sent verbatim, with no processing by the populate command, to the Tcl script that defines the external module change management system. For more information on external modules, see External Modules.

### Related Topics

[Setting Persistent Populate Views and Filters](#)

[Understanding Module Views](#)

[Merging Module Data](#)

[Module Snapshots](#)

[Using a Module Cache](#)

[How DesignSync Handles Legacy Modules](#)

SyncAdmin Help: Default Fetch State

ENOVIA Synchronicity Command Reference: populate command

ENOVIA Synchronicity Command Reference: ls command

ENOVIA Synchronicity Command Reference: lock command

ENOVIA Synchronicity Command Reference: setselector command

Recursion option

Filter field

Exclude field

Module context field

Retain timestamp option

Href filter field

Keys field

Local Versions field

Trigger Arguments

Command Invocation

Command Buttons

## Using a Mirror

A mirror exactly mimics the data set defined for your project vault. Mirrors provide an easy way for multiple users to point to the file versions that comprise their project's data. The file versions in the mirror belong to the configuration defined by the project lead.

### Examples

- The configuration might be the Latest version of files on the main Trunk branch. A mirror for a development branch might be defined to always contain the file versions on that branch with a specific tag. When the file versions comprising the configuration change, the mirror directory automatically updates with the new version.



- If Latest versions are being mirrored and a new version of a file is checked into the vault, the mirror directory updates with this new version. Without mirroring, users need to frequently update their work areas using the **populate** command to reflect the project's current data set.

Mirror directories can be treated in the same way as your DesignSync work areas. For example, you can use commands such as the **url** or **ls** commands on mirror directories.

### Setting Up Your Workspace

Your team leader will have set up a mirror directory for your project. Use the `setmirror` command to associate your workspace with the project's mirror directory. The `setmirror` command does not have a GUI equivalent. See *ENOVIA Synchronicity Command Reference: setmirror help* for more information on this command.

**Note:** you cannot link to a module mirror from a workspace.

All of the workspace's subdirectories automatically inherit the mirror location set for the top level of the workspace. You cannot set a different mirror on a subdirectory from that of its the parent directory.

To determine if your current work area directory is associated with a mirror, use the `url mirror` command. See the *ENOVIA Synchronicity Command Reference: url mirror help* for more information on this command.

**Note:** To resolve the mirror location, DesignSync does not search above the root of a workspace where a `setvault` has been applied.

So if a `setvault` has been applied to a folder (`/Projects/ASIC/alu`) and you apply the `setmirror` command at a higher-level folder (for example, `/Projects/ASIC`), the `setmirror` command is ignored at and below the folder where the `setvault` occurred (`/Projects/ASIC/alu`). See *ENOVIA Synchronicity Command Reference: setvault help* for more information on this command.

Normally, the path to a mirror is stored exactly as specified by the `setmirror` command. If your mirror directory is set to an auto-mounted directory, you can set a registry key for DesignSync to resolve the path instead.

See *DesignSync Data Manager Administrator's Guide: DesignSync Client Commands Registry Settings* for more information.

### Changing the Mirror Directory Associated with Your Workspace

If the mirror directory for your project changes, run the `setmirror` command from the same directory in which the original `setmirror` command was run. This command updates the workspace's mirror association, which is inherited by lower level directories.

To correct existing workspace links to mirror files, run the **populate** command with these options:

```
populate -recursive -mirror -unifystate
```

This command corrects the links to point to the mirror directory's new location.

### Disassociating Your Workspace from a Mirror Directory

If you no longer need to use a mirror directory, you can disassociate your work area directory from the mirror, by using the `setmirror` command.

### Using the `-mirror` Option to Commands

Once you have associated your workspace with a mirror directory, use the `-mirror` option with `populate`, `ci`, `co`, and `cancel` commands (or select Keep a link to Latest (mirror) when performing these operations through the DesignSync GUI). You can also specify that `-mirror` be used by default, if your team leader did not set that for your project. For details, see Object States.

**Note:** You cannot use the `populate -mirror` command (or select Keep a link to Latest (mirror)) to populate a directory containing a module. In addition, the `ci` command ignores the `-mirror` option if you use it when checking in a module.

Having links to files in the mirror directory ensures that you are always referencing the most up-to-date configuration. However, if other mirror users add files to the mirror, they are not automatically exposed to your work area. Therefore, you should periodically populate your work area directory using the **populate -mirror** command.

### Notes:

- When performing the **populate -mirror** operation, DesignSync creates links only if no file or link already exists in your work area directory; DesignSync does not change the state of existing files and links.

To change the state of existing files and links when you populate your working directory, use the **-force** or **-unifystate** option (or select Overwrite local files if they exist or Unify workspace state in the DesignSync GUI), in addition to the `-mirror` option.

**Caution:** Using the **-force** option overwrites any locally modified files.

- You cannot use the `populate -mirror` command to populate a directory containing a module.

### Related Topics

*DesignSync Data Manager Administrator's Guide: Mirroring Overview*

*DesignSync Data Manager Administrator's Guide: Mirrors Versus LAN Caches*

ENOVIA Synchronicity Command Reference Help: setmirror

ENOVIA Synchronicity Command Reference Help: ci

ENOVIA Synchronicity Command Reference Help: populate

ENOVIA Synchronicity Command Reference Help: co

ENOVIA Synchronicity Command Reference Help cancel

## Setting Permissions for the Mirror

All mirror directories must grant full access to the users of the mirror, unless SUID is being used.

Even though mirrored files are read-only, a user must have write access to the mirror because the user's process sometimes updates the mirror directory when checking in a new version. This client write-through always happens when a mirror is set on the workspace using the `setmirror` command. . DesignSync checks that the user has write access to the mirror.

**Note:** It is possible and highly recommended to enforce the read-only intent of mirror directories (SUID) and not require that all users have write access. For information on configuring SUID, see the *ENOVIA Synchronicity DesignSync Data Manager Installation which is located in the Program Directory*. For information on locating the Installation, see Release Information.

DesignSync creates mirror directories with wide-open permissions:

```
777 -- read/write/execute privileges for owner/group/others
```

### Related Topics

Mirroring Overview

Using a Mirror

Administering Mirrors

*DesignSync Data Manager Administrator's Guide: Mirrors versus Cache*

## Changing the Vault for a Design Hierarchy

To change an existing vault association for a design hierarchy, follow the procedure for setting the vault initially (Specifying the Vault Location for a Design Hierarchy) with the **Apply changes recursively** check box selected. You should select this option when you have already associated a vault location with a work area and want to change that association. Selecting this option is equivalent to specifying the **-recursive** option to the **setvault** command.

You can optionally specify the persistent selector list for the hierarchy by following the vault URL with **@<selectorList>**. This URL syntax is equivalent to executing a **setselector** command following the **setvault** command.

Selecting the **Apply changes recursively** option sets the vault on the specified folder and clears any vault setting for subfolders and files in the hierarchy. Clearing the vault settings causes subfolders and files to inherit the vault setting for the top-level folder. If you do not select this option, only the vault associated with the top-level folder is changed -- the folder is not recursed, so the vault associations for the files and subfolders are not changed. If you specified a persistent selector list, DesignSync sets and clears the persistent selector list in the same way it sets and clears the vault for the hierarchy.

### Related Topics

[Specifying the Vault Location for a Design Hierarchy](#)

[Properties - Revision Control](#)

[ENOVIA Synchronicity Command Reference: setvault command](#)

[ENOVIA Synchronicity Command Reference: setselector command](#)


## Setting a Workspace Root

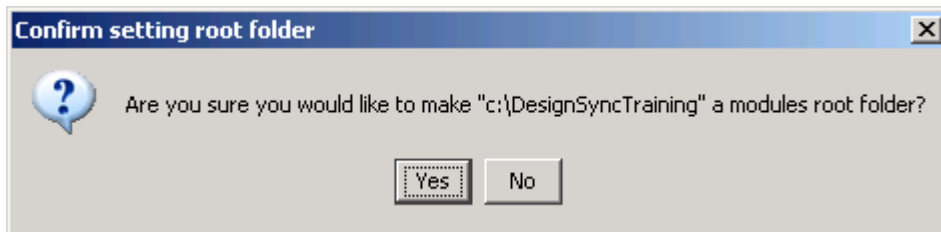
This command allows you to designate the workspace directory to be used as a storage area for a set of local metadata information for a collection of design objects. The metadata includes information about the design files. Setting a workspace root does not create DesignSync object metadata. When you create a module, or checkout or populate files or module based objects, DesignSync stores (or creates) the appropriate metadata for those design objects and stores the metadata in the workspace root folder.

For modules, after the workspace is created/populated with module data and the workspace root set, you can refer to a module by the module instance name, rather than specifying the full module path name.

**Note:** You cannot set a workspace root directory underneath an existing workspace root directory.

**To set a workspace root:**

1. Highlight the folder that you want to set as a workspace root.
2. From the main menu, select **Revision Control=>  Set Root Folder**. A confirmation notice similar to this appears:



3. Click **Yes** to confirm and **No** to cancel.

**Note:** When design data is created or populated with a specified workspace path, the parent of that workspace path is automatically set as the root if there is not root already set.

### Related Topics

ENOVIA Synchronicity Command Reference: mkmod

ENOVIA Synchronicity Command Reference Help: populate

ENOVIA Synchronicity Command Reference Help: co

ENOVIA Synchronicity Command Reference Help: setvault


## Setting Persistent Populate Views and Filters

The Set persistent populate views and filters dialog box is used to apply the views or create the filters that are applied each time a module is populated. The views and/or filters control what is and is not populated into your workspace from the module during the populate command.

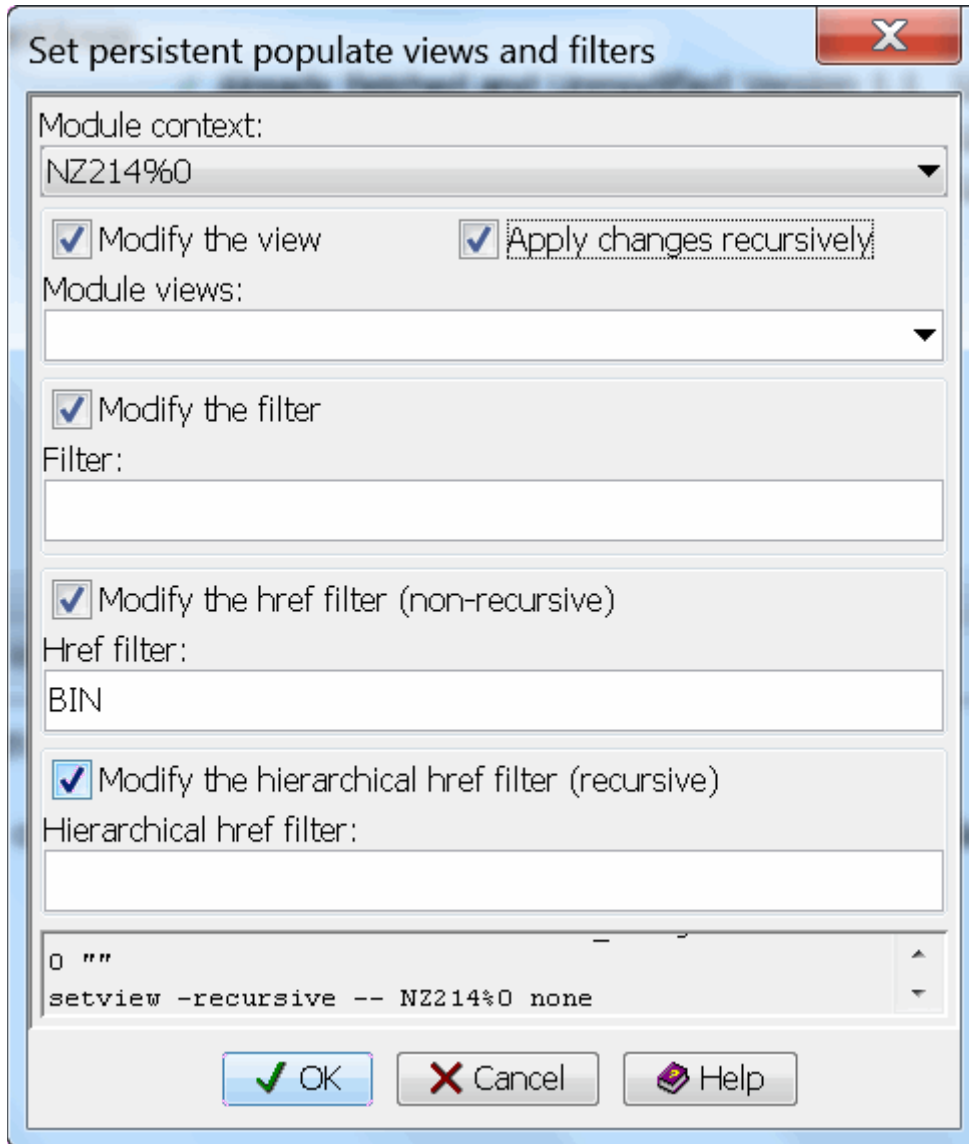
**Note:** The persistent populate parameters are also set when a module is initially populated by the command line when the `-version`, `-view`, `-filter` and/or `-hreffilter` switches are used with the populate command.

**To set persistent settings for future populate commands for a module:**

1. Select the workspace module in the List View pane or the Tree view for which you want to set the persistent views or filters.

2. Select **Revision Control** =>  **Set Persistent Populate Views and Filters**.
3. Enter information or select options as needed from the Set persistent populate views and filters dialog box.
4. Click **OK** to confirm.

Click on the fields in the following illustration for information on each field.



## Field Descriptions

### Module context

Expanding the list-box shows the available module instances for the currently selected base folder. All available module instances are listed alphabetically in the pull-down.

**Note:** There may only be one module listed.

### **Modify the view**

Select this option when you change the persistent view.

### **Apply recursively**

Apply the view recursively through the workspace module hierarchy. By default, this is not selected.

### **Module views**

See the Module Views Field.

### **Modify the filter**

Select this option when you change the filter.

### **Filter**

This field defaults to the current filter setting for the selected workspace module. Enter the expressions that will include and exclude the module members to be populated into your workspace during future populate commands. To remove existing filters, delete the existing text.

### **Modify the href filter**

Select this option when you change the href filter.

### **Href Filter**

This field defaults to the current href filter setting for the selected workspace module. Enter the simple hierarchical reference expressions that will exclude the hierarchical references to be populated into your workspace during future populate commands. To remove existing filters, just delete the existing text.

To use hierarchical href filtering, select Modify the hierarchical href filter. For information on href filters, see Href and Hierarchical Href Filtering.

### **Modify the hierarchical href filter**

Select this option when you change the hierarchical href filter.

### **Hierarchical href Filter**

This field defaults to the current hierarchical href filter setting for the selected workspace module. Enter the hierarchical href expressions that will exclude the hierarchical references to be populated into your workspace during future populate commands. To remove existing filters, delete the existing text.

### **Animated Examples**

Filtering

Persistent populate filter

### **Related Topics**

Filtering Module Data

Filter Field

Href Filter Field

Module Views Field

ENOVIA Synchronicity Command Reference: populate

ENOVIA Synchronicity Command Reference: setfilter

ENOVIA Synchronicity Command Reference: view





# Using DesignSync

## Populating Your Work Area

If you have just joined a project, you need to use **Populate** to check out all the objects from that project into your local working folder. You typically want to populate recursively, (using the **Recursive** option) which traverses a vault folder and recreates the vault folder hierarchy in your work area or traverses a module hierarchy and recreates the hierarchy in your work area. You can also use the Workspace Wizard to help you join a project.

The default mode for populate fetches **Unlocked copies** of objects, unless you have saved a different object state setting with the **Save Settings** button, or your project leader has defined a default fetch state. See [Saving the Setting of an Object's State and SyncAdmin Help: Default Fetch State](#) for information on these two conditions.

After you have initially populated your work area, you need to populate periodically to update your local files, because the vault might have changed since your previous populate. For example, another user might have checked in new design objects or new versions of objects that are already in your work area. Generally, you can perform an incremental populate for these periodic updates. If you need to change the states of files in your work area (for example, if you are changing from locked to unlocked files, or unlocked files to links to the cache), you should perform a full populate. An incremental populate updates only those local folders whose corresponding vault folders have changed, which is faster than a full populate operation. DesignSync performs an incremental populate by default whenever possible, although it automatically reverts to a full populate when necessary. See the [ENOVIA Synchronicity Command Reference:populate](#) command description for the circumstances where DesignSync automatically performs a full populate and for the circumstances where you might choose to specify a full populate.

You can select these types of data for the populate operation:

- A DesignSync folder
- A managed DesignSync (non-module) object
- A module or external module that is already in your workspace (for an update of your workspace)
- A server module branch or server module version (for an initial populate of a workspace)
- An href that is already in your workspace

**Note:** You cannot select an href on the server; to select an href, you must populate the entire module from the server, or filter the data that is fetched.

Specifying an href for a populate is equivalent to specifying the relevant

submodule directly. By specifying an href, the submodule (and the version to be fetched) is identified via the parent module. Specifying an href enables you to fetch a submodule of a module, without necessarily knowing where that module is sourced from, or what version is appropriate.

You can only specify the hrefs directly in the module being addressed, not in its submodules. Selecting an href in your workspace as the object to populate sets the module context accordingly, disabling the Module context field.

- A module folder that is already in your workspace

**Note:** You cannot select a module folder on the server; to select a module folder, you must populate the entire module from the server, or filter the data that is fetched.

- A module member that is already in your workspace

**Note:** You cannot select a module member on the server; to select a module member, you must populate the entire module from the server, or filter the data that is fetched.

- A legacy module already in your workspace
- A legacy module configuration on the server
- A server module branch and one or more server module snapshots. For information on creating a blending environment including a main server module with one or more module snapshots, see Module Snapshots.

#### Additional Notes:

- You can only select one server object at a time.
- You cannot use the **Populate** dialog box to populate a specific version of a module member. The version specification is always associated with the module and not its members. See Specifying Module Objects for Operations for details. To populate a specific version of a module member, use the **populate** command.

Some of the fields in the Populate dialog box are applicable to DesignSync (non-module) data, and some to module data:

Field in the Populate dialog box	Applicable to DesignSync (non-module) data	Applicable to module data
Populate with Unlocked copies	yes	yes
Populate with	yes	yes

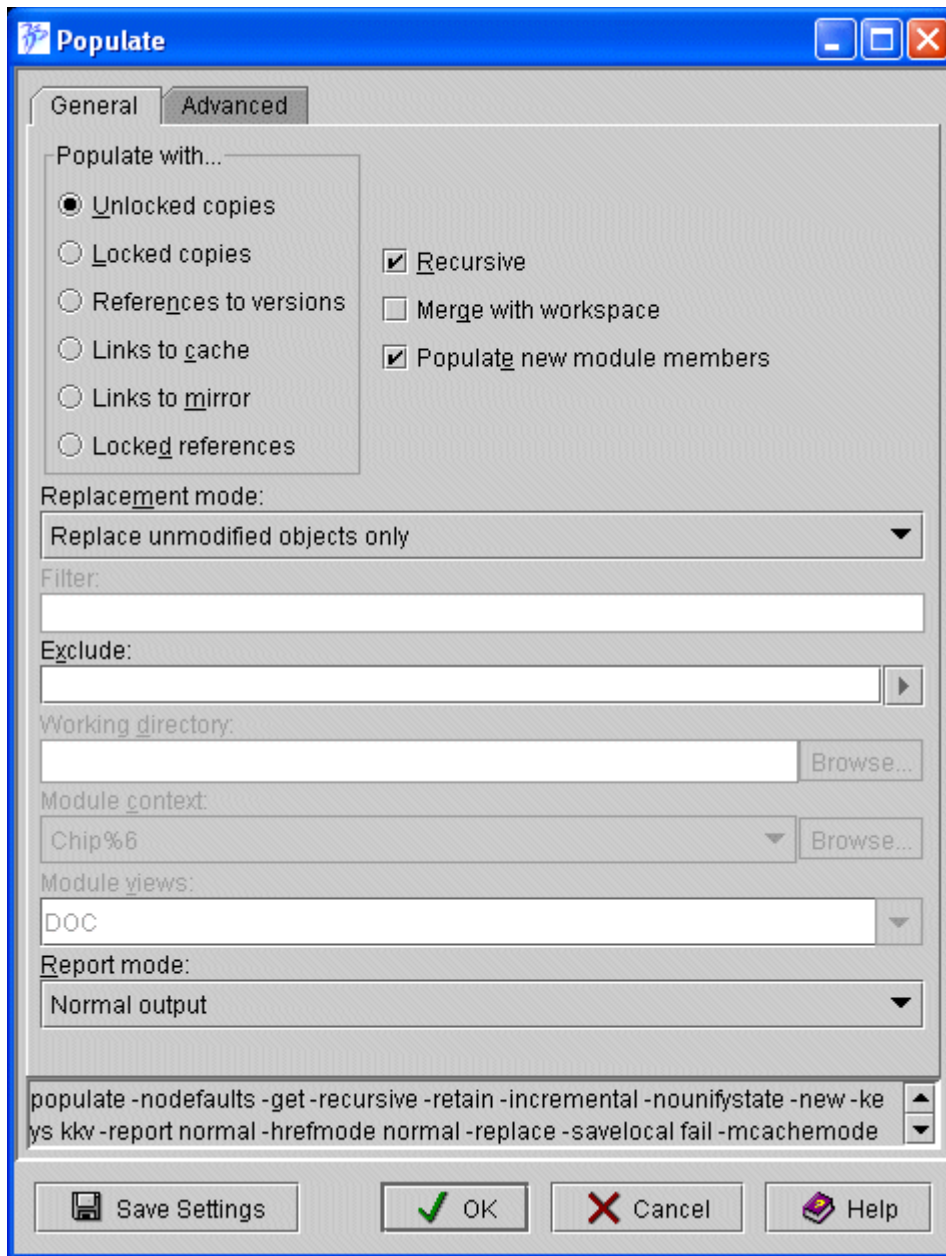
## DesignSync Data Manager User's Guide

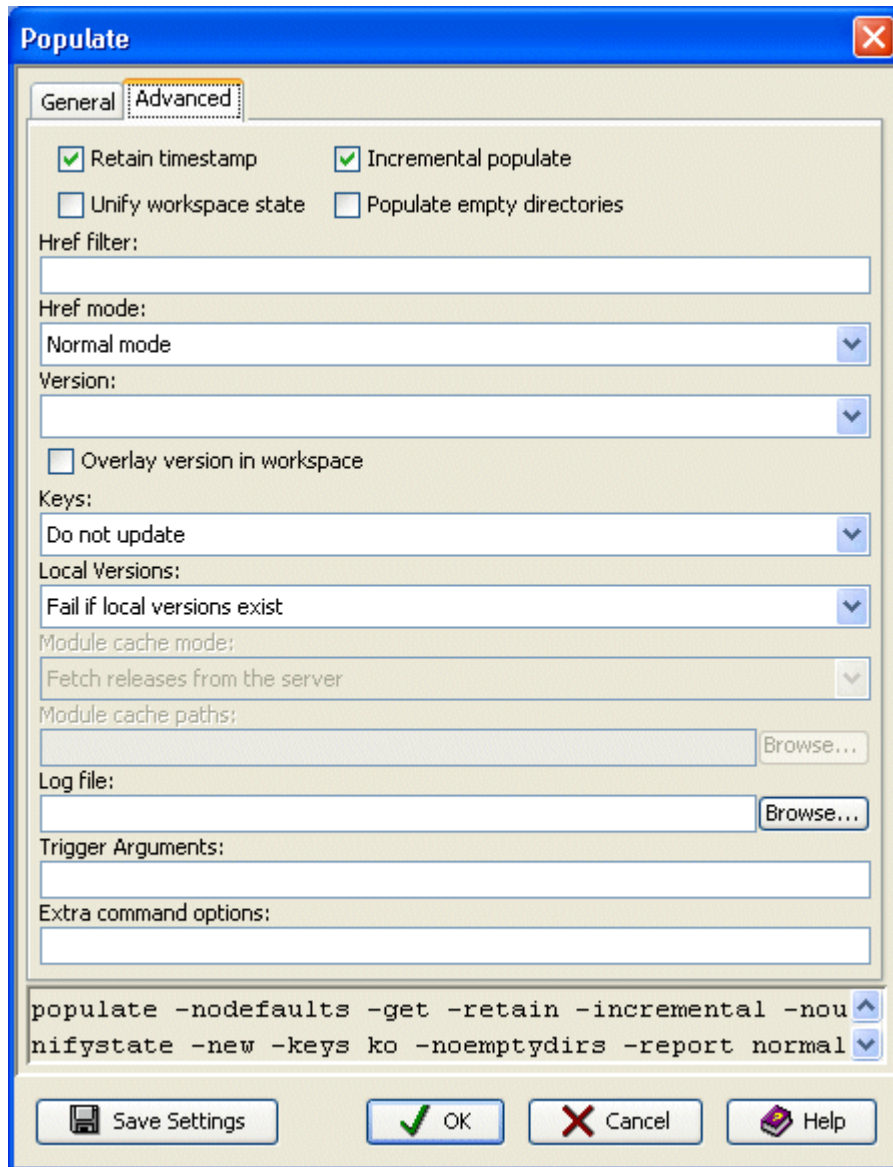
Locked copies		
Populate with References to versions	yes	yes
Populate with Links to cache	yes	yes
Populate with Links to mirror	yes	no
Populate with Locked references	yes	yes
Recursive	yes	yes
Merge with workspace	yes	yes
Populate new module members	no	yes
Replacement mode	yes	yes
Filter	no	yes
Exclude	yes	yes
Working directory	no	yes
Module context	no	yes
Module Views	no	yes
Report mode	yes	yes
Retain timestamp	yes	yes
Unify workspace state	yes	yes
Incremental populate	yes	not typically
Populate empty directories	yes	no
Href filter	no	yes
Href mode	no	yes
Version	yes	yes
Overlay version in	yes	yes

workspace		
Keys	yes	yes
Local Versions	no	yes
Module Cache Mode	no	yes
Module Cache Paths	no	yes
Trigger Arguments	yes	yes

To interrupt a populate operation, Click the **Stop** button. DesignSync completes the processing of any objects being populated, before stopping the command. See the ENOVIA Synchronicity Command Reference: interrupt command line topic for more information.

**Click on the fields in the following illustration for information.**





## Populate Field Descriptions

### Populate with Unlocked copies

After the operation is over, keep an unlocked copy in your work area. This is the default unless your project leader has defined a default fetch state.

Because you have relinquished any lock you may have had on the object, someone else can check the object out from the vault with a lock in order to modify it.

The **Unlocked files are checked out in Read Only mode** option from the **Tools => Options => General** dialog box determines whether these files are read/write or read-only.

### **Populate with Locked copies**

After the operation is over, keep a locked copy in your work area. You can continue to work on the object. Others cannot check in new versions of the object as long as you have the branch locked. (The copy in your work area is known as an original.) This option is not available for legacy modules or legacy module configurations. When operating on 5.0 modules, it is module members that are locked; not whole modules. Thus, this option is mutually exclusive with the **Recursive** option, or with a **Module context** specified. To lock a module, use the **Modules => Lock Branch** dialog box, or the **lock** command. See Locking Module Data for details.

### **Populate with References to versions**

This option lets you acquire references to the versions you have selected.

**Tip:** Do not use this option when operating on a collection object. If you use this option, DesignSync creates a reference in the metadata for the collection object, but member files are not processed and are not included in the metadata.

### **Populate with Links to cache**

Use this option to link to a shared copy of the design object in a cache directory. This option is available only on UNIX platforms.

### **Populate with Links to mirror**

Use this option to link to a shared copy of the design object in a mirror directory. This option is available only on UNIX platforms. This option is not available for module data.

### **Populate with Locked references**

This option lets you acquire a locked reference. If you intend to regenerate the object, create a locked reference to avoid fetching a copy of the object from the vault. This option is not available for legacy modules or legacy module configurations. For other module data, this option is mutually exclusive with the **Recursive** option, or with a **Module context** specified.

### **Merge with workspace**

Select this option if you want to merge the Latest version of an object in the vault with a locally modified version. This option supports the merging work style where multiple team members can check out the Latest version of an object for editing. The first team



member to check his changes in creates the next version; other team members *merge* their local changes with the new Latest version, and then check the merged version in. To learn how the **Merge with workspace** option operates on module data, see Merging Module Data.

If there are no conflicts, then the merge succeeds, leaving you the merged file in your work area. If there are conflicts, a warning message results. You must edit the merged file to resolve the conflicts before DesignSync allows you to check in the merged version. Conflicts are shown as follows:

```
<<<<<<< local
Lines from locally modified version
=====
Lines from Latest version
>>>>>>> versionID
```

The conflicts are considered resolved when the file no longer contains any of the conflict delimiters (7 less-than, greater-than, or equal signs in the first column). The **Status** column of the List View and the **Is** command indicates if a file has unresolved conflicts.

### Populate new module members

This option only applies to module data. If selected, the populate operation fetches objects that are in the vault, but not currently in the workspace (subject to the **Filter** and **Exclude** fields). This is the default behavior. If unselected, the populate operation updates only objects already in the workspace.

### Replacement mode

The **Replacement mode** determines how the populate operation updates your work area with the data you are fetching. Specify one of the following update methods:

- **Do not replace any objects.** For DesignSync data, do not overwrite locally modified files. Do not remove files that do match the requested version.  
  
For module data, preserve local modifications you made to module members, and leave intact any module members that are in your work area that are not in the requested module version.
- This mode causes the least disruption to your work area; however, it may require you to clean up resulting work area data.

- **Replace unmodified objects only.** For DesignSync data update unmodified objects with the current server version and remove any unmodified objects that are no longer present in the vault, for example, if a file was retired.

**Note:** Objects that have been retired, but remain in the workspace or were re-created in the workspace are considered locally modified.

For module data, update module members that have not been locally modified and that are part of the requested module version. Also remove any unmodified module members that are not part of the requested module version.

- This mode, which is the default behavior for module data, leaves intact any module members you have modified in your workspace.
- **Force overwrite of modified objects.** For DesignSync data, overwrite locally modified files. Also remove managed objects that do not match the requested selector.

For module data, replace or remove module members, regardless of whether you have modified them locally or whether they are part of the requested module version.

- The intent of this mode is to make the workspace match the data being requested (subject to the **Filter** and **Exclude** fields), as closely as possible. Unmanaged data is never removed.

This mode removes items that have been added to a module but have not yet been checked in; their "added" indicator is removed. The data, which is unmanaged, is not removed. See Adding a Member to a Module for information adding members to modules.

If there is a conflict with data fetched from another module, that other data is not removed. See Conflict Handling for details.

If populating a module overlaps with another module already in your workspace, data from that other module is not removed.

This mode is mutually exclusive with the **Merge with workspace** option.

The **Replacement mode** only applies to items that are not filtered out by the **Href filter**, **Filter** or **Exclude** options.

### Filter

See Filter Field.

### Exclude

See Exclude Field.

### Working directory

This field is used when initially populating a workspace with a module from a server. The **Working directory** field only applies when a server module is selected as the object to populate. Specify the path to the directory in your work area where you want the fetched module to reside. Click **Browse** to select the work area where you want to place the module. You can also type the absolute path to the directory.

**Note:** If you are populating a directory with links to a module cache, the Working directory must be new (uncreated).

### Module context

See Module Context Field.

### Module Views

See Module Views Field.

### Report mode

For the **Report mode**, choose the level of information to be reported:

- **Brief output:** Brief output mode reports the following information:
  - Failure messages.
  - Warning messages.
  - Version of each module processed.
  - Creation message for any new hierarchical reference populated as a result of a recursive module populate.
  - Removal message for any hierarchical reference. removed as part of a recursive module populate.
  - Success/failure status.
- **Normal output:** In addition to the information reported in Brief:
  - Informational messages for objects that are successfully updated by the populate operation.
  - Messages for objects excluded from the operation (due to exclusion filters or explicit exclusions).
  - Information about all fetched objects.
- **Verbose output:** In addition to the information reported in **Normal** mode:

- Informational message for every object even if it is not updated, for example objects that are skipped because the version in the workspace is the current version.
- For module data, also outputs information about all objects that are filtered.
- **Errors and Warnings only:** Errors and Warnings output mode reports the following information:
  - Failure messages.
  - Warning messages.
  - Success/failure status messages.

### Incremental populate

Perform a fast populate operation by updating only those folders whose corresponding vault folders have been modified. DesignSync performs an incremental populate by default whenever possible, although it automatically reverts to a full populate when necessary.

To change the default populate mode, your DesignSync administrator can use the SyncAdmin tool, or you can use the **Save Settings** button to override the default mode.

**Note:** This option by itself does not cause state changes of objects in your work area (for example, changing from locked to unlocked objects or unlocked objects to links to the cache). DesignSync changes the states of updated objects only. Furthermore, for an incremental populate, DesignSync only processes folders that contain updated objects, so state changes are not guaranteed. Select **Unify workspace state** to change the state of objects in your work area.

When populating a module, if you use the Version field to specify a different version, DesignSync silently ignores the Incremental option and performs a full populate of the module.

**For incremental populate operations:** If you exclude objects during a populate, a subsequent incremental populate will not necessarily process the folders of the previously excluded objects. DesignSync does not automatically perform a full populate in this case. To guarantee that previously excluded objects are fetched, turn off the **Incremental populate** check box for the subsequent populate operation.

This option usually is not relevant for module data. However, there are two circumstances in which you should deselect this option for a full populate of module data:

- To re-fetch data that was manually removed.

The **Unify workspace state** option also re-fetches such data, but for missing

objects to be considered for the operation, you must deselect the **Incremental populate** option.

- To re-fetch objects that are unchanged from the current module version to the new one, but for which the version in the workspace is incorrect.

For example, let's say there is a module `Chip`, containing the member `alu.v`. You have version 1.4 of the module `Chip`. The Latest version of the module is 1.5. There is no change to the file `alu.v` between module `Chip` versions 1.4 and 1.5; `alu.v` is version 1.3 in both module versions. However, you actually have version 1.2 of `alu.v`, having specifically fetched that file from a previous `Chip` module version 1.3. In this case, an incremental populate of the module would not re-fetch the 1.3 version of `alu.v`. A full populate is required to re-fetch the 1.3 version of `alu.v`.

### Unify workspace state

Sets the state of all objects processed, even up-to-date objects, to the specified state (**Unlocked copies**, **Locked copies**, **References to versions**, **Links to cache**, **Links to mirror**, **Locked references**) or to the default fetch state if no state option is specified. See SyncAdmin Help: Defining a Default Fetch State for more information. If turned off, DesignSync changes the state of only those objects that are not up-to-date. If checked, DesignSync changes the state of the up-to-date objects, as well.

The **Unify workspace state** check box:

- Does not change the state of locally modified objects; select the **Replacement Mode** setting **Force overwrite of modified objects** to force a state change and overwrite the local changes.
- Does not change the state of objects not in the configuration; use the **Replacement Mode** setting **Force overwrite of modified objects** to remove objects not in the configuration.
- Does not cancel locks. To cancel locks, you can check in the locked files, select **Revision Control =>Cancel Checkout** to cancel locks you have acquired, or select **Revision Control =>Unlock** to cancel team members' locks.

### Note:

The **Unify workspace state** option is ignored when you lock design objects. If you check out locked copies or locked references, DesignSync leaves all processed objects in the requested state.

### Populate empty directories

Determines whether empty directories are removed or retained when populating a directory. Select this option to retain empty directories.

If you do not select this option, the populate operation follows the DesignSync registry setting for "Populate empty directories". This registry setting is by your DesignSync administrator using the SyncAdmin tool. By default, this setting is not enabled; therefore, the populate operation removes empty directories.

This option is not applicable to module data. See Directory Versioning for background. To prevent specific empty directories from being created by the populate, use the **Filter** field.

### **Href filter**

See Href Filter Field

### **Href mode**

The **Href mode** option lets you specify how hierarchical references should be evaluated in order to identify the versions of submodules to reference when populating a module recursively. This field is only available when operating on module data.

**Normal mode:** Evaluates the selector and fetches the referenced submodules. If the selector resolves to a static version, by default, the hrefmode is set to **'Static'** for the next level of submodules to be populated. If the selector resolves to a dynamic version, the selector is resolved dynamically and the hrefmode remains Normal. For more information, on understanding this behavior, see Module Hierarchy. For more information on understanding dynamic and static version selectors, see Selector Formats. This is the default Href mode.

**Static mode:** Populates the static version of the submodule that was recorded with the href at the time the parent module's version was created.

**Dynamic mode:** Evaluates the selector associated with the href to identify the version of the submodule to populate.

The **Href mode** option is mutually exclusive with the option to fetch **Locked copies**.

### **Version**

Specify the version number or tag (or any selector or selector list) of the objects on which to operate.

For DesignSync folders, this field is set to the folder's persistent selector list by default. The **Version** field has a pull-down list containing suggested selectors; see Suggested Branches, Versions, and Tags for details.

**Note:** Modules do not support auto-branching, so for module data, the Version field can not contain the auto() construct.

When populating top-level module data, specifying a **Module Version** value changes the workspace selector. When populating sub-modules or files-based versions, the workspace selector does not change, even if the objects are being populated for the first time. When populating a module with a version selector list, the persistent selector is set to an environment that can contain module members from different module version. For more information on blended module member versions, see Module Member Tags. To set or change the workspace selector for a submodule, use the swap commands. For more information on swapping submodules, see Edit-In-Place Methodology. To set or change the workspace selector for files-based objects, see Specifying the Vault Location for a Design Hierarchy.

**Note:** If the selected module version is a static selector, such as a specific version number or tag, any changes to the workspace cannot be checked in.

If the only items selected to populate are module member files in the workspace, and all of the selected members are members of the same module, the **Version** field is replaced by a **Module Version** field. And, the **Module context** field will be disabled, because you are populating the selected member files as they existed in a specific version of the module.

If a legacy module on a server is selected as the object to populate, the **Version** field is replaced by a **Configuration** field, with a default value of "<Default>" (the default configuration). You can select a different configuration of the module from the pull-down list.

If a legacy module configuration on a server is selected as the object to populate, the **Version** field is also replaced by a **Configuration** field. The field's value is set to the configuration that was selected as the object to populate.

If a legacy module already in your workspace is selected as the object to populate, the **Version** field is replaced by a **Configuration** field. The value of the field defaults to the currently fetched configuration. You can select a different configuration of the same module from the pull-down list. If you change the configuration to be populated, the **Recursive** option and the **Replace unmodified objects only** mode are both selected.

### Overlay version in workspace

For DesignSync data, if this option is selected, the version specified in the **Version** field is used to indicate the version to be overlaid on the work area. Overlaying a version does not change the current version status as stored in the workspace metadata. The **Overlay version in workspace** option is often used in conjunction with the **Merge with workspace** option, to merge one branch onto another.

To find out how the **Overlay version in workspace** option operates on module data, see Overlaying Module Data.

## Keys

See Keys Field.

## Module cache mode

This field only applies to module data. A populate operation can link to data in a module cache instead of fetching data from the server, to help decrease fetch time and save disk space.

- **Link to module cache.** (UNIX only) Creates a symbolic link from your workspace to the base directory of a module in the module cache. This is the default mode on UNIX platforms.

**Note:** To use this option, the Working directory must be empty. In order to guarantee that the workspace is empty, the GUI client requires that the Working directory not already exist for the initial mcache populate. The command line does not enforce this, but overwrites any duplicate files in the workspace.

- **Copy from the module cache.** Copies a module from the module cache to your work area.

### Notes:

- This mode is the default mode on Windows platforms.
- This mode only applies to legacy modules.
- **Fetch from the server.** Fetches modules from the server.
- This option overrides the default module cache mode registry setting. If the registry value does not exist, the **Module Cache Mode** selection defaults to **Link to module cache** (UNIX platforms) or to **Copy from the module cache** (Windows platforms).

**Note:** When a link to an mcache is created, the fetch mode, specified by the "Populate with" option, is ignored and the module is fetched according to Module cache mode settings.

## Module Cache Paths

This field only applies to module data. Use the **Module Cache Paths** field to specify paths to the module caches that the populate operation searches, when using a **Module Cache Mode**. If your project leader defined a default module cache path (or paths), the **Module Cache Paths** field will be preset with that default path (or paths).

Click **Browse...** to select one or more paths. When a path is selected with the **Browse...** pop-up dialog box, the selected path is added to the end of the **Module Cache Paths** field.



You can also type paths into the **Module Cache Paths** field. You must specify the absolute path to each module cache. To specify multiple paths, separate paths with a comma (.). The paths must exist.

If no **Module Cache Paths** are specified, the populate operation fetches modules from the server.

### **Log populate messages in populate log file**

Use this option to log populate messages to a populate log file. The log file provides easy access to the populate messages to allow for later review. This is particularly useful for complex populate operations such as merging a module version from another branch.

Note: If the specified log file already exists, DesignSync will append the results of this populate operation to the file. If the file cannot be created for any reason, such as the directory specified does not exist, or you do not have write permissions to the directory, the populate operation fails.

### **Trigger Arguments**

See Trigger Arguments Field.

### **Extra command options**

List of command line options to pass to the external module change management system. Any options specified with the Extra Command options field are sent verbatim, with no processing by the populate command, to the Tcl script that defines the external module change management system. For more information on external modules, see External Modules.

### **Related Topics**

[Setting Persistent Populate Views and Filters](#)

[Understanding Module Views](#)

[Merging Module Data](#)

[Module Snapshots](#)

[Using a Module Cache](#)

[How DesignSync Handles Legacy Modules](#)

SyncAdmin Help: Default Fetch State

ENOVIA Synchronicity Command Reference: populate command

ENOVIA Synchronicity Command Reference: ls command

ENOVIA Synchronicity Command Reference: lock command

ENOVIA Synchronicity Command Reference: setselector command

Recursion option

Filter field

Exclude field

Module context field

Retain timestamp option

Href filter field

Keys field

Local Versions field

Trigger Arguments

Command Invocation

Command Buttons

## Changing the State of Objects in Your Work Area

By default, **populate** does not re-fetch file versions that you already have in your workspace. This is a performance optimization.

For example, let's say you currently have cache links in your workspace to version 1.2 of `fileA.txt` and version 1.3 of `fileB.txt`. Version 1.2 is the latest version of `fileA.txt`. Version 1.5 is the latest version of `fileB.txt`. To update your workspace with the Latest versions of all files, you run:

```
stcl> populate -recursive -get
```

The above command fetches a local copy of version 1.5 of `fileB.txt`. The workspace's cache link to version 1.2 of `fileA.txt` remains unchanged, because you already have the requested version of `fileA.txt` in your workspace.

This default behavior is as if the **-nounifystate** option was specified to populate. For populate to instead re-fetch versions that you already have in your workspace in the desired fetch mode, specify the **-unifystate** option. For example, instead of the populate command above, you would run:

```
stcl> populate -recursive -get -unifystate
```

In addition to fetching a local copy of version 1.5 of `fileB.txt`, the above command fetches a local copy of version 1.2 of `fileA.txt`, replacing the workspace's cache link to version 1.2 of `fileA.txt`.

In order to unify the state of all of the managed objects in the workspace, a full populate is performed, as if the **-full** option were specified. A full populate traverses the entire vault hierarchy associated with the workspace, whereas an incremental populate only traverses those sub-directories of the workspace's associated vault that have changed since the workspace was last populated. The **-incremental** option is the default populate behavior.

You can set the default populate behavior to **-incremental** or **-full** using the Synchronicity Administrator (SyncAdmin) Command Defaults pane. However, regardless of whether **-incremental** is set as the default, specifying **-unifystate** results in a full populate. Similarly, even if **-incremental** is specified on the populate command line or selected in the Populate GUI, if **-unifystate** is requested, a full populate is performed.

You can set **-unifystate** as the default populate behavior using the Command Defaults system.

The default behavior of **co** is **-unifystate**. If a user is explicitly fetching objects by using the **co** command or Checkout GUI, the objects are fetched in the state requested by the user.

## Related Topics

[Populating Your Work Area](#)

[Checking Out Design Files](#)

[SyncAdmin Help: Command Defaults](#)

[ENOVIA Synchronicity Command Reference: co](#)

## Specifying Module Objects for Operations

When a module is first populated into the workspace, or when you refer to a module in a server-side command, or when you refer to a module that is not present in the workspace, you must use the full server module address. However, once a module is populated into the workspace, you can refer to the module using a much shorter address.

For example, once a module is populated into the workspace you should be able to simply specify the module name for any subsequent operation:

```
dss> populate <module_name>
```

However, it is important to note that DesignSync supports overlapping modules some of which may have the same name contained within a single module base directory or underneath the same workspace root.

For example, it is possible that your workspace root may contain two modules named Chip (either populated from different servers, or populated as different versions of the same module). To differentiate between workspace modules of the same name, you need to address the module using its module instance name.

### Module Instance Name

DesignSync uses **module instance names** to identify each module as it is populated into the workspace. Set automatically by the server when the module is populated, the module instance name is guaranteed to be a unique identifier for a module within a workspace root directory. The module instance name cannot be specified or changed by the user. The format of the module instance name is:

```
<module name>%<integer>
```

For example, if you populate module “Chip” into your workspace, and there are no other modules named “Chip” present under the workspace root, it will automatically receive the module instance name “Chip%0”. If another module named “Chip” is subsequently populated into a directory under the same workspace root, it would receive the module instance name of “Chip%1”.

The **full workspace address** of a module in a workspace takes the form:

```
<module base directory>/<module instance name>
```

**Note:** It is important to realize that this is the full unique workspace address. In general, it is rare that this form of address will need to be used as an argument to a DesignSync command.

For example, the module ModA is populated into the module base directory `/home/joe/Modules/subdir` and is assigned the module instance name “ModA%0”. The full workspace address of the module will be:

```
/home/joe/Modules/subdir/ModA%0
```

**Note:** It is the module instance name that is used here, not the module name.

### Addressing a Module Object in the Workspace

There are many ways you can address a module in the workspace. DesignSync commands will accept any of the following, and will attempt to resolve the module name automatically:

- The module instance name, providing that the current directory is somewhere below the workspace root directory
- The module name, providing that the current directory is somewhere below the workspace root directory. **Note:** This may not be unique if multiple modules were fetched with the same module name. If a non-unique module name is specified, an error is reported.
- The full workspace address of the module, which is guaranteed to be unique. The full workspace address is the preferred method when needing to reference a module outside the current workspace root.
- The module base directory, although this may not be unique enough to reference a module, as multiple modules may be fetched into the same base directory. If a module base directory is given, then all objects under that directory will be operated on, if the command is operating recursively.

**Note:** Specifying a module base directory does not actually identify a specific module. A module base directory is not appropriate for a command that requires a module as an argument.

### Example

Suppose a workspace, `/home/joe/Modules`, has the following modules populated into it, all with the same base directory:

Module instance “ModA%0” of module “ModA” from server address  
`sync://granite:2647/Modules/ModA;`

Module instance “ModB%0” of module “ModB” from server address  
`sync://granite:2647/Modules/ModB;`

Module instance “ModB%1” of module “ModB” from server address  
`sync://onyx:2647/Modules/ModB;`

## DesignSync Data Manager User's Guide

Module instance “ModC%0” of module “ModC” from server address  
`sync://onyx:2647/Modules/ModC;`

Then the following matches apply, assuming the current working directory is anywhere below the workspace root directory:

```
dss> <command> ModA
```

Matches the single module ModA by its module name

```
dss> <command> ModB%1
```

Matches the single module ModB%1 by its instance name

```
dss> <command> ModC
```

Matches the single module ModC by its module name

```
dss> <command> ModB
```

Matches two modules by their module name, and will fail as being ambiguous.

```
dss> <command> /home/joe/Modules
```

Matches the base directory, but if it is a directory operation, it will continue and run on all five modules if running recursively.

```
dss> <command> Mod*
```

No match. Wildcards on the command line cannot be used to match modules

```
dss> <command> /home/ian/Modules/ModB%0
```

Matches the single module ModB%0 by its full unique workspace address.

```
dss> select ModA
```

This selects the module, and a “select –show” then reports  
`/home/joe/Modules/ModA%0`

### Addressing Hierarchical References in the Workspace

When a hierarchical reference is created from a module, it is given a name. This name can be specified by the user when the hierarchical reference is created and must be unique within the module. If a name is not specified when the hierarchical reference is created, the name will default to the leaf name of the object that is being referenced.

Since hierarchical references have names, we can now address them within the module version. This is achieved for commands that accept hierarchical references as arguments by specifying the hierarchical reference name and a module context:

```
populate -modulecontext Chip ALU
```

The above command would populate the module referenced by the ALU hierarchical reference of module Chip. Clearly, there is a possible name clash here, as the module may contain objects or folders called “ALU” as well as the hierarchical reference called ALU. The hierarchical reference name will take precedence.

Note that wildcard matching *is supported* when specifying hierarchical reference names when module context is supplied. For example, the command:

```
populate -modulecontext Chip AL*
```

would match the ALU hierarchical reference.

## Checking Out Design Data

The **Check Out** dialog box displays the options for the files and folders (directories) you selected. You can exclude files and folders by using the **Exclude** field.

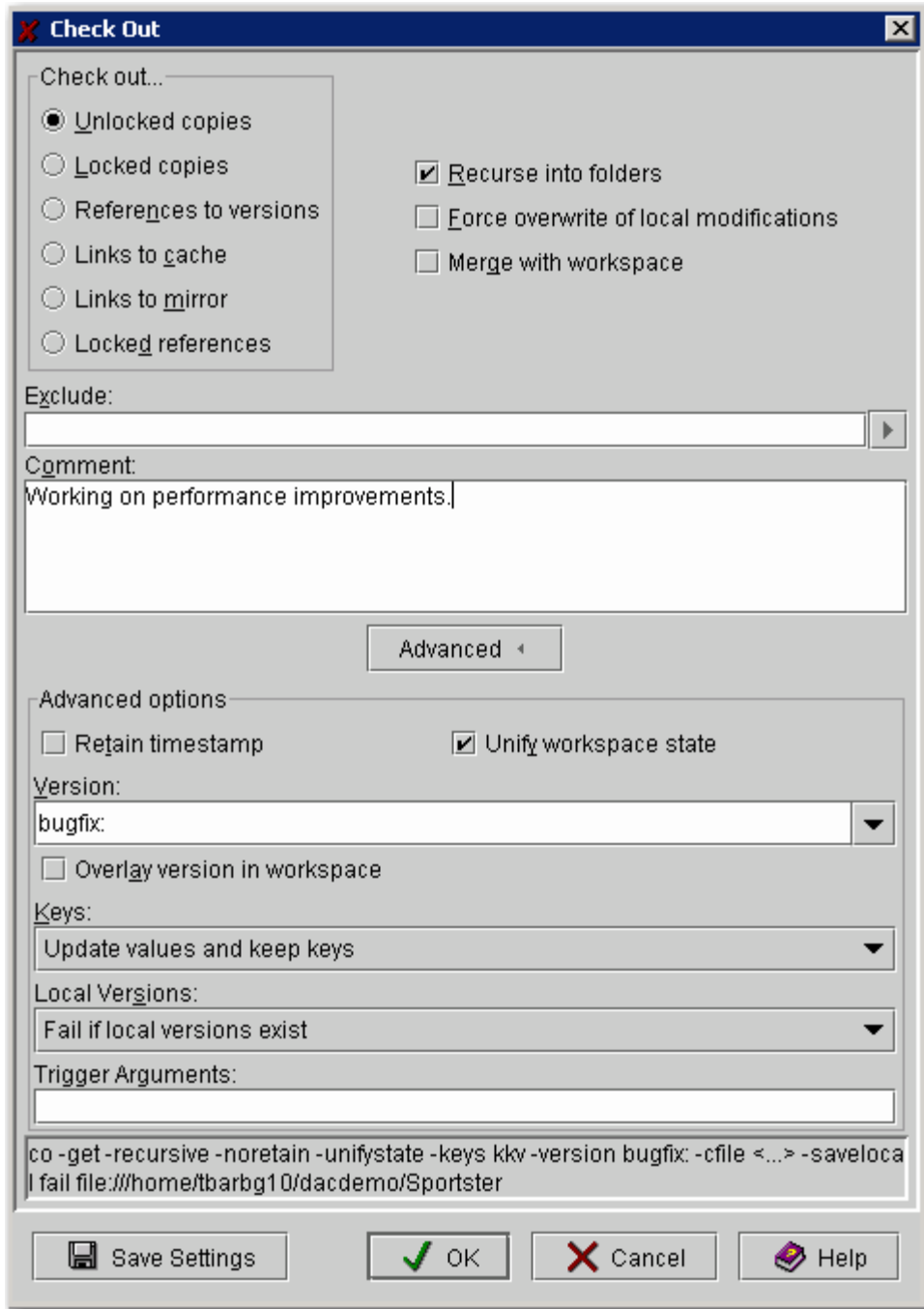
### Notes:

- The Populate dialog box can also be used to fetch individual DesignSync objects. For details, see Populating Your Work Area.
- The **Check Out** dialog box does not operate on module data. To fetch module data, use the **Populate** dialog box, described in Populating Your Work Area.

The default check-out mode is **Unlocked copies** unless you have saved a different object state setting with the **Save Settings** button or your project leader has defined a default fetch state. See Saving the Setting of an Object's State and SyncAdmin Help: Default Fetch State for information on these two conditions.

Click the **OK** button to start the check-out operation. After the check-out operation completes, the success or failure status for each object appears in the Result column of the List View.

**Click on the fields in the following illustration for information.**



## Check Out Field Descriptions

### Check out Unlocked copies

After the operation is over, keep an unlocked copy in your work area. This is the default unless your project leader has defined a default fetch state.



Because you have relinquished any lock you may have had on the file, someone else can check the file out from the vault with lock in order to modify it.

The **Unlocked files are checked out in Read Only mode** option from the **Tools =>Options =>General** dialog box determines whether these files are read/write or read-only.

### **Check out Locked copies**

After the operation is over, keep a locked copy in your work area. You can continue to work on the file. Others cannot check in new versions of the object as long as you have the branch locked. (The copy in your work area is known as an original.)

### **Check out References to versions**

This option lets you acquire references to the versions you have selected.

**Tip:** Do not use this option when operating on a collection object. If you use this option, DesignSync creates a reference in the metadata for the collection object but member files are not processed and are not included in the metadata.

### **Check out Links to cache**

Use this option to link to a shared copy of the design object in a cache directory. This option is available only on UNIX platforms.

### **Check out Links to mirror**

Use this option to link to a shared copy of the design object in a mirror directory. This option is available only on UNIX platforms.

### **Check out Locked references**

This option lets you acquire a locked reference. If you intend to regenerate the object, create a locked reference to avoid fetching a copy of the object from the vault.

### **Recurse into folders**

Performs the operation on all objects in the selected folder and all subfolders.

This option is not available if there are no selected folders.

### **Merge with workspace**

Select this option if you want to merge the Latest version of a file in the vault with a locally modified version. This option supports the merging work style (as opposed to the

locking work style) where multiple team members can check out for editing the Latest version of an object. The first team member to check his changes in creates the next version; other team members 'merge' their local changes with the new Latest version and then check the merged version in. However, you can select **Check out Locked copies** to perform the merge and lock the branch, effectively combining the locking and merging work styles. See Locking and Merging Work Styles to learn more about these work styles.

You can only select this option when you select **Check out Unlocked copies** or **Check out Locked copies**, and you cannot specify any value other than **Latest** for the **Version Selector**. If there are no conflicts, then the merge succeeds, leaving you the merged file in your work area. If there are conflicts, a warning message results. You must edit the merged file to resolve the conflicts before DesignSync allows you to check in the merged version. Conflicts are shown as follows:

```
<<<<<< local  
  
Lines from locally modified version  
  
=====  
  
Lines from Latest version  
  
>>>>>> versionID
```

The conflicts are considered resolved when the file no longer contains any of the conflict delimiters (7 less-than, greater-than, or equal signs in the first column). The **Status** column of the List View and the **Is** command indicates if a file has unresolved conflicts.

### Unify Workspace State

Sets the state of all objects processed, even up-to-date objects, to the specified state (**Unlocked copies**, **Locked copies**, **References to versions**, **Links to cache**, **Links to mirror**, **Locked references**) or to the default fetch state if no state option is specified. See Defining a Default Fetch State for more information. If turned off, DesignSync changes the state of only those objects that are not up-to-date. If checked, DesignSync changes the state of the up-to-date objects, as well.

The **Unify workspace state** check box:

- Does not change the state of locally modified objects; select the **Force overwrite of local modifications** check box to force a state change and overwrite the local changes.
- Does not change the state of objects not in the configuration; use the **Force overwrite of local modifications** check box to remove objects not in the configuration.

- Does not cancel locks. To remove locks, you can either check in the locked files, select **Revision Control =>Cancel Checkout** to cancel locks you have acquired, or select **Revision Control =>Unlock** to cancel team members' locks.

**Note:**

The **Unify workspace state** option is ignored when you lock design objects. If you check out locked copies or locked references, DesignSync leaves all processed objects in the requested state.

**Version**

Specify the version number or tag (or any selector or selector list) of the files on which to operate.

This field is empty by default, which means the persistent selector list for each object is used to determine the version to check out.

The **Version** field has a pull-down list containing suggested selectors; see Suggested Branches, Versions, and Tags for details.

**Overlay Version in Workspace**

If the overlay check box is checked, the version specified in the Version Selector field is used to indicate the version to be overlaid on the work area. Overlaying a version does not change the current version status as stored in the metadata. Overlay is often used in conjunction with the Merge with Local Copy Check Box to merge one branch onto another.

**Related Topics**

Checking in Design Files

Changing the State of Objects in Your Work Area

Operating on Cadence Data

ENOVIA Synchronicity Command Reference: co command

SyncAdmin Help: Default Fetch State

ENOVIA Synchronicity Command Reference: ls

Force overwrite of local modifications option

Exclude field

Comment field

Retain timestamp option

Keys field

Local Versions field

Trigger Arguments

Command Invocation

Command Buttons

## Canceling a Checkout

If you check out an object for editing (with a lock) and then change your mind about creating a new version, you can undo (cancel) the checkout with the **Cancel Checkout** dialog box. When you cancel a checkout, an unlocked copy of the object remains in your work area unless your project leader has defined a default fetch state. You can also request a different object state if you use the **cancel** command. If you have made changes to the object prior to canceling the checkout, you cannot change the object state, thereby overwriting your local modifications, unless you specify the `-force` option to the `cancel` command.

This command only cancels a checkout performed by you. Use the **Unlock** dialog box to unlock an object that is locked by another user.

**Note:** You cannot cancel a checkout on a module member that has been moved or removed in the workspace.

You can select these workspace items for the cancel operation:

- A DesignSync folder
- A managed DesignSync (non-module) object
- A module base folder
- A module folder
- A module member

**Note:** To release the lock on a module branch, use the **Unlock** dialog box.

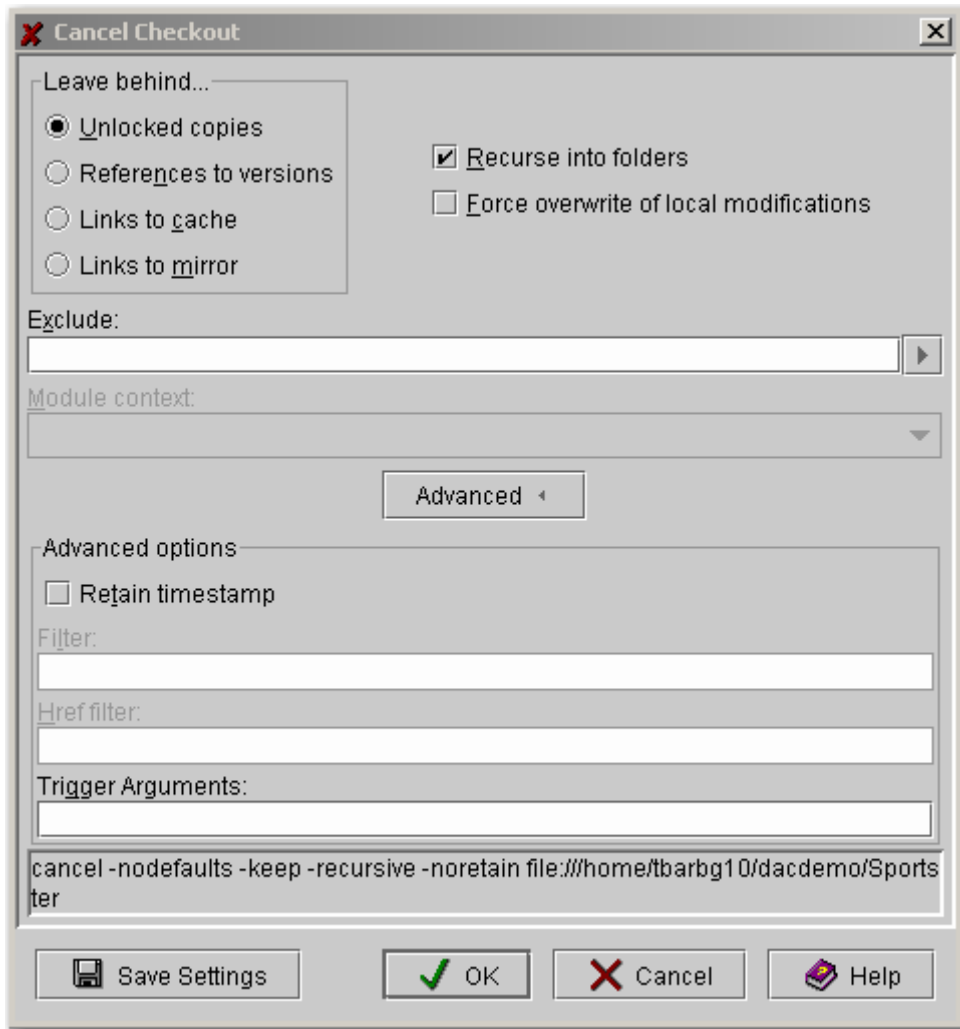
Some of the fields in the Cancel Checkout dialog box are applicable to DesignSync (non-module) data, and some to module data:

Field in the Cancel dialog box	Applicable to DesignSync (non-module) data	Applicable to module data
--------------------------------	--	---------------------------

Leave behind Unlocked copies	yes	yes
Leave behind References to versions	yes	yes
Leave behind Links to cache	yes	yes
Leave behind Links to mirror	yes	no
Recurse into folders	yes	yes
Force overwrite of local modifications	yes	yes
Exclude	yes	yes
Module context	no	yes
Retain timestamp	yes	yes
Filter	no	yes
Href filter	no	yes
Trigger Arguments	yes	yes

To interrupt a recursive cancel operation, click the **Stop** button. DesignSync completes the processing of the current objects being cancelled, before stopping the command. See the ENOVIA Synchronicity Command Reference: interrupt command line topic for more information.

**Click on the fields in the following illustration for information.**



## Cancel Checkout Field Descriptions

### Leave behind Unlocked copies

After the operation is over, keep an unlocked copy in your work area. This is the default unless your project leader has defined a default fetch state.

Because you have relinquished any lock you may have had on the file, someone else can check the file out from the vault with lock in order to modify it.

The SyncAdmin setting **Check out read only when not locking** on the **General tab** determines whether these files are read/write or read-only.

### Leave behind References to versions

This option lets you acquire references to the versions you have selected.

**Tip:** Do not use this option when operating on a collection object. If you use this option, DesignSync creates a reference in the metadata for the collection object but member files are not processed and are not included in the metadata.

### **Leave behind Links to cache**

Use this option to link to a shared copy of the design object in a cache directory. This option is available only on UNIX platforms.

### **Leave behind Links to mirror**

Use this option to link to a shared copy of the design object in a mirror directory. This option is available only on UNIX platforms. This option is not available for module data.

### **Related Topics**

ENOVIA Synchronicity Command Reference: cancel

Unlocking Server Data

Operating on Cadence Data

Recursion option

Force overwrite of local modifications option

Exclude field

Module context field

Filter field

Retain timestamp option

Href filter field

Trigger Arguments

Command Invocation

Command Buttons

## **Checking In Design Data**

The **Check In** dialog box displays the options for checking in the files and folders (directories) you selected. You can exclude files and folders by using the **Exclude** field.

The default check-in mode is **Unlocked copies** unless you have saved a different object state setting with the **Save Settings** button or your project leader has defined a default fetch state. See *Saving the Setting of an Object's State and Defining a Default Fetch State* for information on these two conditions.

### Notes:

- Some files may be pre-excluded from checkin by matching a pattern specified in an exclude file. For more information on exclude files, see *Working with Exclude Files*.
- For DesignSync data, the check-in operation requires that your work area folder be associated with a DesignSync vault. Otherwise, the operation will fail. Usually, you need to set up the vault association only once, as the first step in placing design data under revision control or before you do an initial populate of the work area. For information on setting the vault association, see the *Specifying the Vault Location for a Design Hierarchy*. For information on how to tell if the work area is associated with a vault, see *Verifying That a Vault Has Been Set on a Folder*.
- For collections that have local versions, the check-in operation usually does not change the set of local versions in your workspace. However, there is an exception to this behavior. The check-in operation changes the set of local versions in your workspace when the originally fetched state of the object was Cache or Mirror. In this case, the check-in operation replaces files linked to the cache or mirror with physical copies.
- For module data, if there are workspace changes to a module that were not committed with the checkin (because the changed files were not selected to check in), the module will still be considered locally modified following a successful checkin. The version number of the module in the workspace will be incremented to reflect the new version of the module that was created by the checkin.
- For module data, if there is a newer version of the module in the vault, and the option to Allow version skipping is not selected, then auto-merging may occur.

You can select these types of workspace data for the checkin operation:

- A DesignSync folder
- A managed DesignSync object
- An unmanaged object
- A module – The checkin creates a new version of the module. Any modified member files are checked in. Any objects in an "Added", "Removed", or "Moved" state are also checked in. The module must be populated with a dynamic selector in order to check in any changes.



- A module folder – The checkin creates a new version of the module. Only the selected objects are considered, when determining what member files participate in the checkin. The module must be populated with a dynamic selector in order to check in any changes.

**Note:** When the **Allow check in of new items** option is used with the **Module context** option on a module folder, the operation runs in a folder-recursive manner, checking in any unmanaged non-folder objects in the folder and any subfolders.

- A module member – The checkin creates a new version of the module. Only the selected objects are considered, when determining what member files participate in the checkin. The module must be populated with a dynamic selector in order to check in any changes.

Some of the fields in the Check In dialog box are applicable to DesignSync (non-module) data, and some to module data:

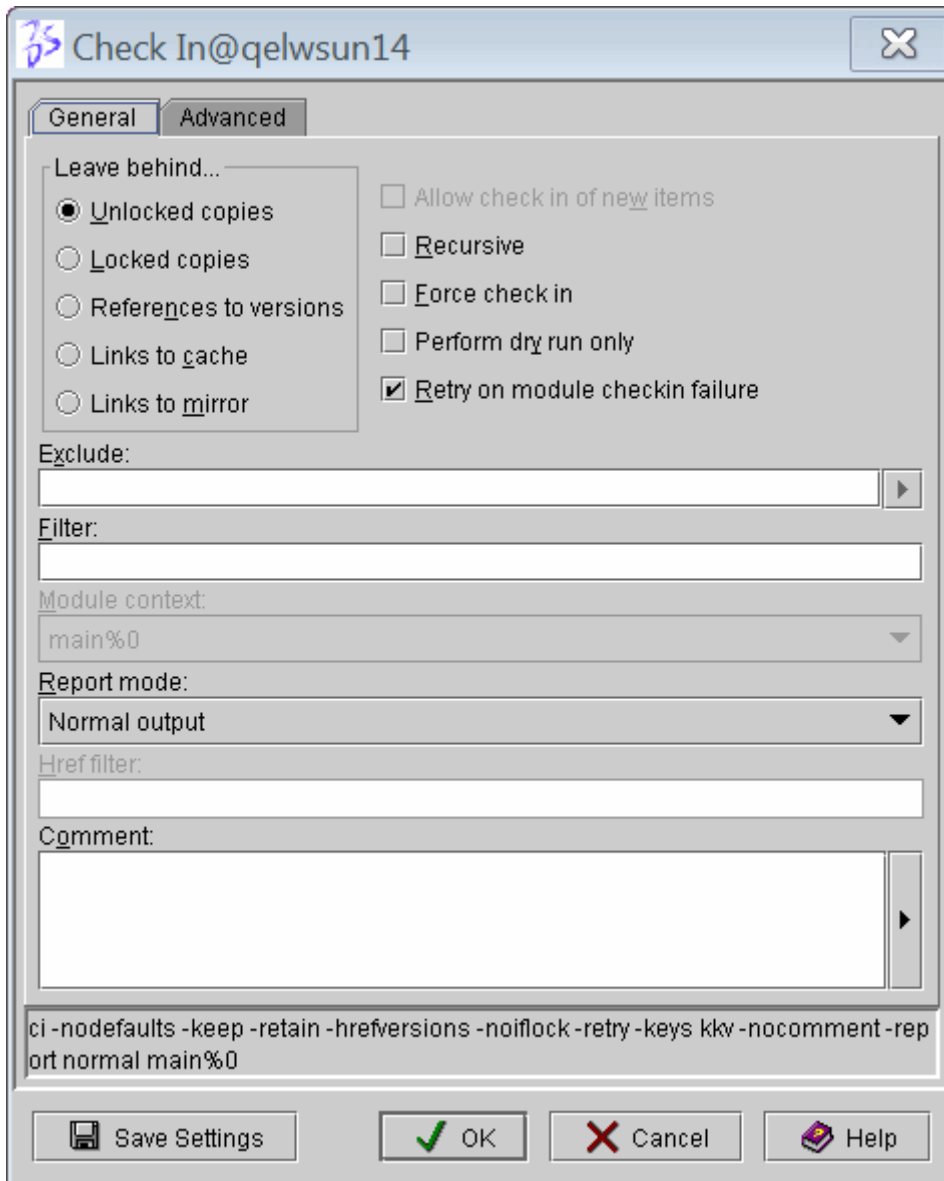
Field in the Check In dialog box	Applicable to File-Based Data	Applicable to module data
Leave behind Unlocked copies	yes	yes
Leave behind Locked copies	yes	yes
Leave behind References to versions	yes	yes
Leave behind Links to cache	yes	yes
Leave behind Links to mirror	yes	no
Allow check in of new items	yes	yes
Recurse into folders	yes	yes
Force check in	yes	yes
Perform dry run only	yes	yes
Retry on Module checkin failure	no	yes
Record href versions	no	yes
Only process locked objects	yes	yes
Exclude	yes	yes
Filter	no	yes
Module context	no	yes

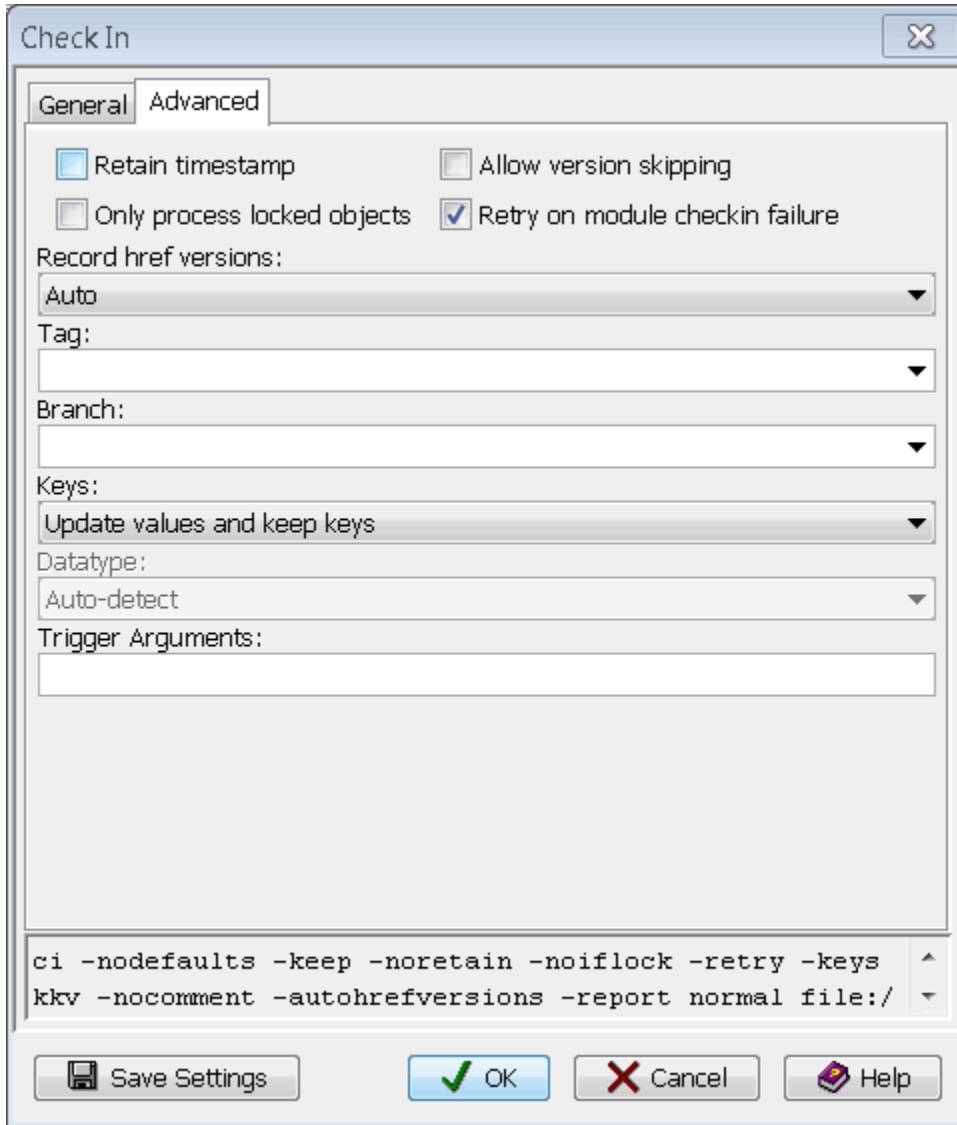
## DesignSync Data Manager User's Guide

Href filter	no	yes
Comment	yes	yes
Retain timestamp	yes	yes
Allow version skipping	yes	yes
Branch	yes	yes
Keys	yes	yes
DataType	yes (New objects only)	yes
Trigger Arguments	yes	yes

To interrupt a checkin operation, Click the **Stop** button. DesignSync completes the processing of any objects being checked in, before stopping the command. See the ENOVIA Synchronicity Command Reference: interrupt command line topic for more information.

**Click on the fields in the following illustration for information.**





## Check In Field Descriptions

### Leave behind Unlocked copies

After the operation is over, keep an unlocked copy in your work area. This is the default unless your project leader has defined a default fetch state.

Because you have relinquished any lock you may have had on the file, someone else can check the file out from the vault with lock in order to modify it.

The **Unlocked files are checked out in Read Only mode** option from the **Tools =>Options** =>General dialog box determines whether these files are read/write or read-only.

### Leave behind Locked copies

After the operation is over, keep a locked copy in your work area. You can continue to work on the file. Others cannot check in new versions of the object as long as you have the branch locked. (The copy in your work area is known as an original.)

### Leave behind References to versions

After a file is checked into the vault, a reference is left behind in your work area. A reference does not have a corresponding file on the file system, but does have DesignSync metadata.

**Tip:** Do not use this option when operating on a collection object. If you use this option, DesignSync creates a reference in the metadata for the collection object but member files are not processed and are not included in the metadata.

### Leave behind Links to cache

Use this option to link to a shared copy of the design object in a cache directory. This option is available only on UNIX platforms.

### Leave behind Links to mirror

Use this option to link to a shared copy of the design object in a mirror directory. This option is available only on UNIX platforms. This option is not available for module data.

### Allow check in of new items

You must select this option if you are checking in an object that has never been checked in before or, in the case of module data, was neither checked in before, nor added to a module. When you check in the object, DesignSync creates a vault for it. This is the vault from which subsequent versions of the object are checked in and out. Once the vault has been created for the object, you do not need to select this option during checkins.

Note that selecting this option also lets you check an object onto a retired branch. The branch is unretired and a new version of the object is created. The retire information is removed from the object history.

If the unmanaged object is being checked in to a module, then smart module detection can be used to determine which module to which the objects are added by using the <Auto-Detect> module context option. You can alternatively specify a **Module context** by selecting a possible module target from the module context drop-down list. For information on how DesignSync determines the target module for checkin, see Understanding Smart Module Detection.

When checking in a folder object to a module, DesignSync automatically performs the operation recursively in a folder-centric way, checking in any unmanaged objects into the target module(s).

**Note:** By default, the Module context option is <Auto-detect> which will attempt to identify the desired module. If no module can be uniquely identified as correct, the command returns a relevant error message.

You may check in unmanaged objects to a new branch using the **Branch** option for DesignSync vault objects only. You cannot check in unmanaged objects to a new module branch; you must first Add them to the module. For more information, see the Branch option.

Files excluded from view by exclude files are not displayed by the DesignSync GUI and are not available for checking in from the GUI. For more information on exclude files, see Working with Exclude Files.

If **Allow version skipping** is selected, and a module member has been removed between the currently fetched module version and the Latest module version, then specifying to **Allow check in of new items** will cause the removed member object to be added back to the module. The version of the member object that is added back into the module is the version that is currently in the workspace. If the version in the workspace has been modified, then a new version of the added back in member object is created.

For example, suppose you have version 1.4 of the Chip module, and version 1.3 of the module member file tmp.txt. The Latest version of Chip is 1.6. Version 1.6 of the Chip module does not contain tmp.txt, because tmp.txt was removed from the module. (See Removing a Member from a Module for details.) Using **Allow version skipping** with **Allow check in of new items** will add the tmp.txt object back into the module, in the module version 1.7 created with your checkin. You could also add the tmp.txt object back into the module by adding the tmp.txt object (see Adding a Member to a Module for details), then checking in with **Allow version skipping**, without having to **Allow check in of new items**.

**Note:** When checking in module data, you cannot specify the **Recursive** or **Branch** options with the **Allow check in of new items** option.

### Force check in

If you check out a file, make no changes to it, and then attempt to check it in, DesignSync informs you that it will not check the file in. If you want to check the file in anyway, you must select this check box. The file will be checked in and a new version created that is identical to the version already in the vault.

Note that you must have a local copy of the file in your working directory for a new version to be created. A new version is not created if the object does not exist or is a reference.

In most cases, not being allowed to check in an unchanged file is reasonable. One reason you may wish to use this option is to keep version numbers synchronized.

### **Perform dry run only**

Select this option to indicate that DesignSync is to treat the operation as a trial run without actually checking in design objects. For module data, module hierarchy processing is included in the output.

This option helps detect whether there are problems that might prevent the checkin from succeeding. Because file and vault states are not changed, a successful dry run does not guarantee a successful checkin. Errors that can be detected without state changes, such as a vault or branch not existing, merge conflicts, or a branch being locked by another user, are reported. Errors such as permissions or access rights violations are not reported by a dry run. Note that a dry run checkin is significantly faster than a normal checkin.

### **Retry on module checkin failure**

Select this option to enable retry for a module checkin failure.

DesignSync will retry to check in a module when there is a communication failure or, if `retryOnModuleCiFailureHook` type client trigger is set, when the conditions set by the trigger are met.

For more information on the `retryOnModuleCiFailureHook`, see [Module Checkin Retry on Failure Trigger Hook](#).

Note: During a communication failure, DesignSync uses the `ModuleFailureRetryAttempts` and `ModuleFailureRetryInterval` registry settings to determine how many retries to attempt before failing the checkin.

### **Record href versions**

Specifies how to process the static hierarchical references associated with the module.

- Auto - Processes the static hrefs based on the type of checkin performed. If the checkin is performed on a module and `Recursive` is selected, DesignSync captures the currently populated versions of the module's hierarchically referenced sub-modules, and records those as part of the next module version, updating the static hierarchical references. If the checkin is performed on a file or folder within a module or a module is specified, but the `Recursive` option is not,

the selected module members are checked in, but the hierarchical references are ignored (not updated) (Default)

- On - When this option is selected and a module is checked in (either an entire module or any of its contents), DesignSync captures the currently populated versions of the module's hierarchically referenced sub-modules, and records those as part of the next module version, updating the static hierarchical references.
- Off - When this option is not selected and a module is checked in, the module members are checked in, but the hierarchical references are ignored (not updated). This is particularly useful if you have out-of-date submodules, or submodule changes that are not ready to be checked in.

This option is ignored for non-module data.

### **Only process locked objects**

Specifies whether to check in all modified objects in the workspace or only targeted files. Changes that are targeted (or locked) are:

- Locked DesignSync vault files or module members.
- Objects that have been added to a module.
- Module members that have been renamed or removed since the last module checkin.

**Only process locked objects** is mutually exclusive with the **Allow checkin of new objects** options.

### **Exclude**

See Exclude Field.

### **Filter**

See Filter Field.

### **Module context**

See Module Context Field.

### **Report mode**

For the **Report mode**, choose the level of information to be reported:

- **Brief output:** Brief output mode reports the following information:
  - Failure messages.
  - Warning messages.



- Informational messages concerning the level of the hierarchy being processed.
- Success/failure status.
- **Normal output:** (Default) In addition to the information reported in Brief mode, output normal mode reports:
  - Informational messages for updated objects.
  - Information about all objects processed.
- **Verbose output:** In addition to the information reported in **Normal** mode:
  - Informational message for every object examined but not updated.
  - Information about all filtered objects.
- **Errors and Warnings only:** Errors and Warnings output mode reports the following information:
  - Failure messages.
  - Warning messages.
  - Success/failure status messages.

### Href filter

See Href Filter Field

### Comment

See Comment Field

### Allow version skipping

By default, this option does not appear in the **Check In** dialog box. For the **Allow version skipping** option appear, the option must be set in the Sync Administrator application. See SyncAdmin's SyncAdmin Help: Command Options for more information.

This option allows you to create a new version, even if the new version is not derived from the Latest version. This happens if your modifications were to a non-Latest version. The new version skips over changes made in intermediate versions, which is why the option is hidden by default.

You typically need to specify **Allow version skipping** when you check into a branch other than the current branch of the data in your workspace, by using the **Branch** field.

For modules, the `-skip` option operates on the module members, but not on module structural changes. When you select `-skip` the objects that are being considered for checkin will replace any version checked in after your workspace was populated. If your workspace contains structural changes, such as moved or removed objects, and they

are not the same as the latest version on the server, you cannot use the `-skip` option to check in those changes.

For example, you have a module containing version 1.2 of `file.txt`. You modify your local copy of `file.txt`. Meanwhile, a later version of the module has been checked in, containing version 1.3 of `file.txt`. Using the **Allow version skipping** option skips over the changes in version 1.3 of `file.txt` and checks in your version as version 1.4. If you were to run a version history with the module manifest option, it would show that version 1.4 of `file.txt` is based on version 1.2, not version 1.3.

If there are structural changes to the module versions between when the workspace was populated and when the workspace module changes are checked in, such as objects, including hierarchical references, added, moved or removed, the changes are incorporated if there are no change conflicts. If a removed file was updated, for example, this creates a change conflict and you must explicitly re-add the file with the **Allow checkin of new items option**.

### Tag

Tags the object version or module version on the server with the specified tag name.

For module objects, all objects are evaluated before the checkin begins. If the objects cannot be tagged, for example if the user does not have access to add a tag or because the tag exists and is immutable, the entire checkin fails.

For other DesignSync objects, if the user does not have access to add a tag, the object is checked in without a tag.

For more information on the access control for the tag command, see ENOVIA Synchronicity Access Control Guide: Access Controls for Tagging. For more information on branch and version tags, see Tagging Versions and Branches.

### Notes:

- If both a tag and a comment are specified for a module version or branch checkin, the comment is also used as the tag comment.
- You cannot tag modules stored on DesignSync server versions prior to 5.1.

### Branch

Specify a branch in the **Branch** field to check in to a branch other than the one from which you checked the object out. The **Branch** field has a pull-down menu from which you can query for existing branches. See Suggested Branches, Versions, and Tags for details.

**Note:** The **Branch** field accepts a branch tag, a version tag, a single auto-branch selector tag, or a branch numeric. It does not accept a selector or selector list. To branch a module, you must specify a branch tag that does not already exist.

When branching a module, you must create a new branch. You cannot specify an existing branch. The module branch checkin creates the new module branch version with the following module member objects:

- Added objects that have not been checked in yet.
- Modified objects belonging to the specified module.
- Unmodified objects belonging to the specified module.
- Objects that are part of the module on the server, but have not been populated into the workspace.
- Objects in the workspace that were removed on the server in a later module version.

**Note:** The module member version in the workspace is always considered the desired version for the ci -branch operation. If you have older member versions in the workspace, those will become the Latest version on the new branch.

When you check a module into the new branch, DesignSync automatically modifies the workspace selector to the Latest version of the new branch tag (<Branch>:Latest).

The **Branch** option, when specified with a module, is mutually exclusive with **Recursive** and **Allow check in of new items**.

## Keys

See Keys Field.

## Datatype

Specify the data type for any module object or any new vault object being checked in.

- **Auto-detect** uses a built-in algorithm to determine whether the object contains only ASCII text or a binary file. (Default)
- **ASCII** creates the new object with a vault data type of ascii.
- **Binary** creates the new object with a vault data type of binary. Binary objects cannot be merged, they can only be replaced. ZIP vaults are always checked in using binary mode, regardless of whether the vault's data type is designated as ascii.

**Note:** To change the data type of an existing vault object, use the url setprop command. For more information, see ENOVIA Synchronicity Command Reference: url setprop. For module objects, you can set the data type when you check in a new module version.

## Trigger Arguments

See Trigger Arguments Field.

## Related Topics

[ENOVIA Synchronicity Command Reference: ci Command](#)

[Checking out Design Files](#)

[Operating on Cadence Data](#)

[Recursion option](#)

[Exclude field](#)

[Filter field](#)

[Module context field](#)

[Comment field](#)

[Retain timestamp option](#)

[Keys field](#)

[Href filter field](#)

[Trigger Arguments](#)

[Command Invocation](#)

[Command Buttons](#)

## Adding a Member to a Module

The Add to Module dialog box adds highlighted objects to a local module in your workspace. The objects must be within the scope of the base directory of the target module and the module must be populated with a dynamic selector. The objects can be files, directories, or collection objects. The locally added objects are checked into the module version that is created by your next checkin. See [Checking In Design Data](#) for more information.



- If the highlighted objects are all files, and the module context can be uniquely determined, the files are added without invoking the Add to Module dialog box. These results are shown in the output window.

- If smart module detection cannot determine the target module, the Select Module Context dialog box is displayed. Once the module context is selected, the results are shown in the output window.
- If the highlighted object contains one or more folder objects, the Add to Module dialog displays.

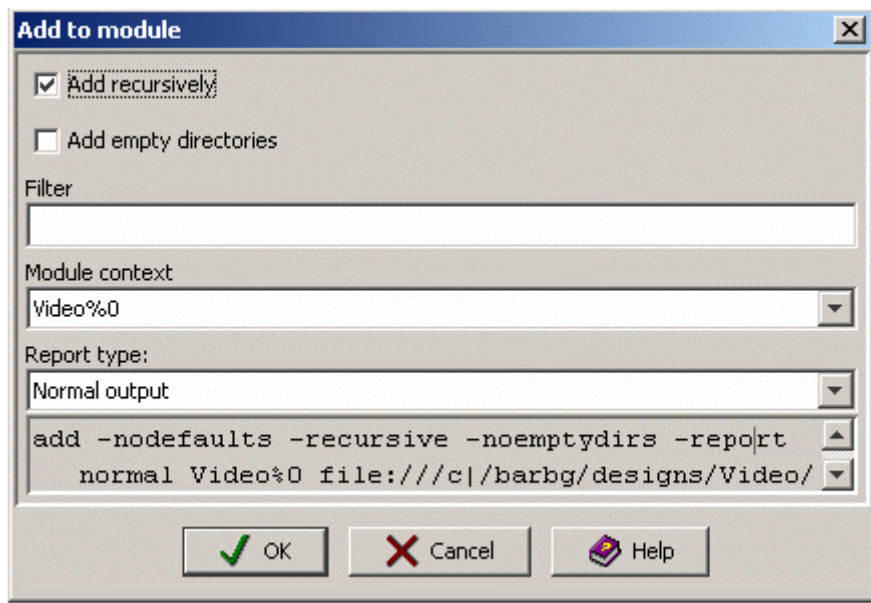
For information about how smart module detection determines the target module, see Understanding Smart Module Detection.

Files excluded from view by exclude files are not displayed by the DesignSync GUI and are not available to Add from the GUI. For more information on exclude files, see Working with Exclude Files.

### To add to a member with Add to Module dialog box:

1. From the main menu, select **Modules =>**  **Add Member** or you can select the  button from the Module Toolbar. You can also select this command from the context menu.
2. The Add to Member dialog box appears. Select options as needed.
3. Click **OK**.

Click on the fields in the following illustration for information.



### Add recursively

For a folder, whether to add only the folder or also recursively add the folder's contents. Note that folders themselves can be module members. If you checked the Add empty directories option, adding a folder without content results in an empty folder as a module member.

By default, the option to **Add recursively** is not selected.

### **Add empty directories**

This option is used with **Add recursively**. When adding members recursively, any folder that contains files is added to the module. The **Add empty directories** option specifies whether directories without content are added to the module.

For example, let's say you're recursively adding "dirA" to a module. "dirA" contains files, and an empty subdirectory "dirB". The option to **Add empty directories** controls whether the empty directory "dirA/dirB" is added.

### **Filter**

Allows you to include or exclude module objects by entering one or more extended glob-style expressions to identify an exact subset of module objects on which to perform the add.

The default for this field is empty.

### **Module Context**

Expanding the list-box shows the available workspace module instances for the currently selected object or objects, including an automatically calculated <Auto-detect> "module context.". All available workspace module instances are listed alphabetically in the pull-down following the calculated <Auto-detect>.

**Note:** If you select <Auto-detect>, and the DesignSync system cannot determine the appropriate module, the command fails with an appropriate error.

### **Report type**

From the pull-down, select the level of information you want to display in the output window:

**Brief output:** Lists errors generated when adding objects to the local module, and success/failure count.

**Normal output:** Lists all objects added to the local module, success/failure count, and beginning and ending messages for the add operation. This is the default output mode.

**Verbose output:** In addition to the information listed for the Normal output mode, lists:

- Skipped objects that are already members of the module.

- Skipped objects that are already members of a different module.
- Skipped objects that are filtered.
- Status messages as each folder is processed.

**Errors and Warnings only:** Lists errors generated when adding objects to the local module, and success/failure count.

## Related Topics

Directory Versioning

ENOVIA Synchronicity Command Reference: add

Filter field

Module context field

Command Invocation

Command Buttons

Context Menu

## Creating Branches

When there is a need for a new branch, a single person, typically a release engineer or project manager, uses the **Make Branch** dialog box to branch all design objects at the same time. Users can then create a new work area for the new branch.

You can select these types of data for the branch operation:

- A DesignSync folder
- A managed DesignSync (non-module) object
- A module version (server only)

**Note:** When a module is branched, the first version on that branch is created immediately, with the content of the module version from which the branch was created.

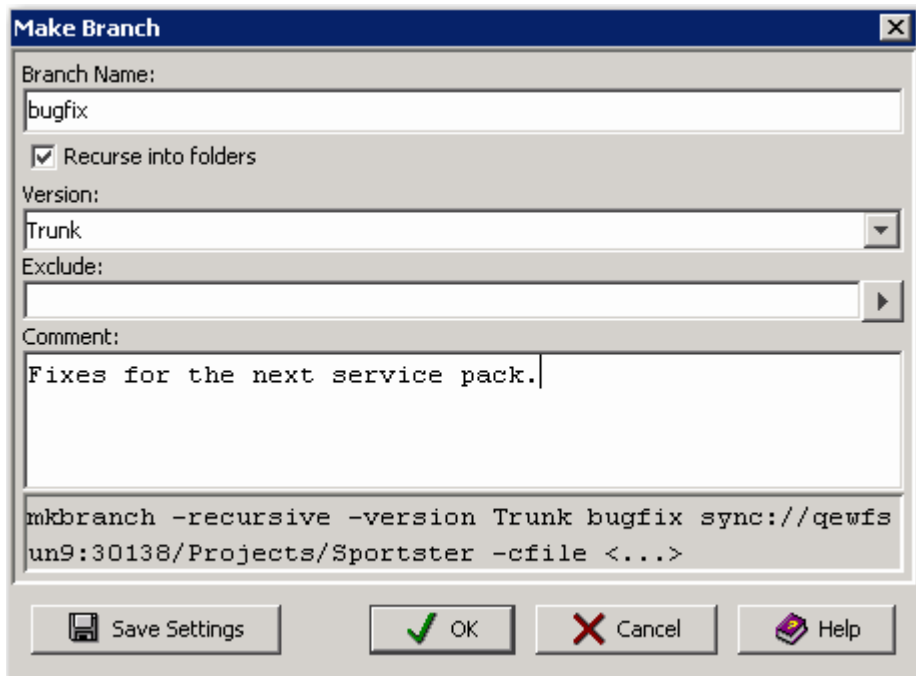
Some of the fields in the Make Branch dialog box are applicable to DesignSync (non-module) data, and some to module versions:

Field in the Make Branch dialog box	Applicable to DesignSync (non-module) data	Applicable to module versions
Branch Name	yes	yes

Recurse into folders	yes	no
Version	yes	no
Exclude	yes	no
Comment	no	yes

To interrupt a recursive branch operation, click the **Stop** button. DesignSync completes the processing of the current objects being branched, before stopping the command. See the ENOVIA Synchronicity Command Reference: interrupt command line topic for more information.

**Click on the fields in the following illustration for information.**



## Make Branch Field Descriptions

### Branch Name

Enter a branch tag name here that is easily understood - for example, "Rel2.1", "ready\_for\_simulation", "current\_demo", "Golden".

Branch names:

- Can contain letters, numbers, underscores (`_`), periods (`.`), hyphens (`-`), and forward slashes (`/`). All other characters, including white space, are prohibited.



- Cannot start with a number and consist solely of numbers and embedded periods (for example, 5, 1.5, or 44.33.22), because there would be ambiguity between the tag name and version/branch dot-numeric identifiers.
- Cannot be any of the following reserved, case-insensitive keywords: Latest, LatestFetchable, VaultLatest, VaultDate, After, VaultAfter, Current, Date, Auto, Base, Next, Prev, Previous, Noon, Orig, Original, Upcoming, SyncBud, SyncBranch, SyncDeleted. Also, avoid using tag names starting with "Sync" (case-insensitive), because this prefix space is reserved for any additional system-defined keywords created in the future.
- DesignSync vaults and legacy modules have an additional restriction: branch names cannot end in --R.

**Note:** Referring to a branch by its branch number is not recommended. DesignSync requires that a branch tag be associated with a branch when it is created. A branch can have more than one branch tag.

### Recurse into folders

For a DesignSync folder, recursively operate on its contents. The set of objects operated on may be reduced by use of the Exclude field. By default, only the contents of the selected folder are operated on. This option, and the **Exclude** field, are not applicable to module data.

### Version

Specify the version number or tag (or any selector or selector list) of the files on which to operate.

The **Version** field has a pull-down menu containing suggested selectors; see Suggested Branches, Versions, and Tags for details.

The **Version** field is active only if you have selected a vault object prior to selecting **Make Branch**. To select a vault object, you can select the local object and then **Go To Vault**. See Viewing the Contents of a Vault for more information.

**Note:** This field is not applicable to module data.

### Related Topics

Parallel (Multi-Branch) Development

Other Branch Operations

Exclude field

Comment field

Command Invocation

Command Buttons

ENOVIA Synchronicity Command Reference: mkbranch

## Tagging Versions and Branches

Tagging is the application of a symbolic name, called a tag, to a version or a branch. Tags can only be applied to objects that are under revision control.

You might identify a set of related file versions that work together by assigning the same tag to them -- for example, "runnable", "Alpha", or "ready\_for\_simulation". In a multi-branch environment (see the Parallel Development topic), you use branch tags to reflect the purpose of the branch -- for example, "rel2.1", "devFeatureX", or "platSolaris". In a single-branch environment, you probably will not be tagging branches but will just work on Trunk, the default tag for branch 1.

A list of tags can be created and displayed from the drop-down menu of the Tag dialog. See SyncAdmin Help: Tags for more information.

The tag operation tags versions or branches of objects in the vault, not the local copies of objects in your work area. Although tags reside on object versions in the vault, the tag operation uses the object versions in your work area to determine the versions to tag in the vault.

**Note:** Applying a tag that already exists on the specified version or branch is considered a successful operation.

If you want to tag an unmanaged object or a locally modified object (whether locked or not) in your work area, you must first check in your changes to create a new version of the object in the vault. Then you can tag that version. If you do not check in a locally modified object before you use the tag command, the tag operation fails for that object and does not tag any version of it in the vault. This is the default behavior of the tag operation.

If you want to tag the version in the vault from which a locally modified object was derived (instead of tagging the new version that contains your changes), you can select the **Tag modified objects** check box. You might use this option, for example, if you have modified objects in your work area and you want to take a "snapshot" of them as they were before you made the changes.

## Tag Naming Conventions

Branch tags and version tags share the same name space. To distinguish version selectors from branch selectors, you append `:<versiontag>` to the branch name; for

example, `Gold:Latest` is a valid branch selector. You can leave off the `Latest` keyword as shorthand; for example, `Gold:` is equivalent to `Gold:Latest`. The selector `Trunk` is also a valid branch selector. `Trunk` is a shorthand selector for `Trunk:Latest`.

You cannot assign the same tag name to both a version and a branch of the same object. For example, a file called `top.v` cannot have both a version tagged `Gold` and a branch tagged `Gold`. However, `top.v` can have a version tagged `Gold` while another file, `alu.v`, can have a branch tagged `Gold`.

Consider adopting a consistent naming convention for branch and version tags to reduce confusion. For example, you might have a policy that branch tags always begin with an initial uppercase letter (`Rel2.1`, for example) whereas version tags do not (`gold`, for example).

## Tagging Module Snapshots

Module snapshots are module versions created on module snapshot branches that include a fixed set of module members. Using the **Tag** command, you can add, remove, or change the member versions contained in the module snapshot. This is the only way to change the content of a module snapshot. You cannot modify the files directly or add or delete them using `Checkin`, `Add`, or `Delete`. When you modify the member versions contained in the module snapshot, you create a new revision on the module snapshot branch containing those changes. This allows you to preserve the manifest of each module snapshot version so it can be recreated if needed.

### To add, remove or change the member version contained in a module snapshot:

1. Select the module member(s) to add or update in the module snapshot.
2. Launch the **Tag** dialog box.
3. Select the desired options including the following:
  - If you are adding or replacing module members Select **Add a Tag**.
  - If you are removing module members select **Delete a Tag**.
  - If you are changing the version of the module member in the snapshot, select the **Replace existing tags** option.
  - For all module snapshot operations, select **Tag module members**.

If you need to remove the tag on a module snapshot, for example to reuse it elsewhere, you can remove the tag from the module version on the snapshot branch. This removes the tag from the module snapshot, but the snapshot branch remains meaning that users can still populate the snapshot using the snapshot branch name.

### To remove the tag on a module version snapshot:

1. Select the module instance in the workspace or the module name on the server.
2. Launch the **Tag** dialog box.
3. Select the desired options including the following:
  - **Delete a Tag.**
  - **Immutable (fixed)**

## DesignSync Objects for Tag

You can select these types of data for the tag operation:

- A DesignSync folder
- A managed DesignSync (non-module) object
- A module version
- A module branch (server only)
- A module member as part of Module Snapshots

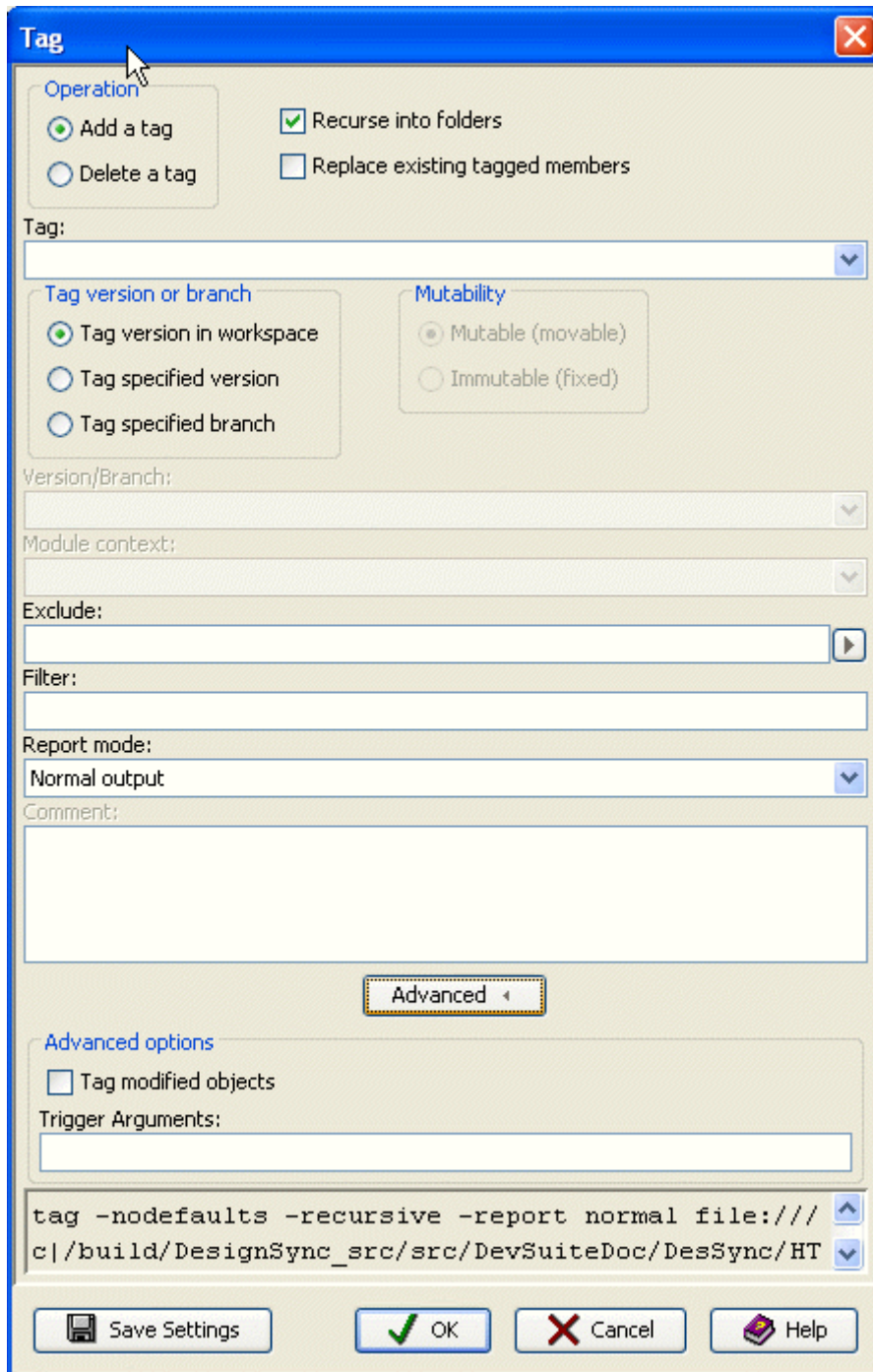
Some of the fields in the Tag dialog box are applicable to DesignSync (non-module) data, and some to module data:

Field in the Tag dialog box	Applicable to DesignSync (non-module) data	Applicable to module data
Add a tag	yes	yes
Delete a tag	yes	yes
Recurse into folders/ Recurse module hierarchy	yes	yes
Replace existing tags	yes	yes
Tag modified objects	yes	no
Tag	yes	yes
Tag version or branch	yes	yes
Mutability	no	yes
Version/Branch	yes	no

Module Context	no	yes
Filter	yes	yes
Exclude	yes	no
Comment	no	yes
Trigger Arguments	yes	no

To interrupt a recursive tag operation, click the **Stop** button. DesignSync completes the processing of the current objects being tagged, before stopping the command. See the ENOVIA Synchronicity Command Reference: interrupt command line topic for more information.

**Click on the fields in the following illustration for information.**



## Tag Field Descriptions

### Add a tag

Select this option to add a tag.

### Delete a tag

Select this option to delete a tag. Because a tag can exist on only one version of a file at a time, the **Tag version or branch** field is ignored when you delete a tag.

### Recurse into folders/Recurse module hierarchy

For a DesignSync folder, recursively operate on its contents. The set of objects operated on may be reduced by use of the Exclude field or the Filter Field. By default, only the contents of the selected folder are operated on. For module members, this option recurses into folders but does not traverse the module hierarchy.

For a DesignSync module instance, recursively operate on the module hierarchy.

**Note:** The module version being tagged is the server version. Any modifications in the workspace, for example, if an older version of the hierarchy is present in the workspace, are ignored. The hierarchical references that the tag command follows are the ones that are checked in as part of the module version on the server.

### Replace existing tags

Move a tag to the version or branch specified in the **Version/Branch** field, even if that tag is already in use on another version or branch. For example, suppose that at the end of every week you want to select the latest files that produce a good demo and tag them "current\_demo". To reuse the "current\_demo" tag in this way, you must check the **Replace existing tags** check box.

By default (if you do not select **Replace existing tags**), a tag operation fails if the tag is already in use, because a tag can be attached to only one version or branch of an object at a time. Note that you can move a tag from a branch to a version or a version to a branch. DesignSync provides a warning message when you do so.

**Note:** If you specify a comment, the tag operation replaces the comment with the new comment. If you do not specify a comment, the operation removes the previous comment associated with tag.

### Tag

Enter a tag name here that is easily understood - for example, "Rel2.1", "ready\_for\_simulation", "current\_demo", "Golden".

Tag names:

- Can contain letters, numbers, underscores (`_`), periods (`.`), hyphens (`-`), and forward slashes (`/`). All other characters, including white space, are prohibited.  
**Note:** The Connected Software and Connected Semiconductor apps do not support the use of forward slash (`/`) in Tag names.

- Cannot start with a number and consist solely of numbers and embedded periods (for example, 5, 1.5, or 44.33.22), because there would be ambiguity between the tag name and version/branch dot-numeric identifiers.
- Cannot be any of the following reserved, case-insensitive keywords: Latest, LatestFetchable, VaultLatest, VaultDate, After, VaultAfter, Current, Date, Auto, Base, Next, Prev, Previous, Noon, Orig, Original, Upcoming, SyncBud, SyncBranch, SyncDeleted. Also, avoid using tag names beginning with "Sync" (case-insensitive); this prefix is reserved for new system keywords.
- DesignSync vaults and legacy modules have an additional restriction: tag names cannot end in --R.

Selecting the down arrow allows you get a list of known tags and selectors. Selecting the Get Tags/Version options, if available, launches the Get Tags/Versions dialog.

### Tag version or branch

Specify which version or branch of the object to tag. These options change depending on the objects selected so you may not see all the options for every object.

- To tag the version of the object in the vault that is the same as the one you have in your work area, select **Tag version in workspace**.
- To tag the module members and create or update a module version on a module snapshot branch, select **Tag module members**.
- To tag an object version in the vault different from the one in your work area, select **Tag specified version** and specify the version number or name in the **Version/Branch** field.
- To tag a specific branch of an object, select **Tag specified branch** and specify the branch name in the **Version/Branch** field.

### Mutability

This field only pertains to modules. The field is ignored when other types of objects are tagged.

When a tag is added, the new tag is marked as **Immutable (fixed)** or **Mutable (movable)**. A mutable tag can be replaced or deleted. To replace or delete an immutable tag, **Immutable (fixed)** must be selected. The default **Mutability** selection is **Mutable (movable)**.

This is not relevant for module snapshot tagging. Module snapshot tagging always creates a content frozen branch (immutable content) with the ability to add and remove hierarchical references and move, add, or delete tags from the members (mutable tags and references.)

### Version/Branch



Specify the version or branch of the objects to tag. If you selected the **Tag specified version** option, you must specify a version selector or selector list. If you selected the **Tag specified branch** option, you must specify a single branch tag, a single version tag, a single auto-branch selector tag, or a branch numeric, but not a selector or selector list.

The **Version/Branch** field has a pull-down menu from which you can query for existing versions and branches. See Suggested Branches, Versions, and Tags for details.

**Note:** This field is not applicable to module data or module snapshots.

### Module context

See Module Context Field.

### Exclude

See Exclude Field.

### Filter

See Filter Field.

### Report Mode

**For the Report mode, choose the level of information to be reported:**

- **Brief output:** Brief output mode reports the following information:
  - Objects that were not tagged (for example, locally modified objects, if you did not specify the `-modified` option).
  - Objects skipped by the tag operation because it created a new configuration map.
  - A count of successes and failures for the tag operation.

**Note:** This count is output only if you are using the `stcl/stclc` command shell.

- **Normal output:** In addition to the information reported in Brief:
  - lists objects that were successfully tagged. (Default)
- **Verbose output:** In addition to the information reported in **Normal** mode:
  - a skip notice for any objects excluded by the `-filter` or `-exclude` options.

### Comment

See Comment Field.

### **Tag modified objects**

For modified DesignSync objects in your work area, tag the version in the vault that you fetched to your work area. You might use this option, for example, if you have modified objects in your work area and you want to take a "snapshot" of them as they were before you made the changes.

If you do not specify this option, when the tag operation encounters a locally modified object, the operation displays an error for the object and does not tag any version of that object in the vault.

**Note:** This option affects modified objects only. If a work area object is unmodified, the tag operation tags the version in the vault that matches the one in your work area.

This option is not applicable to module branches, module versions, and module snapshot branches. It is applicable to module members within an existing module snapshot or when creating a module snapshot.

### **Related Topics**

[What is a Design Configuration?](#)

[What Are Selectors?](#)

[Module Snapshots](#)

[Operating on Cadence Data](#)

[ENOVIA Synchronicity Command Reference: tag Command](#)

[SyncAdmin Help: Tags](#)

[Filter Field](#)

[Exclude field](#)

[Comment field](#)

[Module Context Field](#)

[Trigger Arguments](#)

[Command Invocation](#)

## Command Buttons

## Unlocking Server Data

If you check out a file with a lock and then decide that you do not need to edit the file, you can cancel your check out using the **Cancelling a Checkout** dialog box or the `cancel` command. In cases where you need to unlock a branch that someone else has locked or when you no longer have the file that you checked out in your work area, use the **Unlock** dialog box or the `unlock` command. The following table summarizes when you should use Cancel or Unlock.

Description	Cancel (workspace object)	Unlock (server object)
You hold the lock on an object is in your workspace.	X	
You hold the lock on object not in your workspace		X
Someone else holds the lock on an object in your workspace.		X
Someone else holds the lock on an object not in your workspace.		X

Being able to remove a lock held by another user is an unusual operation. If this capability is not an acceptable policy for your project, you (as the administrator or project leader) can disallow this capability. See the ENOVIA Synchronicity Access Control Guide for more information.

For module data, **Cancel** is typically used to release the lock on a module member. To release the lock on a module branch, **Unlock** must be used. If a module folder in your workspace is selected, and module member locks are released as part of the unlock operation, module member locks in the workspace are also canceled.

You can select these types of data for the unlock operation:

- A DesignSync folder, branch or version (server only)
- A module branch or version (server only)
- A module member (server only)
- A module folder

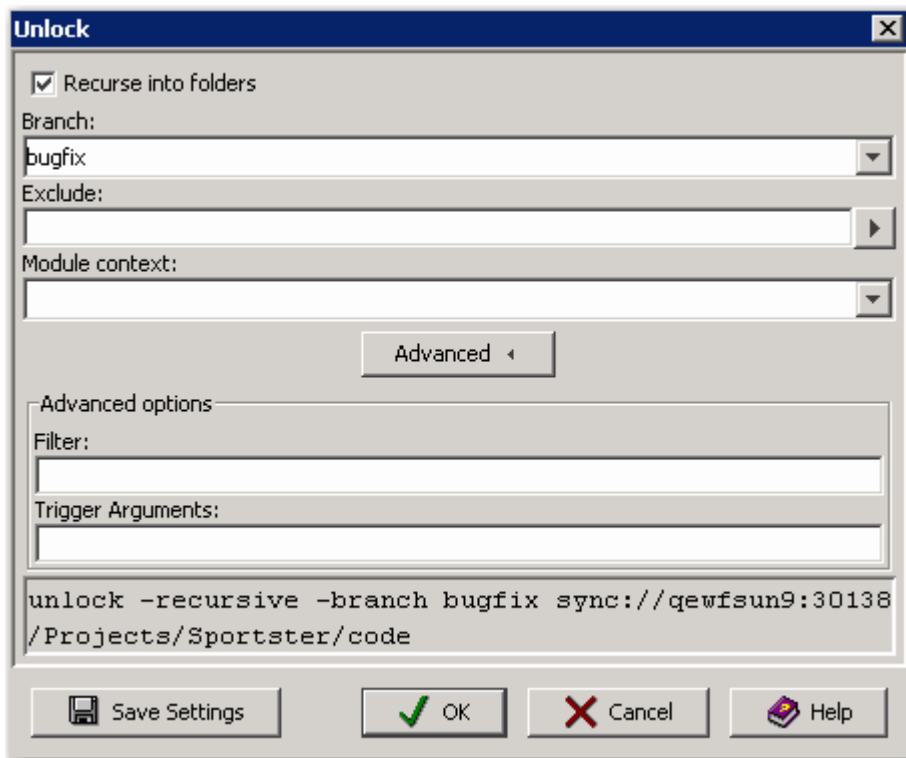
**Note:** You can select multiple objects to unlock within a single unlock operation.

Some of the fields in the Unlock dialog box are applicable to DesignSync (non-module) data, and some to module data:

Field in the Unlock dialog box	Applicable to DesignSync (non-module) data	Applicable to module data
Recurse into folders	yes	yes
Branch	yes	yes
Exclude	yes	yes
Module context	no	yes
Filter	no	yes
Trigger Arguments	yes	yes

To interrupt a recursive unlock operation, click the **Stop** button. DesignSync completes the processing of the current objects being unlocked, before stopping the command. See the ENOVIA Synchronicity Command Reference: interrupt command line topic for more information.

Click on the fields in the following illustration for information.



## Unlock Field Descriptions

## Recurse into folders

For a DesignSync folder, recursively operate on its contents. The set of objects operated on may be reduced by use of the Exclude field. By default, only the contents of the selected folder are operated on.

For a module folder, recursively operate on its contents. The Module context field determines the set of objects operated on. The set of objects operated on may be further reduced by use of the Exclude field and the Filter field. By default, only the contents of the selected folder are operated on.

This option is ignored when a module is selected. The unlock operation does not descend through hierarchical references.

## Branch

Specify the branch(es) to unlock in this field. If you do not specify one or more branches, DesignSync unlocks the branch indicated by the selector of your working area. The **Branch** field has a pull-down menu from which you can query for existing branches. See Suggested Branches, Versions, and Tags for details.

### Notes:

- The Branch field accepts branch tags, version tags, a single auto-branch selector tag, or branch numerics. It does not accept a selector list.
- You can only specify a single branch using the pull-down menu. To select multiple branches, select the desired branches before invoking the unlock command.
- If you have selected multiple server branches to unlock, you cannot specify any other object types within the same command. If other objects are selected with the same command, the command returns an error and does not perform the unlock.

## Related Topics

[Cancelling a Check Out](#)

[Operating on Cadence Data](#)

[ENOVIA Synchronicity Command Reference: unlock](#)

[ENOVIA Synchronicity Command Reference: cancel](#)

[Exclude field](#)

[Module context field](#)

Filter field

Trigger Arguments

Command Invocation

Command Buttons

## **Working with Exclude Files**

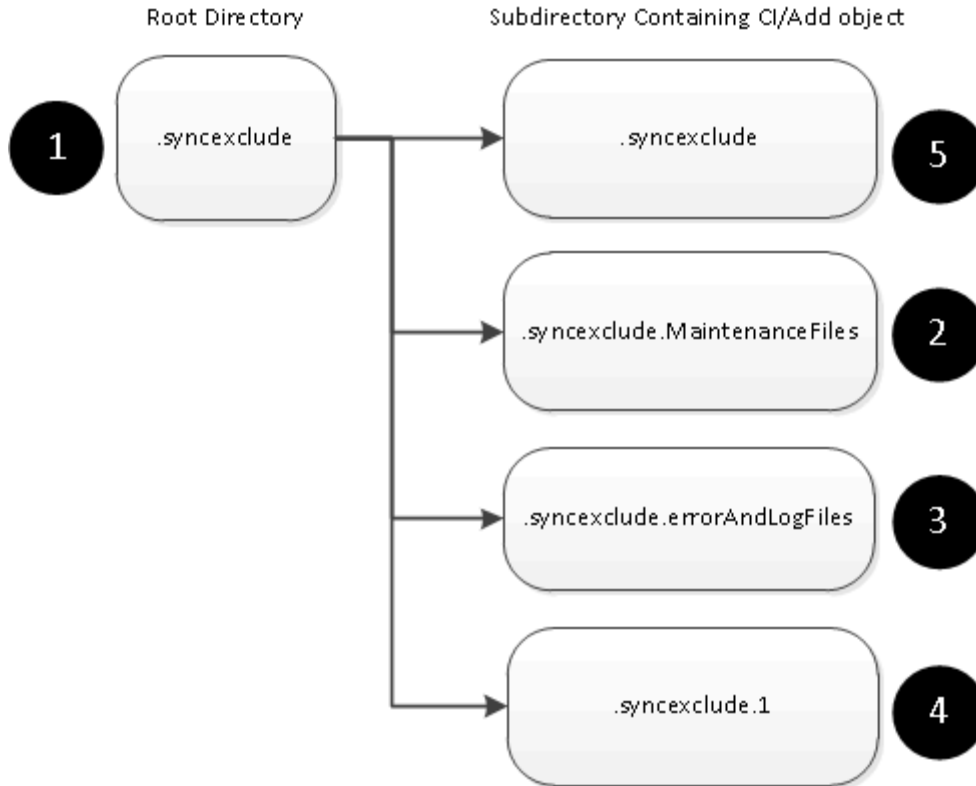
DesignSync features a few different methods for excluding files from revision control operations. Exclude files allow you to exclude unmanaged files from revision control operations. Exclude files are set on a per-directory basis.

### **Exclude File Processing**

Exclude files contain glob-style patterns which are processed in the order they appear within the file. Exclude files are cumulative beginning with the root folder .syncexclude files being processed first, and following down the directory tree to the lowest applicable level.

Syncexclude files always begin with .syncexclude, but can contain an extension. Within a folder, if there is more than one exclude file, the files are processed in alphabetical order. If there is a .syncexclude file, with no suffix, it will be processed last.

### **Exclude File Processing Order**



## Exclude File Formatting

The exclude file is a plain text file that uses glob-style pattern match expressions. You can exclude patterns and include patterns within the file. You may include comments lines beginning with a #.

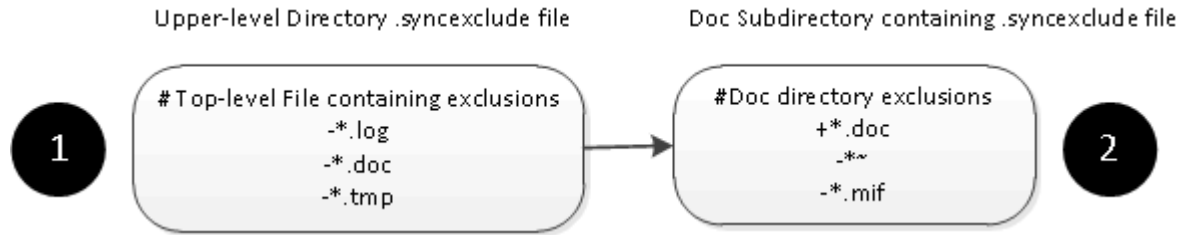
Specify the pattern in one of the following forms:

-<pattern>

+<pattern>

When you use the "-<pattern>" form, you exclude objects that match the specified pattern at the folder level. Using the "- .../<pattern>" will match objects at the folder level and for any subfolders. When you use the "+<pattern>" you create an exception to a previously excluded pattern. An example of using an exclude with an exception might be excluding all .doc files unless they're in the documentation subdirectory.

## Using Exclude and Include in the Exclude Files



So in this example, the base-level .syncexclude, has a line to exclude all doc files. This file is processed first, as shown in the image. In the Doc directory, however, .doc files are accepted, and doc temp files ending with ~ and .mif are excluded. The doc directory and any subfolders of the doc directory would allow .doc files to be included in revision control operations.

## Related Topics

[Adding/Removing Exclusions](#)

[Viewing Exclusions](#)

[Exclude Field](#)

[Filter Field](#)

[Adding a Member to a Module](#)

[Checking in Design Data](#)

## Adding/Removing Exclusions

DesignSync features several ways to add and remove exclusions to a workspace folder.

## Creating and Maintaining Exclusion Files

One of the simplest ways to create and maintain an exclusion file is to create or open the file in a text editor and type in your selections. This provides the optimal control over file naming, and processing order.

1. Create or open the file in the text editor of your choice.
2. Type the desired exclude patterns in the order that you would like them processed, or add new entries in the appropriate processing order. This allows you to create exclude and include filters that operate correctly within the folder, for example:

```
- ".../* .txt"  
+readme.txt
```



Would provide the instruction to filter all .txt files from revision control operations, except for any files with the name readme.txt. So garbage.txt, would be excluded, but readme.txt would be included.

3. Remove any exclusions that are no longer needed.
4. Save the file The file is immediately available for use.

## Add Exclusion Using DesignSync Commands

Another way to add an exclusion is to use the DesignSync **Add Exclusion** command available from the **Revision Control** menu. Select an object to exclude, for example. garbage.txt, and click the **Add Exclusion** command, or select **Add Exclusion** from the context menu. The exclusion pattern is automatically added to all .syncexclude\* files within the folder. If no .syncexclude\* file exists in the folder, DesignSync creates the file ".syncexclude" and adds the exclusion to it.

DesignSync also features an exclude add command which can be typed at the command line. For more information, see the *ENOVIA Synchronicity Command Reference*: exclude add command.

## Remove Exclusion Using DesignSync Commands

The DesignSync **Remove Exclusion** command is available from the **Revision Control** menu. Select an object that you wish to remove or except from the exception list, and select the **Remove Exclusion** command from the **Revision Control** menu or the context menu. If the pattern exactly matches an existing pattern in any of the .syncexclude files within the folder, the exception is removed and a plus exception is added to the file so that the file is not excluded by a higher-level exclusion. If the pattern doesn't match an existing exception, no exception is removed, but the plus exemption is added to the file.

DesignSync also features an exclude remove command which can be typed at the command line. For more information, see the *ENOVIA Synchronicity Command Reference*: exclude remove command.

## Related Topics

[Working with Exclude Files](#)

[Viewing Exclusions](#)

## Viewing Exclusions

DesignSync allows you to view all the exclusions applicable in a workspace folder. The Exclusions view panel shows the list of all exclusions applicable to the specified folder.

This mean it traverses from the selected folder back to the workspace root, examining all `.syncexclude*` files and organizing the output by precedence.

The Exclusions view pane shows the file that contains the exception and the exception rule in the order the rules are applied.

This example was launched from the Chip-Zn32/Doc workspace and shows that there are two `.syncexclude` file (both named `.syncexclude`) one at the higher level excluding all files with the `.doc` file extension and one, within the `doc` directory, including all `.files` with a `.doc` extension.

---

File	Rule
----	----
<code>C:/workspaces/R419/Chip-ZN32/.syncexclude</code>	<code>-*.doc</code>
<code>C:/workspaces/R419/Chip-ZN32/Doc/.syncexclude</code>	<code>+*.doc</code>



## Related Topics

[Working with Exclude Files](#)

[Adding/Removing Exclusions](#)

*Synchronicity Command Reference: exclude list*

## Using Revision Control Keywords

### Revision Control Keywords Overview

The revision control engine used by DesignSync is **RCE**. DesignSync lets you use RCE revision control keywords (sometimes referred to as keys for brevity) in your design files. By including keywords in your files, you can access revision information (such as revision number, author, and comment log), which is stored in the RCE archive that underlies your DesignSync vault.

Use keywords with text files only, not binary files.

The following are the revision control keywords. The keywords are case sensitive.

<b>Keywords</b>	<b>Expansion Contents</b>
\$Aliases\$	List of tag names of the version.
\$Author\$	Who checked the version in.
\$Date\$	The date and time the version was checked in. For modules and module members, the time is stored in GMT. For DesignSync objects, the time is stored in the server's local time.
\$Header\$	Concatenation of Source, Revision, Date, Author, and Locker
\$Id\$	Concatenation of RCSfile, Revision, Date, Author, and Locker
\$KeysEnd\$	Not expanded, and stops further expansion of keys
\$Locker\$	Who has locked this version (empty if not locked)
\$Log\$	The full name of the archive file (Source) followed by the comment log
\$RCSfile\$	The name of the archive file, without the path
\$Revision\$	The version number
\$Source\$	The full name of the archive file, including the path

### Related Topics

Using Revision Control Keywords

## Using Revision Control Keywords

You use revision control keywords by inserting them in your files, typically within comments to keep programs that operate on your files from interpreting the keywords. Keywords are delimited by preceding and trailing dollar signs (\$). There cannot be a space between a keyword and its dollar-sign delimiters. The keywords are case sensitive.

Keywords are expanded (also known as keyword substitution) based on the version of the file that you are checking out. Because the keyword expansion usually changes the length of the file, keywords should only be used with files whose format is position independent, such as text files.

## DesignSync Data Manager User's Guide

You can control keyword expansion from DesignSync using options to the check-in, check-out, and populate commands. For example, you can choose not to expand keyword values when you check out a file, or you can remove keys from a file.

**Note:** Revision control keyword expansion is not supported for:

- files belonging to collections
- the initial checkin of module data from "mkmod -checkin"

The following is an example of keywords in a file before expansion. Note that the keywords appear within comment delimiters, in this case `/*` and `*/`, such as a C file.

```
/*
 * $Aliases$
 * $Author$
 * $Date$
 * $Header$
 * $Id$
 * $Locker$
 * $Log$
 *
 * $RCSfile$
 * $Revision$
 * $Source$
 * $KeysEnd$
 * $Id$
 */
```

By default, keywords are expanded when you check out a file. The following is an example of keyword expansion:

```
/*
 * $Aliases: Key-Example $
 * $Author: jjh $
 * $Date: Thur Jul 4 11:47:20 1997 $
 * $Header: /rca/archive/src/test.c.rca 1.1 Thu Jul 4 11:47:20
1997 jjh Stable $
 * $Id: test.c.rca 1.1 Thu Jul 4 11:47:20 1997 jjh Stable $
 * $Locker: $
 * $Log: test.c.rca $
 *
 * Revision: 1.1 Thu Jul 4 11:47:20 1997 jjh
 * Initial revision
 *
 * $RCSfile: test.c.rca $
 * $Revision: 1.1 $
 * $Source: /rca/archive/src/test.c.rca $
```

```
* $KeysEnd$
* $Id$
*/
```

**Notes:**

- The \$Log\$ keyword, when expanded, permanently adds log information to your file -- later collapsing the keyword leaves the log information in your file. Existing log messages are not replaced. Instead, the new log information is inserted each time the keyword is expanded. \$Log\$ is useful for accumulating a complete change log in a source file, but can result in differences or conflicts when doing merges or file comparisons. \$Log\$ is the only keyword with this behavior.
- When a keyword expansion spans multiple lines, the comment delimiter at the beginning of the line is repeated on subsequent lines. Therefore, make sure that the resulting syntax is valid. For example, in a C file, do not specify:

```
/* $Log$ */
```

The resulting expansion has invalid comment syntax:

```
/* $Log: top.f.rca $
/*
/* Revision: 1.6 Thu Feb 26 09:12:07 1998 Goss
/* *** Fixed defect 1445. *** */
```

Whereas if you specify:

```
/*
* $Log$
*/
```

The resulting expansion is valid:

```
/*
* $Log: top.f.rca $
*
* Revision: 1.6 Thu Feb 26 09:12:07 1998 Goss
* *** Fixed defect 1445. ***
*/
```

**Related Topics**

*DesignSync Data Manager Administrator's Guide: Turning Off Keyword Expansion*

Checking In Design Files

Checking Out Design Files

## DesignSync Data Manager User's Guide

### Populating Your Work Area

ENOVIA Synchronicity Command Reference: ci

ENOVIA Synchronicity Command Reference: co

ENOVIA Synchronicity Command Reference: populate



# Working with Modules

## Specifying Module Objects for Operations

When a module is first populated into the workspace, or when you refer to a module in a server-side command, or when you refer to a module that is not present in the workspace, you must use the full server module address. However, once a module is populated into the workspace, you can refer to the module using a much shorter address.

For example, once a module is populated into the workspace you should be able to simply specify the module name for any subsequent operation:

```
dss> populate <module_name>
```

However, it is important to note that DesignSync supports overlapping modules some of which may have the same name contained within a single module base directory or underneath the same workspace root.

For example, it is possible that your workspace root may contain two modules named Chip (either populated from different servers, or populated as different versions of the same module). To differentiate between workspace modules of the same name, you need to address the module using its module instance name.

### Module Instance Name

DesignSync uses **module instance names** to identify each module as it is populated into the workspace. Set automatically by the server when the module is populated, the module instance name is guaranteed to be a unique identifier for a module within a workspace root directory. The module instance name cannot be specified or changed by the user. The format of the module instance name is:

```
<module name>%<integer>
```

For example, if you populate module “Chip” into your workspace, and there are no other modules named “Chip” present under the workspace root, it will automatically receive the module instance name “Chip%0”. If another module named “Chip” is subsequently populated into a directory under the same workspace root, it would receive the module instance name of “Chip%1”.

The **full workspace address** of a module in a workspace takes the form:

```
<module base directory>/<module instance name>
```



**Note:** It is important to realize that this is the full unique workspace address. In general, it is rare that this form of address will need to be used as an argument to a DesignSync command.

For example, the module ModA is populated into the module base directory `/home/joe/Modules/subdir` and is assigned the module instance name “ModA%0”. The full workspace address of the module will be:

```
/home/joe/Modules/subdir/ModA%0
```

**Note:** It is the module instance name that is used here, not the module name.

### Addressing a Module Object in the Workspace

There are many ways you can address a module in the workspace. DesignSync commands will accept any of the following, and will attempt to resolve the module name automatically:

- The module instance name, providing that the current directory is somewhere below the workspace root directory
- The module name, providing that the current directory is somewhere below the workspace root directory. **Note:** This may not be unique if multiple modules were fetched with the same module name. If a non-unique module name is specified, an error is reported.
- The full workspace address of the module, which is guaranteed to be unique. The full workspace address is the preferred method when needing to reference a module outside the current workspace root.
- The module base directory, although this may not be unique enough to reference a module, as multiple modules may be fetched into the same base directory. If a module base directory is given, then all objects under that directory will be operated on, if the command is operating recursively.

**Note:** Specifying a module base directory does not actually identify a specific module. A module base directory is not appropriate for a command that requires a module as an argument.

#### Example

Suppose a workspace, `/home/joe/Modules`, has the following modules populated into it, all with the same base directory:

Module instance “ModA%0” of module “ModA” from server address  
`sync://granite:2647/Modules/ModA;`

Module instance “ModB%0” of module “ModB” from server address  
`sync://granite:2647/Modules/ModB;`

## DesignSync Data Manager User's Guide

Module instance "ModB%1" of module "ModB" from server address  
`sync://onyx:2647/Modules/ModB;`

Module instance "ModC%0" of module "ModC" from server address  
`sync://onyx:2647/Modules/ModC;`

Then the following matches apply, assuming the current working directory is anywhere below the workspace root directory:

```
dss> <command> ModA
```

Matches the single module ModA by its module name

```
dss> <command> ModB%1
```

Matches the single module ModB%1 by its instance name

```
dss> <command> ModC
```

Matches the single module ModC by its module name

```
dss> <command> ModB
```

Matches two modules by their module name, and will fail as being ambiguous.

```
dss> <command> /home/joe/Modules
```

Matches the base directory, but if it is a directory operation, it will continue and run on all five modules if running recursively.

```
dss> <command> Mod*
```

No match. Wildcards on the command line cannot be used to match modules

```
dss> <command> /home/ian/Modules/ModB%0
```

Matches the single module ModB%0 by its full unique workspace address.

```
dss> select ModA
```

This selects the module, and a "select -show" then reports  
`/home/joe/Modules/ModA%0`

### Addressing Hierarchical References in the Workspace

When a hierarchical reference is created from a module, it is given a name. This name can be specified by the user when the hierarchical reference is created and must be unique within the module. If a name is not specified when the hierarchical reference is created, the name will default to the leaf name of the object that is being referenced.

Since hierarchical references have names, we can now address them within the module version. This is achieved for commands that accept hierarchical references as arguments by specifying the hierarchical reference name and a module context:

```
populate -modulecontext Chip ALU
```

The above command would populate the module referenced by the ALU hierarchical reference of module Chip. Clearly, there is a possible name clash here, as the module may contain objects or folders called “ALU” as well as the hierarchical reference called ALU. The hierarchical reference name will take precedence.

Note that wildcard matching *is supported* when specifying hierarchical reference names when module context is supplied. For example, the command:

```
populate -modulecontext Chip AL*
```

would match the ALU hierarchical reference.

## Creating a Module

The Create a new module dialog box is used to create a new module . You can create a module when:



- The "/Modules" node is selected on a server.
- A category node is selected on a server.
- A folder is selected in a your workspace.

**Note:** These types of folders cannot be used to create a module:

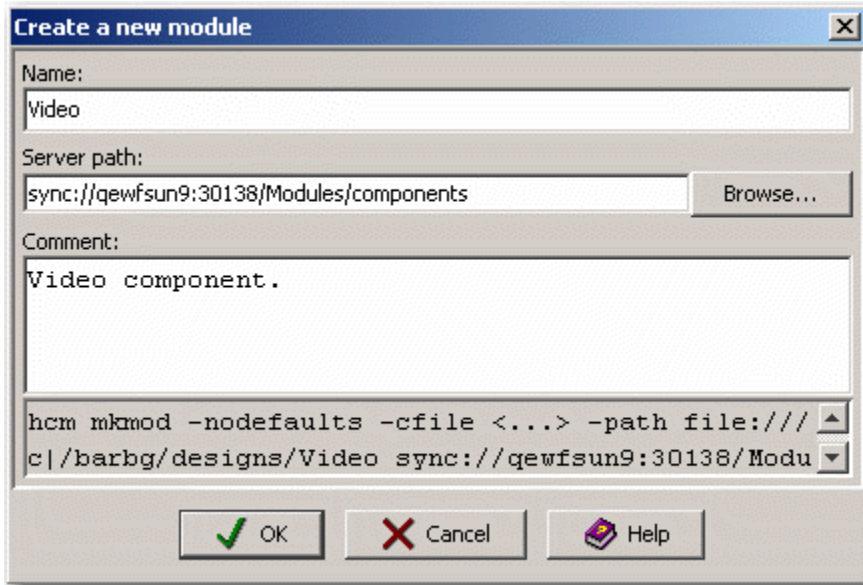
- A managed workspace folder
- A base folder of a legacy module

When a workspace folder is selected, that folder is initialized as the module's base directory. You can then add objects to the module (see Adding a Member to a Module) so that an initial checkin can be performed. See Checking In Design Data for more details.

### To create a module:

1. From the main menu, select **Modules =>  New Module** or select the  button from the Modules toolbar. You can also select **New Module** from the context menu on the Modules node or from a category node.
2. The Create a new module dialog box appears. Select options as needed.
3. Click **OK**.

**Click on the fields in the following illustration for information.**



### Name

The name of the new module. See URL Syntax for a list of illegal characters.

### Server path

The server URL and category path for the new module, in the form:

```
sync://<host>:[<port>]/Modules/[<category>]
```

The `<category>` is optional, and may be a path, such as `/Project1/mymods`. A module *category* is similar to a *path* to a DesignSync project or vault. The term category is used to indicate that this is a virtual path, rather than a physical path. And that its use is intended to categorize modules.

If a server node is selected, the value of the **Server path** is set to the URL of the selected object. As shown in the example, in order to specify a category path, you must add the category as part of the Module name.

If a workspace folder is selected, the Server path must be specified. You can type in a URL, or use the **Browse...** button to navigate to and select the Modules node on a server, or a category folder below the Modules node.

### Related Topics

ENOVIA Synchronicity Command Reference: mkmod

The Module Toolbar

Comment field

Command Invocation

Command Buttons

## Creating a New Version of a Module

A user creates a new version of a module by checking in their modifications to the module. Local modifications may include added data. See [Adding a Member to a Module](#) for details.

A new version of a module is also created when a user:

- Restructures the data within the module (see [Renaming a Module Member](#) for details)
- Removes data from the module (see [Removing a Member from a Module](#) for details)
- Adds a hierarchical reference to the module (see [Creating a Hierarchical Reference](#) for details)
- Removes a hierarchical reference from the module (see [Deleting a Hierarchical Reference](#) for details)

Branch locks are not removed upon checkin. For example, suppose that you lock the Trunk branch of the module Chip. (See [Locking Module Data](#) for details.) Later, you check in either the entire Chip module, or some portion of the Chip module. The branch lock remains in place, until it is removed by an unlock operation. (See [Unlocking Server Data](#) for details.) This enables you to perform multiple operations that modify the module, while retaining the branch lock on the module.

Any amount of data can be added to a module at once, resulting in a single checkin attempt. However, either the entire checkin will succeed, or the entire checkin will fail. I.e. Checkin of a module is an atomic operation. A new version of a module is only created if the entire checkin operation succeeds.

By default, after a failed atomic checkin, a subsequent checkin attempt will utilize information previously sent to the server. This optimizes the amount of data that needs to be transferred to the server, with the retry effectively continuing from where the previous try failed. For details, see [Checking In Design Data](#).

The exception is `mkmod -checkin`, which is a performance optimization for first time checkin of large amounts of data, and is not an atomic operation. In a `mkmod -checkin` operation, any files that fail to checkin are left as is, while those files that do checkin become part of the module. Any files which fail to checkin can be added to the workspace module, and checked in as the next module version.

## Adding a Member to a Module



The Add to Module dialog box adds highlighted objects to a local module in your workspace. The objects must be within the scope of the base directory of the target module and the module must be populated with a dynamic selector. The objects can be files, directories, or collection objects. The locally added objects are checked into the module version that is created by your next checkin. See [Checking In Design Data](#) for more information.

- If the highlighted objects are all files, and the module context can be uniquely determined, the files are added without invoking the Add to Module dialog box. These results are shown in the output window.
- If smart module detection cannot determine the target module, the Select Module Context dialog box is displayed. Once the module context is selected, the results are shown in the output window.
- If the highlighted object contains one or more folder objects, the Add to Module dialog displays.

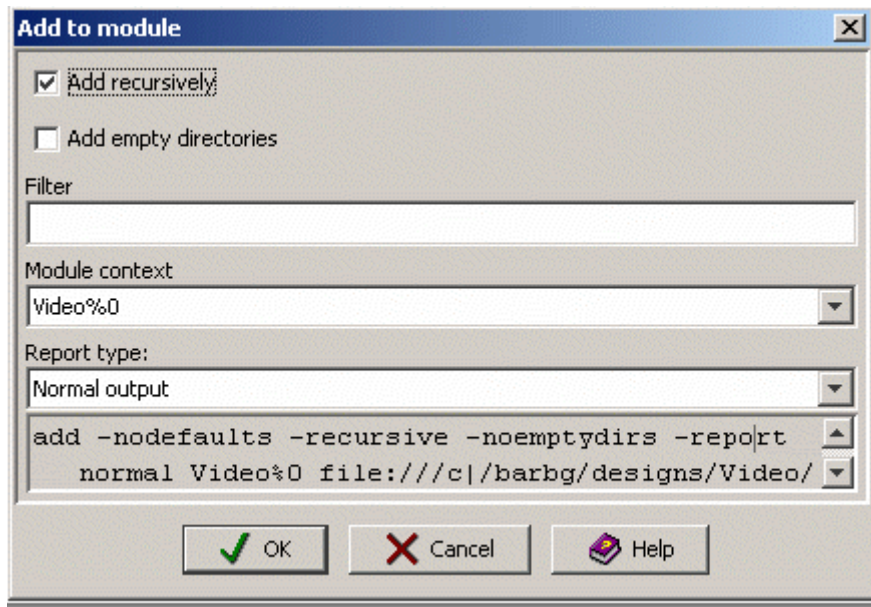
For information about how smart module detection determines the target module, see [Understanding Smart Module Detection](#).

Files excluded from view by exclude files are not displayed by the DesignSync GUI and are not available to Add from the GUI. For more information on exclude files, see [Working with Exclude Files](#).

### To add to a member with Add to Module dialog box:

1. From the main menu, select **Modules =>  Add Member** or you can select the  button from the Module Toolbar. You can also select this command from the context menu.
2. The Add to Member dialog box appears. Select options as needed.
3. Click **OK**.

**Click on the fields in the following illustration for information.**



### Add recursively

For a folder, whether to add only the folder or also recursively add the folder's contents. Note that folders themselves can be module members. If you checked the **Add empty directories** option, adding a folder without content results in an empty folder as a module member.

By default, the option to **Add recursively** is not selected.

### Add empty directories

This option is used with **Add recursively**. When adding members recursively, any folder that contains files is added to the module. The **Add empty directories** option specifies whether directories without content are added to the module.

For example, let's say you're recursively adding "dirA" to a module. "dirA" contains files, and an empty subdirectory "dirB". The option to **Add empty directories** controls whether the empty directory "dirA/dirB" is added.

### Filter

Allows you to include or exclude module objects by entering one or more extended glob-style expressions to identify an exact subset of module objects on which to perform the add.

The default for this field is empty.

### Module Context

Expanding the list-box shows the available workspace module instances for the currently selected object or objects, including an automatically calculated <Auto-detect> "module context.". All available workspace module instances are listed alphabetically in the pull-down following the calculated <Auto-detect>.

**Note:** If you select <Auto-detect>, and the DesignSync system cannot determine the appropriate module, the command fails with an appropriate error.

### Report type

From the pull-down, select the level of information you want to display in the output window:

**Brief output:** Lists errors generated when adding objects to the local module, and success/failure count.

**Normal output:** Lists all objects added to the local module, success/failure count, and beginning and ending messages for the add operation. This is the default output mode.

**Verbose output:** In addition to the information listed for the Normal output mode, lists:

- Skipped objects that are already members of the module.
- Skipped objects that are already members of a different module.
- Skipped objects that are filtered.
- Status messages as each folder is processed.

**Errors and Warnings only:** Lists errors generated when adding objects to the local module, and success/failure count.

### Related Topics

[Directory Versioning](#)

[ENOVIA Synchronicity Command Reference: add](#)

[Filter field](#)

[Module context field](#)

[Command Invocation](#)

[Command Buttons](#)

[Context Menu](#)



## Creating a Hierarchical Reference

The **Create a hierarchical reference** dialog box is used to create a new hierarchical reference from an upper-level module to any of the following objects:

- A module
- An external module
- A legacy module or legacy module configuration
- A DesignSync vault folder
- An IP Gear deliverable

You can create a hierarchical reference when a module version node (server) or 5.0 module base folder (client) is selected. When you create a hierarchical reference, a new module version is created.

### Notes:

You can not create a hierarchical reference from a legacy module or a legacy module configuration.



Hierarchical references to most targets are validated at the time the href is created. That means the following conditions must be met before you can create an hierarchical reference:

- The items you want must already exist. You may only create hierarchical references between objects that already exist.
- The servers that hosts the modules must be available.

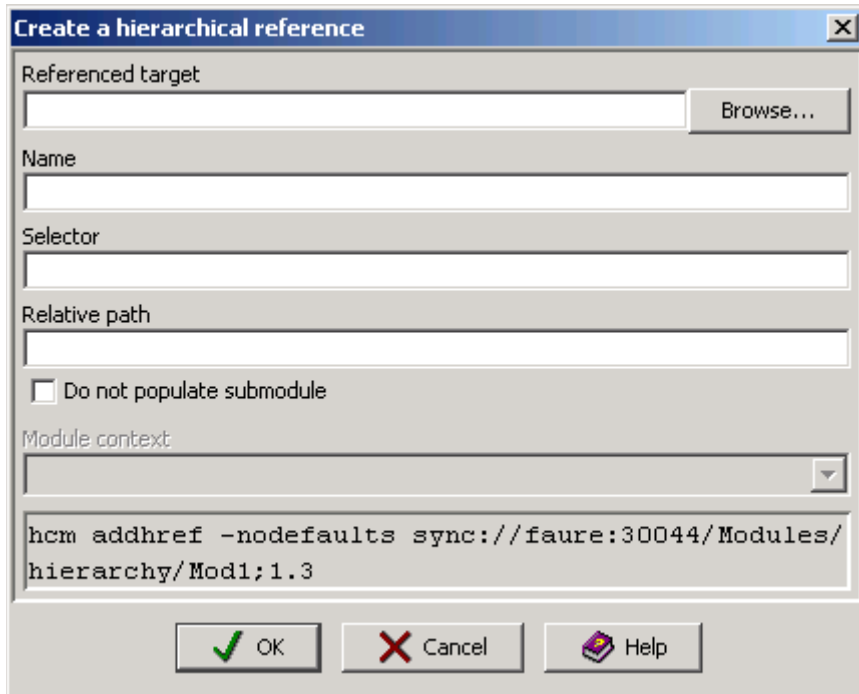
The exception to this rule is external modules. External modules are validated only during a populate operation. This means that neither the objects being referenced nor the server containing the objects need to be available at the time the href is created.

Hierarchical references in module snapshot versions can be changed after module snapshot creation. When a new hierarchical reference is added to a module snapshot, a new module version is created on the snapshot branch. For more information about module snapshots, see Module Snapshots.

### To create a hierarchical reference:

1. From the main menu, select **Modules =>**  **Add Href** or select the  button from the Module Toolbar.
2. The Create a hierarchical reference dialog box appears. Select options as needed.
3. Click **OK**.

Click on the fields in the following illustration for information.



## Field Descriptions

### Referenced target

Enter the referenced target or use the Browse button to select the module, IPGear deliverable, or vault folder to which you want to add a reference. Remember that:

- If the selected node is a server module version, the referenced target can not be a client module instance.
- If the selected node is a 5.0 module base folder, the reference target can not be a 5.0 base folder. The module instance must be specified. References must be modules and not folders.
- The referenced target can not be the base folder of a legacy module.
- You can not use the Browse button to find IPGear deliverable URLs or external modules URLs. You must manually enter the URL in the Referenced Target field.

The format for entering the URL manually is:

- Module (specified by server)

```
sync[s]://<host>[:<port>]/Modules/[<Category>/...]<Module_Name>
```

<host> is the SyncServer on which the module resides.

<port> is the SyncServer port number.

Modules is the container for modules on the server.

*<Category>* is the optional category path to the module.

*<Module\_Name>* is the name of the module.

- Module or legacy module configuration (specified by workspace instance)

*<Module\_Name>%<Instance\_num>*

*<Module\_Name>* is the name of the module.

*<Instance\_num>* is the instance number assigned to the module.

- External module

*sync[s]://ExternalModule/<external\_type>/<external\_data>*

ExternalModule is a constant that identifies this URL as an external module URL.

*<external-type>* is the name of the external module procedure.

*<external-data>* contains the parameters and options to pass to the procedure.

- Legacy module configuration or DesignSync vault (specified by server)

*sync[s]://<host>[:<port>]/<vaultPath>*

*<host>* is the SyncServer on which the module resides.

*<port>* is the SyncServer port number.

*<vault\_Path>* identifies the path to the legacy module configuration or DesignSync vault.

- IP Gear deliverable

*sync[s]://<host>[:<port>]/Deliverable/<ID>*

*<host>* is the SyncServer on which the module resides.

*<port>* is the SyncServer port number.

Deliverable is a constant that indicates this is an IP Gear reference.

<ID> is the deliverable ID number.

By default, this field is empty.

### Additional Notes

- If the “Referenced Target” is an external module, IPGear deliverable, or vault directory, the selector field must remain empty.
- You can not specify a reference target to a legacy module configuration by specifying its vault URL and a selector.

### Name

This field needs a legal value entered.

If you have already selected a referenced target, the last segment of the URL becomes the default name .

If you select a node that is a version 5.0 module base folder and the referenced target is that of a workspace module identifier which includes an instance number, the module instance name is stripped off the hierarchical reference name.

There are some characters that are reserved for DesignSync and cannot be used in the name of an href. For the list of reserved characters, see URL Syntax: Reserved characters.

### Selector

Name of the selector for the referenced target. Depending on what you have entered in other fields, this field may be pre-filled. For example:

- When the object in the Referenced Target field is a server module, the selector field is set to `Trunk:Latest`.
- When the object in the Referenced Target field is a server module branch or version, the selector is set to match. For example, if the version was selected from a node is described by a tag, the selector uses the tag name and branches such as `tagname:Latest`.
- When the object in the Referenced Target field is a server legacy module, the selector is set to `<default>`.
- When the object in the Referenced Target field is a server legacy module configuration, the selector is set to the configuration name.
- When the selected node is a version 5.0 module base folder and the referenced target is a client module instance, the selector field is set to the selector of the referenced module.

**Caution:** If you manually change the value in this field, the value is not replaced afterwards by information generated by changes in other fields.

Note: External modules, legacy modules, and IP Gear references do not take a selector value.

### Relative Path

This field is optional.

Depending on what you have entered in other fields, this field may be pre-filled. Captures the relative path from the module to the sub-module when the module hierarchy is fetched.

- When the object in the referenced target field is server module version, the leaf name of the target is entered into relative path field
- When the selector field contains the 5.0 module base folder and the referenced target field is a workspace module identifier, the relative path between the two base directories is calculated by the system.

### Do not populate submodule

When checked, the submodule is not included in a recursive populate of the parent module. When not checked, a recursive populate of the parent module does include the sub-module. The default is not checked.

### Module Context

The field is only enabled for module base folders and contains the module instance names of the modules based at that folder. These module instance names are listed alphabetically in the pull-down.

### Related Topics

The Module Toolbar

ENOVIA Synchronicity Command Reference: [addhref](#)

ENOVIA Synchronicity Command Reference: [hcm\\_release](#)

ENOVIA Synchronicity Command Reference: [rmhref](#)

ENOVIA Synchronicity Command Reference: [showhrefs](#)

ENOVIA Synchronicity Command Reference: [showstatus](#)

## Removing a Member from a Module



The Remove from module dialog box is used to remove the selected module members from a module. You may commit the change, as well as any other module changes, to the module at the next module checkin, or you can immediately create a new module version without the selected module member. You can use this dialog box when:

- Current module members (files/folders) are selected in the client work area.
- Current module members (files/folders) are selected on the server and all members belong to the same module version, which must be the latest version on a server module branch.
- Current module base folders are selected in the client work area when the folder is also a regular module folder of another module.
- The module in the workspace was populated in dynamic mode and changes can be checked in.

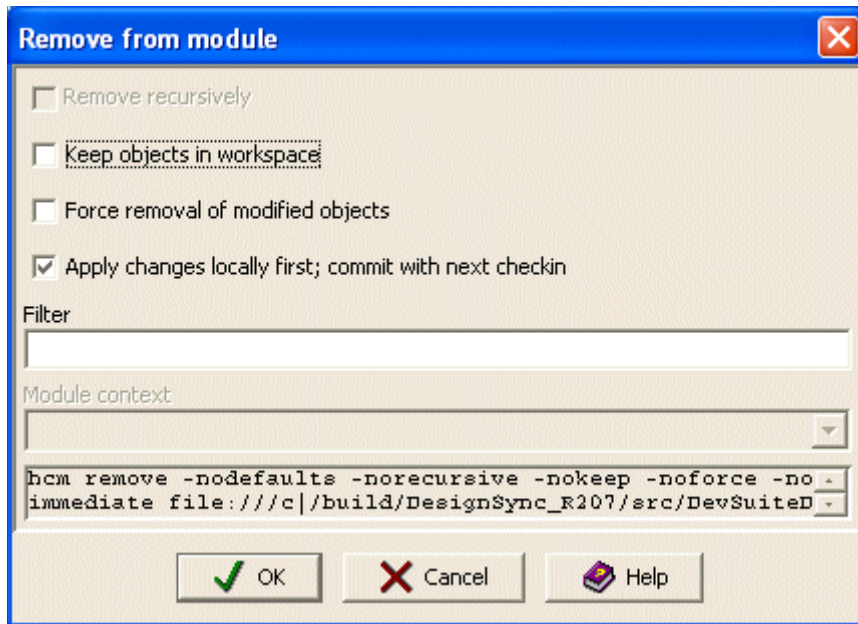
### Note

Removing module members/folders that have not been fetched into the workspace is not supported by the GUI.

### To remove a member from a module:

1. Select the member(s).
2. From the main menu, select **Modules =>**  **Remove Member** or select the  button from the Module Toolbar.
3. The Remove from module dialog box appears. Select options as needed.
4. Click **OK**.

**Click on the fields in the following illustration for information.**



### Remove Recursively

This option is not available when a server object is selected

When checked, the selected folder is removed as well as all module objects in the folder and all subfolders. When not checked, the selected folder is removed only if the folder is empty. The default for this option is unchecked.

### Keep objects in workspace

This option is not available when remove is done directly on a server object.

When checked, the local copies of the removed objects are left in the workspace as unmanaged objects. When unchecked, the local copies of the selected objects become DesignSync references, with the Status "Locally Removed". The default for this option is unchecked.

### Force removal of modified objects

This option is not available when remove is done directly on a server object

When checked, the selected object is removed even if the object in the workspace is not the same as the last checked in version of the object or is locked. If you are removing objects that were added to a module, but never checked in, you choose this option to remove the objects.

When unchecked, the selected object is not removed if it is locked or not identical to the last checked in version or added to the module but never checked in.

The default for this option is unchecked.

### **Apply changes locally first; commit with next checkin**

This option is not available when remove is done directly on a server object

When checked, the selected object(s) is marked for removal during the next module check in. Unless the `Keep_Objects_in_Workspace` option is selected, the object remains in the workspace as a DesignSync reference, with the Status "Locally Removed". The object remains on the server until the change is committed during the next checkin. (Default)

When not checked, DesignSync immediately creates a new module version on the server with the selected objects removed.

**Note:** Objects in the Add state are always immediately removed from the Add state. No new module version is created.

### **Filter**

Allows you to include or exclude module objects by entering one or more extended glob-style expressions to identify an exact subset of module objects on which to perform the remove. For more information, see Filter Field.

The default for this field is empty.

### **Module Context**

This option is only available when the selection set includes one or more client side folders. You can select from the available module instances. The choices are listed in alphabetical order. For more information, see Module Context Field.

The default for this field is empty.

### **Related Topics**

The Module Toolbar

Filter Field

ENOVIA Synchronicity Command Reference: remove

## **Deleting a Hierarchical Reference**

You can delete a hierarchical reference (href) from either from your local work area or from the server.



Hierarchical references in module snapshot versions can be removed after module snapshot creation. When a hierarchical reference is removed from a module snapshot, a new module version is created on the snapshot branch. For more information about module snapshots, see [Module Snapshots](#).

**Note:** If you are removing a hierarchical reference from the workspace, the workspace must be populated with a dynamic module version.

#### To delete a hierarchical reference:

1. Highlight the href you want to delete in either the List View pane or the Tree view.
2. Select **File =>Delete**, or press the **Delete** key, to bring up the Delete dialog. On the server, you can also select a server module, right-click and select **Delete** from the context menu.
3. There are no options to select from the Delete dialog box for a href. Click **OK** to delete the file. You are prompted to confirm the deletion.

**Important:** If you remove the hierarchical reference

in your workspace, you must repopulate the workspace to see the change.

#### Related Topics

[Deleting Design Data](#)

[ENOVIA Synchronicity Command Reference: addhref command](#)

[ENOVIA Synchronicity Command Reference: rmhref command](#)

## Locking Module Data



The Lock module branch dialog box is used to lock a module version on a specified module branch. Locking the branch is useful when creating new branches, creating or deleting tags, or adding to, deleting from, or copying a module's metadata. By locking the module version, you insure that no one else can alter the data while you make your changes.

The Lock module dialog box is available when a module-base directory is selected in the client work area or a module branch is selected on the server.

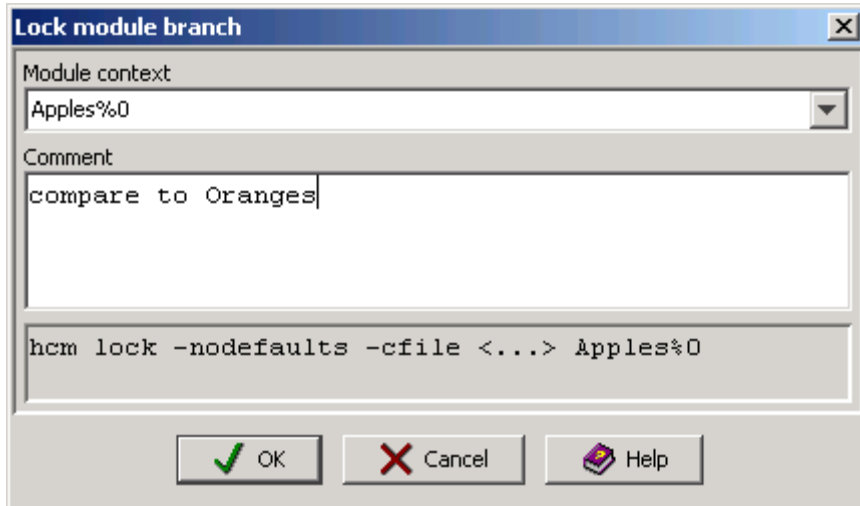
#### Notes:

- You can not lock a module version on the server.
- If a module branch contains members that are themselves locked, then you can only lock that branch if you are the lock owner for all those locked members.

### To lock a module branch:

1. Highlight the module version or module-base directory you want to lock.
2. From the main menu, select **Modules =>**  **Lock branch**. You can also select  **Lock branch** from the context menu.
3. The Lock module branch dialog box appears. Select options as needed.
4. Click **OK**.

Click on the fields in the following illustration for information.



### Field Descriptions

#### Module context

Expanding the list-box shows the available module instances for the currently highlighted base directory. All available module instances are listed alphabetically in the pull-down. **Note:** There may only be one module listed.

#### Comment

Enter the comment information you want other users to know about this locked branch. The default entry is blank. You can use context menu commands to cut, copy, paste, select all, previous comments, and open your default text editor.

### Animated Examples

- Locking a module branch
- Locking module content


## Setting a Workspace Root

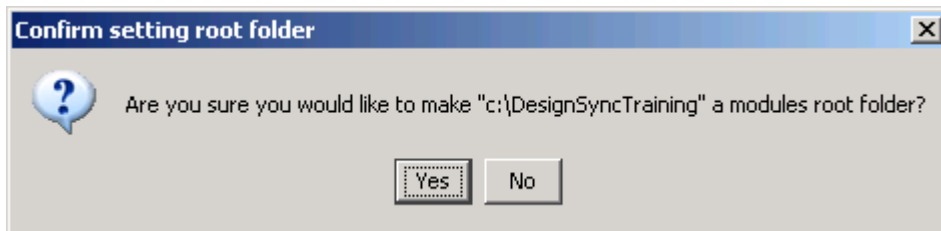
This command allows you to designate the workspace directory to be used as a storage area for a set of local metadata information for a collection of design objects. The metadata includes information about the design files. Setting a workspace root does not create DesignSync object metadata. When you create a module, or checkout or populate files or module based objects, DesignSync stores (or creates) the appropriate metadata for those design objects and stores the metadata in the workspace root folder.

For modules, after the workspace is created/populated with module data and the workspace root set, you can refer to a module by the module instance name, rather than specifying the full module path name.

**Note:** You cannot set a workspace root directory underneath an existing workspace root directory.

#### To set a workspace root:

1. Highlight the folder that you want to set as a workspace root.
2. From the main menu, select **Revision Control=>**  **Set Root Folder**. A confirmation notice similar to this appears:



3. Click **Yes** to confirm and **No** to cancel.

**Note:** When design data is created or populated with a specified workspace path, the parent of that workspace path is automatically set as the root if there is not root already set.

#### Related Topics

ENOVIA Synchronicity Command Reference: mkmod

ENOVIA Synchronicity Command Reference Help: populate

ENOVIA Synchronicity Command Reference Help: co

ENOVIA Synchronicity Command Reference Help: setvault

## Rolling Back a Module

If module changes are made that are not appropriate or correct, you can "roll back" the module to the last good version. Module rollback restores a previous module version by

creating a new module version containing the identical information as the rollback version. All versions between the rollback versions are retained in the server vault.

When a module is rolled back, the module version increments, but all module members contained in the module version return to the versions contained in the original module version. When a modified version of a module member is checked in, DesignSync allocates an appropriate version number such as the next number available sequentially or a branched version number. This maintains the integrity of each checked in module member version and allows any module version to be restored at any time.

The rollback reinstates the module exactly as it was, removing any structural changes to the module. The following table lists the type of structural changes possible, and the result of those changes when rolling back the module version to before the change was made.

Type of structural change	Rollback operation
Removed file or collection	File or collection is added back to the module as a module member.
Removed folder	<p>If the folder was explicitly added in the module version being rolled back to, and was removed in a later version, then the folder is restored to an explicitly added state.</p> <p>If the folder was implicitly present in the rollback version due to the presence of members within it, and in a later version was either explicitly added or was removed (by removing all the members within it), then the folder is restored to the implicitly added state because it contains module members.</p>
Renamed or moved file or collection	The file or collection reverts to the name, location, and version of the object active in the rollback version.
Added file or collection	File or collection is removed from the module. To add the object back to the module, use checkin with the Allow Checkin of New Items and Allow Version Skipping ( <code>ci -new -skip</code> ) options selected.
Removed hierarchical reference	<p>The hierarchical reference is restored to the module.</p> <p><b>Note:</b> DesignSync performs no consistency checking to verify that the</p>

	reference is still valid.
Added hierarchical reference	The hierarchical reference is removed from the module. You can recreate the reference. For information on creating hierarchical references, see <a href="#">Creating a Hierarchical Reference</a> .

## Related Topics

ENOVIA Synchronicity Command Reference Help: rollback command

## Deleting a Module

You can delete a module from either from your local work area or from the server. You cannot delete a legacy module in the workspace, however, you can delete the module base directory for that module. A module cache can only be deleted from the workspace. Deleting the module on the server does not remove the corresponding mcache or any links to the mcache.

Deleting a module or module cache in your workspace will only remove the module from the workspace. It does not affect the server. If you delete a module from the server it does not affect your work area. If the module was fetched into a work area, and the module it was fetched from was deleted, the files remain in your work area.

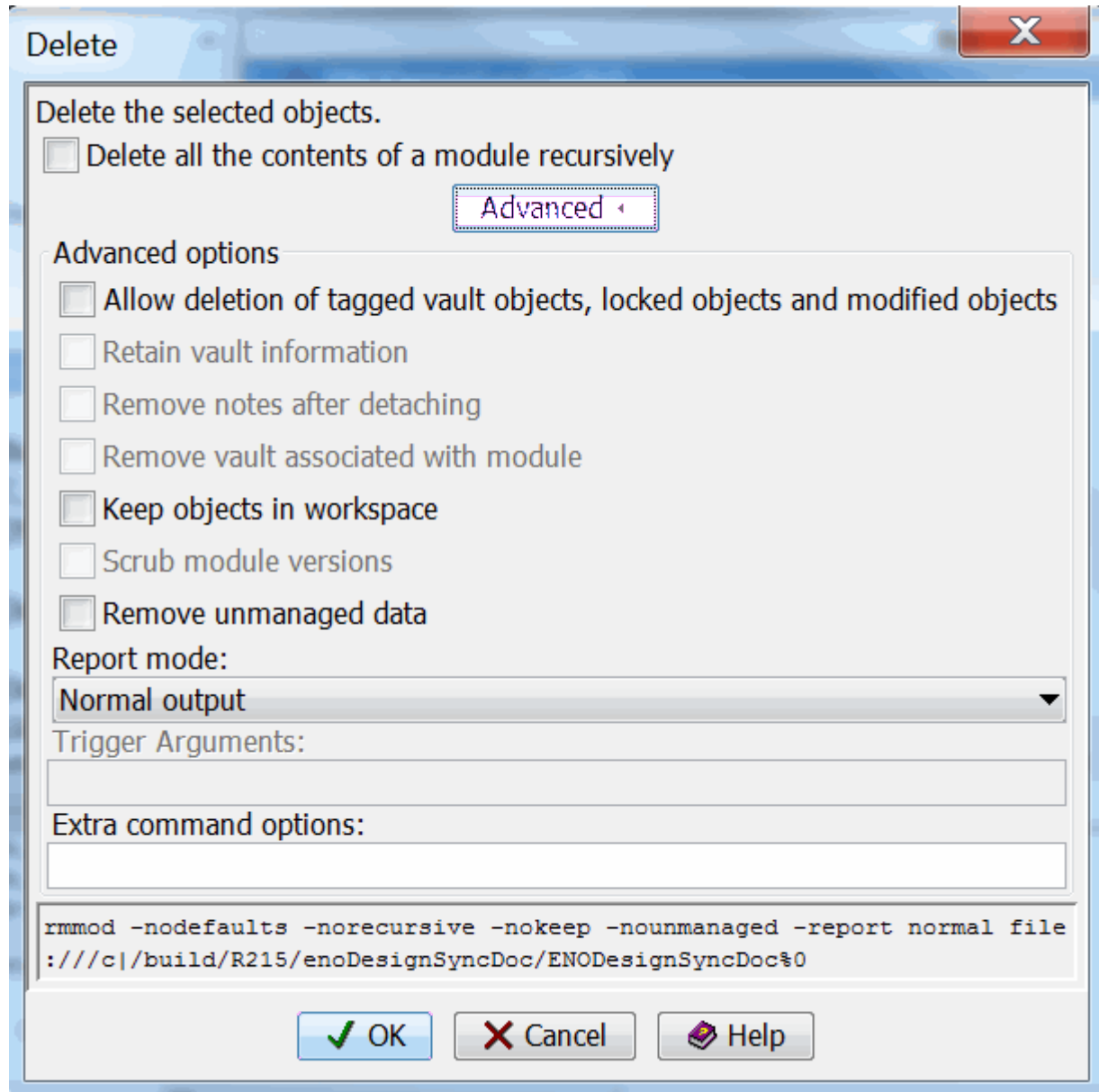
## Notes

- The deletion of a module on the server is not recursive; it does not follow the hierarchical references of the module and delete the referenced modules. To remove an entire module hierarchy, you must remove each module separately. You can delete an entire module hierarchy in your workspace.
- The deletion of a module does not remove hierarchical references to the deleted module and for pre-V5.0 releases configurations of the deleted module.
- If you delete a legacy module without removing the vault and the module resides in the `/Projects` folder, the module and its configurations will still appear in ProjectSync as a ProjectSync project.

## To delete a module:

1. Select the module you want to delete in the List View pane or the Tree view. To delete a workspace module, select the module instance.
2. Select **File =>Delete**, or press the **Delete** key, to bring up the Delete dialog. On the server, you can also select a server module, right-click and select **Delete** from the context menu.
3. Select options as needed.
4. Click **OK** to delete the module. You are prompted to confirm the deletion.

Click on the fields in the following illustration for information on each field.



### Field and Option Descriptions

#### Delete the selected objects/Delete all the contents of a module recursively

Delete the specified module and all modules in the hierarchy beneath it. This is a workspace only operation. You cannot delete the module hierarchy from the server.

#### Allow deletion of tagged vault object, locked objects and modified objects

This option is only available when deleting a module from your workspace and indicates whether locked or modified objects can be deleted from the workspace. It is not applicable to module caches.

If this option is not specified, and there are locked or modified objects in the workspace, then such objects will be left and the module itself will not be deleted from the workspace. The unlocked/unmodified objects will still be deleted from the workspace.

### **Retain vault information**

This option applies to legacy modules on the server. You cannot remove a legacy module from the workspace. Specifies whether the vault folder in which the module contents reside should be removed.

### **Remove notes after detaching**

This option is only available when deleting server legacy modules or server modules.

If checked, deletes all the notes that were attached to the deleted module if the notes have no ties to any other live projects such as in ProjectSync.

### **Remove vault associated with module**

This option is not applicable to module deletion.

### **Keep objects in workspace**

Specifies whether to keep the module member data in the workspace after the module has been removed. This option is only applicable when deleting a workspace module.

When this option is not selected, DesignSync removes all module data, including module members and DesignSync metadata, from the workspace. (Default)

When this option is selected, DesignSync removes only the module metadata; it does not remove the module members. If the module members are links from a cache (populated in -share mode), the server copies the files locally and removes the links.

This option is mutually exclusive with Remove unmanaged data.

### **Scrub module versions**

This option is only applicable when deleting a module version on the server.

### **Remove unmanaged data**

Specifies whether the operation should remove or retain any unmanaged files within the workspace module directory structure.

When this option is not selected, DesignSync does not attempt to remove any unmanaged data and any folders containing unmanaged data remain after the operation completes. (Default)

When this option is selected, DesignSync attempts to remove any unmanaged data within the module directory structure after the module is removed. If the folder(s) containing unmanaged data is not removed (for example, the folder also belongs to another module, or the target module is not removed), then the unmanaged data contained within the folder is also not removed..

**IMPORTANT:** The module delete operation does not list the removed unmanaged objects nor does it include them in the success/failure count. Any folders that become empty as a result of removing unmanaged objects are also silently removed.

This option is only valid for workspace module arguments and is mutually exclusive with Keep objects in workspace.

### Report mode

**For the Report mode, choose the level of information to be reported:**

- **Brief output/Errors and Warnings only:** Brief and Errors and Warnings mode reports the following information:
  - Failure messages.
  - Warning messages.
  - Success/failure status.
- **Normal output:** In addition to the information reported in Brief:
  - The path of each module removed..
- **Verbose output:** In addition to the information reported in **Normal** mode:
  - Information status messages for each stage of module removal.

### Trigger Arguments

See Trigger Arguments.

### Extra command options

List of command line options to pass to the external module change management system. Any options specified with the Extra Command options field are sent verbatim,



with no processing by the populate command, to the Tcl script that defines the external module change management system. For more information on external modules, see External Modules.

## Related Topics

Deleting a Module Cache Link

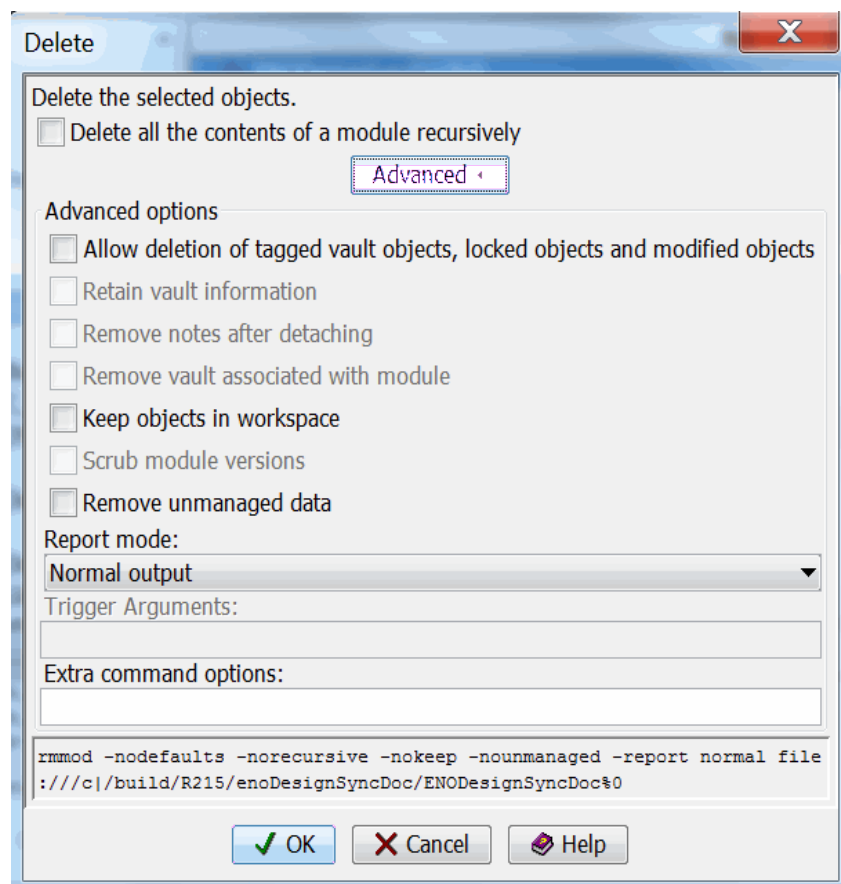
ENOVIA Synchronicity Command Reference: rmmmod

## Deleting a Module Cache Link

You can delete mcache links from your local work area. This does not affect the module cache itself, or the module on the server.

Select the module cache link you want to delete in the Tree View or List View panes. Select **File => Delete**, or press the **Delete** key, to bring up the **Delete** dialog. Click **OK** to delete the folder containing the module cache links. You are prompted to confirm the deletion.

Click on the fields in the following illustration for information.



## Delete Field Descriptions

### Delete the selected objects/Delete all the contents of a module recursively

This field is not applicable to mcache links. If selected, it is silently ignored.

### Module Context

This field is not applicable for mcache links. The module context is implicit.

### Allow deletion of tagged vault objects, locked objects, and modified objects

This field is not applicable to deletion of module cache links. If selected, it is silently ignored.

### Retain vault information

The option to **Retain vault information** is not applicable to the deletion of module cache links.

### Remove notes after detaching

This field is not applicable to module cache links.

### Remove vault associated with module

This field is not applicable to module cache links.

### Keep objects in workspace

This field is not applicable to module cache links.

### Scrub module versions

This option is only applicable when deleting a module version on the server.

### Remove unmanaged data

This option is not applicable to mcache links.

### Report mode

**For the Report mode, choose the level of information to be reported:**

- **Brief output/Errors and Warnings only:** Brief and Errors and Warnings mode reports the following information:

- Failure messages.
  - Warning messages.
  - Success/failure status.
- **Normal output:** In addition to the information reported in Brief:
    - The path of each module removed..
  - **Verbose output:** In addition to the information reported in **Normal** mode:
    - Information status messages for each stage of module removal.

## Trigger Arguments

See Trigger Arguments.

## Extra command options

This option is not applicable to module cache data.

## Related Topics

Deleting Design Data

Using a Module Cache

ENOVIA Synchronicity Command Reference: rmfolder

## Resolving Module Structure Conflicts

When the structure of a module has changed both on the server and locally, this can result in a structure conflict. For more information on recognizing when you have a structure conflict, see Conflict Handling.

### To Resolve a Structure Conflict:

1. Choose which version is correct.
2. If the structure in the workspace is correct, and you have content changes as well, copy the content changes to a temporary directory. If the structure on the server is correct, you can change the member in the workspace to match the server version to resolve the conflict. A merge may be required to merge the content.
3. Populate the workspace using the Replace mode: Force Overwrite of modified objects. (`populate -force`).
4. If you copied changes to a temporary location, move them back into your workspace.
5. Make any additional changes you want to check in.

6. Check in your changes normally.

## Related Topics

Conflict Handling

Populating Your Work Area

ENOVIA Synchronicity Command Reference: populate command

## Overlaying Module Data

You may want to overlay member files from a module version that is a different module version than the one already in your workspace.

Overlaying a module version will:

- Fetch the member files of the requested module version
- Replace the equivalent member files in your workspace
- Retain the current fetched version information of the member files

The status of the workspace member files that were overlaid will be "locally modified".

Note that merging is not involved. There are no conflicts, because conflicts only pertain to merging. This is also different than creating a composite workspace with members from different module version, as is done with Module Member Tags.

You can overlay across branches:

```
populate -overlay <branch>:Latest
```

The **populate** command above will fetch objects from the specified <branch>, and overlay them onto the equivalent objects in the workspace.

### How hrefs are handled

Any hrefs on the branch version that do not exactly match an href in the workspace version will be reported as a failure. No attempt to overlay hrefs is made. Details of a failed href are reported. From the output, you can easily determine the **addhref** and **rmhref** commands needed, if you want the href from the branch version.

Similarly, an href on the branch version that does not exist in the workspace version is reported as a failure, with details of the href output.

Note that hrefs cannot simply be changed or added to a workspace, because any href change results in a new module version. (See [Creating a New Version of a Module](#) for details.) `Populate` fetches data from the vault – it does not create new vault data.

## Examples

The examples below describe several cross-branch overlay scenarios. As background, you will need to understand natural paths and unique identifiers.

Object present on branch and workspace, with the same natural path

Object present on branch and workspace, with different natural paths

Object present on branch but not in workspace version

### **Object present on branch and workspace, with the same natural path**

In this case, a unique identifier exists in the branch version and the workspace version, with the same natural path.

The branch version object is fetched into the workspace, replacing the object already in the workspace. The fetched version number is that of the workspace module version. The status of the object in the workspace will be "locally modified".

However, if there is an unmanaged object in the workspace, or an object from another module, that is using the same natural path, then the fetch will fail.

### **Object present on branch and workspace, with different natural paths**

In this case, a unique identifier exists in the branch version and the workspace version, with different natural paths.

The branch version object is fetched into the workspace, using the workspace version's path. Note that overlaying an object does not change the object's natural path. The overlay applies to the file contents.

The output from the **populate** command will include the workspace path used for the fetch.

### **Object present on branch but not in workspace version**

In this case, a unique identifier exists in the branch version that is not in the workspace version.

The branch version object is fetched into the workspace, and added to the module, so that it is a candidate for the next check-in. The **ls** command will report the object as "Added".

However, if there is an unmanaged object, or an object from a different module, at the natural path in the workspace, then the fetch will fail.

If the workspace module version contains a different object at the same natural path, (i.e. there is a different unique identifier in the workspace version that has the same natural path), then the branch version object will still replace the workspace object. The unique identifier of the workspace object is retained. This results in the contents of one unique identifier replacing the contents of a different unique identifier.

It is possible for the situations above to occur during the same populate. For example, suppose all three of these conditions are met:

- The workspace version contains an object with unique identifier 1111 at natural path file1.txt
- The branch version contains the object with unique identifier 1111 at natural path file2.txt
- The branch version contains the object with unique identifier 2222 at natural path file1.txt

Under the previous rules, both of the branch version objects would be brought to the workspace at natural path file1.txt. The first object fetched causes the workspace object to be locally modified.

If the `-force` option to the `populate` command is used, both branch version objects will be fetched, with the object that was fetched second replacing the (locally modified) object that was fetched first.

If the `-force` option is not used, then the object that is fetched first will remain in the workspace. The second fetch attempt will fail, because the workspace object is locally modified.

### Related Topics

[Merging Module Data](#)

[ENOVIA Synchronicity Command Reference: populate](#)

[ENOVIA Synchronicity Command Reference: addhref](#)

[ENOVIA Synchronicity Command Reference: rmhref](#)

[ENOVIA Synchronicity Command Reference: ls](#)

## Synchronizing Enterprise Developments

When objects that are part of an enterprise system are created, modified, or removed in DesignSync, you can manually synchronize those changes with the Enterprise system. DesignSync auto provides the ability to automatically synchronize Enterprise Developments. For more information about configuring automatic synchronization, see the *DesignSync Administrator's Guide: Site Options*.

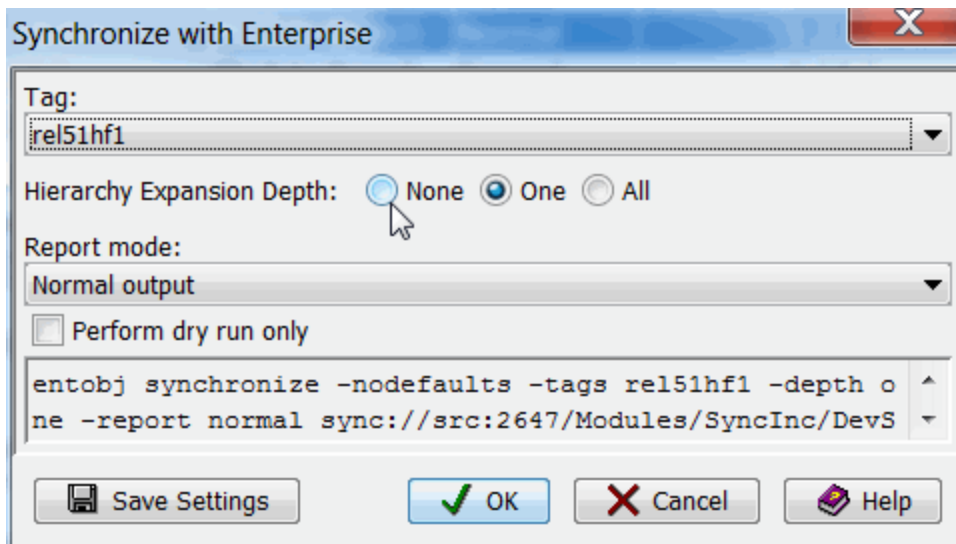
The Synchronize action is available when anyone of the following objects is highlighted on the server:

- Module branch object on the server
- Module version object on the server

### Running the Synchronize command

**Modules =>Enterprise =>Synchronize** opens the **Synchronize** dialog box. Click **OK** to launch the action.

Click on the fields in the following illustration for information.



#### Tag

Dropdown list of the tags associated with the selected object. If there is only one tag, only one tag appears in the list. The first tag in the list is selected by default. .

#### Hierarchy Expansion Depth

Indicates how many levels of the module hierarchy to send to the remote server hosting the associated Enterprise Design system:

- **None** - Synchronizes only the specified module version; does not synchronize any hierarchical references. This may result in an incomplete hierarchy on the server.
- **One** - Synchronizes the first level of the module hierarchy, but do not traverse the hierarchy. This option minimizes the risk of an out-of-date hierarchy adding the potential performance impact of updating the entire hierarchy. (Default)
- **All** - Synchronizes recursively through the entire hierarchy for each module version. This option provides the most complete update information to the server, but can be performance intensive. Hierarchical references to non-module objects are considered "leaf" objects and DesignSync does not attempt to continue traversal through that object or configuration.

### Report mode

For the **Report mode**, choose the level of information to be reported:

- **Brief output** - outputs the status of the running command, command results, and errors.
- **Normal output** - outputs the information contained in the Brief output and information about the enterprise versions being created. (Default),
- **Verbose output** - There is currently no difference between the verbose and normal reports.

**Note:** The report information is also passed to the ENOVIA server.

### Perform dry run only

Select this option to indicate that DesignSync is to treat the operation as a trial run without actually checking in design objects.

This option helps detect whether there are problems that might prevent the synchronization from succeeding.

### See Also

*Enterprise DesignSync Administration User's Guide.*

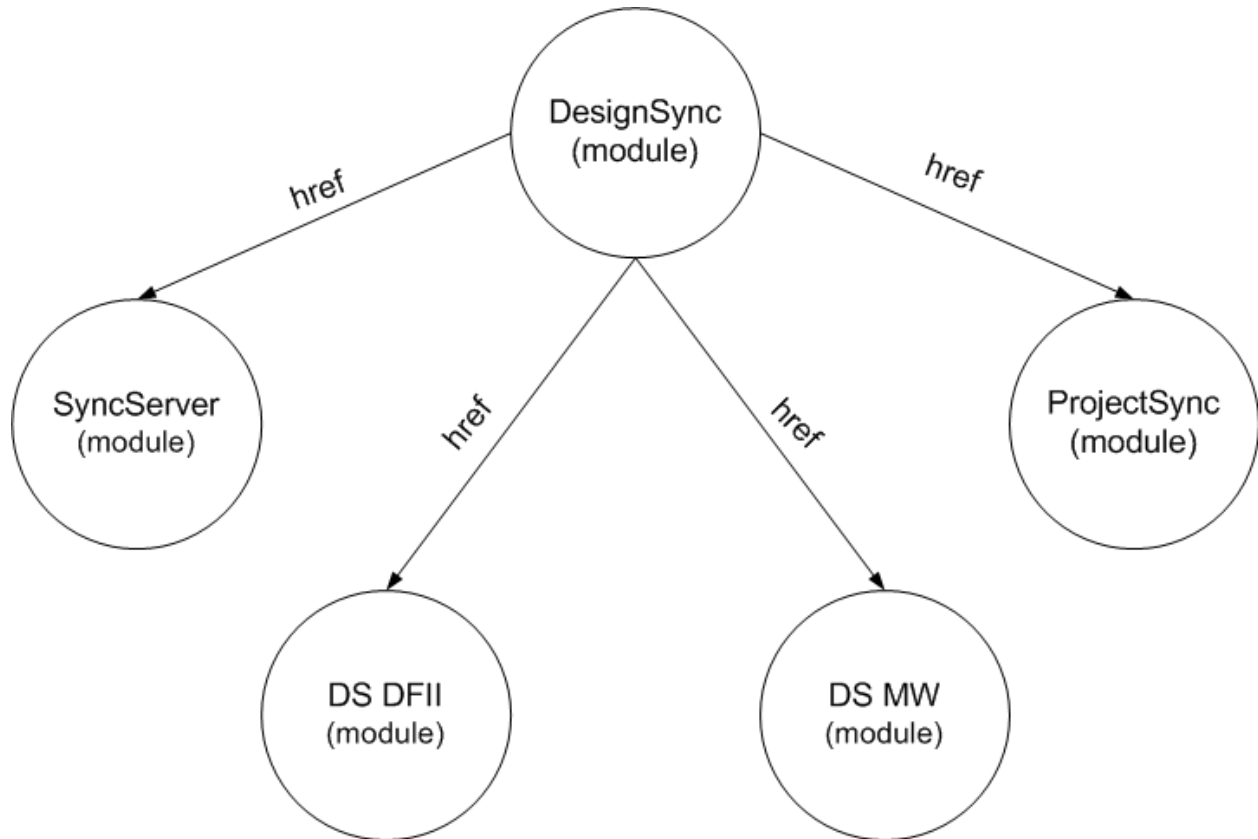
## Reference

### What Is a Module?

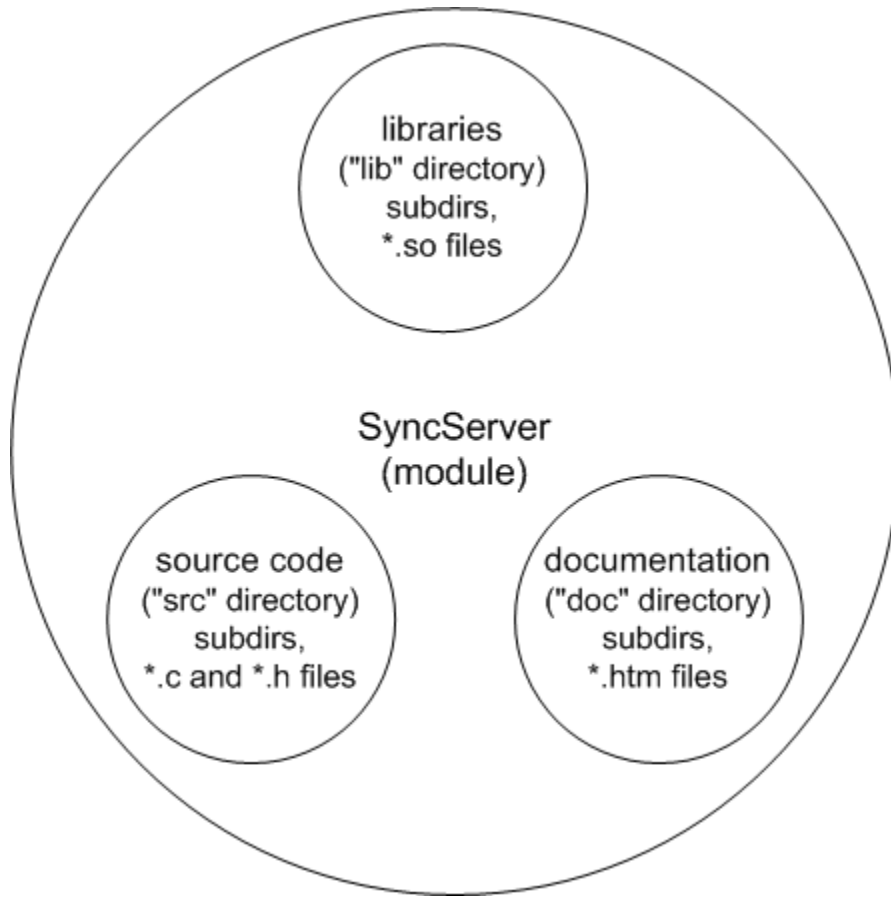
Modules can be thought of as building blocks of a project. A project can be broken down into related chunks of work. Each chunk of work can be any size and of any kind of data. The project is formed by connecting the chunks of work together. Each chunk of work is a *module*. Modules are connected by hierarchical references (*hrefs*).



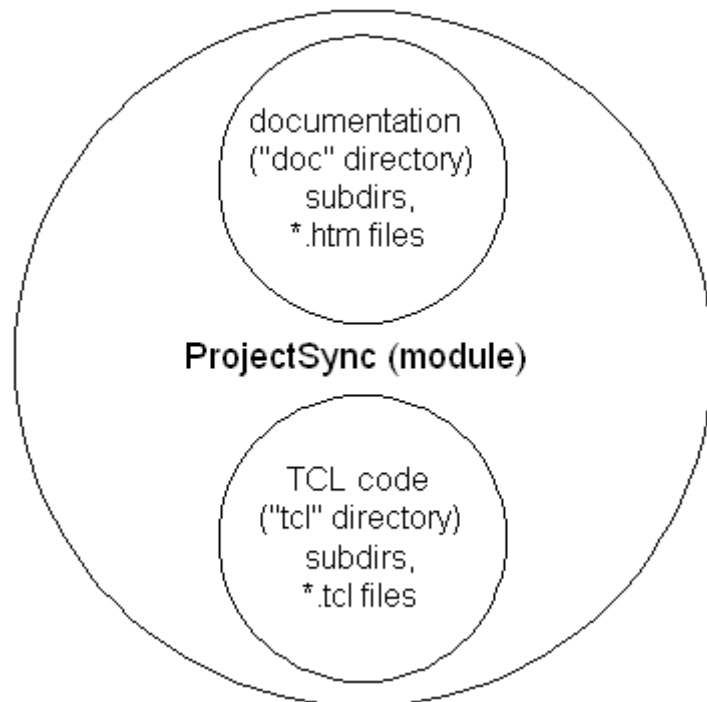
Let's take DesignSync itself as an example. If each of the products within DesignSync has its own dedicated team of people, the DesignSync lead creates DesignSync as a top-level module, consisting only of hierarchical references to submodules for each of the products.



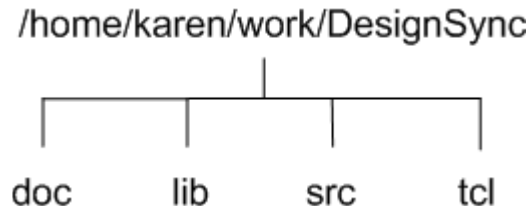
The product leads of the products then define the module hierarchy for their products. For example, the SyncServer module might contain subdirectories for libraries, source code, and documentation.



The ProjectSync module also has code and documentation subdirectories:



Let's say the documentation team is a shared resource for all products, with a few of the writers working on both SyncServer and ProjectSync. A writer might fetch both the DesignSync and ProjectSync modules into the same workspace, `/home/karen/work/DesignSync`, as shown in the example below. Note that top-level "DesignSync" and "ProjectSync" directories are not created. The data within those modules is fetched, resulting in this workspace structure:



Although not shown in the above diagram, each of the directories contains subdirectories.

Since both the SyncServer and ProjectSync modules contain `doc` subdirectories, the `doc` workspace directories will contain data from both of the DesignSync and ProjectSync modules. These are referred to as *overlapping modules*, because they share a common base directory.

The doc writer did not need to fetch all of the content of the SyncServer and ProjectSync modules. She could have fetched only data pertaining to the documentation. See Filtering Module Data and Recursion.

Users define the content of a module and how modules are structured (see Module Hierarchy). Every item within a module is a member of that module (*a module member*). Users work with modules and their data by using DesignSync commands. See Operating on Module Data.

DesignSync automatically manages modules by creating a new version of a module every time the module is modified.

A module is modified when a user:

- Checks in data that was added to the module (using the **add** command followed by the **ci** command)
- Restructures the data within the module (by using the **mvmember** command)
- Removes data from the module (using the **remove** command)
- Adds a hierarchical reference to the module (using the **addhref** command)
- Removes a hierarchical reference from the module (using the **rmhref** command)

The topic Data Management of Modules explains how module data is managed.

Directories, not only their content, are part of a module's data (module members). Consequently, DesignSync automatically manages directories belonging to a module. See [Directory Versioning](#).

### Atomic Checkin and Recovery

You can add any amount of data to a module at once, resulting in a single check-in attempt. However, either the entire checkin succeeds, or the entire checkin fails; checkin of a module is an atomic operation. A new version of a module is created only if the entire check-in operation succeeds.

By default, after a failed atomic checkin, a subsequent check-in attempt utilizes information previously sent to the server. This optimizes the amount of data that needs to be transferred to the server, with the retry effectively continuing from where the previous try failed. For details, see [ENOVIA Synchronicity Command Reference: ci](#) for more information on the `-resume` option.

### Related Topics

[ENOVIA Synchronicity Command Reference: add](#)

[ENOVIA Synchronicity Command Reference: ci](#)

[ENOVIA Synchronicity Command Reference: mvmember](#)

[ENOVIA Synchronicity Command Reference: remove](#)

[ENOVIA Synchronicity Command Reference: addhref](#)

[ENOVIA Synchronicity Command Reference: rmhref](#)

## Data Management of Modules

The topic [Vaults, Versions and Branches](#) explains how objects under data management are version-controlled. As an object's development progresses, new versions are created, such as versions 1.1, 1.2, and 1.3 of an object, on the default `Trunk` branch (branch 1). You can branch an object, with parallel development of the object taking place on the side branch, for example, versions 1.1.1.1, 1.1.1.2, and 1.1.1.3 on the branch 1.1.1.

When working with module data, the module object is version-controlled; module members are not independently version-controlled. The software's internal data management of module members is transparent to users of the software.

Creating a module (using the `mkmod` command) creates version 1.1 of that module, without any module content. To create module content, use the `add` command to add files and directories to the module. Then use the `ci` command to check in the files and directories to the module. The checkin creates version 1.2 of the module. The files and directories are now members of the module.

When changes to the module content occur, DesignSync creates new versions of the module. See *What Is a Module?* for operations that cause a new version of a module to be created. As with ordinary DesignSync data, you can branch a module so that parallel development of the module can take place on the side branch.

### Related Topics

Module Locking

Module Branching

Module Merging

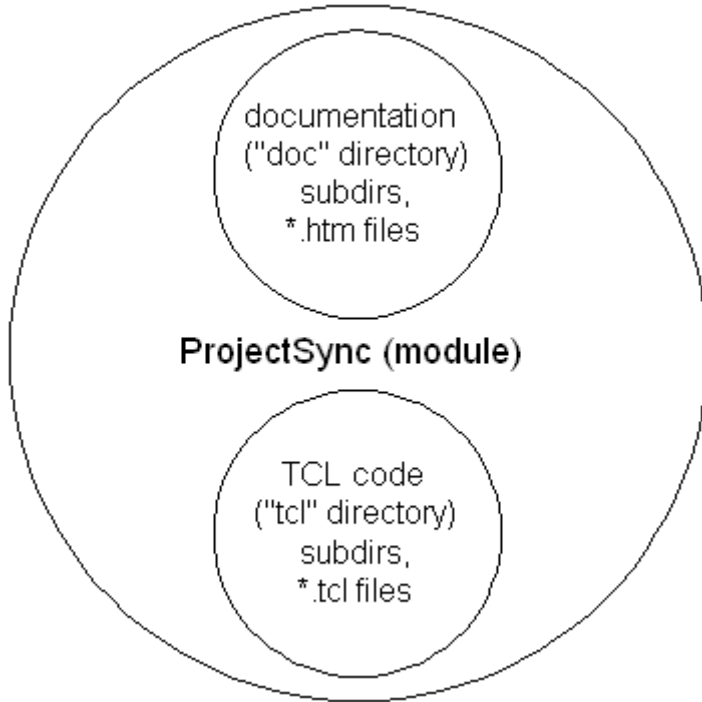
ENOVIA Synchronicity Command Reference: `add`

ENOVIA Synchronicity Command Reference: `ci`

ENOVIA Synchronicity Command Reference: `mkmod`

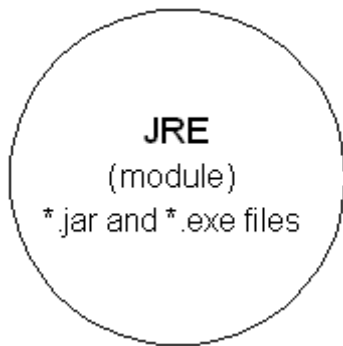
## Operating on Module Data

You can run DesignSync commands on modules or on the contents of a module (a module's members). Let's use the ProjectSync module in *What Is a Module?* as an example:



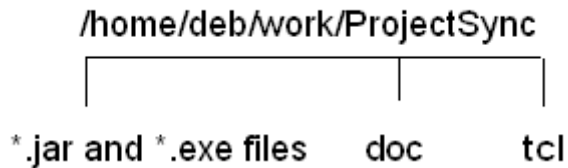
ProjectSync is a top-level module with subdirectories for its code and documentation.

Let's add a module to the module hierarchy for the JRE (Java Runtime Environment):



Both the SyncServer and ProjectSync use the JRE, so both the SyncServer and ProjectSync modules have hierarchical references to the JRE module which were added using the **addhref** command.

Fetching the ProjectSync module hierarchy into the workspace directory, `/home/deb/work/ProjectSync`, results in this structure:



Although not shown in the above diagram, there are subdirectories and files below the `doc` and `tcl` directories. Note that a "JRE" directory was not created. DesignSync followed the reference to the JRE module, and fetched the content of the JRE module.

`/home/deb/work/ProjectSync` is the base directory of the ProjectSync module.

You can run commands that operate on module data on:

- A module, such as all data belonging to the ProjectSync module (all data in the `doc` and `tcl` subdirectories -- not the `*.jar` and `*.exe` files belonging to the referenced JRE module)
- Individual module members, such as the JRE module's `*.jar` and `*.exe` files
- Workspace folders within a base directory, such as the `tcl` subdirectory in the `/home/deb/work/ProjectSync` base directory

Operations can be *module-centric* or *folder-centric*. See the Module Recursion topic for details.

### Notes:

- The `co` command does not apply to modules or module members; use `populate` to fetch modules and module members.
- To tag module data, specify the module's location *on the server*. Also, you can tag only modules, not individual module members.

### Animated Examples

- Operating on a module
- Operating on a module's contents

### Related Topics

Filtering Module Data

Module Locking

Module Branching

Module Merging

ENOVIA Synchronicity Command Reference: modules

ENOVIA Synchronicity Command Reference: addhref

ENOVIA Synchronicity Command Reference: co

ENOVIA Synchronicity Command Reference: tag

## Auto-Merging

You can check-in from a non-Latest module version, without having to merge first if the objects you modified have not been updated on the server. This is achieved by an *auto-merge check-in*. This functionality applies to any operation that creates a new module version, such as Renaming a Module Member, or Creating a Hierarchical Reference.

The auto-merge capability may be disabled for all module data on a server, forcing a strict policy to check-in only from Latest. For details, see SyncAdmin's Modules Options.

Auto-merging can only be used to update a module when the objects modified locally are identical to the objects they replace on the server. If there is a change to the object on the server, you must either perform a merge to incorporate the changes into your workspace, or use the skip option with the checkin operation to update the server. Note that auto-merging happens at the file level. The contents of files are not automatically merged. Auto-merge will not overwrite changes that other users have made to the module.

The `-skip` option to the `ci` command can be used to skip over changes that were made in subsequent module versions. The `-skip` option is applied to individual module members. To disallow the use of the `-skip` option, your project leader can define a client trigger.

When an auto-merge occurs, the fetched version number of the module is not updated at the end of the operation. This is because there may be changes in intermediate versions of the module, that are not reflected in the workspace.

Output from the `ci` command includes details of an auto-merge.

Similarly, structural changes that result in a new version of a module, such as adding or removing href's, or renaming member files, do not require that your workspace have the Latest version of a module.

Note: The automerge functionality is not valid for the following operations:

- Adding a file to a module - because the file did not exist in the previous version of the module, it must be explicitly added (`add`) or checked in with the Allow Checkin of New Items option (`ci -new`).



- Removing a file from a module - Removing a file from a module (`remove`) immediately creates a new module version without the removed module member. Therefore this operation never requires an auto-merge.

### Examples

The examples below all begin with version 1.4 of the module Chip in your workspace. Version 1.4 consists of the files: file1, file2 and file3.

Latest version 1.5 of the module contains a new version of file1. You modified file2 and file3.

Module version 1.5 contains a new version of file2. Module version 1.6 contains a new version of file3. You modified file1 and file2.

Module version 1.5 contains the new file file4. You created and added file4.

Module version 1.5 renamed file1 to file4. You modified file2.

Module version 1.5 renamed file1 to file4. You modified file1.

Module version 1.5 does not contain file1. You modified file1.

**Latest version 1.5 of the module contains a new version of file1. You modified file2 and file3.**

The check-in proceeds, because none of the files you modified were altered in later versions of the module. The new module version 1.6 is created, containing the updated file2 and file3 from your workspace, as well as the file1 from the module version 1.5.

The fetched version number of the module in your workspace is not updated, remaining at version 1.4. This is because the workspace version does not have all of the changes that were made in intermediate versions. The status of file2 and file3, as reported by `ls -report S`, is “Out-of-sync”. The module status, as reported by `showstatus`, is “Needs Update”.

View an animated illustration of this example.

**Module version 1.5 contains a new version of file2. Module version 1.6 contains a new version of file3. You modified file1 and file2.**

The check-in fails, with an error message such as:

```
file2: Error: Newer version exists in vault
```

This is because your change to file2 conflicts with the new version of that file that was introduced in version 1.5. File content is not automatically merged.

You should `populate -merge` the module, to merge in the later changes.

## DesignSync Data Manager User's Guide

View an animated illustration of this example.

**Module version 1.5 contains the new file file4. You created and added file4.**

Your local copy of file4 is in the "Added" state, from having run the add command.

The check-in fails, with an error message such as:

```
file4: Error: New object already exists in vault
```

The two `file4` files are different, so the error cannot be resolved by merging. Instead, either:

- remove the other `file4` from the module, using the `remove` command; or
- rename the other `file4` in the module, using the `mvmember` command; or
- rename the `file4` in your workspace, using the `mvmember` command; or
- delete the `file4` in your workspace, using the `rmfile` command, if your local `file4` is not needed.

View an animated illustration of this example.

**Module version 1.5 renamed file1 to file4. You modified file2.**

The check-in proceeds, because there is no overlap in the changes. Your modified `file2` gets checked into the new module version 1.6. As with the first example, the fetched version number of the module in your workspace is not updated. The workspace still has version 1.4 of the module. Also, `file1` remains in your workspace, since the changes in version 1.5 have not been fetched. A subsequent `populate` is required, to reflect the rename of `file1` to `file4`.

View an animated illustration of this example.

**Module version 1.5 renamed file1 to file4. You modified file1.**

The check-in fails, with an error message such as:

```
file1: Failed: Object has been moved
```

You can either manually accept the change, by using `populate -merge` to merge with the new module version. Or run `mvmember` directly on the server module, to move the `file1` member back to its original path.

View an animated illustration of this example.

**Module version 1.5 does not contain file1. You modified file1.**

The check-in fails, with an error message such as:

```
file1: Failed: Object does not exist in Latest version of
module
```

Because file1 is not in the Latest version of the module, you are attempting to check-in a new file. This requires that the `-new` option to the `ci` command be used.

Or, you can use the `add` command, to re-add the object, and then perform the `ci`. The `ls` command will report the object as "Added".

View an animated illustration of this example.

### Related Topics

Modules Options

Adding a member to a module

Removing a file from a module

Renaming a Module Member

Creating a Hierarchical Reference.

ENOVIA Synchronicity Command Reference: add

ENOVIA Synchronicity Command Reference: remove

ENOVIA Synchronicity Command Reference: mvmember

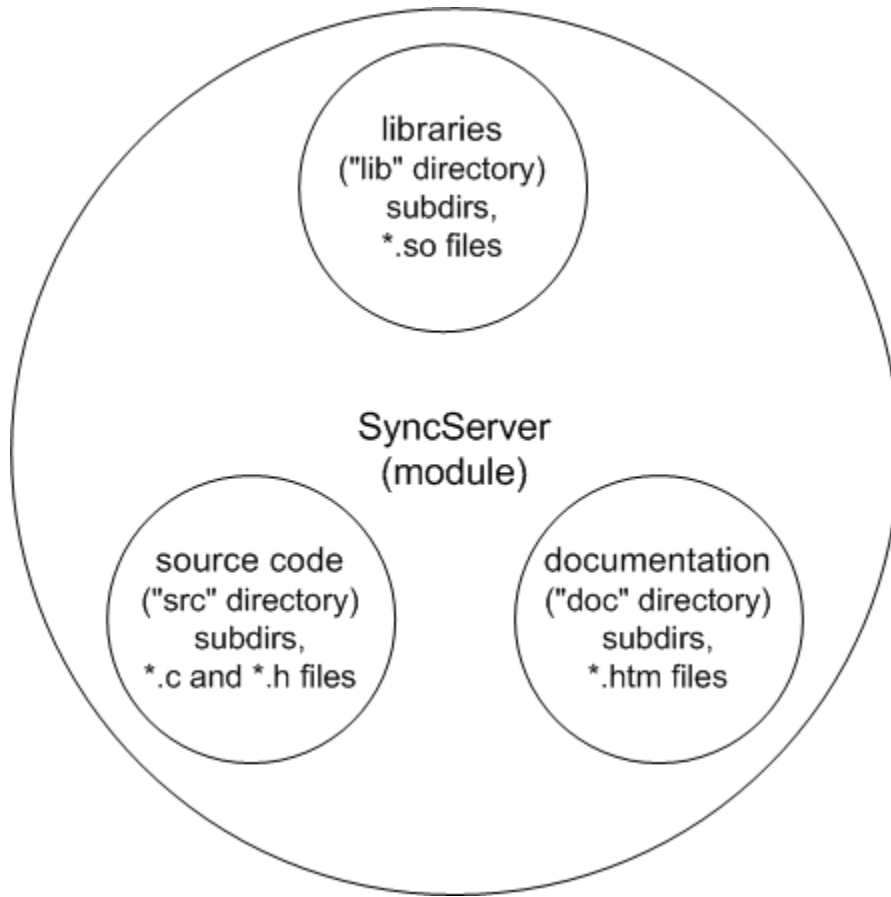
ENOVIA Synchronicity Command Reference: rmfile

ENOVIA Synchronicity Command Reference: ci

## Understanding Module Views

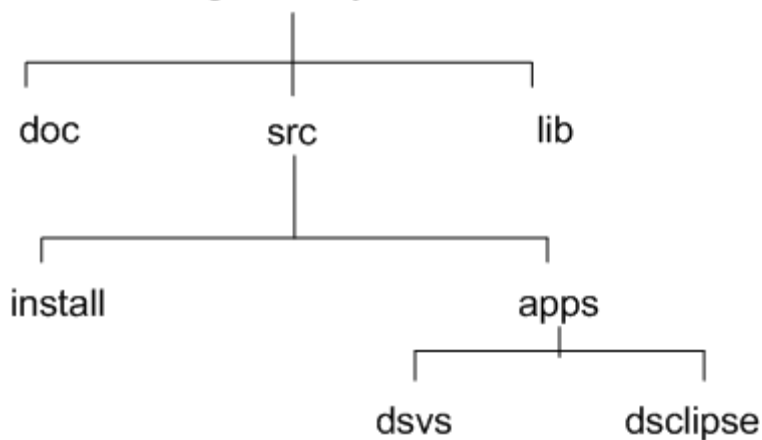
A design environment frequently includes modules with thousands of member files and many sub-modules. In most cases, you only need access to a subset of this data. Your DesignSync administrator or project leader can define sets of filtering rules to create a particular view of the module that only includes the files you need. This **module view** is stored on the DesignSync server for convenient repeated use. You can then apply one or more of the module views when populating a workspace on the client.

Let's use the SyncServer module hierarchy in What Is a Module? as an example. The SyncServer module has subdirectories for libraries, source code, and documentation:



Drilling down a little into the module, shows us a multiple application development environment.

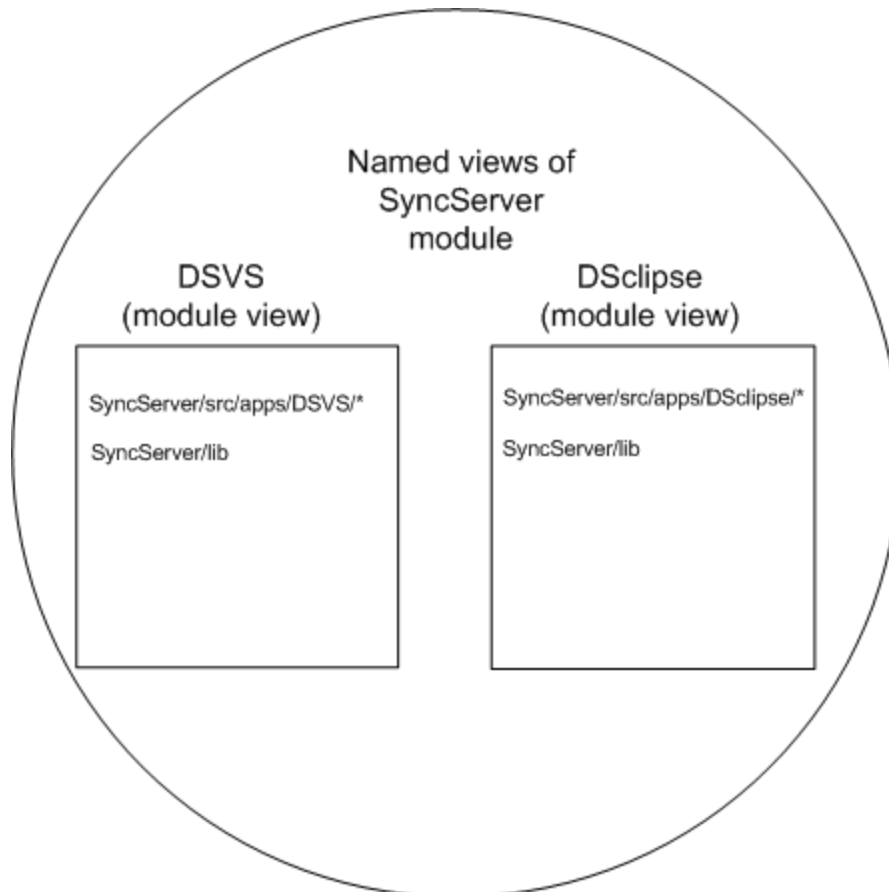
/home/barbg/work/SyncServer



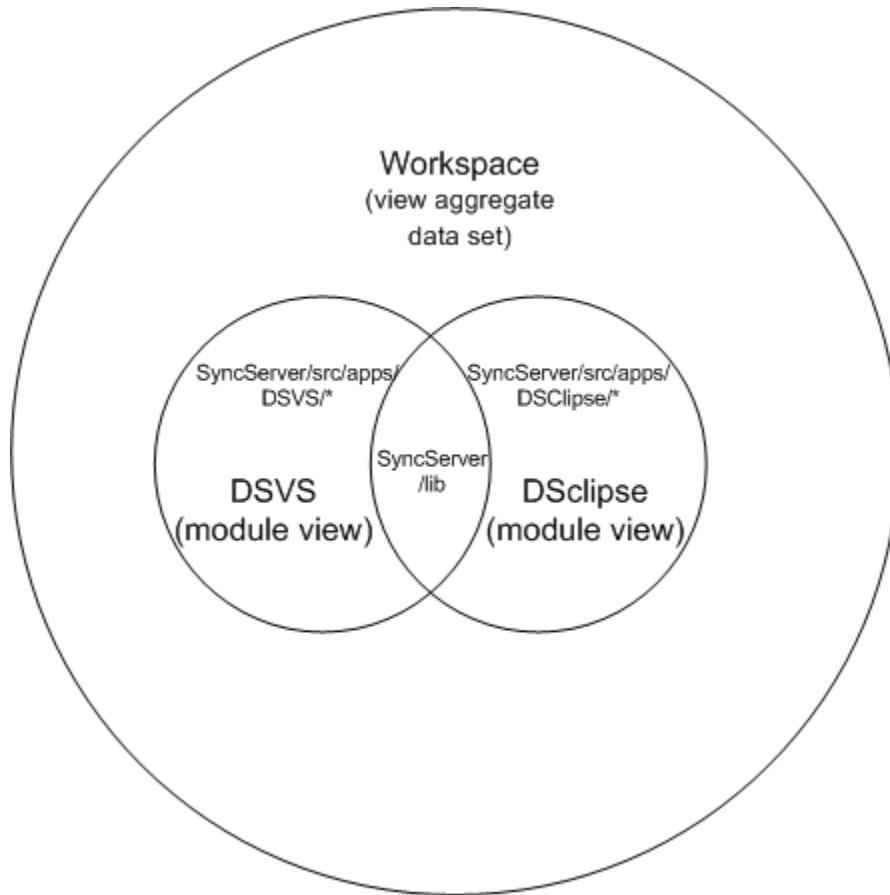
The administrator or project leader creates a definition consisting of filters and hrefilters and uploads them to server as a named module view. In this example, there are many logical users including the dsvs programming team, and the dsclipse programming

team. To support these users, the administrator can create the module views: DSVS, DSclipse.

In our example, these views contain the complete copies of the specific application directory and the lib directory.



The contents of a single view is the **initial view data set**. When you populate more than one view into your workspace, for example if you are on both the DSVS and DSclipse development teams, you populate the union of two view initial data sets, or an **aggregate view data set**:



After the candidate members of the view being populated are identified, DesignSync applies filters or hrefilters that have been set with the Filter (-filter), Exclude (-exclude), or Href filter (-hrefilter) options. The final dataset populated into the workspace after all views and filter have been applied is called the **filtered view**.

### Modules Views in a Module Hierarchy

If your project uses a module hierarchy, the DesignSync administrator or project manager can use the same named view in all the modules and submodules that contain the necessary files for a complete work environment. When the project hierarchy has been set up for module views, you can populate recursively to recursively populate the hierarchy containing the desired view into your system.

**Note:** If any module in the module hierarchy does not contain the named view, the populate will fetch the entire module. Thus, using our example above, if the library files were not contained within the SyncServer module, but were instead a separate submodule called lib, the administrator could create the DSVS and DScclipse views within the lib module to identify the specific library files that project needs, or by not creating the views, allow the user to populate the entire lib module.

### Related Topics

Filtering Module Data

Module Hierarchy

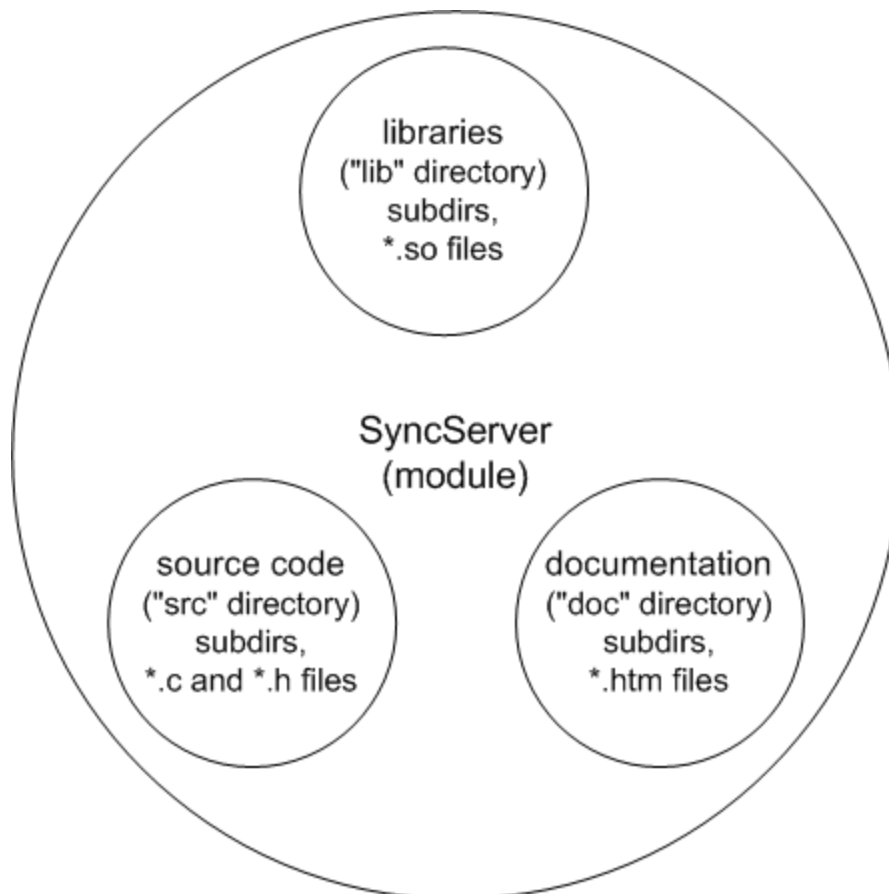
Populating Your Work Area

*ENOVIA Synchronicity DesignSync Data Manager Administrator's Guide: Overview of Module Views*

*ENOVIA Synchronicity DesignSync Data Manager Administrator's Guide: Creating Module View Definitions*

## Filtering Module Data

You can narrow operations on modules to a subset of the module's data. Let's use the SyncServer module hierarchy in *What Is a Module?* as an example. The SyncServer module has subdirectories for libraries, source code, and documentation:



## DesignSync Data Manager User's Guide

The "doc" directory consists of \*.htm files and an "images" directory containing \*.gif and \*.vsd files.

A SyncServer writer does not need to fetch the entire contents of the "doc" directory. Instead, the writer can fetch only the "images" directory. This results in the workspace structure:

/home/barbg/work/SyncServer



Note that the /home/barbg/work/SyncServer workspace directory was created by the user. The doc subdirectory in the workspace was created by the fetch. No files directly below the doc subdirectory were fetched, because the user specified that only files in the doc/image directory be fetched.

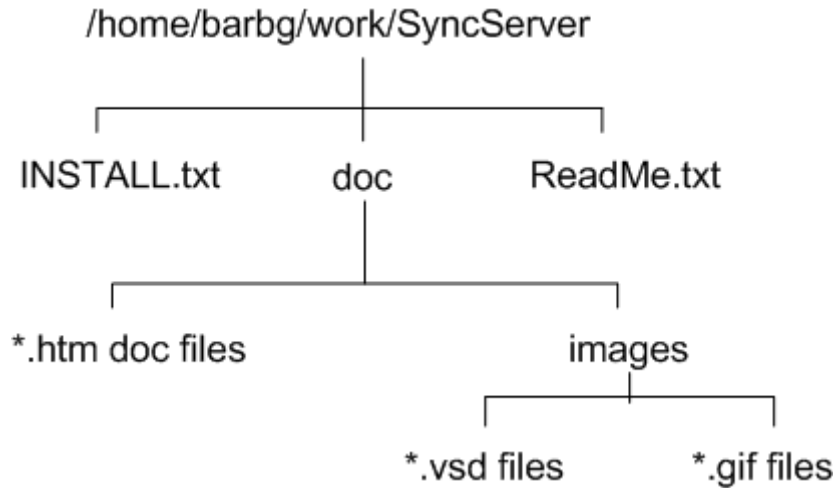
The writer can further narrow module operations to filenames matching a specified pattern, such as \*.vsd files in the image directory.

You can exclude or include data by using the `-filter` option to commands such as **populate**. By default, the `-filter` option excludes data. To operate only on certain data, you need to first exclude all data, then include specific data, by using the "-" and "+" designators to the `-filter` option. See the command documentation for descriptions and examples of the `-filter` option.

Continuing the SyncServer example, the release engineer for SyncServer adds the files `INSTALL.txt` and `ReadMe.txt` to the SyncServer module (by using the `add` and `ci` commands). The DesignSync documentation is no longer isolated to the `doc` subdirectory.

To fetch all documentation files pertaining to DesignSync, a DesignSync writer needs to fetch the \*.txt files in the DesignSync module and the DesignSync module's entire `doc` subdirectory. This results in the workspace structure:

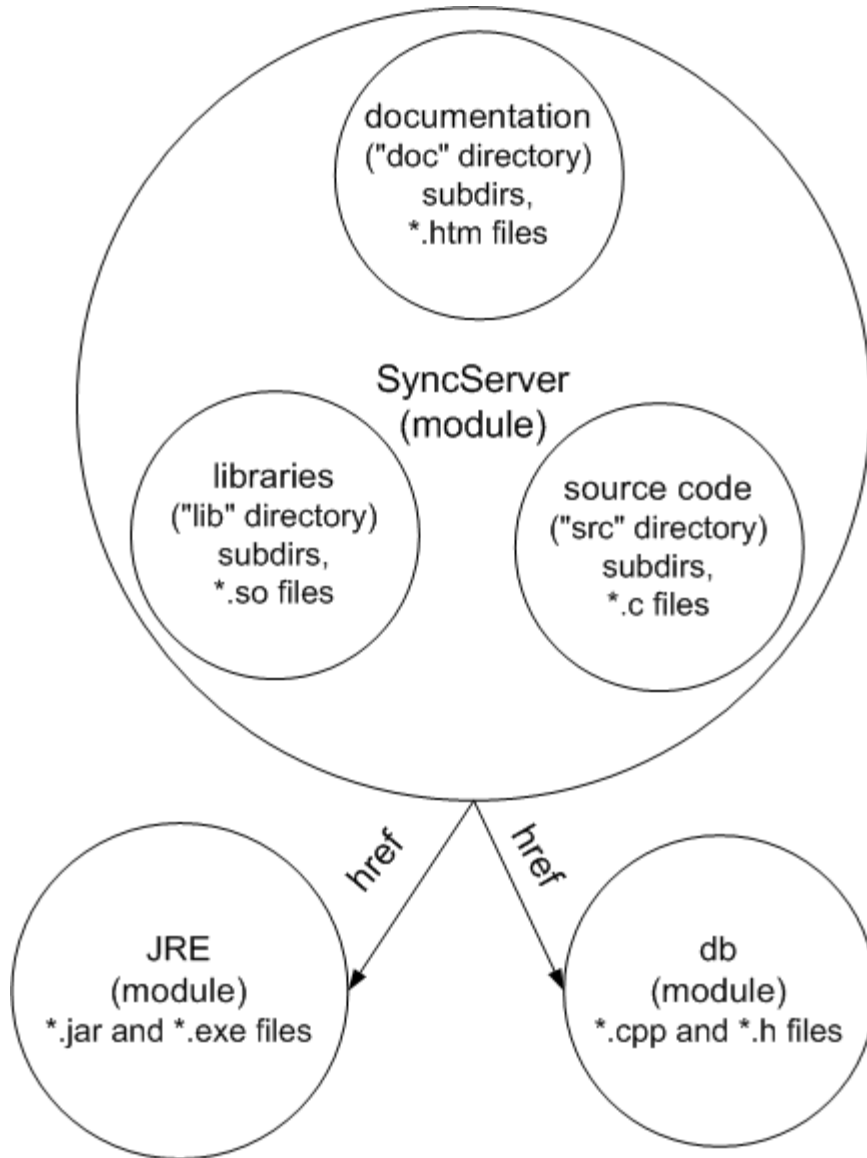




Note that the `/home/barbg/work/SyncServer` workspace directory was created by the user. The `doc` subdirectory in the workspace was created by the fetch.

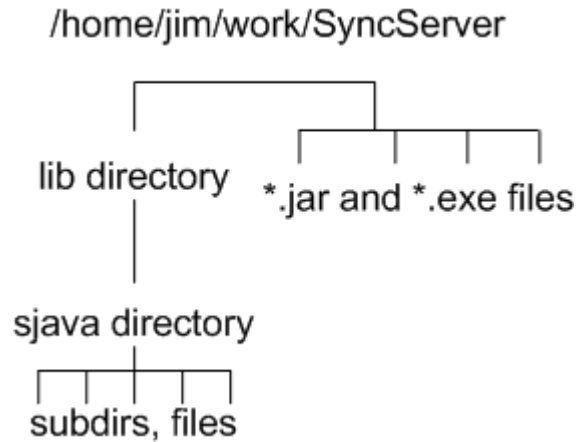
You can also filter module data by excluding the hierarchical references followed by recursive operations. (Hierarchical references can only be *excluded* from operations -- there is no include inverse.)

To demonstrate, let's add two modules to the module hierarchy. The two modules are "JRE" (for the Java Runtime Environment) and "db" (for the database engine). After adding those two modules by using the `addhref` command, the DesignSync module hierarchy is:



A developer who wants to work on the Java libraries needs the `java` data in the `lib` subdirectory of the SyncServer module and the referenced JRE module. To do so, the developer uses the **populate** command, specifying that the `lib/*java` data be included (using the `-filter` option discussed above), and specifying that the referenced `db` module be excluded (using the `-hreffilter` option). See the command documentation for descriptions and examples of the `-hreffilter` option.

The populate results in the workspace structure:



Note that the user created the `/home/jim/work/SyncServer` workspace directory. The `lib` directory and its contents were created by the `fetch`. Note that a "JRE" directory was not created. DesignSync followed the reference to the JRE module and fetched the content of the JRE module.

Operations can be *module-centric* or *folder-centric*. See the Module Recursion topic for details.

#### Animated Examples

- Filtering
- Persistent populate filter

#### Related Topics

Operating on Module Data

Setting Persistent Populate Filters

ENOVIA Synchronicity Command Reference: `setfilter`

ENOVIA Synchronicity Command Reference: `populate`

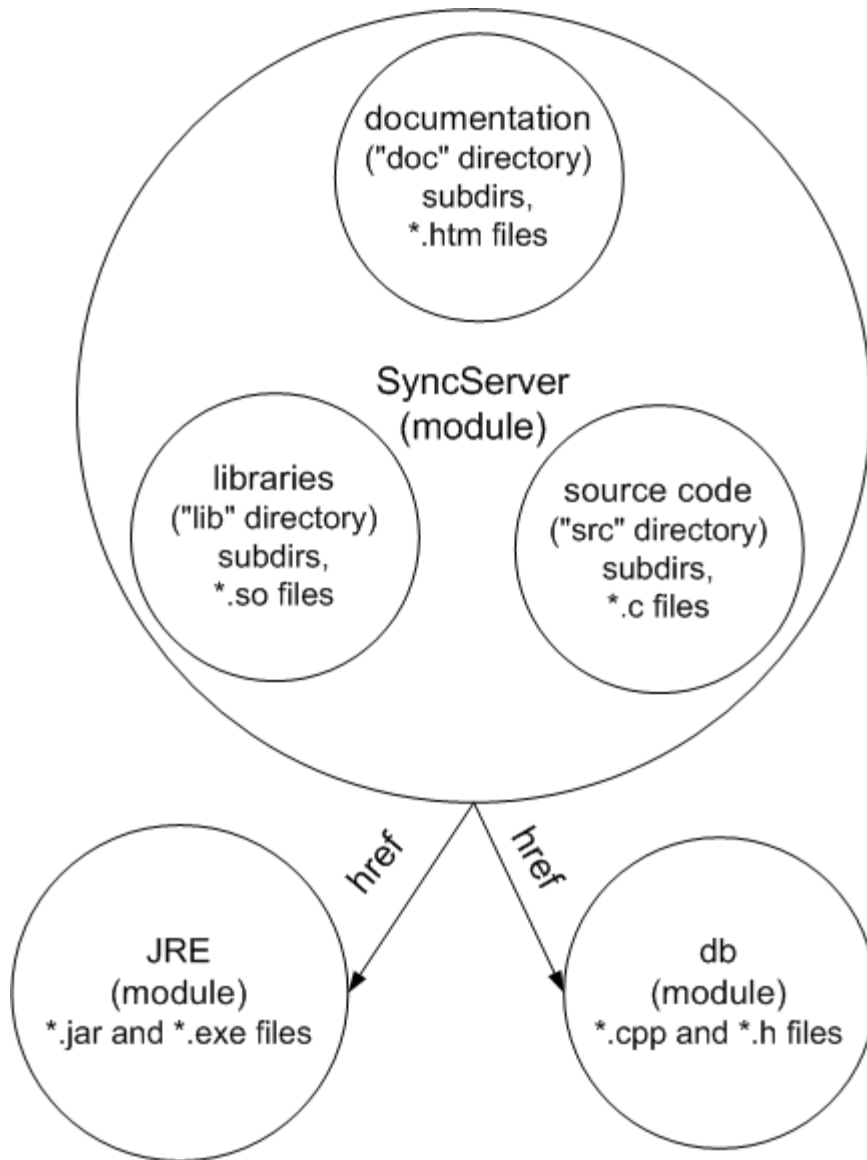
ENOVIA Synchronicity Command Reference: `add`

ENOVIA Synchronicity Command Reference: `ci`

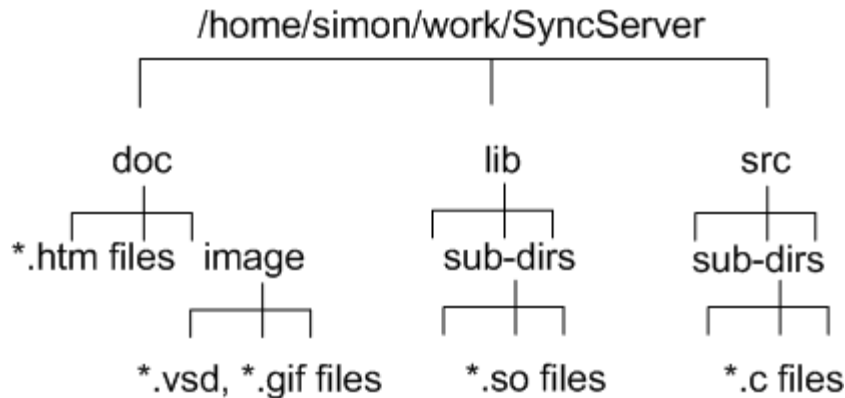
ENOVIA Synchronicity Command Reference: `addhref`

## Module Recursion

Operations can operate on an entire module hierarchy, a single module, a folder within a module, or a submodule that's referenced by a module. Let's use the SyncServer module hierarchy in Filtering Module Data as an example. The SyncServer module contains directories with data, and hierarchical references (*hrefs*) to the JRE and db modules:



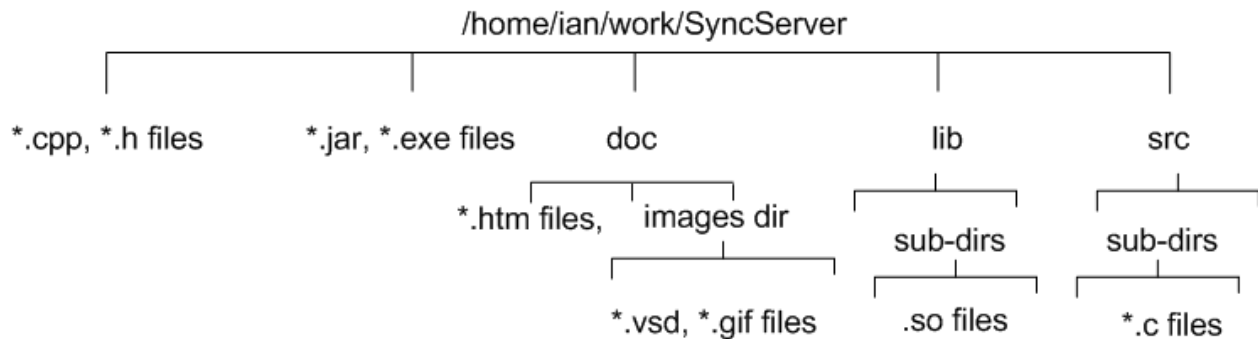
By default, **populate** is not recursive; populate fetches only the module's data, resulting in the workspace structure:



Note that the user created the `/home/simon/work/SyncServer` workspace directory.

A module is considered a single object. A populate of a single module fetches all of the module's members, except for hierarchical references. Because the operation is not recursive, DesignSync does not follow hierarchical references.

When using **populate** with the `-recursive` option, the entire SyncServer module hierarchy is fetched, resulting in the workspace structure of:



Note that the user created the `/home/ian/work/SyncServer` workspace directory. Because the `-recursive` option was specified, DesignSync followed the referenced JRE and db modules and fetched their data.

**Note:** For information about which module versions are populated when a module hierarchy is recursively into a workspace, see Module Hierarchy.

Operations can handle modules (*module-centric*) or workspace directories (*folder-centric*).

#### Module-Centric Operations

Operations are *module-centric* if they operate on:

## DesignSync Data Manager User's Guide

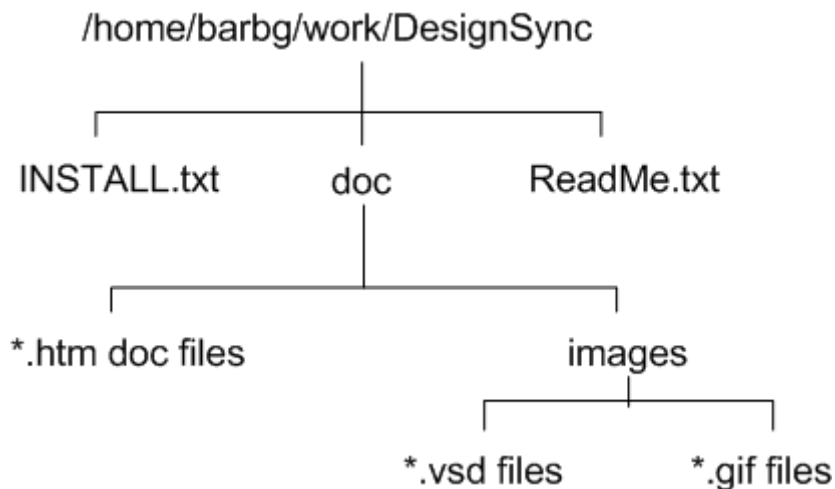
- An individual module's content, such as the entire content of the `/home/simon/work/SyncServer` workspace shown above
- An entire module hierarchy, such as the entire content of the `/home/ian/work/SyncServer` workspace shown above
- A subfolder of a specific module, such as the `lib` subdirectory in either of the `/home/simon/work/SyncServer` or `/home/simon/work/SyncServer` workspaces shown above

You can specify a referenced module as the object of a *module-centric* operation. Referenced modules are described in the topic Module Hierarchy.

### Folder-Centric Operations

To illustrate *folder-centric* operations, let's return to the DesignSync example in What Is a Module?. Both the SyncServer and ProjectSync submodules contain a "doc" directory, which has `*.htm` files and an "image" subdirectory (with `*.vsd` and `*.gif` files). The top-level DesignSync module contains `*.txt` files. A documentation writer working on both SyncServer and ProjectSync fetches into their workspace only files pertaining to the documentation (as described in Filtering Module Data).

DesignSync fetches both the SyncServer and ProjectSync modules into the same workspace, `/home/barbg/work/DesignSync`, in the example below. Note that top-level "DesignSync" and "ProjectSync" directories are not created. It is the data within those modules that is fetched, resulting in this workspace structure:



Since both the SyncServer and ProjectSync modules contain `doc` subdirectories, the `doc` workspace directories will contain data from both of the SyncServer and ProjectSync modules. These are referred to as *overlapping modules*, because they share a common base directory.

A *folder-centric* operation operates on a workspace directory, ignoring module boundaries. The workspace directory can belong to multiple modules. For example, the

`doc` and `images` directories in the `/home/barbg/work/DesignSync` workspace shown above belong to both the `SyncServer` module and the `ProjectSync` module.

### Animated Examples

- Module-centric operations on a module
- Folder-centric operations
- Module-centric operations on a sub-folder
- Module-centric operations on hierarchical references

### Related Topics

Module Hierarchy

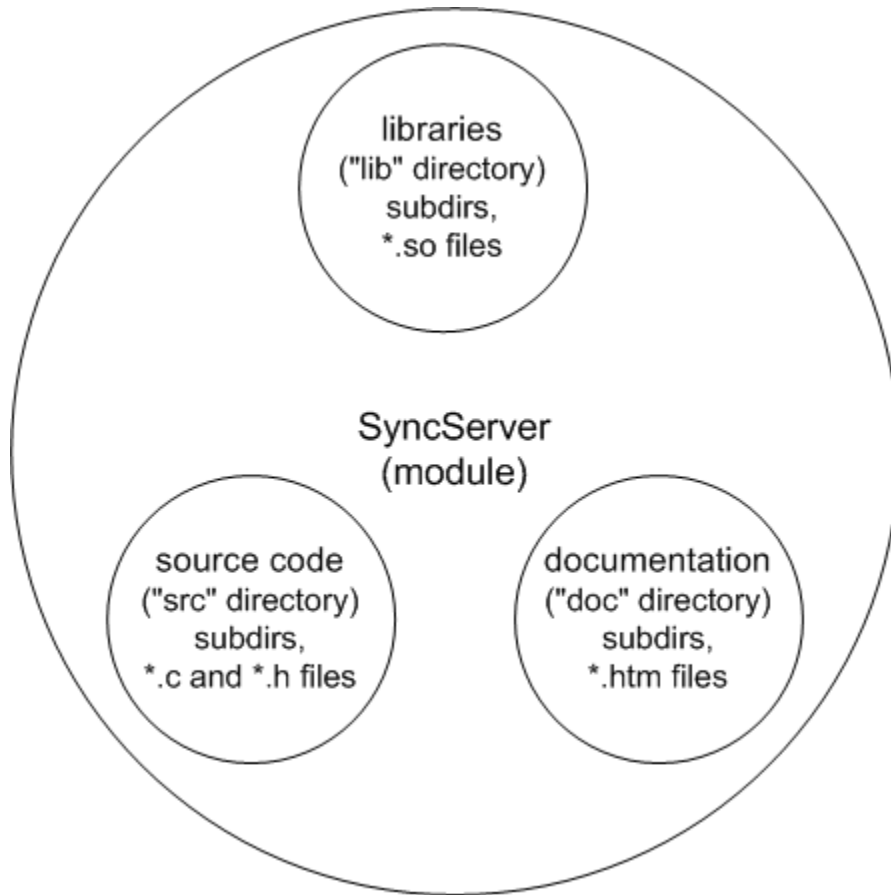
Populating Your Work Area

ENOVIA Synchronicity Command Reference: `populate`

## Module Locking

A module is managed by DesignSync, with a new version of a module created when the content of the module is modified. (See [What Is a Module?](#) for details.) By default, DesignSync creates all versions on the `Trunk` branch. You can branch modules (see [Module Branching](#)), with new versions of the module created on the branch on which you are working.

It is a *branch* of a module that is locked, not an entire module. You can also lock module content. Let's use the `SyncServer` module hierarchy in [What Is a Module?](#) as an example. The `SyncServer` module contains directories, with data in those directories:



A user can lock the branch of a module, thus reserving the right to create the next version of the module. In this example, that means only the user who locked the SyncServer module's branch can create a new version of the SyncServer module on that branch.

Let's say the user `marci` locks the `Trunk` branch (branch 1) of the SyncServer module (by using the **lock** command). No one else can modify the content of the SyncServer module (on the `Trunk` branch) until `marci` releases the lock (for example, by using the **unlock** command).

If a branch of a module is not locked, users can lock items within the module (by using the **populate** command with the `-lock` option). For example, documentation writer `karen` can lock `*.htm` files in the `doc` directory of the SyncServer module, while developer `jim` locks files in the `lib` directory's `java` subdirectory. When `karen` checks in her locked and modified files, she creates a new version of the SyncServer module. Likewise, when `jim` checks in his changes, he creates a new version of the SyncServer module. Note that `karen` having checked in her changes first did not prevent `jim` from checking in his changes. That is because of auto-merging.

Locking a module member file reserves the right to modify, move, or remove the member file. Only the user who locked a member file can make changes to that



member file (either its modification, removal from the module, or move within the module). Another user might check in modifications to other unlocked member files, thus creating a new module version, as long as the modifications do not affect the locked member file.

When a new version of the module results from a change made to the member file that was locked, the lock on the member file is not released. This is so the user can continue to work on the member file, without having to reobtain the lock. The user who locked a member file can release the lock, by using the **cancel** command.

### Animated Examples

- Locking a module branch
- Locking module content

### Related Topics

[ENOVIA Synchronicity Command Reference: lock](#)

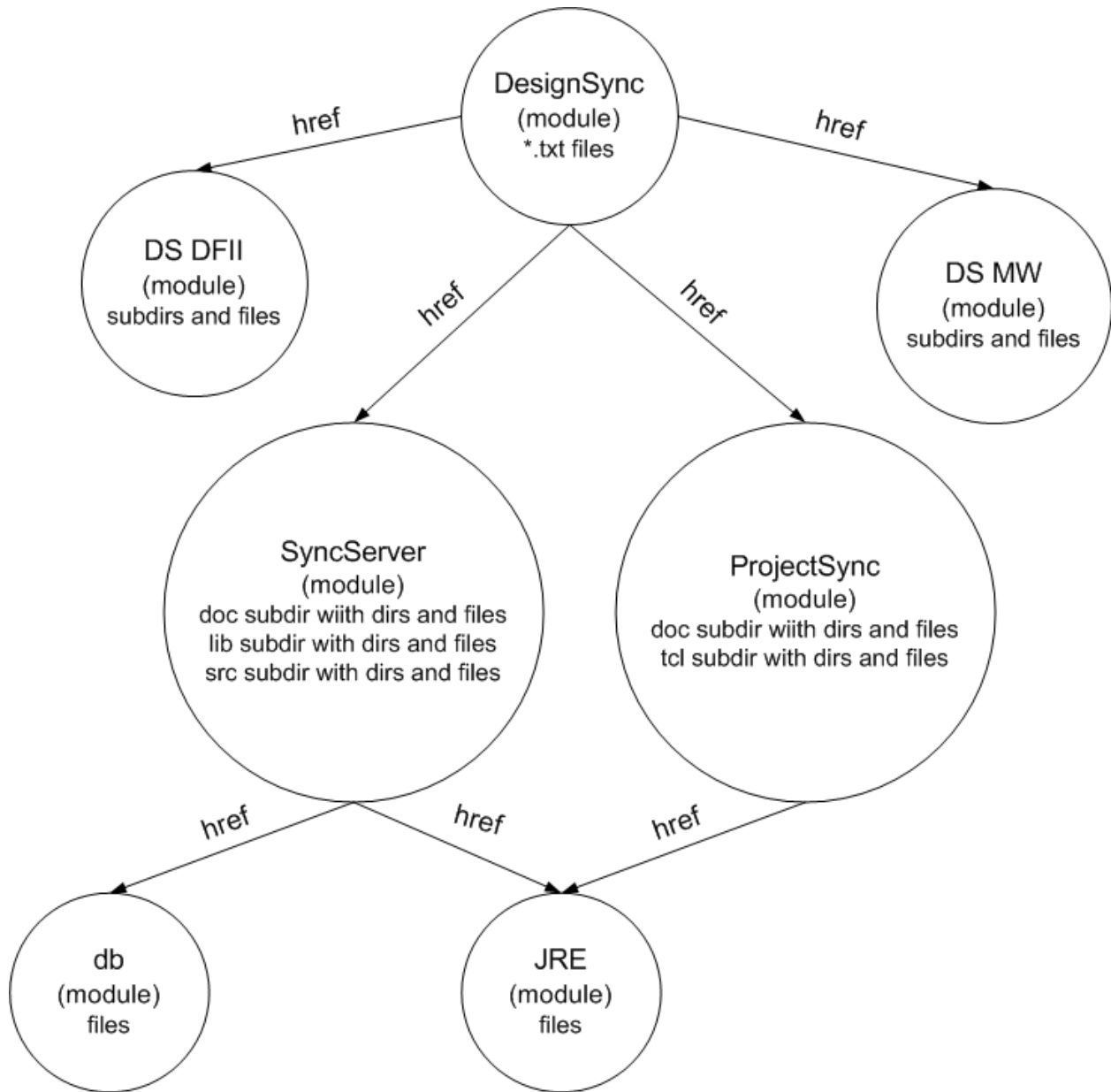
[ENOVIA Synchronicity Command Reference: unlock](#)

[ENOVIA Synchronicity Command Reference: populate](#)

[ENOVIA Synchronicity Command Reference:cancel](#)

## Module Hierarchy

A design hierarchy is comprised of modules, with the modules connected through hierarchical references (*hrefs*). The href connecting two modules indicates which module references the other. To illustrate, let's use the DesignSync module hierarchy, from earlier examples in this document:



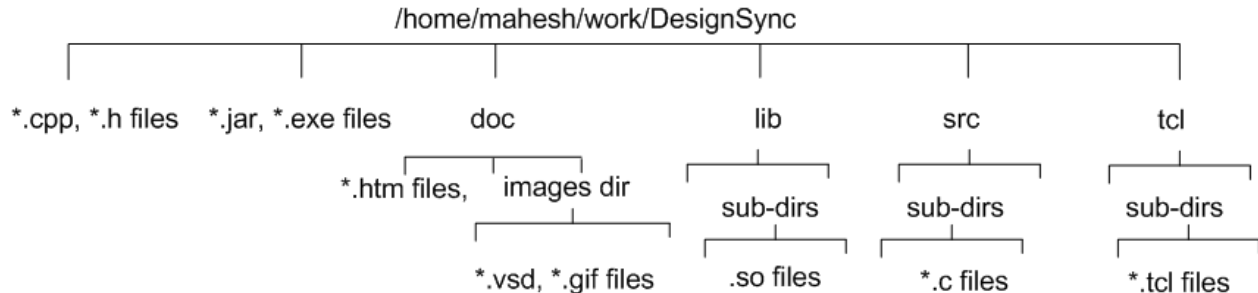
All of the arrows in the above diagram represent hierarchical references, which were added by using the **addhref** command. The arrow direction indicates that the href is from one module to another module. You can also remove hierarchical references by using the **rmhref** command.

An href from a module is considered to be a member of that module. As with any change to a module's contents, a change to a module's hierarchical references (either adding an href or removing an href) creates a new version of the module.

Fetching only the DesignSync module fetches only its content, \*.txt member files. Following the DesignSync hrefs (by specifying the `-recursive` option to the **populate**

command) also fetches the content of the SyncServer, DS DFII, and ProjectSync modules, and the content of their referenced db and JRE modules.

A developer working on both SyncServer and ProjectSync (and not on other products in DesignSync recursively fetches the DesignSync and ProjectSync modules into the same workspace, `/home/mahesh/work/DesignSync`, in the example below. Note that DesignSync does not create top-level "SyncServer" and "ProjectSync" directories, but fetches the data within those modules, resulting in this workspace structure:



Note that if the user who added an href had specified a *relative path* (by using the `-relpath` option to the **addhref** command), the referenced module's data would have been fetched into a subdirectory relative to path in the workspace.

For example, if the href from the SyncServer module to the db module had been added with the `-relpath` value, `database`, the fetch would have created a `database` directory below `/home/mahesh/work/DesignSync`, containing the db module's `*.cpp` and `*.h` files.

If the href from the SyncServer module to the db module had been added with the `-relpath` value `src`, the db module's `*.cpp` and `*.h` files would have been fetched into the `src` directory. The `/home/mahesh/work/DesignSync` workspace's `src` directory would then have data from both the SyncServer and db modules.

Operations in the `/home/mahesh/work/DesignSync` workspace can be either *module-centric* or *folder-centric*. See the Module Recursion topic for details.

## Understanding how href modes and module recursion build your data hierarchy

When you populate a workspace, there are three different modes to specify how hierarchical references should be evaluated in order to identify the versions of submodules to reference when populating a module recursively. These href modes are "normal," "dynamic," and "static."

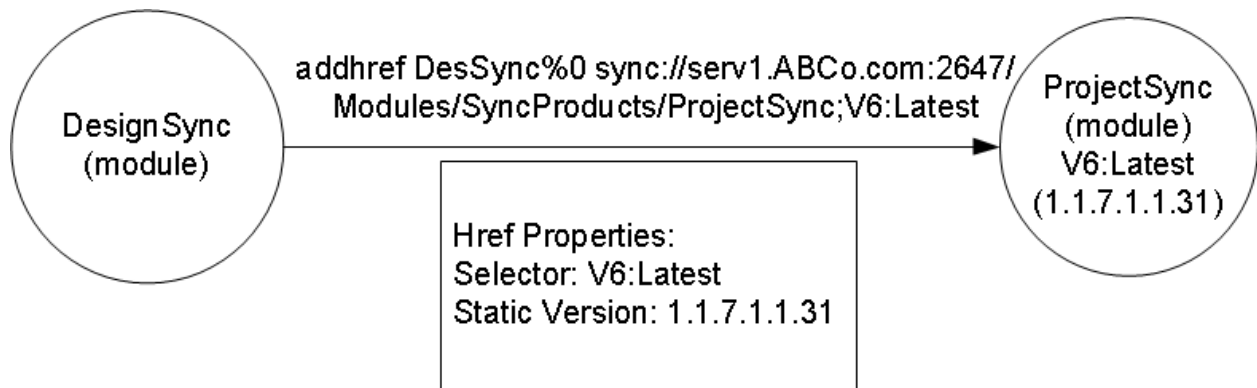
**Note:** To understand how a module hierarchy is constructed and processed, it is helpful to understand that selectors can be either dynamic, meaning that they might change, for example, `Trunk:Latest` will always point to the latest version on the Trunk branch, which

changes each time a checkin is performed; or static, meaning that the version indicated will always be the same, for example a version number. For information on understanding selectors, see Selector Formats.

When your href is initially created, DesignSync saves the following information about the version referenced:

- the selector used to refer to the submodule.
- the numeric version of the referenced submodule at the time the href was created or updated.

So, for example, using the initial example above, to create a reference from the DesignSync top level module to the V6 branch of the ProjectSync module, the reference might look like this:

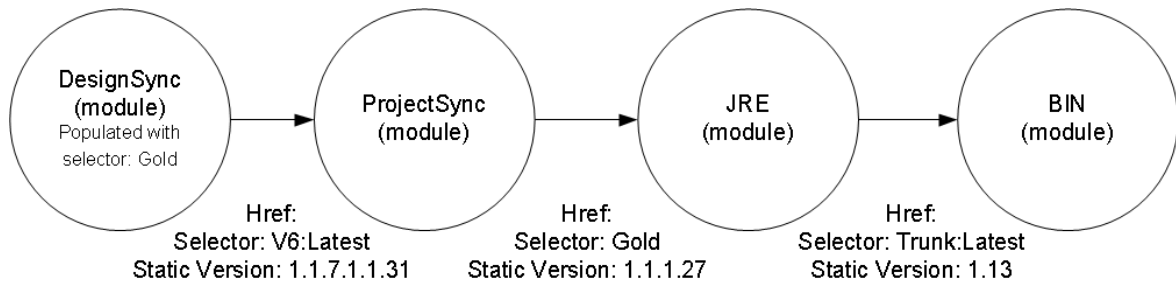


DesignSync stores:

- the selector information, in this case V6:Latest
- and the specific (static) version number that corresponds to the module version at the time the href was created, in this case 1.1.7.1.1.31

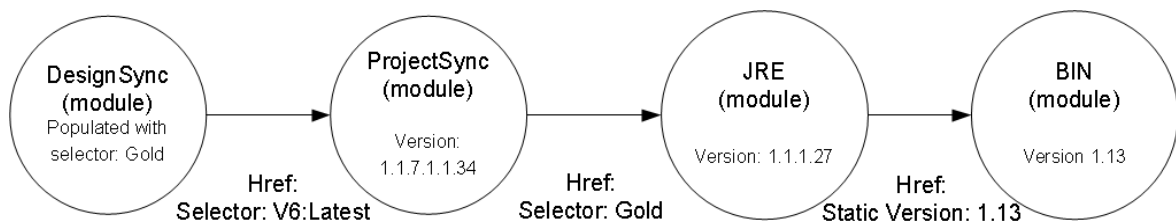
### Understanding Href Mode Traversal

DesignSync always resolves the selector of the topmost module when fetching a module hierarchy to the workspace. When populating recursively, the href mode by default is normal. This mode causes DesignSync to resolve the selector of each hierarchical reference and fetch the referenced version of the submodule. If the selector is a static version; for example, a version ID or a version tag, then the hrefmode from that point forward changes to always follow the static version in each subsequent href in the submodule's module hierarchy. If the selector is a dynamic selector, for example a branch selector, then the hrefmode stays in normal mode and the same process is repeated for the next level of submodules. So, for example, using this hierarchy:



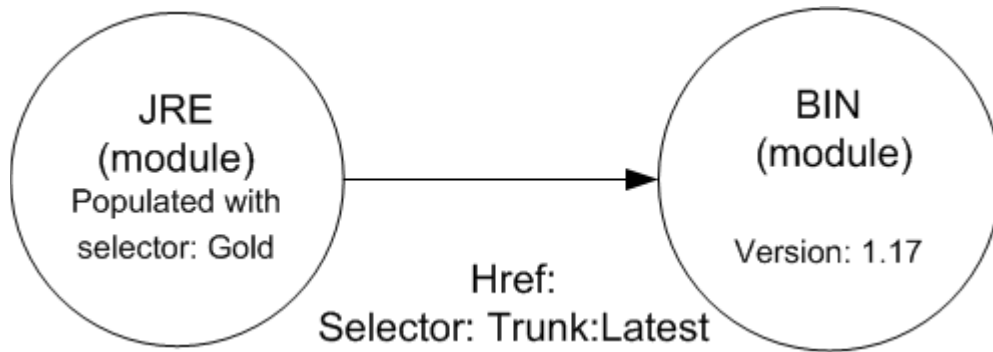
### Populating recursively with “-hrefmode normal”

Populating the DesignSync module with hrefmode normal first resolves the Gold selector of the top-level module, DesignSync, then resolves the V6:Latest selector in the href to the ProjectSync module. It fetches the latest version on the V6 branch of the ProjectSync module, which may be later than version 1.1.7.1.1.31. Since the V6:Latest selector is a branch selector, which is dynamic, the hrefmode stays in normal mode. Populate then processes the href to the JRE module. It resolves the Gold selector for JRE and references the appropriate version of that module. The Gold selector may reference a different version than the stored static version, 1.1.1.27. Because the Gold selector is a version tag, which is a static selector, the hrefmode now switches to static. Populate then uses the static version information in the href to fetch version 1.13 to the BIN module as shown in the following image:



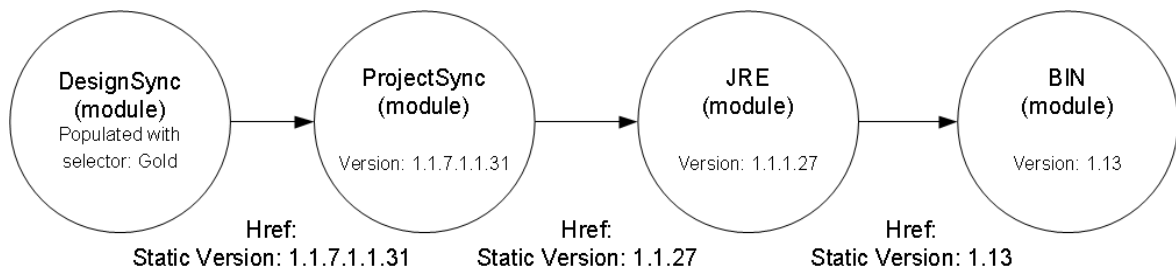
**Note:** In the previous example, development has continued on the ProjectSync module and those development updates are reflected in the example resulting in a different hierarchy than the one originally captured at the time the hrefs were created.

If you were to explicitly populate the JRE submodule, DesignSync would begin with JRE as the top level module and resolve the href to BIN in normal mode, rather than static mode as shown above. Assuming development has continued on the BIN module, the resulting hierarchy could look something like this:



### Populating with “-hrefmode static”

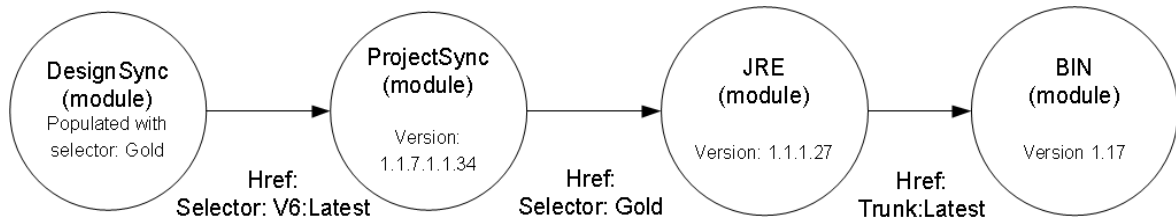
Using the static hrefmode when populating always forces the hrefs to use the static version information stored in each href. For example, using our existing data hierarchy, after fetching the DesignSync module, the 1.1.7.1.1.31 version of the ProjectSync module is fetched regardless of what is the latest version on the ProjectSync module’s V6: branch. Likewise, the 1.1.1.27 version of the JRE module is fetched regardless of what version the Gold selector in the href resolves. Finally, the 1.13 version of the BIN module is fetched. This results in a module hierarchy that looks like this:



**Note:** When you populate a workspace with a static hierarchical reference, you cannot check in any changes made in the workspace. Actively developing workspaces must be populated in dynamic mode so the checkin will be able to update the workspace with the new dynamic version after checkin.

### Populating with “-hrefmode dynamic”

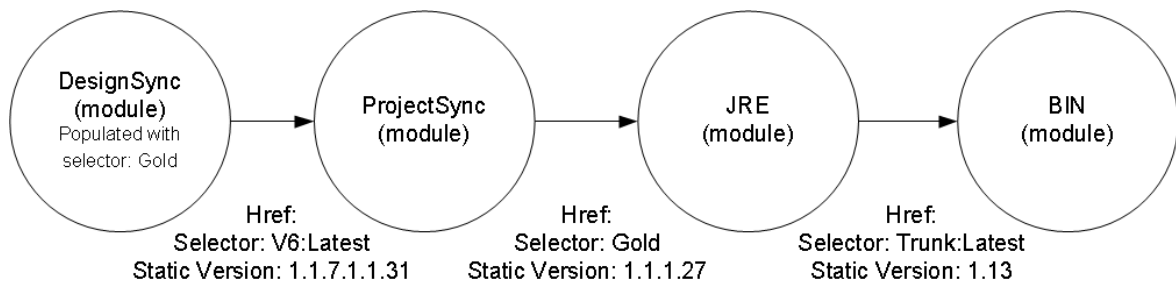
Using the dynamic hrefmode when populating always forces the hrefs to use the selector that’s stored in each href. In the above example, each of the 3 hrefs have their selectors resolved to determine which version of the ProjectSync, JRE, and BIN submodules are fetched. Assuming development has continued on the ProjectSync and BIN submodules as described in the Populating with “-hrefmode normal” section, the resulting hierarchy populated would look like this:



### An Alternate Method of Module Hierarchy Traversal

DesignSync also supports a different working methodology which examines the selector used to populate the top-level module in a hierarchy and always uses the selector to determine what hrefmode to use to operate on the rest of the module hierarchy. This methodology is defined by going into SyncAdmin and enabling "Change traversal mode with static selector on top level module" option in SyncAdmin.

This methodology produces a controlled hierarchy that always evaluates the selectors the same way regardless of the entry point in the hierarchy in which populate has run. So, continuing to our example module hierarchy would result in this:



The selector used to fetch the DesignSync module is a static version selector named Gold. If hrefmode "normal" is used for populate and the alternate methodology is in use, then the hrefmode will switch to static immediately after populating the topmost DesignSync module. Thus, the static version of all 3 hrefs are followed to fetch the ProjectSync, JRE, and BIN modules. Without the methodology set, as discussed above, the hrefmode won't switch from "normal" to "static" until after the href to JRE is followed (because the href selector to JRE was a static version).

If the selector used to fetch the DesignSync module was dynamic in nature, for example, a branch tag like Trunk:Latest, and hrefmode normal is in use, the hierarchical traversal stays in normal mode resulting in the same hierarchy described in Populating with "–hrefmode normal."

If you were to populate the JRE module recursively, instead of using normal mode for that selector, it would recognize that the submodule, BIN, should be populated statically as a referenced submodule and populate the static version of BIN, if needed.

**Important:** Using this methodology, if the selector used for the topmost module is a static selector (i.e., version ID or version tag), then the hrefmode switches immediately to static when processing the rest of the module hierarchy. This produces a controlled hierarchy that always evaluates the selectors the same way regardless of the entry point in the hierarchy in which populate has run.

### Animated Examples

- Creating module hierarchy
- Creating a peer structure module hierarchy
- Modifying module hierarchy

### Hrefs and Hierarchical Href Filtering

When building a data hierarchy in your workspace, you may choose to populate hierarchical references selectively, excluding hierarchical references that are unnecessary locally, or perhaps duplicated within the hierarchy.

You can filter an href in two ways:

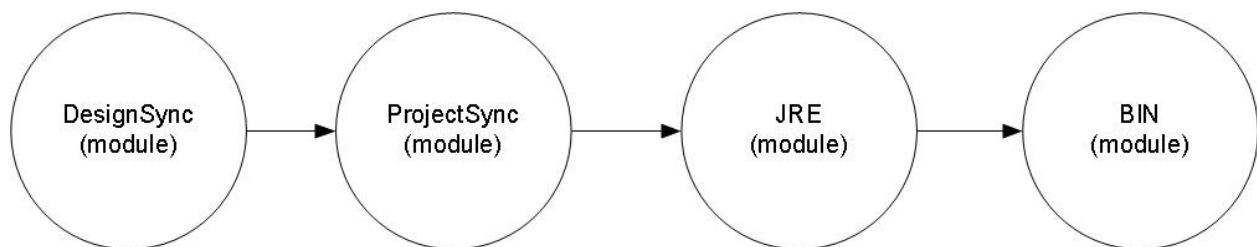
- simple href filter
- hierarchical href filter

You can specify any number of href or hierarchial href filters to the populate operation.

Note: You can only specify a hierarchical filter during an initial populate. You can specify a simple hierarchical filter during any populate, but it will not persist. If you need to change the persistent simple or hierarchical hreffilter on an existing workspace, you must use Setting Persistent Populate Views and Filters or the setfilter command.

### Simple Href Filtering

A simple href filter filters at any level of the hierarchy. To expand on our previous example:

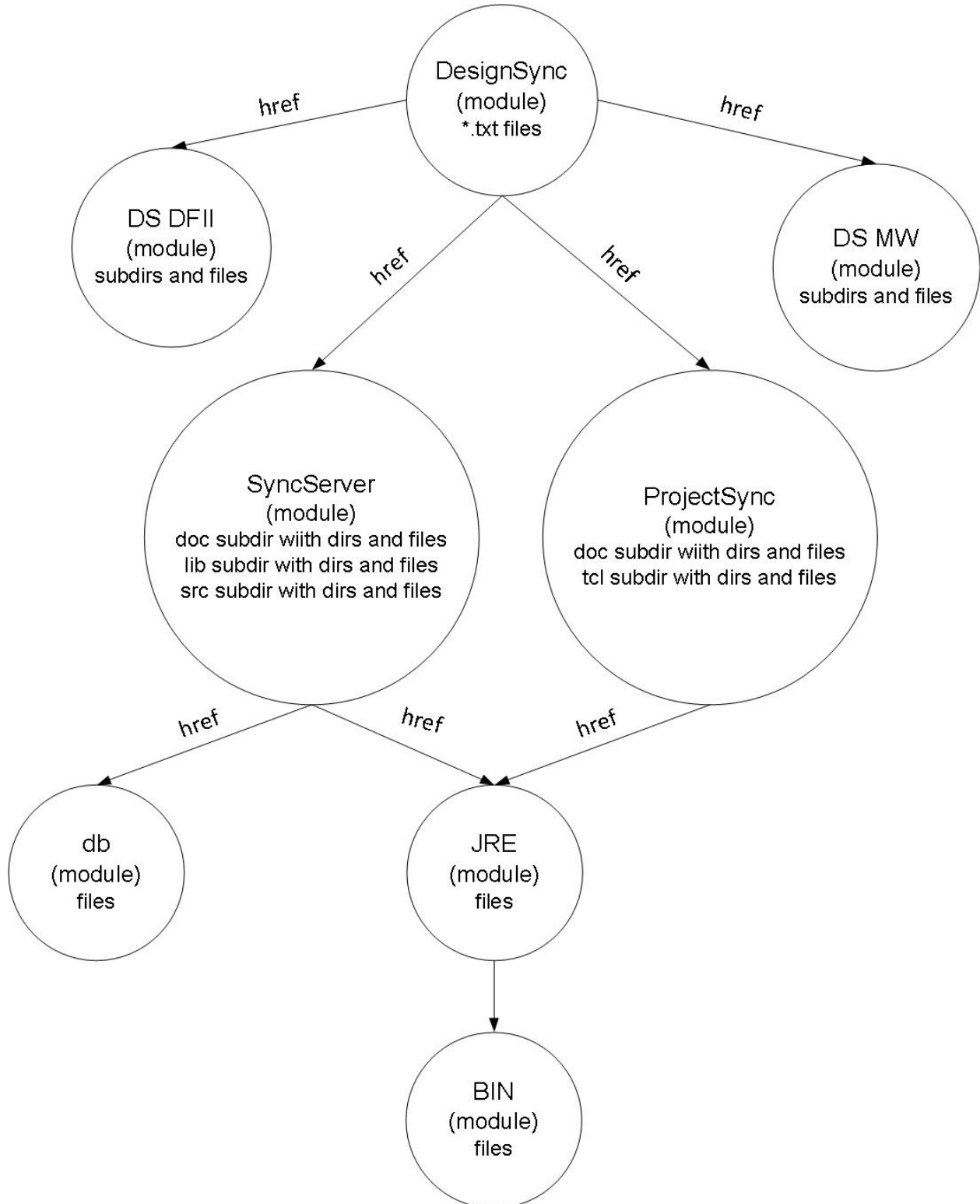


If you specify "JRE" to the -hreffilter option of populate, it will omit populating the JRE href wherever it is referenced, and any submodules beneath JRE, such as BIN pictured in the image above, will also not be populated.



**Hierarchical Href Filtering**

A hierarchical href filter filters at a specified level of a hierarchy, for using our initial example, you see that two modules in the hierarchy refer to the same submodules:



Rather than populate two copies of the JRE module, one as a submodule to SyncServer and the other as a submodule to ProjectSync, the user can use a hierarchical href filter to filter out one of the JRE submodules. In this example, if the user was working primarily in the ProjectSync module, she could filter out the duplicate module from the SyncServer module by specifying "SyncServer/JRE" as the value of the Href filter option during the populate.

### Notes:

- To avoid populating hierarchical references unilaterally, populate without the Recursive option selected.
- Href filtering is always exclusion based, unlike the standard Filter option which can be used to both exclude or include objects.

### Related Topics

[ENOVIA Synchronicity Command Reference:addrhref](#)

[ENOVIA Synchronicity Command Reference:rmhref](#)

[ENOVIA Synchronicity Command Reference:populate](#)

## Folder Versioning

Conceptually, *folder (or directory) versioning* preserves the state of a folder, such that the folder's structure and file content at that moment can always be retrieved.

A folder, like a file, is a module member. A folder can be:

- Added to a module (by using the `add` command followed by the `ci` command)
- Removed from a module (by using the `remove` command)
- Moved within a module (by using the `mvmember` command)

Any of the above actions cause a new version of the module to be created. Consequently, you can always retrieve the state of a folder (its structure and content), based on the version number (or version tag) of the module to which the directory belongs.

Note that when a folder is moved in a module, all of the contents of that folder are also moved. Similarly, when a folder is removed from a module, all of the contents of that folder are also removed.

The **mvmember** operation changes the natural path of an object, from this module version forward *only*.

#### Animated Examples

- Moving a file
- Moving a folder

#### Related Topics

Data Management of Modules

ENOVIA Synchronicity Command Reference: add

ENOVIA Synchronicity Command Reference: ci

ENOVIA Synchronicity Command Reference:remove

ENOVIA Synchronicity Command Reference: mvmember

## Module Branching

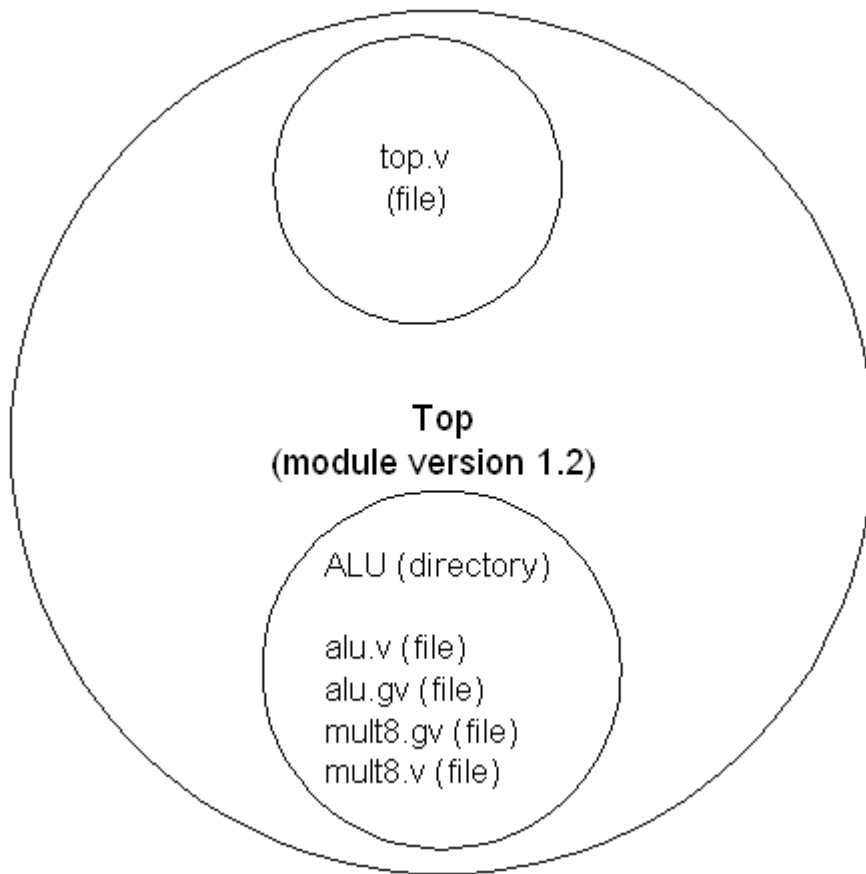
When working with module data, it is a module object that is version-controlled. For background, see Data Management of Modules. As with ordinary DesignSync data, a module can be branched.

A module always has a branch 1 (the default `Trunk` branch) and an initial version 1.1 on branch 1. (Version 1.1 of the module results from creating the module, using the **mkmod** command.) New versions of the module are created as the module's content changes. See *What Is a Module?* for details.

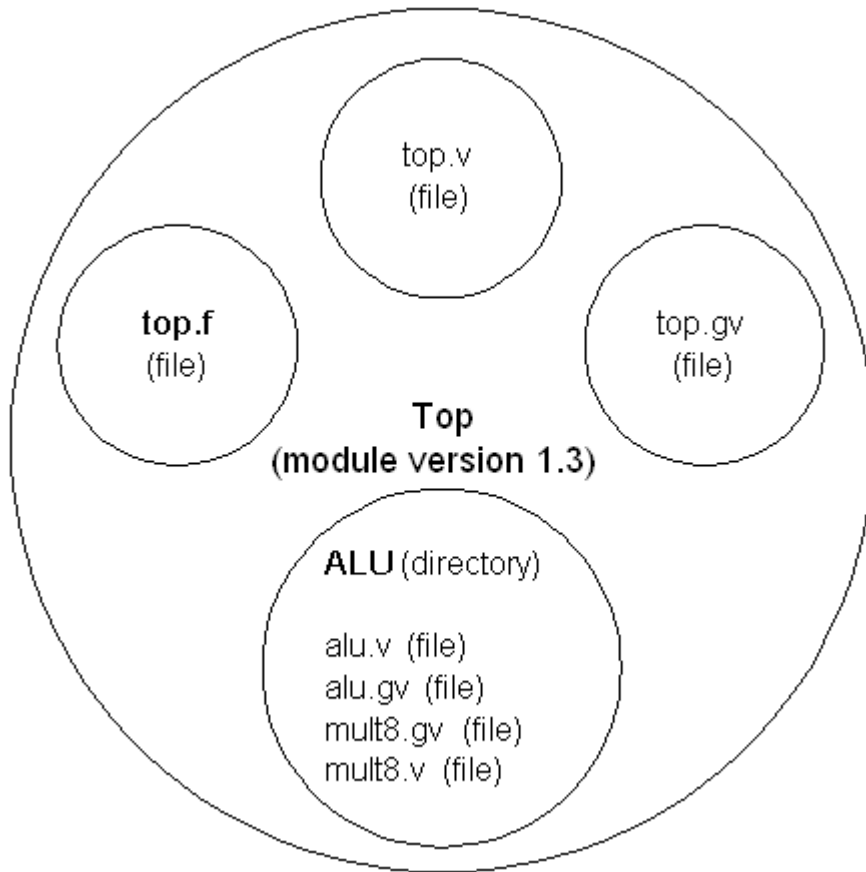
A side branch can originate from any version of a module. The content of the module version from which the side branch originates is used as the content of the initial module version on the new side branch.

For example, let's say there are versions 1.1, 1.2, and 1.3 of the module `Top`. Version 1.1 of the `Top` module does not have any content (version 1.1 resulted from **mkmod**).

Version 1.2 of the `Top` module contains:



Version 1.3 of the Top module contains:

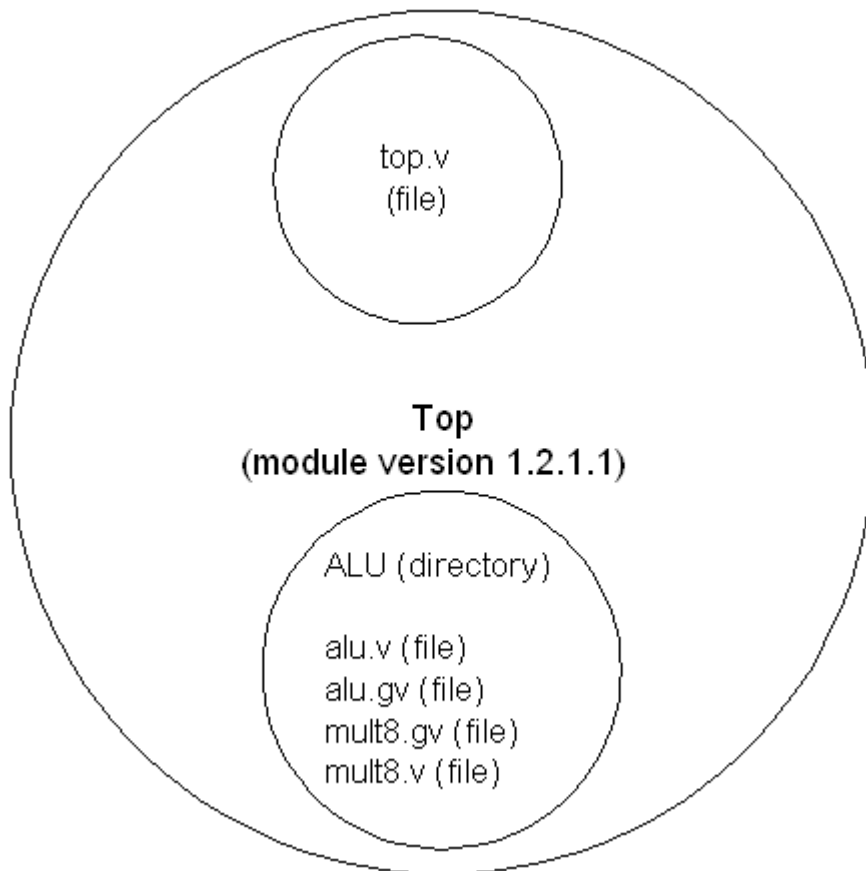


Note that although all of the files shown in module version 1.2 are also shown in module version 1.3, the content of those files might differ between the two module versions.

Using the **mkbranch** command on version 1.2 of the Top module creates:

- Branch 1.2.1 of the Top module
- Version 1.2.1.2 of the Top module

The initial version (1.2.1.2) on the new branch (1.2.1) has the same content as the module version from which the side branch originated:



The initial version on the new branch also contains the hierarchical references that are in the version from which it was branched.

### **Animated Examples**

Branching a module

### **Related Topics**

Module Merging

ENOVIA Synchronicity Command Reference: mkmod

ENOVIA Synchronicity Command Reference: mkbranch

## **Merging Module Data**

Merging combines changes made within a branch. General concepts pertaining to merging are described in these topics:

- Merge Conflicts
- Two-Way Merge
- Three-Way Merge
- Merge Edges

In the above topics, examples of merging text files are used to illustrate their concepts. The concepts are applicable to both file merging and module merging. These additional concepts pertain to just module merging:

- Module Merge Edges
- Unique Identifiers
- Module Structural Changes
- Merge Types and Forms
- In-branch\_Merging
- Cross-Branch Merging

### Module Merge Edges

The Merge Edges topic shows how merge edges are created and used. When merging module data, merge edges are created for individual member objects. This helps with further merges (of the member file contents) at a later date. These merge edges take part in the individual file merges of the members.

Merge edges for the module are automatically created the merge is a in-branch merge and the entire module is the merge candidate. The merge edge is created when a check-in of the entire module is performed, and if an auto-merge does not take place upon check-in. In other words, the merge edge is only created if the workspace version at the time of the check-in is the Latest version on the branch. Module-level merge edges help with later re-merges of the module.

Merge edges for the module can be manually created after the changes resulting from a cross-branch merge have been checked in. To create a merge edge, use the `mkedge` command. If the merge edge is no longer valid or needed, you can remove it with the `rmedge` command.

### Unique Identifiers

Every object and folder in the vault of a module is assigned a unique identifier. These unique identifiers are used within the module's versions. There is a mapping from the unique identifier to a *natural path*. The natural path is the path where that object is placed under the module base directory. The natural path can change from one module version to another, as a result of using the `mvmember` command. Changing the natural path in this manner does not change the object's unique identifier.

### Module Structural Changes

A module typically consists of many directories and files. A module can also refer to its sub-modules by using hrefs. We refer to a set of directories, files, and hrefs as a module structure. Users can modify a module's structure by adding, moving, and removing directories and files, and by adding, and removing hrefs. Versions of a module can therefore have different module structures, with module merging involving the merging of structural changes.

### Merge Types and Forms

There are two primary types of merging that take place in the module environment. Both use the `-merge` option to the **populate** command:

#### Merging of file contents

For example, let's say you modified version 1.4 of `file.txt` in module version 1.8. Someone else checked in version 1.5 of `file.txt` as part of module version 1.9. You want to merge their changes to `file.txt` with your changes.

Module member file content will be merged, similar to when merging non-module managed files. The **diff** command can be used to compare individual objects from different module versions. Also see [Comparing Modules](#), for how to compare the contents of different module versions.

#### Merging of structural changes

For example, suppose that on your development branch, a file that was originally populated with path `/dir/a`, was renamed (by using the `mvmember` command), to be populated at `/dir/b`. You want to merge that change onto your development branch.

An object that was renamed in the module version being fetched will be fetched into a new position in the workspace (`/dir/c` in this example). Any local modifications to the file at the old position in the workspace (`/dir/b` in this case) are merged into the fetched file (`/dir/c`), and the old file (`/dir/b`) removed.

As another example, let's say an object is removed from a module (by using the `remove` command), so that it is not in the module version that is being fetched. The merge operation will remove the object from the workspace. This may require the `-force` option to the `populate` command, if the workspace object is locally modified.

### In-Branch Merging

Merging a module version within a branch is performed using the `populate -merge` command. This fetches the version of the module associated with the module selector, and merges into that any changes in the workspace files. After the merge, the workspace has the module version associated with the module selector, with some members modified.



### Examples

Let's say you have version 1.2 of module ROM in your workspace, from using the selector `Trunk:Latest`. You modified the files `verilog/rom.v` and `DOC/Rom.doc`, and want to merge in the changes that are in the Latest version of the module. The Latest version of the ROM module is 1.3. Version 1.3 contains a new version of `verilog/rom.v`. `DOC/ROM.doc` has not changed in the Latest version.

You run:

```
populate -merge ROM
```

Without the `-merge` option, the `populate` would fail on the `verilog/rom.v` object, because the object is modified. With the `-merge` option, your changes to `verilog/rom.v` are merged into the file being fetched. If there are conflicting changes, those are reported in the output from `populate`. After resolving any conflicts, you can check-in your modifications, creating version 1.4 of the ROM module. Version 1.4 contains new versions of both `verilog/rom.v` and `DOC/RAM.doc`. You now have the Latest version of ROM, version 1.4, in your workspace. This is demonstrated in this animated illustration.

Next, someone else creates the file `verilog/rom_sub.v`, and checks it into the ROM module, creating version 1.5 of ROM. In your workspace, you also create a file named `verilog/rom_sub.v`. The `verilog/rom_sub.v` is either unmanaged, or in the "Added" state, from your having used the `add` command. You also created a local file named `verilog/rom2.v`. `verilog/rom2.v` is in the "Added" state, from your having used `add`.

Attempting to `populate -merge` fails on the `verilog/rom_sub.v` file. You delete the local `verilog/rom_sub.v` file, after which a `populate -merge` succeeds. The `verilog/rom2.v` file is not a member of the module version being merged, so remains in the "Added" state. A subsequent check-in creates version 1.6 of the ROM module, with the added `verilog/rom2.v` file. You now have the Latest version of ROM, version 1.6, in your workspace. This is demonstrated in this animated illustration.

### Cross-Branch Merging

Merging a module version from a different branch is performed using the **`populate -merge -overlay <selector>[...]`** command. This fetches the version of the module specified by overlay selector(s) and merges those into the workspace. Unlike In-Branch Merging, cross-branch merging does not automatically accept most structural changes into the new version of the module. Because of the complexity of the module merge across branches, you must review the changes and determine whether to accept them. Structural changes are processed as follows:

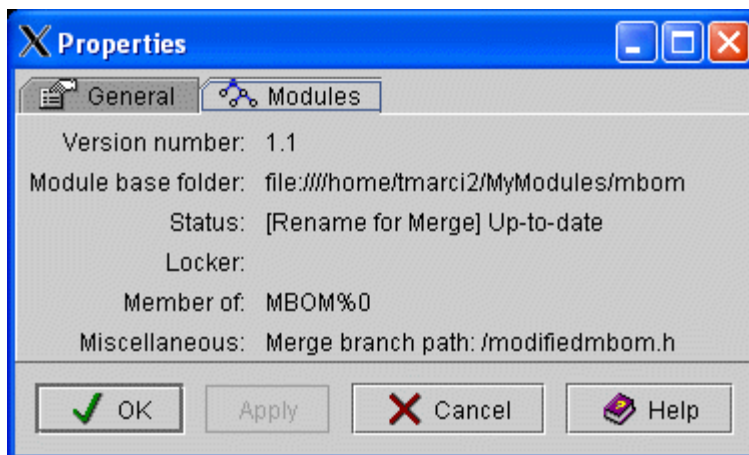
- **Removed objects** - If an object is present in the local workspace, but has been removed on the merge version, it is marked with a `ci_remove` property to indicate that it was removed from the branch. If you want to remove it from the merged

module version, you must manually remove the file from the workspace before creating the new module.

- **Removed objects** - If an object is present in the local workspace, but has been removed on the merge version, it is marked with a `ci_remove` property to indicate that it was removed from the branch. If you want to remove it from the merged module version, you must manually remove the file from the workspace before creating the new module.
- **Added objects** - If an object is present in the merge version, but not in local workspace, it is added to the module and is checked into the module when the next checkin operation on the module or the module member is performed.

**Note:** This is the only automatic operation performed by the cross-branch module merge.

- **Renamed objects** - If an object has a different natural path, meaning that it was moved or renamed, the module member in the workspace retains the same name or location in the workspace, and a metadata property is added to the object to indicate the new path name. To determine what files have been moved, review the populate status information, log file, or run the `ls` command with the `-merge rename` option. To see the name of the object on the merge branch, review the populate command output, log file, or Miscellaneous property listed on the Properties page for the object.



**Note:** If a file marked as renamed is subsequently renamed again, or removed from the module, the metadata property indicating that the file was renamed by merge may persist. To clear the property, perform the `mvmember` or `remove` command on the workspace object, or manually clear the property using the `url rmprop` command.

- **Added or Removed hierarchical references** - Hierarchical reference changes cannot be merged. You must manually adjust your hierarchical references.

After the merge, the workspace has the module version associated with the workspace, with some members modified. To incorporate the changes, check in the module to create a new module version.

### **Related Topics**

[Module Merging](#)

[Auto-Merging Check-in](#)

[Overlaying Module Data](#)

[Module Objects Properties](#)

[ENOVIA Synchronicity Command Reference: populate](#)

[ENOVIA Synchronicity Command Reference: diff](#)

[ENOVIA Synchronicity Command Reference: add](#)

[ENOVIA Synchronicity Command Reference: remove](#)

[ENOVIA Synchronicity Command Reference: mvmember](#)

[ENOVIA Synchronicity Command Reference: lock](#)

[ENOVIA Synchronicity Command Reference: rmfile](#)

[ENOVIA Synchronicity Command Reference: ci](#)

[ENOVIA Synchronicity Command Reference: unlock](#)

[ENOVIA Synchronicity Command Reference: ls](#)

[ENOVIA Synchronicity Command Reference: url rmpop](#)

[ENOVIA Synchronicity Command Reference: mkedge](#)

[ENOVIA Synchronicity Command Reference: rmedge](#)

## **Module Merging**

When working with module data, it is a module object that is version-controlled. For background, see [Data Management of Modules](#). The contents of each version are the files, folders, and hrefs that make up that version. As with ordinary DesignSync data, a module can be branched.

Module branching is used to allow a parallel development of several variants of a module. There are three situations where merging is needed:

- You have a non-Latest version of a module in your workspace, and you want to check in your local changes to the module. This requires auto-merging.
- You want to merge a module version of the same branch as that of your workspace into your workspace. This is in-branch merging.
- You want to overlay a version of the module from a different branch, over the module version that is in your workspace. This is cross-branch overlaying.
- You want to merge a version of the module from a different branch as that of your workspace into your workspace. This is cross-branch merging.

### Auto-Merging

In this scenario, you want to **ci** your local modifications to a module. The module version fetched into your workspace is not the `Latest` version on the target branch. You do not want to always specify the `-skip` option, as that would skip over changes that were made to the same files that you are checking in. The `ci` can proceed, if there is **no** overlap between the objects that are participating in the checkin, and the changes between the version that was fetched into your workspace and the `Latest` version of the module. This capability is *auto-merging*.

In other words, as long as you are working independently from others, on different objects, you can still check in even though someone else has created a new module version with their own independent changes. Note that this auto-merging happens at the file level; the contents of files are not automatically merged.

Auto-merging also applies to other commands that can create a new module version: **mvmember**, **remove**, **addhref**, and **rmhref**. These commands are also allowed, if there is no overlap between the objects being operated on and the changes between the current and latest versions.

When an auto-merge operation occurs, the version number of the module in your workspace is not updated, since there might be changes in the intermediate versions that are not reflected in your workspace copy of the module. A subsequent **populate** is required to update the version number of the module in your workspace.

For further details, see Auto-Merging.

### In-Branch Merging

You merge a module version within a branch using the `-merge` option to the **populate** command. In-branch merging fetches the version of the module associated with the module selector, and merges into that any changes in the files in your workspace, leaving the workspace with the fetched module version, but with some members modified.

For example, let's say you're working with the module `Chip`, using the selector `Trunk:Latest`, and you have version 1.5 in your workspace. You have modified the files `tmp.txt` and `verilog.v`, and want to merge in the changes that are in the Latest version of the module.

The Latest version of the `Chip` module is 1.6. Version 1.6 contains a new version of `verilog.v`, but no newer version of `tmp.txt`. If you were to simply populate:

```
populate Chip
```

the `populate` would fail on the `verilog.v` object, because the object is modified.

If you instead merge:

```
populate -merge Chip
```

DesignSync merges your changes to `verilog.v` into the `verilog.v` file being fetched into your workspace.

If there is a file in your workspace that is either unmanaged or added with the `add` command, and that file has also been added to the version of the module you are merging with, a conflict occurs. In this case, an error is reported for that file, similar to an attempt to `populate` without `-merge` when a file is modified.

If there is a file in your workspace that you added with `add` and that file is not a member of the merged module version, that file remains in an “Added” state; it will be added to the module by your next `ci` to that module.

For further details, see [In-Branch Merging](#).

### Cross-Branch Overlaying

When data is overlaid across branches, DesignSync fetches objects from the requested branch, and overlays them onto the existing equivalent objects. There is no merging.

The command used for performing cross-branch overlay is:

```
populate -overlay <branch:Latest>
```

In this case, there can be no conflicts, as a conflict is, by definition, related to a merge.

For further details, see [Overlaying Module Data](#).

### Cross-Branch Merging

When data is merged across branches, DesignSync fetches the objects from the requested branch and merges them with the equivalent objects from the branch already loaded in your workspace. You merge two module versions from different versions using the `-merge -overlay` options to the **populate** command. Cross-branch merging fetches the specified version of the module, and merges it into your workspace over the existing equivalent objects. The workspace is still associated with the same module branch as before the merge.

For example, let's say you're working with the module `Chip`, and you have the `Trunk:Latest` version, version 1.5 in your workspace. Independently, you have a set of changes in a branched release that you want to integrate into the Trunk branch.

The branched release of the `Chip` module is `Rel2:Latest`. If you've modified the files as followed:

- Edited `verilog.v`
- Removed `tmp.txt`
- Added `Chip.doc`

You run the following command to merge your module into your current workspace.

```
populate -merge -overlay Rel2:Latest
```

DesignSync performs the following actions:

- Merges your changes to `verilog.v` into the `verilog.v` file being fetched into your workspace.
- Creates `Chip.doc` in your workspace in the added state so it automatically checks in when you check in the module. If you had an unmanaged copy of `Chip.doc` in your workspace, or used `add` to add the file to the module, a conflict occurs and DesignSync identifies the conflict to allow you to determine which version of `Chip.doc` you want to keep in the workspace.
- Identifies in the output message that `tmp.txt` was removed in the branch module and marks the `tmp.txt` file as removed in the metadata. This allows you to determine whether you want to remove the file, or leave it in the module. If you want to remove the file, you must manually remove it from the workspace version and check in the module. This removes the file from the module version on the server.

If there is a file in your workspace that you added with `add` and that file is not a member of the merged module version, that file remains in an “Added” state; it will be added to the module by your next `ci` to that module.

### Related Topics

Module Branching

## Merging Module Data

ENOVIA Synchronicity Command Reference: ci

ENOVIA Synchronicity Command Reference: mvmember

ENOVIA Synchronicity Command Reference: remove

ENOVIA Synchronicity Command Reference: addhref

ENOVIA Synchronicity Command Reference: rmhref

ENOVIA Synchronicity Command Reference: populate

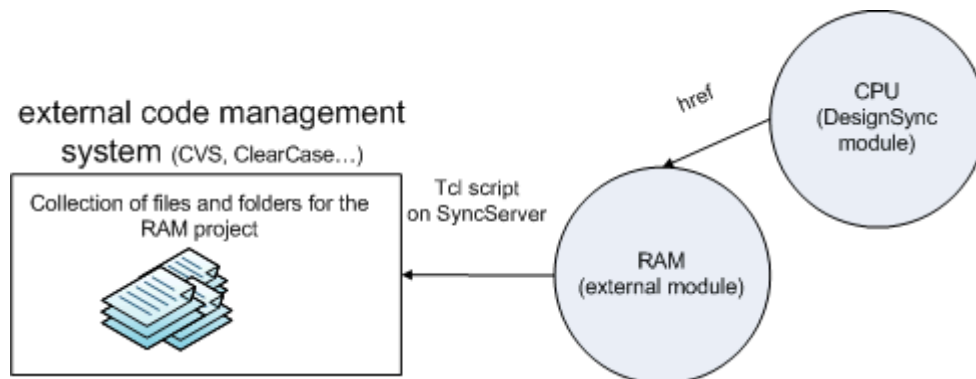
ENOVIA Synchronicity Command Reference: add

## External Modules

DesignSync supports the ability to create a hierarchical reference from a DesignSync module to an external module. An external module is an object or collection of objects managed by another code management system exposed as a module through DesignSync. Using external modules allows you to manage code dependencies between module objects in DesignSync and files checked in to other code management systems.

### Using external modules in a design hierarchy

The external module must be part of a module hierarchy. You cannot create an external module as a top-level module. The external module must be a leaf on the module hierarchy. DesignSync can only populate the external module itself. The external module may have not have references to sub-modules. The external module cannot overlap with any other module data. It can only contain objects from one external module. To illustrate, let's look at a module hierarchy the includes an external module.



In this example, the DesignSync module CPU depends on code managed by another team that uses a different code management system. The DesignSync administrator creates a Tcl script which is placed on the SyncServer containing the information needed to connect to the external code management system and extract the information using the correct options for the external system. The Project lead creates a hierarchical reference to the external module which uses the Tcl script to create a module manifest on the server that stores information about the objects that are part of the RAM external module.

When the workspace is populated recursively, the RAM external module is populated like any other module, according to the settings defined in the Tcl script containing the definition of the external module and the settings selected by populate. Practically, this means you can apply properties, such as a filter, either as part of the hierarchical reference or as part of the populate.

This flexibility allows the Project lead to exclude or include only specific files that the team will need, and, if specific team members wish to apply additional filtering, they can do so in the populate.

Once an external module has been populated, other commands can be available to operate on the external module data. External modules supports the DesignSync command-line commands:

- Populate
- Tag
- Showstatus
- Showhref
- ls
- Enterprise Synchronization
- Swap Replace/Restore
- Rmmod

**Note:** The DesignSync commands that are extended to work with external modules must have access to commands from the external CM system. This means that the external CM system commands must be installed on any user's machine that runs DesignSync commands that in turn work on external module data.

### Creating External Modules

Within a standard DesignSync module, you create a hierarchical reference that refers to an external module. The external module reference contains a pointer to a Tcl script that provides the information about the external code management system that allows DesignSync to use those objects. For information about creating the Tcl script, see the DesignSync Administrator's Guide: Defining the External Modules Interface.



After creating the href to the external module, you populate it exactly as you would any other href, by specifying the href name argument to populate or populating the parent module with the -recursive option.

## Viewing the Contents and Status of External Modules

DesignSync supports viewing the contents, status, and hierarchical reference status of external modules. Performing these operations will show you how the external module available in DesignSync differs from the source module in its different code management system.

The user runs the DesignSync command with a list of desired options, specified with -xtras to the external code management system. The external module in the workspace contains a pointer to a Tcl script that provides the information about the source for the external module.

### Related topics

Module Hierarchy

Creating Module Hierarchy: Overview

Creating a Hierarchical Reference

Populating Your Work Area

DesignSync Administrator's Guide: Overview of External Modules

DesignSync Administrator's Guide: Defining the External Modules Interface

## Module Member Tags

Module member tags (sometimes called, "snapshots," in the documentation for clarity) are a collection of versionable module members that are tagged from a workspace. When you tag a set of member versions in a workspace, you create a new module member tag branch on the server. Using a branch allows you to maintain a snapshot as a versionable object, updating tags and hierarchical references as needed.

Module member tags allow you to capture a subset of a module workspace at any given moment in time, and recreate it. This can be useful to preserve a specific set of files for testing or releasing that set of files without interrupting the normal development workflow.

When you create a module member tags by tagging the desired module members, DesignSync creates a special tagged branch for the module. When you create the snapshot, you provide a tag name; the module branch is created with the name `SNAPSHOT_<tag_name>`. The specific module member tag version is `<tag_name>`.

Operations on tagged module member tagged versions are always workspace-centric. This means the operations occur on the objects loaded in the workspace. If a folder is specified with recursion, the operation traverses the folder.

The module member tag operations are atomic with respect to the server. In order to execute the tag operation, all objects within a module must be processed successfully. If any object fails the entire operation fails for that module. For example, if you tag module members in your workspace belonging to different modules and you do not have tag access for one of the modules or module members, the tag fails for that module only. The other modules, assuming no other errors within them, are updated successfully.

The module member tag operations are not atomic with respect to the workspace. For example, if you have a moved, removed, or added a file that has not been checked in, it does not cause the entire tag operation to fail. You receive an error message for any individual workspace object that failed, and the operation itself succeeds.

Hierarchical references within module member tagged versions must be manually added or removed. DesignSync does not automatically include hierarchical references already in the workspace in a new tagged version, nor does it update hierarchical references in the module member tagged version when it is versioned by adding or removing tags. After the module member tagged version has been created, you can add the desired hierarchical references to the module member tagged version, and update, remove, or add new hierarchical references as needed.

Operations that can create a module version with structural or content changes, such as `add`, `remove`, `checkin`, `mvmember`, `rollback`, and `populate` with the `-lock` option, are not allowed with on a module member tagged branch. These tagged module member versions are intended to be used as is, with content frozen. The only operations allowed are Creating a Hierarchical Reference, Deleting a Hierarchical Reference, and Tagging operations (adding, removing, or moving tag names on module members). This allows you to create the perfect, immutable, test or release version.

### **Populating Module Member Tagged Versions**

After the module member tag has been created, you can populate the version into a local workspace for viewing, testing, or integrating into other work.

When you populate a module member tagged version as a fixed workspace for viewing or testing, you use the snapshot tag as a selector. This can be either the full snapshot branch and version name or the simple tag name. When you populate a snapshot

module, you can update tags on module members or hrefs within your workspace, but cannot checkin any content or other structural changes to the module members or the module.

When you populate a module member tag to integrate with other work, you typically populate with a comma separated list of selector list ending with the default selector. This populates from the default selector list first and replaces any matching objects with the member objects from the selected versions.

This results in a workspace that uses the default or main selector as the base and the destination for any checkins, but some or all of the module member objects from the snapshot workspaces. For example using the following version to populate:

```
Beta,Alpha,Trunk:Latest
```

The Populate command creates a module manifest from the main selector, Trunk:Latest, and layers that with the contents of the Alpha blend, and then the Beta blend. The final manifest is then sent to the client. The server uses the natural path of the objects and the uuid to determine which module members to replace.

When hierarchical references are populated non-recursively as part of the operation, the hierarchical reference versions come from the main selector list, not from the specified module snapshots.

When the hierarchical references are populated recursively during the initial populate the module members within the populated submodules also inherit the selector list. If hierarchical references are not populated recursively during the initial populate, they will not overlay member items from the selector list on subsequent populates; they will only contain the objects from the main selector. You can set or change the selector recursively by Populating Your Work Area with the selector list as the Version option (or using `populate -version`) to specify the selector list) or using Set Vault Association.

#### Notes:

- If the main selector list is a snapshot branch, you will not be able to check in any changes from the workspace.
- When populating a selector list, the module member objects in the specified snapshot are populated instead of the objects from the main selector. Populate will never attempt to merge the members. If you want to merge data from a module snapshot into your workspace, you should populate your snapshot with the `Merge_with_workspace` and `Overlay_version_in_workspace` options into a workspace that has the default selector defined as the desired destination for checkin.
- Any hierarchical references that are defined as a static module version indicated by the selector on the href will not inherit from the selector list, even if the initial populate was recursive through the module hierarchy.

### Related Topics

Populating Your Work Area

Tagging Versions and Branches

Creating a Hierarchical Reference

Deleting a Hierarchical Reference

Selector Formats

Specifying the Vault Location for a Design Hierarchy

## Edit-In-Place Methodology

A module version that has been populated by an href can be manually replaced by another module version, using the swap commands. This is analogous to allowing a brick to be removed from a wall and replaced (in the same location) with a different version of the same brick. A primary use of this edit-in-place methodology is to replace a statically fetched sub-module within a baseline (i.e. static) module hierarchy with the latest version on a branch so that the sub-module can be developed within a baseline framework.

The swap capabilities:

- Change the selector of a sub-module already present in the workspace and re-populate it using the new selector.
- Avoid reverting the sub-module via a recursive populate of a parent module.

This results in a workspace in which a sub-module can be replaced with a different version of the same module and developed/tested within the surrounding framework of other modules that define a released hierarchy.

swap replace replaces the version of a module in the workspace with a different version of the same module. The replace operation updates the selector and href mode, and calls populate recursively to replace one version of a workspace module with another version of the same module. The populate operation uses all persistent populate controls (such as filters).

swap show shows the currently swapped module versions in the workspace. This information is useful when an end user needs to know what modules have been updated for development and test.

`swap restore` restores a previously swapped module to the version defined by a parent module in the workspace. The restore operation calls `populate` recursively using all persistent `populate` controls (such as filters).

#### **populate of a swapped sub-module**

The `swap replace` and `swap restore` commands always perform a full recursive `populate`, applying the full mode to the entire hierarchy of the swapped module. The `populate` operation uses the selector and `hrefmode` specified to the `swap replace` command, rather than using the selector and `hrefmode` determined by the parent. The edit-in-place methodology changes the selector and `href` mode as necessary to ensure that the desired sub-module versions are replaced in the workspace. Persistent settings (such as filters) associated with the original module version will be applied to the new swapped module version.

**Note:** Normal `href` mode for the swap commands uses the value of the **Change traversal mode with static selector on top level module** setting.

If all of these conditions are met:

1. the module being swapped is an mcache link or the `swap replace/restore` command is issued in `-force` mode.
2. the default mcache mode is to link to modules in the module cache
3. the specified selector is static
4. the modified selector resolves to a module version found in a module cache

Then the `populate` operation will replace the existing mcache link or file copies with a new link pointing to the new version of the module.

**Note:** The `populate` command can replace fetched module instances with mcache links if the `-force` option is used and all other conditions are met.

If the selector of a fetched module instance is modified the `populate` command will not replace the existing module instance with an mcache link even if all other conditions for mcache linking are met. Instead, the `populate` command will refetch the module instance using the modified selector.

When `populate` is run in verbose mode, its output indicates when the selector of a swapped sub-module is being used.

#### **ci of a swapped sub-module**

The `ci` command will not check in a module with a static selector. For a recursive operation, the `ci` command will continue following the module hierarchy in the workspace looking for modules that can be checked in (e.g. swapped modules with dynamic selectors).

During a recursive ci, the href from a current parent module version to a swapped module is carried over to the new parent module version without change. I.e., The href to a swapped module is not updated. The purpose of swapping a module is to develop and test it within a module framework, not capture new versions of the parent module that reflect a static hierarchy containing the swapped module versions. Whoever is responsible for integration will capture new versions of the parent module.

The output from ci indicates when modules are not checked in because they have static selectors. The output also indicates when hrefs are not updated because their selectors do not match the actual selector of the sub-module in the workspace.

### **Related Topics**

Module Hierarchy

## **Understanding Smart Module Detection**

DesignSync features the ability to identify the module to which a new object should be added or checked in without forcing the user to explicitly specify the target module.

### **Identifying the module target**

The DesignSync smart module detection algorithm identifies the target module by the following rules:

1. If there is only a single workspace module populated into the workspace at or above the object being added, that module is identified as the only possible target module.
2. If there are multiple workspaces modules above the object being added, but the folder containing the object is a member of only a single module either explicitly, by having been added to the module, or implicitly, by only containing objects belonging to a single module, that module is identified as the only possible target.
3. If there are multiple workspace modules above the object being added and the containing folder contains objects belonging to multiple modules, all the workspace modules are candidate module. If the user has defined a moduleSelectionHook, it is used within a client trigger to determine the workspace. If there is no moduleSelectionHook defined, smart module detection fails and the operation fails with an appropriate error message.

### **Related Topics**

Adding a Member to a Module

Checking in Design Data

## ENOVIA Synchronicity DesignSync Data Manager Administrator's Guide: Module Selection Hook

### Conflict Handling

Because DesignSync provides a client/server environment, it is possible that conflicts can arrive between the version locally and the version checked into the server. This topic shows you how to identify module conflicts both at the structure level and on individual members.

Module Structure Conflict Handling

Module Member Conflict Handling

#### Module Structure Conflict Handling

When the structure of a module has changed both on the server and locally, this can result in a structure conflict. A structure conflict can be seen by including report modes for both server and workspace in your view (this is the default value of the Status column in the GUI) or by running the `ls` command with both

In this example, the file, `regs6.v` has been independently renamed (moved) on both the server and the workspace.

Name	Result	Type	Status
Stack_pointer%0		Module	
inc_dec_sp.gv		File	Up-to-date
inc_dec_sp.v		File	Up-to-date
reg6.gv		File	Up-to-date
regs6.v	✓ Already Fetched and Unmodified Version 1.1	File	Workspace Status: Locally Moved Needs Merge   Server Status: [Moved]

The `ls` command output when specified with report modes including workspace (`-report +D`) and server (`-report +S`) status shows the same information.

```
regs6.v 10/17/2011 13:35 Moved Needs Merge [Moved] 1.1
```

To find out the new object path on the server, you can use the Vault Browser or the Version History with the module manifest mode selected, or the `vhistory` command with the `-report Q` option.

When you have a module structure conflict, you can resolve it by forcing an overwrite of modified changes in your workspace or by forcing a checkin of your workspace. For more information, see Resolving Module Structure Conflicts.

## Module Member Conflict Handling

Because modules can overlap in a workspace, it is possible that a populate will try to fetch a file, and find that there is a file from another module already in the target workspace address. For example, let's say modules `Chip` and `ALU` are both fetched to the module base directory `basedir`. Both contain the file `README.txt`. When the second module is fetched, it will find that there is already a `README.txt` from the other module. This issue is referred to as a *populate conflict*.

A populate conflict is handled as follows; the first module to populate a file will win. Subsequent populates of other modules with conflicting objects will lead to the conflicting object being reported as a failure to fetch, thereby increasing the failure count of the populate.

### Related Topics

Populating Your Work Area

Merge Conflicts

Merging Module Data

Overlaying Module Data

*ENOVIA Synchronicity Command Reference:populate*

## Module Version Updating

The *fetches version* number is the version number of an object that is in your workspace. If a new version of a module is being fetched, the *fetches version* number is updated at the end of a successful **populate** command. If the command is not completely successful, then the fetched version number is not updated.

Other scenarios to consider are:

- If a member could not be fetched because it is already in the workspace, and is modified, then the fetch of that member fails. Consequently, the overall operation fails, and the module's fetched version number is not updated.
- If a member is skipped due to:
  - the `-filter` option
  - the `-exclude` option
  - the `-nonew` option

then that member is not fetched. Therefore, the workspace does not contain the full module version, so its fetched version number is not updated.



- If a member is due for removal from the workspace (because the new module version being fetched does not contain it), and:
  - - the module member (that is due for removal) is modified in the workspace
    - the `-force` option is not used

then that member is unaffected. After the fetch attempt, the workspace has the complete contents of the module, (plus the member that was not removed), so the module's fetched version number is updated.

Not having the Latest version of a module as the fetched version is not necessarily a problem. The auto-merge system ensures that subsequent check-ins can still proceed. However, a potential downstream effect is that the **showstatus** command will continue to show the module as Needs Update, until a successful populate is performed, fetching the Latest version.

#### Related Topics

Populating Your Work Area

Auto-Merging

Merging Module Data

ENOVIA Synchronicity Command Reference: populate

ENOVIA Synchronicity Command Reference: showstatus

## Using a Module Cache

As a performance optimization, your team leader may fetch modules into a module cache on the LAN, for you to link to. This is faster than your fetching data from a server. On UNIX, the use of symbolic links also saves disk space. The link is created to the base directory of the requested module version in the module cache. On Windows, symbolic links are not supported.

When you fetch module data using the Populate dialog box or **populate** command, you can opt to link to a module in the module cache. You can also specify the module cache paths to search for the module data. If the requested module version is not found in the module cache, DesignSync fetches the module from the server.

Your team leader might have defined default module cache paths and a default module cache mode. You can override these values by setting your own default values. See *DesignSync Data Manager Administrator's Guide*: Module Options for more information.

## DesignSync Data Manager User's Guide

Since the copy mode for an mcache is ignored for modules, your team lead should fetch the modules from the server into the mcache using the share mode. This forces the module contents to get fetched into the DesignSync cache (different from an mcache).

Symlinks are created in the mcache to point to these files in the DesignSync cache. If you subsequently use the copy mode for mcaches to get full copies of the module contents from the mcache instead of the server), the '-mcachemode copy' switch is ignored. The populate operation uses the default 'from local' copy mode to fetch the files from the DesignSync cache.

When the module cache is created, it is assigned a workspace instance to allow to the creator to manage it like any other module in their workspace.

When users create links to the module cache, a symbolic link to the base directory of the module in the mcache is created. This link is assigned a module instance and can be managed like any other module in the workspace. The link has the object type "Link to Mcache."

### Related topics

[What is a Module Cache?](#)

[Displaying Module Cache](#)

*DesignSync Data Manager Administrator's Guide: Setting Up a Module Cache*

[ENOVIA Synchronicity Command Reference: populate](#)



# Working with Files and Directories

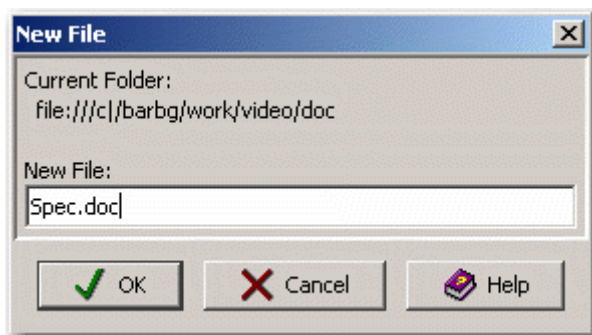
## Creating Files

You can create new files. Files can only be created locally; you cannot create files on the SyncServer.

Select **File => New => File** to bring up the **New File** dialog. Click **OK** to create the new file.

On Windows, the editor associated with the file is invoked, with the file open in the editor. If there is no editor associated with the file's type, the default editor is used. If you need to change the default editor used by DesignSync, you would change it the SyncAdmin Tool. See SyncAdmin Help: General Options for more information.

Click on the fields in the following illustration for information.



## New File Field Descriptions

### Current Folder

This field is not editable. It shows the folder from which the **New File** dialog was invoked.

### New File

Enter the name of the file you want to create. You can specify:

- The file name or a relative path to create the file relative to the Current Folder
- An absolute path

## Related Topics

Deleting Files

## Moving and Renaming Files

You can move and rename local files, both managed and unmanaged, using the **mvfile** command. The DesignSync graphical interface does not support moving and renaming files.

### Related Topics

ENOVIA Synchronicity Command Reference : mvfile Command

Moving and Renaming Folders

## Adding a Member to a Module



The Add to Module dialog box adds highlighted objects to a local module in your workspace. The objects must be within the scope of the base directory of the target module and the module must be populated with a dynamic selector. The objects can be files, directories, or collection objects. The locally added objects are checked into the module version that is created by your next checkin. See Checking In Design Data for more information.

- If the highlighted objects are all files, and the module context can be uniquely determined, the files are added without invoking the Add to Module dialog box. These results are shown in the output window.
- If smart module detection cannot determine the target module, the Select Module Context dialog box is displayed. Once the module context is selected, the results are shown in the output window.
- If the highlighted object contains one or more folder objects, the Add to Module dialog displays.

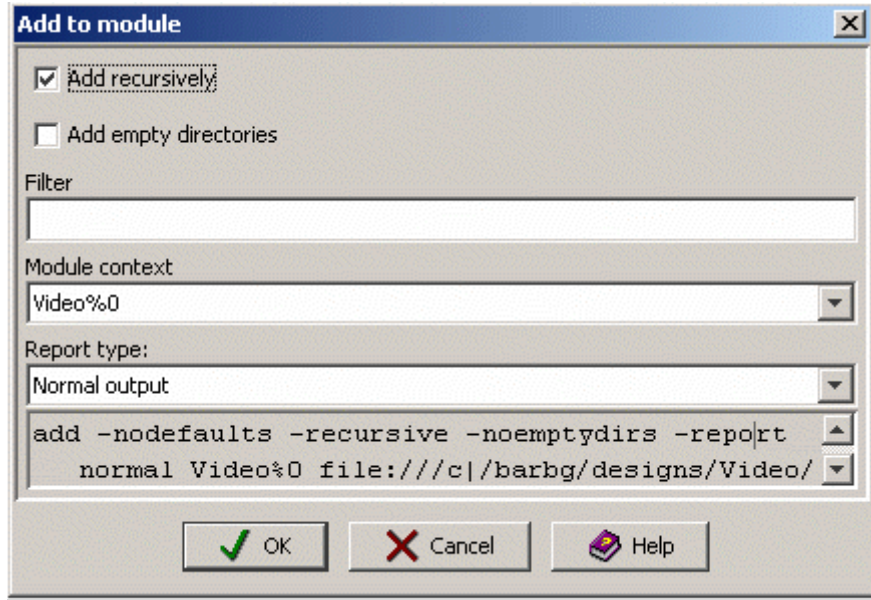
For information about how smart module detection determines the target module, see Understanding Smart Module Detection.

Files excluded from view by exclude files are not displayed by the DesignSync GUI and are not available to Add from the GUI. For more information on exclude files, see Working with Exclude Files.

### To add to a member with Add to Module dialog box:

1. From the main menu, select **Modules =>  Add Member** or you can select the  button from the Module Toolbar. You can also select this command from the context menu.
2. The Add to Member dialog box appears. Select options as needed.
3. Click **OK**.

Click on the fields in the following illustration for information.



### Add recursively

For a folder, whether to add only the folder or also recursively add the folder's contents. Note that folders themselves can be module members. If you checked the Add empty directories option, adding a folder without content results in an empty folder as a module member.

By default, the option to **Add recursively** is not selected.

### Add empty directories

This option is used with **Add recursively**. When adding members recursively, any folder that contains files is added to the module. The **Add empty directories** option specifies whether directories without content are added to the module.

For example, let's say you're recursively adding "dirA" to a module. "dirA" contains files, and an empty subdirectory "dirB". The option to **Add empty directories** controls whether the empty directory "dirA/dirB" is added.

### Filter

Allows you to include or exclude module objects by entering one or more extended glob-style expressions to identify an exact subset of module objects on which to perform the add.

The default for this field is empty.

## Module Context

Expanding the list-box shows the available workspace module instances for the currently selected object or objects, including an automatically calculated <Auto-detect> "module context.". All available workspace module instances are listed alphabetically in the pull-down following the calculated <Auto-detect>.

**Note:** If you select <Auto-detect>, and the DesignSync system cannot determine the appropriate module, the command fails with an appropriate error.

## Report type

From the pull-down, select the level of information you want to display in the output window:

**Brief output:** Lists errors generated when adding objects to the local module, and success/failure count.

**Normal output:** Lists all objects added to the local module, success/failure count, and beginning and ending messages for the add operation. This is the default output mode.

**Verbose output:** In addition to the information listed for the Normal output mode, lists:

- Skipped objects that are already members of the module.
- Skipped objects that are already members of a different module.
- Skipped objects that are filtered.
- Status messages as each folder is processed.

**Errors and Warnings only:** Lists errors generated when adding objects to the local module, and success/failure count.

## Related Topics

Directory Versioning

ENOVIA Synchronicity Command Reference: add

Filter field

Module context field

Command Invocation

Command Buttons

## Context Menu

### Moving a module member

You can move module members using the Move modules members dialog box when:

- One or more current module members or module folders are selected in the client work area or on the server.
- A module base folder in the client work area can be selected if it is also a module member folder for another module.

**Note:** Unless all the selected objects are members of the same module in the workspace or the same module version on the server, you can not invoke the Move module members dialog box.

When moving a workspace member, you must be working with the Latest version of the member present in the workspace. You can modify your copy of the object and move the object before checking the object back in. By default, the **Move Member** command does not check in the modified object, it only changes the path or object name of the specified object in the workspace. When you perform the next checkin operation, the specified object is renamed on the server and any content modifications are checked in.

**Note:** If you deselect the Apply changes locally first; commit with next checkin option the module version created with the when you run the Move Member command does not have the local content modifications. If you select the **Apply changes locally first; commit with next checkin** option (default), the module version created with the checkin operation has both the name change and the updated content.

You cannot move an object that another user has locked. If you move an object you have locked, you retain the lock after the move has completed.

When you move objects that have been added to the workspace but not checked in, the workspace members are renamed and remain in the added state to be processed during the next checkin.

### Some notes on moving folders

#### Moving module folders on the server

The Move module members dialog box is used to move module folders on the server. When a folder is moved on the server, the contents of the folder move with it. However, the following objects do not move with the folder:

- Any module members locked by another user.
- Any non-versioned objects in the workspace.





If all the contents of a folder are moved, the folder is removed from the server, and if the module is populated with the **-force** option, it is also removed from the workspace.

### Moving module folders in the workspace

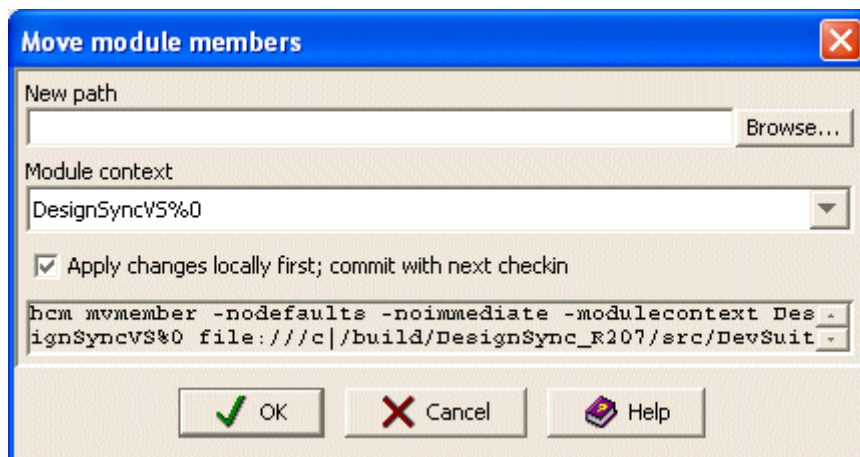
The Move module members dialog box is used to move a workspace folder. When the workspace folder moves, the following happens:

- The contents within the folder (files or collection objects) are moved to the specified location in the workspace. If the Apply changes locally first; commit with next checkin is not selected, the folder contents are moved on the server immediately and DesignSync creates a new module version. Otherwise changes are included in the module version created during the next checkin.
- The folder is removed from the workspace, if it was not explicitly added. The folder is does not move to the new location.

#### To move a module member:

1. Select the member.
2. From the main menu, select **Modules =>**  **Move Member** or select the  button from the Modules toolbar. The **Move module members** dialog box appears.
3. Select options as needed.
4. Click **OK**.

Click on the fields in the following illustration for information.



#### New path

Select the new path for the module member(s) or use the Browse button to select the path.

- If the selected objects were from the workspace, the browse button invokes a browser rooted at the base directory for the module context selected.
- If the selected objects were on the server or rooted at the module version, there is a new folder button on the browse dialog to allow you to create a new folder.
- The browser selection is restricted to folders.
- If a relative path is specified, it is relative to the module base folder on the client and the top of the module on the server; it is not relative to the selected object(s).

**Note:** If the selected objects are locked by other users, the move will fail.

### Module Context

This option is only available when the selected objects reside on the client. Expanding the list-box shows the available module instances for the currently selected module members. All available module instances for the current selected base folder are listed alphabetically in the pull-down.

**Note:** There may only be one module listed. If the selected object is a module base folder, all modules based at this folder are selectable from the list-box.

### Apply changes locally first; commit with next checkin

This option is not available when the move operation is done directly on a server object

When checked, the selected object(s) is marked to be moved during the next module check in. (Default)

When not checked, DesignSync immediately creates a new module version on the server with the selected objects moved.

### Related Topics

The Module Toolbar

Renaming a Module Member

ENOVIA Synchronicity Command Reference: populate

ENOVIA Synchronicity Command Reference: mvmember

## Using the Moving Modules Members dialog box

While you use the same Move module members dialog box to move or renaming a module member, the information you enter for each mode is a little different. If you are moving the location of a module member, see Moving a module member.

If you are renaming a module member, use the instructions below.

## Renaming a module member

You can rename module members using the Move modules members dialog box when:

- One or more module members or folders are selected in the client work area or on the server.
- A module base folder in the client work area if it is also a module member folder for another module

**Note:** Unless all the selected objects are members of the same module in the workspace or the same module version on the server, you can not invoke the Move module members dialog box.

Things to remember about renaming:

- A module context is required to both limit what is renamed and to make it clear what is being renamed.
- Renaming an object in the workspace, by default, does not update the server object until the change is committed by performing a Check in on the module. For more information, see Apply changes locally first; commit with next checkin.
- Renaming a specified a server object directly does not affect any workspaces containing the moved object until that workspace is updated.
- Renaming a folder automatically means renaming all the members in that folder. Renaming affects that whole directory cone.
- Items which have been previously added to a module but not checked in may also be renamed, and remain in the added state. When all items in a rename meet this criteria, no new module version is created, since the only changes are in the workspace

## Renaming folders on the server

The Move module members dialog box is used to rename module folders on the server. When a folder is renamed on the server, the contents of the folder follow it. However, the following objects do not follow within the folder:

- Any module members locked by another user.
- Any non-versioned objects in the folder in the workspace.



If all the contents of a folder are renamed, the previous folder name is removed from the server, and if the module is populated with the **-force** option, the previous folder name is also removed from the workspace.

## Renaming folders in the workspace

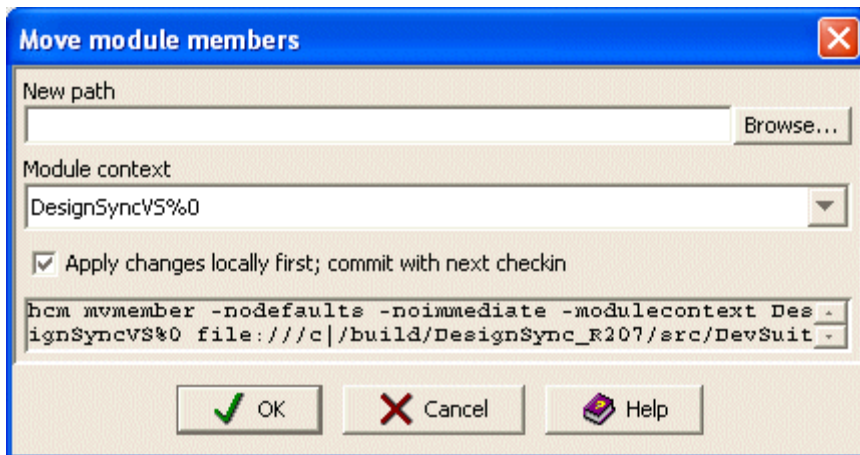
The Move module members dialog box is used to rename a workspace folder. When the workspace module folder is renamed, the following happens:

- The contents within the module folder (files or collection objects) are renamed.
- The previous folder is removed from the workspace.

### To rename a member from a module:

1. Select the member.
2. From the main menu, select **Modules =>**  **Move Member** or select the  button from the Modules toolbar.
3. Select options as needed.
4. Click **OK**.

Click on the fields in the following illustration for information.



### New path

Enter the new path name for the module member(s) or use the Browse button to select the new path name.

- If the selected objects were from the workspace, the browse button invokes a browser rooted at the base directory for the module context selected.
- If the selected objects were on the server or rooted at the module version, there is a new folder button on the browse dialog to allow you to create a new folder.
- The browser selection is restricted to folders.
- If a relative path is specified, it is relative to the module base folder on the client and the top of the module on the server; it is not relative to the selected object(s).

**Note:** If the selected objects are locked by other users, the rename will fail.

### Module Context

This option is only available when the selected objects reside on the client. All the available module instances for the currently selected module members are listed alphabetically in the pull-down.

**Note:** There may only be one module listed if the selected object is a module base folder.

### **Apply changes locally first; commit with next checkin**

This option is not available when a rename operation is done directly on a server object.

When checked, the selected object(s) is marked to be renamed in the workspace immediately and on the server during the next module check in. (Default)

When not checked, DesignSync immediately creates a new module version in on the server with the selected objects renamed and immediately renames the selected objects in the workspace.

### **Related Topics**

The Module Toolbar

Moving a module member

ENOVIA Synchronicity Command Reference: populate

ENOVIA Synchronicity Command Reference: mvmember

## **Creating Folders**

You can create new folders, either locally or on the SyncServer. To create a folder:

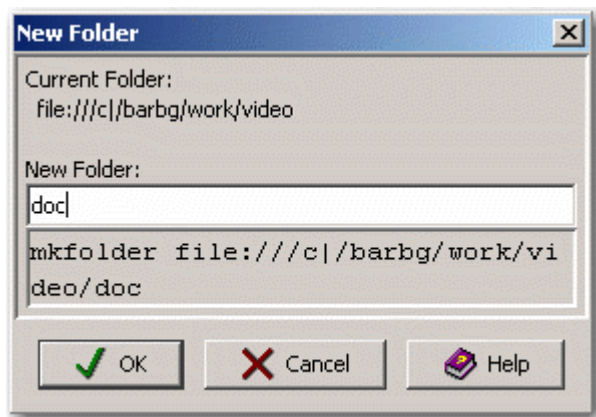
1. Navigate to the folder or SyncServer location in which you want to create a new folder. A folder you specify with a relative path name will be created relative to your current folder.
2. Select **File =>New =>Folder** to bring up the **New Folder** dialog box.
3. Enter the name of the folder you want to create. You can specify:
  - The folder name or a relative path to create the folder relative to **Current Folder**.
  - For a local folder, an absolute path.
  - On a SyncServer, a **sync:** URL
4. Click **OK** to create the new folder.

A success or error message is displayed in the Output Window.

**Notes:**

- The permissions of the new folder are inherited from the parent folder.
- When creating local folders, you must have write privileges for the parent directory.
- DesignSync creates whatever folders are needed to create the specified path. This behavior is similar to UNIX's `mkdir -p` command.
- You can restrict the ability to create server-side folders (sync: protocol). See the Access Control Guide for details.

Click on the fields in the following illustration for information.



## New Folder Field Descriptions

### Current Folder

This field is not editable. It shows the folder from which the New Folder dialog was invoked.

### New Folder

Enter the name of the folder you want to create. You can specify:

- The folder name or a relative path to create the folder relative to Current Folder.
- For a local folder, an absolute path.
- On a SyncServer, a **sync: URL**.

### Related Topics

[Deleting Local Folders](#)

[Deleting Server Folders](#)

ENOVIA Synchronicity Command Reference: mkfolder Command

Command Invocation

Command Buttons

Trigger Arguments

## Moving and Renaming Folders

You can move and rename local folders, whether the folder is associated with a vault or not, using the **mvfolder** command. The DesignSync graphical interface does not support moving and renaming folders.

### Related Topics

ENOVIA Synchronicity Command Reference: mvfolder Command

Moving and Renaming Design Files

## Removing Objects

### Deleting Design Data

You can delete the following objects from DesignSync:

- Local Files
- Local Folders
- Versions from a vault
- Vaults
- Server Folders
- Modules
- Hierarchical References
- Module Cache (mcache) Links

In addition, there are DesignSync command-line commands for each of these object types (**rmfile**, **rmfolder**, **rmversion**, **rmvault**, **rmmod**, **rmhref**).

To delete a branch, use the **purge** command.

### Related topics

ENOVIA Synchronicity Command Reference: rmfile

ENOVIA Synchronicity Command Reference: rmfolder

ENOVIA Synchronicity Command Reference: rmversion

ENOVIA Synchronicity Command Reference: rmvault

ENOVIA Synchronicity Command Reference: rmmmod

ENOVIA Synchronicity Command Reference: rmhref

ENOVIA Synchronicity Command Reference: purge

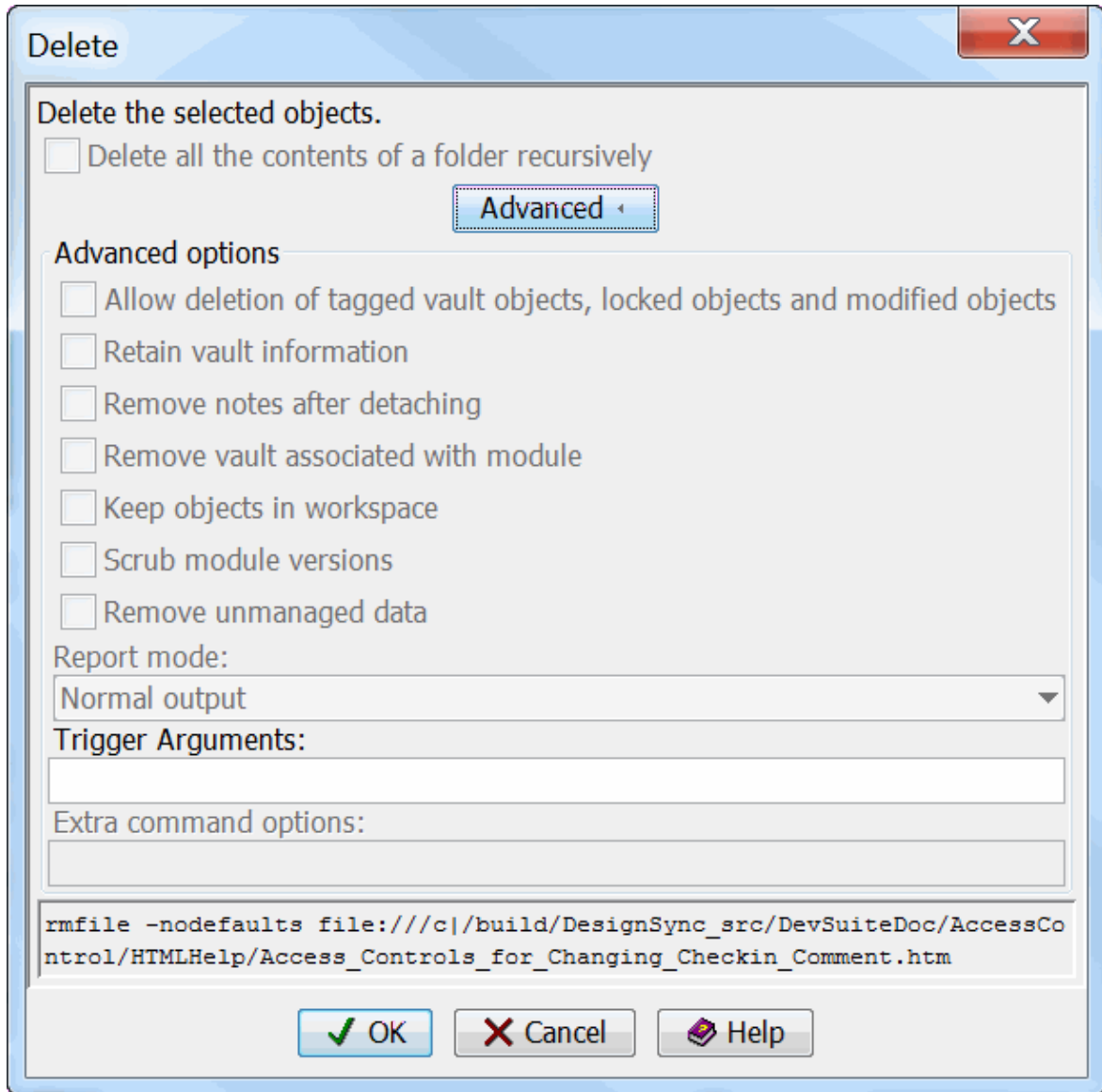
## Deleting Files

You can delete files from your local work area (whether revision controlled or not). You cannot delete a member of a collection

Select the file you want to delete in the List View pane. Multiple files can be selected. Select **File => Delete**, or press the **Delete** key, to bring up the **Delete** dialog. Click **OK** to delete the file. You will be prompted to confirm the deletion.

**Click on the fields in the following illustration for information.**





### Delete Field Descriptions

#### Delete the selected objects/Delete all contents of a folder recursively

This field is not applicable to file deletion.

#### Allow deletion of tagged vault objects, locked objects and modified objects

This field is not applicable to file deletion.

#### Retain vault information

This field is not applicable to file deletion.

### **Remove notes after detaching**

This field is not applicable to file deletion.

### **Remove vault associated with module**

This field is not applicable to file deletion.

### **Keep objects in workspace**

This option is only applicable to module deletion.

### **Scrub module versions**

This option is only applicable when deleting a module version on the server.

### **Remove unmanaged data**

This option is only applicable to module deletion.

### **Report mode**

This option is only applicable to module deletion.

### **Trigger Arguments**

See Trigger Arguments.

### **Extra command options**

This option is only applicable to module deletion.

### **Related Topics**

Deleting Design Data

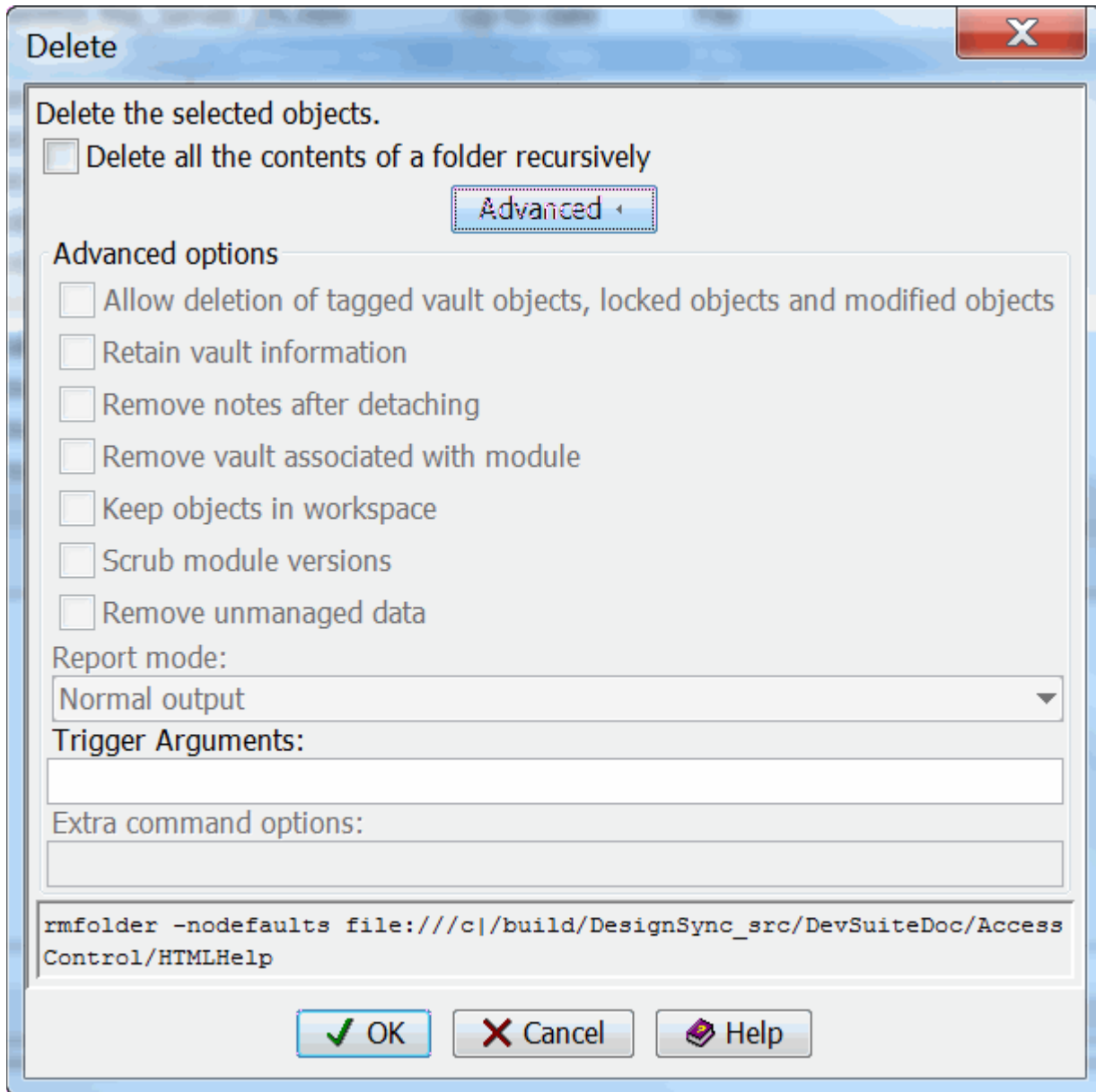
ENOVIA Synchronicity Command Reference: `rmfile`

## **Deleting Folders**

You can delete folders from your local work area whether their contents are revision controlled or not. You cannot delete your current folder or any parent of your current folder.

Select the folder you want to delete in the Tree View or List View panes. Multiple folders can be selected. Select **File => Delete**, or press the **Delete** key, to bring up the **Delete** dialog. Click **OK** to delete the folder. You will be prompted to confirm the deletion.

Click on the fields in the following illustration for information.



## Delete Field Descriptions

### Delete the selected objects/Delete all contents of a folder recursively

By default, you cannot delete a folder unless it is empty. When the recursive option is chosen, delete the specified folder and all folders in the hierarchy beneath it. The

contents of the folders are also deleted, similar to the UNIX command **rm -rf**. Use this option with caution.

### **Allow deletion of a tagged vault options, locked objects and modified objects**

This field is not applicable to deletion of a local folder.

### **Retain vault information**

The option to **Retain vault information** is not applicable to the deletion of local folders. Selecting or de-selecting the option does not affect the operation.

### **Remove notes after detaching**

This field is not applicable to folder deletion.

### **Remove vault associated with module**

This field is not applicable to folder deletion.

### **Keep objects in workspace**

This option is only applicable to module deletion.

### **Scrub module versions**

This option is only applicable when deleting a module version on the server.

### **Remove unmanaged data**

This option is only applicable to module deletion.

### **Report mode**

This option is only applicable to module deletion.

### **Trigger Arguments**

See Trigger Arguments.

### **Extra command options**

This option is only applicable to module deletion.

### **Related Topics**

Deleting Design Data

Creating Folders

ENOVIA Synchronicity Command Reference: rmfolder

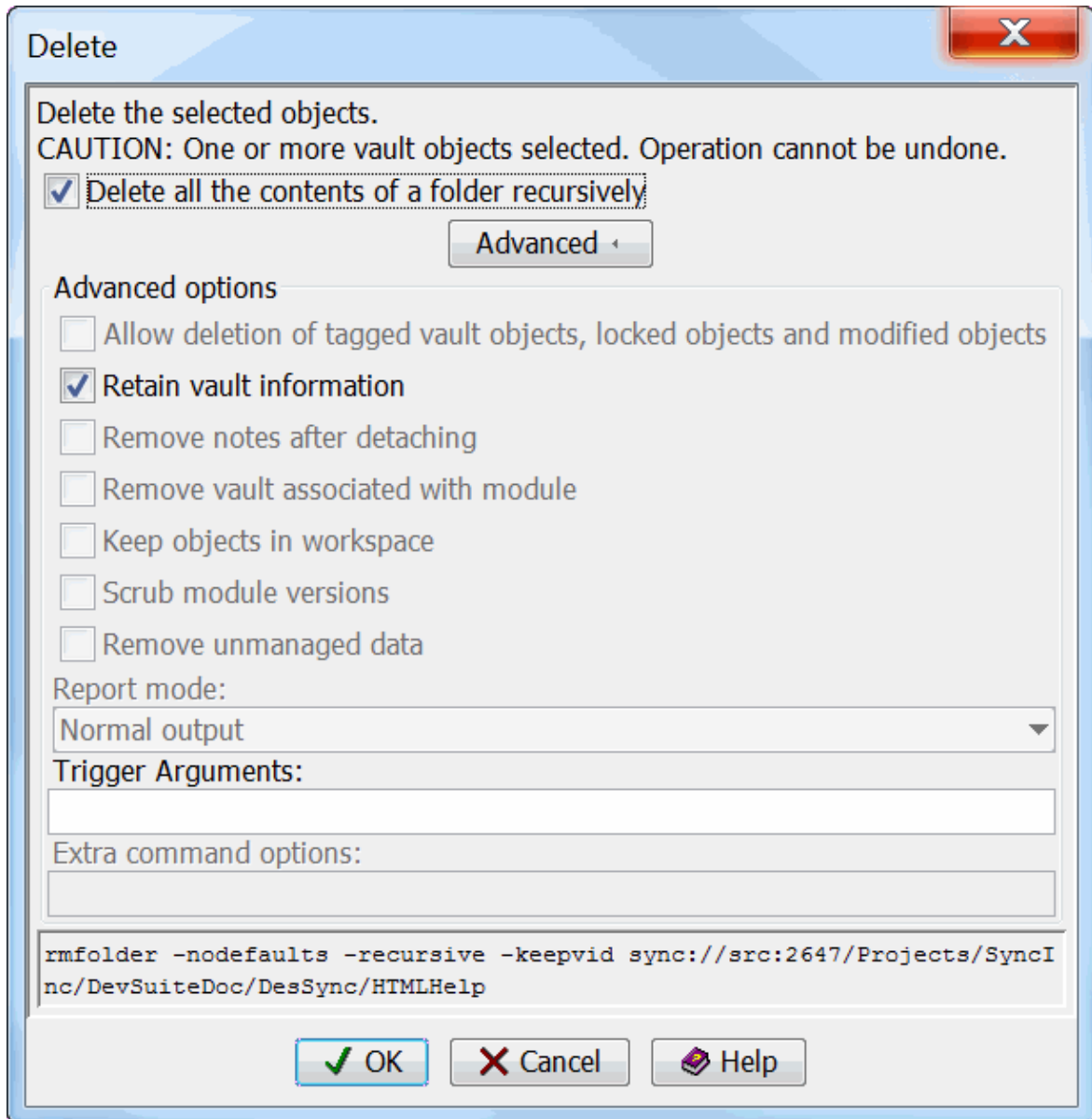
## Deleting Server Folders

A folder on the server, and the vault data contained in the folder, can be deleted. By default, the ability to delete server data is restricted. See the ENOVIA Synchronicity Access Control Guide for details.

From your workspace, select **Go => Go to Vault** to identify the vault folder to be deleted. Or navigate to its server location. Select the vault folder in the Tree View or List View panes. Multiple vault folders can be selected.

Select **File =>Delete**, or press the **Delete** key. You will be prompted to confirm the deletion.

**Click on the fields in the following illustration for information.**



## Delete Field Descriptions

### Delete the selected objects/Delete all contents of a folder recursively

Delete the specified folder and all folders in the hierarchy beneath it. The contents of the folders are also deleted, similar to the UNIX command **rm -rf**. Use this option with caution. All vaults in the specified folder are deleted, regardless of whether a branch is locked, or whether there are tagged versions. By default, you cannot delete a folder unless it is empty.

### Allow deletion of a tagged vault options, locked objects and modified objects

This field is not applicable to vault folder deletion. When the option to **Delete all the contents of a folder** is selected, all vaults in the specified folder are deleted, regardless of whether a branch is locked, or whether there are tagged versions.

### **Retain vault information**

Retain the version number of the last version in the vault. This is the default behavior, so that version numbers are not reused if a vault of the same name is later created.

### **Remove notes after detaching**

This field is not applicable to server folder deletion.

### **Remove vault associated with module**

This field is not applicable to server folder deletion.

### **Keep objects in workspace**

This option is only applicable to module deletion.

### **Scrub module versions**

This option is only applicable when deleting a module version on the server.

### **Remove unmanaged data**

This option is only applicable to module deletion.

### **Report mode**

This option is only applicable to module deletion.

### **Trigger Arguments**

See Trigger Arguments.

### **Extra command options**

This option is only applicable to module deletion.

### **Related Topics**

Deleting Design Data

Deleting Vaults

### Deleting Versions of a Design Object

ENOVIA Synchronicity Command Reference: rmfolder

Trigger Arguments

Command Invocation

## Deleting Vaults

Deleting a vault removes all versions of a design object from the SyncServer (but does not remove any corresponding file in your local work area) and should therefore be used with caution. If there is a design object that is no longer part of your project, consider retiring the object instead of deleting the object's vault. You can unretire a retired object, but you cannot recover a deleted vault.

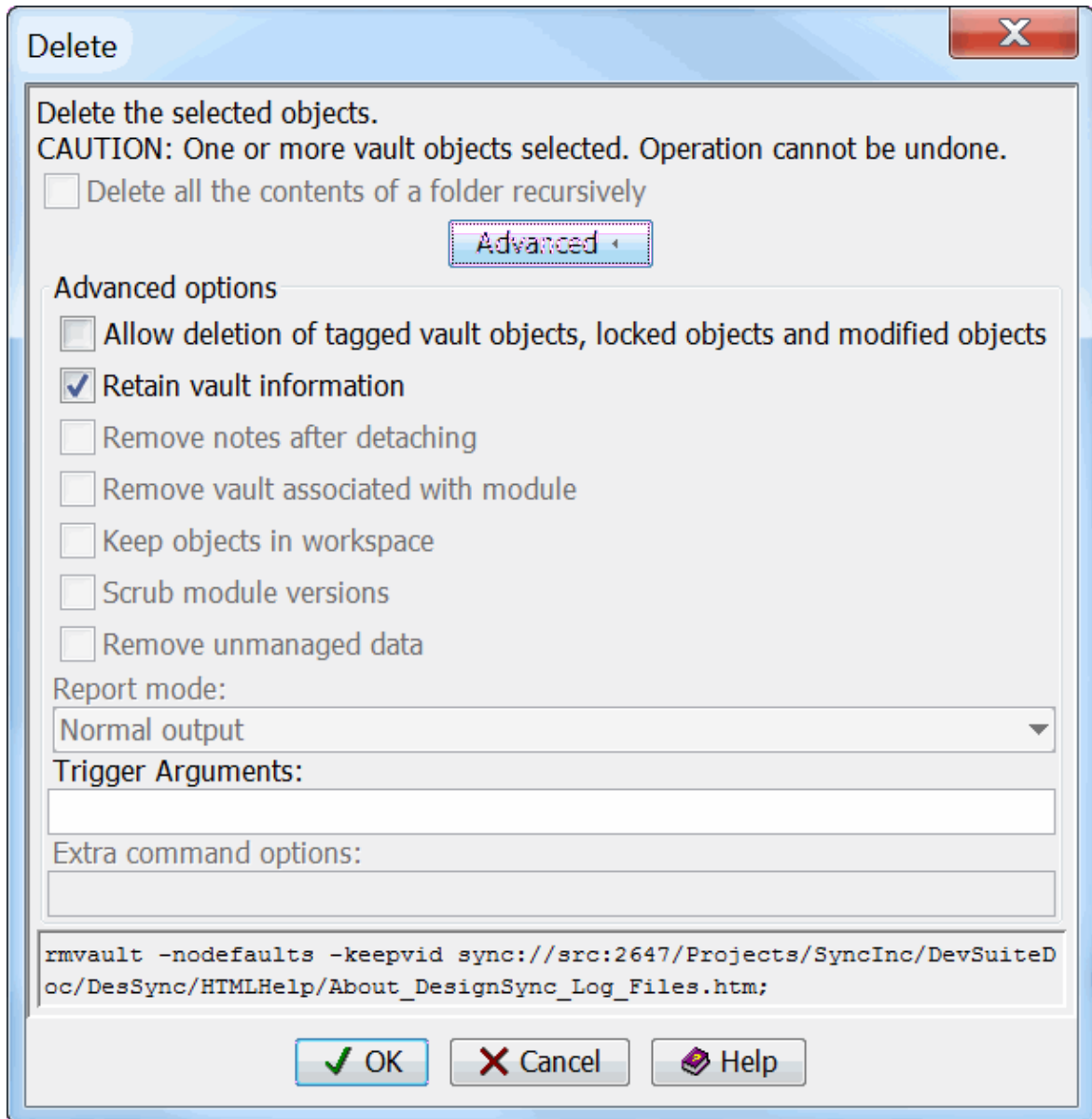
By default, the ability to delete server data is restricted. See the ENOVIA Synchronicity Access Control Guide for details.

From your workspace, select **Go => Go to Vault** to identify the vault to be deleted. Or navigate to its server location. Select the vault in the Tree View or List View panes. Multiple vaults can be selected.

Select **File =>Delete**, or press the **Delete** key. You will be prompted to confirm the deletion.

**Click on the fields in the following illustration for information.**





### Delete Field Descriptions

#### Delete all the contents of a folder/Delete all contents of a folder recursively

This field is not applicable to vault deletion.

#### Allow deletion of tagged vault objects, locked objects and modified objects

Lets you delete a vault that has tagged versions or a locked branch. Use this option with caution:

- A tagged version may be a necessary part of a configuration.

- A locked branch typically indicates someone is editing the design object and therefore the design object is still active.

### **Retain vault information**

Retain the version number of the last version in the vault. This is the default behavior, so that version numbers are not reused if a vault of the same name is later created.

### **Remove notes after detaching**

This field is not applicable to vault deletion.

### **Remove vault associated with module**

This field is not applicable to vault deletion.

### **Keep objects in workspace**

This option is only applicable to module deletion.

### **Scrub module versions**

This option is only applicable when deleting a module version on the server.

### **Remove unmanaged data**

This option is only applicable to module deletion.

### **Report mode**

This option is only applicable to module deletion.

### **Trigger Arguments**

See Trigger Arguments.

### **Extra command options**

This option is only applicable to module deletion.

### **Related Topics**

[Deleting Design Data](#)

[Deleting Versions from a Vault](#)

Retiring Branches

ENOVIA Synchronicity Command Reference: rmvault

Trigger Arguments

Command Invocation

Command Buttons

## Deleting Versions from a Vault

DesignSync allows you to delete versions of an object from the vault. Deleting a version does not remove any corresponding file in your local work area.

You cannot delete:

- The first version of an object on any branch (for example, 1.1, 1.1.1.1, 1.3.2.1, and so on).
- A branch-point version (for example, if 1.2.1 is a branch, you cannot delete version 1.2).
- The Latest version on a locked branch (for example, if someone checks out version 1.3 with a lock, you cannot delete version 1.3 from the vault until the lock is released).

**Note:** You cannot recover a deleted version, so delete versions with caution. Before deleting versions from the vault, you may want to verify with your project team that the versions can be safely deleted.

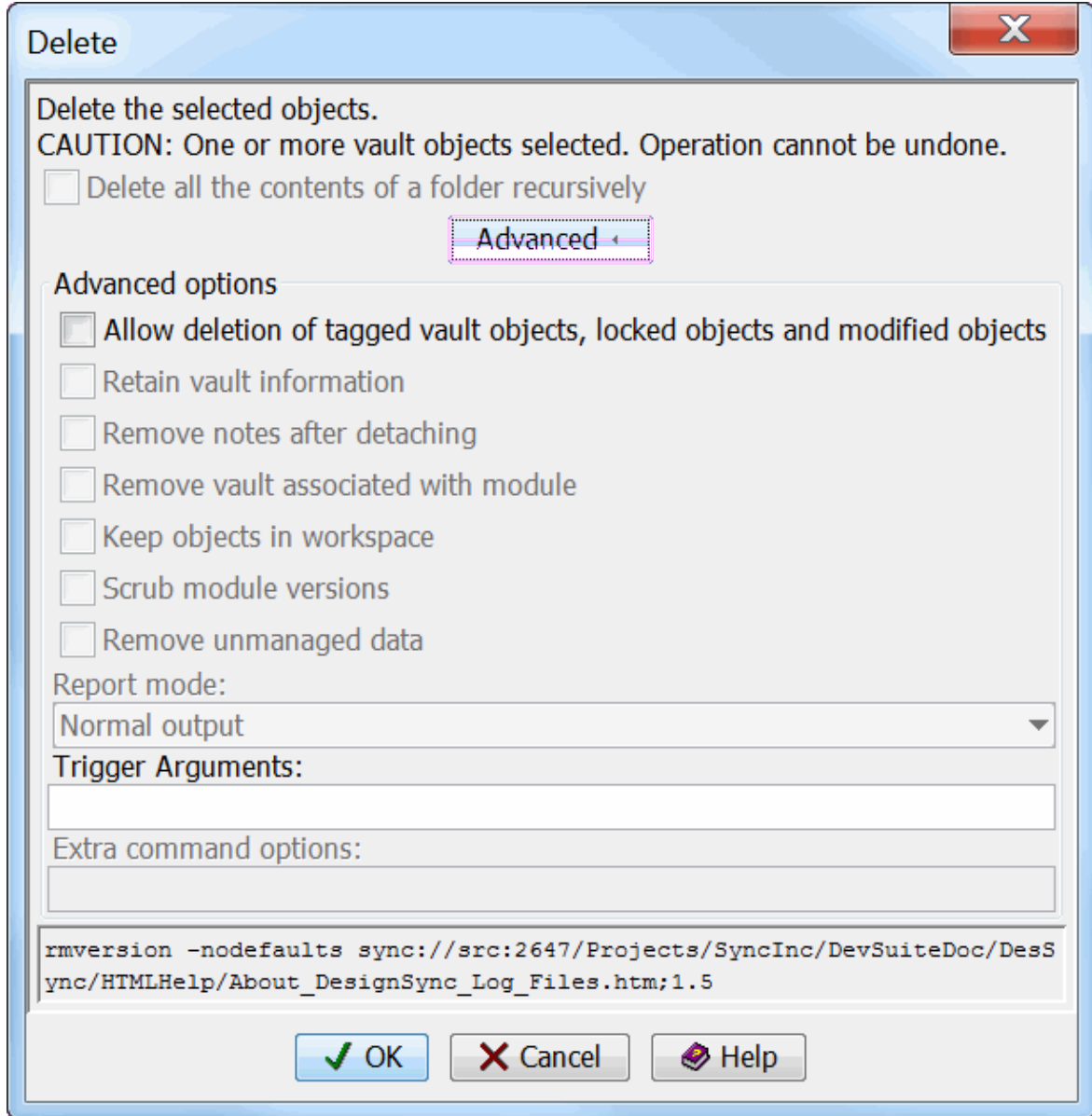
You can clean up the vault by deleting old versions of objects using the `purge` command. This command deletes a range of versions of an object on a single branch in the vault.

You can use the `purge` command on files, objects, or folders. The command also provides options that let you delete all versions of an object except the last <n> number of versions or delete all versions created before a specified date.

**To delete vault versions using the DesignSync graphical interface:**

1. From your workspace, select **Go => Go to Vault** to identify the vault whose versions are to be deleted. Or navigate to the vault's location on the server. Select the vault versions in the Tree View or List View panes.
2. Select **File =>Delete**, or press the **Delete** key. You will be prompted to confirm the deletion.

Click on the fields in the following illustration for information.



#### Delete Field Descriptions

#### Delete the selected objects/Delete all contents of a folder recursively

This field is not applicable to version deletion.

#### Allow deletion of tagged vault objects, locked objects and modified objects

Deletes tagged versions from the vault. Use this option with caution, because deleting a tagged version changes (possibly damaging) a configuration. The "locked vault" aspect of the field name is not applicable to version deletion.

**Retain vault information**

This field is not applicable to version deletion.

**Remove notes after detaching**

This field is not applicable to version deletion.

**Remove vault associated with module**

This field is not applicable to version deletion.

**Keep objects in workspace**

This option is only applicable to module deletion.

**Scrub module versions**

Searches for and removes orphaned module members; module member versions no longer referenced by any module versions. By default, this option is not selected.

**Remove unmanaged data**

This option is only applicable to module deletion.

**Report mode**

This option is only applicable to module deletion.

**Trigger Arguments**

See Trigger Arguments.

**Extra command options**

This option is only applicable to module deletion.

**Related Topics**

Deleting Design Objects

ENOVIA Synchronicity Command Reference: rmversion Command

ENOVIA Synchronicity Command Reference: purge Command

Trigger Arguments

### Command Invocation

## Retiring Design Data

If a design object is obsolete or no longer part of the design project, you can retire the object's branch. If you work on objects that reside on a single development branch (the default Trunk branch), retiring the sole branch of an object retires the object.

The retire and unretire operations only apply to managed DesignSync objects. The **Retire** dialog box is not active when a folder is selected, or when module data is selected. For module data, if your intention is for a selector to no longer resolve to a branch, remove the relevant branch tag from the module.

Retiring a branch prevents the branch from participating in future populate operations and prevents new versions from inadvertently being created on the branch. Users must specify the checkin of new items, which will unretire the branch.

When you retire an object branch, DesignSync preserves the following information about the retire operations.

- Date of the retire
- Time of the retire
- Username of the user who performed the retire

This information is displayed in the version history of the branch. If the branch is then unretired, the retire information is removed. When you display the version history of an unretired object, there is no record maintained of the branch having been retired.

**Tip:** If you want to preserve information about the retire, you can check in the unretired file with a comment detailing the retire history.

Suppose that you populate your work area with files A, B, and C from the Trunk branch. Further along in development, file B is no longer needed. So you retire its Trunk branch. When you or other team members populate your work area again, the populate operation does not copy file B to your work area.

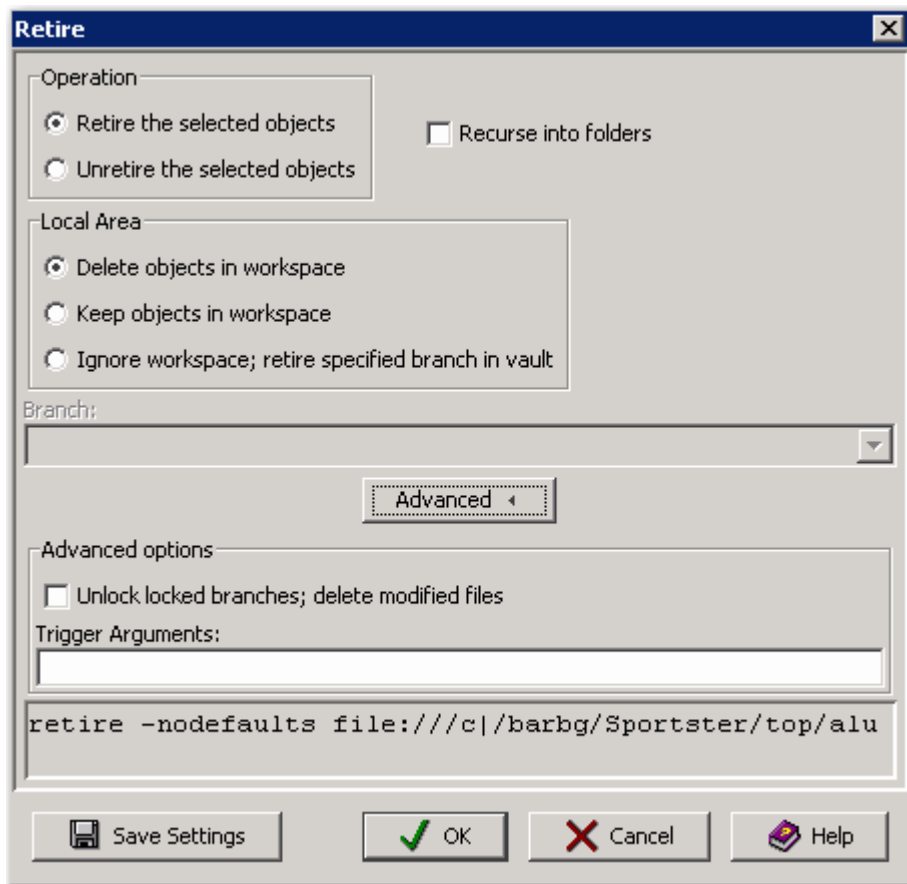
You may want to inform your team before you retire a file's branch in case someone wishes to continue using the file. ProjectSync's Email Notification can be used for the communication. Ask your team leader to review the RevisionControl Notes Overview.

The Status column of the List View in DesignSync GUI and the `ls` command both indicate if the current branch of an object in your work area is retired. You can also display the Version History of an object, to determine whether a branch of an object is retired.

If you decide that a retired object needs to be active again, you can use the Retire dialog box to unretire the object's branch.

**Note:** Retired files show up in the list generated by **Show Potential Checkouts** as italicized items

**Click on the fields in the following illustration for information.**



## Retire Field Descriptions

### Operation

The Retire dialog box lets you retire or unretire the selected object's branch. Select the option for the action you want to perform:

- **Retire the selected objects.** Prevent the selected branch from participating in future populate operations. Prevent the creation of new versions on the branch, unless the `new` option is specified for the checkin operation.
- **Unretire the selected objects.** Let the selected branch once again participate in populate operations. Allow the creation of new versions on the branch.

## Recurse into folders

For a DesignSync folder, recursively operate on its contents. By default, only the contents of the selected folder are operated on. This option does not apply to module data.

## Local Area

How you want DesignSync to handle the object in your work area after its branch has been retired:

- **Delete objects in workspace.** If you have not modified the object, DesignSync deletes the object from your work area. (If you have modified the object in your work area, DesignSync keeps the object in your work area.) This option is the default.
- **Keep objects in local workspace.** The object remains in your work area. However, if your workspace contains a symbolic link to a copy of the object in the mirror, the option to **Keep objects in local workspace** is not available. That is because if the mirror directory is for Latest file versions, the retired object is automatically removed from the mirror.
- **Ignore workspace; retire specified branch in vault.** The object in your work area is not affected. You must specify the branch to retire, in the **Branch** field.

## Branch

This option is used in conjunction with **Ignore workspace; retire specified branch in the vault** to retire a branch other than the branch of the objects in your work area.

The **Branch** field has a pull-down menu from which you can query for existing branches. See Suggested Branches, Versions, and Tags for details.

**Note:** The Branch field accepts a branch tag, a version tag, a single auto-branch selector tag, or a branch numeric. It does not accept a selector list.

## Unlock locked branches; delete modified files

Unlock locked branches prior to retiring them. Also, if you selected **Delete objects in workspace**, your local copy of the object will be deleted, even if you have modified the local object. To keep your local objects, select **Keep objects in workspace**.

**Note:** Use this option with caution; it removes the lock even if it is held by someone else.

## Related Topics

Deleting Files or Versions from a Design Project



Operating on Cadence Data

ENOVIA Synchronicity Command Reference: retire Command

Trigger Arguments

Command Invocation

Command Buttons

## Removing a Member from a Module



The Remove from module dialog box is used to remove the selected module members from a module. You may commit the change, as well as any other module changes, to the module at the next module checkin, or you can immediately create a new module version without the selected module member. You can use this dialog box when:

- Current module members (files/folders) are selected in the client work area.
- Current module members (files/folders) are selected on the server and all members belong to the same module version, which must be the latest version on a server module branch.
- Current module base folders are selected in the client work area when the folder is also a regular module folder of another module.
- The module in the workspace was populated in dynamic mode and changes can be checked in.

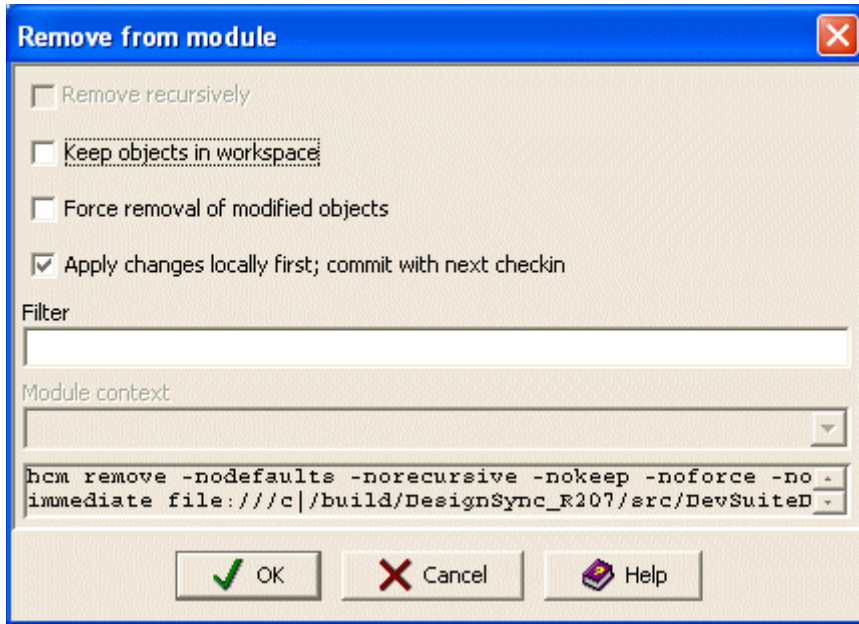
### Note

Removing module members/folders that have not been fetched into the workspace is not supported by the GUI.

### To remove a member from a module:

1. Select the member(s).
2. From the main menu, select **Modules =>**  **Remove Member** or select the  button from the Module Toolbar.
3. The Remove from module dialog box appears. Select options as needed.
4. Click **OK**.

Click on the fields in the following illustration for information.



### Remove Recursively

This option is not available when a server object is selected

When checked, the selected folder is removed as well as all module objects in the folder and all subfolders. When not checked, the selected folder is removed only if the folder is empty. The default for this option is unchecked.

### Keep objects in workspace

This option is not available when remove is done directly on a server object.

When checked, the local copies of the removed objects are left in the workspace as unmanaged objects. When unchecked, the local copies of the selected objects become DesignSync references, with the Status "Locally Removed". The default for this option is unchecked.

### Force removal of modified objects

This option is not available when remove is done directly on a server object

When checked, the selected object is removed even if the object in the workspace is not the same as the last checked in version of the object or is locked. If you are removing objects that were added to a module, but never checked in, you choose this option to remove the objects.

When unchecked, the selected object is not removed if it is locked or not identical to the last checked in version or added to the module but never checked in.

The default for this option is unchecked.

### **Apply changes locally first; commit with next checkin**

This option is not available when remove is done directly on a server object

When checked, the selected object(s) is marked for removal during the next module check in. Unless the `Keep_Objects_in_Workspace` option is selected, the object remains in the workspace as a DesignSync reference, with the Status "Locally Removed". The object remains on the server until the change is committed during the next checkin. (Default)

When not checked, DesignSync immediately creates a new module version on the server with the selected objects removed.

**Note:** Objects in the Add state are always immediately removed from the Add state. No new module version is created.

### **Filter**

Allows you to include or exclude module objects by entering one or more extended glob-style expressions to identify an exact subset of module objects on which to perform the remove. For more information, see Filter Field.

The default for this field is empty.

### **Module Context**

This option is only available when the selection set includes one or more client side folders. You can select from the available module instances. The choices are listed in alphabetical order. For more information, see Module Context Field.

The default for this field is empty.

### **Related Topics**

The Module Toolbar

Filter Field

ENOVIA Synchronicity Command Reference: remove

# Comparing Files

## Common Diff Operations

The diff operations provided in the **Tools => Compare Files** menu provide the most common file comparison operations. They do not display a dialog; instead they immediately perform the comparison operation using default settings for all options and display the results in the view pane.

**Note:** DesignSync also allows you to compare workspaces and configurations with the Compare tool available from **Tools => Reports => Compare**.

The following common diff operations are available:

- **Show Local Modifications** compares the selected object in your work area with the original version that you checked out. This report shows changes made in your work area since the object was checked out. This is a 2-way diff operation, so the Revised Diff Format option may be used. You can select Revised format as the Display Option in the Advanced Diff dialog, or set Revised as the default Diff Format via SyncAdmin.
- **Compare to Latest** performs a 3-way diff comparing the selected object in your work area with the current version in the vault, using the original version in the vault as the common ancestor. This report can be used to show the results that occur if a **Checkout** operation with the **Merge with workspace** option (**co - merge**) is performed, including any conflicts that occur.
- **Compare Original to Latest** compares the original version of the selected object in your work area with the latest version in the vault. This report shows changes made to the vault by others since the object was checked out. This is a 2-way diff operation, so the Revised Diff Format option may be used. You can select Revised format as the Display Option in the Advanced Diff dialog, or set Revised as the default Diff Format via SyncAdmin.
- **Compare to Previous Version** compares the currently selected version of the object in your work area or on your server with the last checked in version of the object. If the selected object is the first version on a branch, the previous version is the last content change prior to the branch. If the selected object is a module member, the previous version is the previous version of the selected module member, not necessarily the previous module version. If the compared objects contain merge edges, they are included in the comparison. This is a 2-way diff operation, so the Revised Diff Format option may be used. You can select Revised format as the Display Option in the Advanced Diff dialog, or set Revised as the default Diff Format via SyncAdmin.
- **Compare 2 Files** compares the two selected objects. You can use this report to compare two files or two versions of the same file. Depending on the default

settings, if two versions of a file are selected, this report may attempt to identify a common ancestor for a 3-way diff. If two different files are selected, then this performs a 2-way diff, in which case the Revised Diff Format option may be used. You can select Revised format as the Display Option in the Advanced Diff dialog, or set Revised as the default Diff Format via SyncAdmin.

The default settings used by the common diff operations can be modified by selecting options in the Advanced Diff dialog box and pressing the **Save Settings** button.

See Reading Diff Results for information on interpreting diff output.

## Related Topics

[Advanced Diff Options](#)

[Graphical Diff Utility](#)

[Identifying Changed Objects](#)

[SyncAdmin Help: diff Format](#)

[SyncAdmin Help: Customizing Diff Output](#)

[ENOVIA Synchronicity Command Reference: diff](#)

[ENOVIA Synchronicity Command Reference: co](#)

## Advanced Diff Options

**Advanced Diff** is a tool for displaying the differences between two files or two versions of the same file. You can compare the two files directly (a 2-way comparison) or against a common ancestor (3-way comparison). When comparing ASCII (text) files, Advanced Diff provides detailed difference and conflict information.

To access the complete set of options for comparing files, select **Tools => Compare Files => Advanced Diff**.

### Version-Extended Naming

When invoking Advanced Diff from DesignSync, you can specify the files you want to compare using version-extended filenames, which consist of the filename, followed by a semicolon (;), followed by a version number or tag. For example, alu.v;1.2 is version 1.2 of alu.v, and alu.v;golden is the version that is tagged 'golden'. You can also use the following reserved tags:

**Orig** The version in the vault from which your local version originated; for example, alu.v;Orig.

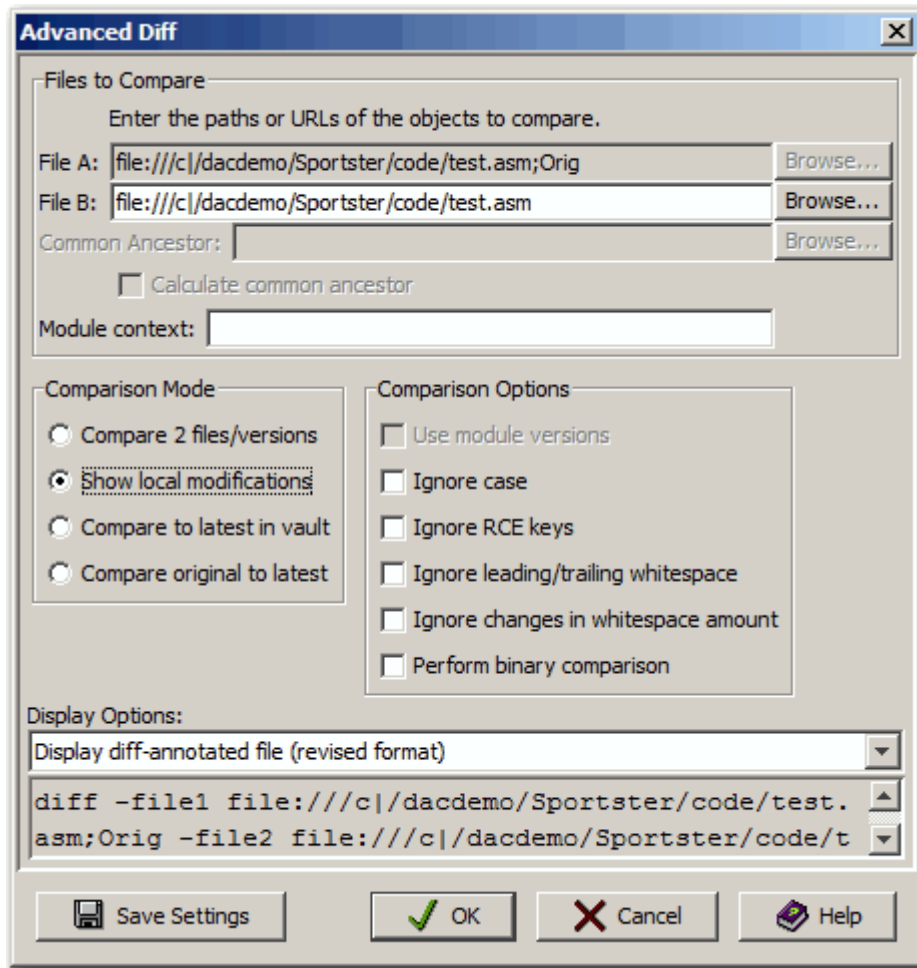
**Latest** The most recent version in the vault; for example, alu.v;Latest. Often Latest and Orig are the same version. For example, you fetch alu.v;1.4, which is the most recent version in the vault. Version 1.4 is both Latest and Orig. If your teammate then checks in version 1.5, Orig is still version 1.4, but Latest is now version 1.5.

### Notes:

- For modules, you can specify version extended filenames to the local workspace version, but not to the server version, using the sync URL. For example: alu.v;1.2 is a valid construct, but sync://srv2.ABCo.com:2647/Modules/ALU/alu.v;1.2 is not. To specify a module member that is not loaded into your workspace, use the Module context field to select the module, then specify the file as a simple version extended file reference: /alu.v;1.2.
- When specifying a module version, you can specify whether the version refers to the module member version or the module version of the desired object by using the **Use module version** comparison option.

DesignSync interprets these version-extended filenames and fetches files from the vault as needed, placing them in your DesignSync cache. These cached files are used exclusively by the graphical Diff utility, if you have one; DesignSync users will not get links to these files if they populate from the cache.

**Click the fields in the following illustration for information.**



Click **OK** to execute the Advanced Diff command. The results are displayed in the Diff View window, unless you selected the **Display output in GUI** option.

## Notes

Click **Save Settings** to set the current settings as the default for the Advanced Diff dialog. The Comparison Options set in the Advanced Diff dialog and saved via **Save Settings** applies to subsequent Advanced Diff invocations, to all common diff operations invoked via **Tools => Reports => Compare**, and via actions from the Changed Object Browser.

If two objects are selected when the **Tools => Compare Files => Advanced Diff** menu option is chosen, the comparison mode is initially set to **Compare 2 Files/Versions**; otherwise, it is initially set to the default setting defined in **Show local modifications**.

## Field Descriptions

### Files to Compare

Specify one or two files depending on the type of comparison you want to perform. If you choose to compare 2 files or versions in the Comparison Mode, enter the URLs of the objects to compare in the **File A** and **File B** fields.

Optionally, you may specify a common ancestor. If you specify a common ancestor, a 3-way diff is performed. For example, if you specify the following:

<b>File A</b>	file:///home/aurora/projects/alu/sym12.dat;exper1
<b>File B</b>	file:///home/aurora/projects/alu/sym12.dat;release17
<b>Com. Anc.</b>	file:///home/aurora/projects/alu/sym12.dat;golden

the **exper1** and **release17** versions of the given file are compared using the **golden** version as the common ancestor.

Instead of specifying a common ancestor, you can select **Calculate Common Ancestor**, and the common ancestor will be determined, using the object's branch and merge history. If no common ancestor can be determined, a 2-way diff is performed. For example, if you select **Calculate Common Ancestor** and specify the following:

<b>File A</b>	file:///home/aurora/projects/alu/sym12.dat;1.12.1.5
<b>File B</b>	file:///home/aurora/projects/alu/sym12.dat;1.17

a 3-way diff will be performed using version 1.12 as a common ancestor (unless a merge has created a more recent common ancestor).

If you are comparing files that are not present in the workspace, you may use the sync URL to specify a non-module object. If you are specifying a module member that is not in the workspace, use the Module context field to specify the module, and the Files to compare field to specify the member's natural path, name and version.

**Note:** The **Select File to Compare** browser does not display DesignSync references.

### Module Context

Specify the module context. You can specify a module instance or the sync URL to the module. If you are specifying a module member that is not located in the workspace, the module context uniquely identifies the source module.

In some cases, DesignSync will calculate the default module context for you.

- If the objects being compared are both located on the server and within the same module, the default is the module name on the server.



- If there is only one object specified, or one object is in the workspace, while the other object is on the server, the module context field automatically fills with the server object version.

## Comparison Mode

Specify one of the following types of comparison:

- **Compare 2 files/version** allows you to compare any two objects or versions of an object.
- **Show local modifications** compares the object in your work area with the original that was checked out, showing you the changes made in your work area.
- **Compare to latest in vault** compares the original version of an object with the latest version in the vault; showing changes made by others since you checked the object out.
- **Compare original to latest** compares the object in your work area with the current version in the vault, using the original as a common ancestor.

## Comparison Options

Select from the following comparison options:

- **Use module versions:** Select this option to use module versions rather than module member versions when comparing two files/versions.
- **Ignore case:** Select this option for case-insensitive comparisons.
- **Ignore RCE keys:** Determines whether differences in RCE keyword values are ignored. RCE keywords are tokens (such as `$Revision: 1.32 $`, `$AUTHOR$`, `$LOG$`) that you can add to your files to provide revision information (such as revision number, author, and comment log). If you select this option, DesignSync hides the keyword values (collapses the keywords) prior to comparing the files. For example:

First line of File A:

```
$Id: Advanced_Diff.htm.rca 1.32 Wed Dec 30 06:37:31 2015
FYL Experimental $ $ $ $ $ $ $ Exp $
```

First line of File B:

```
$Id: Advanced_Diff.htm.rca 1.32 Wed Dec 30 06:37:31 2015
FYL Experimental $ $ $ $ $ $ $ Exp$
```

Advanced Diff reports the difference unless you select the **Ignore RCE keys**

option, in which case Advanced Diff collapses each line to: `$Id:`

```
Advanced_Diff.htm.rca 1.32 Wed Dec 30 06:37:31 2015 FYL
Experimental $ Advanced_Diff.htm.rca 1.25 Wed Aug 17
```

## DesignSync Data Manager User's Guide

```
07:32:07 2011 mhopkins Experimental $ Advanced_Diff.htm.rca
1.24 Tue Aug 9 10:59:15 2011 mhopkins Experimental $
Advanced_Diff.htm.rca 1.23 Mon Mar 14 19:27:46 2011 e88
Experimental $ Advanced_Diff.htm.rca 1.21 Thu Jan 27
08:33:25 2011 mhopkins Experimental $ Advanced_Diff.htm.rca
1.20 Thu May 13 13:02:19 2010 mhopkins Experimental $ 1.10
Thu May 12 12:07:30 2005 mmf Experimental $.
```

Differences in keyword usage and placement are always reported. For example:

First line of File A:

```
$Id: Advanced_Diff.htm.rca 1.32 Wed Dec 30 06:37:31 2015 FYL
Experimental $ $ $ $ $ $ $ $
```

First line of File B:

```
$Author: FYL $
```

Advanced Diff reports the difference irrespective of the Ignore RCE keys setting because the keywords themselves, not just the keyword values, are different.

### Notes:

- `$LOG$` when expanded, permanently adds log information to your files. The **Ignore RCE keys** option does not hide these log messages prior to performing a comparison. Advanced Diff may flag differences or conflicts (if log information has been edited by hand) in your files. Creating first version on the Trunk branch, in the new DevSuite doc vault location, with the content of the docmain:Latest files from the old doc vault location. is the only keyword with this behavior.
- Advanced Diff honors the `$KeysEnd$` keyword -- any expanded keywords after `$KeysEnd$` are compared fully and literally.
- **Ignore leading/trailing white space:** Determines whether white space (spaces, tabs) differences at the beginning or end of a line are ignored. For example, if a line in one file starts with a tab character whereas the same line in the other file starts with a space, Advanced Diff ignores the difference if you select this option.
- **Ignore changes in white space amount:** Determines whether differences in white space within a line are ignored. For example, if a line in one file has three spaces between two words whereas the same line in the other file has only one space, Advanced Diff ignores the difference if you select this option.
- **Perform binary comparison:** Performs the comparison in binary mode. When comparing files in binary mode, Advanced Diff only reports whether the files are identical or different. No other comparison options are available in binary mode.

## Display Options

The Display Options determines how the results of a diff operation are displayed. The Display Options field in the Advanced Diff dialog, when saved via **Save Settings**, only affect subsequent invocations of the Advanced Diff dialog. It does not influence the common diff operations. The result of the common diff operations are all controlled by SyncAdmin's Diff Format settings.

You can choose one of the following display options:

- **Display diff-annotated file (revised format)** uses a DesignSync-specific diff output format to display the complete file with annotations indicating both the change and the change type (addition, deletion, modification). This format is only valid for a 2-way diff, such as **Show local modifications**. You can also create a 2-way diff comparison by invoking **Compare 2 files/versions** when no common ancestor is defined, by selecting separate files. For more information on the revised format, see Revised Diff Format.
- **Display only the diffs (standard format)** uses the standard UNIX output format to display only lines that have been added, removed, or changed.
- **Display only the diffs (unified format)** uses the gnu unified output format to display both lines that are the same and lines that have changed.
- **Display only the diffs (syncdiff format)** uses a DesignSync-specific diff output format to display only lines that have been added, removed, or changed.
- **Display diff-annotated file** displays both lines that are the same in both files and lines that have changed.
- **Display output in GUI** displays the results using either the built-in DesignSync graphical diff tool, or a user-defined external graphical Diff tool. For more information on graphical Diff tools, see Graphical Diff Utility.

Results are displayed in the Diff View tab page of the main DesignSync panel. If an external graphical Diff tool is registered a separate window is created for the diff output.

See Reading Diff Results for examples of the different types of output.

## Related Topics

Common Diff Operations

Identifying Changed Objects

Reading Diff Results

ENOVIA Synchronicity Command Reference: diff

## Reading Diff Results

The format of your diff results is determined by your **Display Options** selection on the Advanced Diff panel. (See Advanced Diff Options for details.) The output from the different types of diff operations is described in the following sections:

Display diff-annotated file (revised format)

Display only the diffs (standard format)

Display only the diffs (unified format)

Display only the diffs (syncdiff format)

Display diff-annotated file

Display output in GUI

### **Display diff-annotated file (revised format)**

This format shows the complete file with annotated lines that have been added, removed, or changed. This view displays in a new tab, Diff, in the View Pane area of the DesignSync GUI. In this view, you see line numbers for each line in the file and graphical elements indicating where changes occur and what those changes are. This format is used for 2-way diffs, with the diff results displaying in a single window viewer.

For more information on the Revised diff format, see Revised Diff Format.

### **Display only the diffs (standard format)**

This format shows lines that have been added, removed, or changed in standard UNIX output format.

In this view, you see the line numbers where changes occurred in the original file, the type of change, the line numbers in the edited file, and the text of the change. The type of change is indicated as follows:

a - An addition.

d - A deletion.

c - A change.

For example, the view pane might display output like the following:

```
5a6  
> sample/dflop0 type="cell"  
7d7
```

```

< sample/dflop2 type="cell"
8a9
> sample/dflop4 type="cell"
12c13
< sample/fflop type="cell"
---
> sample/fflop0 type="cell"

```

The first section shows lines that were added, as indicated by the `a` between the line numbers. In the edited file, one new line, 6, was added following line 5 in the original file. The `>` indicates an added line, followed by the text of the addition.

The second section shows a line that was deleted, as indicated by the `d` between the line numbers. In the edited file, one line was deleted after line 7 in the original file. This deletion became line 7 in the new file (because of the line added as line 6). The `<` indicates a line that was removed, followed by the text of the deletion.

The third section shows that a line was changed. In the original file, the line was 12; it is now line 13 of the edited file. The text above the three-dash line is the original text; the text below the line is the edited text.

## Display only the diffs (unified format)

This format shows lines that have changed using the GNU unified output format. The edited lines are presented in the context of the sections of the files where they appear.

In this view, the original file is preceded by three dashes (`---`); the edited file is preceded by three pluses (`+++`). The line enclosed in ampersands (`@@`) indicates the changed sections of the two files. The first range of line numbers shows the affected section of the original file; the second range shows the affected section of the edited file.

In the main body of the code, lines common to both files begin with a space character. The lines that differ begin with one of the following characters:

- + Indicates a line added to this position.
- Indicates a line removed from this position.

For example, the view pane might display output like the following:

```

---
sync://zen.our_company.com:2647/Projects/smallLib/FFlops.Cat;1.1
+++ file:///c:/Projects/smallLib/FFlops.Cat
@@ -3,13 +3,14 @@
 sample/dffpc_ type="cell"
 sample/dffpp_c_ type="cell"

```

```
sample/dflop type="cell"  
+sample/dflop0 type="cell"  
  sample/dflop1 type="cell"  
-sample/dflop2 type="cell"  
  sample/dflop3 type="cell"  
+sample/dflop4 type="cell"  
  sample/dl type="cell"  
  sample/dla type="cell"  
  sample/dlc_ type="cell"  
-sample/fflop type="cell"  
+sample/fflop0 type="cell"  
  sample/fflop1 type="cell"  
  sample/fflop2 type="cell"  
  sample/fflop3 type="cell"
```

The + characters at the beginnings of lines 4 and 8 indicate lines that were added in your edited file. The - at the beginning of line 6 indicates a line that was removed. Lines 12 and 13 show a line that was changed; the original line is preceded by a - character and the change in the edited file is indicated by the + character.

### Display only the diffs (syncdiff format)

This format shows lines that have been added, removed, or changed in the syncdiff format.

In this view, the original file is file A; the edited file is file B. The information above the dashed line (====) tells you what type of change was made and the original line number. The corresponding information below the dashed line gives you the text of the change and the new line number.

The lines that differ begin with one of the following characters:

- + Indicates an added line.
- Indicates a deleted line.
- < and > indicate a changed line.

For example, the view pane might display output like the following:

```
Comparing:   (A, B)  
(A)  
sync://zen.our_company.com:2647/Projects/smallLib/FFlops.Cat;1.1  
(B) file:///c:/Projects/smallLib/FFlops.Cat  
Added (B6)  
=====
```

```

    B5  + sample/dflop0 type="cell"
Deleted (A7)
=====
    A6  - sample/dflop2 type="cell"
Added (B9)
=====
    B8  + sample/dflop4 type="cell"
Changed (A12, B13)
=====
    A11 < sample/fflop type="cell"
-----
    B12 > sample/fflop0 type="cell"

```

The first change is a line that was added. In file B, this line becomes line 6. Below the dashed line are the original line number (B5) and the text of the added line. The second change is a line that was removed. In file A, this line originally followed line 7. Below the dashed line are the new line number and the text of the deleted line. The final section shows a changed line. In file A, the line was 11 and is shown as removed (<); in file B the line is now 12 and is shown as added (>).

## Display diff-annotated file

This format shows all the lines in both the original file and the edited file. The first column shows the line numbers in the original file; the second column shows the line numbers in the edited file.

The lines that differ begin with one of the following characters:

- + Indicates an added line.
- Indicates a deleted line.
- < and > indicate a changed line.

For example, the view pane might display output like the following:

```

1   1   TDMCHECKPOINT="1.0"
2   2   sample/dffp type="cell"
3   3   sample/dfipc_ type="cell"
4   4   sample/dfpp_c_ type="cell"
5   5   sample/dflop type="cell"
6     + sample/dflop0 type="cell"
6   7   sample/dflop1 type="cell"
7     - sample/dflop2 type="cell"
8   8   sample/dflop3 type="cell"
9     + sample/dflop4 type="cell"

```

## DesignSync Data Manager User's Guide

```
9   10   sample/dl type="cell"
10  11   sample/dla type="cell"
11  12   sample/dlc_ type="cell"
12      < sample/fflop type="cell"
13      > sample/fflop0 type="cell"
13  14   sample/fflop1 type="cell"
14  15   sample/fflop2 type="cell"
15  16   sample/fflop3 type="cell"
```

The + characters at the beginnings of lines 6 and 9 in the edited file indicate lines that were added in the edited file. The - at the beginning of line 7 indicates a line that was removed in the edited file. Lines 12 and 13 show a line that was changed; the original line is preceded by a < character and the change in the edited file is indicated by the > character.

When you use diff-annotated output, you can click the right mouse button on the view pane to display a context menu with the following choices:

- **Previous Diff** scrolls to the previous group of added, deleted, or changed lines.
- **Next Diff** scrolls to the next group of added, deleted, or changed lines.
- **Previous Conflict** scrolls to the previous group of lines in conflict. This menu choice is available only with 3-way diffs.
- **Next Conflict** scrolls to the next group of lines in conflict. This menu choice is available only with 3-way diffs.
- **Diff Properties** displays the Customize Diff dialog box.

## Display Output in GUI

This format shows changes graphically using either the built-in DesignSync graphical diff tool, or a user-defined external graphical Diff tool. Diffs that display in a multiple window viewer use this display mode. For more information on graphical Diff tools, see Graphical Diff Utility.

### Related Topics

[Advanced Diff Options](#)

[Common Diff Operations](#)

[Graphical Diff Utility](#)

## Revised Diff Format

Revised Diff format is used when a 2-way diff is performed, typically via **Show Local Modifications** from the Changed Objects Browser or **Tools => Compare Files**. The



Revised Diff output format shows a unified file containing all the Diffs marked for review. The Diff results are displayed in a new text tab labeled **Diff**, in the View Pane.

For the Revised Diff format to be used, set Revised Diff as the default diff format via SyncAdmin's Diff Format. Or, select the Revised Diff format in the Display Options field of the Advanced Diff form.

The Revised Diff format is only used when a 2-way diff is performed, such as for **Show Local Modifications**. Even if Revised Diff is set as the default Diff Format, if a 3-way diff is performed, such as **Compare to Latest**, then the diff results will be displayed using the 3-way Annotated Diff format.

## Using Revised Diff Format

The Revised Diff Format displays the following types of changes:

- Additions
- Deletions
- Changes between the selected files

```

1 1  # $Revision: 1.1.3.6 $ $Date: Mon Dec 15 16:01:50 2008 $
2 2  #####
3 3  < # Copyright (c) 1997-2008 Dassault Systemes. All rights reserved.
3 3  > # Copyright (c) 1997-2011 Dassault Systemes. All rights reserved.
4 4  # Use of this source code is restricted to the terms of your
5 5  # license agreement with Dassault Systemes. Any use, reproduction,
6 6  # distribution, copying or re-distribution of this code outside
7 7  # the scope of that agreement is a violation of U.S. and International
8 8  # Copyright laws.
9 9  #####
10 10
11 11 - # modNotes.tcl
12 12 #
13 12 # Use this server-side script to create and attach a
14 14 < # note to various types of web objects. The coorpt
15 15 < # ProjectSync defined notes.
13 13 > # note to various types of web objects.
16 14 #
17 15 # Place the script in <SYNC_SITE_CUSTOM>/share/tcl or
18 16 # <SYNC_SITE_CUSTOM>/servers/<machine>/<port>/share/tcl
19 17 # and call the script using the URL script request or
20 18 # rstcl command below.
21 19 #
22 20 # URL script request:
23 21 # http://machine:port/scripts/isynch.dll?panel=TclScript&file=modNotes.tcl
24 22 #
25 23 # rstcl command:
26 24 # rstcl -server sync://<machine>:<port> -script modNotes.tcl
25 + #
26 + # This script creates ProjectSync defined notes.
27 + #
28 + #

```

## Line Numbers First Version

This column shows the line numbers from the first file version participating in the diff. When there is added text (text that appears in the second file version, but not the first version), this column is blank. .

## Line Numbers Second Version

This column shows the line numbers from the second file version participating in the diff. When there is deleted text (text that appears in the first file version, but not the second version), this column is blank. .

## Change type

This column shows the type of change represented in the files.

- + shows that the line was added to the file.
- - shows that the line was removed from the file.
- < shows the "old" version of a changed line (from the first version).
- > shows the "new" version of the changed line (from the second version).

## Diff results

This section shows a composite of the text from the files being compared, meaning that includes text from both file versions. Changed, Added, and Deleted text are indicated by the colors specified in the SyncAdmin GUI Diff settings in the **For 2-way Diffs (single window)** section. Deleted lines are also indicated with strike-through formatting on the text.

## Revised Diff Format Actions

### Next Diff

This option moves the window focus to next marked Diff in the file.

### Previous Diff

This option moves marked window focus to the previous marked Diff in the file.

### Next Conflict

This option is never enabled when the Revised Diff format is generated, because Revised Diff is a result format for 2-way diffs only. Conflicts cannot arise in a 2-way diff operation, so are never shown in the Revised Diff results.

### Previous Conflict

This option is never enabled when the Revised Diff format is generated, because Revised Diff is a result format for 2-way diffs only. Conflicts cannot arise in a 2-way diff operation, so are never shown in the Revised Diff results.

### Find

See Searching for Text.

## Related Topics

Reading Diff Results

Advanced Diff

Identifying Changed Objects

## Graphical Diff Utility

DesignSync contains a built-in graphical diff utility which can be used for file comparison and conflict resolution. This utility has the ability to highlight diffs and conflicts between the two versions of a selected file.

DesignSync also provides the ability to plug-in an additional Diff utility for file comparisons. Historically this has been used for a graphical Diff Utility. DesignSync provides a built-in graphical Diff utility, but users who are already comfortable with a specific Diff utility may prefer to configure DesignSync to use that tool. You configure your DesignSync client to use an external graphical Diff utility with a set of registry keys. For more information, see *DesignSync Data Manager Administrator's User's Guide*: DesignSync diff Display Registry Settings. The rest of this topic focuses on the built-in graphical Diff Utility.

**There are three ways to invoke graphical diff viewers:**

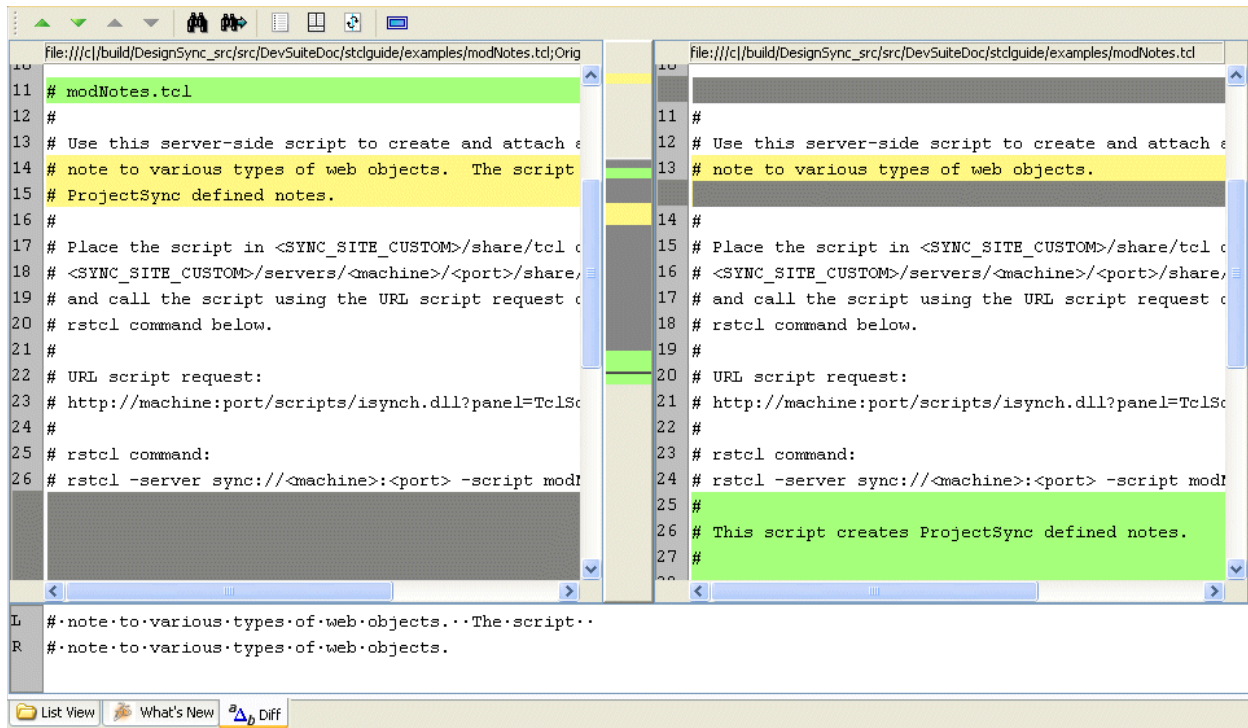
1. Run the Changed Objects Browser on a workspace folder. Select one of the objects in the **Changed** tab, bring up its context menu, and invoke the common diff operation that is listed on the context menu. SyncAdmin's Diff Format must be set to the default **Use graphical diff tool** for the built-in diff viewers to be used.
2. Select a text file in the View Pane and invoke a common diff operation from the **Tools => Compare Files** menu. SyncAdmin's Diff Format must be set to the default **Use graphical diff tool** for the built-in diff viewers to be used.
3. Select a text file in the View Pane and invoke **Tools => Compare Files => Advanced Diff**. In the Advanced Diff dialog, set the Display Options field to **Display output in GUI**.

The result of the diff operation is displayed in a new text tab labeled **Diff**.

## Using Graphical Diff format

The Diff results in a new text tab labeled **Diff**, in the View Pane. The Graphical Diff format shows both files, side-by-side, with the first file version on the left and the second

file version on the right. Between the files is a scroll-bar which highlights the location of the differences in the file. This central bar shows symbolically all differences in the file as well as the current vertical position of the left and right windows in respect to the file. The knob on the central bar represents the current left and right visible windows in relation to the whole file. The colored bands on the knob represent currently visible diff and conflict blocks. Clicking on the left mouse button with the cursor on the central bar moves the left and right window content to the corresponding vertical position.



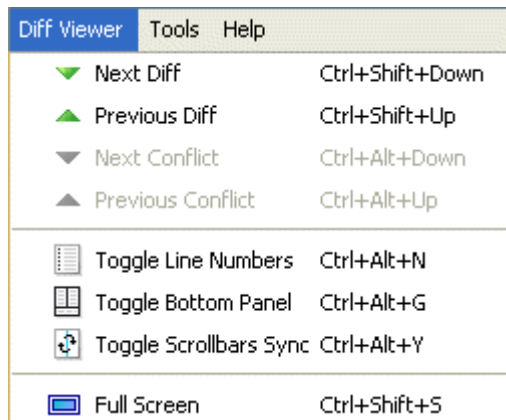
You can customize the colors that indicate what type of change you are viewing using SyncAdmin, in the **For multi-window Diff viewers/editors** section. By default, added lines are shown with light green background and changed lines are shown with yellow background.

### Notes

- When files are compared using the common ancestor option (a 3-way diff), if a line from the common ancestor has been changed in one version and deleted in another, the viewer shows the change present in **second** file, not the first. For example, if the line has changed in the first file and was removed in the second file, it is shown as removed in the Diff view.
- The empty lines filled with dark grey cannot be changed by the user. These lines are used to pad the smaller side of a block, so that the comparable file content stays in sync.

## Graphical Diff Format Tools

The graphical diff tool provides a set of actions to help you navigate the information provided in the graphical diff. The tools are available both from a **Diff Viewer** menu and a set of controls on the top of the Diff window.



### Next Diff

This option moves the window focus to next marked Diff in the file.

### Previous Diff

This option moves marked window focus to the previous marked Diff in the file.

### Next Conflict

This option moves the marked window focus to the next unresolved conflict in the file. This allows you to focus your attention on not what changed in the file, but where the conflicts are between the versions so you can resolve them. If there are no conflicts in the file, or you are at the last conflict in the file this option is grey and unclickable. To resolve conflicts, use the Conflict Editor.

### Previous Conflict

This option moves the marked window focus to the previous unresolved conflict in the file. This allows you to focus your attention on not what changed in the file, but where the conflicts are between the versions so you can resolve them. If there are no conflicts in the file, or you are at the first conflict in the file, this option is grey and unclickable. To resolve conflicts, use the Conflict Editor.

### Find

See Searching for Text.

### **Find Next**

Advances to the next instance of the text specified in the **Find** window.

### **Toggle Line Numbers**

By default, the line numbers display on the left side of the each panel. You can select this option to toggle whether line numbers display.

### **Toggle Bottom Panel**

By default, there is a bottom panel below the compare window that shows exactly what the change is, indicating which change is part of which version. The **L** and **R** labels in this window designate the Left and Right diff windows. You can select this option to toggle whether this window displays. Even if this window isn't displayed the GUI windows still show the Diff information.

### **Toggle Scrollbars Sync**

By default, the scrollbar between the two windows keeps the windows synchronized with each other. When you move forward in one window, DesignSync rolls the other window forward. You can select this option to enable or disable this feature. When the scrollbar sync is disabled, scrolling in one window does not advance the other window to match the content.

### **Full Screen**

By default, the Diff display opens in a tab in the View panel. This option allows you to toggle between that view and a Diff view that takes up the entire screen.

**Note:** The Full Screen display can be placed in the background to allow you to work in the DesignSync GUI before returning to the display.

### **Related Topics**

Advanced Diff

Identifying Changed Objects



# Displaying Information

## Showing Potential Checkouts

Sometimes, when you are working on a project, another user will check in new files into the vault. Often, you will use `populate` to keep your work area fully synchronized with the vault. However, in some circumstances you may want to only checkout some of the newly added files. The menu choice **Revision Control => Show Potential Checkouts** makes it easy to do this.

If you are viewing your work area in the List View, and you click **Revision Control => Show Potential Checkouts**, the List View is updated to include all of the objects that exist in the vault but not in the work area. Files and collections are described in the Type column as "Potential Checkout"; folders are described as "Potential Checkout Folder."

Checking out a potential folder creates a corresponding local folder in your workspace, as if you had used the `mkfolder` command. You can browse into the new workspace folder and perform revision control operations on it. For example, you can **Show Potential Checkouts** for the folder or `populate` it.

Once these objects are displayed in the List View, you can select them and choose the **Revision Control => Check Out** menu choice to check them out into your work area. Potential checkouts are only displayed in the List View until the List View is next refreshed by any operation.

### Notes:

- The **Show Potential Checkouts** menu choice operates on the object selected in the Tree View, regardless of the active selection.
- **Check Out** is the only operation permitted on possible checkout objects.
- **Show Potential Checkouts** is not applicable to module data.

### Related Topics

Checking out Design Files

ENOVIA Synchronicity Command Reference: `mkfolder`

ENOVIA Synchronicity Command Reference: `populate`

## Identifying Changed Objects

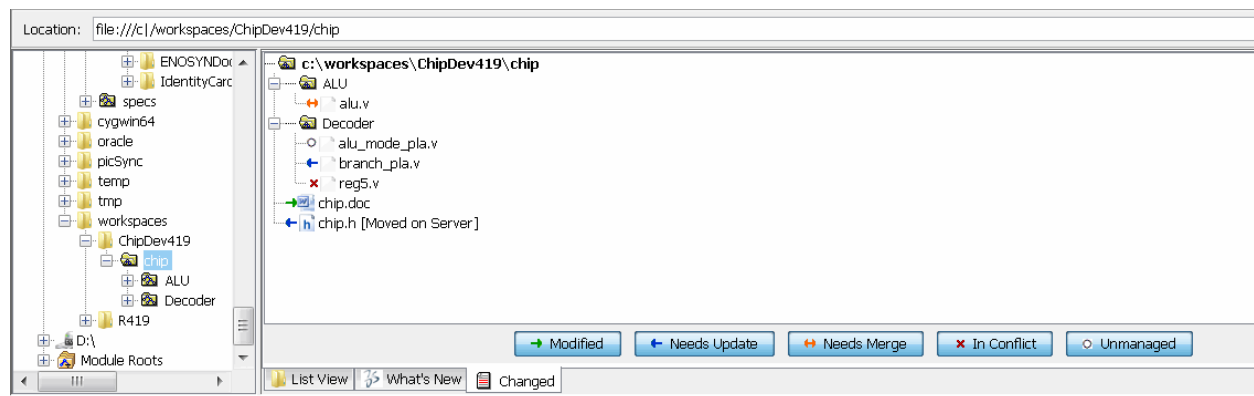
You can identify objects that you have modified in your work area by selecting **Tools => Reports => Changed Objects**. The **Changed Objects** report is also available from the



context menu when a folder is selected in the Tree View or List View. When you select this report, DesignSync performs a recursive search to find all managed objects that are changed in the specified folder and its subdirectories. The folder that the report starts from does not have to be the root of a work area. It can be any folder, even a folder that is above a workspace root directory.

When the search completes, DesignSync displays the changed objects in a directory tree structure, in a **Changed** tab. The root of the tree is the directory where you started the search. Subdirectories are shown only if those subdirectories contain modified or unmanaged objects. Using the context menu for folders, you can expand and collapse the displayed contents. Objects that have been changed are displayed under the directory in which they are located. The type of change is indicated by a change icon next to the file name. Structural changes will be identified with labels to the right of the filename identifying the structural change, such as "Moved Locally" or "Moved on Server".

**Click on the buttons in the following illustration for information.**



You can select objects in the display and operate on them through the menu or toolbar selections. You can also right-click on any object in the display to see a context menu showing common revision control operations used from this view.

For example, you select a Needs Merge object and right-click to select **Compare to Latest**. By default this displays differences in a **Diff** tab, using a 3-way Diff Viewer. You could select an In Conflict object and right-click to select **Resolve Conflicts**. This opens a Merge Conflict Editor which you use to resolve the conflicts.

If you select a folder and perform an operation recursively, it will operate on the folder's entire contents, as though you had selected the folder from the List View. The **Changed** tab's display is automatically updated when its selected files or folders are operated on.

### Modified

This button toggles the display to show or hide objects that have been locally modified. A new DesignSync session will use the button's last setting.

### **Needs Update**

This button toggles the display to show or hide objects with a more current version on the server. A new DesignSync session will use the button's last setting.

### **Needs Merge**

This button toggles the display to show or hide objects that require a merge (both local and server objects have been modified since the last workspace populate). A new DesignSync session will use the button's last setting.

### **In Conflict**

This button toggles the display to show or hide objects that have been merged and now contain unresolved conflicts. A new DesignSync session will use the button's last setting.

### **Unmanaged**

This button toggles the display to show or hide unmanaged objects in the workspace. Some files may be pre-excluded from checkin by matching a pattern specified in an exclude file. Excluded files do not display with the unmanaged objects. For more information on exclude files, see [Working with Exclude Files](#). A new DesignSync session will use the button's last setting.

### **Related Topics**

[Graphical Diff Utility](#)

[Merge Conflict Editor](#)

[Populating Your Work Area](#)

## **Displaying Contents of Vault Data**

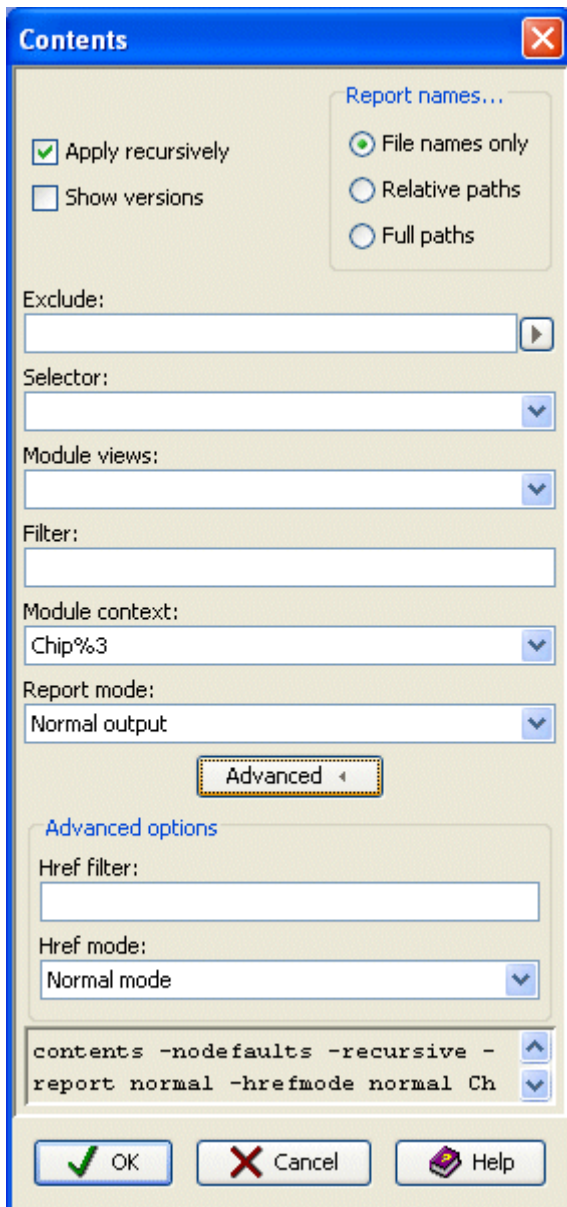
The **Contents** report lists data belonging to either a DesignSync configuration or to a module. The report is available when any of these objects are selected:

- a workspace folder associated with a DesignSync vault
- a server folder
- a module folder (within the context of a module)
- a module
- a module branch

- a module version (if the report is run for a module version, the data items comprising that module version are shown)

**Tools => Reports => Contents** opens the **Contents** dialog box. Click **OK** to display the results in a new text tab labeled **Contents**, in the View Pane.

**Click on the fields in the following illustration for information.**



## Contents Field Descriptions

### Show versions

Show the version numbers of the objects reported. If the report is run on a module version, the member version numbers for the data items in a module version are reported.

### Report names

Specify how the name of each object within a directory is shown:

- File names only - Show only the object name; present its parent directory as an absolute path (default)
- Relative paths - Show the path to the object as relative to where the command was started
- Full paths - Show the absolute path to the object

### Selector

Identify which version of an object to report.

For a DesignSync folder, select **Get selectors** to display configurations of the associated workspace vault. Choose any selector except **Date()** and **VaultDate()**. If a selector is not specified, the current selector is used for a workspace folder, and `Trunk:Latest` used for a server folder.

For a module, if a selector is not specified, the current selector is used for a workspace module, and `Trunk:Latest` used for a server module.

### Module views

See Module views field.

### Report verbosity

The level of additional information reported:

**Normal output:** Include header information and progress lines. This is the default output mode.

**Verbose output:** For DesignSync folders, include information about configuration mappings.

### Href mode

When reporting on a module recursively, how hierarchical references should be evaluated. This field is only available when reporting on module data. If a workspace module is being reported on, and a **Selector** is not specified, then the hierarchy in the workspace is followed. In that case, the **Href mode** is ignored.

**Normal mode:** Uses the **Change traversal mode with static selector on top level module** set in SyncAdmin to determine how hrefs are followed. If a reference resolves to a static version, the hrefmode is set to 'static' for the next level of submodules to be populated. (Default)

**Static mode:** Report on the static version of the sub-module that was recorded with the href at the time the parent module's version was created.

**Dynamic mode:** Evaluate the **Selector** to identify the version of the sub-module to report on.

## Related Topics

ENOVIA Synchronicity Command Reference: contents Command

Recursion option

Exclude field

Filter field

Module context field

Hreffilter field

Command Invocation

Command Buttons

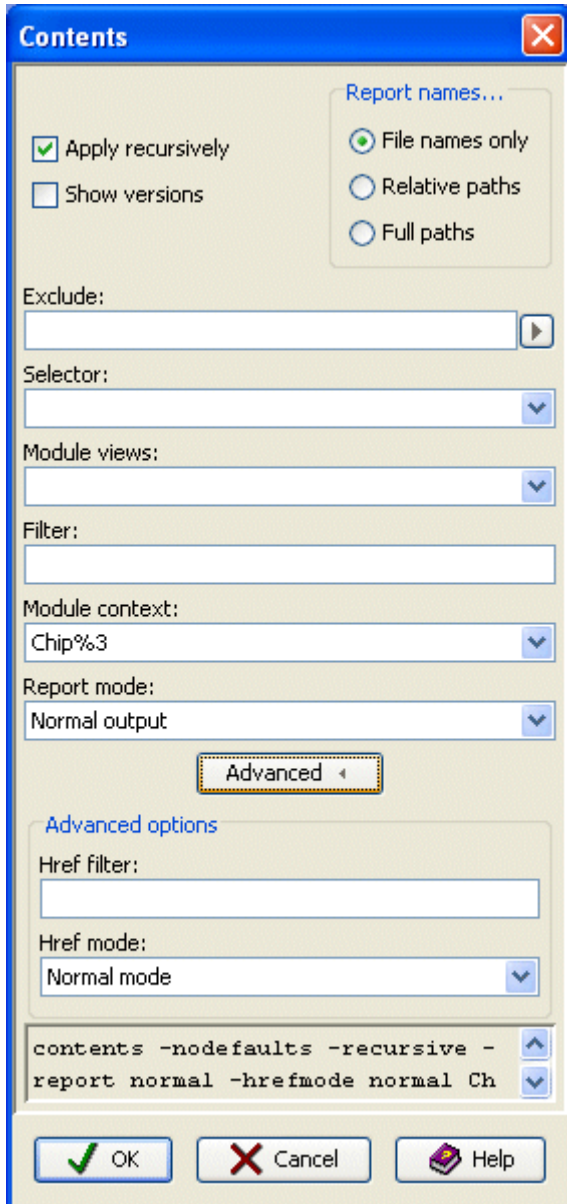
## Displaying Contents of Vault Data

The **Contents** report lists data belonging to either a DesignSync configuration or to a module. The report is available when any of these objects are selected:

- a workspace folder associated with a DesignSync vault
- a server folder
- a module folder (within the context of a module)
- a module
- a module branch
- a module version (if the report is run for a module version, the data items comprising that module version are shown)

**Tools => Reports => Contents** opens the **Contents** dialog box. Click **OK** to display the results in a new text tab labeled **Contents**, in the View Pane.

**Click on the fields in the following illustration for information.**



## Contents Field Descriptions

### Show versions

Show the version numbers of the objects reported. If the report is run on a module version, the member version numbers for the data items in a module version are reported.

### Report names

Specify how the name of each object within a directory is shown:

- File names only - Show only the object name; present its parent directory as an absolute path (default)
- Relative paths - Show the path to the object as relative to where the command was started
- Full paths - Show the absolute path to the object

## Selector

Identify which version of an object to report.

For a DesignSync folder, select **Get selectors** to display configurations of the associated workspace vault. Choose any selector except **Date()** and **VaultDate()**. If a selector is not specified, the current selector is used for a workspace folder, and `Trunk:Latest` used for a server folder.

For a module, if a selector is not specified, the current selector is used for a workspace module, and `Trunk:Latest` used for a server module.

## Module views

See Module views field.

## Report verbosity

The level of additional information reported:

**Normal output:** Include header information and progress lines. This is the default output mode.

**Verbose output:** For DesignSync folders, include information about configuration mappings.

## Href mode

When reporting on a module recursively, how hierarchical references should be evaluated. This field is only available when reporting on module data. If a workspace module is being reported on, and a **Selector** is not specified, then the hierarchy in the workspace is followed. In that case, the **Href mode** is ignored.

**Normal mode:** Uses the **Change traversal mode with static selector on top level module** set in SyncAdmin to determine how hrefs are followed. If a reference resolves to a static version, the hrefmode is set to 'static' for the next level of submodules to be populated. (Default)

**Static mode:** Report on the static version of the sub-module that was recorded with the href at the time the parent module's version was created.

**Dynamic mode:** Evaluate the **Selector** to identify the version of the sub-module to report on.

## Related Topics

ENOVIA Synchronicity Command Reference: contents Command

Recursion option

Exclude field

Filter field

Module context field

Hreffilter field

Command Invocation


Command Buttons

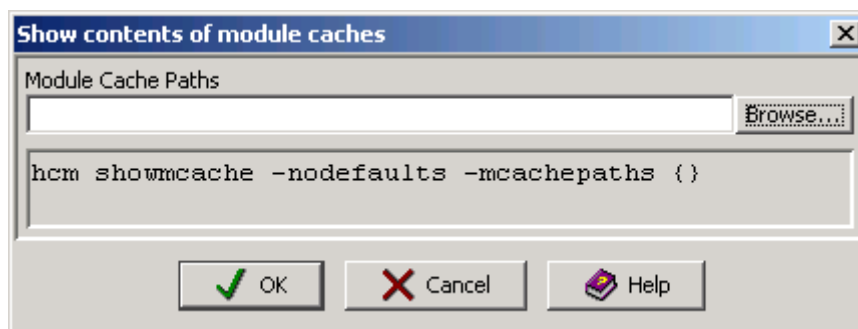
## Displaying a Module Cache

The show module cache functionality in the DesignSync GUI helps you to determine whether or not a module is already in an module cache (mcache) before you:

- Fetch the module into it
- Removing the module from it.

**To show the contents of a module cache:**

1. From the main menu, highlight the module.
2. Select **Modules => Show =>  Module Cache**. The Show contents of module caches dialog box displays



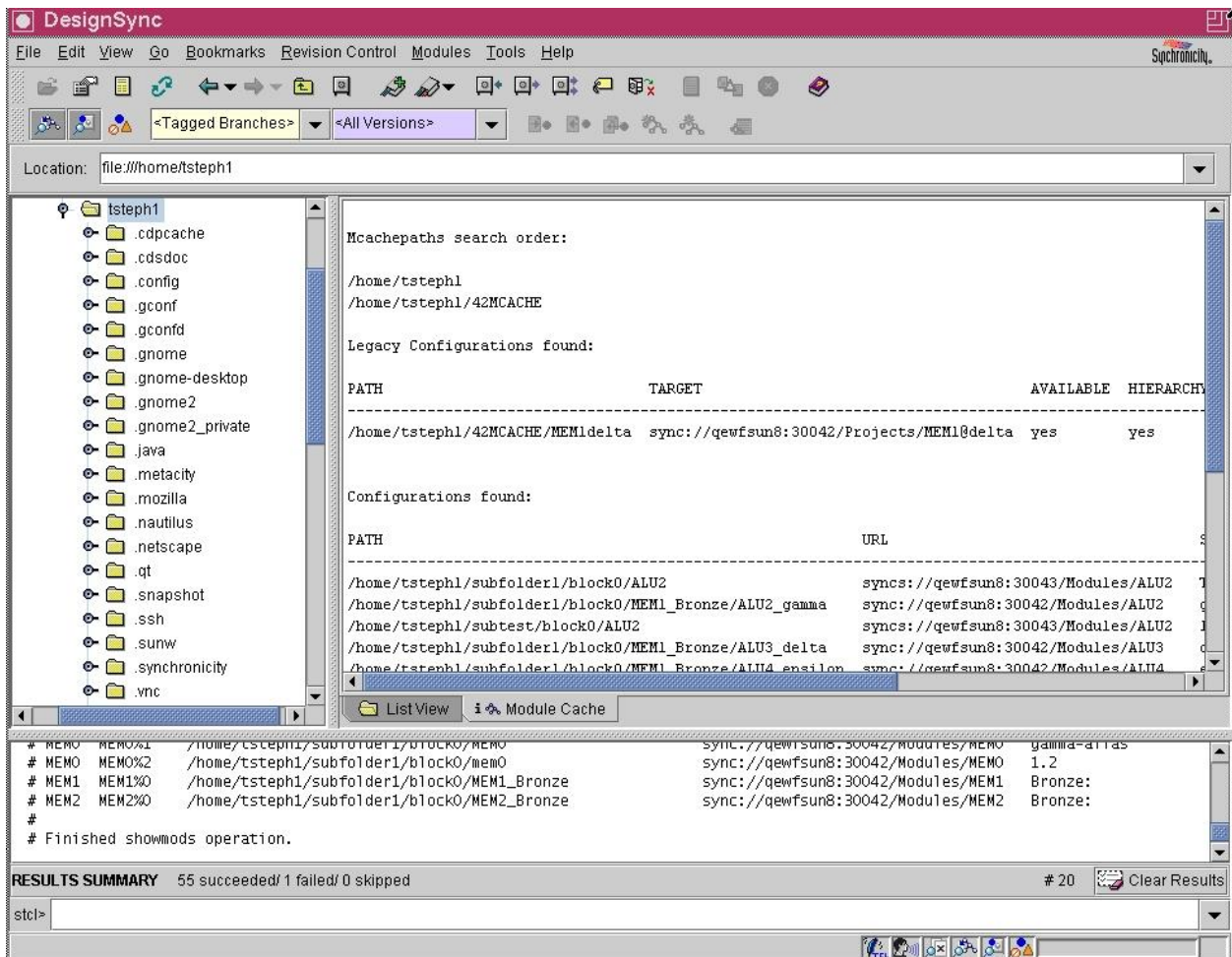
3. The module cache path is pre-filled in with the value of the default module cache path as defined by your project lead. You can accept this path, you can enter a



different module caches path, or you can click **Browse** to browse to the correct path.

**Note:** The paths must exist. If this field contains an empty string, an error is displayed.

4. Click **OK**. The results from the Show a Module Cache command is displayed in a new text tab labeled **Module Cache** in the list view.




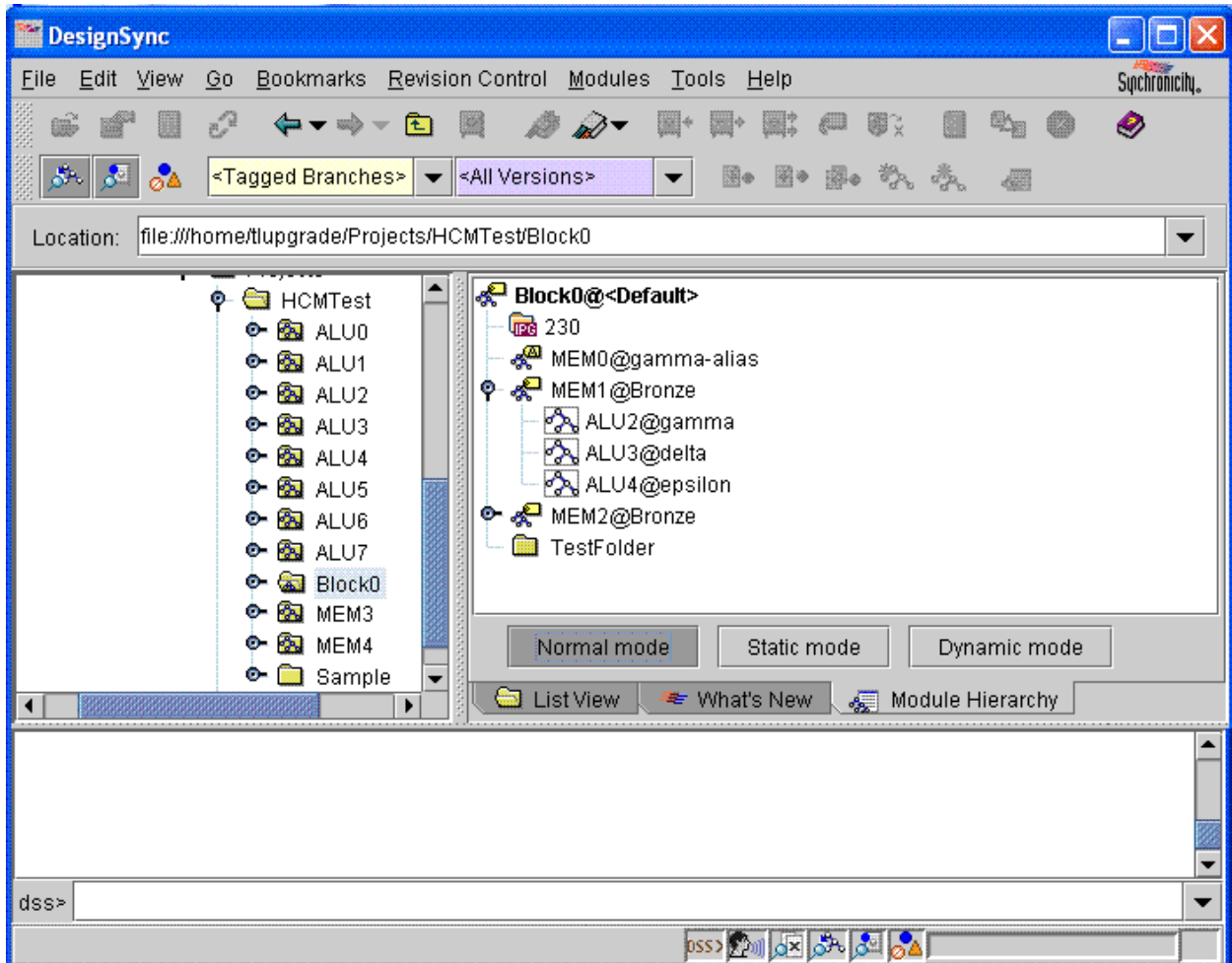
## Displaying Module Hierarchy

The Show Module Hierarchy command is available when any one of the following objects is highlighted:

- Legacy module configuration
- Legacy module alias
- Legacy module release
- Module version on server
- Legacy module base directory

- A 5.0 module base directory in client work area

The results from the Show Module Hierarchy command is displayed in a new text tab labeled  **Module Hierarchy** in the list view pane.



When viewing the hierarchy of 5.0 module, you can choose one of three display options from buttons at the bottom of the Module Hierarchy tab. These are:

- **Normal mode:** The behavior in normal mode is dependant on whether the registry setting "Change traversal mode with static selector on top level module" is enabled or disabled. If the traversal mode is disabled (default), the top level module is evaluated according to the selector; if the selector is dynamic, it is evaluated dynamically, if the selector is static, it is evaluated statically. Subsequent levels in the hierarchy are evaluated statically. If the traversal mode is enabled, all levels are evaluated statically. For more information setting the traversal mode, see the *ENOVIA Synchronicity DesignSync Data Manager Administrator's Guide: Modules*. For more information on understanding the module hierarchy, see *Module Hierarchy*.

- **Static mode:** The static version is the version that was recorded at the time each hierarchical reference was created from a parent module to a sub-module
- **Dynamic mode:** The dynamic version is that version that shows of the module's hierarchical references are displayed for all levels of the hierarchy. Selectors associated with all hierarchical references are evaluated individually to identify the version to be displayed.

The default display is Normal mode. This command is available from the context menu when a node highlighted is in the module hierarchy tree. These options are not applicable for legacy modules.

## To display module hierarchy

1. Highlight the node for which you want to see the module hierarchy. You can see Module Hierarchy for these legacy module objects:
  - Server Side – Module Alias
  - Server Side – Module Release
  - Server Side – Module Configuration
  - Client Workspace – Configuration Base

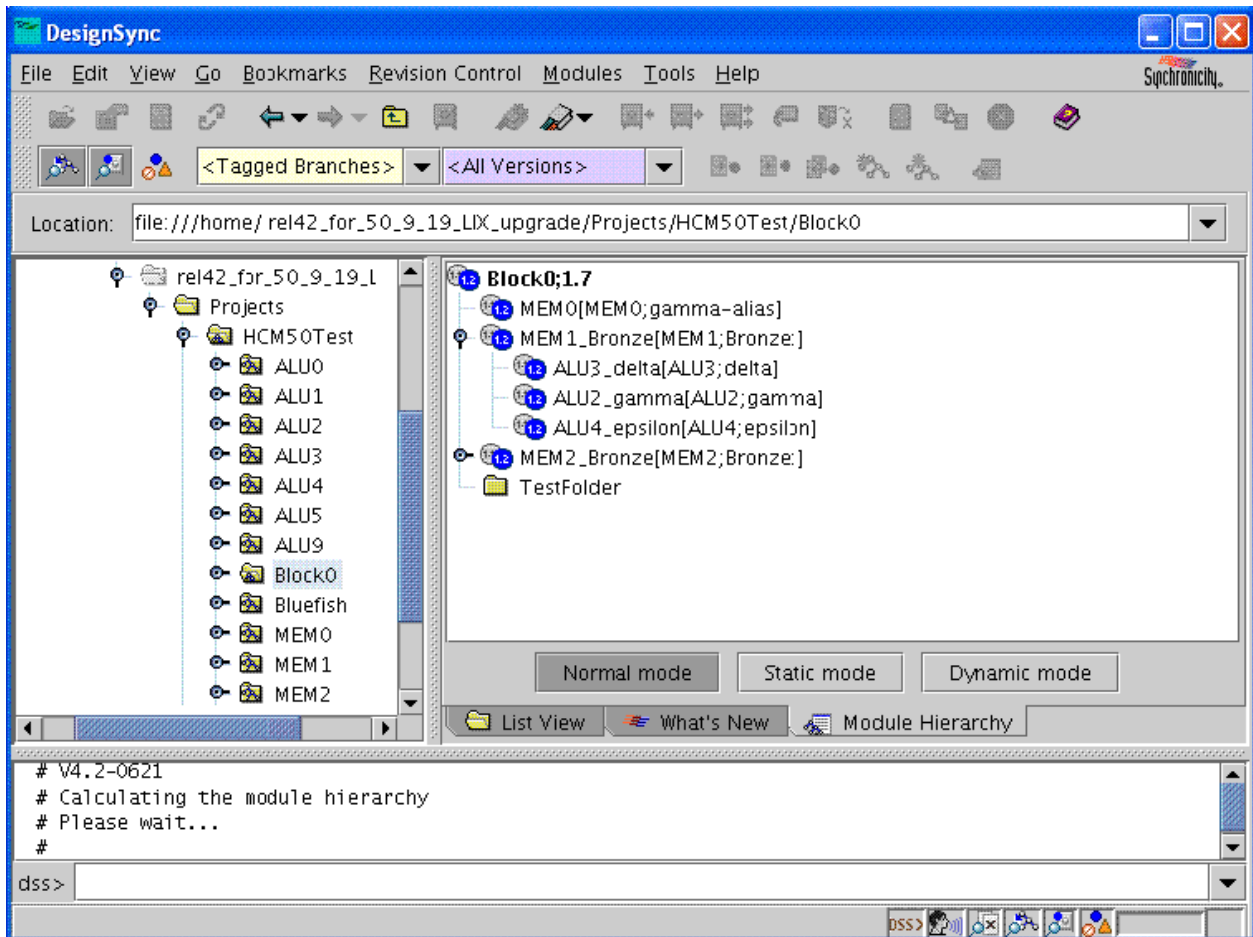
For legacy modules, you can highlight either Server side – Module Version or Client Workspace – Module base objects.

2. From the main menu, select **Modules => Show =>  Module Hierarchy.**

## Results

### For legacy objects:

- The text displayed for a configuration shows both the module name and the configuration name such as `module@config`.
- When it is a default configuration, `module@<Default>` displays.
- DesignSync vault nodes displays the vault name.
- IP Gear Deliverables displays the Deliverable Number. Mouse-overs on the nodes in the module hierarchy tree display the full object URL.



## For current objects:

**Note:** If current module contains hierarchical references to a legacy module, the Module Hierarchy tab displays the information on the objects as described above for legacy objects.

- The top level node in the hierarchy displays in the version filter format that is currently selected in the GUI.
- For sub-modules, the text for a module version displays both the module name and the module version number or the module tag name depending on which mode is used when the hierarchy is browsed and the type of the displayed hierarchical reference.
- For a static hierarchical reference, the module name and version displays.
- For dynamic hierarchical references, the tag name displays. In cases where the hierarchical reference was created by using a selector list, the module name and resolving tag also displays.
- For a DesignSync vault, the vault name displays.
- For an IP Gear deliverable, the Deliverable Number displays.

Mouse-overs on the nodes in the module hierarchy tree display the full object URL.

When you are browsing from a module base folder in the workspace, and there are multiple modules based at the folder, the Select Module Context dialog appears so you can select the module on which to show the hierarchy.

## Related Topics

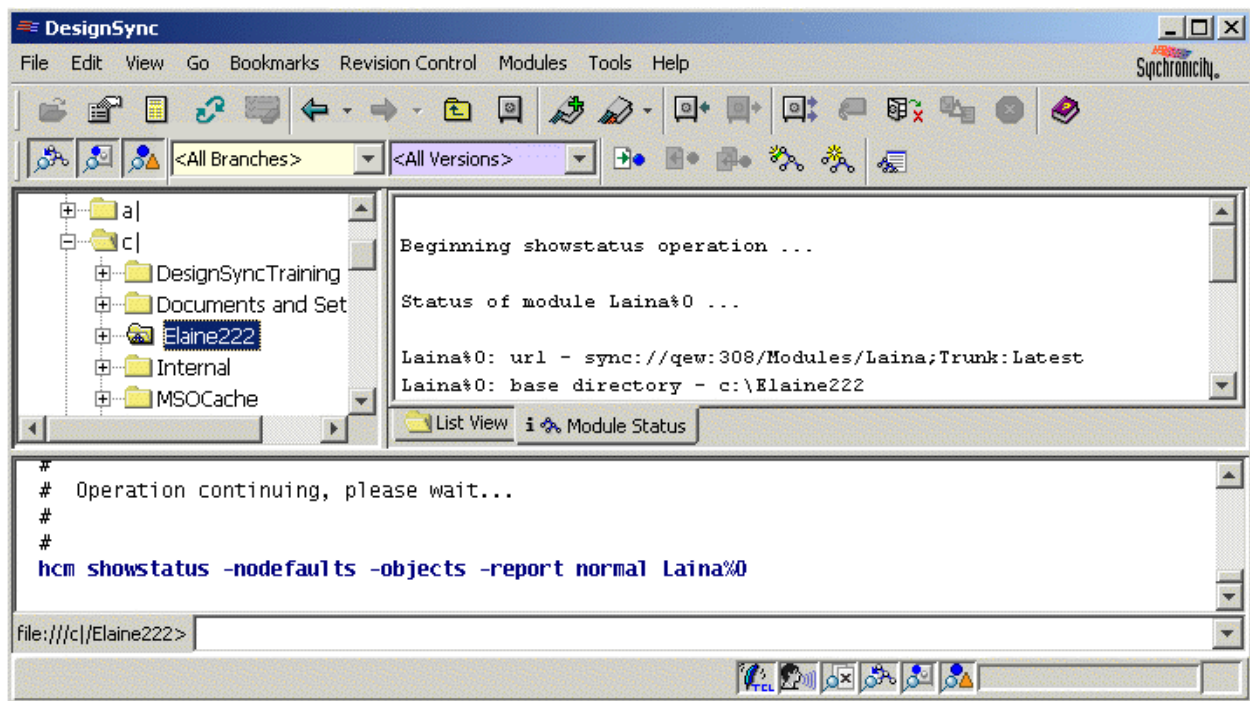
Module Hierarchy

Context menu

Module Context Field

## Displaying Module Status


The Show status dialog is available when a legacy module configuration or current module-base node (tree or list view) is selected in the client work area. The results from the show status command is displayed in a new text tab labeled **Module Status**.



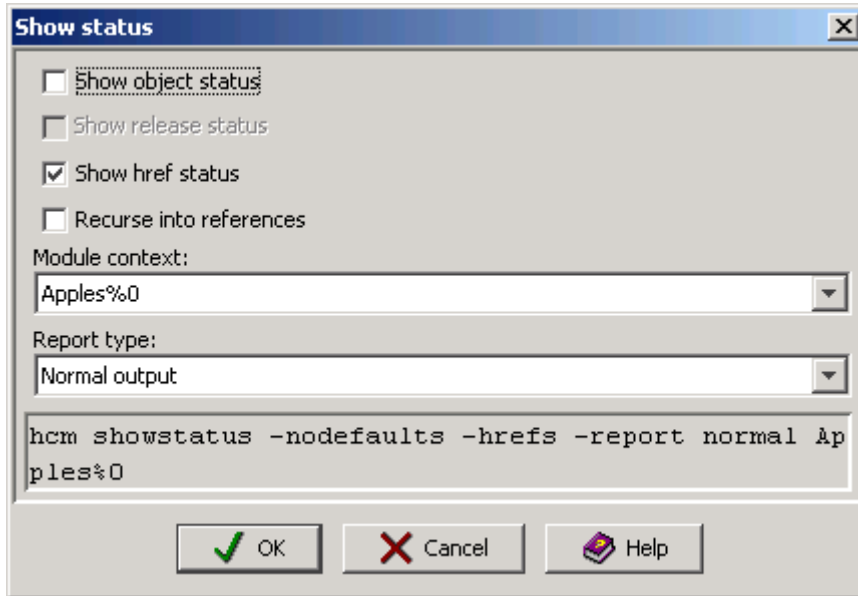
The Module Status tab displays:

- Updated module versions for dynamic hierarchical references.
- New hierarchical references.
- New module members in the modules.
- Old module members removed from the module.
- Removed or added hierarchical references or legacy modules.

To show the status of a module:

1. From the main menu, highlight the module.
2. Select **Modules => Show =>  Module Status.**
3. Select options as needed.
4. Click **OK**.

Click on the fields in the following illustration for information.



### Show object status

When checked, the status of each object in the workspace is compared with the server module. This status displays in the Module Status tab.

When not checked, only the status of the module and the status of its hierarchical references displays in the Module Status tab.

By default, this option is unchecked.

### Show release status

This option is only available when a legacy module base directory is selected in the client work area.

When checked, a recursive report will stop at released modules These items are not displayed on the Module Status tab:

- a release
- an alias

- an IP Gear deliverable
- a vault folder residing on a server without modules support
- a reference that does not exist locally

By default, this option is unchecked.

**Note:** To display the current status of the hierarchical references of releases in your work area, both the Show release status option and the Show object status option must be selected. Otherwise, the status of hierarchical references of releases is always listed as up-to-date.

### Show href status

When checked, the hierarchical references are verified to determine if the reported status is current. When not checked, the hierarchical references are not verified.

By default, this option is unchecked.

### Recurse into references

When checked, the Module Status tab displays the status for the specified module and all referenced modules.

**Note:** Hierarchical references that point to IP Gear deliverables, servers that do not support modules, aliases, or references are only displayed if that item exists locally.

When not checked, the Module Status tab displays the status for the specified module only.

By default, this option is unchecked.

### Module context

This field is available when a non legacy module base folder is selected. The list box has the available module instances for the base folder listed in alphabetical order.

### Report type

Select the type of report from the pull down list to be displayed on the Module Status tab. The default choice is Normal output. There are four report modes:

- **Brief output** – Displays a summary and lists hierarchical references that are out-of-date. Lists file status for files that are out-of-date. Also displays a table of conflicts if conflicts exist between the expected submodule and the actual submodule.

- **Normal output** – Displays the status of the hierarchical references and file status for the module. Displays a table of conflicts if conflicts exist between the expected submodule and the actual submodule.
- **Summary output** – Displays the target and base directory of the module, the status of each module, and the overall status of the module in the workspace. Also displays a table of conflicts if conflicts exist between the expected submodule and the actual submodule.
- **Verbose output** – Displays the status of the hierarchical reference, additional information about whether the hierarchical references need updating, and file status for the module. Also displays a table of conflicts if conflicts exist between the expected submodule and the actual submodule.

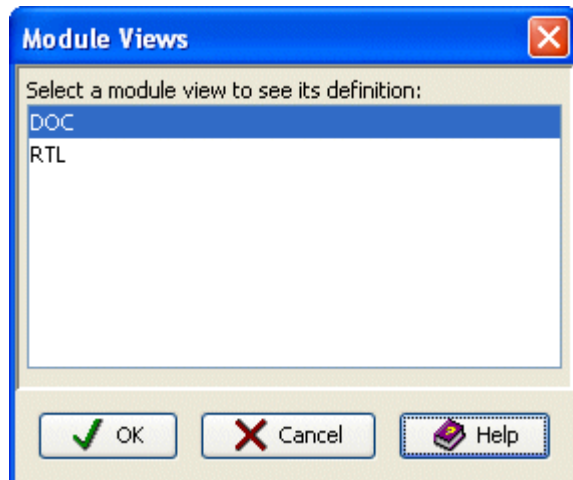
**Note:** You can set the `-report normal` mode to report on the "needs update" status of hierarchical references with the `ShowHrefsNeedCheckinStatus` registry key. For more information on setting the registry key, see the *DesignSync Administrator's Guide*.

## Displaying Module Views

When you select a module view, you can open it by double-clicking it or selecting **Show | Modules Views** from the **Modules** menu in order to view the definition.

The definition displays in a Module View Panel in the View Pane.

When more than one view definition is appropriate, DesignSync displays a chooser window to allow you select the module to view.



Select the desired module view and press **OK**, or **Cancel** to exit the chooser window and cancel the display of a module view definition.

## Displaying Module Where Used



The Module Where Used report displays a list of modules containing a hierarchical references to the specified DesignSync object. This allows you to easily identify which modules contain a particular referenced object version. This functionality is particularly useful when a defect is identified in a referenced object and you want to trace it and see what versions of the software contained that code.

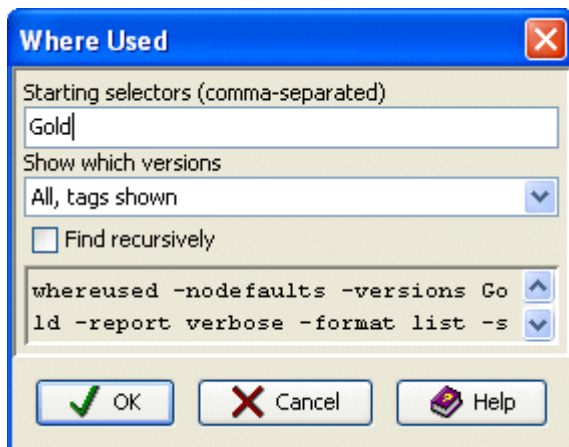
The Module Where Used report is available when anyone of the following objects is highlighted on the server:

- Module or Module version(s) on server
- Legacy module configuration(s)
- Legacy module alias(es)
- Legacy module release(s)
- Legacy module base directory
- Vault folder

## Running the Where Used command

**Modules =>Show =>Where Used** opens the **Where Used** dialog box. Click **OK** to display the results in a new text tab labeled **Where Used**, in the View Pane.

Click on the fields in the following illustration for information.



### Starting Selectors (comma-separated)/Starting configurations (comma-separated)

Specify a comma-separated list of desired version numbers, selectors, or configurations. If you have a module, vault folder, or legacy module selected, this field is populated with the default selector (Trunk:Latest for modules and vault folders and <Default> for legacy modules). If you select one or more specific versions or configurations, the field is populated with those version and is not editable.

#### Notes:

- If the selector field is empty when the command is submitted, DesignSync automatically uses the default selector.
- The title of the field changes depending on the type of selected object. When a module configuration is selected, the field title is **Starting configurations (comma-separated)**, otherwise the field title is **Starting Selectors (comma-separated)**.

### Show which versions

Specify a filter to control the information received using the drop down list options:

- **All, tags shown** - Displays all tags and all reference locations, including references that are not tagged. (Default)
- **All, tags not shown** - Displays all reference locations, but does not display tag information.
- **Only with immutable tags** - Displays only reference locations tagged with an immutable tag and the name of the immutable tag.

**Note:** Using the **Only with immutable tags** option may not display all versions in which an immutable tag is used. The where used command automatically filters the display from the starting point until it reaches the last immutable tag in a reference tree.

- **Only with version tags** - Displays any reference location that has a version tag and the name of the tag.

### Find recursively

Determines whether to show the locations in which the version is explicitly referenced, or show all modules in which the version is implicitly or explicitly referenced. An explicit reference exists when there is a direct reference link between the module and the target. An implicit reference exists when the module and target are not directly connected, but within the module's hierarchy exists a reference to the target. For example: if the Chip module references the Gold version of the ALU module, and the Gold version of the ALU module references the Gold version of the ROM module, the Chip module contains an implicit reference to the ROM module and an explicit reference to the ALU module.

If selected, the command output shows both implicit and explicit references to the specified version. If not selected, the output shows only explicit references. (Default)

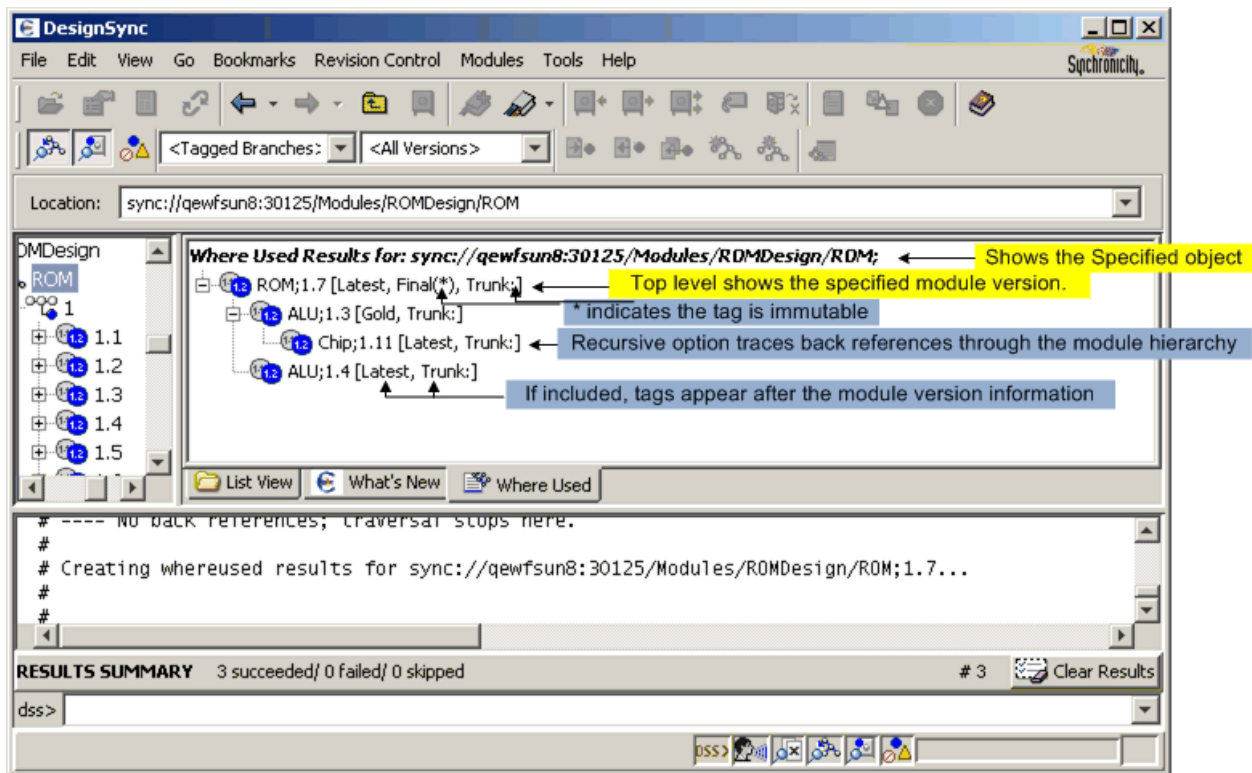
### Understanding the Where Used command output

The Where Used output is displayed in the **Where Used** tab in the View pane.

The selected object is the first object in the Where Used list. If it is referenced by a module an Expand/Collapse button allows you follow the references.

Tags, if included, appear in a comma-separated list within square brackets (**[*version*tag, *branch*tag, *immutable*version\*]**) after the module version name. Branch tags are indicated by a trailing colon (:). Immutable tags are indicated with a trailing asterisk (\*).

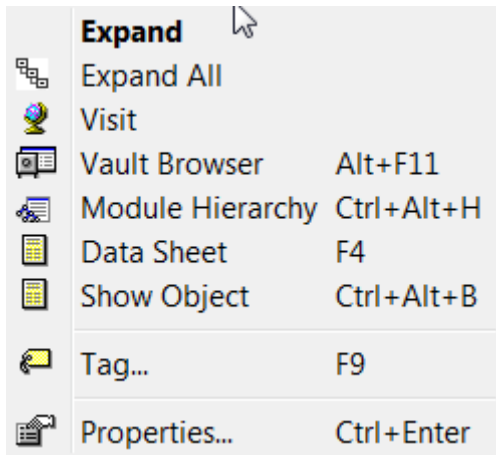
The following example has the following options set, All, tags shown and Recursive. The example traces where the ROM version tagged "Final" is used.



## Where Used Actions

You can select one or more objects in the vault browser and launch the context menu to perform DesignSync operations directly on the selected objects.

Click on the fields in the following illustration for information.



### Vault browser object context menu

<b>Action/Option</b>	<b>Result</b>
<b>Expand/Collapse</b>	Expands or collapses the selected module version.
<b>Expand All</b>	Expands or collapses all the module versions.
<b>Visit</b>	Switches to the List View and displays the object versions on the server.
<b>Vault Browser</b>	Opens the vault browser view with the selected object as the initial object version.
<b>Module Hierarchy</b>	Opens the module hierarchy view for the selected module version.
<b>Data Sheet</b>	Opens the data sheet for the object.
<b>Show Object</b>	Opens the object revisions on an enterprise server associated with a DesignSync module object in the default web browser.
<b>Tag</b>	Tags the selected module version.
<b>Properties</b>	Opens the properties pages for the object.

## Displaying Enterprise Objects

This command shows the object revisions on an enterprise server associated with a DesignSync module object in the default web browser. This provides a quick method to update information on an enterprise object, for example, a defect, immediately after checking in the related code.

This command is active when a user selects one of the following module objects:

- Workspace module instance
- Server module branch
- Server module version

The Enterprise object is associated with the DesignSync server using the SyncAdmin interface. For more information on configuring Enterprise server associations, see the *DesignSync Administrator's Guide: ENOVIA Servers*.

## Compare the Contents of Two Areas

The **Compare** report compares the contents of two areas.

The report is available when any of these objects are selected:

- a workspace folder associated with a DesignSync vault
- a server folder
- a module folder (within the context of a module)
- a module

You may also invoke the Compare dialog without first selecting any objects. The objects to compare are then selected on the dialog itself.

### Notes:

- The contents of a module version can be compared against the contents of another version of the same module, or against a workspace that contains a version of that module.
- In comparing collections, the compare operation compares only collections and not collection members.
- To display differences between files, DesignSync provides Diff tools available from Tools =>Compare Files and Tools =>Compare Files =>Advanced Diff.
- If you have moved module members in the workspace, they will appear twice in the compare output, once in their original location and once in their new location with "First only" and "Second only" status values.

**Tools =>Reports =>Compare** opens the **Compare** dialog box. Click **OK** to display the comparison results in a new text tab labeled **Compare**, in the View Pane.

**Click on the fields in the following illustration for information.**

Compare Workspaces/Selectors ✕

**Operation:**  
Compare workspace to vault ▾

**Workspace Path:**  
D:\build\Specs Browse...

**Alternate Path:**  
Browse...

**First Selector:**  
▾

**Second Selector:**  
▾

**Module views:**  
▾

**Filter:**

**First module context:**  
Specs%0 ▾

**Second module context:**  
▾

**Second href mode:**  
Normal mode ▾

Advanced ▾

**Advanced options**

<input checked="" type="checkbox"/> Apply recursively	<input type="checkbox"/> Show objects that are the same
<input type="checkbox"/> Show history	<input type="checkbox"/> Show object paths

**Exclude:**

**Href filter:**

**First href mode:**  
Normal mode ▾

**Report verbosity:**  
Normal output ▾

compare -nodefaults -recursive -report normal file:///d:/build/Specs/Specs%0

✓ OK ✕ Cancel 📖 Help

## Compare Workspaces/Selectors Field Descriptions

### Operation

Choose from four options:

- Compare workspace to vault
- Compare workspace to another path
- Compare workspace to a selector
- Compare two selectors of a workspace's vault

### Workspace Path

Enter the workspace path you want to use in the Compare action. Click **Browse...** to display the **Select Path** dialog and choose your workspace.

### Alternate Path

This option is available when you select the "Compare workspace to another path"  
**Operation.** Enter the path you want to use for comparison. **Click Browse...** to display the **Select Path** dialog and choose your workspace.

### First Selector

This option is available when you select the "Compare workspace to a selector"  
**Operation.** Enter the selector you want to use for the comparison. For a DesignSync folder, select **Get selectors** to display configurations of the associated workspace vault. Choose any selector except **Date()** and **VaultDate()**.

### Second Selector

This option is available when you select the "Compare two selectors of a workspace's vault"  
**Operation.** Enter the second selector you want to use for the comparison. For a DesignSync folder, select **Get selectors** to display configurations of the associated workspace vault. Choose any selector except **Date()** and **VaultDate()**.

### Module views

See Module Views Field.

### Filters

See Filters Field.

### First module context

This option is available when you select a single module base folder to compare. The module instance names based at the module base folder specified in the **Workspace Path** field are listed. A **first module context** is required, when comparing a single module base folder.

### **Second module context**

This option is available when you select two module base folders to compare. The module instance names based at the module base folder specified in the **Alternate Path** field are listed. When comparing two module base folders, the **First module context** and **Second module context** are both optional. Selecting the empty entry results in a folder centric comparison; the module base folders are used for the comparison. Selecting a module context results in a module centric comparison; the selected module contexts are used for the comparison.

### **Apply Recursively**

See Recursive Option.

### **Show history**

Report the checkin comment and other details for the version history back to the common ancestor of the two versions reported.

### **Show objects that are the same**

Report items that are the same version, in addition to items that are different.

### **Show object paths**

Specify how the name of each object within a directory is shown. If selected, show the path to the object as relative to where the command was started. If not selected, show only the object name, and present the object's parent directory as an absolute path. Showing only the object name is the default.

### **Exclude**

See Exclude Field.

### **Href Filter**

See Href Filter Field.

### **First href mode**



This option is available when you select the "Compare workspace to a selector"

**Operation.** When reporting on a module recursively, this field specifies how hierarchical references should be evaluated. This field is only available when reporting on module data. If a workspace module is being reported on, and a **Selector** is not specified, then the hierarchy in the workspace is followed. In that case, the **Href mode** is ignored.

**Normal mode:** Uses the **Change traversal mode with static selector on top level module** set in SyncAdmin to determine how hrefs are followed. If a reference resolves to a static version, the hrefmode is set to 'static' for the next level of submodules to be populated. (Default)

**Static mode:** Report on the static version of the sub-module that was recorded with the href at the time the parent module's version was created.

**Dynamic mode:** Evaluate the **Selector** to identify the version of the sub-module to report on.

### Second href mode

This option is available when you select the "Compare two selectors of a workspace's vault" **Operation.** When reporting on a module recursively, this field specifies how hierarchical references should be evaluated. This field is only available when reporting on module data. If a workspace module is being reported on, and a **Selector** is not specified, then the hierarchy in the workspace is followed. In that case, the **Href mode** is ignored.

**Normal mode:** Uses the **Change traversal mode with static selector on top level module** set in SyncAdmin to determine how hrefs are followed. If a reference resolves to a static version, the hrefmode is set to 'static' for the next level of submodules to be populated. (Default)

**Static mode:** Report on the static version of the sub-module that was recorded with the href at the time the parent module's version was created.

**Dynamic mode:** Evaluate the **Selector** to identify the version of the sub-module to report on.

### Report verbosity

The level of additional information reported:

**Brief output:** Include header information and progress lines. Directories that contain only items in one of the areas being compared, or for which all items in the two areas being compared are identical, are not expanded to show their contents.

**Normal output:** Expand directories that would be skipped in **Brief output** mode, because they are present in one of the areas being compared, or because all items in the two areas being compared are identical. This is the default output mode.

**Verbose output:** For DesignSync folders, include information about configuration mappings.

### Related Topics

ENOVIA Synchronicity Command Reference: compare Command

Filter field

Module Views\_Field

Module context field

Recursion option

Exclude field

Href filter field

Command Invocation

Command Buttons

## Compare the Contents of Two Areas

The **Compare** report compares the contents of two areas.

The report is available when any of these objects are selected:

- a workspace folder associated with a DesignSync vault
- a server folder
- a module folder (within the context of a module)
- a module

You may also invoke the Compare dialog without first selecting any objects. The objects to compare are then selected on the dialog itself.

### Notes:

- The contents of a module version can be compared against the contents of another version of the same module, or against a workspace that contains a version of that module.
- In comparing collections, the compare operation compares only collections and not collection members.
- To display differences between files, DesignSync provides Diff tools available from Tools =>Compare Files and Tools =>Compare Files =>Advanced Diff.
- If you have moved module members in the workspace, they will appear twice in the compare output, once in their original location and once in their new location with "First only" and "Second only" status values.

**Tools =>Reports =>Compare** opens the **Compare** dialog box. Click **OK** to display the comparison results in a new text tab labeled **Compare**, in the View Pane.

**Click on the fields in the following illustration for information.**

Compare Workspaces/Selectors ✕

**Operation:**  
Compare workspace to vault

**Workspace Path:**  
D:\build\Specs Browse...

**Alternate Path:**  
Browse...

**First Selector:**  
▼

**Second Selector:**  
▼

**Module views:**  
▼

**Filter:**

**First module context:**  
Specs%0 ▼

**Second module context:**  
▼

**Second href mode:**  
Normal mode ▼

**Advanced** ▼

**Advanced options**

<input checked="" type="checkbox"/> Apply recursively	<input type="checkbox"/> Show objects that are the same
<input type="checkbox"/> Show history	<input type="checkbox"/> Show object paths

**Exclude:**  
▶

**Href filter:**

**First href mode:**  
Normal mode ▼

**Report verbosity:**  
Normal output ▼

compare -nodefaults -recursive -report normal file:///d:/build/Specs/Specs%0

OK Cancel Help

## Compare Workspaces/Selectors Field Descriptions

### Operation

Choose from four options:

- Compare workspace to vault
- Compare workspace to another path
- Compare workspace to a selector
- Compare two selectors of a workspace's vault

### Workspace Path

Enter the workspace path you want to use in the Compare action. Click **Browse...** to display the **Select Path** dialog and choose your workspace.

### Alternate Path

This option is available when you select the "Compare workspace to another path" **Operation**. Enter the path you want to use for comparison. **Click Browse...** to display the **Select Path** dialog and choose your workspace.

### First Selector

This option is available when you select the "Compare workspace to a selector" **Operation**. Enter the selector you want to use for the comparison. For a DesignSync folder, select **Get selectors** to display configurations of the associated workspace vault. Choose any selector except **Date()** and **VaultDate()**.

### Second Selector

This option is available when you select the "Compare two selectors of a workspace's vault" **Operation**. Enter the second selector you want to use for the comparison. For a DesignSync folder, select **Get selectors** to display configurations of the associated workspace vault. Choose any selector except **Date()** and **VaultDate()**.

### Module views

See Module Views Field.

### Filters

See Filters Field.

### First module context

This option is available when you select a single module base folder to compare. The module instance names based at the module base folder specified in the **Workspace Path** field are listed. A **first module context** is required, when comparing a single module base folder.

### **Second module context**

This option is available when you select two module base folders to compare. The module instance names based at the module base folder specified in the **Alternate Path** field are listed. When comparing two module base folders, the **First module context** and **Second module context** are both optional. Selecting the empty entry results in a folder centric comparison; the module base folders are used for the comparison. Selecting a module context results in a module centric comparison; the selected module contexts are used for the comparison.

### **Apply Recursively**

See Recursive Option.

### **Show history**

Report the checkin comment and other details for the version history back to the common ancestor of the two versions reported.

### **Show objects that are the same**

Report items that are the same version, in addition to items that are different.

### **Show object paths**

Specify how the name of each object within a directory is shown. If selected, show the path to the object as relative to where the command was started. If not selected, show only the object name, and present the object's parent directory as an absolute path. Showing only the object name is the default.

### **Exclude**

See Exclude Field.

### **Href Filter**

See Href Filter Field.

### **First href mode**

This option is available when you select the "Compare workspace to a selector"

**Operation.** When reporting on a module recursively, this field specifies how hierarchical references should be evaluated. This field is only available when reporting on module data. If a workspace module is being reported on, and a **Selector** is not specified, then the hierarchy in the workspace is followed. In that case, the **Href mode** is ignored.

**Normal mode:** Uses the **Change traversal mode with static selector on top level module** set in SyncAdmin to determine how hrefs are followed. If a reference resolves to a static version, the hrefmode is set to 'static' for the next level of submodules to be populated. (Default)

**Static mode:** Report on the static version of the sub-module that was recorded with the href at the time the parent module's version was created.

**Dynamic mode:** Evaluate the **Selector** to identify the version of the sub-module to report on.

### Second href mode

This option is available when you select the "Compare two selectors of a workspace's vault" **Operation.** When reporting on a module recursively, this field specifies how hierarchical references should be evaluated. This field is only available when reporting on module data. If a workspace module is being reported on, and a **Selector** is not specified, then the hierarchy in the workspace is followed. In that case, the **Href mode** is ignored.

**Normal mode:** Uses the **Change traversal mode with static selector on top level module** set in SyncAdmin to determine how hrefs are followed. If a reference resolves to a static version, the hrefmode is set to 'static' for the next level of submodules to be populated. (Default)

**Static mode:** Report on the static version of the sub-module that was recorded with the href at the time the parent module's version was created.

**Dynamic mode:** Evaluate the **Selector** to identify the version of the sub-module to report on.

### Report verbosity

The level of additional information reported:

**Brief output:** Include header information and progress lines. Directories that contain only items in one of the areas being compared, or for which all items in the two areas being compared are identical, are not expanded to show their contents.

**Normal output:** Expand directories that would be skipped in **Brief output** mode, because they are present in one of the areas being compared, or because all items in the two areas being compared are identical. This is the default output mode.

**Verbose output:** For DesignSync folders, include information about configuration mappings.

### Related Topics

ENOVIA Synchronicity Command Reference: compare Command

Filter field

Module Views\_Field

Module context field

Recursion option

Exclude field

Href filter field

Command Invocation

Command Buttons

## Displaying Version History

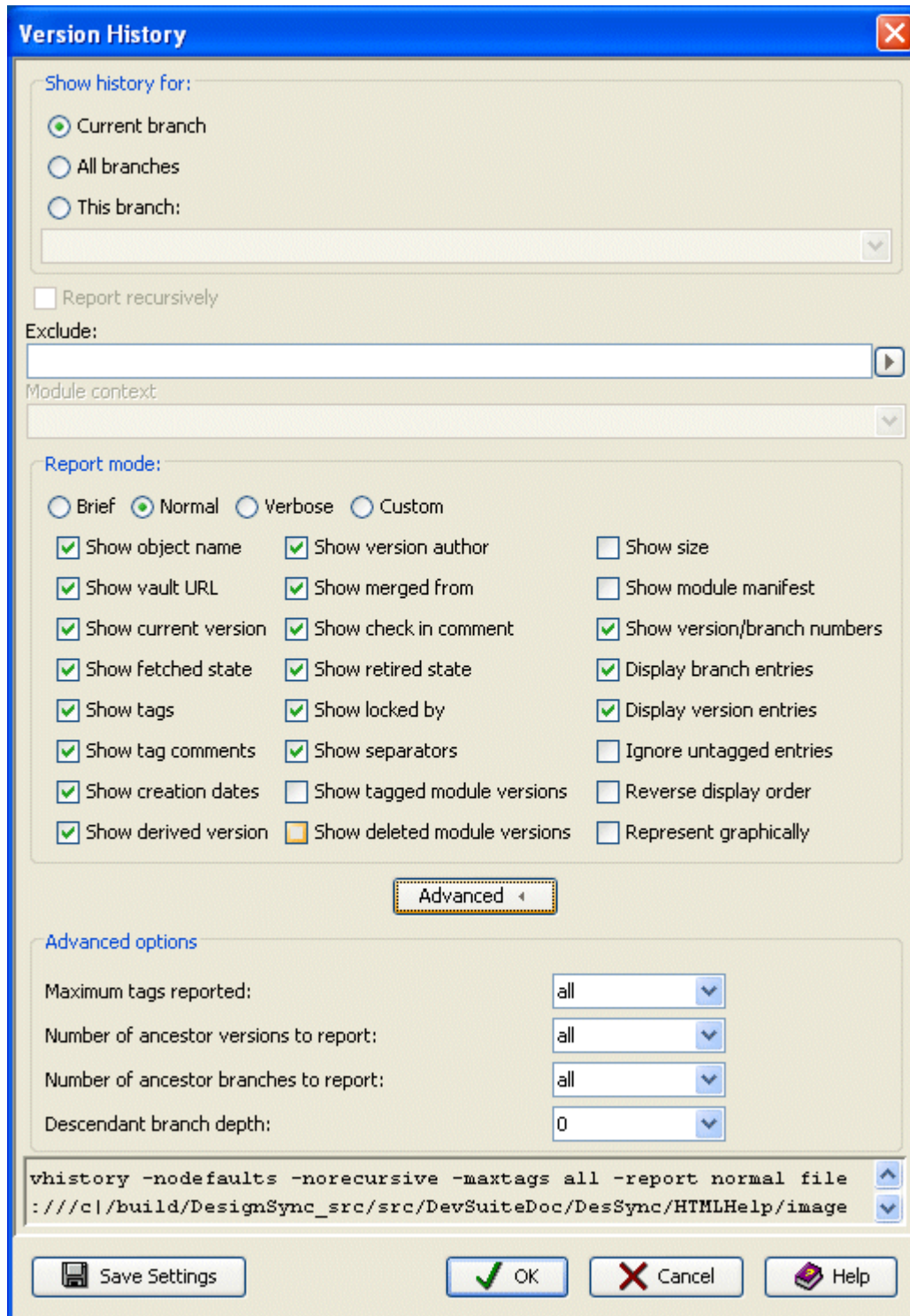
The Version History report displays version history for managed objects. If the history report is run from a workspace, local status is also reported.

**Tools =>Reports =>Version History** opens the **Version History** dialog box. This report is available when files, folders or the base directory of a module is selected in a local workspace. The report is also available when a server object is selected (vaults, versions, branches, modules, module versions, or module branches).

Click **OK** to display the results in a new text tab labeled **Version History**, in the View Pane.

**Click on the fields in the following illustration for information.**





## Version History Field Descriptions

### Show history for

The default setting shows the history of the **Current branch** (the branch of the object that is in the workspace). You can also show the history for **All branches** of the object, or of a specific branch (**This branch**).

To select the history of a particular branch, select "Show history for This branch". The field below becomes active and you can type the branch name or click the pull-down menu to select a branch.

### Report recursively

Whether to descend through sub-folders of the starting folder, or only report on the objects in the selected folder. This option is available for local folders or server (non-module) folders. The option is unchecked by default.

### Exclude filter

See Exclude Filter.

### Module context

Specifies the module context. Use this option to identify a module member that is not in the workspace or to restrict the report to module versions that affect any of the members specified on the command line.

### Report mode

The type of information that will be reported. The choices **Brief**, **Normal**, and **Verbose** represent defined reports. Selecting one of these defined reports automatically enables all of the report options that comprise the selected mode. Report options not in the selected mode are automatically disabled. The **Normal** report mode is selected by default. Select the **Custom** report mode to specify your own combination of report options.

### Show object name

Show the workspace path to the object, or to the vault URL.

### Show vault URL

Show the vault URL associated with a workspace object.

### Show current version

Show the version currently in the workspace.

### Show fetched state

Show the fetched state in the workspace.

**Show tags**

Show branch and version tags. Immutable tags are shown with "(immutable)" appended.

**Show tag comments**

Show the comments associated with version and branch tags. Tag comments are only available for module data.

**Show creation dates**

Show a version's creation date.

**Show derived version**

Show the numerical parent version. This maintains the continuity between versions for merge and rollback operations.

**Note:** If a merge, skip, rollback or overlay operation occurs to create this version, the referenced version is shown as "Merged from" version.

**Show version author**

Show a version's author.

**Show merged from**

Show the version used to create the from current version when the current version was created as the result of a rollback, merge, skip, or overlay operation requiring an alternate parent version.

**Show check in comment**

Show a version's check in comments, and any checkout comments. For DesignSync objects, checkout comments are only visible from the workspace in which the checkout occurred. For module objects, the branch lock comment is visible to all users.

**Show retired state**

Show whether a branch is retired, the username of the user who retired the file, and the date and time of the retire. This is not applicable to module data, so is not reported for module data.

**Note:** You must also select the Display Branch entries in order to view the retired information.

### **Show locked by**

Show the lock owner of a locked branch. For DesignSync objects, also show the "version -> upcoming version" information.

### **Show separators**

Show separators between items and versions.

### **Show tagged module versions**

Show module version that have tags, even if a module member being queried version has not been changed in that module version.

### **Show deleted module versions**

Show module version that were purged or deleted.

### **Show size**

Show the size of the object version in KB.

Note: Collections and module versions, both of which contain more than one object, display with a size of zero.

### **Show module manifest**

For a module, show the manifest of changes in each version. For a module member, show only the changes to that member.

### **Show version/branch numbers**

Show the version number for versions, and the branch number for branches. For branches, indicate whether any versions exist on the branch.

### **Display branch entries**

Show information for branch objects.

### **Display version entries**

Show information for version objects.

**Ignore untagged entries**

Do not show entries that have no tags.

**Reverse display order**

Show the versions/branches in reverse numeric order.

**Represent graphically**

Show a graphical representation of the version history, as a text graph.

**Maximum tags reported**

The maximum number of tags shown for any object. You can select a value from the pull-down list, or type in a positive integer. By default, all tags are shown. This option is only available when Show tags is selected.

**Number of ancestor versions to report**

How many versions back to report. By default, all versions on a branch are reported. You can select a value from the pull-down list, or type in a positive integer. Specifying the number of ancestor versions to report sets the Descendant branch depth value to 0. This option is only available when Show history for **Current branch** or **This branch** is selected.

**Number of ancestor branches to report**

How many branches back to report. By default, only versions on the specified branch are reported. You can select a value from the pull-down list, or type in a positive integer. Specifying the number of ancestor branches to report sets the Descendant branch depth value to 0. This option is only available when Show history for **Current branch** or **This branch** is selected.

**Descendant branch depth**

The number of levels of descendant branches to report, from the starting branch. By default, the report is limited to the starting branch (a value of 0). You can select a value from the pull-down list, or type in a positive integer. Specifying a descendant branch depth sets the Number of ancestor versions to report and the Number of ancestor branches to report to **all**.

**Related Topics**

ENOVIA Synchronicity Command Reference: vhistory Command

Exclude field

Command Invocation

Command Buttons


## Controlling the Display of Module Information

### Displaying module versions

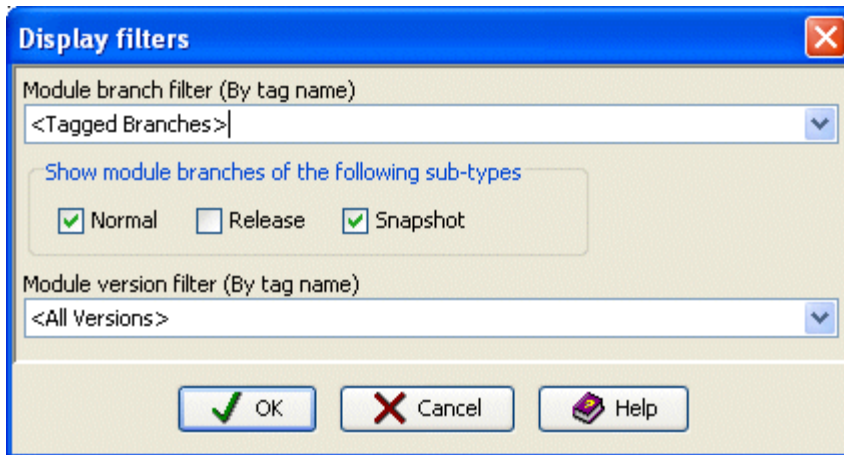
By default, the last 50 versions on a module branch are shown. The maximum number of versions displayed is configurable. For details, see SyncAdmin Help: Modules Options.

### Using Display Filters

The Display Filters dialog box is used to set filters that control what is displayed in the tree and list view.

1. From the main menu, select **View =>**  **Display Filters**.
2. Select options as needed.
3. Click **OK**.

Click on the fields in the following illustration for information.



### Display Filter Field Descriptions

#### Module branch filter

Enter or select the module branch filter. Valid values can be any glob-style patterns, <All Branches>, or <Tagged Branches>. The pull down list contains the two default values and the last five glob-style pattern values used.

When viewing modules on a server, the branches shown in the tree and list view are filtered to only show branches that have any tag that matches the filter. The default choice is <Tagged Branches>.

### **Show module branches of the following sub-types**

Select the object sub-type by which to filter branches when browsing the server. You must have at least one branch sub-type selected.

### **Module version filter**

Enter or select the module version filter. Valid values can be any glob-style patterns, <All Versions>, or <Tagged Versions>. The pull down list contains the two default values and the last five glob-style pattern values used.

When viewing modules on a server, the versions shown in the tree and list view are filtered to only show versions that have any tag that matches the filter. The default choice is <All Versions>.

### **Related Topics**

[Tagging Versions and Branches](#)

[ENOVIA Synchronicity Command Reference: tag Command](#)

[SyncAdmin Help: Modules Options](#)

[SyncAdmin Help: Tags](#)

## **Exploring Modules**

The Modules Explorer is headed by the Module Roots folder node. DesignSync adds a module workspace root node to the Modules Explorer:

- When you navigate to a folder in the Tree View that contains at least one workspace module root.
- When you select a client-side module base directory from the Bookmarks menu.
- When you enter a url in the Location field to navigate to a client-side module base directory.

These module roots remain in the Modules Explorer until you exit the GUI.

Expanding a module root in the Tree View reveals all of the module instances included in the module root directory. Expanding a module instance reveals all the folders that are members of that module instance.

When you select a module instance in the Tree View, the List View displays the module members and hierarchy that belong to that module instance.

### **To specify a module workspace root node to always include in the Modules Explorer:**

1. In the Tree view, highlight the module root to include.
2. Select **Modules | Add Initial Module Root**.

### **To remove a module workspace root node from the Modules Explorer:**

1. In the Tree view, highlight the module root to remove.
2. Select **Modules | Remove Initial Module Root**.

Note: If you select a module object in a subsequent session, the module root will be added to the Modules Explorer for the duration of that session.

## **Annotate Tool**

### **Using Annotate**

The Annotate tool provided in the **Tools => Annotate** menu graphically displays the selected text file object annotated with the following information:

- Last-modified version.
- Author credited with the changes.
- Date the modification was checked in.

The Annotate results in a new text tab labeled **Annotate**, in the View Pane.

### **To open a file in annotate:**

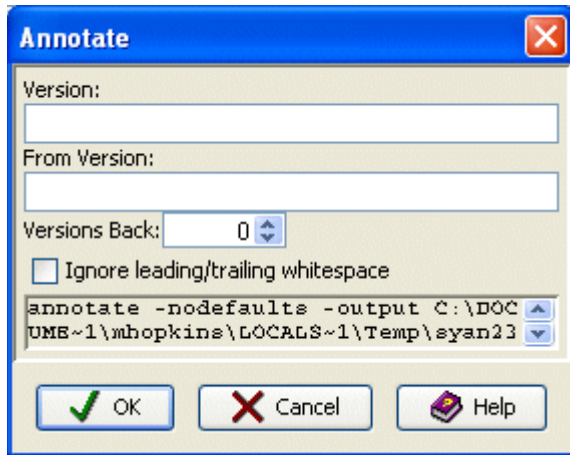
1. Select a text file in the View Pane. The file can be in the local workspace, or a server-side version.
2. Select **Tools | Annotate** to open the Annotate dialog box.

The result of the Annotate operation is displayed in a new text tab labeled **Annotate**.



3. Optionally use the Annotate tools to locate or highlight information in the Annotate tab.

Click on the fields in the following illustration for information.



### Version

Type the version selector of a file to display.

If no version is specified, DesignSync uses the version loaded in the workspace. (Default)

A non-module DesignSync file object can take any valid single selector. A module member must be specified by the module member version number.

### From Version

Specifies the selector of the first version to consider when creating the annotated document. The versions included in the annotation begin with the specified version and end with the version that resolves to the selector specified with the **From Version** option. The specified selector must resolve to a version on a path from the annotated version to the vault root.

#### Notes:

- When referring to a module member by version number, use the module version number.
- If neither the Versions Back nor the **From Version** option is specified, the annotate includes the entire object history, beginning with the vault root. (Default)

### Versions Back

Specifies the number of versions to consider when creating the annotated document. The versions included in the annotation begin with the specified version and each version is processed until the specified number of versions back is reached, then the annotated file is generated.

**Note:** If neither the **Versions Back** nor the From Version option is specified, the annotate includes the entire object history, beginning with the vault root. (Default)

### **Ignore leading/trailing whitespace**

Determines whether white space (spaces, tabs) differences at the beginning or end of a line are ignored. Select this option to ignore leading and trailing whitespaces. Leave this option unselected to treat leading and trailing whitespace changes as significant. (Default)

For example, if the most recent change to a line was to replace multiple spaces with a single tab, selecting this option ignores that change and uses the last textual change (or in-line whitespace change) to determine the last modification information.

### **Related Topics**

Annotate Actions

Highlighting the Annotate Results








Common Diff Operations

## **Annotate Actions**

### **Annotate Actions**

Annotate provides some tools to allow you to locate information in the annotated file. While you're in the Annotated tab, you can launch the context menu to perform DesignSync operations directly on the selected object or to find strings within the file, or highlight certain information.

**Click on the fields in the following illustration for information.**

 Copy	Ctrl+C
Select All	Ctrl+A
 Find...	Ctrl+F
 Find Next	F3
 Save...	Ctrl+S
 Visit	
 Properties...	Ctrl+Enter
 Highlight...	
Previous Highlight	
Next Highlight	

<b>Action/Option</b>	<b>Result</b>
<b>Copy</b>	Copies the selected text into the paste buffer. You can select text with the mouse or use Select All to select the full contents of the window.  <b>Note:</b> You cannot paste into the Annotate tab. It is read only. You can save the annotated file and edit it.
<b>Select All</b>	Select all the text in the Annotate tab.
<b>Find</b>	Enter a search string to locate particular text within the Annotate tab.
<b>Find Next</b>	Finds the next instance of the search string specified for Find.
<b>Save</b>	Saves the annotated version of the selected file to the file name you specify. Conventionally, annotated files should be saved with the .ann extension, which, by default is excluded from DesignSync checkin operations.
<b>Visit</b>	Switches to the List view and displays the object version containing the selected line. This option is only active if one line is selected.
<b>Properties</b>	Opens the properties pages for the selected version. This option is only active if one line is selected.
<b>Highlight</b>	Opens the Highlight dialog box.
<b>Previous Highlight</b>	Moves your focus in the Annotate window to the previous cluster of highlighted text.
<b>Next Highlight</b>	Moves your focus in the Annotate window to the next cluster of highlighted text.

**Related Topics**

## Using Annotate

### Highlighting the Annotate Results

## Highlighting the Annotate results

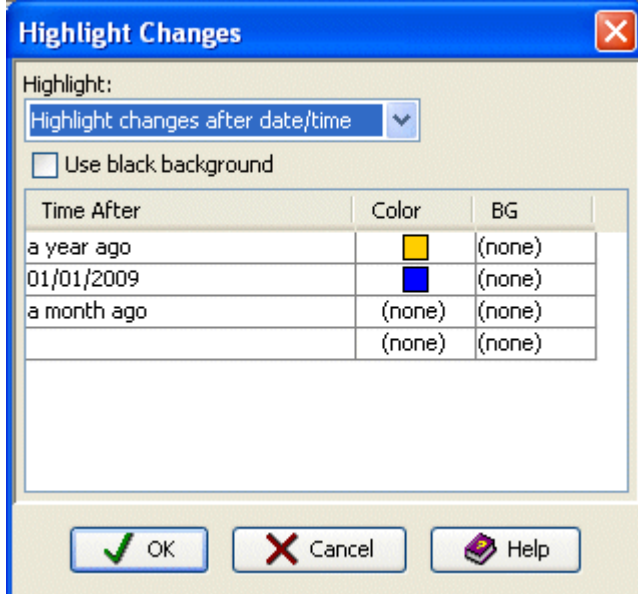
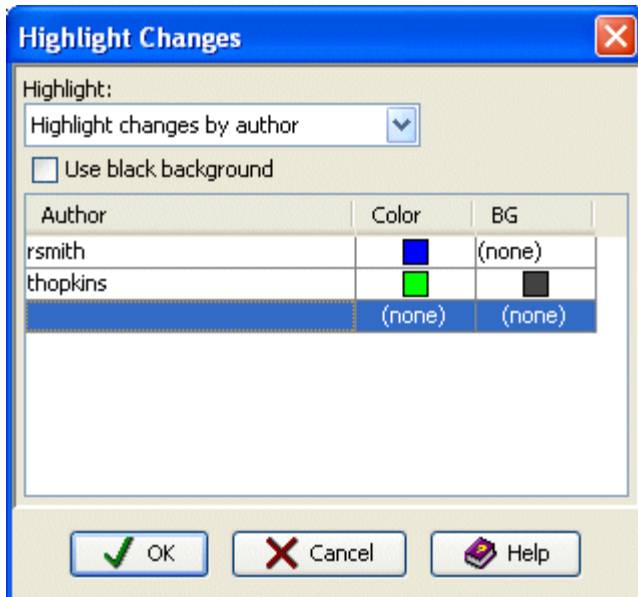
You can highlight key information in the annotated file to allow you to easily locate important changes. Using the highlight option, you can track different conditions, changing the text color or the background color to easily locate key information in the file. For example, if you have a regression and you've identified the file responsible, you can use a date or version selector to isolate the information that changed in the file.

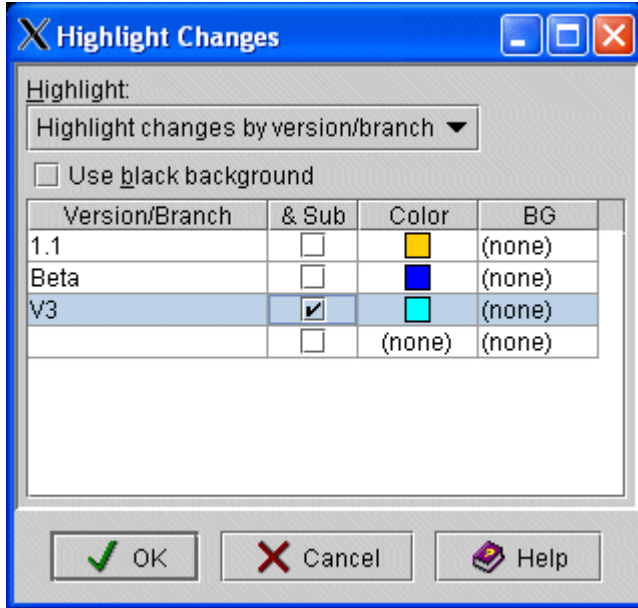
You can highlight based on any the following information:

- By date
- By version
- By author

**Note:** You may select as many of the same type of change as you like, but you may only select one type of change to apply at a time. For example, you could highlight changes by three different authors with different colors, but you cannot specify a version to highlight as well.

**Click on the fields in the following illustration for information.**





## Highlight

- **No Highlighting**

Select this to remove any already set highlight conditions. (Default)

- **Highlight changes by author**

Select this to apply highlight conditions by username of the version author.

- **Highlight changes after date/time**

Select this to apply highlight conditions by date or time AFTER the selected date or time.

- **Highlight changes by version/branch**

Select this to apply highlight conditions by version or branch name or numeric.

## Use black background

Sets the Annotate tab window to white text on a black background, instead of the default black text on a white background. You can also set the background color of lines that meet particular conditions, by selecting the color in the conditions list.

## Conditions list

- **Highlight changes by author**

Type username(s) of the author(s) to highlight and the highlight color options.

- **Highlight changes after date/time**

Type date(s). Any lines after the selected date are highlighted with the selected highlight color options. To select a period between two dates, select the color options you want for the start date and no color settings for the end date. The date can be specified in any valid format for a date specified as a DesignSync selector.

- **Highlight changes by version/branch**

Type the version or branch selector to highlight. You can use a tag or a numeric version of branch number.

- **& Sub**

Select this option to include any lines changed in the sub-branches of the selected branch or version in the highlight condition.

## Vault Browser Tool

### Vault Browser Overview

The Vault Browser tool provided in the **Tools => Vault Browser** menu graphically displays the genealogy of a DesignSync object by displaying branches and versions as objects. The Vault Browser also provides easy access to the important operations on object versions, such as viewing the content of a version or comparing the content of two versions.

The Vault Browser history graph is created when the **Vault Browser** command is run. The history is not automatically updated with any changes made while you're viewing the history.

#### Notes:

For module objects, you can examine the entire module as a single object or examine an individual module member. If you select a workspace module base directory that contains multiple modules, you are prompted to select the module context.

For non-module vault objects, each object must be separately examined in the vault browser. You cannot examine a group of vault objects, such as a legacy configuration, or a set of tagged objects, in a single operation.

When working with modules, the vault browser includes both module merge edges and module member merge edges. You can specify a display color for these objects in the Vault Browser options page in SyncAdmin.

The Vault Browser Window opens in a new text tab labeled **Vault Browser** in the View Pane.

### Objects Viewable in the Vault Browser

The Vault Browser can be invoked after selecting any of the following objects:

- Workspace file under revision control (either file-based or a module member)
- Workspace module instance
- Workspace module member
- Server-side file vault
- Server-side file branch
- Server-side file version
- Server-side module vault
- Server-side module branch
- Server-side module version
- Server-side module member

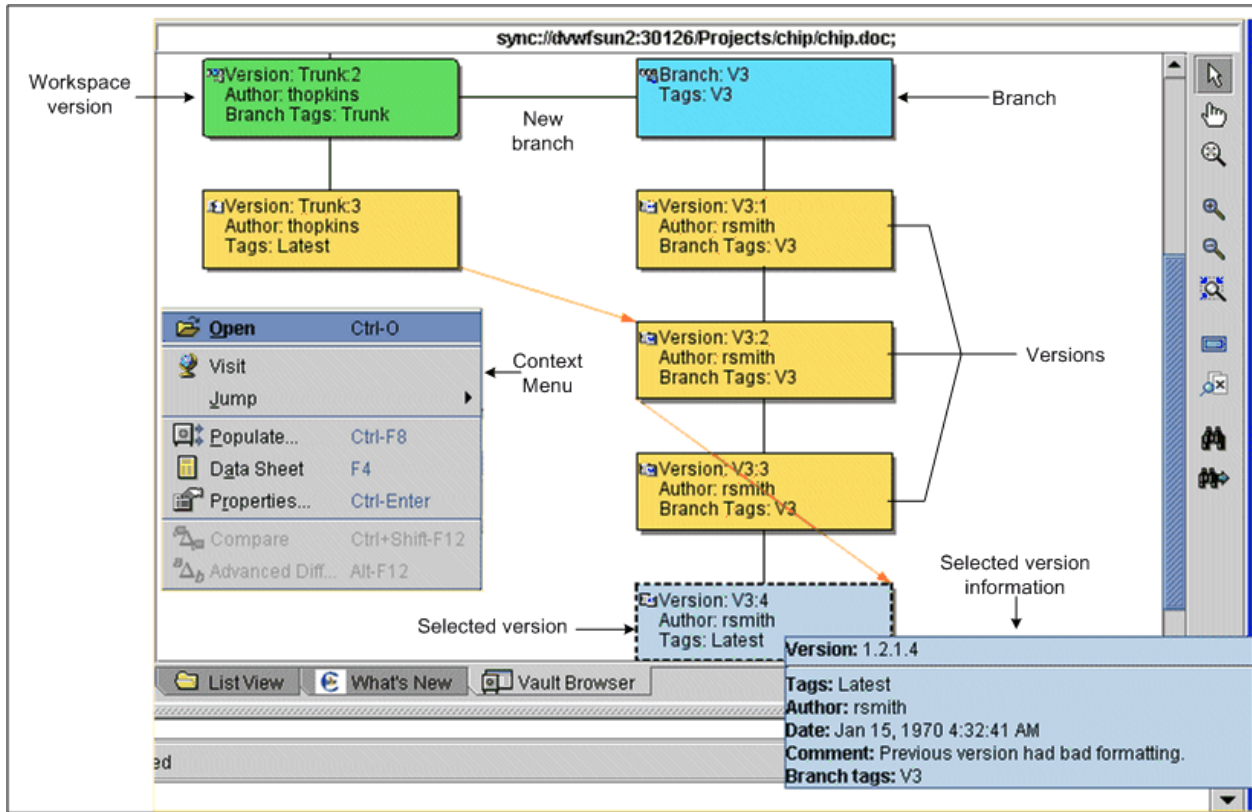
**Note:** For module members, only module versions that contain a change (content or structural) to the specified member are shown in **Show All Nodes** graph. To view other objects, including module instances, show all versions and branches.

### The Vault Browser window

The Vault Browser window shows the history of the selected object by showing each version sequentially. It opens with the Vault Browser window centered on the initial object version, the version from which the browser is invoked. Use the horizontal and vertical scroll-bars to move around in the Vault Browser window. To change the zoom level, use the Vault Browser Tools.

**Click on the fields in the following illustration for information.**





### Object SyncURL

The object SyncURL for the object always appears in the top bar of the Vault Browser window. This URL refers to the vault object itself, not to a specific object version. If the SyncURL is longer than the size of the Vault Browser window, the top bar becomes scrollable.

For file-based vault objects, the URL is the sync URL of the object, without the version number, for example: `sync://data2.ABCo.com:2647/Projects/Chip/chip.c`

For modules objects, the URL is the server-side sync URL of the module, for example:

`sync://data2.ABCo.com:2647/Modules/Chip`

For module member vault, the URL is the sever-side sync URL of the module and the natural path of the member, for example:

`sync://data2.ABCo.com:2647/Modules/Chip;chip.c`

### Branch object

The initial version on each branch, or the branch point version, is represented by a blue rectangle that includes the following information:

## DesignSync Data Manager User's Guide

- Branch name
- Branch tags

If you mouse-over the branch object, you can always see the following additional information:

- Branch numeric

### **Version object**

Each version is represented by a yellow rectangle that includes the following information:

- Branch name and version number
- Version author
- Branch tags
- Comment
- Number of branches connected from a particular version (Branch-point version only)

**Note:** When you shrink the display, not all the fields may show.

If you mouse-over the version object, you can always see the following additional information:

- Version numeric
- Creation date and time

### **Initial object version**

The version of the object selected when you launch the vault browser is shown in a green rectangle. It displays all the information available for a Version object.

### **Selected object**

If you select an object, or multiple objects, in the object browser, the selected objects turn blue-grey. You can launch the context menu to perform DesignSync operations directly on objects selected in the vault browser.

### **Related Topics**

Vault Browser Actions

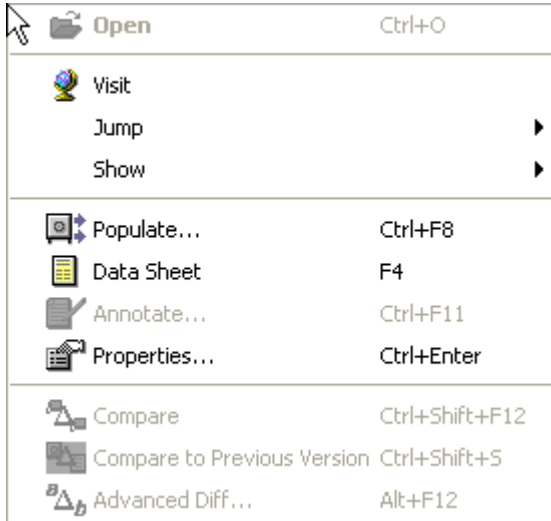
Vault Browser Tools

Finding Objects

## Vault Browser Actions

You can select one or more objects in the vault browser and launch the context menu to perform DesignSync operations directly on the selected objects.

Click on the fields in the following illustration for information.



### Vault browser object context menu

<b>Action/Option</b>	<b>Result</b>
<b>Open</b>	Opens the selected object in the default editor.
<b>Visit</b>	Switches to the List View and displays the object versions on the server. You may only select a single object for this option.

<p><b>Jump</b></p>	<p>Moves your focus to within the vault browser. You may only select a single object for this option.</p> <ul style="list-style-type: none"> <li>• <b>Forward</b> - moves your focus to the Latest child of the object selected. This may be the last version on a branch or the last branch emanating from a version.</li> <li>• <b>Backward</b> - moves your focus to the parent of the object selected. This is the initial object version on the selected branch. By repeatedly jumping backward the user will traverse the path between any object and the root of the history tree.</li> <li>• <b>Merged From</b> - moves your focus to the “merged from” version of the selected object. This option is only active when the object selected was created by an operation that created a merge edge; for example, skip, rollback, or merge.</li> <li>• <b>Merged To</b> - moves your focus to the "merged to" version of the selected object. This option is only active when another object was created from this object by an operation that created a merge edge; for example skip, rollback, or merge.</li> <li>• <b>Ancestor</b> - moves your focus to the “ancestor” version of a 3-way merge. This option is only active when the object selected was created by an operation that created a merge edge; for example, skip, rollback, or merge.</li> <li>• <b>Member Ancestor</b> moves your focus to the module version containing the parent of the selected member version. (Module Member version only)</li> <li>• <b>Member Descendents</b> moves your focus to the module version containing descendents of the selected member version. If there is more than one descendent for a member, you can select which descendent to view from the Select a Member Descendant dialog. (Module Member version only)</li> </ul> <p><b>Note:</b> When you jump, the object you jump to becomes the new selected object.</p>
<p><b>Show</b></p>	<p>Click to add or remove highlight features for the specified show option.</p> <ul style="list-style-type: none"> <li>• <b>Member Where Used</b> - highlights all the module versions in which member version is used.</li> <li>• <b>Member Genealogy</b> - adds arrows for ancestors and descendants of the member allowing you to track the genealogy of the module member.</li> <li>• <b>Clear</b> - all highlighting and arrows</li> </ul>

<b>Populate</b>	Launches the populate dialog. You may only select a single object for this option.
<b>Data Sheet</b>	Opens the data sheet for the object. You may only select a single object for this option.
<b>Annotate</b>	Launches the annotate tool on the object. You may only select a single object for this option.
<b>Properties</b>	Opens the properties pages for the object. You may only select a single object for this option.
<b>Compare</b>	Compares two selected module versions. You must select two files for this option.
<b>Compare to Previous Version</b>	Compares the content (diff) of the selected version with the previous file version in the vault. You must select a single file or module member for this command.
<b>Advanced Diff</b>	Launches the Advanced Diff dialog box. You must select two files for this option.

**Related Topics**

Vault Browser Overview

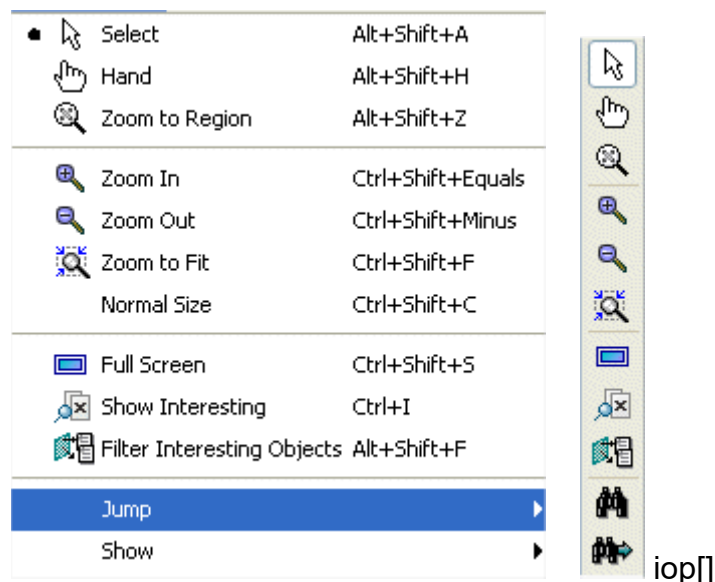
Vault Browser Tools

Finding Objects in the Vault Browser

**Vault Browser Tools**

The vault browser provides a set of navigation tools to help you navigate the information provided in the vault browser. The tools are available both from a Vault Browser menu, after the Vault Browser has been invoked, and a set of controls on the right side of the Vault Browser window.

**Click on the fields in the following illustration for information.**



<b>Action/Option</b>	<b>Result</b>
<b>Select</b>	<p>Click to select objects, such as branches and versions, in the vault browser window. To select an item, click on the item, or press and hold the left mouse button and draw a rectangle that includes all the objects you want to select.</p> <p><b>Tip:</b> You can also select multiple objects by pressing the CTRL key while clicking the left mouse button over the desired objects.</p>
<b>Hand</b>	Click to navigate within the vault browser window by clicking within the window and dragging a cursor to the desired section of the vault tree.
<b>Zoom to Region</b>	Click to select a region to zoom in on, by drawing a rectangle that becomes the focus point of the view.
<b>Zoom In</b>	Click to zoom in on vault browser window. The center point remains the same.
<b>Zoom Out</b>	Click to zoom out on vault browser window. The center point remains the same.
<b>Zoom to Fit</b>	Click to adjust the graph to fit in the window.
<b>Normal Size</b>	Click to adjust the graph to the original viewing size.

<b>Full Screen</b>	Click to open the history in a separate full-sized window. To close the window, click the Full Screen button again or press ESC. The same options are available in full-screen mode as in-line mode.
<b>Show Interesting</b>	<p>Click to toggle the graph visibility mode. In the “interesting” mode only the following objects are shown:</p> <ul style="list-style-type: none"> <li>• Tagged versions.</li> <li>• Tagged branches.</li> <li>• Branches, which include at least one version.</li> <li>• Branch point versions.</li> <li>• Endpoint versions of merge edges.</li> <li>• The initial object version selected when you invoked the vault browser.</li> <li>• Versions created by other users.</li> <li>• Snapshot branches.</li> </ul> <p>The objects displayed in the Show Interesting Node mode can be configured by using “Filter Interesting Objects.”</p>
<b>Filter Interesting Objects</b>	Click to select which interesting items you want displayed in the vault browser. The options vary depending on the type of object selected. For more information, see Filter Interesting Objects.
<b>Find</b>	Invokes the Find dialog box.
<b>Find Next</b>	Click next to advance to the next version that matches the last criteria in the Find dialog box. It searches down the version history.
<b>Jump</b>	Moves your focus to within the vault browser. For information on the specific <b>Jump</b> options, see Vault Browser Actions.
<b>Show</b>	Click to add or remove highlight features for the specified show option. For information on the specific <b>Show</b> options, see Vault Browser Actions.

**Related Topics**

Vault Browser Overview

Vault Browser Actions

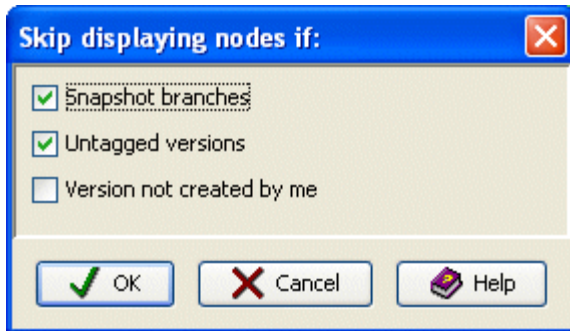
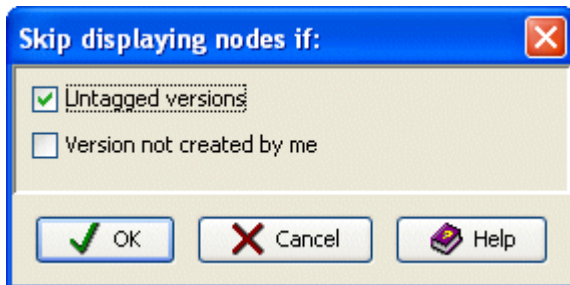
## Filter Interesting Dialog

Using the Filter interesting objects option allows you to control which objects are displayed when using the Vault Browser with the Show Interesting mode selected. Checking any option removes a version from the Vault Browser display if they do not meet any other inclusion criteria.

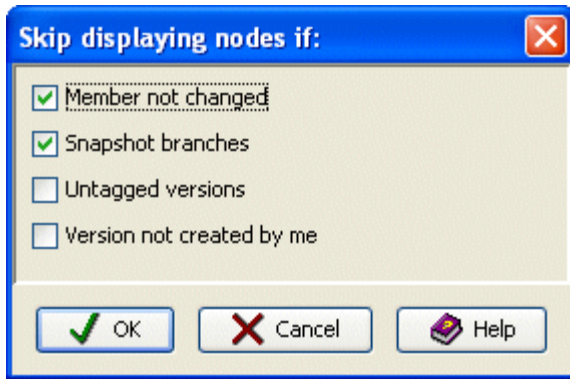
**Note:** Versions or branches excluded by selecting filter conditions are shown if there is a visible sub-graph rooted at that version or branch.

The filter interesting dialog changes depending on what type of object you have selected in the vault browser. The following illustrations shows file-based vaults, modules vaults, and module member vaults.

**Click on the fields in the following illustrations for information.**







### Untagged Versions

Check to remove any versions that are not explicitly tagged from the vault browser display.

### Version not created by me

Check to remove any versions that were not created by the user who is running the vault browser.

### Snapshot branches

Check to remove any versions that are on a snapshot branch. This is applicable only to module and module member versions.

### Member not changed

Check to remove any versions that are unmodified. This is only applicable to module member versions.

### Related Topics

Vault Browser Tools

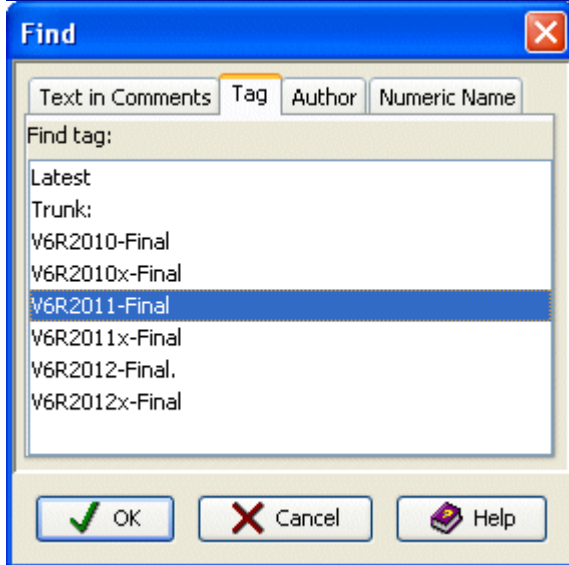
## Finding Objects in the Vault Browser

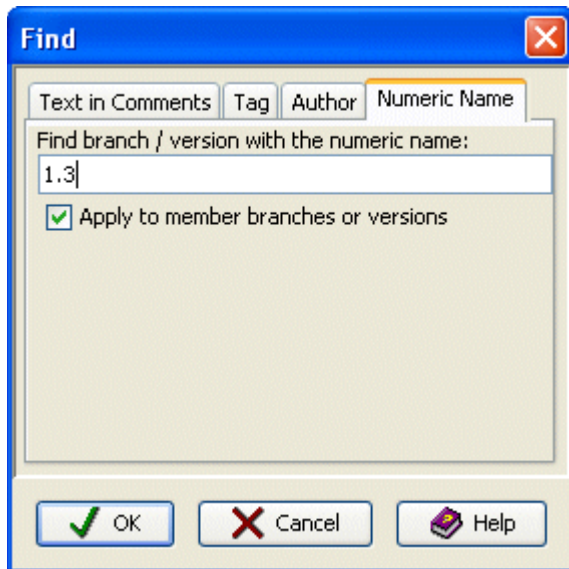
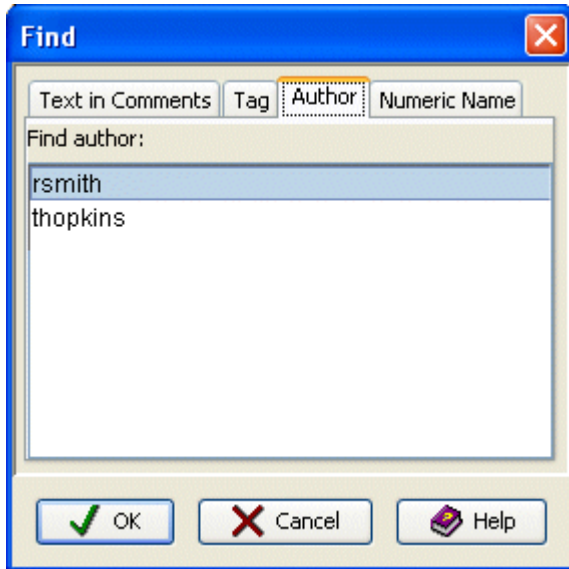
The vault browser allows you to search the object versions by comments, tag selector, author, or branch/version number.

### Notes:

- If Show Interesting is selected, the find is filtered to search only those objects.
- Find always finds the first appearance of the element being searched for. To find subsequent versions matching the selected search, use the Find Next button.

Click on the fields in the following illustration for information.





### Find what text

Enter the text to search for. DesignSync searches the comments field for the versions to find the matching string. The text in the comment must be an exact match for the text typed into the Find dialog.

By default, **Find what text** is no case-sensitive.

### Match case

Select this check box to enable case-sensitivity for the text search.

### **Find tag**

Select the tag to search for from the list. The tag list contains all the branch and version tags that have historically applied to the selected object.

### **Find author**

Select the username of the author to search for from the list. The list contains all the authors that have historically created new versions of the selected object.

### **Find Numeric Name**

Enter the numeric version identifier of the branch or version. This is a quick way to locate a desired numeric version in a vault browser report containing a lot of versions.

**Note:** This must be an exact match. It does not select partial matches or allow wildcards, for example, 1.2.11 (branch selector) will not match the versions on that branch (1.2.11.1, 1.2.11.2, etc.)

### **Apply to member branches or versions**

This is active when browsing a module member vault. It allows the user to choose which numeric names, modules or members are used for searching.

### **Related Topics**

[Vault Browser Overview](#)

[Vault Browser Tools](#)



# Working in SITaR

## Using SITaR as a SITaR Designer

### The Designer Role

Designers work on the design files within the project sub-modules. Design workspaces start with a baseline release of all of the sub-modules.

- Designers use the `sitr update` command to make individual sub-modules editable within their design workspace, leaving the other sub-modules at the baseline.
- Designers working on a specific sub-module coordinate their work to create a version of the sub-module that is ready to for submittal as part of the new baseline.
- Designers test their sub-module work in progress versus the existing baseline versions of the other sub-modules, or they can selectively fetch specific versions of other sub-modules for testing.

When the lead Designer responsible for a sub-module believes that it is ready, they submit it for integration.

**Note:** SITaR can be configured to preserve the exact module hierarchical structure, or context, that the submodule is submitted. For information on configuring SITaR to preserve context, see [Defining and Enabling Module Context](#).

#### Related Topics

[Creating a Workspace](#)

[Changing a Sub-Module](#)

[Synchronizing a Module with the Baseline](#)

[Submitting a Sub-Module for Integration](#)

[Synchronizing all Sub\\_Modules with the Baseline](#)

[ENOVIA Synchronicity Command Reference: `sitr update`](#)

[ENOVIA Synchronicity Command Reference: `sitr submit`](#)

### Creating a Workspace

To use a design in the SITaR workflow, a designer must use the `sitr populate` command to fetch the design into his workspace. The design is represented as a module

hierarchy. The top-level or container module name and the path into which it will be fetched are defined by the SITaR environment variables.

**Note:** You must have already created the directory to store the designer workspace and the name must match the name specified in the SITaR environment variables.

```
stcl> sitr populate
```

The sitr populate commands fetches the baseline version of the design.

**Note:** Before beginning to use SITaR, a designer must define a set of SITaR environment variables that control the behavior of the SITaR commands. For more information, see SITaR Environment Variables.

#### Related Topics

ENOVIA Synchronicity Command Reference: sitr populate

## Editing a Sub-Module

Once the design has been fetched a designer may begin development on one or more of the sub-modules contained in the design. When developing a sub-module, you must change it to a “development” state using the sitr update command.

```
stcl> sitr update <ModuleName>
```

The sitr update command above will fetch the latest version of the specified sub-module from the appropriate branch into the workspace, replacing the sub-module version previously fetched as part of the baseline. The contents of the fetched sub-module may now be modified as desired. When modifications are complete, use the DesignSync ci command to check them in and create a new version of the sub-module.

#### Related Topics

ENOVIA Synchronicity Command Reference: sitr update

ENOVIA Synchronicity Command Reference: populate

## Synchronizing a Module with the Baseline

While development work is done on one sub-module, releases of other sub-modules may be submitted and successfully integrated into new baseline releases. The sitr populate command can be used to pick up any new baseline releases of sub-modules that are not in the “development” state of a designer’s workspace, without affecting any development work.

```
stcl> sitr populate
```

### Related Topics

ENOVIA Synchronicity Command Reference: sitr populate

## Submitting a Sub-Module for Integration

When all changes to a sub-module have been checked in and the new sub-module version is ready to share with the design team, the sitr submit command is used to tag the sub-module version and inform the SITaR integrators that it is ready for integration.

**Note:** If development work on a sub-module is being done in more than one workspace, you must check-in all of the local changes, and populate all of the checked-in changes from other workspaces prior to running sitr submit.

```
stcl> sitr submit <WorkspaceDir>
```

The command above will apply DesignSync tag to the latest version of the Trunk branch of the specified sub-module. SITaR has a default naming convention for sub-module tags. The convention takes the form “vX.Y” where X is a major release number and Y is a minor release number. Each time a sub-module is submitted for integration, SITaR automatically increments the minor release number and applies the tag to the latest version on the Trunk branch of the sub-module.

**IMPORTANT:** When the developer submits a module, he can capture all developer workspace information, such as any replaced submodule versions, and save it in a "context" module which can be used by the integrator to reproduce the submodule and determine if there are there are incompatibilities introduced by other modules. If saving the context module is enabled, the developer should simply submit the module as usual. For more information on setting up the context module, see *Defining and Enabling Module Context*. For information about how the integrator populates a module with the module context, see *Recreating the Developer's Workspace*.

For more information on SITaR version numbers, see the *ENOVIA Synchronicity Command Reference: sitr submit help*.

### Related Topics

ENOVIA Synchronicity Command Reference: sitr submit

## Synchronizing all Sub-Modules with the Baseline

Once the designer has submitted the submodule, the integrator is responsible for integrating the new version of the sub-module into the design, testing and releasing a new version of the baseline, if needed.



After a baseline containing the submitted version of the sub-module has been released, the sub-module designers have two options:

1. Continue development of the Trunk: files in the existing sub-module workspaces, and update all of the other sub-modules to the new baseline using the `sitr populate` command. This allows designers to fetch baseline changes to sub-modules that are not currently under development in their workspace without affecting the sub-modules that are under development. For more information, see [Synchronizing a Module with the Baseline](#).
2. Update all of the sub-modules in their workspaces to match the new baseline.

The `-force` option to the `sitr populate` command tells it to “force” the current baseline into a workspace, including replacing all sub-modules that are under development work with their current baseline versions.

```
stcl> sitr populate -force
```

If, after forcing the current baseline into a workspace, a designer wants to continue development work on specific sub-modules, the sub-modules should be placed in the “development” state using the `sitr update` command as described in [Changing a Sub-Module](#).

#### Related Topics

[Changing a Sub-Module](#)

[Synchronizing a Module with the Baseline](#)

[ENOVIA Synchronicity Command Reference: sitr update](#)

[ENOVIA Synchronicity Command Reference: sitr populate](#)

## Branching a Sub-Module

The Designer role can use the `sitr mkbranch` command to create a sub-module branch from any sub-module version.

The `sitr mkbranch` command can perform the following actions:

- Create a branch from any sub-module version. You can specify the version by sync URL, workspace directory, or workspace module instance name.
- Update your local workspace with the new branch version.
- Change the local workspace selector to point to the new branch.

The format for the `sitr mkbranch` command is:

```
stcl> sitr mkbranch [-populate] <BranchName> <Module>
```

### Related Topics

Branching in SITaR

Parallel (Multi-Branch) Development

ENOVIA Synchronicity Command Reference: sitr mkbranch command

## Using SITaR as a SITaR Integrator

### The Integrator Role

Integrators maintain the container module. The integration workspaces contain the latest version of the container module on the specified branch (Trunk: by default).

- Integrators use the sitr lookup command to find module versions that are candidates for integration into the next baseline, including versions of sub-modules submitted for consideration by the designers.
- Integrators use the sitr select and sitr integrate commands to manipulate the sub-module hrefs associated with the Trunk:Latest version of the container module, and to create a container module version that is a candidate for the next baseline.
- Integrators run tests within the integration workspace to insure that the versions of the sub-modules in the candidate version of the container module function properly together.
- If the sub-module versions do not function properly, the Integrators report their findings to the designers. The designers can modify the sub-modules causing the problem and submit new versions, or the integrators can choose different versions of the sub-modules and continue testing.
- Integrators can recreate a developer workspace, which may be using a different set of submodules than the official integration, to assist in testing.

When the integrators are satisfied with the test results, they release a new baseline version of the container module.

### The Integration Workspace

An Integration workspace is used to integrate, test, and release new baselines of the container module. Integration workspaces are inherently different than design workspaces.

Within design workspaces, designers populate the baseline version of the project, and then perform the development work on the Trunk:Latest versions of individual sub-modules.

Within the integration workspaces, integrators manipulate the Trunk:Latest version of the container module, but do not modify the contents of any individual sub-modules. Any sub-module changes must be made in the design workspaces and submitted to the integrator.

## Creating an Integration Workspace

To use a design using the SITaR workflow, integrators use the `sitr populate` command to fetch the Trunk:Latest version of the container module into their workspaces.

```
stcl> sitr populate
```

### Notes:

- Before beginning to use SITaR, a designer must define a set of SITaR environment variables that control the behavior of the SITaR commands. For more information, see [SITaR Environment Variables](#).
- When the designer captures the module context when submitting a module, the integrator can populate a workspace with the full context information using the standard DesignSync `populate` command. For more information, see [Recreating the Developer's Workspace](#).

### Related Topics

[ENOVIA Synchronicity Command Reference: sitr populate](#)

## Workflow for Updating the Container Module

The typical flow for an SITaR integrator is:

1. Locate modules versions to be considered for integration into the container module.
2. Select module versions to add or remove from the container module.

**Note:** You select the changes individually, but you can integrate multiple selections with a single integration command.

3. Integrate the selected changes into the Trunk:Latest version of the container module.
4. Test the Trunk:Latest version of the container.
5. If the tests fail, repeat the select/integrate/test process.
6. If the tests pass, Release a new baseline version of the container module.

## Related Topics

Locating Submitted Modules for Integration

Selecting Sub-Modules for Integration

Integrating Selected Changes into the Container Module

Testing the Integration Version of the Container Module

Releasing a New Baseline

## Locating Submitted Modules for Integration

The `sitr lookup` command displays versions of the existing container sub-modules that have been created submitted (or tagged) since the time that the most recent baseline release was created.

```
stcl> sitr lookup
```

The `sitr lookup` command can also be used to look up the tagged module history of a specified module, or to look up all tagged module version since a specified date and time. For more information, see the *ENOVIA Synchronicity Command Reference: sitr lookup* command.

Note: The lookup command searches across all modules residing on servers defined in the `sync_servers.txt` file.

## Related Topics

*DesignSync Data Manager Administrator's Guide: SyncServer List Files*

*ENOVIA Synchronicity Command Reference: sitr lookup*

## Selecting Sub-Modules for Integration

The `sitr select` command targets tagged module versions for addition to or deletion from the container module. The module is added to the default relative path location specified by the `sitr_relpath` environment variable, unless the `-relpath` option is used to select a different relpath. If a sub-module already exists in the relpath selected, the old sub-module is targeted for deletion during the next integration.

```
stcl> sitr select <module>@<version>
```

```
stcl> sitr select -delete <module>@<version>
```

**Note:** When using overlapping modules, if you try to add a module to a relpath that already contains a module, you must specify the module using the `-name` option. For more information on specifying options to `sitr select`, see the *ENOVIA Synchronicity Command Reference: sitr select*.

For a list of the current selected changes, specifying 'sitr select' with no arguments:

```
stcl> sitr select
```

**Note:** at any time you can clear all of the selected changes by either exiting DesignSync, or by using the `-clear` option to `sitr select`:

```
stcl> sitr select -clear
```

**Tip:** The `sitr update` command assumes all editing is done on the Trunk branch, however SITaR allows integrators to select any tagged version of a sub-module, including versions that are on a non-Trunk: branch. To provide a stable workflow, you should avoid integrating non-Trunk: versions of a sub-module into the container. If you decide to integrate a non-Trunk version however, you should only do it if there are no plans to change the contents of the associated sub-modules for the project.

#### Related Topics

ENOVIA Synchronicity Command Reference: `sitr select`

## Integrating Selected Changes into the Container Module

After the changes for integration have been selected, you integrate them into the Trunk:Latest version of the container by using the `sitr integrate` command. Then you populate the new contents of the container into the integration workspace.

```
stcl> sitr integrate
```

#### Related Topics

ENOVIA Synchronicity Command Reference: `sitr integrate`

## Testing the Integration Version of the Container Module

The `sitr integrate` command updates the Trunk:Latest version of the container, but does not release a new baseline. The SITaR workflow assumes that after the `sitr integrate` command is used to create the new Trunk:Latest version of the container into an integration workspace, the Integrator will run tests on the contents of the workspace to ensure that everything functions properly together.

If there are problems with the sub-modules being integrated, the integrators can either contact the appropriate designers to have them update and submit new versions of the problem sub-modules, or re-execute the `sitr select/sitr integrate` process to try different combinations of different versions of the sub-modules to attempt to create a version of the container module that passes the tests.

### Locating a Context Module Version

SITaR stores the module context information in a separate module, called a context module, which is associated on a per-client basis with SITaR environment variables. Each member version in the context module is associated with a specific SITaR submittal.

In order for the integrator to know which SITaR submittal to populate should they need to test against a copy of the developer's workspace, SITaR includes a lookup command. The SITaR context command allows the integrator to either list all the module members along with the submittal version information or the context module version associated with a specified submittal.

```
stcl> sitr context -allconfigs | -release <release>  
<SITR_Submodule_Name>
```

For more information, see the *ENOVIA Synchronicity Command Reference: sitr context* command.

### Recreating the Developer's Workspace

The integrator may want to recreate a developer workspace to view the differences between the environment the developer worked in; for example, if the developer is working with an older or a pre-release copy of the module that has not been integrated into the baseline being tested by the integrator.

#### Recreating the Developer's Workspace:

1. Determine which submittal version of the submodule is of interest using the `sitr context` command:

```
'sitr context -release <releaseName> <SubModuleName>'
```

The command returns a tcl list. Within the TCL list is a `sitr_url` property containing the SYNC URL for context module version. the `sitr_modified_modules` value contains a list of the next level dynamic selectors containing local modifications.

2. Create a new workspace populated with the URL obtained from the `sitr_context` command. Use the standard DesignSync populate command, not the `sitr populate` command.

```
'populate -recursive -hrefmode static<sitr_ul>
```

This populates the workspace in static mode to fetch the versions of the modules that were present in the developer's workspace at the time.

**Note:** You should not perform attempt to modify or checkin this workspace. In order to make and check in modifications, either use the usual integration workspace, or repopulate this workspace in `-href` dynamic mode.

## Releasing a New Baseline

Once the new Trunk:Latest version of the container module passes all necessary tests, the Integrator can establish a new baseline using the `sitr release` command. Creating a new release of the baseline increments the version tag associated with the current baseline release and applies the new tag to the Trunk:Latest version of the container module. It also updates the baseline tag defined by the SITaR environment variables to the appropriate version of the container module so developers can populate their workspaces with the latest baseline.

```
stlc> sitr release
```

Note: For more information on SITaR version tags, see the *ENOVIA Synchronicity Command Reference*: `sitr submit` command help.

### Related Topics

Synchronizing all Sub-Modules with the Baseline

ENOVIA Synchronicity Command Reference: `sitr release`

## Branching a Container or Sub-Module

The Integrator can use the `sitr mkbranch` command to branch either a container module or sub-module.

The `sitr mkbranch` command can perform the following actions:

- Create a branch from the specified container or sub-module version. You can specify the version by sync URL, workspace directory, or workspace module instance name.
- Update your local workspace with the new branch version.
- Change the local workspace selector to point to the new branch.
- Integrate the newly created sub-module into the container module.

The format for the `sitr mkbranch` command is:

```
stcl> sitr mkbranch [-populate] [-integrate] <BranchName>  
<Module>
```

### Related Topics

Branching in SITaR

Parallel (Multi-Branch) Development

ENOVIA Synchronicity Command Reference: `sitr mkbranch` command

## Configuring SITaR

### SITaR Environment Variables

SITaR uses environment variables to simplify the SITaR command options by defining default behaviors and roles.

Most of the SITaR commands cannot run if the SITaR environment variables are not defined.

The SITaR environment variables are:

#### **sitr\_alias**

Defines the name of the tag used by SITaR to define the latest qualified 'baseline' release of the container module, for example:

```
export sitr_alias=baseline
```

#### **sitr\_automcache**

Determines whether automatic mcaching is on or off. By default automcaching is on. To disable automcache, set the value to 0. If the scripted mirror functionality is used to maintain the mcache, turning off `sitr_automcache` provides a performance enhancement to `sitr populate`.

An example of disabling the environment variable is:



```
export sitr_automcache=0
```

### **sitr\_branch**

Defines the default branch of the container module. This default is used by the sitr integrator for the sitr integrate and sitr release commands. If the variable is not defined, "Trunk" is used as the default branch.

An example of setting the environment variable is:

```
export sitr_branch=rel2.6
```

### **sitr\_container**

Defines the name of the container module for the project, for example:

```
export sitr_container=projxContainer
```

Note: This documentation uses 'projxContainer' as the name of the container project to be unambiguous about which module is the container module. In practice, the container module can have any desired name.

### **sitr\_context\_required**

Indicates whether the module context information is captured during a submit type action.

There are two possible values:

- 0 (zero) - Indicates that the module context information is not gathering during submit and release operations. This means that you may not be able to recreate the workspace conditions exactly, for example, if, during test failure, you want to recreate the test in the workspace to see why it worked before integration.
- 1 (one) - Indicates that the module context information is preserved when the workspace is submitted. This means that should there be a need, for example, during a test failure, the integrator could exactly reproduce the development workspace that submitted the change. (Default)

### **sitr\_context\_module**

If the sitr\_context\_required variable is enabled (sitr\_context\_required=1), this variable must be set to a valid module to store the context for submit and release actions within sitr. Note: Do not specify a selector for the module. To specify a branch selector, use the sitr\_context\_branch variable.

### **sitr\_context\_branch**

If the sitr\_context\_required variable is enabled, this variable can be set to a branch selector. The branch selector, in conjunction with the sitr\_context\_module defines the module and branch information for the module used to store submittal context information. If no branch is specified, sitr uses the default Trunk: branch.

### **sitr\_integrator\_update**

Indicates whether the integrator is allowed to run the "sitr update" command. Ordinarily the user must have a Design role, to be allowed to run the "sitr update" command.

There are two possible values:

- 0 (zero) - Indicates that the integrator is not allowed to run the "sitr update" command. (Default)
- 1 (one) - Indicates that the integrator is allowed to run the "sitr update" command.

### **sitr\_min\_comment**

Indicates whether a minimum comment is required by any sitr command that has a -comment option.

- 0 (zero) or no variable - indicates that no minimum comment is required.
- 1 (one) or greater - indicates the minimum comment length required

### **sitr\_relpath**

Defines the default relationship between the container module workspace directory (defined by sitr\_workdir), and the base directory of each sub module.

There are two possible values for sitr\_relpath:

- Cone - Indicates that the default directory structure is hierarchical. Each of the sub modules is placed in a sub directory of the sitr\_workdir, for example:

If you have sub modules named "ALU" and "memory" their default relative paths are: ./ALU and ./memory, and, given sitr\_workdir=~ /Workspaces/ projxContainer, their base directories are:

~/Workspaces/ projxContainer/ ALU

~/Workspaces/ projxContainer/memory

- Peer - Indicates that the default directory structure is flat. Each of the sub modules is placed in directories parallel to sitr\_workdir, For example:

If you have sub-modules "ALU" and "memory", their relative paths are ./ALU and ./memory, and, given sitr\_workdir=~ /Workspaces/ projxContainer, their base directories are:

~/Workspaces/ ALU

~/Workspaces/memory

An example of setting the environment variable is:

```
export sitr_relpath=Cone
```

### **sitr\_role**

Defines the SITaR role for the user. There are 2 possible values:

- Design - Users modifying the individual sub modules
- Integrate - Users defining and maintaining the container module

An example of setting the environment variable is:

```
export sitr_role=Design
```

### **sitr\_server**

Defines the URL for of the server hosting the container module, for example:

```
export sitr_server=sync://syncServer1:2647
```

**Note:** The 'sitr\_server' is the first server searched by the 'sitr lookup' and 'sitr select' commands, and the default server used by the 'sitr mkmod' command.

### **sitr\_workdir**

Defines the workspace directory of the container module, for example:

```
export sitr_workdir=~/Workspaces/ projxContainer
```

**Note:** The examples above show setting the environments on UNIX in bourne shell (sh).

**Tip:** Create project specific setup files that define the SITaR environment variables.

Users can then "source" the appropriate project setup file prior to executing any SITaR commands. This reduces setup time, minimizes the possibility of user error, and guarantees that all users are using a controlled environment defined for them.

### **Related Topics**

Sample SITaR Environment Variable File

ENOVIA Synchronicity Command Reference: sitr env

## **Sample SITaR Environment Variable File**

## DesignSync Data Manager User's Guide

This example shows a SITaR environment variable file for the Designer team. This example assumes a UNIX csh environment.

```
setenv sitr_role Design

setenv sitr_container projxContainer

setenv sitr_alias baseline

setenv sitr_automcache 0

setenv sitr_server sync://syncServer1:2647

setenv sitr_workdir ~/Workspaces/projxContainer

setenv sitr_relpath Cone

setenv sitr_min_comment 10

setenv sitr_context_required 1

setenv sitr_context_module
sync://syncServer1:2647/Modules/Context/projxContext

setenv sitr_context_branch Trunk
```

### Related Topics

SITaR Environment Variables

## Creating a SITaR Container Module

The SITaR container modules "contain" the sub modules that make up the design.

Usually, the container module only has hrefs to sub modules. All design files should reside within the sub modules and not within the container module.

To create a container module you:

1. Set up the SITaR environment variables as described in the SITaR Environment Variables topics.
2. Execute the `sitr mkmod` command with the `-top` option

```
sitr mkmod -top
```

The SITaR environment variables are used to create the specified SITaR container module on the specified SITaR server.

After the SITaR container module has been created, the individual sub modules can be created and/or integrated in to the container module.

### Related Topics

Creating a SITaR Sub-Module

ENOVIA Synchronicity Command Reference: `sitr mkmod`

## Defining and Enabling Module Context

SITaR has the capability to recreate the developer workspace exactly, for example if a different submodule is used to replace the expected submodule, by saving the module context in a separate module at the same time the developer's module is submitted. Once the context module is defined, and context preservation is enabled, the developer automatically updates the context module with every submittal -- there is no additional action required by the developer.

Then, if needed, the integrators can populate their workspace as described [Recreating the Developer's Workspace](#).

**Tip:** For ease of deployment, this procedure is going to recommend creating a SITaR environment script that can be sourced by the `.login` or shell initialization script (`.profile`, `.cshrc` etc.). You are not required to set your variables this way, but the variables included in this procedure must be set in order to enable module context preservation. For best practices, a site should maintain two versions of the script, one for developers and one for integrators; setting all of the SITaR environment variables for the appropriate user environment.

### Defining Module Context:

1. Create or modify the SITaR environment script to include the context variables. To enable module context preservation, you must set:
  - `sitr_context_required` to 1 to enable module context preservation.
  - `sitr_context_module` to the server URL for the module that will hold the context information. This module is a module ONLY to store the module context information. It is not the top module in your SITaR configuration, or any submodule in the architecture..

You can optionally set:

`sitr_context_branch` to store the branch name. If no branch name is selected, the Trunk branch is the default.

**Note:** Do not specify the branch with a `:` identifier.

2. Save the SITaR environment script to a memorable name such as `.sitr_developer` or `.sitr.context`.
3. Source the environment script within the default `.login` or shell initialization script (`.cshrc`, `.profile`, etc.), so that it is available to your entire team:  

```
source ~/.sitr_developer
```
4. Distribute the script to your users.

## Creating a SITaR Sub-Module

All design work should reside inside the sub modules of a SITaR project. The SITaR sub-modules are created with the `sitr mkmod` command.

```
stcl> sitr mkmod -name <Module>
```

When the `sitr mkmod` command creates a sub module, it tags the Trunk:Latest version of the module as v1.1, and submits it for integration. If the module does not already exist, the v1.1 module version tagged is empty.

**Note:** You may need to populate your workspace to see the module.

## Examples of using the `sitr mkmod` command

The '`sitr mkmod`' command can be used to:

- Create a new/empty module on the default SITaR server.

```
sitr mkmod -name ALU
```

- Create a new/empty module on a server that is not the default SITaR server.

```
Exp: sitr mkmod -vaultpath  
sync://syncServer2:2647/Modules/ALU2
```

- Define an existing module as a SITaR module.

```
sitr mkmod -vaultpath  
sync://syncServer2:2647/Modules/OLD_ALU
```

## Related Topics

SITaR Environment Variables

## Creating a SITaR Container Module

ENOVIA Synchronicity Command Reference: `sitr mkmod`

ENOVIA Synchronicity Command Reference: `sitr env`

## Creating an Initial Baseline Release

After the container module and desired sub modules have been created, an integrator should create the initial baseline release of the project to release to the designers. This allows them to begin working in a SITaR workflow environment.

### To create the initial baseline release:

1. Define all the SITaR environment variables for the integrator as described in SITaR Environment Variables.
2. Use the 'sitr select' command to select all of the versions of the sub modules being added to the container.
3. Use the 'sitr integrate' command to integrate the selected sub modules into the Trunk: version of the container module, and populate the Trunk: version into their integration workspace.
4. Run any necessary tests on the Trunk: version of the container.
5. Use the 'sitr release' command to release the v1.1 version of the container.

### Example of creating a baseline configuration:

You are the integrator of the projx project and you want the initial baseline release of projx to have 3 sub modules:

projx - To contain the top level design files for projx

ALU - To contain the ALU design files

memory - To contain the memory design files.

You have used 'sitr mkmod' to create the initial release of both the projx and ALU sub modules, and you are using the v1.5 version of the memory module.

You have defined a cone structure for your SITaR environment. Your workspace directories should look like this:

```
~/Workspaces/projxContainer/projx
```

## DesignSync Data Manager User's Guide

```
~/Workspaces/projxContainer/ALU
```

```
~/Workspaces/projxContainer/memory
```

Within DesignSync, select the appropriate module versions for addition into the projxContainer module at the default relative path of each sub module:

```
stcl> sitr select projx@v1.1
```

```
stcl> sitr select ALU@v1.1
```

```
stcl> sitr select memory@v1.5
```

**Note:** If you are unsure what modules are available to be integrated, use the sitr lookup command.

Integrate the above selections into the Trunk: version of the projxContainer module, and populate this version into the integration workspace.

```
stcl> sitr integrate
```

Test the Trunk: version of the projxContainer module, if necessary. Create the v1.1 release of the projxContainer module, and tag it as the 'baseline' release.

```
stcl> sitr release
```

The baseline release is now ready to be used by the designers in a development environment.

### Related Topics

[SITaR Environment Variables](#)

[Creating a SITaR Container Module](#)

[Creating a SITaR Sub-Module](#)

[ENOVIA Synchronicity Command Reference: sitr lookup](#)

[ENOVIA Synchronicity Command Reference: sitr select](#)

[ENOVIA Synchronicity Command Reference: sitr integrate](#)

[ENOVIA Synchronicity Command Reference: sitr release](#)



## Branching a Container or Sub-Module

The Integrator can use the `sitr mkbranch` command to branch either a container module or sub-module.

The `sitr mkbranch` command can perform the following actions:

- Create a branch from the specified container or sub-module version. You can specify the version by sync URL, workspace directory, or workspace module instance name.
- Update your local workspace with the new branch version.
- Change the local workspace selector to point to the new branch.
- Integrate the newly created sub-module into the container module.

The format for the `sitr mkbranch` command is:

```
stcl> sitr mkbranch [-populate] [-integrate] <BranchName>
<Module>
```

### Related Topics

Branching in SITaR

Parallel (Multi-Branch) Development

ENOVIA Synchronicity Command Reference: `sitr mkbranch` command

## Reference

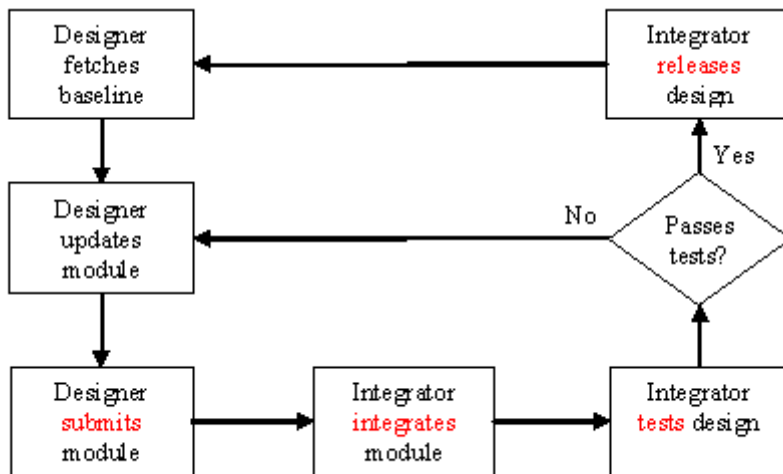
### Overview of SITaR Workflow

SITaR (Submit, Integrate, Test, and Release) is a simple workflow for projects that consist of multiple blocks, or modules, developed by several contributors. The workflow is based upon the concept of two distinct project roles:

- **Designers** contribute to the development of blocks or modules.
- **Integrators** verify the stability of the blocks or modules, and assemble them into functional packages from which designers continue to develop. This provides a qualified and stable baseline for development work.

By working from a qualified baseline, a designer is able to develop a module within a known, trusted environment. When module development has reached the point where a new version should be available to the entire design team, the Designer submits (the ‘S’ in SITaR) the new version of the module for integration.

At this point the Integrator may choose to integrate (the 'I' in SITaR) the new module version into the design. The integration process may include several new module versions submitted by any number of contributors. The integrator tests (the 'T' in SITaR) the new module versions within the overall design to ensure that everything functions together. When satisfied with the test results, the Integrator creates a new release (the 'R' in SITaR) of the design. This release then becomes the new baseline from which module development continues.



### Designer and Integration Workspaces

Integration workspaces and Design workspaces should be kept separate. Design work should never be done inside an Integration workspace. Integrators who sometimes do Design work should maintain separate Integration and Design workspaces.

Because Integration workspaces are used to do testing prior to release of a new baseline, it is imperative that contents of the workspace exactly matches what will be released as the baseline. If Design work is being done within an Integration workspace, it is possible for the tests to pass, due to the presence of the Trunk: version of a sub-module in the workspace that is not currently part of the Trunk: version of the container module, and therefore will not be part of the baseline to be created.

#### Related Topics

SITaR Module Structure

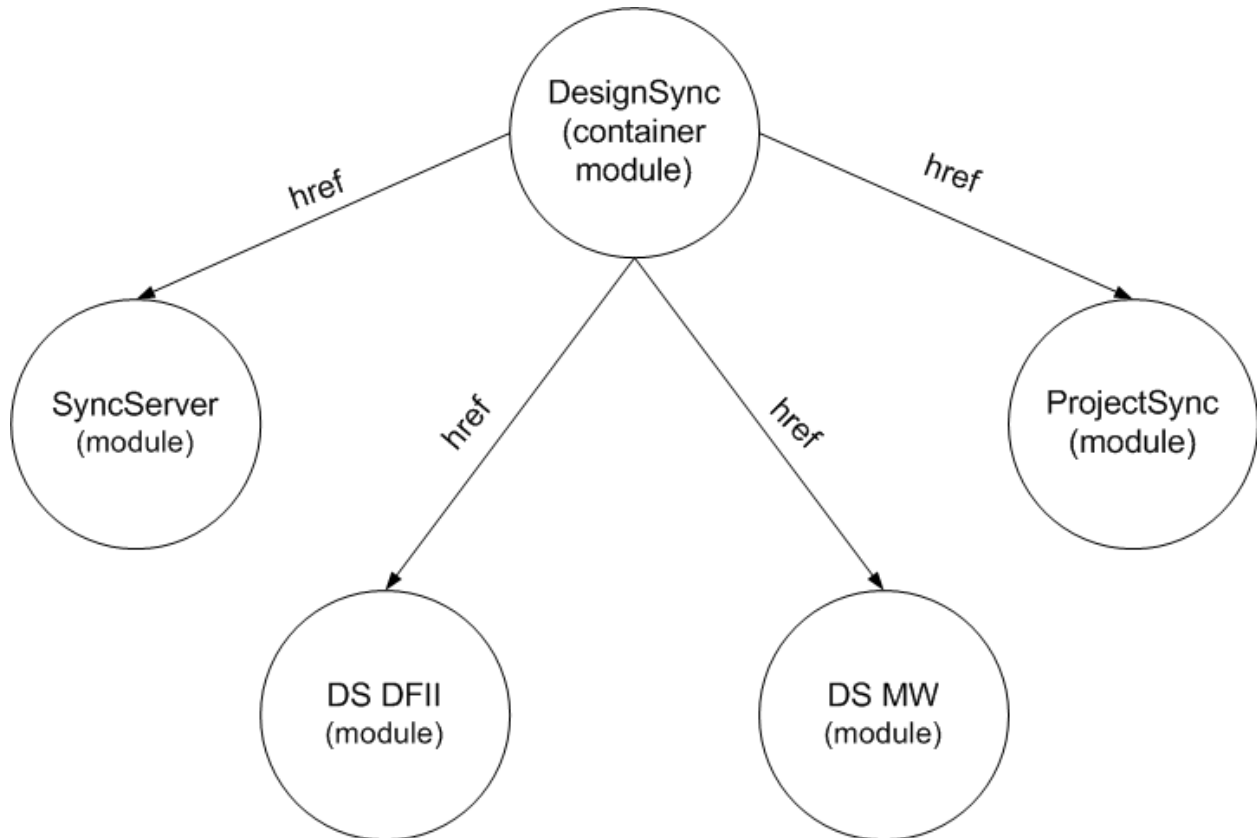
The Designer Role

The Integrator Role

### SITaR Module Structure

SITaR provides a simple use model, with a two levels of hierarchy. A project is partitioned into a “Container Module” and a number of connected sub-modules. All of the design work resides in the sub-modules. Hierarchical references (hrefs) within the container module bind the sub-modules together into a functioning system.

Using DesignSync as an example, DesignSync itself is composed of components (sub-modules) that together become the DesignSync product. The DesignSync lead creates DesignSync as a container module, consisting only of hierarchical references to submodules for each of the products.



**Note:** SiTAR does support overlapping modules within a single module base directory, although this example does not show that configuration.

### Designing and Implementing Your Module Structure

When beginning a SITaR project, you must design your module structure. This includes determining how to partition the design into a number of sub-modules. Usually the project is broken along functional lines. For design data management, however, it is useful to also consider the number of people that will be working on each of the sub-modules. If the team working on an individual sub-module is too large, or does not

communicate well, efficient management of the changes to the design can become a burden.

Ideally, there will be a small number of designers assigned to any given sub-module. Keeping the design teams small helps facilitate communication, and minimizes design conflicts that can arise when multiple people are modifying the same design concurrently.

SITaR helps facilitate concurrent design of the top-level project, but the sub-module design teams must handle the concurrent design process within any given sub-module.

### **Related Topics**

Overview of SITaR Workflow

## **Branching in SITaR**

SITaR (Submit, Integrate, Test, and Release) supports parallel (Multi-branch) development for the container module and sub-modules in the SITaR methodology.

The Designers can branch any sub-module.

The Integrator can branch the container module, a sub-module, or link a sub-module on any branch to the container module.

### **Related Topics**

Parallel (Multi-Branch) Development

Branching a Sub-Module (Develop role)

Branching a Container or Sub-Module (Integrate role)

ENOVIA Synchronicity Command Reference: `sitr mkbranch` command



# Techniques

## Getting Started with the GUI

### Using the DesignSync GUI

This document provides instructions to help you create a work area and quickly get started using DesignSync to manage your design data.

---

#### Assumed Environment:

(UNIX platform, locking work style)

The instructions in this document assume that:

- You are working in a UNIX environment.
- The DesignSync tools have been installed.
- To use DesignSync, you invoke the DesignSync GUI (**DesSync**) and use it to perform revision control operations.
- In your environment:
  - You have set the **SYNC\_DIR** variable. (This environment variable defines the path to your DesignSync installation directory; you need to set this variable before running DesignSync.)
  - The **PATH** variable includes **\$SYNC\_DIR/bin** so that the operating system can find the DesSync executable.

This document uses a scenario to illustrate the steps you use to get started. In the scenario:

- You are a designer working on a project called ASIC.
- On a DesignSync server (also called a SyncServer) your project leader or DesignSync tools administrator has created a SyncServer folder where you can put your design data for your ASIC project. The SyncServer has the <host name>:<port> of `ca-srvr.Acompany.com:2647` (a remote vault). The URL for the Projects folder is: `sync://ca-srvr.Acompany.com:2647/Projects`.
- Your project team follows the **locking work style** where you always check out with a lock an object that you plan to edit. Other team members can fetch (check out without a lock) the object, but no one else should make changes while you hold the lock.
- Your project leader has set up a project directory with subdirectories for the project team members. The project directory has the UNIX hierarchy:

```
/data/devel/ASIC/users
user1 user2 user3 . .
```

- Your user directory is `/data/devel/ASIC/users/user3`.

## Creating a Work Area

Suppose you are a designer and you have developed an ALU design. Now you are ready to place your design files under revision control, so you can share them with the other members of your project team.

To put your files under revision control and share them with team members, you first must associate your local top-level design directory with a SyncServer folder (also called a vault folder). Then you check in your files to the vault.

Once you check in your files, the design directory you associated with the vault folder becomes your DesignSync work area. A **work area** is a directory hierarchy that reflects the data hierarchy in the vault. A work area is also the location from which you check file versions into and out of the vault. You are the only user accessing your work area, and you have full control over the files and files versions that exist there.

### Task 1: Associate your local directory with the SyncServer folder and check in files.

To put your files under revision control, you can use the DesignSync Workspace Wizard. The Wizard helps you perform two tasks: associate your local directory with the vault and check in files.

1. Invoke DesignSync. (From your UNIX shell, enter **DesSync**.)

The DesignSync main window appears and then the Workspace Wizard. (If the Workspace Wizard does not appear, select **Revision Control => Workspace Wizard**.)

2. Click **Create a new project** (I have files that I want to put under revision control, creating a new project vault). Then click **Next**.
3. In the **Select Vault** dialog, type the URL of the project where you will place your files or select one from the pull-down list. Then click **Next**.

The URL specifies the sync protocol plus the SyncServer name and port number and then the vault folder. For example, you might enter:

```
sync://ca-srvr.ACompany.com:2647/Projects/ASIC/alu8
```

(In certain cases, DesignSync displays a SyncSecurity dialog. Enter your user name and password as defined in the ProjectSync User's Profile. Then click **OK**.)

**Note:** The SyncServer folder does not have to exist for you to specify it. For example, suppose the `ASIC` folder exists on the SyncServer but there is no `alu8` subfolder. When you specify a vault folder that does not exist (`alu8`), DesignSync displays a warning so you can make sure that your vault path is correct and then creates the vault folder when you check in design data.

4. In the **Select Workspace** dialog, select the top level directory for the files you want to put under revision control (`alu8`) and click **Next**.

If the Wizard does not display the directory, click **Browse** to set **another directory**.

Optional: Type the name of a bookmark in the Bookmark field for future reference.

5. In the **Exclude Files** dialog, specify the names or file types of files you do not want to place under revision control. Then click **Next**.

For example, suppose you do not want to put under revision control temporary files, log files, certain files you used in an experiment, or files with notes to yourself. Your specification might be: `temp, *.log, experiment1.*, notes.txt`.

6. In the **Specify Object State** dialog, in answer to the question "What should be left behind when the checkin completes?", select **Unlocked copies**. Then click **Next**.
7. In the **Specify Check-In Comment** dialog, type a comment about the files you are checking in. Then click **Next**.
8. In the **Finished** dialog, click **Finish**.

The Workspace Wizard:

- Creates the `alu8` vault folder on the SyncServer and associates it with (`ca-srvr.ACompany.com:2647/Projects/ASIC/alu8`) with your project work area directory (`/data/devel/ASIC/users/user3/alu8`).
- Sets the persistent selector list for the work area. This list specifies the version or branch a revision control command operates on when no version or branch is explicitly specified for the operation (through the **Version Selector** field or the **-version** and **-branch** options to commands).
- Checks in the files to the vault (the SyncServer folder you specified)
- Fetches copies of those files from the vault to your workspace.

The files are now under DesignSync revision control. To access these new files in the vault, other team members need to use DesignSync to repopulate their work areas.

**Tip:** Repopulate your work area periodically to keep it up-to-date, reflecting the most recent versions and any new files added to the vault. When you use `populate` to update your work area, DesignSync performs an incremental `populate` operation. An



incremental populate is a kind of fast populate operation which fetches copies of only the vault folders that have changed.

---

## Creating a Work Area

To join the development effort for a project, you first must create and populate a work area in your user directory. A **work area** is a directory hierarchy that reflects the data hierarchy in the vault. A work area is also the location from which you check file versions into and out of the vault. You are the only user accessing your work area, and you have full control over the files and file versions that exist there.

### Task 1: Create and populate your work area folder

To create and populate your work area, you can use the DesignSync Workspace Wizard. The Wizard helps you perform two tasks: associate your work area with the vault and populate your work area with files from the project.

1. Invoke DesignSync. (From your UNIX shell, enter **DesSync**.)

The DesignSync main window appears and then the Workspace Wizard. (If the Workspace Wizard does not appear, select **Revision Control => Workspace Wizard**.)

2. Click **Join an existing project (I want to create a new workspace by populating files from an existing project vault)**. Then click **Next**.
  - o In the **Select Vault** dialog, type the URL of the project you want to join or select one from the pull-down list. Then click **Next**.  
The URL specifies the sync protocol plus the SyncServer name and port number and then the vault folder. For example, you might enter:  
`sync://ca-srvr.ACompany.com:2647/Projects/ASIC`

(In certain cases, DesignSync displays a SyncSecurity dialog. Enter your user name and password as defined in the ProjectSync User's Profile. Then click **OK**.)
4. In the **Select Workspace** dialog, select the directory to be the top level directory for the work area and click **Next**.

If the directory does not exist, you can create it by clicking **Browse** and creating a new folder. To set up your work area for the ASIC project files for example, under the `/data/devel/ASIC/users/user3` directory you might create the `Latest` subdirectory. Then you would select the `Latest` directory and click **Next**.

Optional: Type the name of a bookmark in the Bookmark field for future reference.

5. In the **Specify Selector or Configuration** dialog, specify the branch or version of design files you want to be put in your work area. Then click **Next**.
  - **If your environment does not use branches**, you can accept the default selector of **Trunk**. (Trunk is the default branch tag for branch 1, the default branch.) DesignSync fetches the Latest version of files on the Trunk branch.
  - **If your development environment uses branches**, type the branch selector for your project's files (for example `Rel14:Latest`).
6. In the **Specify Object State** dialog, in answer to the question "In what state should the objects be populated?", select **Unlocked copies**.
7. In the **Specify the Mirror** dialog, select the mirror directory. If you are not using a mirror model, click **Next**.
8. In the **Finished** dialog, click **Finish**.

The Workspace Wizard:

- Associates the vault folder (`ca-srvr.ACompany.com:2647/Projects/ASIC`) with your project work area directory (`Latest`).
- Sets the persistent selector list for the work area. This list specifies the version or branch a revision control command operates on when no version or branch is explicitly specified for the operation (through the **Version Selector** field or the **-version** and **-branch** options to commands).
- Fetches copies of project files from the vault.

**Tip:** Repopulate your work area periodically to keep it up-to-date, reflecting the most recent versions and any new files added to the vault. When you use populate to update your work area, DesignSync performs an incremental populate. An incremental populate is a kind of fast populate operation which fetches copies of only the vault folders that have changed.

---

## Creating File Versions

As you go about your daily design tasks, you'll need to make modifications to the design data files. You make all of these modifications in your private work area to files that you have checked out of the vault. When you're ready to share your modifications with other users, you can check your files into the SyncServer vault, where other users can access them.

### Task 2: Checking out a file for editing

To make changes and create a new version, you should first check out the file with a lock. When you check out the file with a lock, DesignSync fetches a copy of the file to your work area and sets the permissions to read-write. Then it locks the file in the vault, so that only you, the user holding the lock, can check in a newer version of the file.

**To check out a file with a lock:**

1. In DesignSync, click the file you want to check out (for example, `decoder.v`).

To select more than one file in a folder, press **Ctrl** and click each file.

2. Select **Revision Control => Check Out**.
3. In the Check Out dialog:
  - Click **Locked Copies**.
  - Type a comment if you want. (A comment usually is not required for a checkout and you can always add a comment when you check in the file.)
4. Click **OK**.

After checking out a file to your work area, you can invoke an editor appropriate for the file and make changes.

**Tip: To "undo" a checkout** when you have a file locked but have decided not to make changes, select **Revision Control => Cancel Checkout**. See Task 4: Releasing a lock on a file.

**To check out with a lock all files in a folder and all of its subfolders:**

Suppose you want to check out for edit not only all the files in the `top` folder but also all of the files in its `decoder`, `alu`, and `register` subfolders (a recursive checkout). All you have to do is select the `top` folder and perform a check-out operation:

1. In the DesignSync window, select the folder that contains the files and subfolders you want to check out (`top`).

To select more than one folder, press **Ctrl** and click each folder.

2. Select **Revision Control => Check Out**.
3. In the Check Out dialog:
  - Click **Unlocked Copies**.
  - Type a comment, if you want. (A comment is not required for a checkout. You can always add a comment when you check in the file.)
4. Click **Check Out**.

DesignSync checks out all the files in the `top` folder, in all its subfolders, in all their subfolders, and so on down through the hierarchy. DesignSync performs this type of checkout (the equivalent of the **co -recursive** command) by default when you perform a check-out operation from the DesignSync GUI. (To see the command options DesignSync uses, look at the command display box at the bottom of the Check Out dialog.)

**Task 3: Checking in a file**

Once you have made changes to a file, you need to check in the file to create a new version in the vault and to make it available to other team members.

1. In the DesignSync main window, select the file you want to check in (for example, `decoder.v`).

To select more than one file in a folder, press **Ctrl** and click each file.

2. Select **Revision Control => Check In**.
  3. In the Check In dialog:
    - Click **Unlocked copies**.
    - Type a comment, if you want. (It is a good idea to include a comment explaining the changes you made. In addition, your project leader may have set up access controls that require a check-in comment of a certain length.)
  4. Click **OK**.
- If you checked out a file for editing (by selecting the **Locked copies** option) but did not change it, the check-in operation creates no new version. Instead, it releases the lock and marks the file as read-only in your work area.

### To check in a folder of files:

If you checked out for edit many files in a work area, you can perform a blanket checkin to check them all in at once. This blanket checkin checks in all files that have been modified. Suppose, for example, that you want to check in all files in the `top` folder and in its `decoder`, `alu`, and `register` subfolders

1. In the DesignSync main window, select the folder that contains the files and subfolders that you want to check in (for example, `top`).
2. Select **Revision Control => Check In**.
3. In the Check In dialog:
  - Click **Unlocked copies**.
  - Type a comment, if you want. (It is a good idea to include a comment explaining the changes you made. In addition, your project leader may have set up access controls that require a check-in comment of a certain length.)
4. Click **OK**.

DesignSync inspects all files in the `top` folder and its subfolders and checks in any files that were modified. DesignSync performs this type of checkin (the equivalent of the **ci -recursive** command) by default when you perform a check in from the GUI. (To see the commands DesignSync uses, look at the command display box at the bottom of the Check In dialog.)

### Task 4: Releasing a lock

Suppose another team member notifies you that she needs to make changes to `decoder.v`, a design file that you have checked out with a lock. To release the lock on the file and make it available for others to edit, you can either:

- Check in the file (**Revision Control => Check In**), which releases the lock and puts a copy of the modified file into the vault. (In most situations, this is the action you'll probably want to take.)
- Cancel the check-out operation you performed on that file (**Revision Control => Cancel Checkout**). This option effectively performs an "un"checkout operation on the file you checked out with a lock: it releases the lock and (if your project leader defined a **default fetch state** of **share**) keeps a copy of the file in the directory.

To cancel a checkout (and release the lock) on a file:

1. In the DesignSync main window, select the file (for example, `decoder.v`). To select more than one file in a folder, in the DesignSync main window, press **Ctrl** and click each file.
2. Select **Revision Control => Cancel Checkout**.

Some points about how the **Cancel Checkout** option works:

- It cancels only a checkout that you performed. **To unlock a file that is locked by another user, select Revision Control => Unlock.**
- **If you have modified the file and want to keep your changes**, check in the file. See Task 3: Checking in a file.
- If you select a folder the DesignSync GUI and then select **Cancel Checkout**, DesignSync cancels the checkout on all of the files in the folder and in all of its subfolders.

## Configuration/Release Management

When you have in your work area a set of design files that meet some requirement, you'll want to mark this file set for future reference. For example, when the design passes simulation, you might want to mark the current versions of your design files as "passing". You can mark files by selecting the **Revision Control => Tag**. (The set of files that share a common tag is sometimes called a **configuration**.) You can also use the **Tag** selection to retrieve specific file versions when creating a work area in the future.

**Note:** This section describes working with DesignSync design configurations. SITaR configurations use a similar concept, which is described in Overview of SITaR Workflow. ProjectSync has a different concept of a configuration. For information, see ProjectSync Help: What Are Configurations?

## Task 5: Creating a design configuration/release

To create a design configuration, first select the items (files or folders) you want to have in the configuration; then select **Revision Control => Tag**. However, before you perform the tag operation, you should check any modified files into the vault and update your work area with a populate operation.

### Before you tag:

- **Check in files that you have modified.**
- The tag operation attaches a tag (a text string) to a specific file version. However, DesignSync attaches tags only to file versions in the vault, not to local copies of files. For this reason, before you perform a **Tag** operation, check in any files that are locally modified or checked out with a lock in your work area.

If you do not check in locally modified objects before you use the tag command, the tag operation displays an error message for each locally modified object and does not tag any version of those objects in the vault. This is the default behavior of the tag operation. For example, suppose you check out `decoder.v` (version 1.3) with a lock and modify it. If you try to tag the modified version, DesignSync displays an error message. To have the tag added to your modified version, you need to first check in `decoder.v`. DesignSync then creates version `decoder.v;1.4` in the vault. When you add a tag to the file, DesignSync tags version 1.4.

- **Make sure your work area is up-to-date.**

Although tags reside on file versions in the vault, the version of the file in your work area determines the version that DesignSync tags in the vault. Usually, your work area will contain all the versions you want to tag, especially if you have been simulating or verifying with a set of design files. But it may not. Your design files may not be copies of the latest checked in files. If you want your work area to remain current, it may be necessary to populate prior to simulating or verifying.

### To tag one or more files:

Suppose you want to tag several files in your work area with the string of `passing`:

1. In the DesignSync main window, select the files you want to tag.  
To select more than one file, press **Ctrl** and click each file.
2. Select **Revision Control => Tag**.
3. Click **Add a tag**.
4. In the **Tag:** field, type the tag name (for example, `passing`).
5. Click **Tag version in workspace**.
6. Click **OK**.

**To tag all file versions in a folder and all its subfolders within your work area:**

Suppose you want to tag all the files in the `top` folder of your work area with the string of `passing` and also have the tag applied to all files in its `decoder`, `alu`, and `register` subfolders:

1. In the DesignSync main window, select the folder (`top`) you want to tag. To select more than one folder, press **Ctrl** and click each folder.
2. Select **Revision Control => Tag**.
3. Click **Add a tag**.
4. Click **Recurse into folders**.
5. In the **Tag:** field, type the tag name (for example, `passing`).
6. Click **Tag version in workspace**.
7. Click **OK**.

DesignSync tags all the files in the `top` folder and in all its subfolders.

**If you change a file later and want to mark the new file version with an existing tag**, you can use the **Replace existing tags** option of the **Tag** dialog to move the tag to the new version.

For example, suppose you tag a set of files with the tag `passing`. Later you change the `decoder.v` file to correct an error in the header and check in the file, creating a new version (`1.2`, for example). This new version, of course, does not have the tag. To move the tag to the version `1.2`:

1. Select the `decoder.v` file and select **Revision Control => Tag**.
2. Click **Add a tag**.
3. Click **Replace existing tags**.
4. In the **Tag:** field, type the tag name, type the name of the existing tag (in this example, `passing`) you want to move to the new version of `decoder.v`.
5. Click **Tag version in workspace**.
6. Click **OK**.

#### Notes:

- This example of the tag operation does not specify a version with the file name. Checking in the file first created a new version in the vault and updated the work area. From the version in the work area, DesignSync determines the version to tag in the vault.
- This tag operation is the equivalent of the **tag -replace** command.

**You can also specify the file version to which DesignSync attaches the tag**, rather than letting the version in the workspace determine the version to be tagged. Use the **Tag specified version** option of the **Tag** dialog. For example:

1. Select the `decoder.v` file and select **Revision Control => Tag**.
2. Click **Add a tag**.
3. In the **Tag:** field, type `initial_vers`.
4. Click **Tag specified version**.
5. In the **Version/Branch:** field, type version you want to tag, for example, `1.1`.

Alternately, you can display a list of versions and choose the one you want to tag. Click on the down arrow and select **Get versions/tags**. In the **Get Tags/Versions** dialog, select a branch (most often it will be Current branch). Then select **Get versions** and the specific version you want.

6. Click **OK**.

This capability to tag a file version can be quite useful, considering that the tag string can be a version parameter. One use is to create a new configuration based on a previous configuration. For example:

1. Select the folder that contains the objects you want to tag and then select **Revision Control => Tag**.
2. Click **Add a tag**.
3. In the **Tag:** field, type `relB`.
4. Click **Tag specified version**.
5. In the **Version/Branch** field, type `relA`, where `relA` is an existing version.
6. Click **OK**.

DesignSync applies the tag `RelB` to file versions that have the `RelA` tag attached.

To remove a tag string from a file, use the **Delete a tag** option of the **Tag** dialog:

1. Select the `decoder.v` file and select **Revision Control => Tag**.
2. Click **Delete a tag**.
3. In the **Tag:** field, type the tag you want to delete, for example, `temp_tag1`.
4. Click **OK**.

This tag operation deletes the `temp_tag1` tag from the `decoder.v` file. (This tag operation is the equivalent of the **tag -delete** command.)

**To remove a tag string from all files in a folder and in all its subfolders**, select the folder and use the **Delete a tag** option of the **Tag** dialog:

1. Select the `top` folder and select **Revision Control => Tag**.
2. Click **Delete a tag**.
3. Click **Recurse into folders**.
4. In the **Tag:** field, type the tag you want to delete, for example, `temp_tag1`.
5. Click **OK**.



This tag operation deletes the `temp_tag1` tag from files in the `top` folder and all its subfolders. (This tag operation is the equivalent of the **tag -delete -recursive** command.)

### Task 6: Creating a work area based on a configuration/release

The operations that fetch a file version to a work area (populate, checkout) have a **Version** field where you can specify the version to be fetched.

While you can use the version selector to retrieve a specific numeric version, such as 1.5, the real power is in using a tag string as the version selector. For example, let's say you want to create a new work area that holds certain ASIC files--those that have been tagged with the `RelA` tag.

**To create a work area based on a configuration:**

1. In the DesignSync main window, select your user directory for the project (`/data/devel/ASIC/users/user3`).
2. Select **File => New => Folder** and create a new top level folder (`ASIC2`) for this configuration.
3. Select the top level folder you just created (`RelA`).
4. Select **Revision Control => Set Vault Association**.
5. In the **Modify the Vault** field, type the URL of the vault folder that contains the files you want to use as the basis for your new work area (in our example, `sync://ca-srvr.Acompany.com:2647/Projects/ASIC`).
6. In the **Modify the Selector** field, type the tag name you want to use to populate the new work area (`RelA`).
7. Click **OK**.
8. Select **Revision Control => Populate**
9. Click **Unlocked copies**.
10. Click **Recurse into folders** if not already selected.
11. Click **Incremental populate**, if not already selected.
12. Click **OK**.
  - Specifying a tag name for **Version** causes the populate operation to fetch into your work area file versions that have the specified tag (in our example, `RelA`). The operation does not remove any file versions that do not have the `RelA` tag. In your `RelA` work area, you can create any new files you want. However, these new files will not have any tag unless you attach one with a tag operation.
  - The **Recurse into folders** option causes DesignSync to populate not just the current folder (`RelA`) but all its subfolders, recreating the vault folder hierarchy in your work area.

**To delete from your work area all file versions that are not part of a specified configuration (that is, not tagged with a specified string),** perform a populate operation, selecting the **Force overwrite of local modifications** option. For example,

after DesignSync performs the following populate operation, only files tagged with the `RelC_proposed` tag exist in the your work area:

1. In the DesignSync main window, click the folder you want to update.
2. Check that the vault association between your work area folder and the vault folder has been set appropriately. **Select File => Properties** and click the **Revision Control** tab. (For our example, the vault that the `RelA` work area is associated with should be `sync://ca-srvr.Acompany.com:2647/Projects/ASIC`.)
3. Select **Revision Control => Populate**.
4. Click these options:
  - o **Unlocked copies**
  - o **Recurse into folders**
  - o **Force overwrite of local modifications**
5. Click **Advanced** and in the **Version** field, type `RelC_proposed`.
6. Click **OK**.

**WARNING:** The **Force overwrite of local modifications** option deletes from your work area any file versions that are not tagged with the specified string, including any files that are checked out for edit or locally modified. Use this option with care. However, this option does not affect unmanaged files (files that are not under DesignSync revision control) in your work area.

---

## Working with Files in Your DesignSync Work Area

Although directories and files in your work area may look the same as your other UNIX directories and files, your work area contains information (metadata) that DesignSync uses to manage objects under its revision control. To ensure that DesignSync can track file versions and perform revision control operations successfully, **always use DesignSync to create, delete, rename, and move directories and files in your work area.**

- To **create a directory** within your work area, in the DesignSync main window, select **File => New => Folder** (`mkfolder` command).
  - To **delete directories or files** within your work area, in the DesignSync main window, select the folder or file and then select **File => Delete** (`rmfolder` and `rmfile` commands).
  - To **rename or move directories or files** within your work area, use the DesignSync `mvfolder` and `mvfile` commands.
- 

## Getting Started with the Command Shell

## Using the DesignSync Command Shell

This document provides instructions to help you create a work area and quickly get started using DesignSync to manage your design data.

---

### Assumed Environment

**(UNIX platform, dssc or stcl command shell, shared cache, locking work style)**

The instructions in this document assume that:

- You are working in a UNIX environment.
- The DesignSync tools have been installed.
- To use DesignSync, you invoke the DesignSync command shell (**dssc**) and use DesignSync commands. (You can also use the stcl command shell.)
- In your environment:
  - You have set the **SYNC\_DIR** variable. (This environment variable defines the path to your DesignSync installation directory; you need to set this variable before running DesignSync.)
  - The **PATH** variable includes **\$SYNC\_DIR/bin** so that the operating system can find the dssc executable.

This document uses a scenario to illustrate the steps you use to get started. In the scenario:

- You are a designer working on a project called ASIC.
- On a DesignSync server (also called a SyncServer) your project leader or DesignSync tools administrator has created a SyncServer folder where you can put your ASIC design files. The SyncServer has the <host name>:<port> of `ca-srvr.Acompany.com:2647` (a remote vault). The URL for the Projects folder is: `sync://ca-srvr.Acompany.com:2647/Projects`.
- Your project leader has set up a **shared cache** on the Local Area Network (LAN) that holds copies of each version of all the project's files. Project members reference these files in the cache instead of storing copies of files in their own work areas.
- Your project team follows the **locking work style** where you always check out with a lock an object that you plan to edit. Other team members can fetch (check out without a lock) the object, but no one else should make changes while you hold the lock.
- Your project leader has set up a user directory with subdirectories for the project team members. The directory has the UNIX hierarchy:

```
/data/devel/ASIC/users
user1 user2 user3 . .
```

- Your user directory is `/data/devel/ASIC/users/user3`.

---

## Creating a Work Area - Putting Files Under Revision Control

Suppose you are a designer who has developed a design for an ALU. You are ready to place your design files under revision control, so that you can share them with the other members of your project team.

To put your files under revision control and share them with team members, you first must associate your local top-level design directory with a SyncServer folder (also called a vault folder). Then you check in your files to the vault.

Once you check in your files, the design directory you associated with the vault folder becomes your DesignSync work area. A **work area** is a directory hierarchy that reflects the data hierarchy in the vault. A work area is also the location from which you check file versions into and out of the vault. You are the only user accessing your work area, and you have full control over the files and files versions that exist there.

### Task 1: Associate your work area directory with the vault (SyncServer folder).

Before you can put your design files under revision control, your `alu8` directory needs to be associated with a SyncServer folder. To make the association, use the `dssc` or `stcl` command **setvault**.

To associate your work area with a SyncServer folder:

1. Invoke the DesignSync command shell. (At the UNIX prompt, enter `dssc`.)
2. Use the DesignSync **cd** command to change directory to the top level directory for the files you want to put under revision control.
3. Use the DesignSync **setvault** command to associate your directory with a corresponding SyncServer folder for your project. The SyncServer folder is specified with a URL that uses the sync protocol plus the SyncServer name and port number and then the vault folder (`sync://ca-srvr.Acompany.com:2647/Projects/ASIC/alu8`).

**Note:** The SyncServer folder does not have to exist for you to specify it. For example, suppose the `ASIC` folder exists on the SyncServer but there is no `alu8` subfolder. When you specify a vault folder that does not exist (`alu8`), DesignSync displays a warning so you can make sure that your vault path is correct and then creates the vault folder when you check in design data.

```
%dssc
dss> cd /data/devel/ASIC/users/user3/alu8
dss> setvault sync://ca-
```

```

srvr.Acompany.com:2647/Projects/ASIC/alu8 .
dss> exit
%
```

## Task 2: Check in your files.

The next step is to perform the initial checkin of your ALU design files into the SyncServer vault folder.

To check in of all files in the `alu8` folder and all of its subfolders:

1. In your work area, change directory to the top-level directory containing the files you want to check in.
2. Use the DesignSync command `ci` to check in the files. It is a good idea to include a comment (with the `-comment` option) explaining the changes you made. In addition, your project leader may have set up access controls that require a check-in comment of a certain length.

```

dss> cd /data/devel/ASIC/users/user3/ASIC/alu8
dss> ci -comment "8-bit ALU" -new -share -recursive
/data/devel/ASIC/users/user3/ASIC/alu8
dss> exit
%
```

- The `-new` option causes DesignSync to create a new version (1.1) of each file on a new branch (1).
- The `-recursive` option causes DesignSync to check in all of the files in the ASIC folder and in all of its subfolders, recreating your work area ASIC directory in the vault folder. To check in recursively, you must specify the `-recursive` option; otherwise DesignSync checks in the files in the current folder only.
- You use the `-share` option when your project has a shared (LAN) cache. This option causes the `ci` command to fetch copies of files into the project's shared (LAN) cache, not into your own work area. Instead, your work area contains links to the files in the cache. The checkin command fetches files into the cache only if they are not already there. For this reason, using `ci -share` is the fastest method for creating a work area and makes efficient use of disk space for your team.

**Note:** If your project leader has set up a **default fetch state** of **share**, you do not need to use the `-share` option. DesignSync automatically includes it with the `populate`, `co`, `ci`, and `cancel` commands.

Other team members will see the files you placed under revision control when they populate their work areas.

**Tip:** You also should populate your work area periodically to keep it up to date, reflecting the most recent versions and new files in the vault. To update your work area, use the `populate` command.

By default, the `populate` command performs a fast populate operation using the **incremental** option. This option causes the `populate` command to fetch copies of only the vault folders that have changed.

Now you are ready to check files out and in, creating new file versions.

---

## Creating a Work Area - Joining a Project Already Under Revision Control

To join the development effort for a project, you first must create and populate a work area in your user directory. A **work area** is a directory hierarchy that reflects the data hierarchy in the vault. A work area is also the location from which you check file versions into and out of the vault. You are the only user accessing your work area, and you have full control over the files and file versions that exist there.

### Task 1: Associate your work area folder with a SyncServer folder.

Your work area folder needs to be associated with a SyncServer folder. To make the association, use the `dssc` or `stcl` command **setvault**.

To associate your work area with a SyncServer folder:

1. Invoke the DesignSync command shell. (At the UNIX prompt, enter `dssc`.)
2. Use the DesignSync `cd` command to change directory to your user directory for the project.
3. Use the DesignSync `mkfolder` command to create a new folder (`Latest`) for the project files you plan to work on. This is the root (top level) folder of your work area. (A folder is a directory that is under DesignSync revision control.)
4. Use the DesignSync `cd` command to change directory to the top level folder you just created.
5. Use the DesignSync `setvault` command to associate the SyncServer folder (`ca-srvr.Acompany.com:2647/Projects/ASIC`) with your project work area folder (`/data/devel/ASIC/users/user3/Latest`). The SyncServer folder is specified with a URL that uses the `sync` protocol plus the SyncServer name and port number and then the vault folder (`sync://ca-srvr.Acompany.com:2647/Projects/ASIC`).

```
% dssc
dss> cd /data/devel/ASIC/users/user3
dss> mkfolder Latest
dss> cd Latest
dss> setvault sync://ca-srvr.Acompany.com:2647/Projects/ASIC .
dss> exit
%
```

## Task 2: Populate the work area with project data

The next step is to fill your work area folder structure with directories and files that already have been checked in to the SyncServer for this project. The most common approach is to get a copy of the most recent version of every file by using the **populate** command.

To populate your work area with copies of the data and files in the vault:

1. Change directory to the folder you created to hold your project files.
2. Use the DesignSync **populate** command to fill your work area folder with copies of all project directories and files.

```
% dssc
dss> cd /data/devel/ASIC/users/user3/Latest
dss> populate -recursive -share
dss> exit
%
```

- You can abbreviate command words, if you choose. For example:

```
% dssc pop -rec
```

- The **-recursive** option causes DesignSync to populate not just the current folder (`Latest`) but all its subfolders, recreating the vault folder hierarchy in your work area. To populate recursively, you must specify the **-recursive** option; otherwise DesignSync populates just the current folder.
- You use the **-share** option to populate your work area when your project has a shared file cache. This option causes the **populate** command to fetch copies of files into the project's shared file cache, not into your own work area. Instead, your work area contains links to the files in the cache. The populate command fetches files into the cache only if they are not already there. For this reason, using **populate -share** is the fastest method for creating a work area and makes efficient use of disk space for your team.

**Note:** If your project leader has set up a **default fetch state** of **share**, you do not need to use the **-share** option. DesignSync automatically includes it with the **populate**, **co**, **ci**, and **cancel** commands.

**Tip:** Repopulate your work area periodically to keep it up-to-date, reflecting the most recent versions and any new files added to the vault. To update your work area, use the **populate** command.

By default, the populate command performs a fast populate operation using the **-incremental** option. This option causes the **populate** command to fetch copies of only the vault folders that have changed.

Now you are ready to check files out and in, creating new file versions.

---

## Creating File Versions

As you go about your daily design tasks, you'll need to make modifications to the design data files. You make all of these modifications in your private work area to files that you have checked out with a lock from the vault . When you're ready to share your modifications with other users, you can check your files into the SyncServer vault, where other users can access them.

### Task 3: Checking out a file for editing

To make changes and create a new version, you should first check out the file with a lock. When you check out the file with a lock, DesignSync fetches a copy of the file to your work area and sets the permissions to read-write. Then it locks the file in the vault, so that only you, the user holding the lock, can check in a newer version of the file.

#### To check out a file with a lock:

1. In your work area, change your directory to the folder that contains the file you want to check out.
2. Use the DesignSync command **co -lock** to check out the file with a lock.

```
% dssc
dss> cd /data/devel/ASIC/users/user3/Latest/top/decoder
dss> co -nocomment -lock decoder.v
dss> exit
%
```

Use the **-nocomment** option when you want to check the file out without describing your intended changes. You can still add a comment (with the **-comment** option) when you check in the file.

After checking out a file to your work area, you can invoke an editor appropriate for the file and make changes.

**Tip:** To "undo" a checkout when you have a file locked but have decided not to make changes, use the **cancel** command. See Task 5: Releasing a lock on a file.

#### To check out with a lock all files in one folder:

1. In your work area, change your directory to the folder that contains the files you want to check out.



## 2. Use the DesignSync command **co -lock** and a wildcard (\*).

```
% dssc
dss> cd /data/devel/ASIC/users/user3/Latest/top/decoder
dss> co -nocomment -lock *
dss> exit
%
```

This command checks out all of the files in the `decoder` folder of the ASIC project.

**To check out with a lock all files in a folder and all of its subfolders:**

Suppose you want to check out for edit not only all the files in the `top` folder but also all of the files in its `decoder`, `alu`, and `register` subfolders. You use the `-recursive` option in addition to the `co -lock` command:

1. In your work area, change directory to the folder higher up in the hierarchy. For example, if you are working in the `/users/user3/Latest/top` folder, change directory to the `Latest` folder.
2. Use the DesignSync command **co -lock -recursive** to check out the files.

```
% dssc
dss> pwd
/data/devel/ASIC/users/user3/Latest/top
dss> cd /data/devel/ASIC/users/user3/Latest
dss> co -recursive -lock top
dss> exit
%
```

The **-recursive** option causes DesignSync to check out all files in the `top` folder, in all its subfolders, in all their subfolders, and so on down through the hierarchy.

### Task 4: Checking in a file

Once you have made changes to a file, you need to check in the file to create a new version in the vault and to make it available to other team members.

**To check in a file:**

1. In your work area, change your directory to the folder that contains the file you want to check in.
2. Use the DesignSync **ci** command to check in the file to the vault. It is a good idea to include a comment (with the **-comment** option) explaining the changes you made. In addition, your project leader may have set up access controls that require a check-in comment of a certain length.

## DesignSync Data Manager User's Guide

```
% dssc
dss> cd /data/devel/ASIC/users/user3/Latest/top/decoder
dss> ci -comment "added power-up state vector" -share decoder.v
dss> exit
%
```

- If you checked out a file for editing (using the **co -lock** command) but did not change it, the check-in operation creates no new version. Instead, it releases the lock and marks the file as read-only in your work area.
- The **-share** option causes the check-in operation to put a copy of the changed file into the shared cache (instead of your work area) and create a link in your work area to the file version in the cache.

### To check in all files in a folder and all of its subfolders (a "blanket" checkin):

If you checked out for edit many files in a work area, you can perform a blanket checkin from a folder higher up in the hierarchy. This blanket operation checks in all files that have been modified.

1. In your work area, change directory to the folder higher up in the hierarchy. For example, if you are working in the `/data/devel/ASIC/users/user3/Latest/top` folder, change directory to the `Latest` folder.
2. Use the DesignSync command **ci -recursive** to check in the files. It is a good idea to include a comment (with the **-comment** option) explaining the changes you made. In addition, your project leader may have set up access controls that require a check-in comment of a certain length.

```
% dssc
dss> pwd
/data/devel/ASIC/users/user3/Latest/top
dss> cd /data/devel/ASIC/users/user3/Latest
dss> ci -comment "extensive changes for reset" -share -recursive
top
dss> exit
%
```

The **-recursive** option causes DesignSync to inspect all files in the `top` folder and its subfolders in your work area and check in any files that were modified.

### Task 5: Releasing a lock

Suppose another team member notifies you that she needs to make changes to `decoder.v`, a design file that you have checked out with a lock. To release the lock on the file and make it available for others to edit, you can either:

- Check in the file with the **ci** command, which releases the lock and puts a copy of the modified file into the vault. (In most situations, this is the action you'll probably want to take.)
- Cancel the check-out operation you performed on that file by using the **cancel** command. This command effectively performs an "un"checkout operation on the file you checked out with a lock: it releases the lock and, when used with the **-share** option, keeps a copy of the file in the cache directory.

To cancel a checkout (and release the lock) on a file:

1. Make sure you are in the folder containing the file.
2. Use the DesignSync **cancel** command to release the lock.

```
% dssc
dss> cd /data/devel/ASIC/users/user3/Latest/top/decoder
dss> cancel -share decoder.v
dss> exit
%
```

Some points about how the **cancel** command works:

- It cancels only a checkout that you performed. **To unlock a file that is locked by another user, use the **unlock** command.**
- **If you have modified the file but you do not want to keep your changes, use **cancel -force**.** This command cancels your checkout and replaces the locally modified file with a valid version from the vault.
- **If you have modified the file and want to keep your changes, check in the file with the **ci** command.** See Task 4: Checking in a file.

---

## Configuration/Release Management

When you have in your work area a set of design files that meet some requirement, you'll want to mark this file set for future reference. For example, when the design passes simulation, you might want to mark the current versions of your design files as "passing". You can mark files with the DesignSync **tag** command. (The set of files that share a common tag is sometimes called a **configuration**.) You can also use the **tag** command to retrieve specific file versions when creating a work area in the future.

**Note:** This section describes working with DesignSync design configurations. SITaR configurations use a similar concept, which is described in Overview of SITaR Workflow. ProjectSync has a different concept of a configuration. For information, see ProjectSync Help: What Are Configurations?

### Task 6: Creating a design configuration/release

## DesignSync Data Manager User's Guide

To create a design configuration, first select the items (files or folders) you want to have in the configuration; then use the **tag** command. However, before you perform the tag operation, you should check any modified files into the vault and update your work area using the **populate** command.

### Before you tag:

- **Check in files that you have modified.**
- The **tag** command attaches a tag (a text string) to a specific file version. However, DesignSync attaches tags only to file versions in the vault, not to local copies of files. For this reason, before you use the **tag** command, check in any files that are locally modified or checked out with a lock in your work area.

If you do not check in locally modified objects before you use the tag command, the tag operation displays an error message for each locally modified object and does not tag any version of those objects in the vault. This is the default behavior of the tag operation. For example, suppose you check out `decoder.v` (version 1.3) with a lock and modify it. If you try to tag the modified version, DesignSync displays an error message. To have the tag added to your modified version, you need to first check in `decoder.v`. DesignSync then creates version `decoder.v;1.4` in the vault. When you add a tag to the file, DesignSync tags version 1.4.

- **Make sure your work area is up-to-date.**

Although tags reside on file versions in the vault, the version of the file in your work area determines the version that DesignSync tags in the vault. Usually, your work area will contain all the versions you want to tag, especially if you have been simulating or verifying with a set of design files. But it may not. Your design files may not be copies of the latest checked in files. If you want your work area to remain current, it may be necessary to populate prior to simulating or verifying.

### To tag all file versions in a folder and all its subfolders within your work area:

Suppose you want to tag all the files in the `top` folder of your work area with the string `passing` and also have the tag applied to all files in its `decoder`, `alu`, and `register` subfolders (recursive behavior):

1. In your work area, change directory to the folder that contains the files and subdirectories with files you want to tag.
2. Use the DesignSync command **tag -recursive** to tag all the files in that folder (`top`) and all the files in its subfolders.

```
% dssc
dss> cd /data/devel/ASIC/users/user3/Latest/top
dss> tag -recursive passing .
```

```
dss> exit
%
```

**If you change a file later and want to mark the new file version with an existing tag, you can use the `tag -replace` command to move the tag to the new version.**

For example, suppose you tag a set of files with the tag `passing`. Later you change the `decoder.v` file to correct an error in the header and check in the file, creating a new version. The new version, of course, does not have the tag. To move the tag to the new version, you use the `tag -replace` command and specify the file's name.

```
% dssc
dss> cd /data/devel/ASIC/users/user3/Latest/top/decoder
dss> ci -share -com "corrected error in header" decoder.v
dss> tag -replace passing decoder.v
dss> exit
%
```

**Note:** This example of the `tag` command does not specify a version with the file name. Checking in the file first created a new version in the vault and updated the work area. From the version in the work area, DesignSync determines the version to tag in the vault.

**You can also specify the file version to which DesignSync attaches the tag, rather than letting the version in the work area determine the version to be tagged. Use the `-version` option. For example:**

```
% dssc
dss> cd /data/devel/ASIC/users/user3/Latest/top/decoder
dss> tag -version 1.1 initial_vers decoder.v
dss> exit
%
```

This example attaches the tag `initial_vers` to version 1.1 of the `decoder.v` file.

This capability to tag a file version can be quite useful, considering that the tag string can be a version parameter. One use is **to create a new configuration based on a previous configuration**. For example:

```
% dssc
dss> cd /data/devel/ASIC/users/user3/Latest
dss> tag -version RelA RelB -recursive .
dss> exit
%
```

## DesignSync Data Manager User's Guide

This command operates on all files in the `Latest` folder and all its subfolders and applies the tag `RelB` to all file versions that have the `RelA` tag attached. Then you can use the **tag -replace** command to move the `RelB` tag to different file versions as you identify differences between the `RelA` and `RelB` configurations.

**To remove a tag string from a single file**, use the **tag -delete** command. For example:

```
% dssc
dss> cd /data/devel/ASIC/users/user3/Latest/top/decoder
dss> tag -delete temp_tag1 decoder.v
dss> exit
%
```

**To remove a tag string from all files in a folder and in all its subfolders**, use the **tag -delete -recursive** command. For example:

```
% dssc
dss> cd /data/devel/ASIC/users/user3/Latest/top
dss> tag -delete temp_tag1 -recursive .
dss> exit
%
```

This tag command deletes the `temp_tag1` tag from files in the `top` folder and all its subfolders.

### Task 7: Creating a work area based on a configuration/release

The commands that fetch a file version to a work area (**populate**, **co**) accept a **-version** option where you can specify the version to be fetched.

While you can use the **-version** option to retrieve a specific numeric version, such as 1.5, the real power is in using a tag string as the version specifier. For example, suppose you want to create a new work area that holds ASIC files that have been tagged with the `RelA` tag.

**To create a work area based on a configuration:**

1. Change directory to your user directory for the project (`/data/devel/ASIC/users/user3`).
2. Use the DesignSync **mkfolder** command to create a new folder (`RelA`) for this configuration.
3. Change directory to the top level folder you just created (`RelA`).
4. Use the DesignSync **setvault** command to associate the SyncServer folder (`sync://ca-srvr.Acompany.com:2647/Projects/ASIC`) with your new work area folder (`RelA`).

5. Change directory to the `RelA` folder in the new work area.
6. Use the DesignSync command **populate -share -recursive** command with the **-version** option to fill your new work area folder with links to the cache for all file versions that have the `RelA` tag.

(As always, the **-share** option causes the **populate** command to populate the shared cache and create links from your work area to the cache.)

```
% dssc
dss> cd /data/devel/ASIC/users/user3
dss> mkfolder RelA
dss> cd RelA
dss> setvault sync://ca-srvr.Acompany.com:2647/Projects/ASIC .
dss> populate -share -recursive -version RelA
dss> exit
%
```

**Note:** This command sequence creates a new work area (`/data/devel/ASIC/users/user3/RelA`) and populates it with copies of only those ASIC files in the vault that have a tag of `RelA`. In your `RelA` work area, you can create any new files you want. However, these new files will not have any tag unless you attach one with the **tag** command.

The **populate -version <tagname>** command fetches into your work area file versions that have the specified tag (in our example, `RelA`). The command does not remove any file versions that do not have the `RelA` tag.

You can also use the **tag** and **populate -version <tagname>** command to **add existing files to your new work area**. Suppose you modify some files in the first work area you created (`/data/devel/ASIC/users/user3/Latest`) and then decide that you want to include the changes in your new workarea (`RelA`). To put copies of the changed files in your new work area, you:

1. Check in the modified files to the vault.
2. Tag each modified file in the old work area (`Latest`) with the tag string `MyChanges`.
3. Change directory to your new workarea  
`/data/devel/ASIC/users/user3/RelA`.
4. Use the **populate -version** command to fetch copies of the tagged files into `RelA`.

For example:

```
% dssc
dss> pwd
/data/devel/ASIC/users/user3/Latest/top/decoder
dss> tag MyChanges decoder.v decoder.gv
```

## DesignSync Data Manager User's Guide

```
dss> cd /data/devel/ASIC/users/user3/RelA
dss> populate -share -recursive -version MyChanges
dss>exit
%
```

The resulting work area contains the RelA release plus all file versions tagged with MyChanges.

**To delete from your work area all file versions that are not part of a specified configuration (not tagged with a specified string), use the DesignSync command `populate -force`.** For example, after DesignSync executes the following commands, only files tagged with the RelC\_proposed tag exist in the your work area:

```
% dssc
dss> cd /data/devel/ASIC/users/user3/RelA
dss> populate -share -force -recursive -version RelC_proposed
dss> exit
%
```

**WARNING:** The `-force` option deletes from your work area any file versions that are not tagged with the specified string, **including any files that are checked out for edit or locally modified. Use this option with care.** However, the `-force` option does not affect unmanaged files (files that are not under DesignSync revision control) in your work area.

---

## Working with Files in Your DesignSync Work Area

Although directories and files in your work area may look the same as your other UNIX directories and files, your work area contains information (metadata) that DesignSync uses to manage objects under its revision control. To ensure that DesignSync can track file versions and perform revision control operations successfully, **always use DesignSync commands to create, delete, rename, and move directories and files in your work area.**

- To **create a directory** within your work area, use the **mkfolder** command.
- To **delete directories or files** within your work area, use the **rmfolder** and **rmfile** commands.
- To **rename or move directories or files** within your work area, use the **mvfolder** and **mvfile** commands.



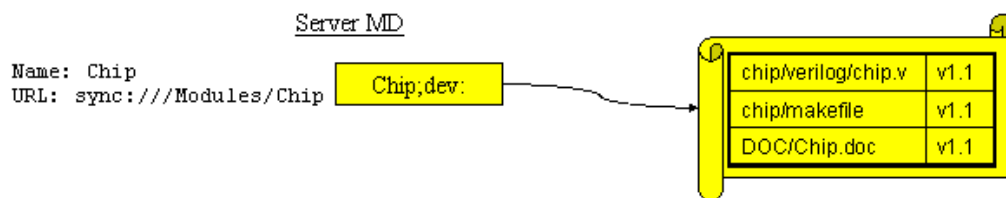


# Tutorials

## Creating Modules and Module Data

### Module Hierarchy: Module Structure

The animated illustration below will continually repeat, advancing a step every 5 seconds. In this use case, "Server MD" refers to the server metadata. Read an overview of module hierarchy.



Step 1 of 4

View the individual steps.

### Creating Module Hierarchy: Overview

The animated examples below should be reviewed in the order they are listed. Each animated topic has a hyperlink to step-by-step illustrations. View the module structure that is used in all of the examples below.

1. Create the Module

2. Add Files and Check In
3. Add an HREF to a Module in the Workspace
4. Populate with Dynamic HREF Mode
5. Add an HREF to a Module not in the Workspace

Read an overview of module hierarchy.

## Creating Module Hierarchy: Create the Module

The animated illustration below will continually repeat, advancing a step every 5 seconds. In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy.

Workspace Directory Structure

MyModules /home/ian/MyModules

Step 1 of 9

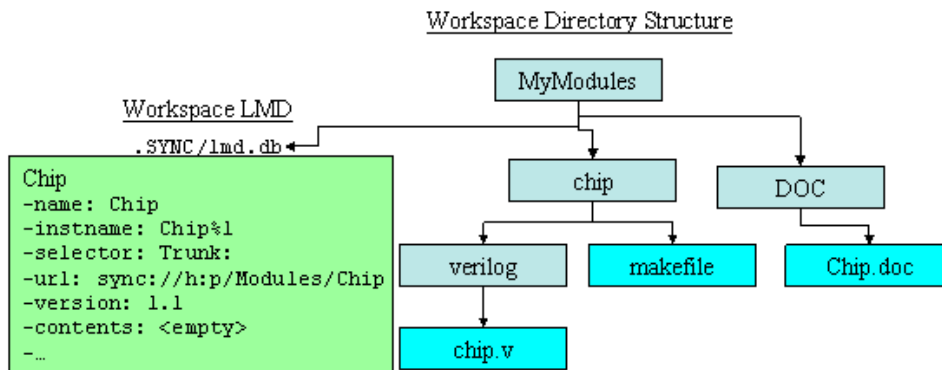
View the individual steps.

View the module structure used in this example.

View the subsequent example of creating module hierarchy: Add Files and Check In

## Creating Module Hierarchy: Add Files and Check In

The animated illustration below will continually repeat, advancing a step every 5 seconds. In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy.



Step 1 of 9

View the individual steps.

View the module structure used in this example.

View the subsequent example of creating module hierarchy: Add an HREF to a Module in the Workspace

## Creating Module Hierarchy: Add an HREF to a Module in the Workspace

The animated illustration below will continually repeat, advancing a step every 5 seconds. In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy.

Workspace LMD

MyModules

```
Fetch the "Chip" module:  
stcl> cd /home/ian/MyModules  
stcl> populate -get sync://h:p/Modules/Chip;dev:Latest
```

Step 1 of 8

View the individual steps.

View the module structure used in this example.

View the subsequent example of creating module hierarchy: Populate with Dynamic HREF Mode

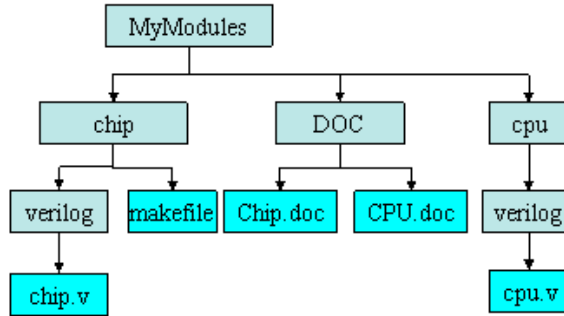
## Creating Module Hierarchy: Populate with Dynamic HREF Mode

The animated illustration below will continually repeat, advancing a step every 5 seconds. In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy.

Workspace LMD

```
Chip  
-name: Chip  
-version: 1.3
```

```
CPU  
-name: CPU  
-version: 1.2  
-hrefs:
```



Step 1 of 7

View the individual steps.

View the module structure used in this example.

View the subsequent example of creating module hierarchy: Add an HREF to a Module not in the Workspace

## Creating Module Hierarchy: Add an HREF to a Module not in the Workspace

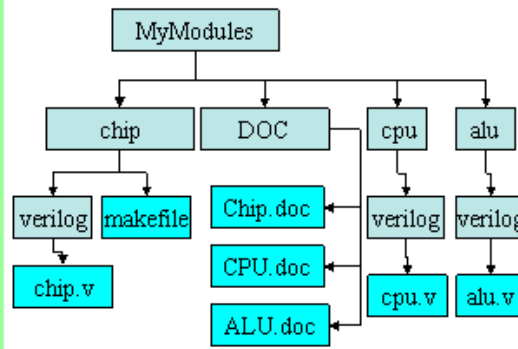
The animated illustration below will continually repeat, advancing a step every 5 seconds. In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy.

Workspace LMD

```
Chip
-name: Chip
-version: 1.3
```

```
CPU
-name: CPU
-version: 1.3
-hrefs: sync://.../ALU
...
```

```
ALU
-name: ALU
-version: 1.4
-hrefs:
```



Step 1 of 5

View the individual steps.

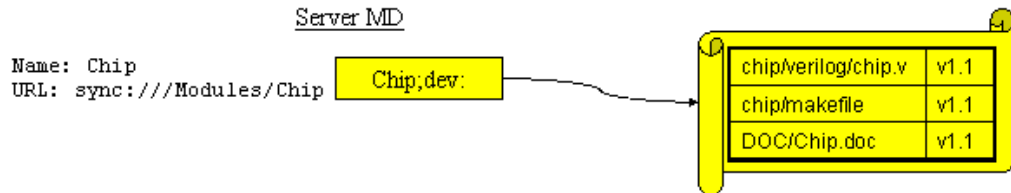
View the module structure used in this example.

View the subsequent example of creating module hierarchy: Creating a Peer Structure Module Hierarchy.

## Creating a Peer Structure Module Hierarchy

The animated illustration below will continually repeat, advancing a step every 5 seconds. In this use case, "Server MD" refers to the server metadata. Read an overview of module hierarchy.

This example creates a peer structure, in which referenced modules, when fetched, are at the same directory level as the module that references the submodule.



Step 1 of 5

View the individual steps.

## Updating Module Hierarchy

### Modifying Module Hierarchy: Overview

The animated examples below should be reviewed in the order they are listed. Each animated topic has a hyperlink to step-by-step illustrations. View the module structure that is used in all of the examples below.

1. The ALU team develops a new "Gold" version of their module.
2. Chip team members use the new version of the ALU module..
3. The CPU team reverts to an earlier version of the ALU module.

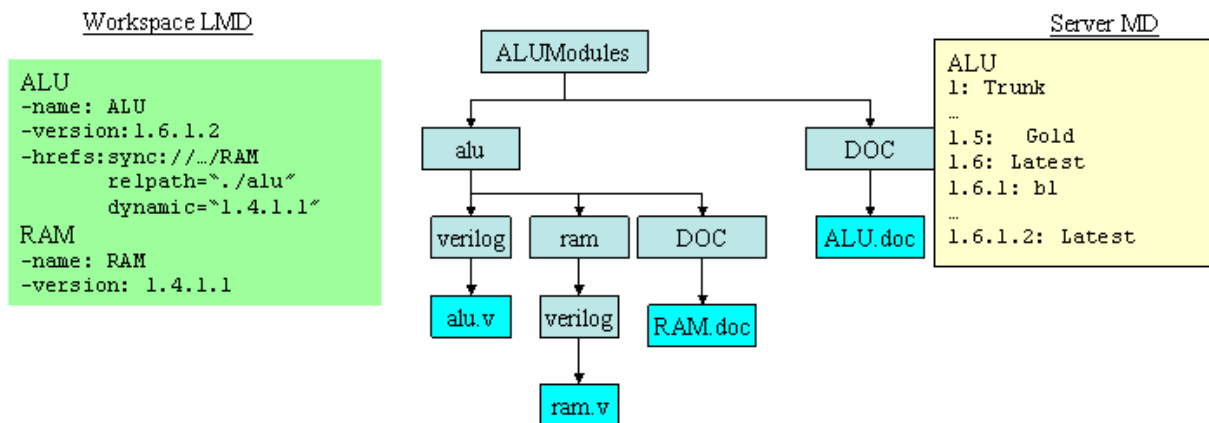
Read an overview of module hierarchy.

### Modifying Module Hierarchy: New "Gold" Version of ALU Created



The animated illustration below will continually repeat, advancing a step every 5 seconds. In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy.

The ALU team has developed a new "Gold" version of their module, with new features. The "Gold" version is intended to replace all earlier versions. The new "Gold" version is version 1.6.1.2 of the ALU module. The new "Gold" ALU version uses a private version of the RAM module. The private version of the RAM module is different than the general version of the RAM module used previously by both ALU and CPU (in the "Creating Module Hierarchy" use cases).



Step 1 of 5

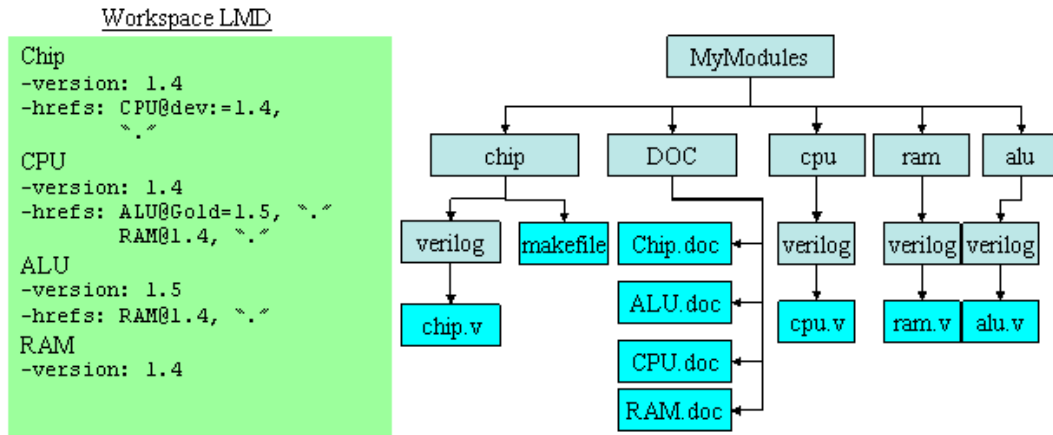
View the individual steps.

View the module structure used in this example.

View the subsequent example of modifying module hierarchy: Chip team members use the new ALU version

## Modifying Module Hierarchy: Chip Team Uses New ALU Version

The animated illustration below will continually repeat, advancing a step every 5 seconds. In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy.



Step 1 of 9

View the individual steps.

View the module structure used in this example.

View the subsequent example of modifying module hierarchy: The CPU team reverts to an earlier version of ALU.

## Modifying Module Hierarchy: CPU Team Reverts to Earlier ALU Version

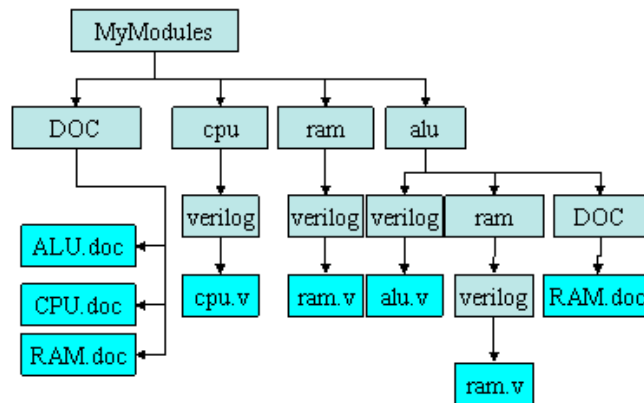
The animated illustration below will continually repeat, advancing a step every 5 seconds. In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy.

The CPU team performs tests with the "Gold" version of ALU. The CPU team decides that the new ALU features are not required, and come at the cost of a larger floor plan. The CPU team reverts to the "RelA" version of the ALU.

```

Workspace LMD
CPU
-version: 1.5
-hrefs: ALU@Gold=1.6.1.2, `.`
        RAM@1.4, `.`
RAM
-version: 1.4
-instname: RAM
-basedir: .
ALU
-version: 1.6.1.2
-hrefs: RAM@1.4.2.4, `./alu`
RAM
-version: 1.4.2.4
-instname: RAM%1
-basedir: ./alu

```



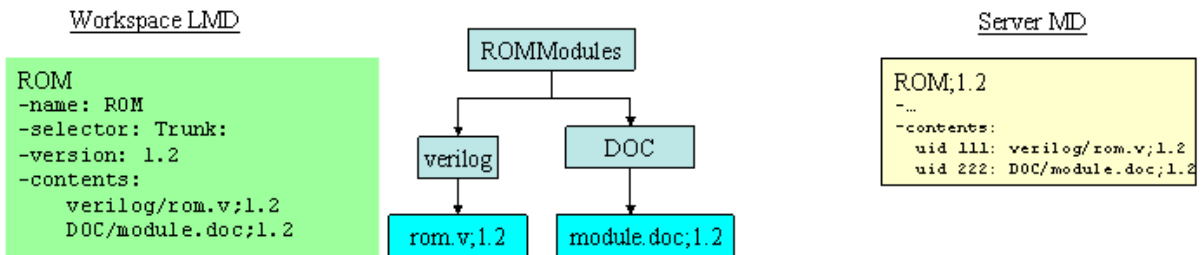
Step 1 of 6

View the individual steps.

View the module structure used in this example.

## Moving a File

The animated illustration below will continually repeat, advancing a step every 5 seconds. In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of directory versioning.



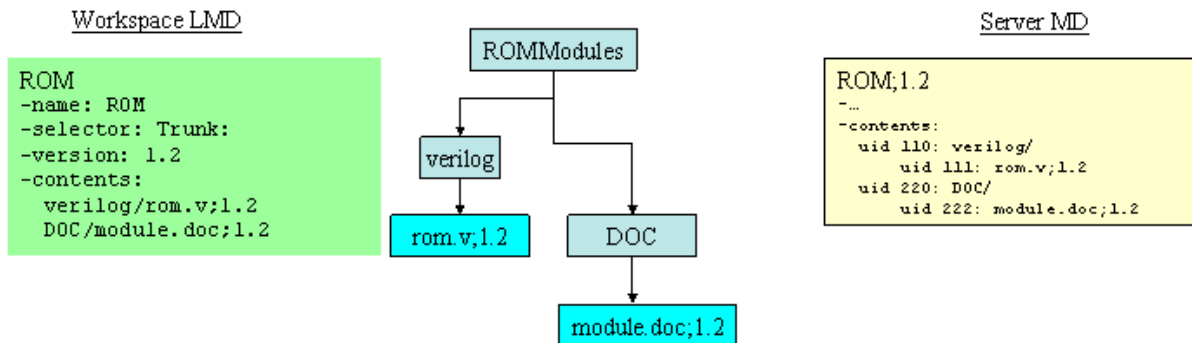
Step 1 of 6

View the individual steps.

View an animated example of moving a folder.

## Moving a Folder

The animated illustration below will continually repeat, advancing a step every 5 seconds. In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of directory versioning.



Step 1 of 6

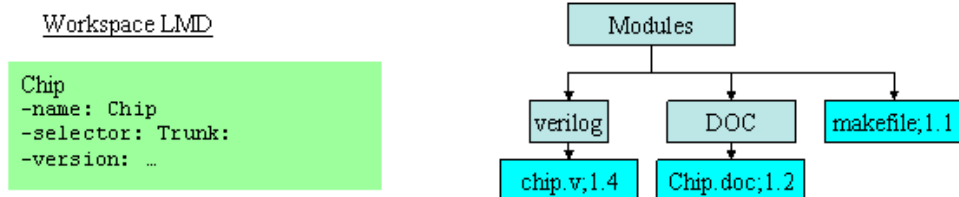
View the individual steps.

View an animated example of moving a file.

## Operating with Module Data

### Operating on a Module

The animated illustration below will continually repeat, advancing a step every 5 seconds. In this use case, "Workspace LMD" refers to the local metadata. Read an overview of operating on module data.



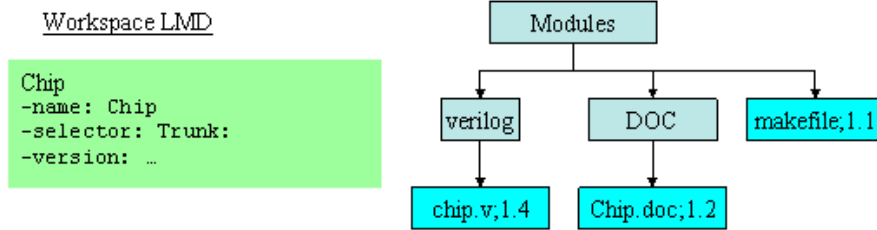
Step 1 of 8

View the individual steps.

View an animated example of operating on a module's contents.

## Operating on a Module's Contents

The animated illustration below will continually repeat, advancing a step every 5 seconds. In this use case, "Workspace LMD" refers to the local metadata. Read an overview of operating on module data.



Step 1 of 7

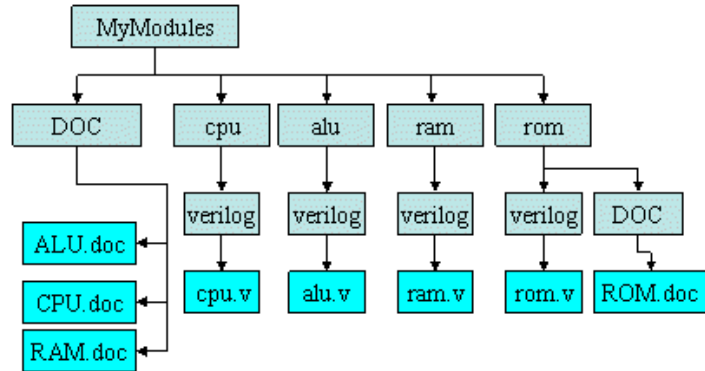
View the individual steps.

View an animated example of operating on a module.

## Filtering

The animated illustration below will continually repeat, advancing a step every 5 seconds. Read an overview of filtering module data.

Without any filtering, whole module hierarchy



Step 1 of 12

View the individual steps.

View an animated example of a persistent populate filter.

## Persistent Populate Filter

The animated illustration below will continually repeat, advancing a step every 5 seconds. In this use case, "Workspace LMD" refers to the local metadata. Read an overview of filtering module data.



MyModules

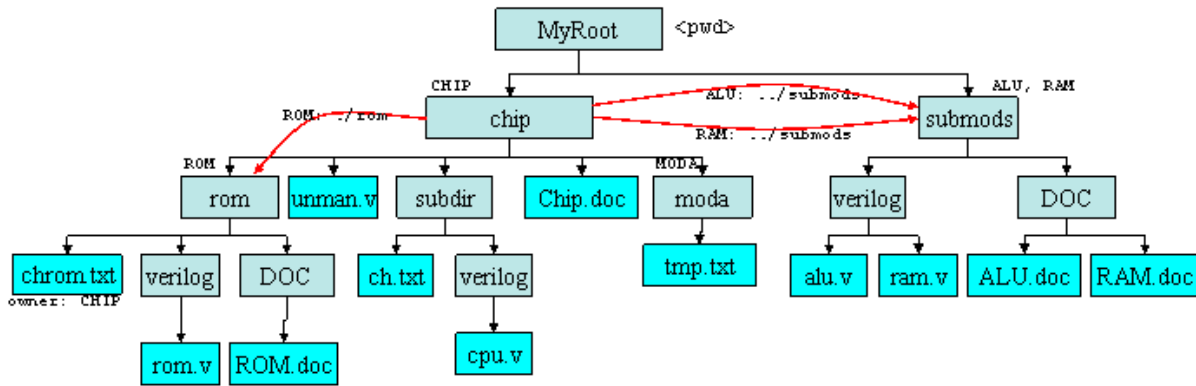
Step 1 of 7

View the individual steps.

View an animated example of filtering.

## Folder-Centric Operations

The animated illustration below will continually repeat, advancing a step every 5 seconds. Read an overview of module recursion.



Step 1 of 5

View the individual steps.

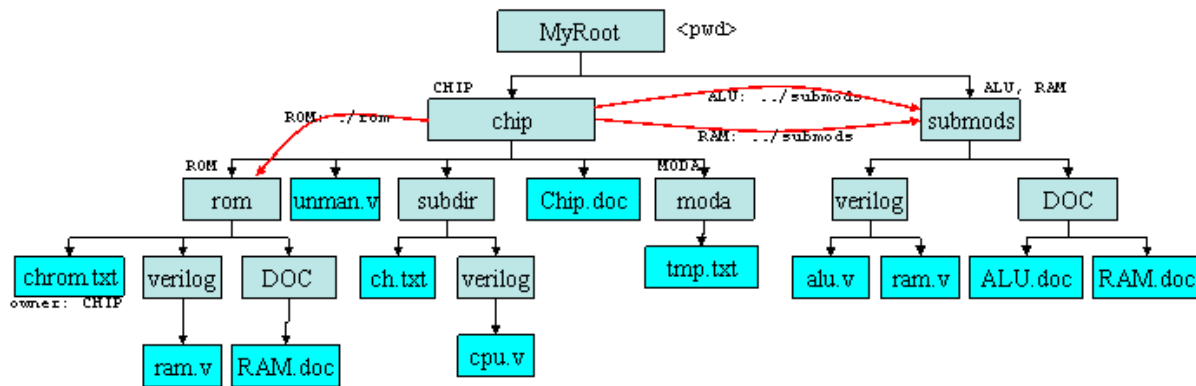
View an animated example of module-centric operations on a module.

View an animated example of module-centric operations on a sub-folder.

View an animated example of module-centric operations on hierarchical references.

## Module-Centric Operations on a Module

The animated illustration below will continually repeat, advancing a step every 5 seconds. Read an overview of module recursion.



Step 1 of 3

View the individual steps.

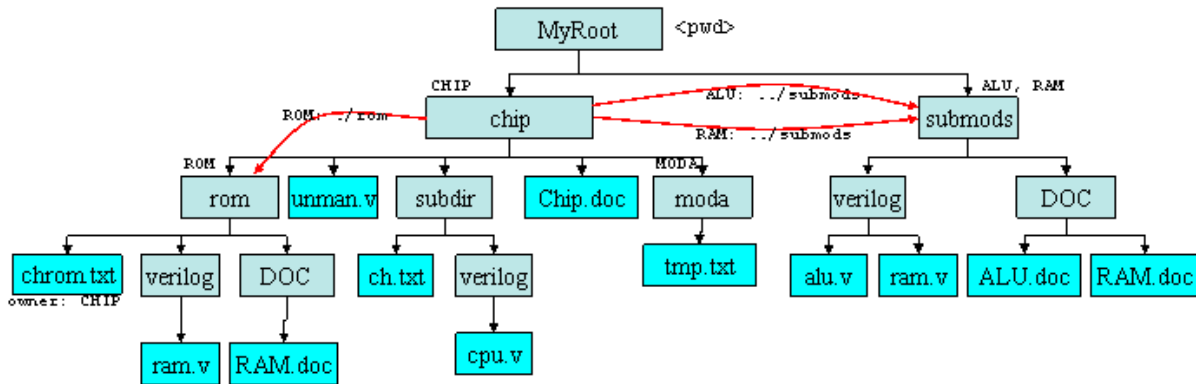
View an animated example of module-centric operations on a sub-folder.

View an animated example of module-centric operations on hierarchical references.

View an animated example of folder-centric operations.

## Module-Centric Operations on a Subfolder

The animated illustration below will continually repeat, advancing a step every 5 seconds. Read an overview of module recursion.



Step 1 of 5

View the individual steps.

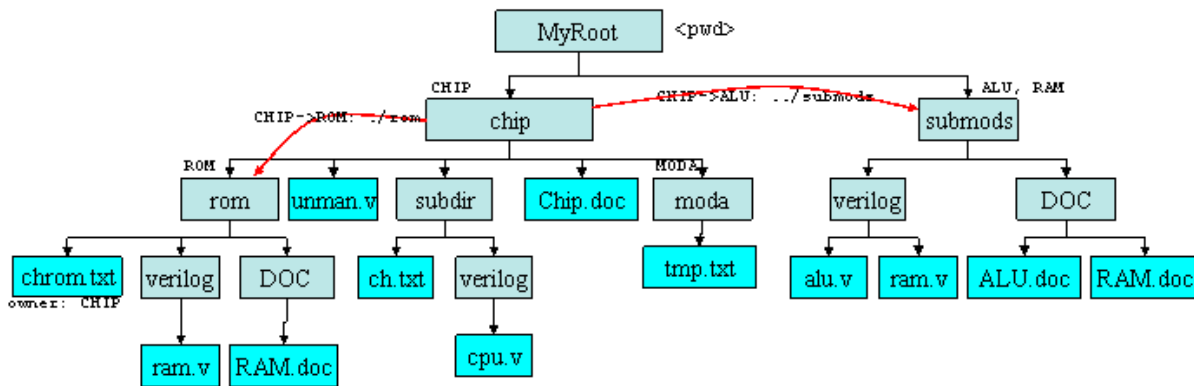
View an animated example of module-centric operations on a module.

View an animated example of module-centric operations on hierarchical references.

View an animated example of folder-centric operations.

## Module-Centric Operations on an HREF

The animated illustration below will continually repeat, advancing a step every 5 seconds. Read an overview of module recursion.



Step 1 of 5

View the individual steps.

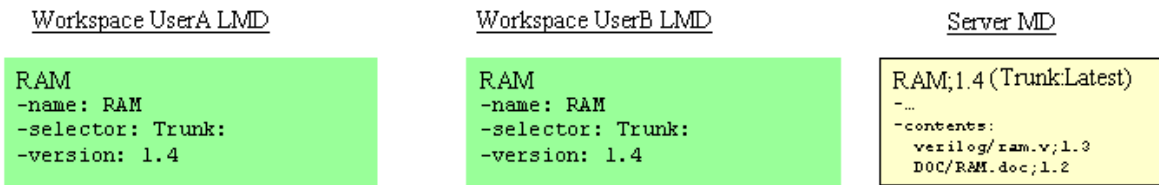
View an animated example of module-centric operations on a module.

View an animated example of folder-centric operations.

View an animated example of module-centric operations on a sub-folder.

## Locking a Module Branch

The animated illustration below will continually repeat, advancing a step every 5 seconds. In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module locking.



Step 1 of 7

View the individual steps.

View an animated example of locking module content.

## Locking Module Content

The animated illustration below will continually repeat, advancing a step every 5 seconds. In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module locking.

Workspace UserA LMD

```
RAM
-name: RAM
-selector: Trunk:
-version: 1.4
-content:
  verilog/ram.v;1.3
  DOC/RAM.doc;1.2
```

Workspace UserB LMD

```
RAM
-name: RAM
-selector: Trunk:
-version: 1.4
-content:
  verilog/ram.v;1.3
  DOC/RAM.doc;1.3
```

Server MD

```
RAM;1.4 (Trunk:Latest)
-contents:
  verilog/ram.v;1.3
  DOC/RAM.doc;1.2
```

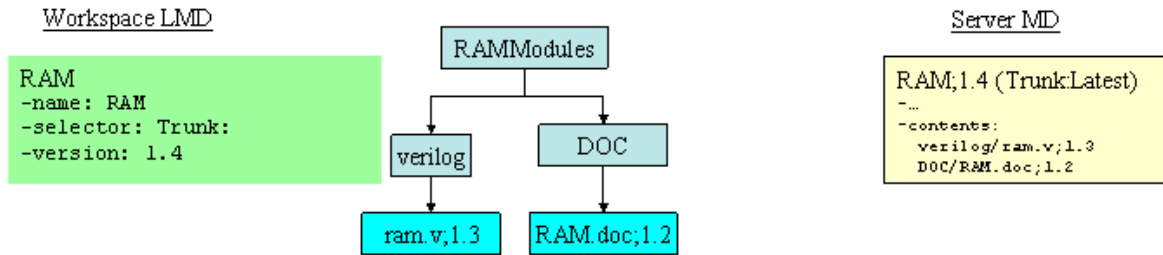
Step 1 of 12

View the individual steps.

View an animated example of locking a module branch.

## Branching a Module

The animated illustration below will continually repeat, advancing a step every 5 seconds. In this use case, the RAM team is producing a new version of the RAM module, with an "automatic undo" feature. The new version is created as a side-branch of version 1.4. "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module branching.



Step 1 of 6

View the individual steps.

## Merging and Modules

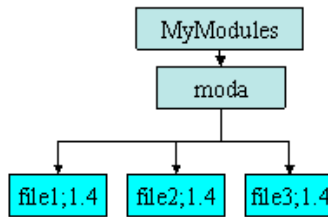
### Auto-Merging Locally Added Files

The animated illustration below will continually repeat, advancing a step every 5 seconds. In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read a description of this use case.



Workspace LMD

```
ModA
-name: ModA
-selector: Trunk:
-version: 1.4
-contents:
  file1;1.4
  file2;1.4
  file3;1.4
```

Server MD

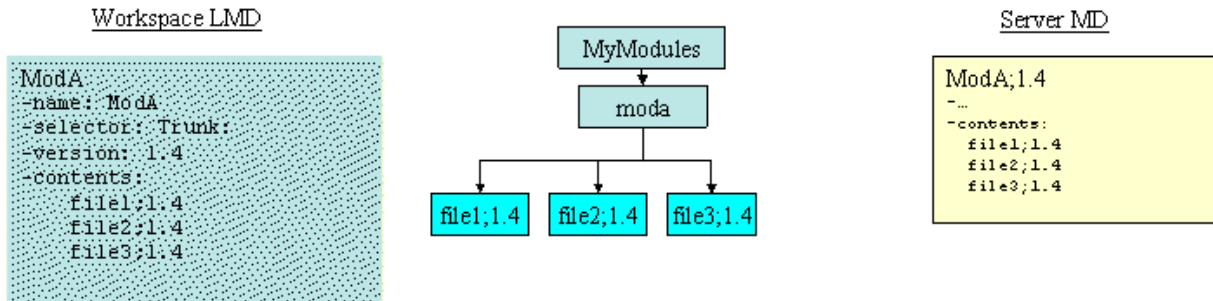
```
ModA;1.4
-...
-contents:
  file1;1.4
  file2;1.4
  file3;1.4
```

Step 1 of 4

View the individual steps.

## Auto-Merging Locally Modified Files

The animated illustration below will continually repeat, advancing a step every 5 seconds. In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read a description of this use case.



Step 1 of 6

View the individual steps.

## Auto-Merging Locally Modified Files Removed from the Module

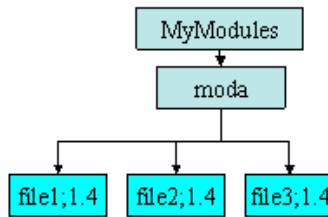
The animated illustration below will continually repeat, advancing a step every 5 seconds. In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read a description of this use case.

Workspace LMD

```

ModA
-name: ModA
-selector: Trunk:
-version: 1.4
-contents:
  file1;1.4
  file2;1.4
  file3;1.4

```

Server MD

```

ModA;1.4
-...
-contents:
  uid 111: file1;1.4
  uid 222: file2;1.4
  uid 333: file3;1.4

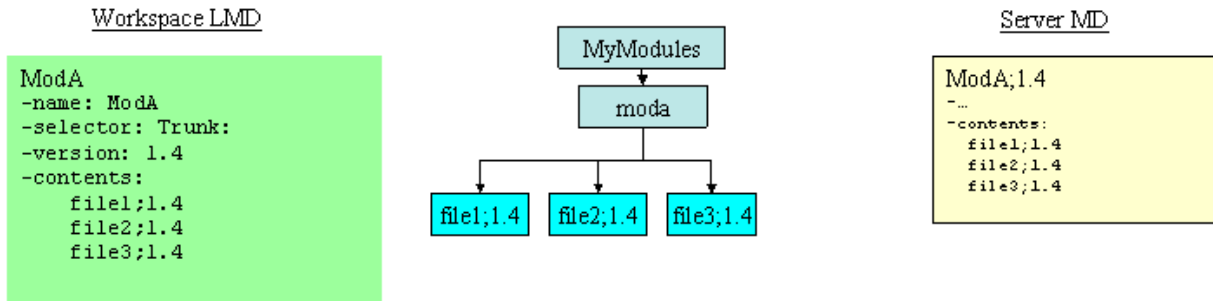
```

Step 1 of 5

View the individual steps.

## Auto-Merging Non-Latest Locally Modified Files

The animated illustration below will continually repeat, advancing a step every 5 seconds. In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read a description of this use case.



Step 1 of 4

View the individual steps.

### Auto-Merging Locally Modified Files Renamed in the Module

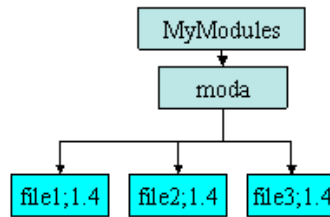
The animated illustration below will continually repeat, advancing a step every 5 seconds. In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read a description of this use case.

Workspace LMD

```

ModA
-name: ModA
-selector: Trunk:
-version: 1.4
-contents:
  uid 111: file1;1.4
  uid 222: file2;1.4
  uid 333: file3;1.4

```

Server MD

```

ModA;1.4
-...
-contents:
  uid 111: file1;1.4
  uid 222: file2;1.4
  uid 333: file3;1.4

```

Step 1 of 4

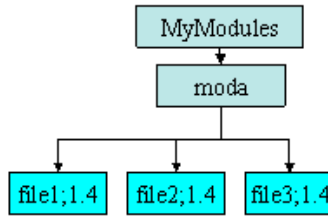
View the individual steps.

## Auto-Merging Locally Modified Files with Other Files Renamed in the Module

The animated illustration below will continually repeat, advancing a step every 5 seconds. In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read a description of this use case.

Workspace LMD

```
ModA
-name: ModA
-selector: Trunk:
-version: 1.4
-contents:
  uid 111: file1;1.4
  uid 222: file2;1.4
  uid 333: file3;1.4
```



Server MD

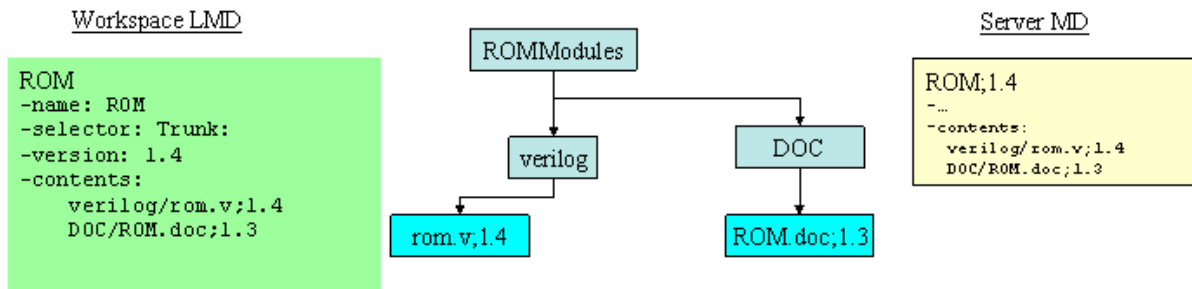
```
ModA;1.4
-...
-contents:
  uid 111: file1;1.4
  uid 222: file2;1.4
  uid 333: file3;1.4
```

Step 1 of 4

View the individual steps.

## In-Branch Merging of Locally Added Files

The animated illustration below will continually repeat, advancing a step every 5 seconds. In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read a description of this use case.

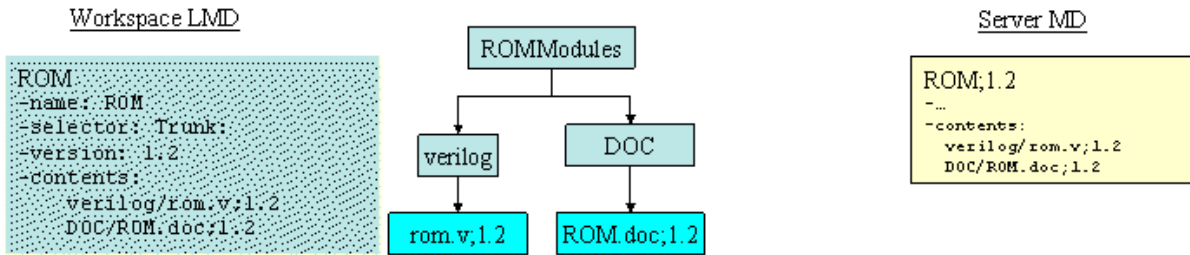


Step 1 of 6

View the individual steps.

## In-Branch Merging of Locally Modified Files

The animated illustration below will continually repeat, advancing a step every 5 seconds. In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read a description of this use case.



Step 1 of 6

View the individual steps.

## Step-by-Step Use Cases

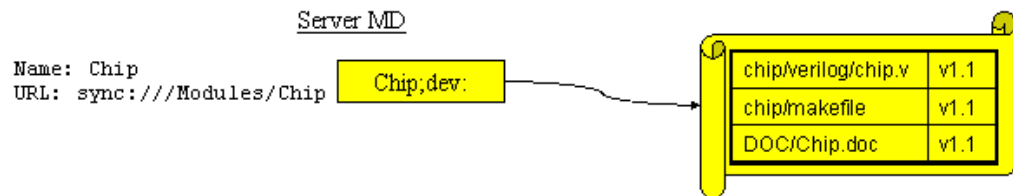
### Creating Modules and Module Data

#### Module Hierarchy: Module Structure

##### Step 1: Module Structure

In this use case, "Server MD" refers to the server metadata. Read an overview of module hierarchy.



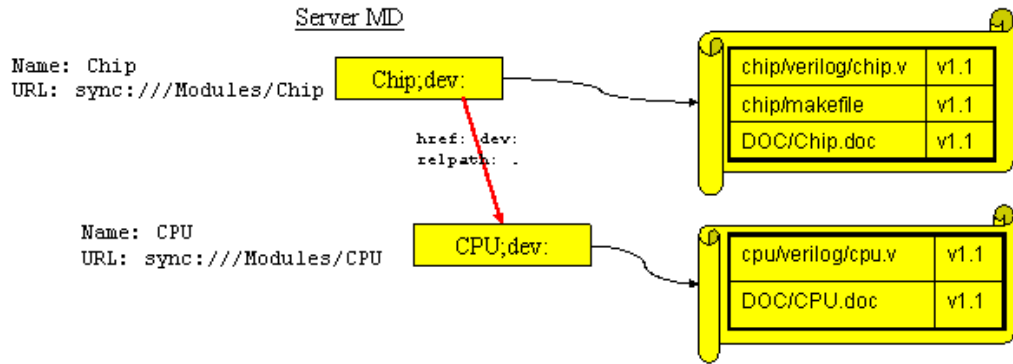


Step 1 of 4

View the next step.

### Step 2: Module Structure

In this use case, "Server MD" refers to the server metadata. Read an overview of module hierarchy.

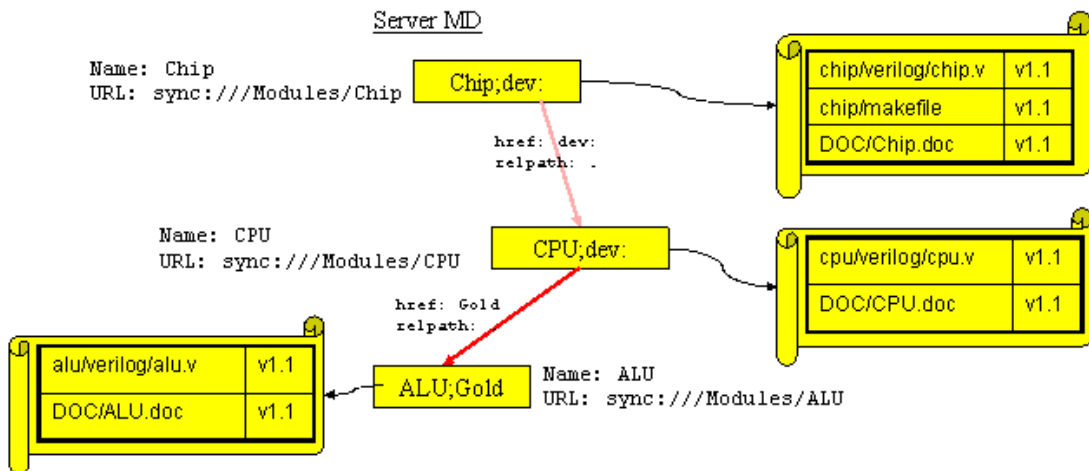


Step 2 of 4

View the next step.

### Step 3: Module Structure

In this use case, "Server MD" refers to the server metadata. Read an overview of module hierarchy.

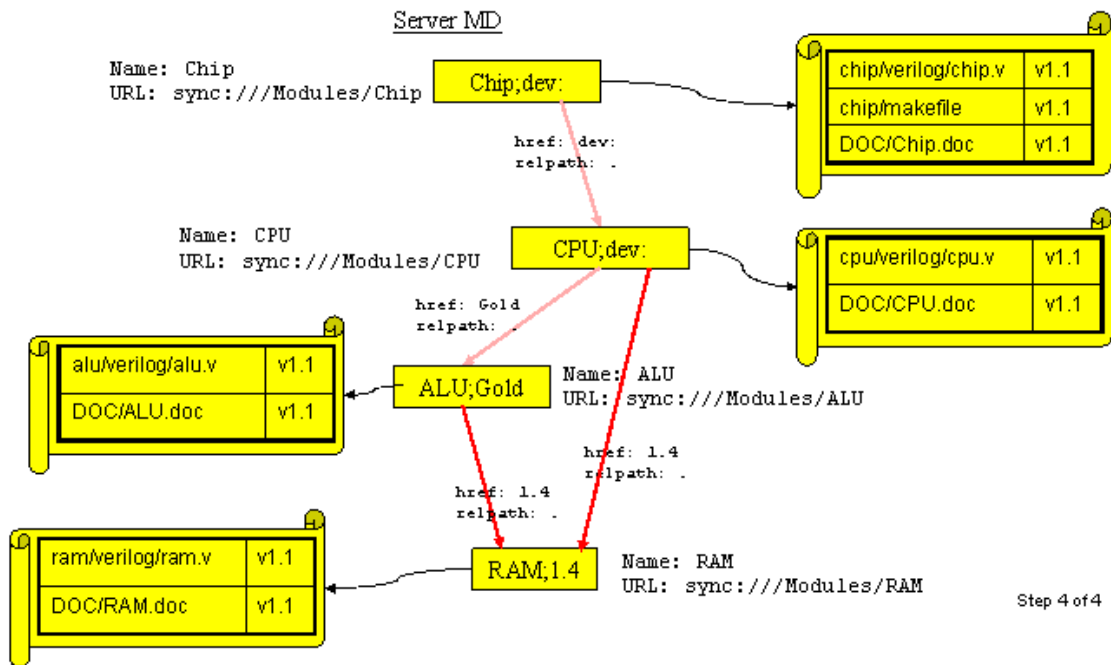


Step 3 of 4

View the next step.

**Step 4: Module Structure**

In this use case, "Server MD" refers to the server metadata. Read an overview of module hierarchy.



## Creating Module Hierarchy: Create the Module

### Step 1: Create the Module

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.

Workspace Directory Structure

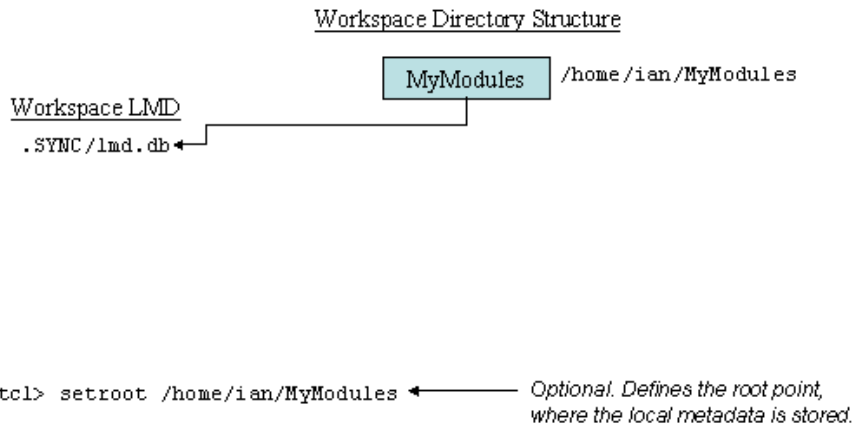
MyModules /home/ian/MyModules

Step 1 of 9

View the next step.

**Step 2: Create the Module**

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.

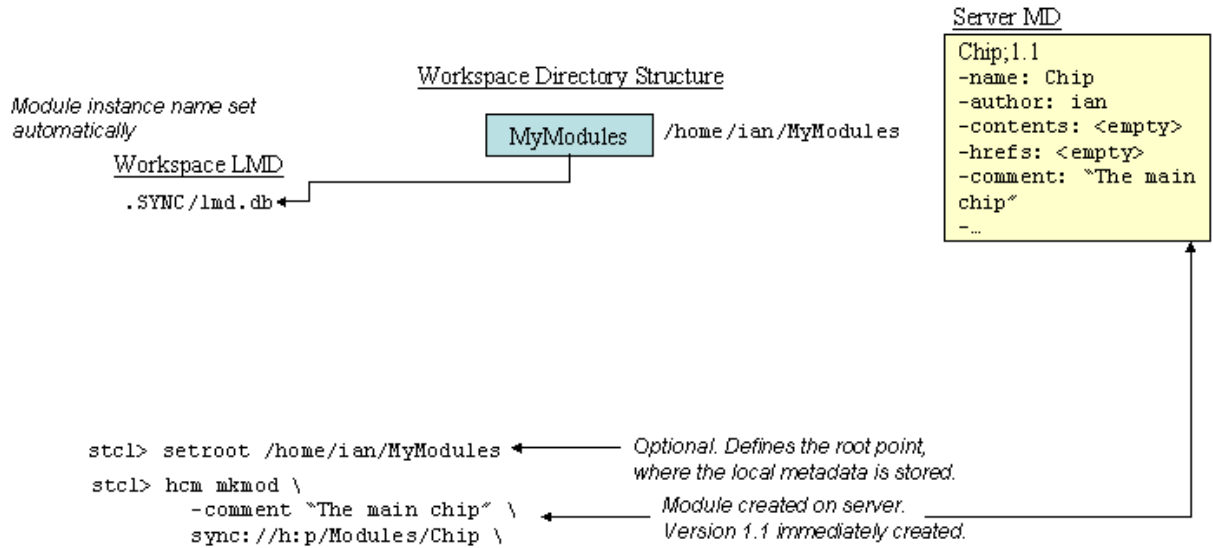


Step 2 of 9

View the next step.

### Step 3: Create the Module

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.

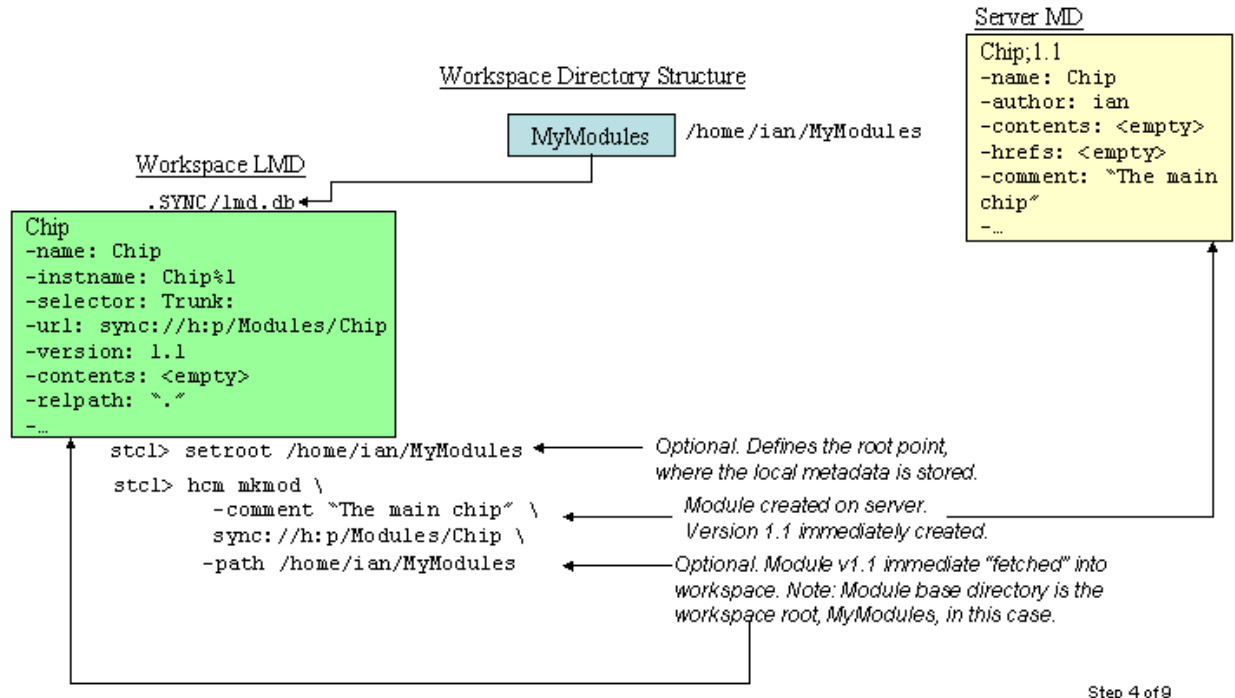


Step 3 of 9

View the next step.

#### Step 4: Create the Module

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.

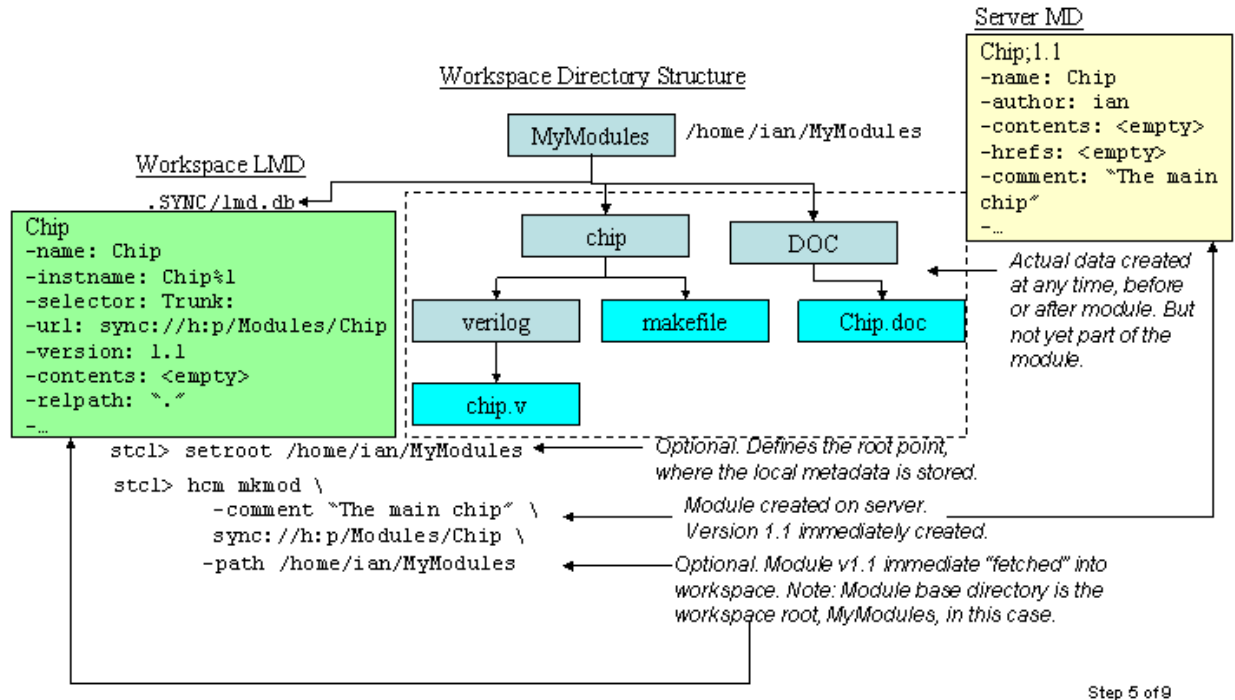


View the next step.

### Step 5: Create the Module

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.

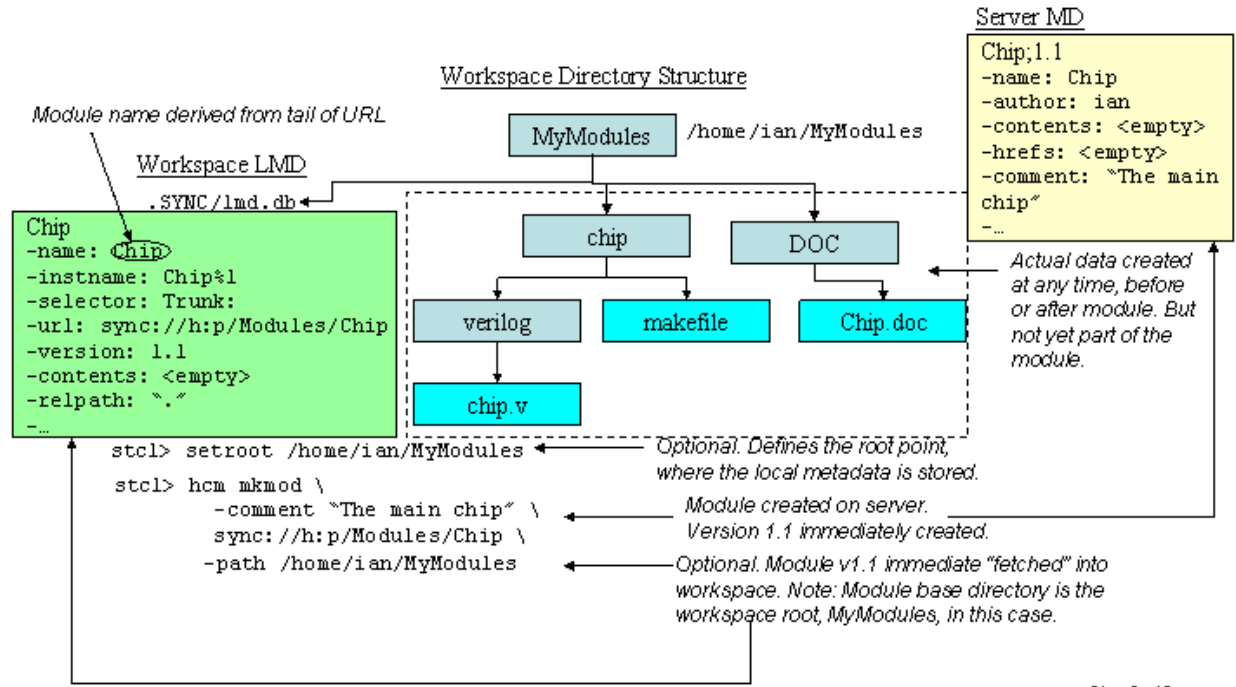




View the next step.

### Step 6: Create the Module

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.

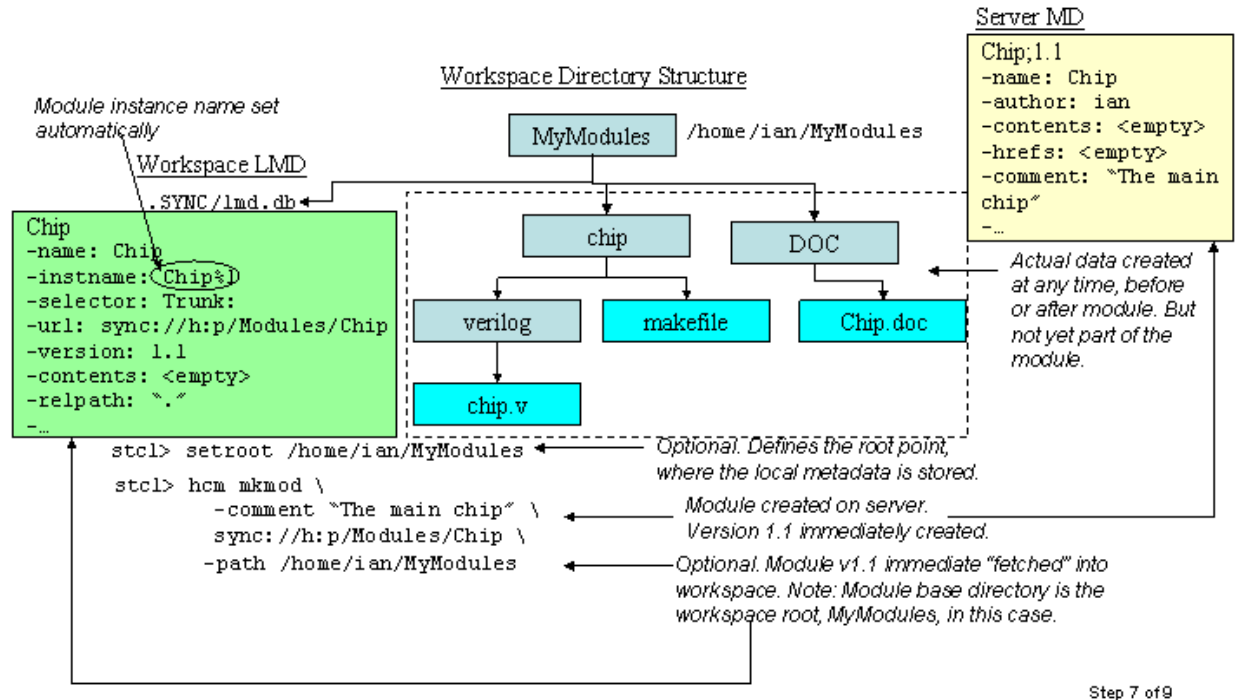


Step 6 of 9

View the next step.

### Step 7: Create the Module

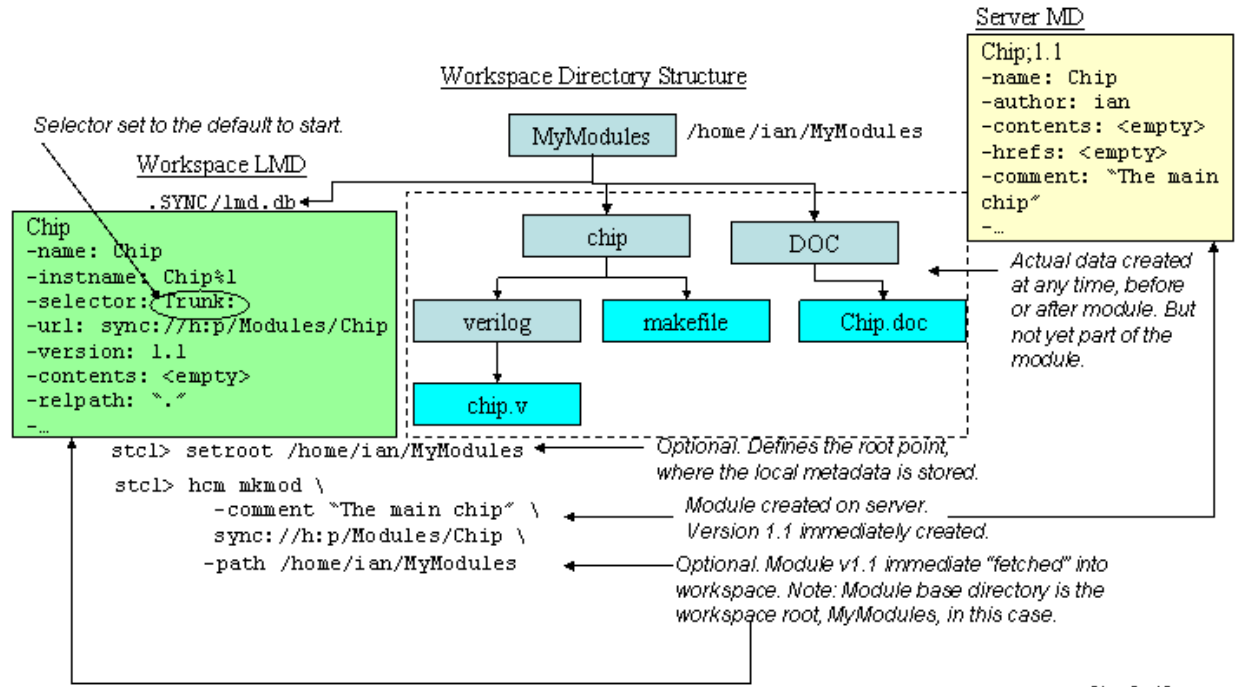
In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.



View the next step.

### Step 8: Create the Module

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.

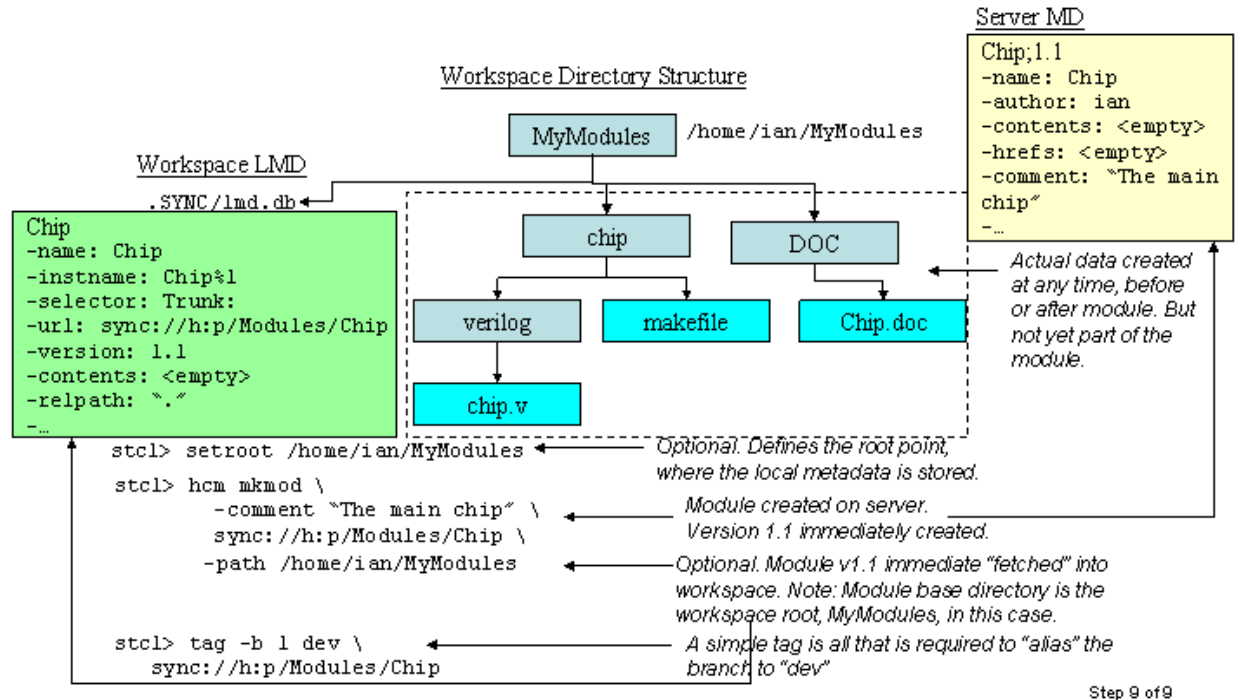


Step 8 of 9

View the next step.

### Step 9: Create the Module

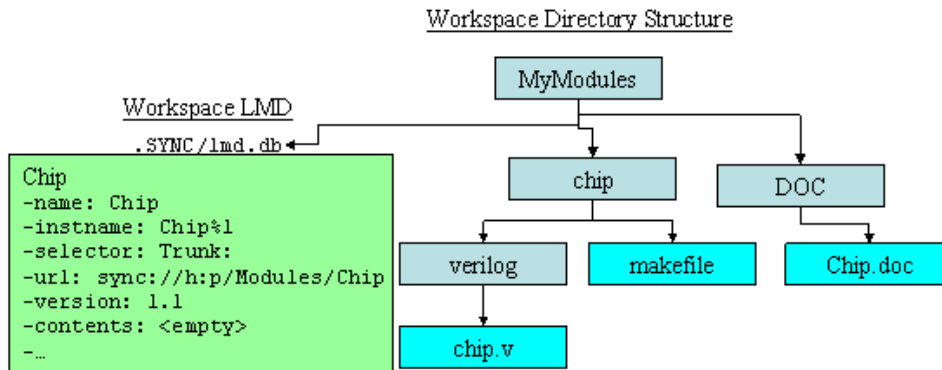
In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.



## Creating Module Hierarchy: Add Files and Check In

### Step 1: Add Files and Check In

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.

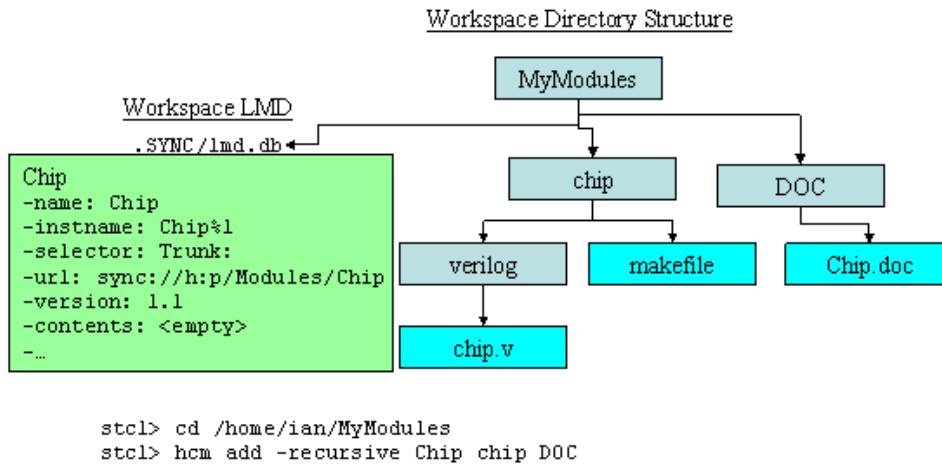


Step 1 of 9

View the next step.

**Step 2: Add Files and Check In**

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.

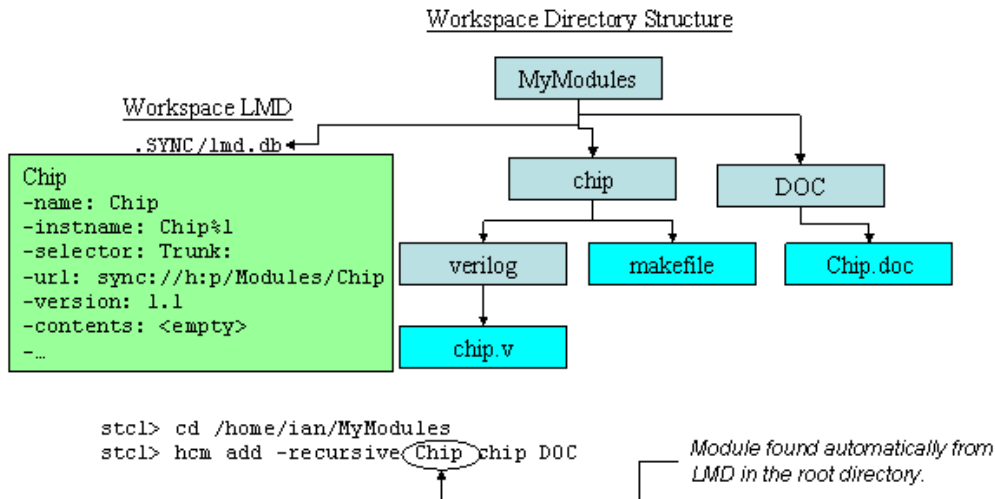


Step 2 of 9

View the next step.

### Step 3: Add Files and Check In

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.



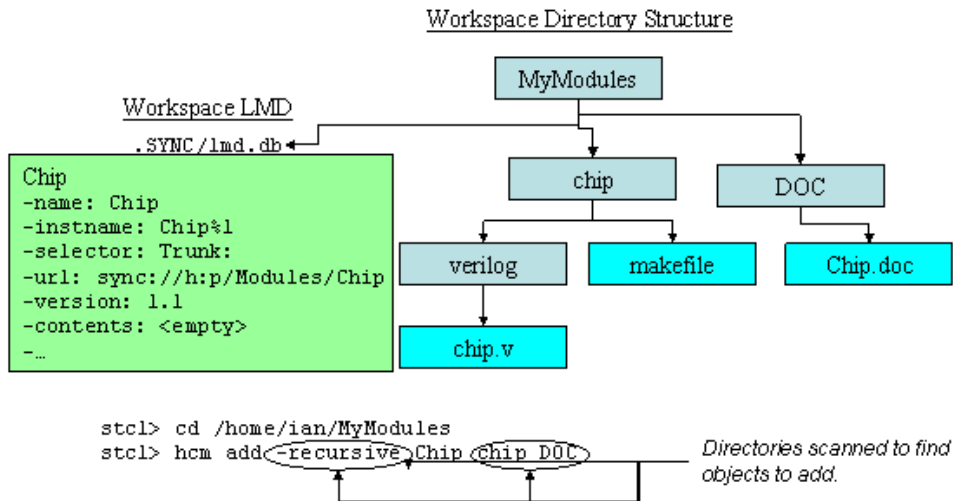
Step 3 of 9

View the next step.

#### Step 4: Add Files and Check In

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.



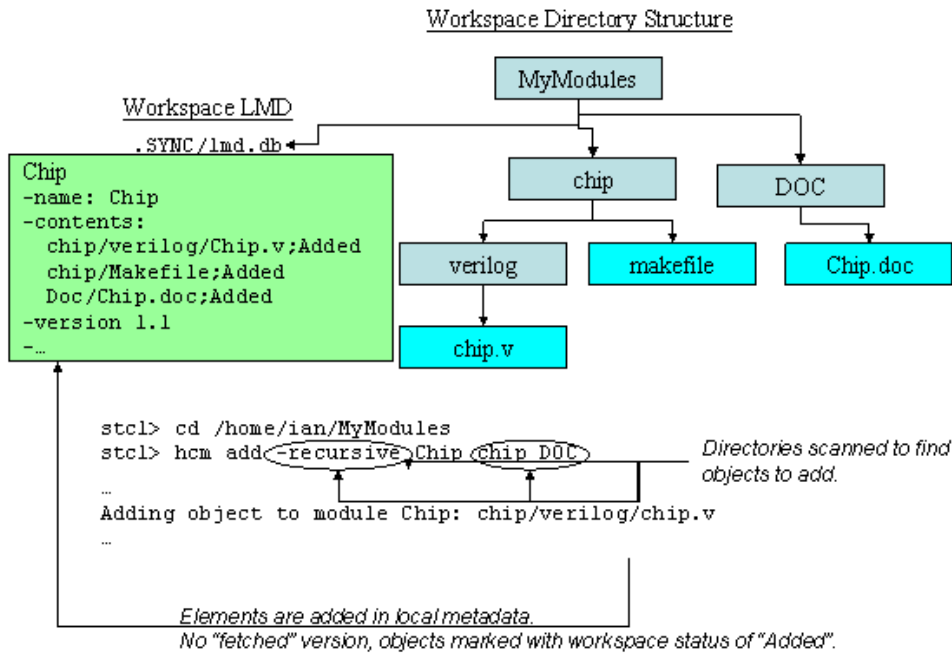


Step 4 of 9

View the next step.

### Step 5: Add Files and Check In

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.

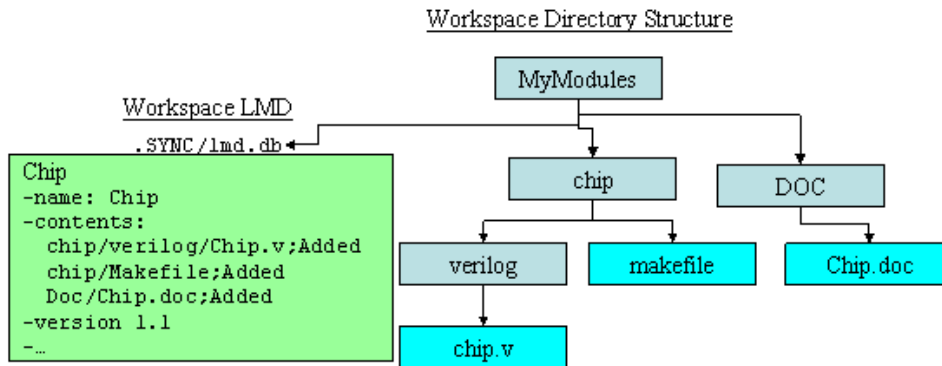


Step 5 of 9

View the next step.

### Step 6: Add Files and Check In

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.



```

stcl> cd /home/ian/MyModules
stcl> hcm add -recursive Chip chip DOC
...
Adding object to module Chip: chip/verilog/chip.v
...

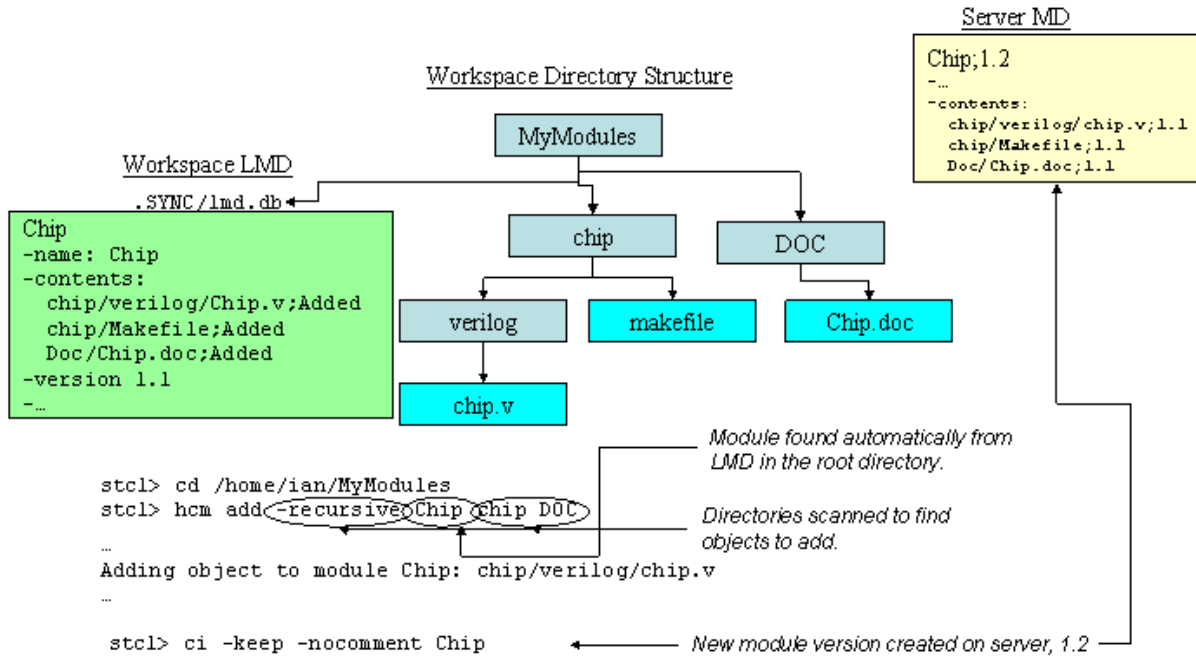
stcl> ci -keep -nocomment Chip
  
```

Step 6 of 9

View the next step.

### Step 7: Add Files and Check In

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.

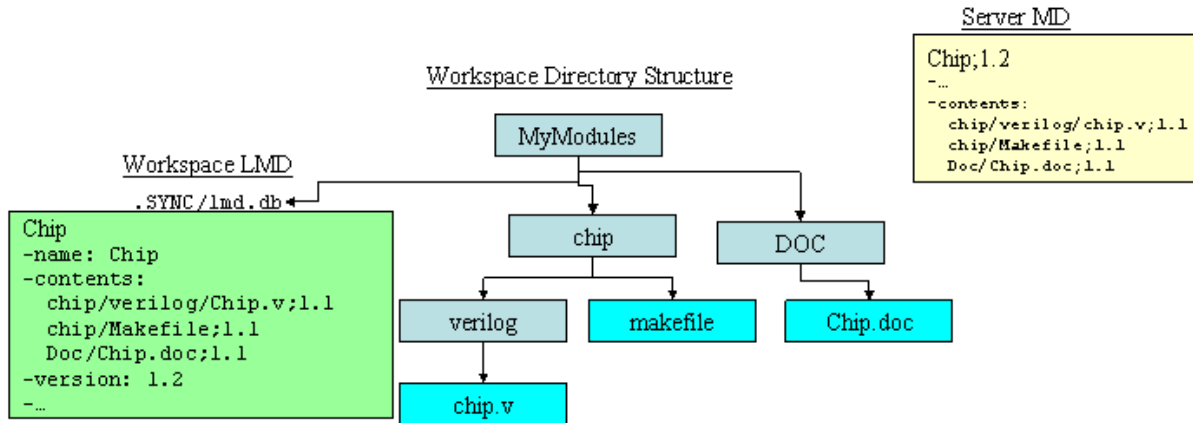


Step 7 of 9

View the next step.

### Step 8: Add Files and Check In

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.



```

stcl> cd /home/ian/MyModules
stcl> hcm add -recursive Chip chip DOC
...
Adding object to module Chip: chip/verilog/chip.v
...

stcl> ci -keep -nocomment Chip
Local version updated, and member versions now set

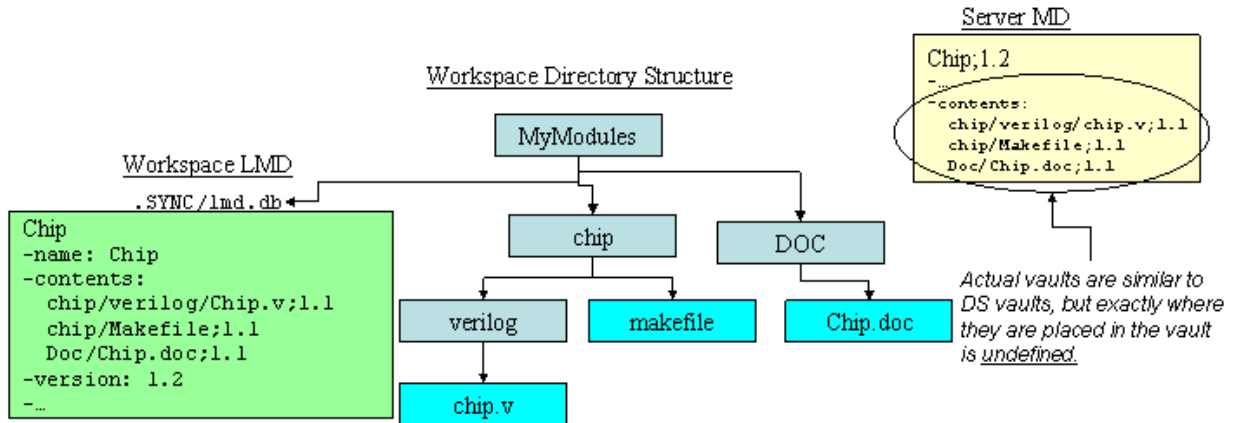
```

Step 8 of 9

View the next step.

### Step 9: Add Files and Check In

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.



```

stcl> cd /home/ian/MyModules
stcl> hcm add -recursive Chip chip DOC
...
Adding object to module Chip: chip/verilog/chip.v
...

stcl> ci -keep -nocomment Chip
  
```

Step 9 of 9

## Creating Module Hierarchy: Add an HREF to a Module in the Workspace

### Step 1: Add an HREF to a Module in the Workspace

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.

Workspace LMD

MyModules

```
Fetch the "Chip" module:  
stcl> cd /home/ian/MyModules  
stcl> populate -get sync://h:p/Modules/Chip;dev:Latest
```

Step 1 of 8

View the next step.

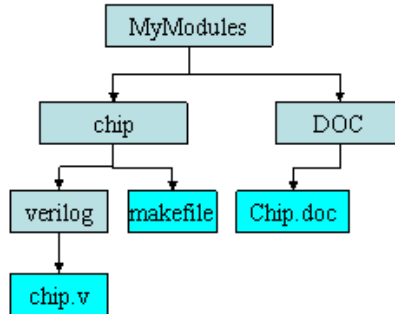
### Step 2: Add an HREF to a Module in the Workspace

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.

# DesignSync Data Manager User's Guide

## Workspace LMD

```
Chip
-name: Chip
-version: 1.2
-contents:
  chip/verilog/Chip.v;1.1
  chip/Makefile;1.1
  Doc/Chip.doc;1.1
-hrefs: <empty>
```



Fetch the "Chip" module:

```
stcl> cd /home/ian/MyModules
stcl> populate -get sync://h:p/Modules/Chip;dev:Latest
```

Step 2 of 8

View the next step.

### Step 3: Add an HREF to a Module in the Workspace

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.

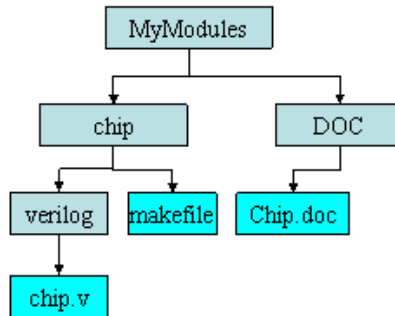


Workspace LMD

```

Chip
-name: Chip
-version: 1.2
-contents:
  chip/verilog/Chip.v;1.1
  chip/Makefile;1.1
  Doc/Chip.doc;1.1
-hrefs: <empty>

```



```

Fetch the "Chip" module:
stcl> cd /home/ian/MyModules
stcl> populate -get sync://h:p/Modules/Chip;dev:Latest
Fetch the "CPU" module:
stcl> populate -get sync://h:p/Modules/CPU;dev:Latest

```

Step 3 of 8

View the next step.

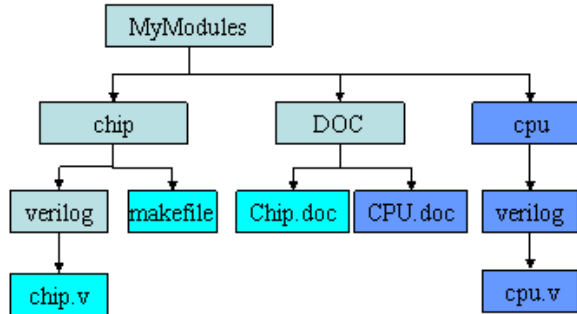
**Step 4: Add an HREF to a Module in the Workspace**

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.

## Workspace LMD

```
Chip
-name: Chip
-version: 1.2
-contents:
  chip/verilog/Chip.v;1.1
  chip/Makefile;1.1
  Doc/Chip.doc;1.1
-hrefs: <empty>
```

```
CPU
-name: CPU
-version: 1.2
-contents:
  cpu/verilog/CPU.v;1.1
  Doc/Chip.doc;1.1
```



```
Fetch the "Chip" module:
stcl> cd /home/ian/MyModules
stcl> populate -get sync://h:p/Modules/Chip;dev:Latest
Fetch the "CPU" module:
stcl> populate -get sync://h:p/Modules/CPU;dev:Latest
```

Step 4 of 8

View the next step.

### Step 5: Add an HREF to a Module in the Workspace

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.

## Workspace LMD

```

Chip
-name: Chip
-version: 1.2
-contents:
  chip/verilog/Chip.v;1.1
  chip/Makefile;1.1
  Doc/Chip.doc;1.1
-hrefs: <empty>

```

```

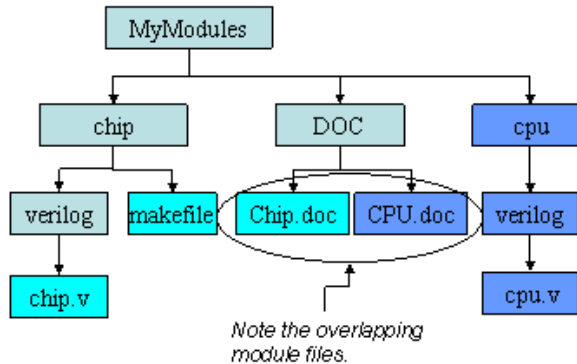
CPU
-name: CPU
-version: 1.2
-contents:
  cpu/verilog/CPU.v;1.1
  Doc/Chip.doc;1.1

```

```

Fetch the "Chip" module:
stcl> cd /home/ian/MyModules
stcl> populate -get sync://h:p/Modules/Chip;dev:Latest
Fetch the "CPU" module:
stcl> populate -get sync://h:p/Modules/CPU;dev:Latest

```



Step 5 of 8

View the next step.

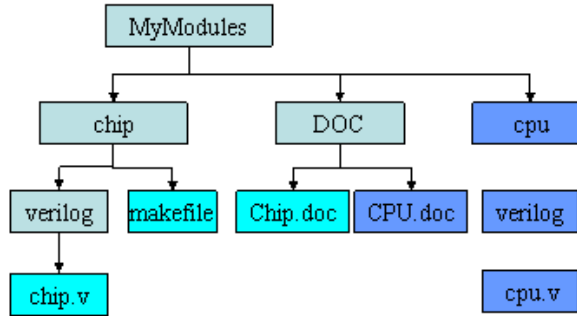
### Step 6: Add an HREF to a Module in the Workspace

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.

## Workspace LMD

```
Chip
-name: Chip
-version: 1.2
-contents:
  chip/verilog/Chip.v;1.1
  chip/Makefile;1.1
  Doc/Chip.doc;1.1
-hrefs:
```

```
CPU
-name: CPU
-version: 1.2
-contents:
  cpu/verilog/CPU.v;1.1
  Doc/Chip.doc;1.1
```



Fetch the "Chip" module:

```
stcl> cd /home/ian/MyModules
```

```
stcl> populate -get sync: //h:p/Modules/Chip;dev:Latest
```

Fetch the "CPU" module:

```
stcl> populate -get sync: //h:p/Modules/CPU;dev:Latest
```

Add href from Chip to CPU:

```
stcl> hcm addhref Chip CPU
```

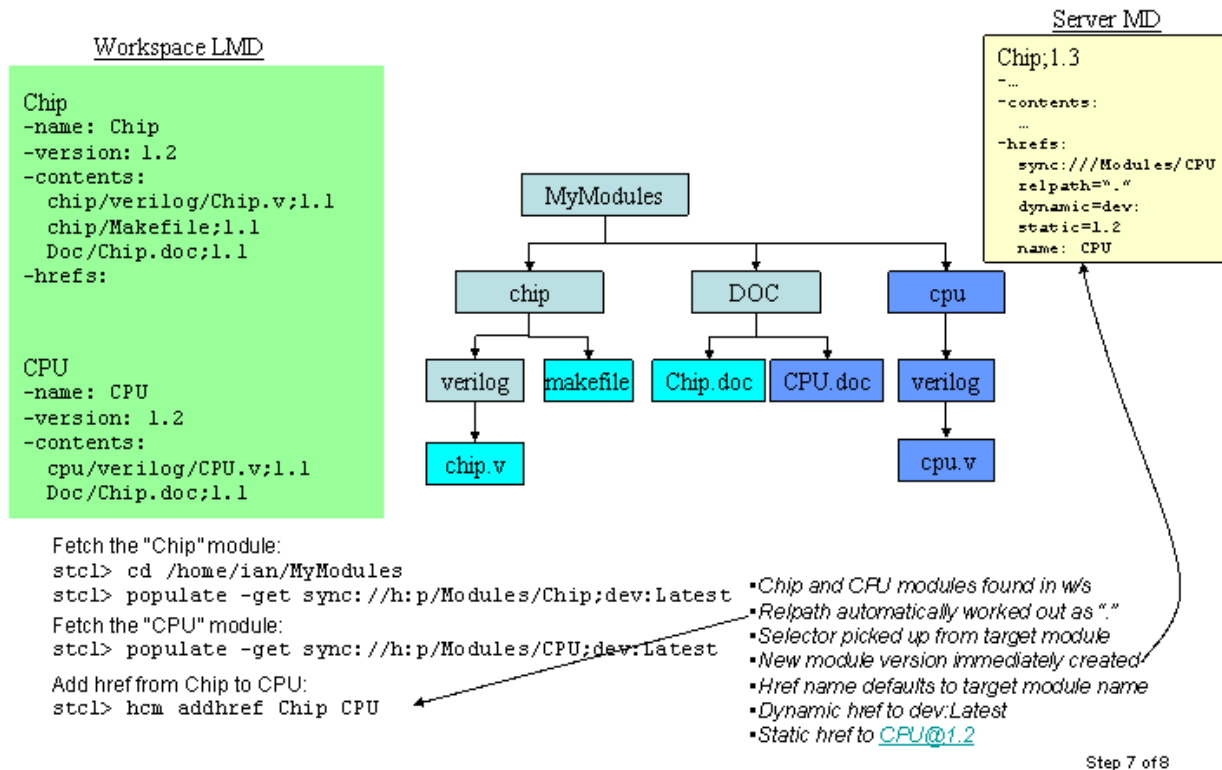
- Chip and CPU modules found in w/s
- Relpath automatically worked out as "."
- Selector picked up from target module

Step 6 of 8

View the next step.

### Step 7: Add an HREF to a Module in the Workspace

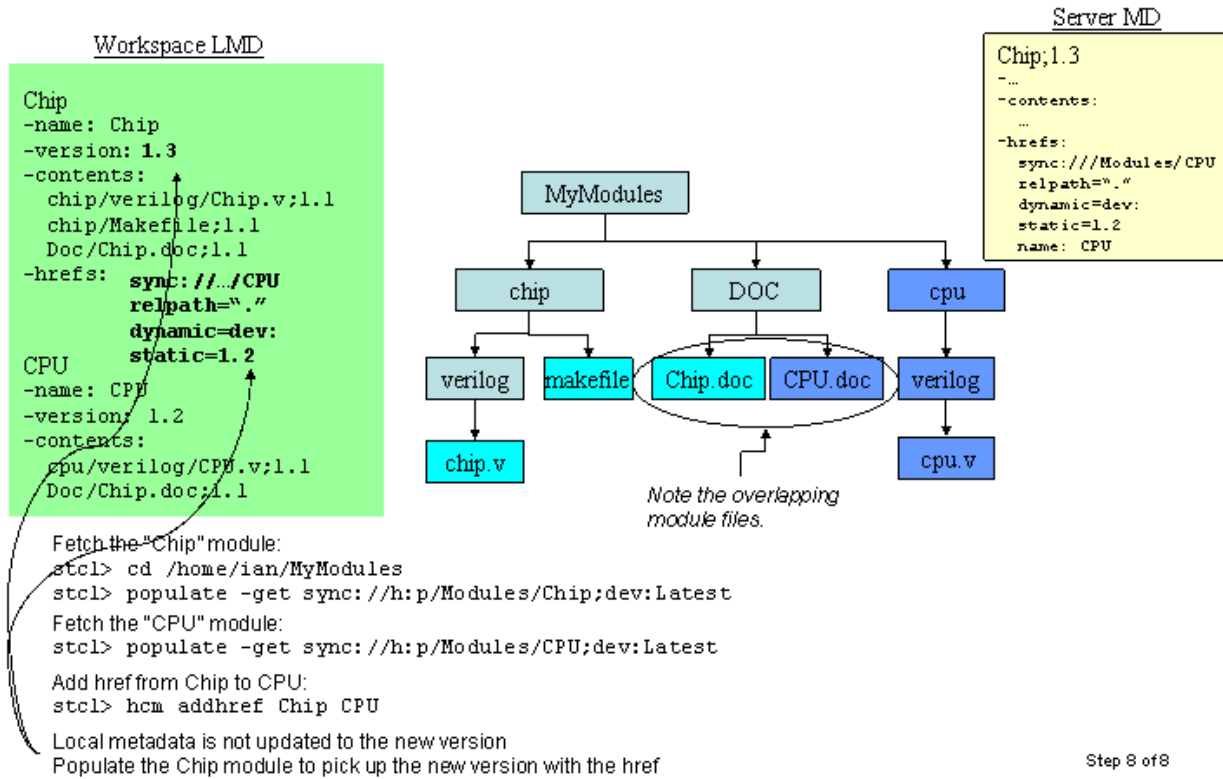
In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.



View the next step.

### Step 8: Add an HREF to a Module in the Workspace

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.



## Creating Module Hierarchy: Populate with Dynamic HREF Mode

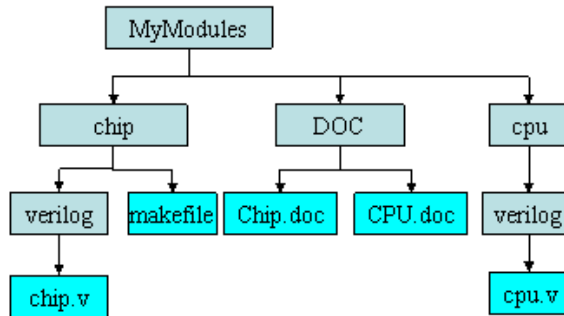
### Step 1: Populate with Dynamic HREF Mode

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.

Workspace LMD

```
Chip
-name: Chip
-version: 1.3
```

```
CPU
-name: CPU
-version: 1.2
-hrefs:
```

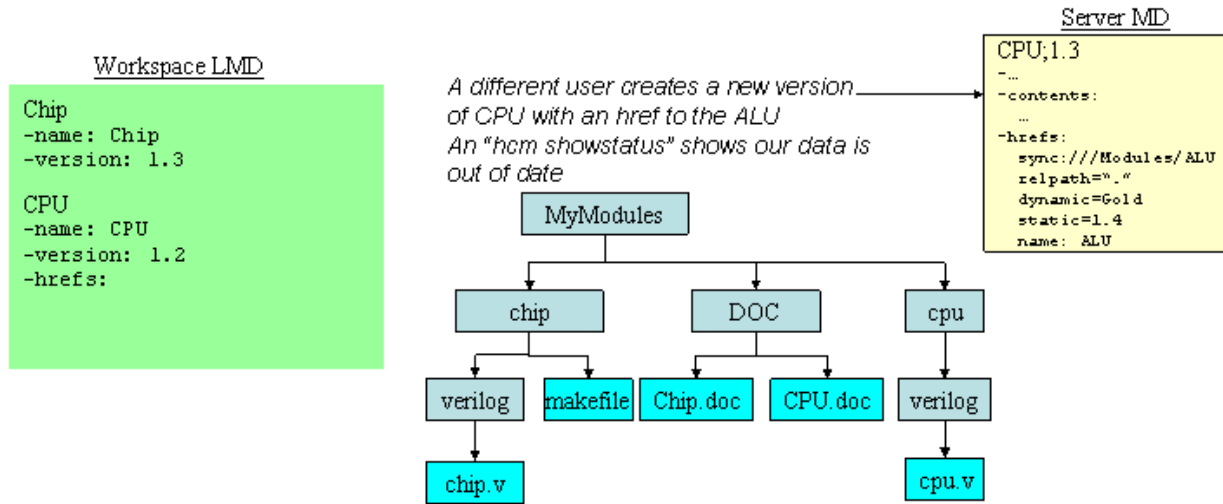


Step 1 of 7

View the next step.

### Step 2: Populate with Dynamic HREF Mode

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.



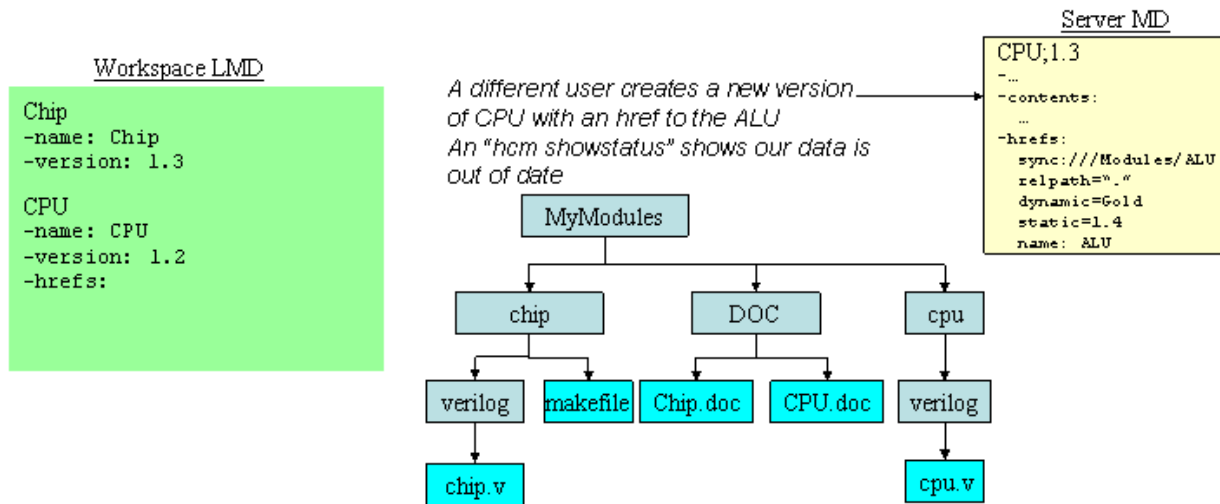
Step 2 of 7

View the next step.

### Step 3: Populate with Dynamic HREF Mode

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.





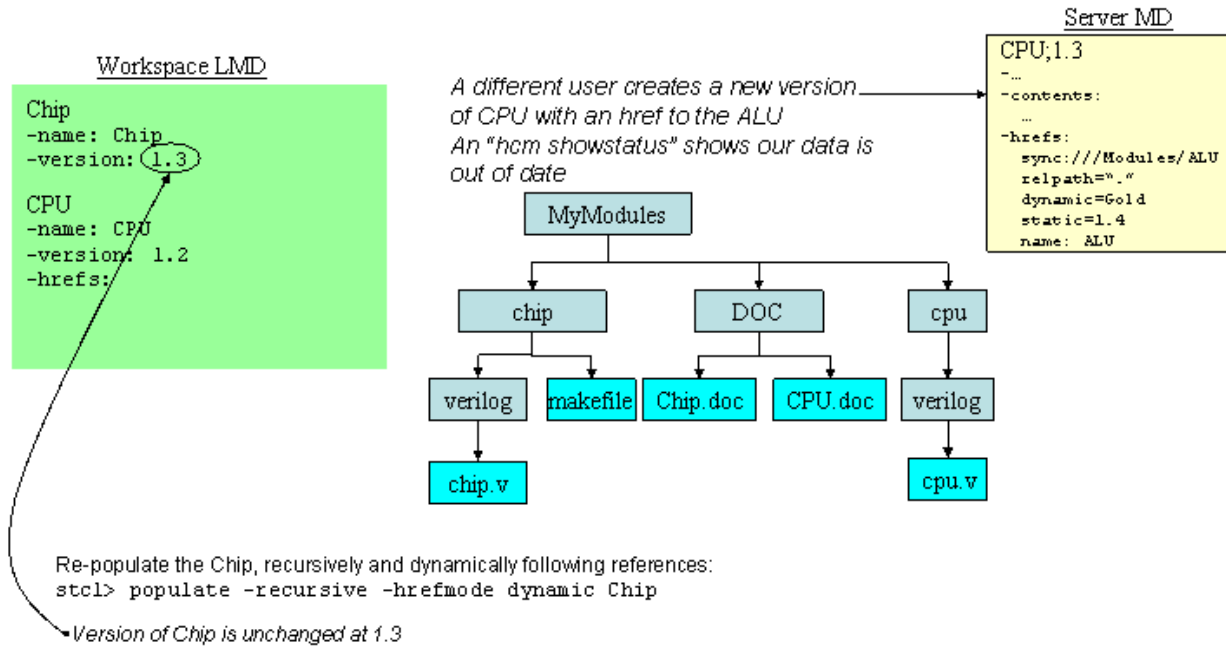
Re-populate the Chip, recursively and dynamically following references:  
 stcl> populate -recursive -hrefmode dynamic Chip

Step 3 of 7

View the next step.

#### Step 4: Populate with Dynamic HREF Mode

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.

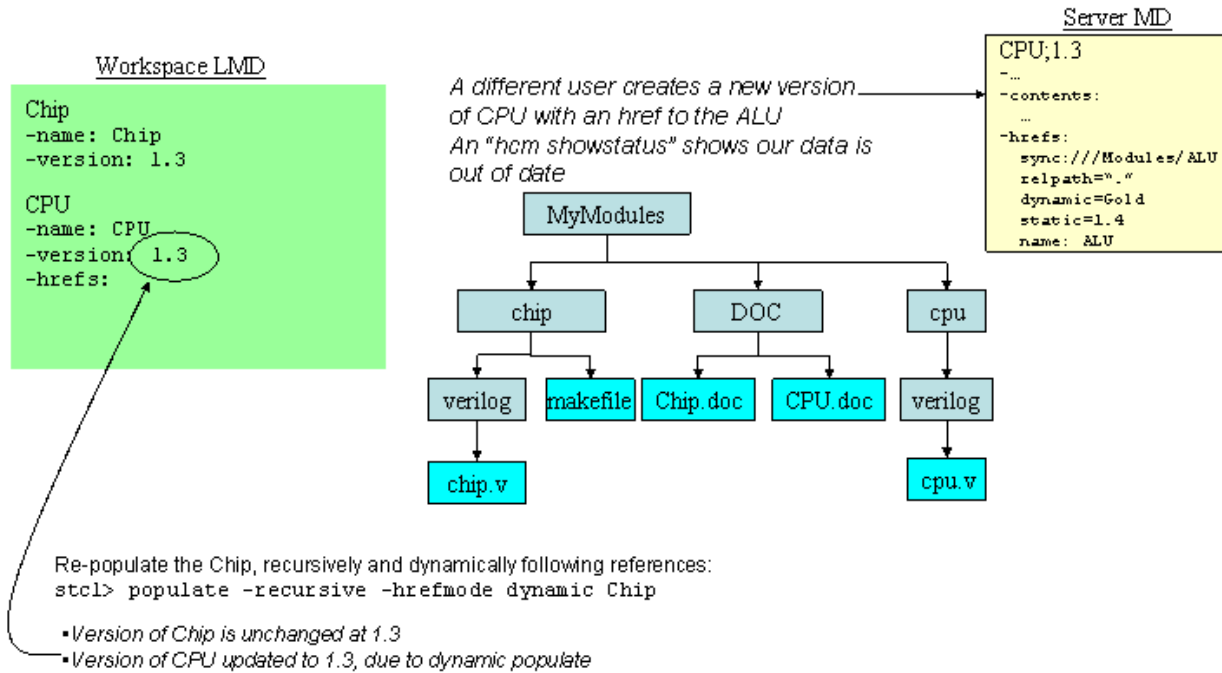


Step 4 of 7

View the next step.

### Step 5: Populate with Dynamic HREF Mode

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.

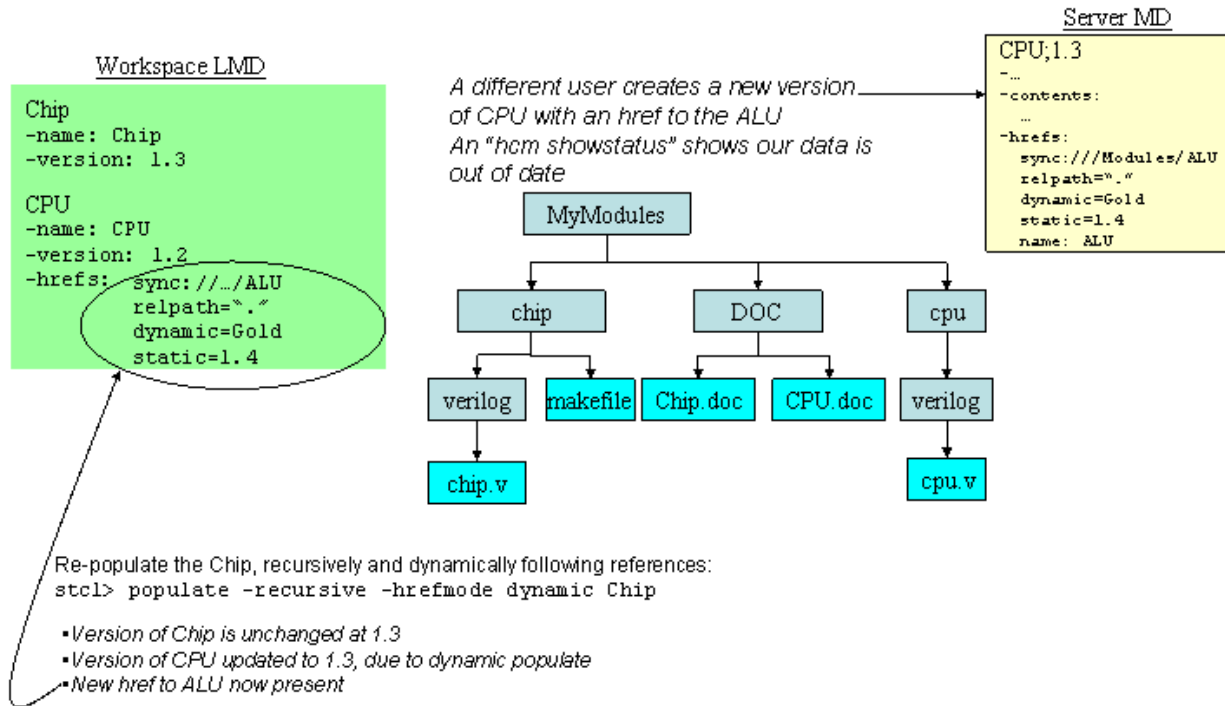


Step 5 of 7

View the next step.

### Step 6: Populate with Dynamic HREF Mode

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.

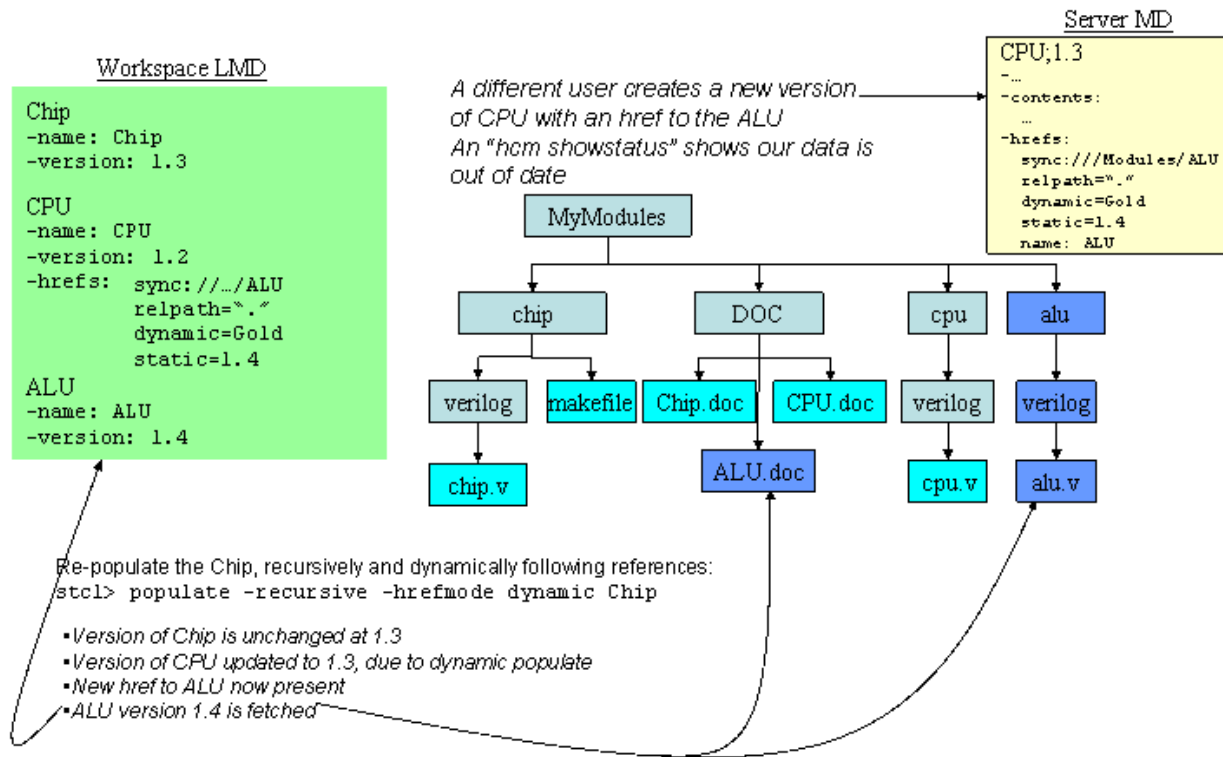


Step 6 of 7

View the next step.

### Step 7: Populate with Dynamic HREF Mode

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.



Step 7 of 7

## Creating Module Hierarchy: Add an HREF to a Module not in the Workspace

### Step 1: Add an HREF to a Module not in the Workspace

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.

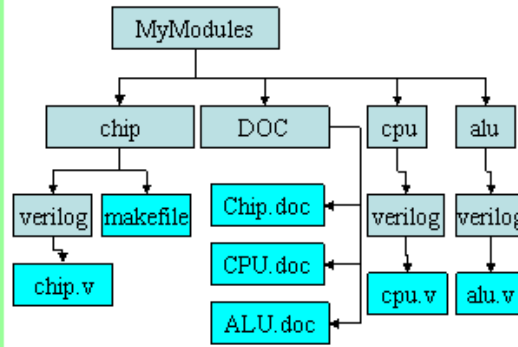
# DesignSync Data Manager User's Guide

## Workspace LMD

```
Chip  
-name: Chip  
-version: 1.3
```

```
CPU  
-name: CPU  
-version: 1.3  
-hrefs: sync://.../ALU  
...
```

```
ALU  
-name: ALU  
-version: 1.4  
-hrefs:
```



Step 1 of 5

View the next step.

### Step 2: Add an HREF to a Module not in the Workspace

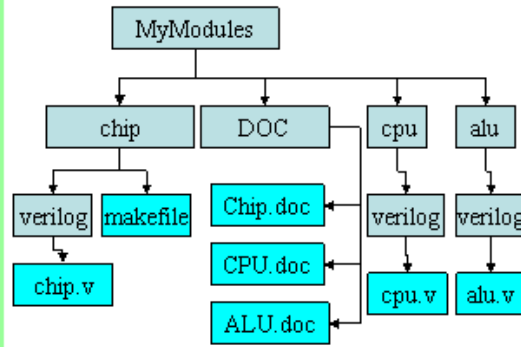
In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.

Workspace LMD

```
Chip
-name: Chip
-version: 1.3
```

```
CPU
-name: CPU
-version: 1.3
-hrefs: sync://.../ALU
...
```

```
ALU
-name: ALU
-version: 1.4
-hrefs:
```



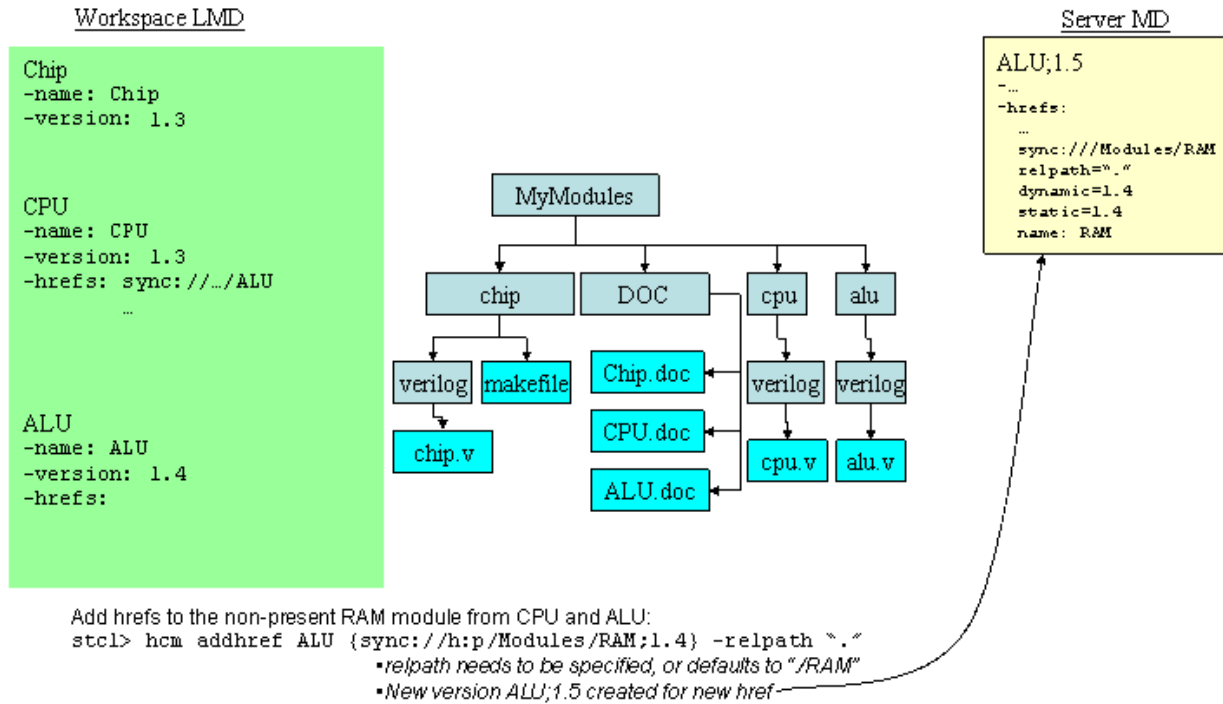
Add hrefs to the non-present RAM module from CPU and ALU:

Step 2 of 5

View the next step.

### Step 3: Add an HREF to a Module not in the Workspace

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.



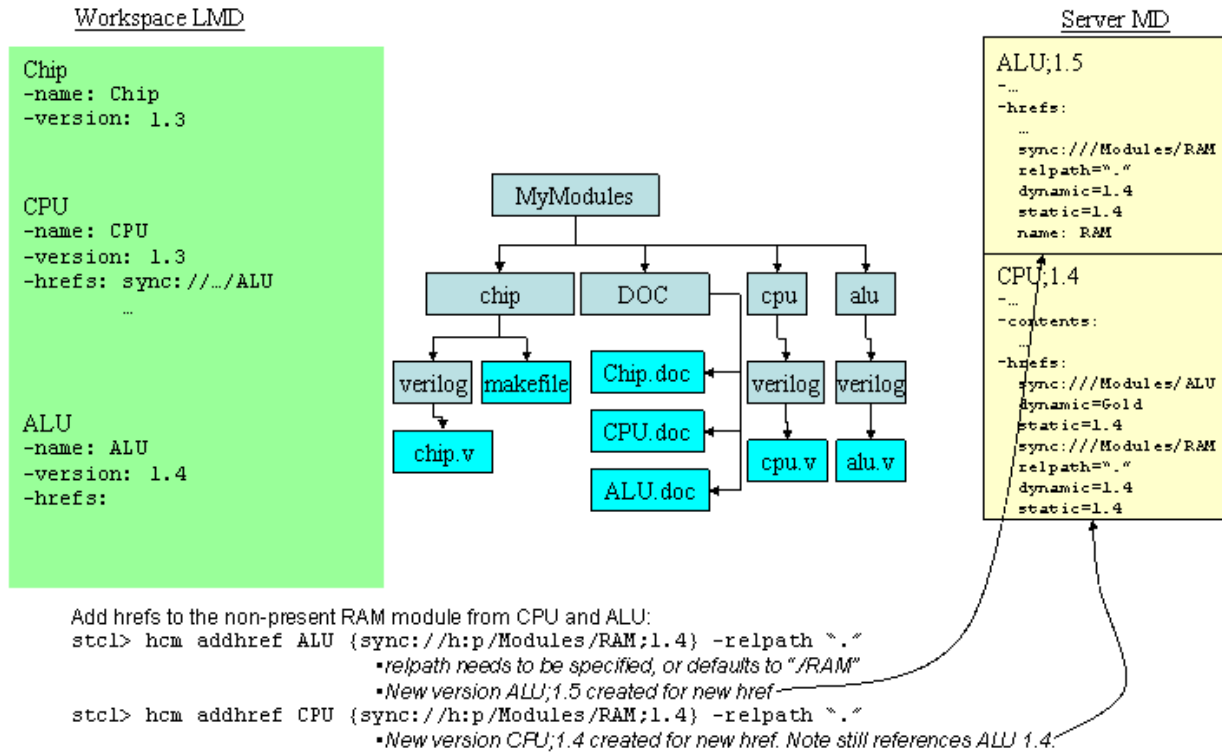
Step 3 of 5

View the next step.

**Step 4: Add an HREF to a Module not in the Workspace**

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.



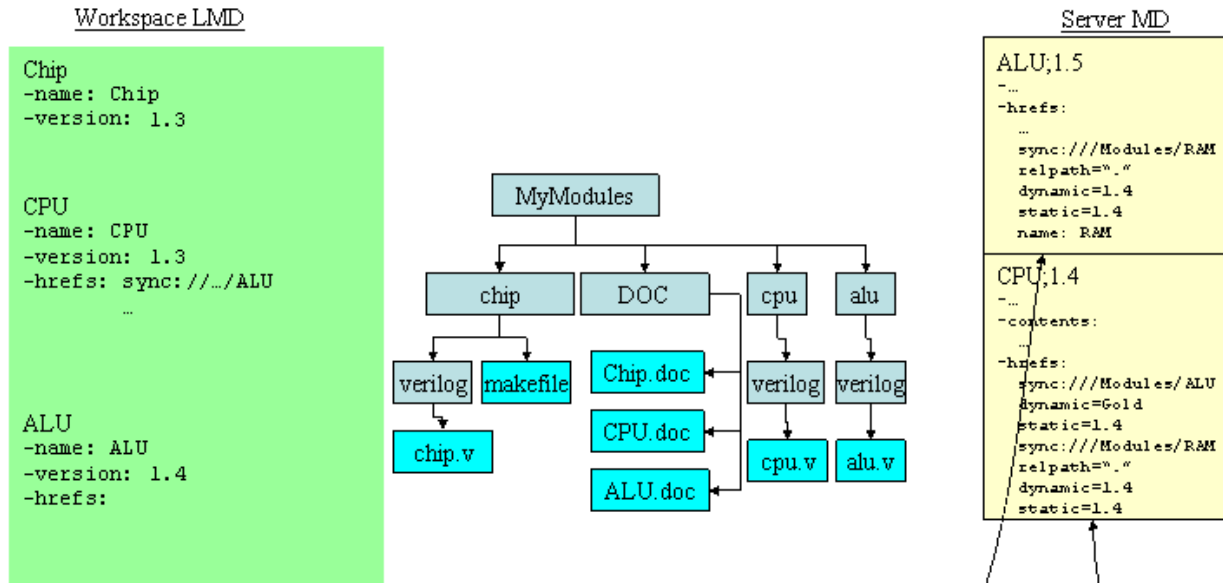


Step 4 of 5

View the next step.

### Step 5: Add an HREF to a Module not in the Workspace

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.



Add hrefs to the non-present RAM module from CPU and ALU:

```

stcl> hcm addhref ALU {sync://h:p/Modules/RAM;1.4} -relpath `."
    •relpath needs to be specified, or defaults to "/RAM"
    •New version ALU;1.5 created for new href
stcl> hcm addhref CPU {sync://h:p/Modules/RAM;1.4} -relpath `."
    •New version CPU;1.4 created for new href. Note still references ALU 1.4.
    
```

Move the "Gold" tag to the new version of ALU

```

stcl> tag -replace Gold sync://h:p/Modules/ALU
    
```

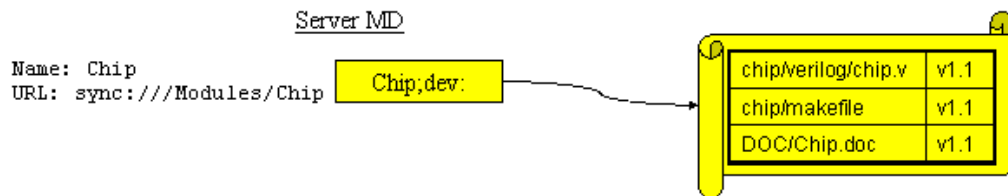
Step 5 of 5

## Creating a Peer Structure Module Hierarchy

### Step 1: Creating a Peer Structure Module Hierarchy

In this use case, "Server MD" refers to the server metadata. Read an overview of module hierarchy.

This example creates a peer structure, in which referenced modules, when fetched, are at the same directory level as the module that references the submodule.



Step 1 of 5

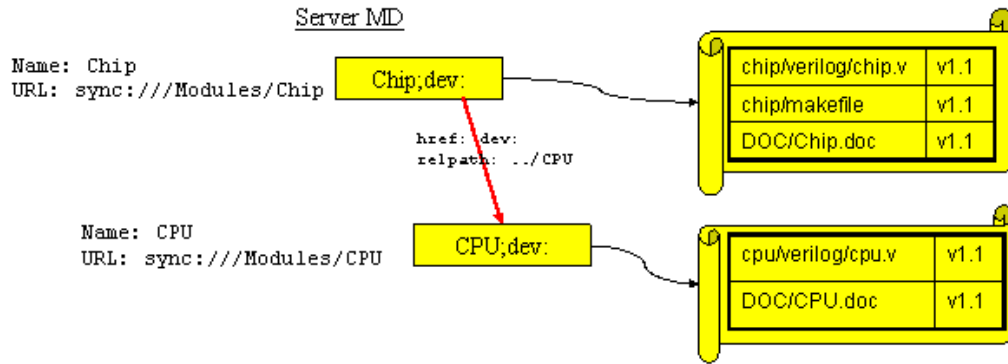
View the next step.

### Step 2: Creating a Peer Structure Module Hierarchy

In this use case, "Server MD" refers to the server metadata. Read an overview of module hierarchy.

# DesignSync Data Manager User's Guide

This example creates a peer structure, in which referenced modules, when fetched, are at the same directory level as the module that references the submodule.



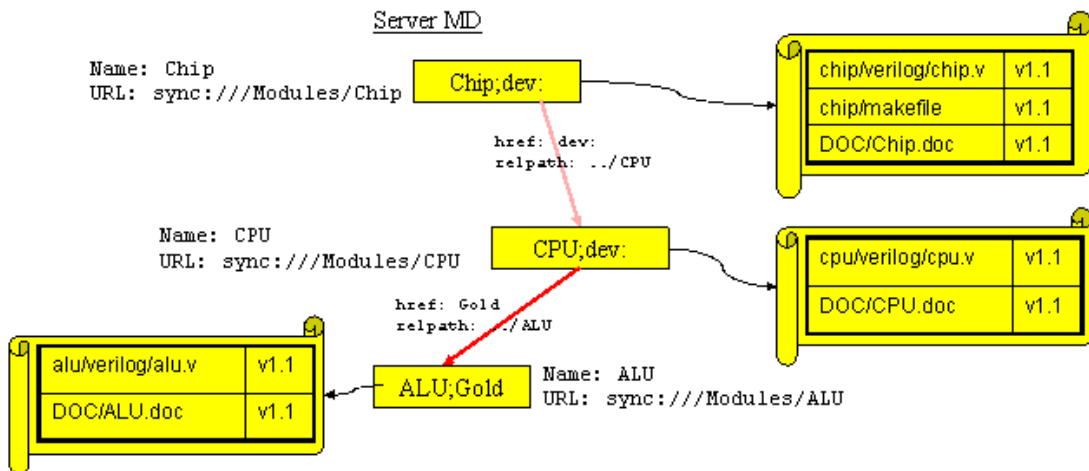
Step 2 of 5

View the next step.

### Step 3: Creating a Peer Structure Module Hierarchy

In this use case, "Server MD" refers to the server metadata. Read an overview of module hierarchy.

This example creates a peer structure, in which referenced modules, when fetched, are at the same directory level as the module that references the submodule.



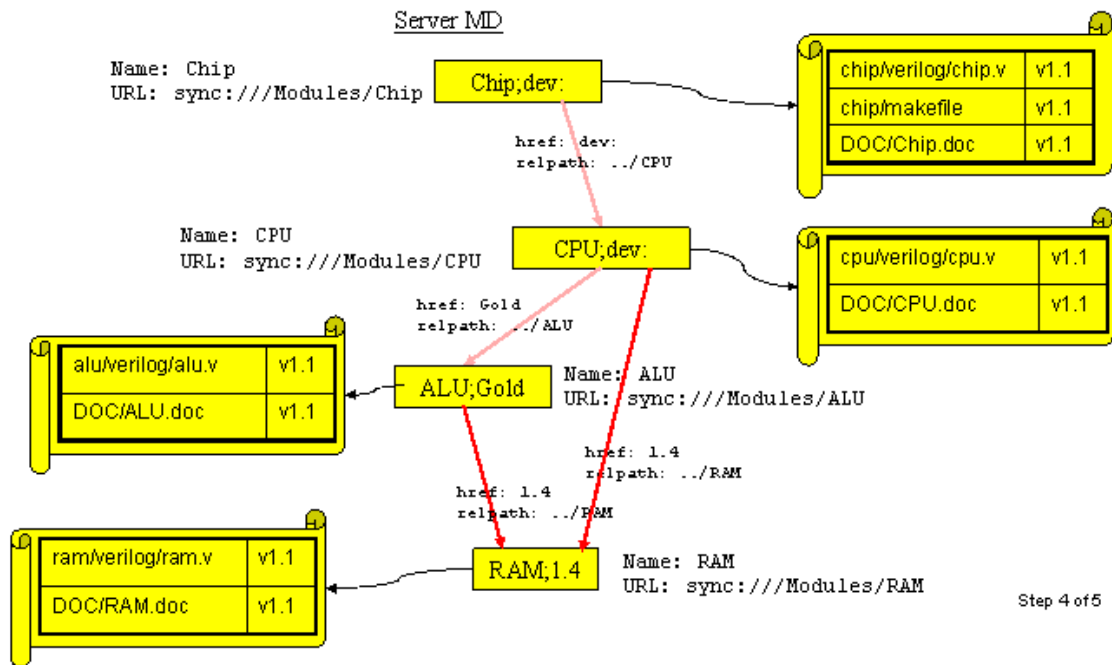
Step 3 of 5

View the next step.

#### Step 4: Creating a Peer Structure Module Hierarchy

In this use case, "Server MD" refers to the server metadata. Read an overview of module hierarchy.

This example creates a peer structure, in which referenced modules, when fetched, are at the same directory level as the module that references the submodule.

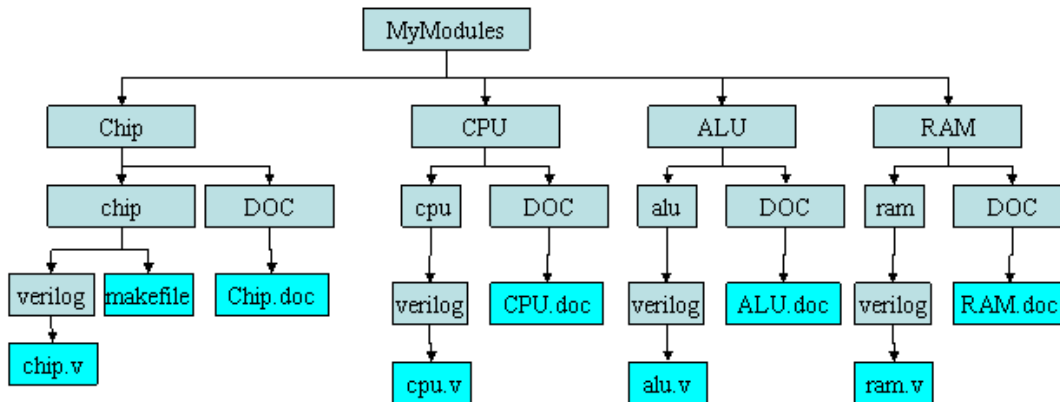


View the next step.

### Step 5: Creating a Peer Structure Module Hierarchy

In this use case, "Server MD" refers to the server metadata. Read an overview of module hierarchy.

Fetching the module Chip with a base directory of MyModules/Chip creates a peer structure:



Step 5 of 5

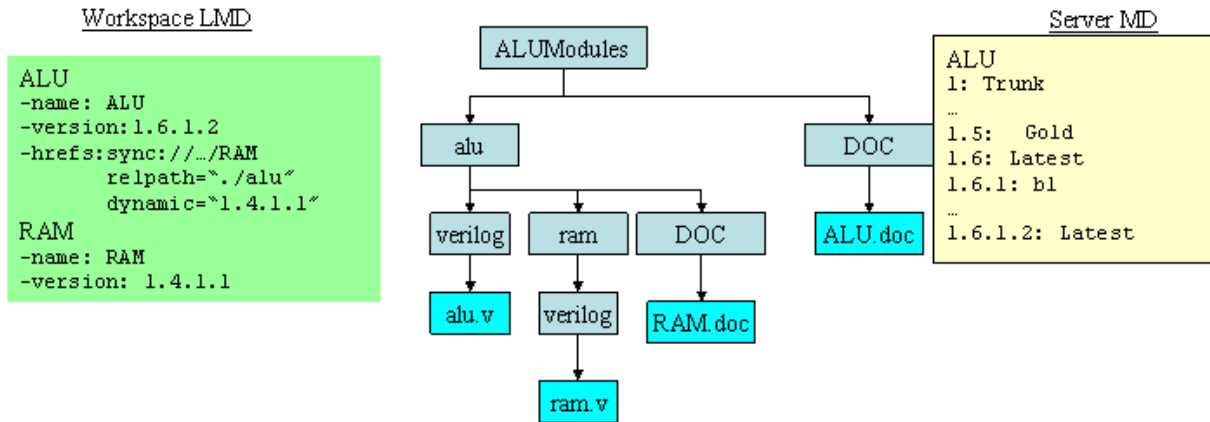
## Updating Module Hierarchy

### Modifying Module Hierarchy: New "Gold" Version of ALU Created

#### Step 1: New "Gold" Version of ALU Created

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.

The ALU team has developed a new "Gold" version of their module, with new features. The "Gold" version is intended to replace all earlier versions. The new "Gold" version is version 1.6.1.2 of the ALU module. The new "Gold" ALU version uses a private version of the RAM module. The private version of the RAM module is different than the general version of the RAM module used previously by both ALU and CPU (in the "Creating Module Hierarchy" use cases).



Step 1 of 5

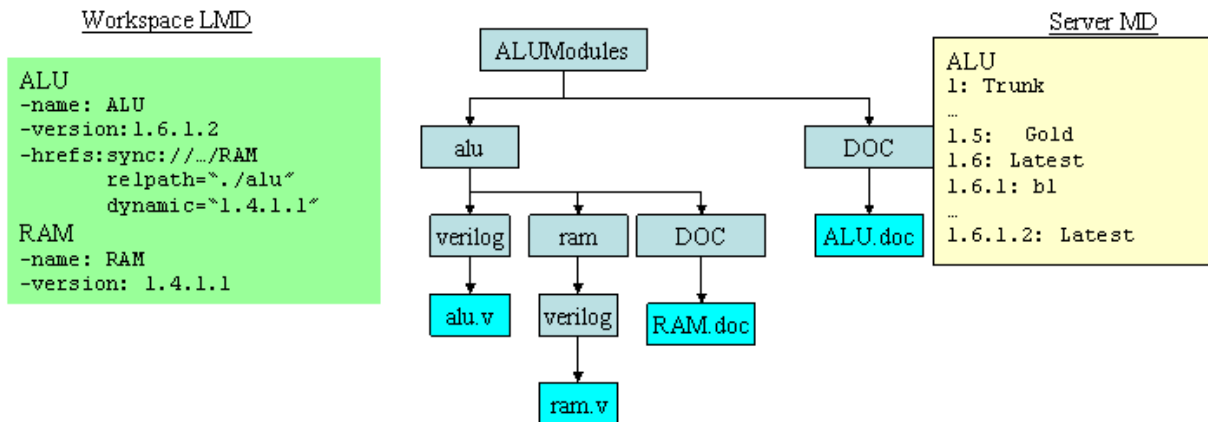
View the next step.

**Step 2: New "Gold" Version of ALU Created**

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.

The ALU team has developed a new "Gold" version of their module, with new features. The "Gold" version is intended to replace all earlier versions. The new "Gold" version is version 1.6.1.2 of the ALU module. The new "Gold" ALU version uses a private version of the RAM module. The private version of the RAM module is different than the general version of the RAM module used previously by both ALU and CPU (in the "Creating Module Hierarchy" use cases).





Tag the old Gold version with a new tag, so we can return to it:

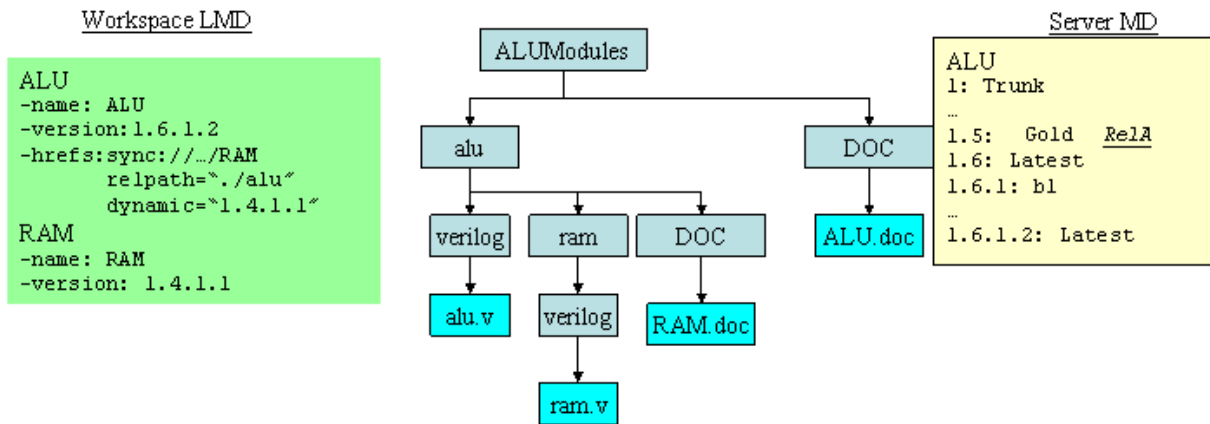
Step 2 of 5

View the next step.

### Step 3: New "Gold" Version of ALU Created

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.

The ALU team has developed a new "Gold" version of their module, with new features. The "Gold" version is intended to replace all earlier versions. The new "Gold" version is version 1.6.1.2 of the ALU module. The new "Gold" ALU version uses a private version of the RAM module. The private version of the RAM module is different than the general version of the RAM module used previously by both ALU and CPU (in the "Creating Module Hierarchy" use cases).



Tag the old Gold version with a new tag, so we can return to it:

```

stcl> tag -immutable -version Gold ReIA \
sync: //h:p/Modules/ALU
    
```

*That is, "ReIA" becomes an "alias" of "Gold"  
Making immutable stops the tag being moved/deleted*

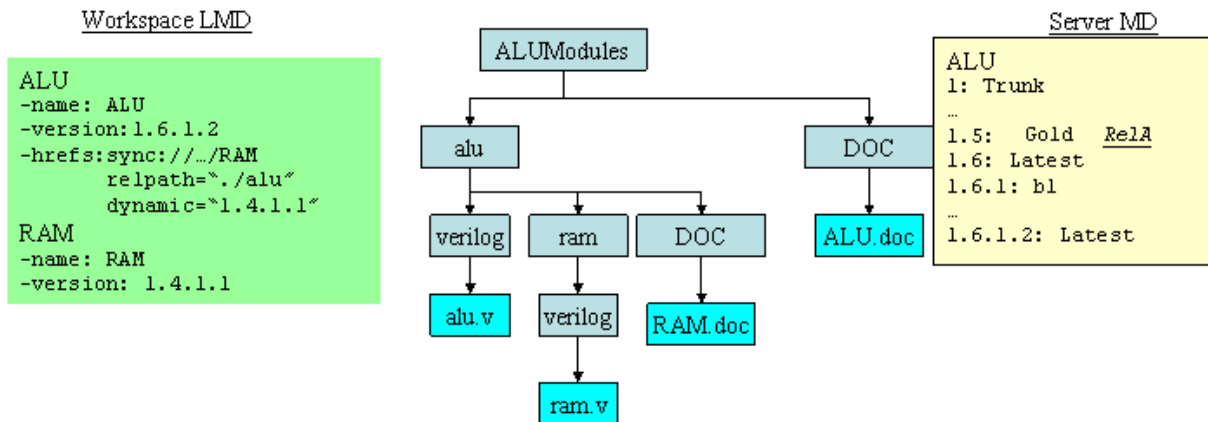
Step 3 of 5

View the next step.

#### Step 4: New "Gold" Version of ALU Created

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.

The ALU team has developed a new "Gold" version of their module, with new features. The "Gold" version is intended to replace all earlier versions. The new "Gold" version is version 1.6.1.2 of the ALU module. The new "Gold" ALU version uses a private version of the RAM module. The private version of the RAM module is different than the general version of the RAM module used previously by both ALU and CPU (in the "Creating Module Hierarchy" use cases).



Tag the old Gold version with a new tag, so we can return to it:

```

stcl> tag -immutable -version Gold ReIA \
sync: //h:p/Modules/ALU

```

Promote the new version of the ALU to be the "Gold" version:

*That is, "ReIA" becomes an "alias" of "Gold"*  
*Making immutable stops the tag being moved/deleted*

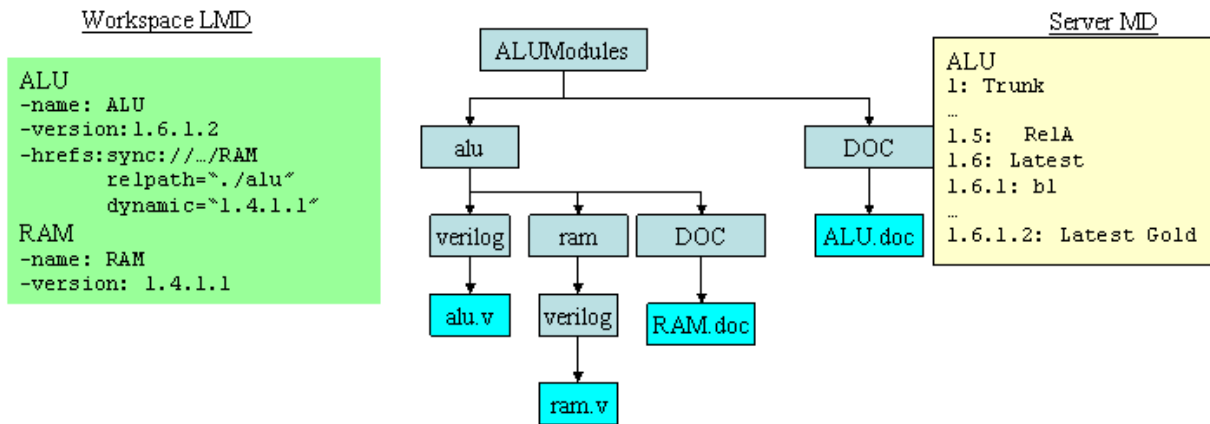
Step 4 of 5

View the next step.

### Step 5: New "Gold" Version of ALU Created

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.

The ALU team has developed a new "Gold" version of their module, with new features. The "Gold" version is intended to replace all earlier versions. The new "Gold" version is version 1.6.1.2 of the ALU module. The new "Gold" ALU version uses a private version of the RAM module. The private version of the RAM module is different than the general version of the RAM module used previously by both ALU and CPU (in the "Creating Module Hierarchy" use cases).



Tag the old Gold version with a new tag, so we can return to it:

```

stcl> tag -immutable -version Gold ReIA \
sync: //h:p/Modules/ALU
    
```

*That is, "ReIA" becomes an "alias" of "Gold"*  
*Making immutable stops the tag being moved/deleted*

Promote the new version of the ALU to be the "Gold" version:

```

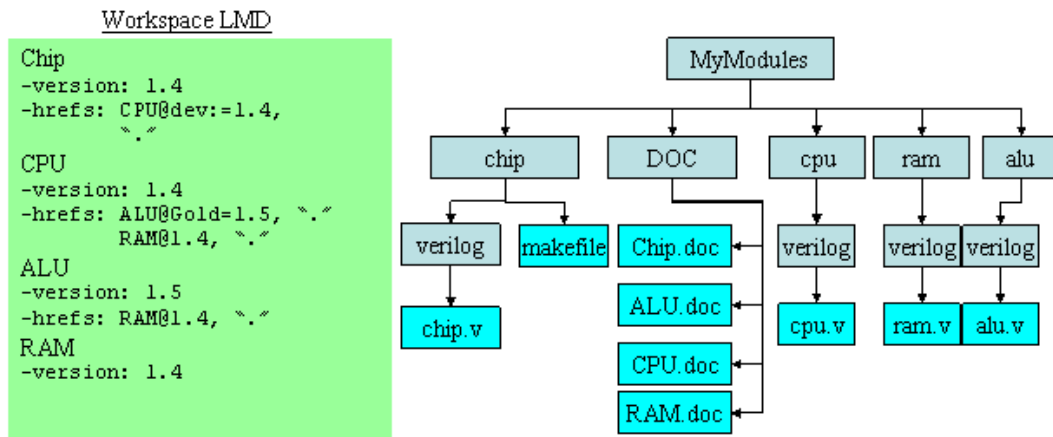
stcl> tag -replace Gold -version b1:Latest sync: //h:p/Modules/ALU
    
```

Step 5 of 5

## Modifying Module Hierarchy: Chip Team Uses New ALU Version

### Step 1: Chip Team Uses New ALU Version

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.

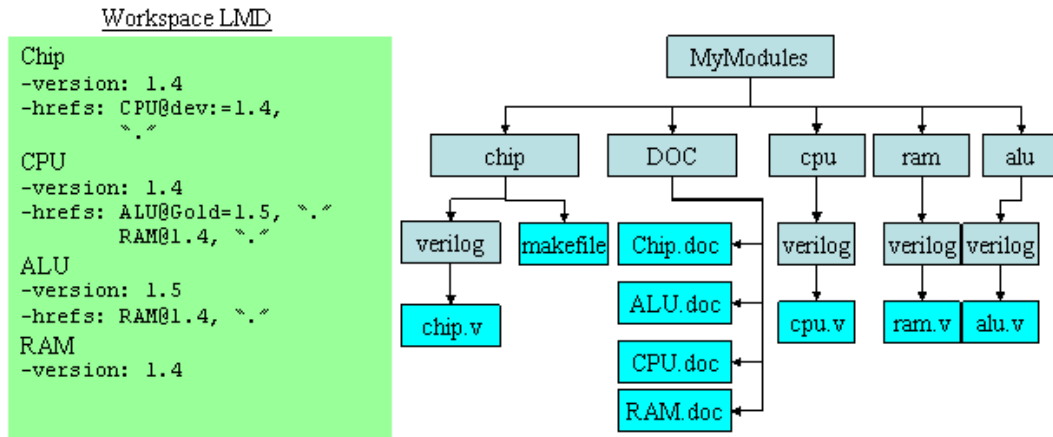


Step 1 of 9

View the next step.

### Step 2: Chip Team Uses New ALU Version

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.



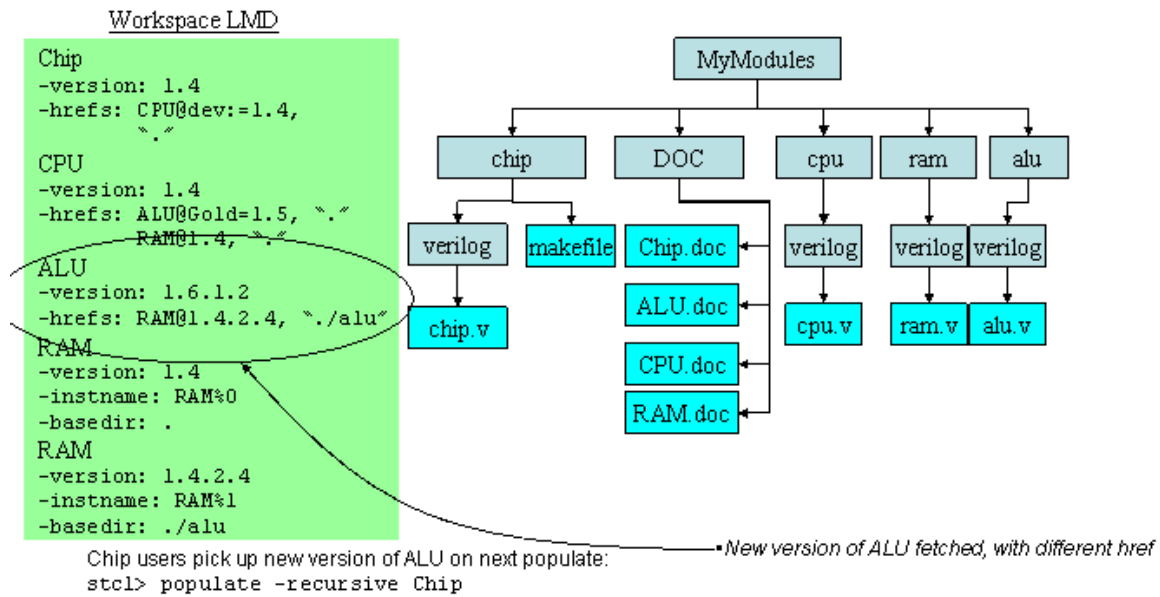
Chip users pick up new version of ALU on next populate:  
stcl> populate -recursive Chip

Step 2 of 9

View the next step.

### Step 3: Chip Team Uses New ALU Version

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.

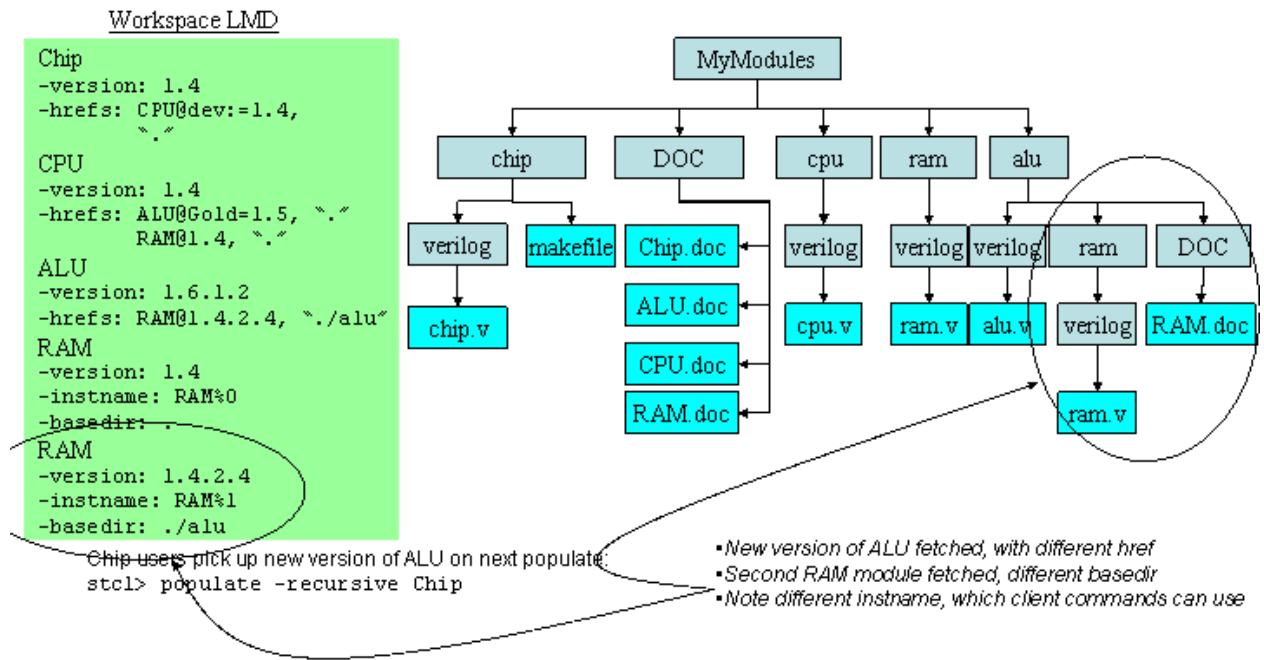


Step 3 of 9

View the next step.

#### Step 4: Chip Team Uses New ALU Version

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.



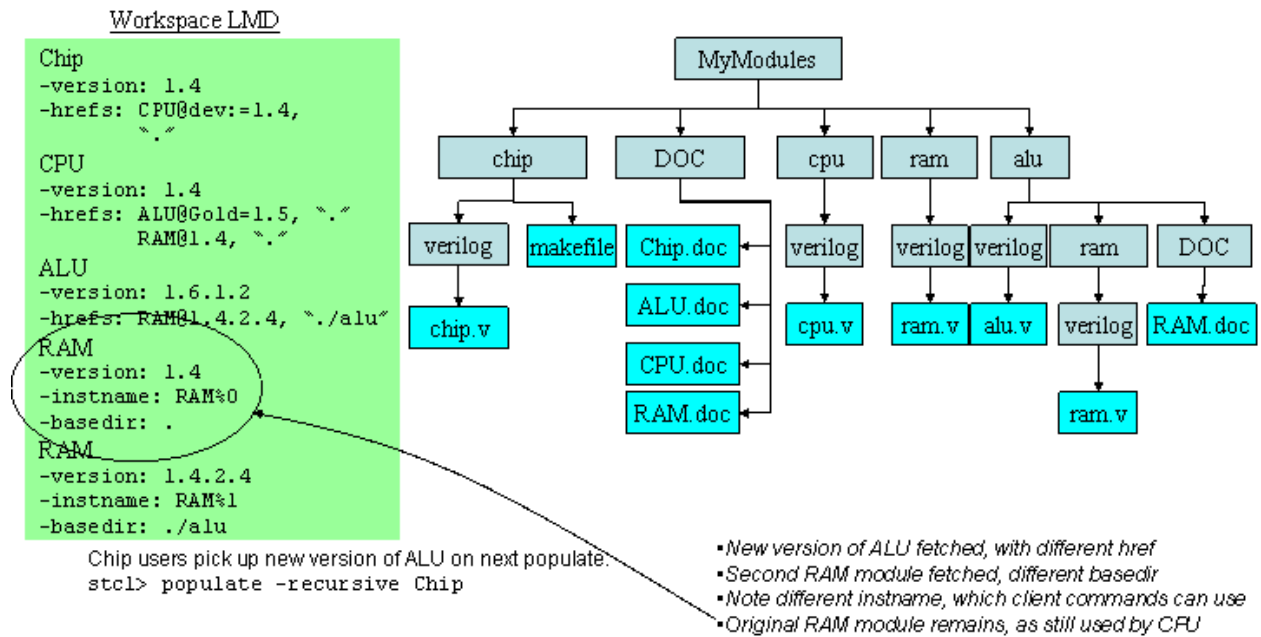
Step 4 of 9

View the next step.

### Step 5: Chip Team Uses New ALU Version

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.



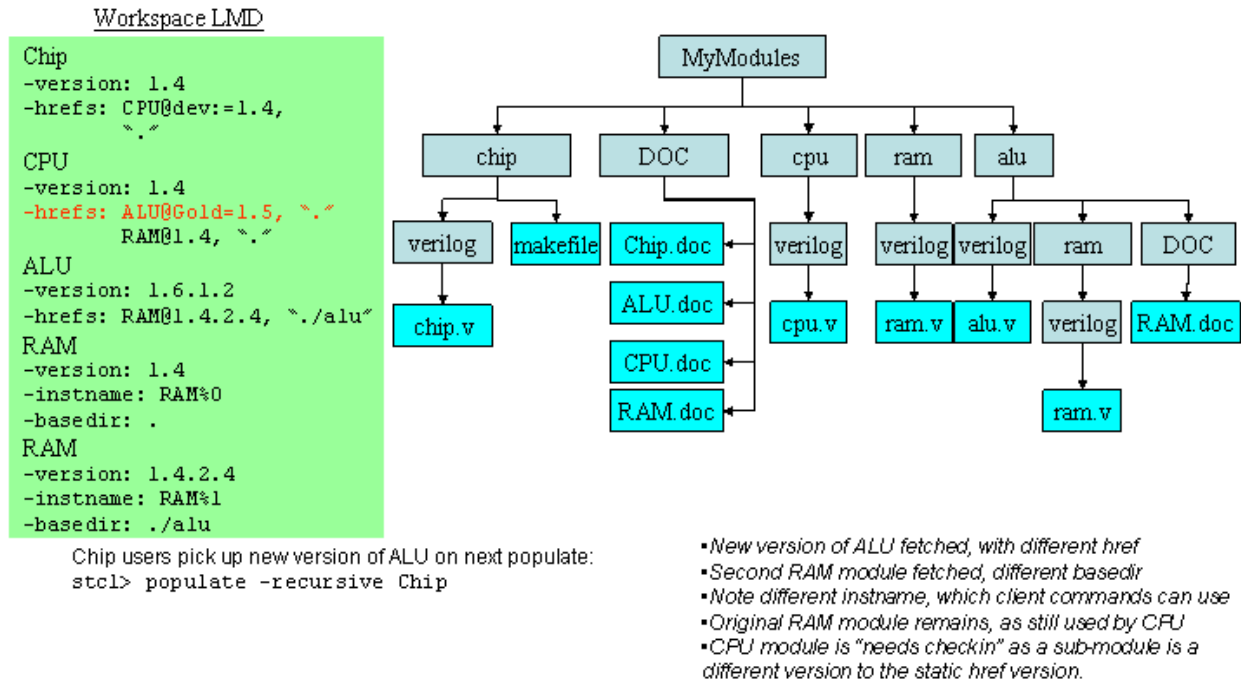


Step 5 of 9

View the next step.

### Step 6: Chip Team Uses New ALU Version

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.



Step 6 of 9

View the next step.

### Step 7: Chip Team Uses New ALU Version

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.

## Workspace LMD

```

Chip
-version: 1.4
-hrefs: CPU@dev=1.4,
      `.`

CPU
-version: 1.4
-hrefs: ALU@Gold=1.5, `.`
      RAM@1.4, `.`

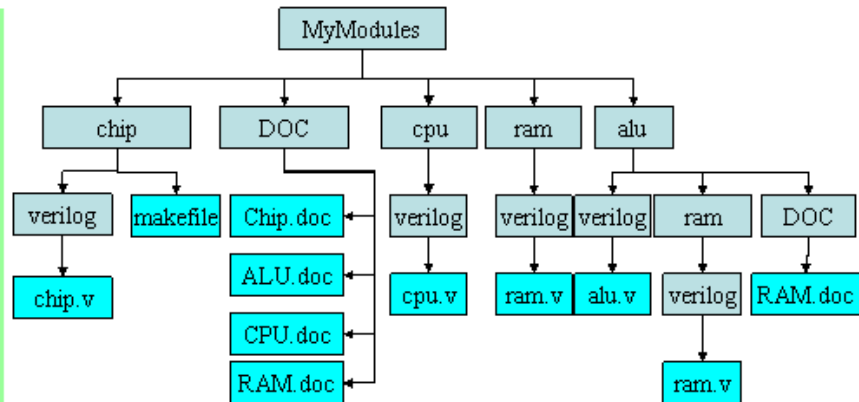
ALU
-version: 1.6.1.2
-hrefs: RAM@1.4.2.4, `./alu`

RAM
-version: 1.4
-instname: RAM%0
-basedir: .

RAM
-version: 1.4.2.4
-instname: RAM%1
-basedir: ./alu

```

Chip users pick up new version of ALU on next populate:  
 stcl> populate -recursive Chip



- New version of ALU fetched, with different href
- Second RAM module fetched, different basedir
- Note different instname, which client commands can use
- Original RAM module remains, as still used by CPU
- CPU module is "needs checkin" as a sub-module is a different version to the static href version.
- Could equally have simply populated:
  - populate -rec CPU (or ALU)
  - populate -rec -modulecontext CPU ALU

Step 7 of 9

View the next step.

### Step 8: Chip Team Uses New ALU Version

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.

Workspace LMD

```

Chip
-version: 1.4
-hrefs: CPU@dev=1.4,
      `.`

CPU
-version: 1.4
-hrefs: ALU@Gold=1.5, `.`
      RAM@1.4, `.`

ALU
-version: 1.6.1.2
-hrefs: RAM@1.4.2.4, `./alu`

RAM
-version: 1.4
-instname: RAM%0
-basedir: .

RAM
-version: 1.4.2.4
-instname: RAM%1
-basedir: ./alu
                    
```

Chip users pick up new version of ALU on next populate:  
`stcl> populate -recursive Chip`

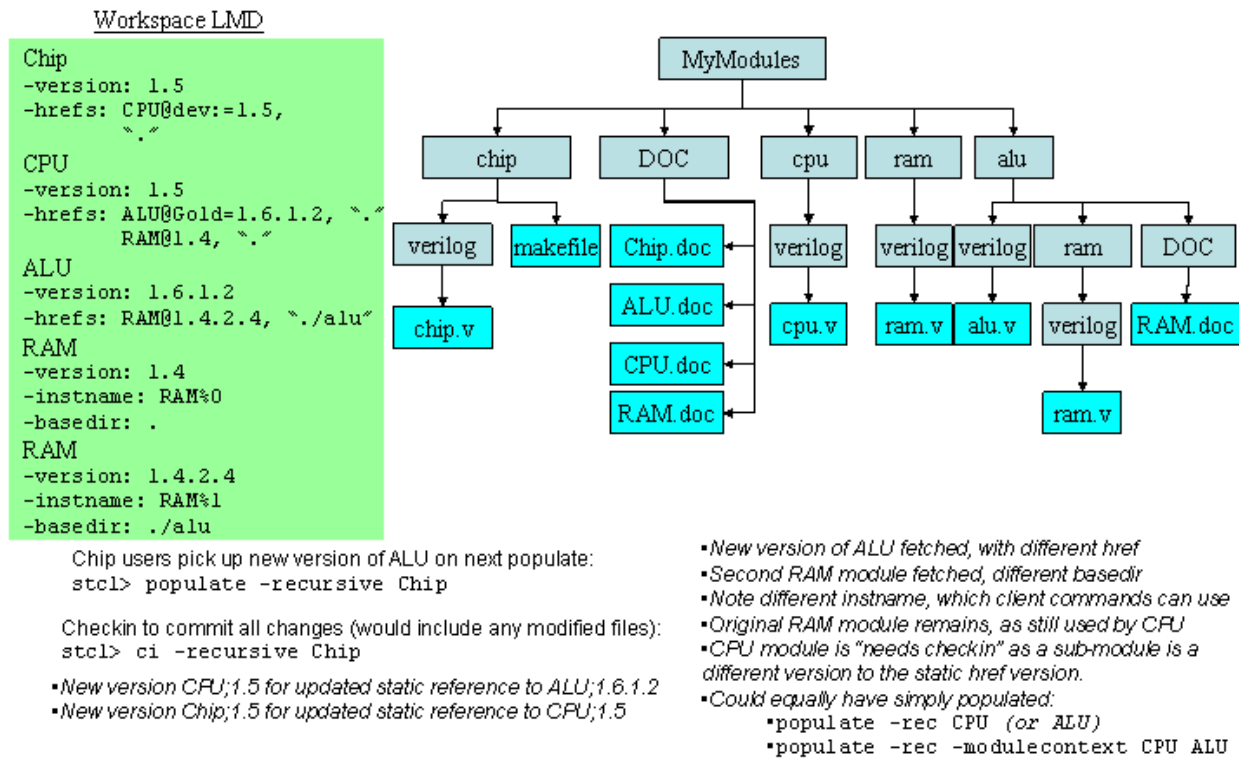
Checkin to commit all changes (would include any modified files):  
`stcl> ci -recursive Chip`

- New version of ALU fetched, with different href
- Second RAM module fetched, different basedir
- Note different instname, which client commands can use
- Original RAM module remains, as still used by CPU
- CPU module is "needs checkin" as a sub-module is a different version to the static href version.
- Could equally have simply populated:
  - populate -rec CPU (or ALU)
  - populate -rec -modulecontext CPU ALU

View the next step.

**Step 9: Chip Team Uses New ALU Version**

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.



Step 9 of 9

## Modifying Module Hierarchy: CPU Team Reverts to Earlier ALU Version

### Step 1: CPU Team Reverts to Earlier ALU Version

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.

The CPU team performs tests with the "Gold" version of ALU. The CPU team decides that the new ALU features are not required, and come at the cost of a larger floor plan. The CPU team reverts to the "ReIA" version of the ALU.

Workspace LMD

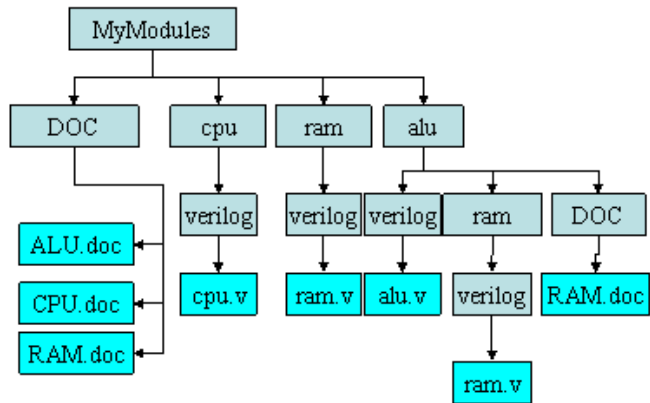
```

CPU
-version: 1.5
-hrefs: ALU@Gold=1.6.1.2, \".\"
      RAM@1.4, \".\"

RAM
-version: 1.4
-instname: RAM
-basedir: .

ALU
-version: 1.6.1.2
-hrefs: RAM@1.4.2.4, \"/alu\"

RAM
-version: 1.4.2.4
-instname: RAM%1
-basedir: ./alu
    
```



Step 1 of 6

View the next step.

**Step 2: CPU Team Reverts to Earlier ALU Version**

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.

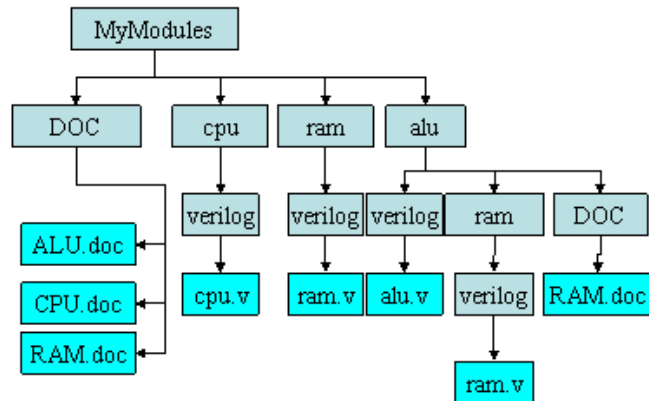
The CPU team performs tests with the "Gold" version of ALU. The CPU team decides that the new ALU features are not required, and come at the cost of a larger floor plan. The CPU team reverts to the "ReIA" version of the ALU.

## Workspace LMD

```

CPU
-version: 1.7
-hrefs: ALU@RelA=1.5, \."
        RAM@1.4, \."
RAM
-version: 1.4
-instname: RAM
-basedir: .
ALU
-version: 1.6.1.2
-hrefs: RAM@1.4.2.4, \./alu"
RAM
-version: 1.4.2.4
-instname: RAM%1
-basedir: ./alu

```



```

CPU user removes and replaces HREF to ALU:
stcl> hcm rmhref CPU ALU
CPU version 1.6 created
stcl> hcm addhref CPU [url vault ALU]\;relA
CPU version 1.7 created

```

Step 2 of 6

View the next step.

### Step 3: CPU Team Reverts to Earlier ALU Version

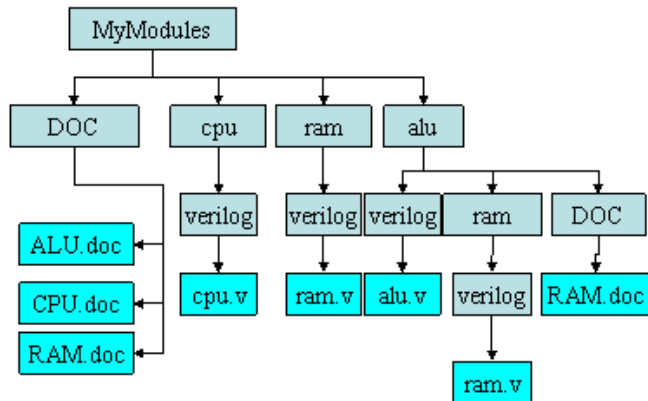
In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.

The CPU team performs tests with the "Gold" version of ALU. The CPU team decides that the new ALU features are not required, and come at the cost of a larger floor plan. The CPU team reverts to the "RelA" version of the ALU.

Workspace LMD

```

CPU
-version: 1.7
-hrefs: ALU@RelA=1.5, \."
        RAM@1.4, \."
RAM
-version: 1.4
-iname: RAM
-basedir: .
ALU
-version: 1.6.1.2
-hrefs: RAM@1.4.2.4, \./alu"
RAM
-version: 1.4.2.4
-iname: RAM%1
-basedir: ./alu
    
```



```

CPU user removes and replaces HREF to ALU:
stcl> hcm rmhref CPU ALU
CPU version 1.6 created
stcl> hcm addhref CPU [url vault ALU]\;relA
CPU version 1.7 created

CPU users re-populates to fetch correct ALU:
stcl> populate -rec CPU
    
```

Step 3 of 6

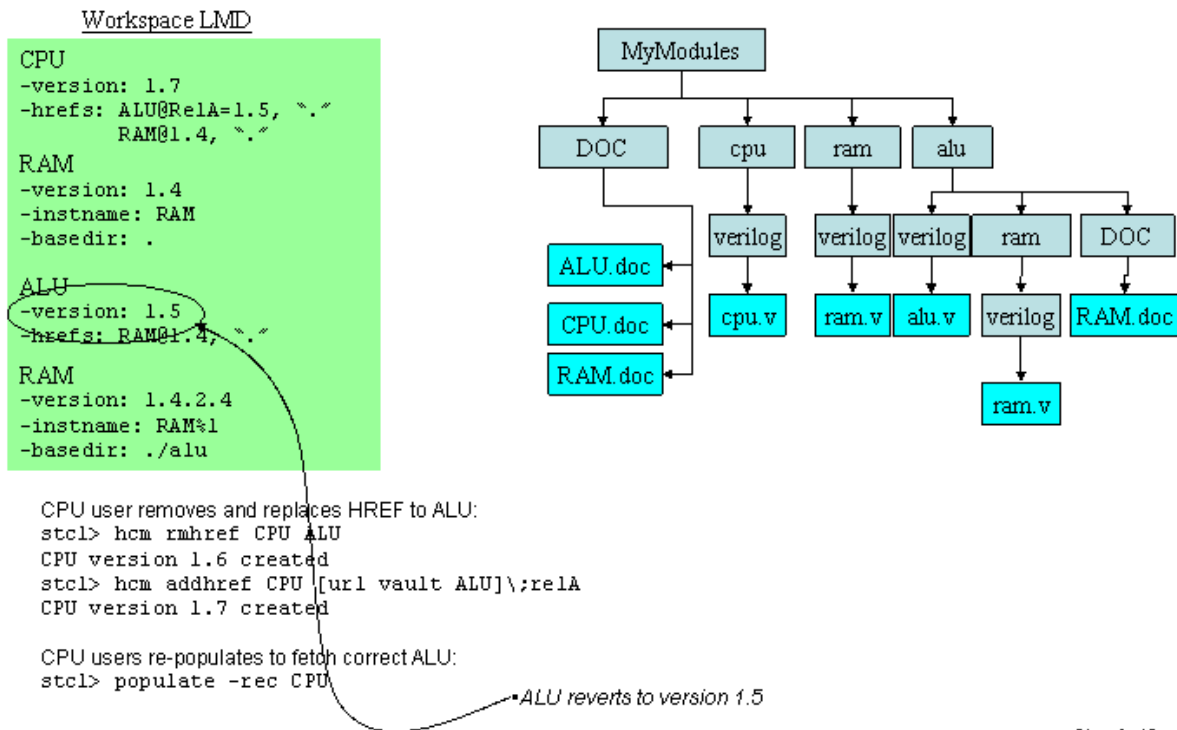
View the next step.

**Step 4: CPU Team Reverts to Earlier ALU Version**

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.

The CPU team performs tests with the "Gold" version of ALU. The CPU team decides that the new ALU features are not required, and come at the cost of a larger floor plan. The CPU team reverts to the "RelA" version of the ALU.





View the next step.

### Step 5: CPU Team Reverts to Earlier ALU Version

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.

The CPU team performs tests with the "Gold" version of ALU. The CPU team decides that the new ALU features are not required, and come at the cost of a larger floor plan. The CPU team reverts to the "RelA" version of the ALU.

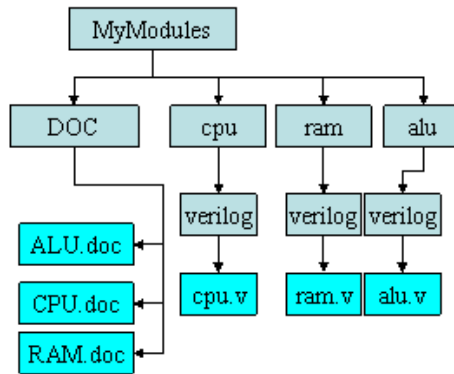
Workspace LMD

```

CPU
-version: 1.7
-hrefs: ALU@RelA=1.5, \."
        RAM@1.4, \."

RAM
-version: 1.4
-instname: RAM
-basedir: .

ALU
-version: 1.5
-hrefs: RAM@1.4, \."
    
```



```

CPU user removes and replaces HREF to ALU:
stcl> hcm rmhref CPU ALU
CPU version 1.6 created
stcl> hcm addhref CPU [url vault ALU]\;relA
CPU version 1.7 created
    
```

```

CPU users re-populates to fetch correct ALU:
stcl> populate -rec CPU
    
```

- ALU reverts to version 1.5
- Second version of RAM is removed as no longer used

Step 5 of 6

View the next step.

**Step 6: CPU Team Reverts to Earlier ALU Version**

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module hierarchy. View the module structure that is used in this example.

The CPU team performs tests with the "Gold" version of ALU. The CPU team decides that the new ALU features are not required, and come at the cost of a larger floor plan. The CPU team reverts to the "RelA" version of the ALU.

## Workspace LMD

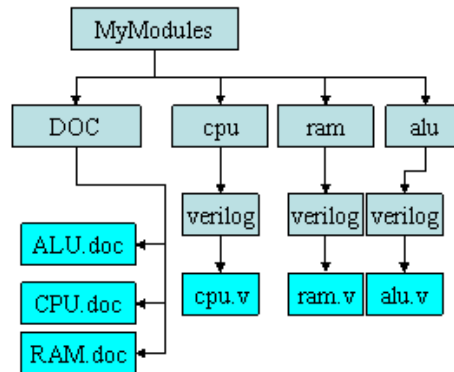
```

CPU
-version: 1.7
-hrefs: ALU@relA=1.5, \."
        RAM@1.4, \."

RAM
-version: 1.4
-instname: RAM
-basedir: .

ALU
-version: 1.5
-hrefs: RAM@1.4, \."

```



```

CPU user removes and replaces HREF to ALU:
stcl> hcm rmhref CPU ALU
CPU version 1.6 created
stcl> hcm addhref CPU [url vault ALU]\;relA
CPU version 1.7 created

```

```

CPU users re-populates to fetch correct ALU:
stcl> populate -rec CPU

```

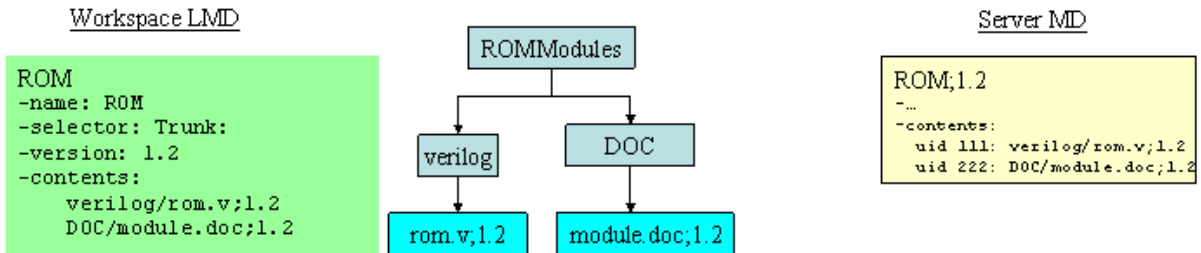
- ALU reverts to version 1.5
- Second version of RAM is removed as no longer used
- New version of CPU is now ready for the Chip users to fetch <sup>Step 6 of 6</sup>

## Moving a File

### Step 1: Moving a File

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of directory versioning.

# DesignSync Data Manager User's Guide

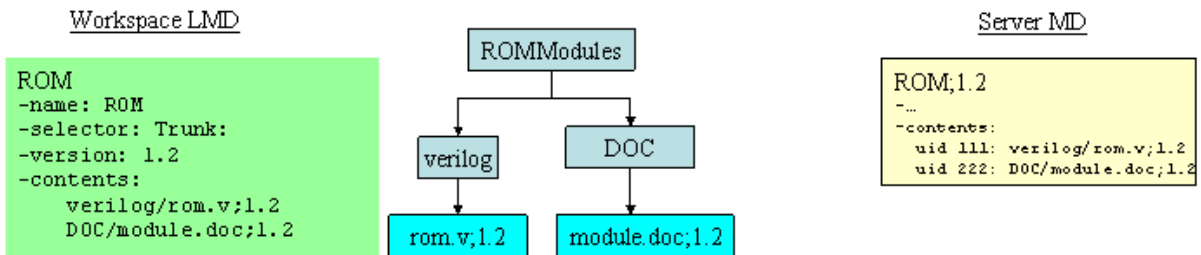


Step 1 of 6

View the next step.

## Step 2: Moving a File

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of directory versioning.



Move a file:

```
stcl> hcm mvmember -modulecontext ROM \
  DOC/module.doc DOC/ROM.doc
Moving objects in module ROM*0...
DOC/module.doc : Moved object to DOC/ROM.doc
ROM*0 : Created new module version 1.3
ROM*0 : Version of module in workspace updated.
```

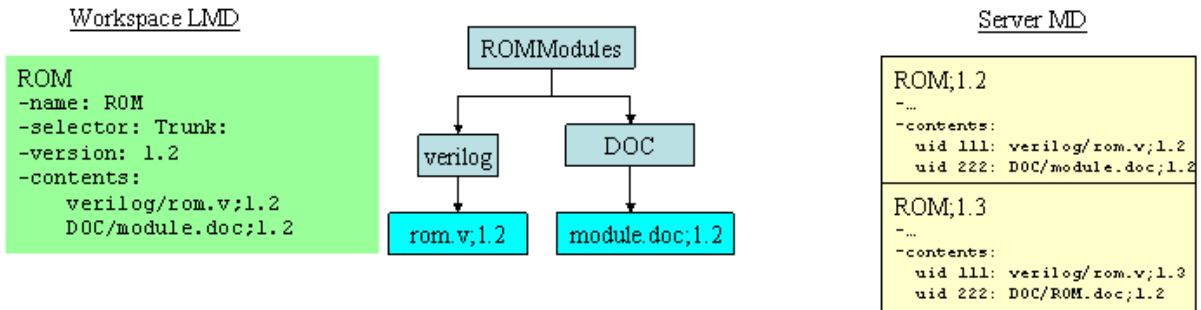
Step 2 of 6

View the next step.

### Step 3: Moving a File

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of directory versioning.

# DesignSync Data Manager User's Guide



Move a file:

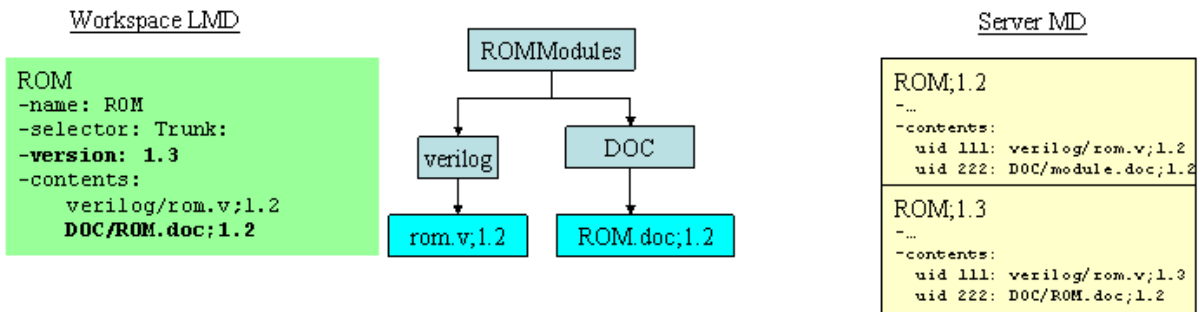
```
stcl> hcm mvmember -modulecontext ROM \      *New module version created
      DOC/module.doc DOC/ROM.doc
Moving objects in module ROM*0...
DOC/module.doc : Moved object to DOC/ROM.doc
ROM*0 : Created new module version 1.3
ROM*0 : Version of module in workspace updated.
```

Step 3 of 6

View the next step.

## Step 4: Moving a File

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of directory versioning.



Move a file:

```
stcl> hcm mvmember -modulecontext ROM \
      DOC/module.doc DOC/ROM.doc
```

- *New module version created*
- *File moved in workspace and module version updated*

Moving objects in module ROM\*0...

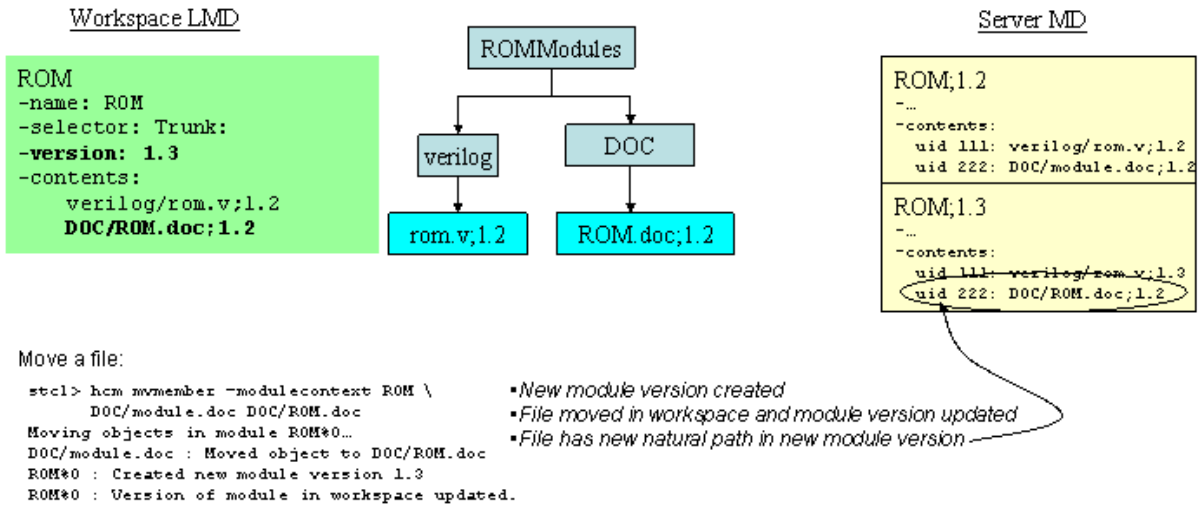
```
DOC/module.doc : Moved object to DOC/ROM.doc
ROM*0 : Created new module version 1.3
ROM*0 : Version of module in workspace updated.
```

Step 4 of 6

View the next step.

### Step 5: Moving a File

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of directory versioning.



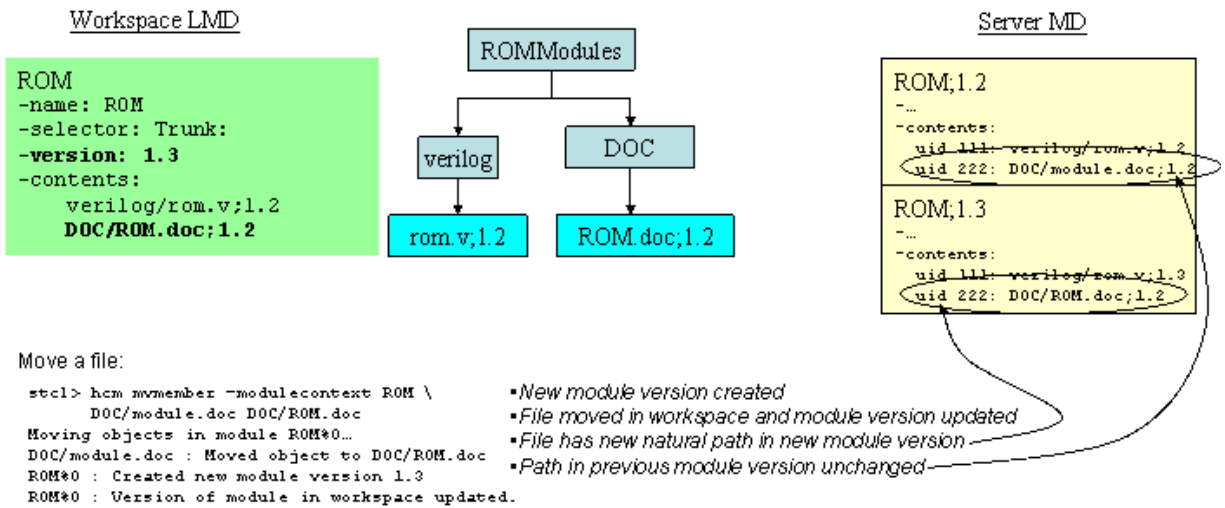
Step 5 of 6

View the next step.

### Step 6: Moving a File

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of directory versioning.





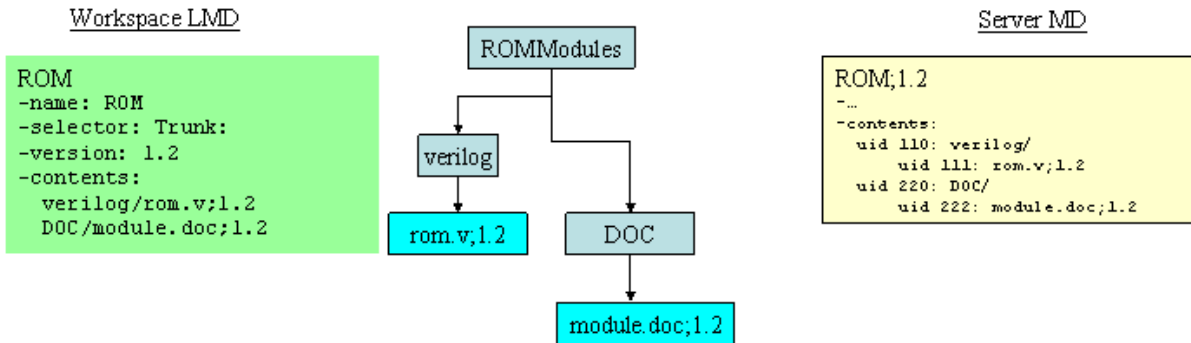
Step 6 of 6

## Moving a Folder

### Step 1: Moving a Folder

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of directory versioning.

# DesignSync Data Manager User's Guide

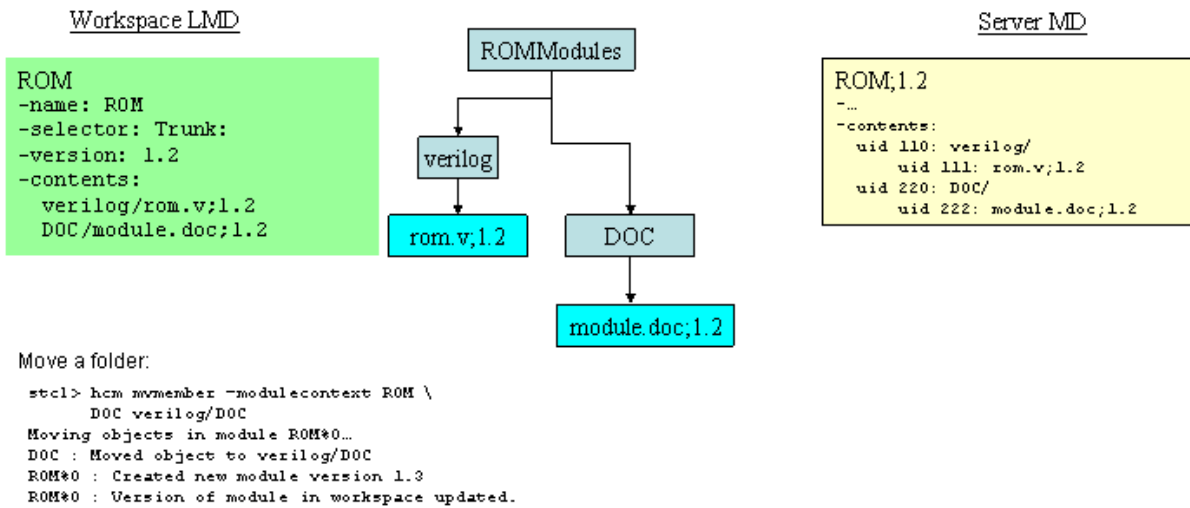


Step 1 of 6

View the next step.

## Step 2: Moving a Folder

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of directory versioning.



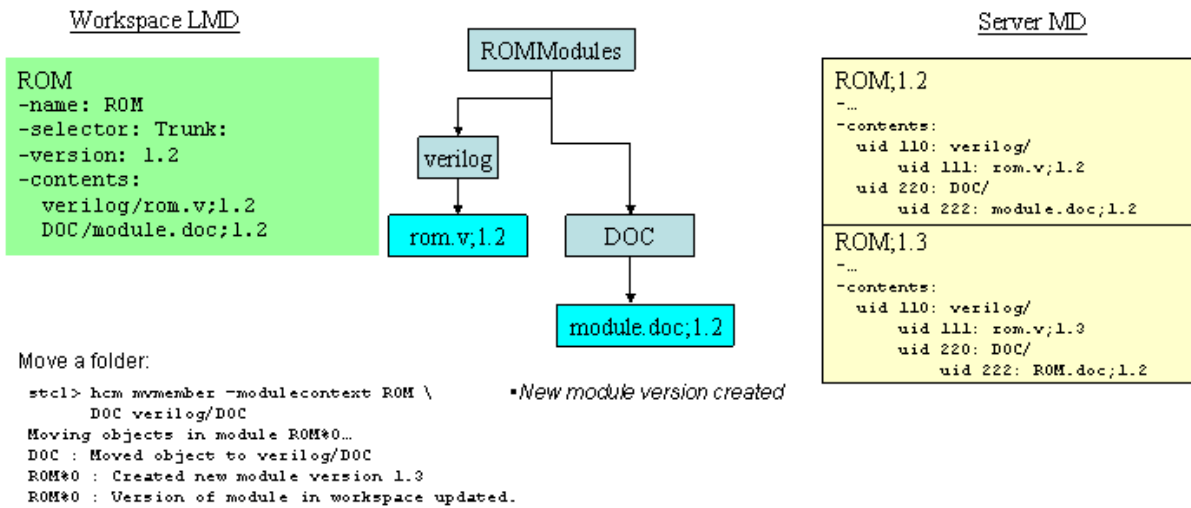
Step 2 of 6

View the next step.

### Step 3: Moving a Folder

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of directory versioning.

# DesignSync Data Manager User's Guide

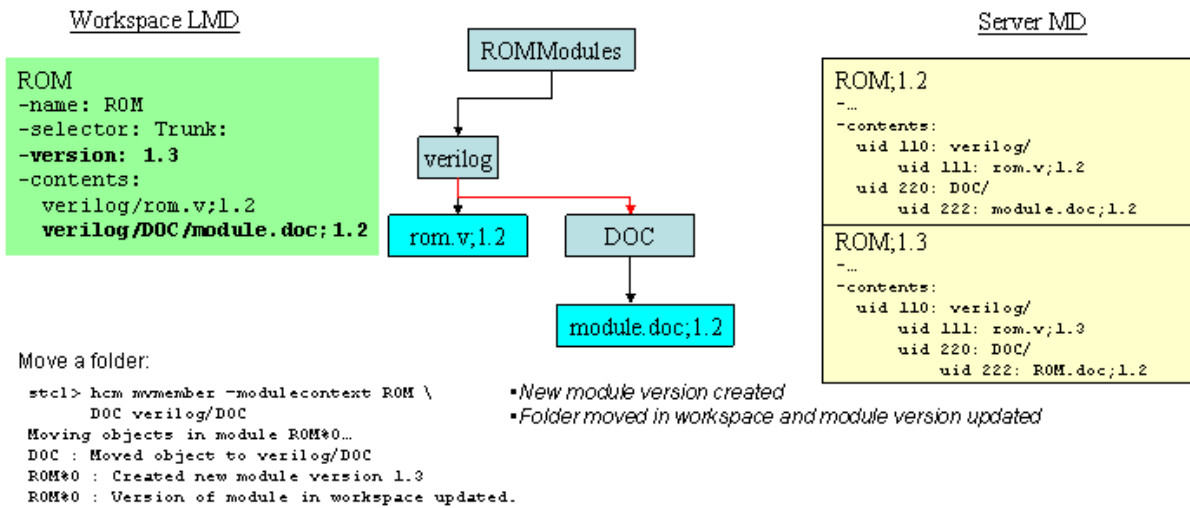


Step 3 of 6

View the next step.

## Step 4: Moving a Folder

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of directory versioning.

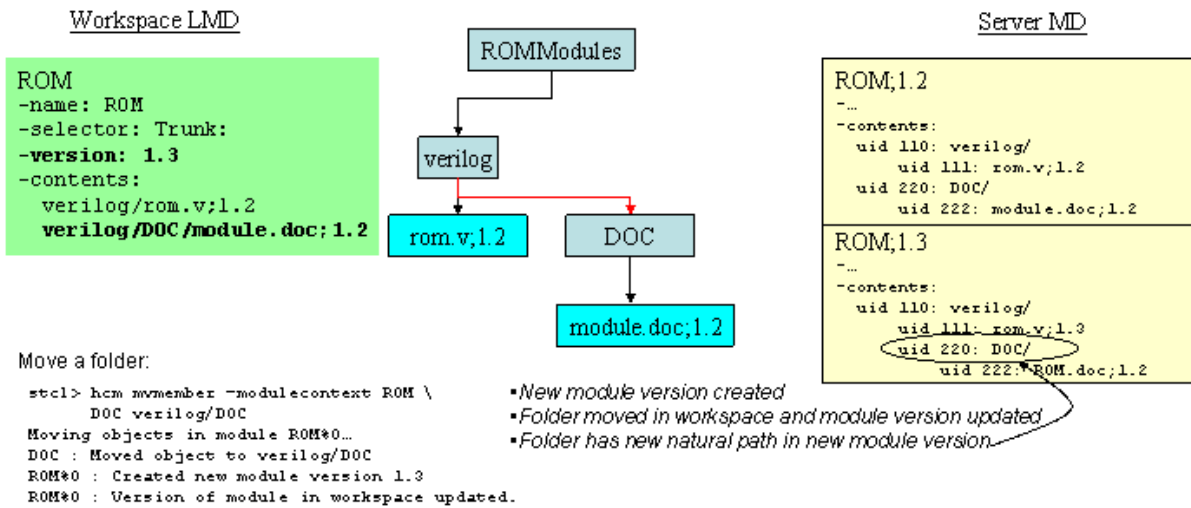


Step 4 of 6

View the next step.

### Step 5: Moving a Folder

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of directory versioning.

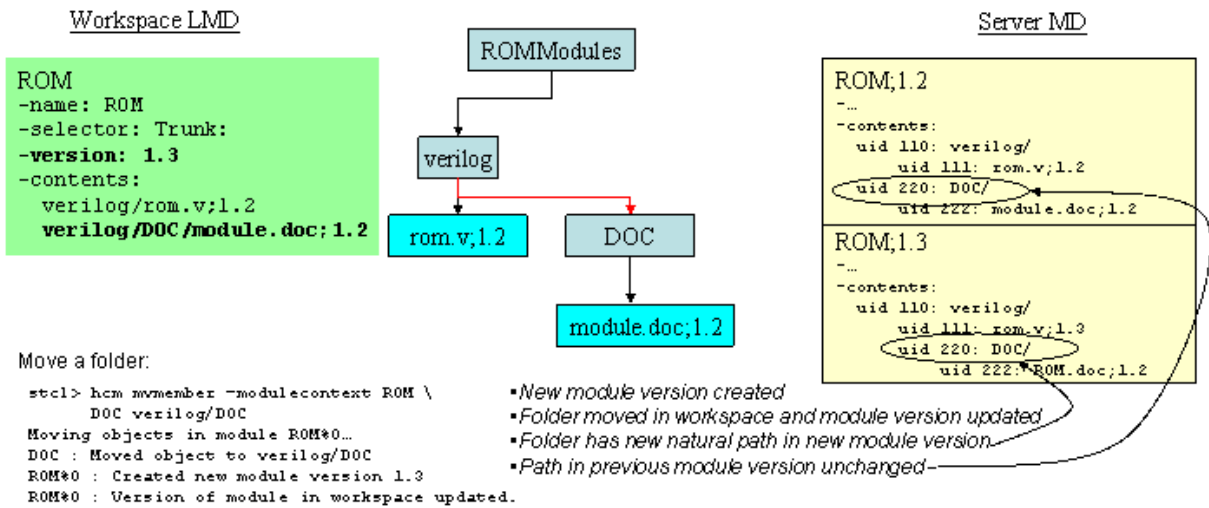


Step 5 of 6

View the next step.

## Step 6: Moving a Folder

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of directory versioning.



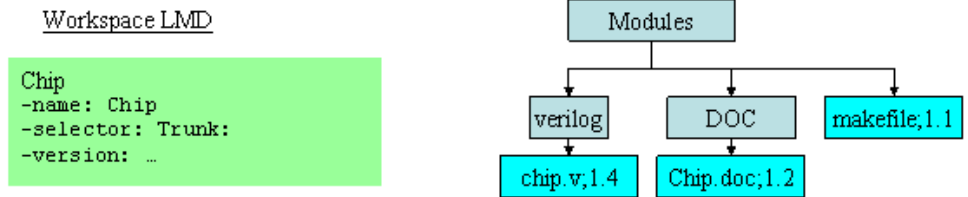
Step 6 of 6

## Operating with Module Data

### Operating on a Module

#### Step 1: Operating on a Module

In this use case, "Workspace LMD" refers to the local metadata. Read an overview of operating on module data.



Step 1 of 8

View the next step.

## Step 2: Operating on a Module

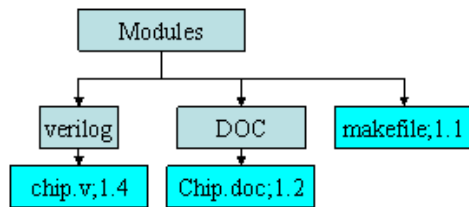
In this use case, "Workspace LMD" refers to the local metadata. Read an overview of operating on module data.



Workspace LMD

```
Chip
-name: Chip
-selector: Trunk:
-version: ...
```

```
stcl> populate Chip
...
Populated version 1.4 of module Chip
```



•Latest version of module contents populated

Step 2 of 8

View the next step.

### Step 3: Operating on a Module

In this use case, "Workspace LMD" refers to the local metadata. Read an overview of operating on module data.

Workspace LMD

```
Chip
-name: Chip
-selector: Trunk:
-version: ...
```

```
stcl> populate Chip
...
Populated version 1.4 of module Chip
stcl> ci Chip
...
Created version 1.5 of module Chip
```

```
graph TD
  Modules[Modules] --> verilog[verilog]
  Modules --> DOC[DOC]
  Modules --> makefile[makefile,1.1]
  verilog --> chip_v[chip.v,1.4]
  DOC --> chip_doc[Chip.doc,1.2]
```

- Latest version of module contents populated
- All modified members are checked in to vault
- To also search for unmanaged items:  
`stcl> ci -modulecontext Chip -rec .../Modules -new`

Step 3 of 8

View the next step.

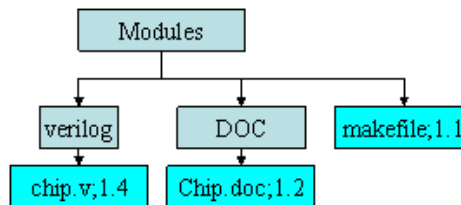
## Step 4: Operating on a Module

In this use case, "Workspace LMD" refers to the local metadata. Read an overview of operating on module data.

Workspace LMD

```
Chip
-name: Chip
-selector: Trunk:
-version: ...
```

```
stcl> populate Chip
...
Populated version 1.4 of module Chip
stcl> ci Chip
...
Created version 1.5 of module Chip
stcl> tag MyTag \
  [url vault Chip][url versionid Chip]
Tagging: Chip : Added tag MyTag to version 1.5
```



- Latest version of module contents populated

- All modified members are checked in to vault
- To also search for unmanaged items:  
`stcl> ci -modulecontext Chip -rec .../Modules -new`

- Module version is tagged

Step 4 of 8

View the next step.

**Step 5: Operating on a Module**

In this use case, "Workspace LMD" refers to the local metadata. Read an overview of operating on module data.

Workspace LMD

```
Chip
-name: Chip
-selector: Trunk:
-version: ...
```

```
stcl> populate Chip
...
Populated version 1.4 of module Chip
stcl> ci Chip
...
Created version 1.5 of module Chip
stcl> tag MyTag \
  [url vault Chip][url versionid Chip]
Tagging: Chip : Added tag MyTag to version 1.5
stcl> cancel Chip
cancelling: chip.v : Check out cancelled.
```

- Latest version of module contents populated
- All modified members are checked in to vault
- To also search for unmanaged items:  
`stcl> ci -modulecontext Chip -rec .../Modules -new`
- Module version is tagged
- All content locks in the workspace cancelled

Step 5 of 8

View the next step.

## Step 6: Operating on a Module

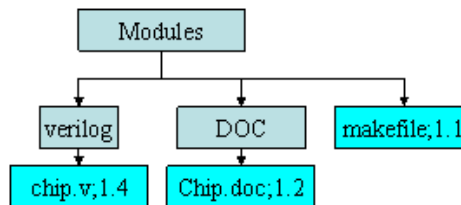
In this use case, "Workspace LMD" refers to the local metadata. Read an overview of operating on module data.

Workspace LMD

```
Chip
-name: Chip
-selector: Trunk:
-version: ...
```

```
stcl> populate Chip
...
Populated version 1.4 of module Chip
stcl> ci Chip
...
Created version 1.5 of module Chip
stcl> tag MyTag \
  [url vault Chip][url versionid Chip]
Tagging: Chip : Added tag MyTag to version 1.5
stcl> cancel Chip
cancelling: chip.v : Check out cancelled.

stcl> hcm showstatus Chip
Status of module Chip:
...
```



- Latest version of module contents populated

- All modified members are checked in to vault
- To also search for unmanaged items:  
`stcl> ci -modulecontext Chip -rec .../Modules -new`

- Module version is tagged

- All content locks in the workspace cancelled

- Status of module in workspace displayed

Step 6 of 8

View the next step.

**Step 7: Operating on a Module**

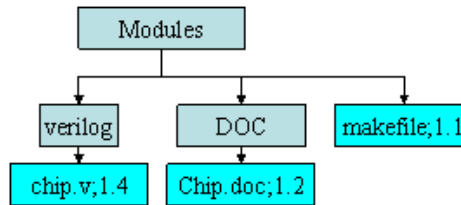
In this use case, "Workspace LMD" refers to the local metadata. Read an overview of operating on module data.

## Workspace LMD

```
Chip
-name: Chip
-selector: Trunk:
-version: ...
```

```
stcl> populate Chip
...
Populated version 1.4 of module Chip
stcl> ci Chip
...
Created version 1.5 of module Chip
stcl> tag MyTag \
  [url vault Chip][url versionid Chip]
Tagging: Chip : Added tag MyTag to version 1.5
stcl> cancel Chip
cancelling: chip.v : Check out cancelled.

stcl> hcm showstatus Chip
Status of module Chip:
...
stcl> url versionid Chip
1.5
```



- Latest version of module contents populated
- All modified members are checked in to vault
- To also search for unmanaged items:  
`stcl> ci -modulecontext Chip -rec .../Modules -new`
- Module version is tagged
- All content locks in the workspace cancelled
- Status of module in workspace displayed
- url commands run on modules return module specific information

Step 7 of 8

View the next step.

### Step 8: Operating on a Module

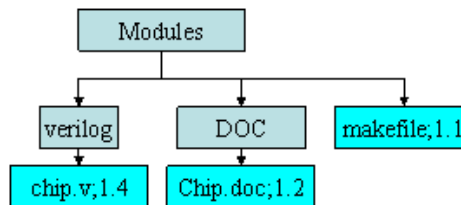
In this use case, "Workspace LMD" refers to the local metadata. Read an overview of operating on module data.

Workspace LMD

```
Chip
-name: Chip
-selector: Trunk:
-version: ...
```

```
stcl> populate Chip
...
Populated version 1.4 of module Chip
stcl> ci Chip
...
Created version 1.5 of module Chip
stcl> tag MyTag \
  [url vault Chip][url versionid Chip]
Tagging: Chip : Added tag MyTag to version 1.5
stcl> cancel Chip
cancelling: chip.v : Check out cancelled.

stcl> hcm showstatus Chip
Status of module Chip:
...
stcl> url versionid Chip
1.5
stcl> co Chip
Error: co operation not available on modules
```



- Latest version of module contents populated

- All modified members are checked in to vault
- To also search for unmanaged items:  
`stcl> ci -modulecontext Chip -rec .../Modules -new`

- Module version is tagged

- All content locks in the workspace cancelled

- Status of module in workspace displayed

- url commands run on modules return module specific information

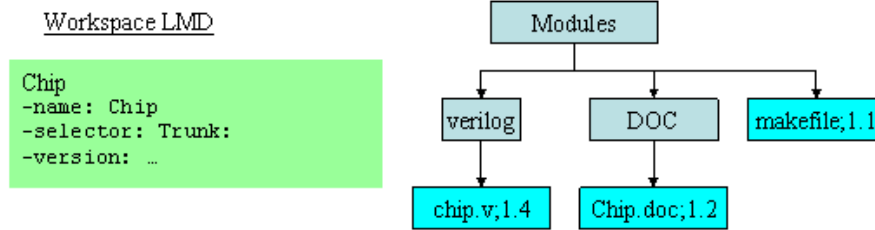
- co not allowed on modules

Step 8 of 8

## Operating on a Module's Contents

### Step 1: Operating on a Module's Contents

In this use case, "Workspace LMD" refers to the local metadata. Read an overview of operating on module data.



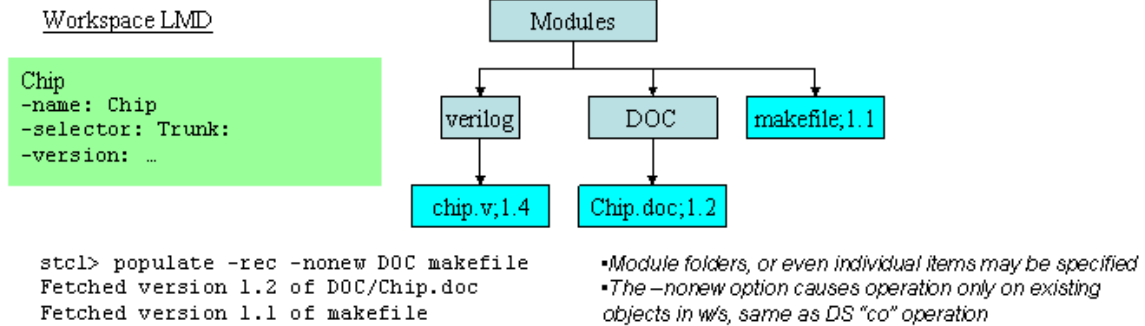
Step 1 of 7

View the next step.

## Step 2: Operating on a Module's Contents

In this use case, "Workspace LMD" refers to the local metadata. Read an overview of operating on module data.



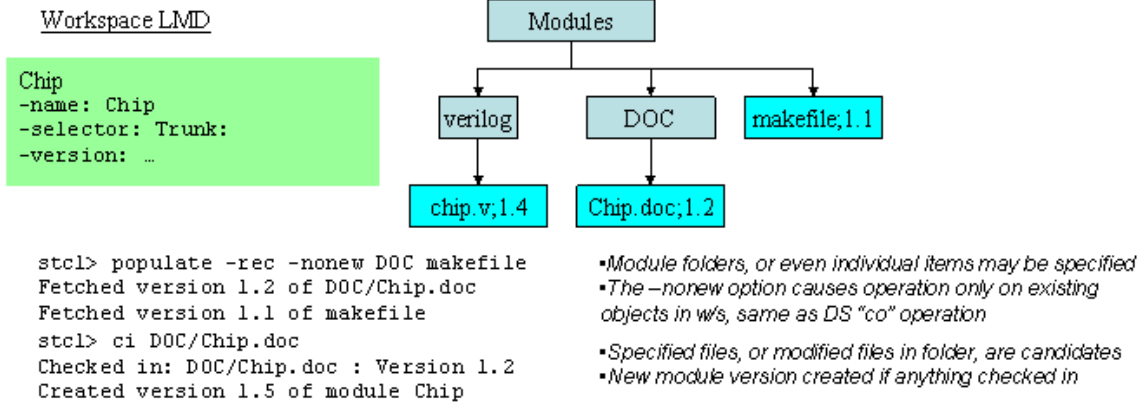


Step 2 of 7

View the next step.

### Step 3: Operating on a Module's Contents

In this use case, "Workspace LMD" refers to the local metadata. Read an overview of operating on module data.

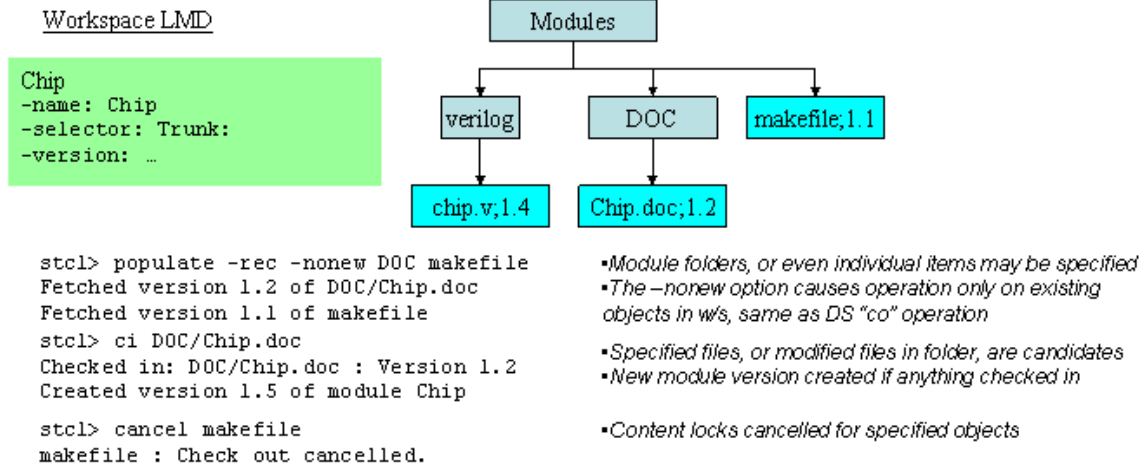


Step 3 of 7

View the next step.

## Step 4: Operating on a Module's Contents

In this use case, "Workspace LMD" refers to the local metadata. Read an overview of operating on module data.



Step 4 of 7

View the next step.

### Step 5: Operating on a Module's Contents

In this use case, "Workspace LMD" refers to the local metadata. Read an overview of operating on module data.

Workspace LMD

```
Chip
-name: Chip
-selector: Trunk:
-version: ...
```

```
graph TD
  Modules --> verilog
  Modules --> DOC
  Modules --> makefile_1.1[makefile,1.1]
  verilog --> chip_v_1.4[chip.v,1.4]
  DOC --> Chip_doc_1.2[Chip.doc,1.2]
```

```
stcl> populate -rec -nonew DOC makefile
  Fetched version 1.2 of DOC/Chip.doc
  Fetched version 1.1 of makefile
  stcl> ci DOC/Chip.doc
  Checked in: DOC/Chip.doc : Version 1.2
  Created version 1.5 of module Chip

  stcl> cancel makefile
  makefile : Check out cancelled.

  stcl> ls -rec verilog
  ...
```

- Module folders, or even individual items may be specified
- The `-nonew` option causes operation only on existing objects in w/s, same as DS "co" operation
- Specified files, or modified files in folder, are candidates
- New module version created if anything checked in
- Content locks cancelled for specified objects
- `ls` command run on folders/objects
- `ls` command can be run on modules, but does not report them as individual items in the output

Step 5 of 7

View the next step.

## Step 6: Operating on a Module's Contents

In this use case, "Workspace LMD" refers to the local metadata. Read an overview of operating on module data.

Workspace LMD

```
Chip
-name: Chip
-selector: Trunk:
-version: ...
```

```

graph TD
    Modules[Modules] --> verilog[verilog]
    Modules --> DOC[DOC]
    Modules --> makefile["makefile,1.1"]
    verilog --> chip_v["chip.v,1.4"]
    DOC --> chip_doc["Chip.doc,1.2"]
  
```

```

stcl> populate -rec -nonew DOC makefile
  Fetched version 1.2 of DOC/Chip.doc
  Fetched version 1.1 of makefile
stcl> ci DOC/Chip.doc
  Checked in: DOC/Chip.doc : Version 1.2
  Created version 1.5 of module Chip

stcl> cancel makefile
makefile : Check out cancelled.

stcl> ls -rec verilog
...
stcl> url versionid makefile
1.1
  
```

- Module folders, or even individual items may be specified
- The `-nonew` option causes operation only on existing objects in w/s, same as DS "co" operation
- Specified files, or modified files in folder, are candidates
- New module version created if anything checked in
- Content locks cancelled for specified objects
- `ls` command run on folders/objects
- `ls` command can be run on modules, but does not report them as individual items in the output
- `url` commands run on members generally return member specific information

Step 6 of 7

View the next step.

### Step 7: Operating on a Module's Contents

In this use case, "Workspace LMD" refers to the local metadata. Read an overview of operating on module data.

Workspace LMD

```
Chip
-name: Chip
-selector: Trunk:
-version: ...
```

```
graph TD
    Modules[Modules] --> verilog[verilog]
    Modules --> DOC[DOC]
    Modules --> makefile["makefile,1.1"]
    verilog --> chip_v["chip_v,1.4"]
    DOC --> chip_doc["Chip.doc,1.2"]
```

```
stcl> populate -rec -nonew DOC makefile
  Fetched version 1.2 of DOC/Chip.doc
  Fetched version 1.1 of makefile
stcl> ci DOC/Chip.doc
  Checked in: DOC/Chip.doc : Version 1.2
  Created version 1.5 of module Chip

stcl> cancel makefile
makefile : Check out cancelled.

stcl> ls -rec verilog
...
stcl> url versionid makefile
1.1

stcl> co makefile
Error: co operation not available on modules
```

- Module folders, or even individual items may be specified
- The `-nonew` option causes operation only on existing objects in w/s, same as DS "co" operation
- Specified files, or modified files in folder, are candidates
- New module version created if anything checked in
- Content locks cancelled for specified objects
- ls command run on folders/objects
- ls command can be run on modules, but does not report them as individual items in the output
- url commands run on members generally return member specific information
- co not allowed on module members

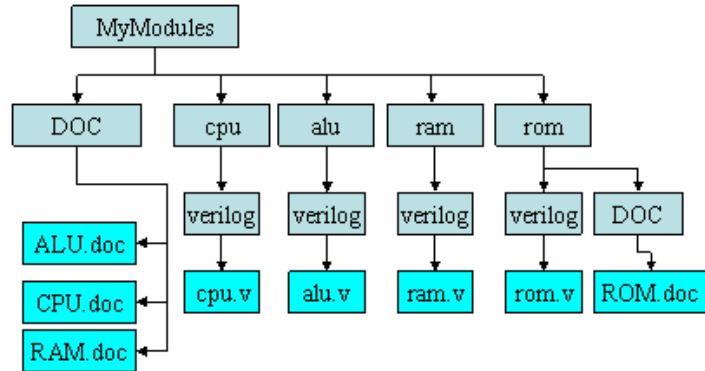
Step 7 of 7

## Filtering

### Step 1: Filtering

Read an overview of filtering module data.

Without any filtering, whole module hierarchy



Step 1 of 12

View the next step.

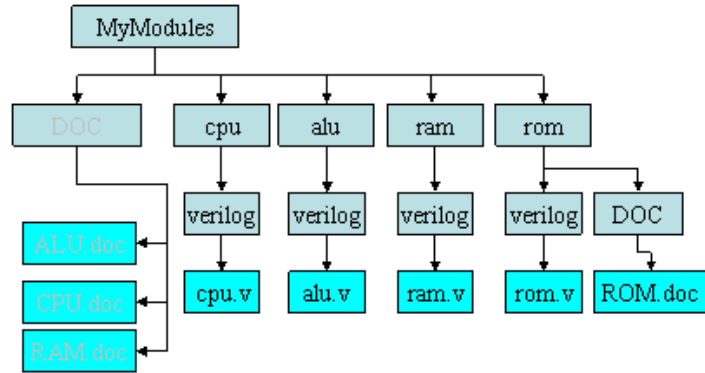
### Step 2: Filtering

Read an overview of filtering module data.

# DesignSync Data Manager User's Guide

Without any filtering, whole module hierarchy

Filter the "DOC" dir, but only the top level:  
`stcl> ... -filter DOC`  
Whole contents are filtered, but `./rom/DOC` dir remains.



Step 2 of 12

View the next step.

## Step 3: Filtering

Read an overview of filtering module data.

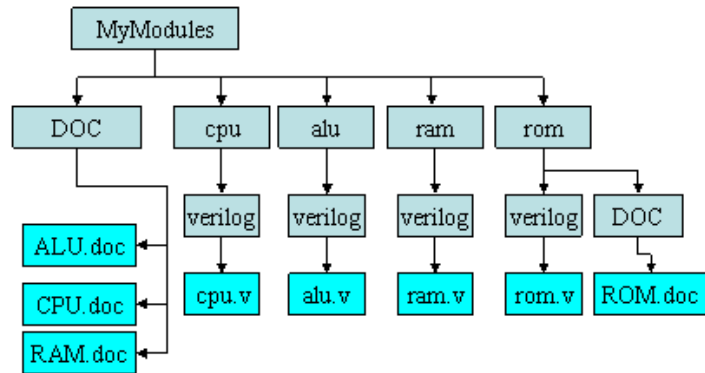


Without any filtering, whole module hierarchy

Filter the "DOC" dir, but only the top level:

```
stcl> ... -filter DOC
```

Whole contents are filtered, but ./rom/DOC dir remains.



→ No "-" needed at front, filters exclude by default.

Step 3 of 12

View the next step.

#### Step 4: Filtering

Read an overview of filtering module data.

# DesignSync Data Manager User's Guide

Without any filtering, whole module hierarchy

Filter the "DOC" dir, but only the top level:

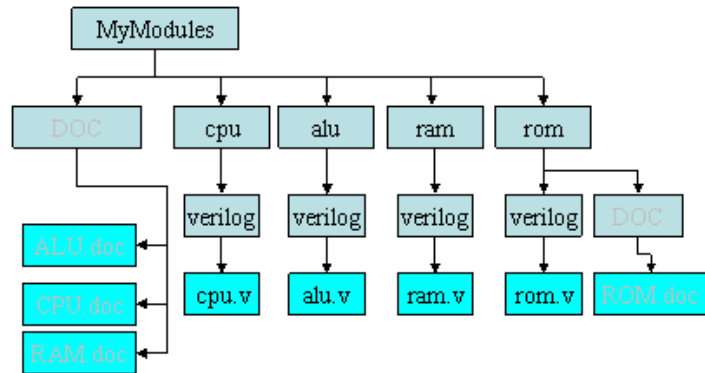
```
stcl> ... -filter DOC
```

Whole contents are filtered, but ./rom/DOC dir remains.

Filter the "DOC" dir, at all levels:

```
stcl> ... -filter .../DOC
```

Whole contents are filtered, with "..." matching any number of dir levels.



• No "-" needed at front, filters exclude by default.

Step 4 of 12

View the next step.

## Step 5: Filtering

Read an overview of filtering module data.

Without any filtering, whole module hierarchy

Filter the "DOC" dir, but only the top level:

```
stcl> ... -filter DOC
```

Whole contents are filtered, but ./rom/DOC dir remains.

Filter the "DOC" dir, at all levels:

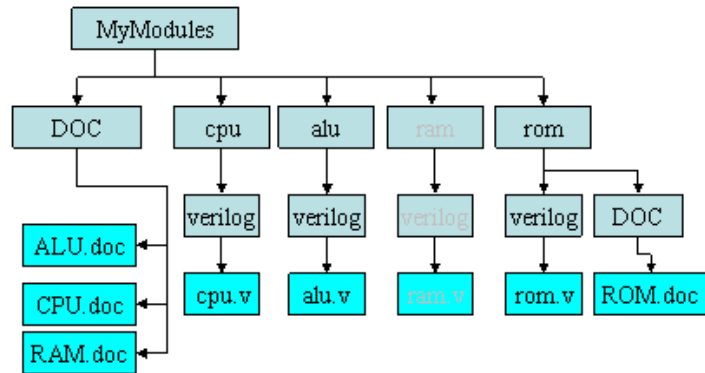
```
stcl> ... -filter ../DOC
```

Whole contents are filtered, with "..." matching any number of dir levels.

Filter the ram directory:

```
stcl> ... -filter ram
```

Contents of ram dir filtered, but not the doc file



• No "-" needed at front, filters exclude by default.

Step 5 of 12

View the next step.

### Step 6: Filtering

Read an overview of filtering module data.

# DesignSync Data Manager User's Guide

Without any filtering, whole module hierarchy

Filter the "DOC" dir, but only the top level:

```
stcl> ... -filter DOC
```

Whole contents are filtered, but ./rom/DOC dir remains.

Filter the "DOC" dir, at all levels:

```
stcl> ... -filter ../DOC
```

Whole contents are filtered, with "..." matching any number of dir levels.

Filter the ram directory:

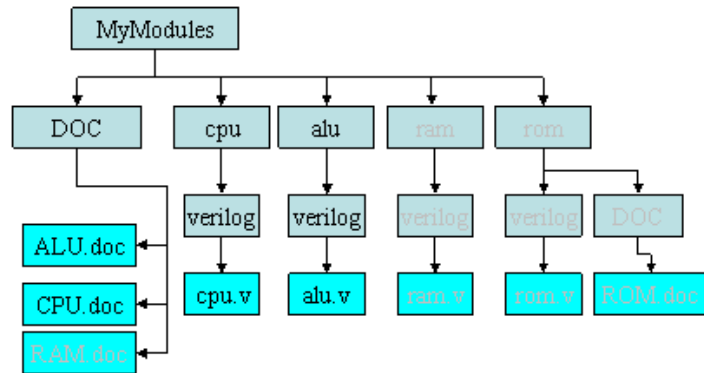
```
stcl> ... -filter ram
```

Contents of ram dir filtered, but not the doc file

Filter the RAM and ROM hrefs:

```
stcl> ... -hreffilter R*
```

Whole of RAM/ROM modules filtered, wherever the href is in the module hierarchy



• No "-" needed at front, filters exclude by default.

Step 6 of 12

View the next step.

## Step 7: Filtering

Read an overview of filtering module data.

Without any filtering, whole module hierarchy

Filter the "DOC" dir, but only the top level:

```
stcl> ... -filter DOC
```

Whole contents are filtered, but ./rom/DOC dir remains.

Filter the "DOC" dir, at all levels:

```
stcl> ... -filter ../DOC
```

Whole contents are filtered, with "..." matching any number of dir levels.

Filter the ram directory:

```
stcl> ... -filter ram
```

Contents of ram dir filtered, but not the doc file

Filter the RAM and ROM hrefs:

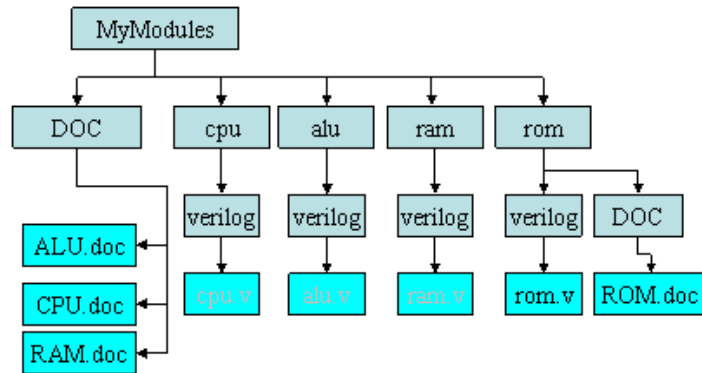
```
stcl> ... -hreffilter R*
```

Whole of RAM/ROM modules filtered, wherever the href is in the module hierarchy

Filter the verilog files, except the rom.v:

```
stcl> ... -filter ../*.v,+../r*.v,-../ram.v
```

All .v files removed, then all r\*.v files added back, then ram.v removed



• No "-" needed at front, filters exclude by default.

Step 7 of 12

View the next step.

### Step 8: Filtering

Read an overview of filtering module data.

Without any filtering, whole module hierarchy

Filter the "DOC" dir, but only the top level:

```
stcl> ... -filter DOC
```

Whole contents are filtered, but ./rom/DOC dir remains.

Filter the "DOC" dir, at all levels:

```
stcl> ... -filter ../DOC
```

Whole contents are filtered, with "..." matching any number of dir levels.

Filter the ram directory:

```
stcl> ... -filter ram
```

Contents of ram dir filtered, but not the doc file

Filter the RAM and ROM hrefs:

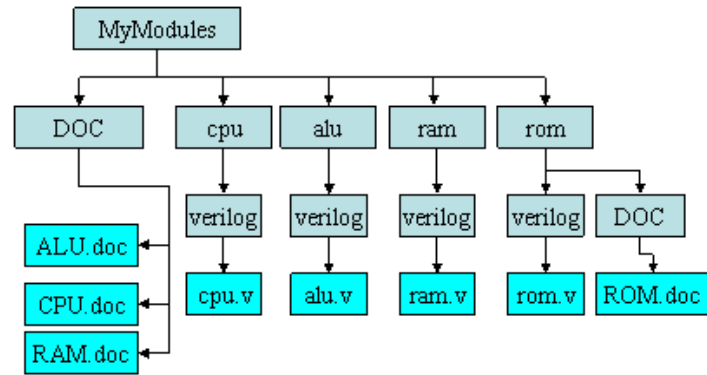
```
stcl> ... -hreffilter R*
```

Whole of RAM/ROM modules filtered, wherever the href is in the module hierarchy

Filter the verilog files, except the rom.v:

```
stcl> ... -filter ../*.v,+../r*.v ../ram.v
```

All .v files removed, then all r\*.v files added back, then ram.v removed



- No "-" needed at front, filters exclude by default.
- "-" shown here just for clarity

Step 8 of 12

View the next step.

## Step 9: Filtering

Read an overview of filtering module data.

Without any filtering, whole module hierarchy

Filter the "DOC" dir, but only the top level:

```
stcl> ... -filter DOC
```

Whole contents are filtered, but ./rom/DOC dir remains.

Filter the "DOC" dir, at all levels:

```
stcl> ... -filter .../DOC
```

Whole contents are filtered, with "..." matching any number of dir levels.

Filter the ram directory:

```
stcl> ... -filter ram
```

Contents of ram dir filtered, but not the doc file

Filter the RAM and ROM hrefs:

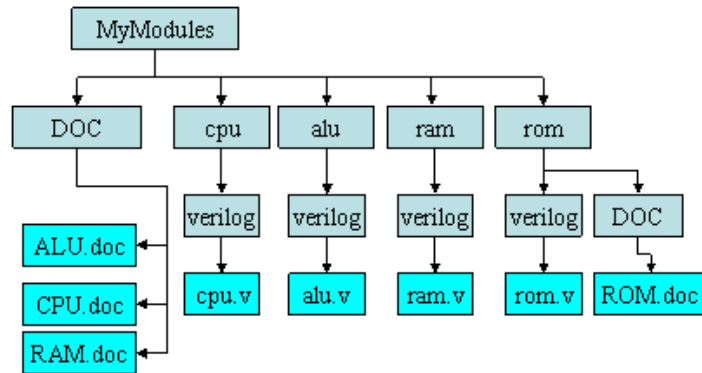
```
stcl> ... -hreffilter R*
```

Whole of RAM/ROM modules filtered, wherever the href is in the module hierarchy

Filter the verilog files, except the rom.v:

```
stcl> ... -filter .../*.v,+.../r*.v,-.../ram.v
```

All .v files removed, then all r\*.v files added back, then ram.v removed



- No "-" needed at front, filters exclude by default.
- "-" shown here just for clarity
- No paths allowed in href filters – all or nothing

Step 9 of 12

View the next step.

## Step 10: Filtering

Read an overview of filtering module data.

Without any filtering, whole module hierarchy

Filter the "DOC" dir, but only the top level:

```
stcl> ... -filter DOC
```

Whole contents are filtered, but ./rom/DOC dir remains.

Filter the "DOC" dir, at all levels:

```
stcl> ... -filter ../DOC
```

Whole contents are filtered, with "..." matching any number of dir levels.

Filter the ram directory:

```
stcl> ... -filter ram
```

Contents of ram dir filtered, but not the doc file

Filter the RAM and ROM hrefs:

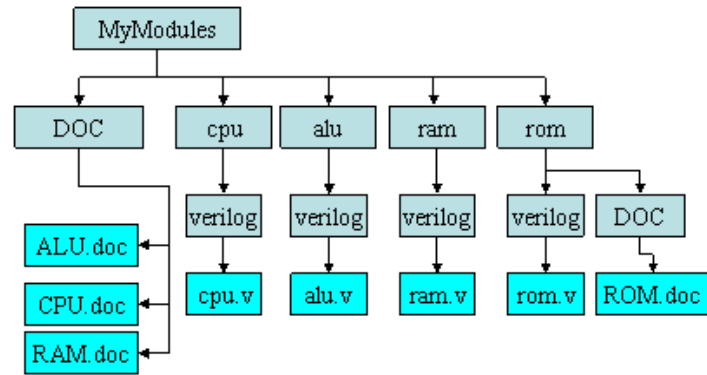
```
stcl> ... -hreffilter R*
```

Whole of RAM/ROM modules filtered, wherever the href is in the module hierarchy

Filter the verilog files, except the rom.v:

```
stcl> ... -filter ../*.v,+../r*.v,-../ram.v
```

All .v files removed, then all r\*.v files added back, then ram.v removed



- No "-" needed at front, filters exclude by default
- "-" shown here just for clarity
- No paths allowed in href filters – all or nothing
- Hrefs can only be excluded NOT included

Step 10 of 12

View the next step.

## Step 11: Filtering

Read an overview of filtering module data.



Without any filtering, whole module hierarchy

Filter the "DOC" dir, but only the top level:

```
stcl> ... -filter DOC
```

Whole contents are filtered, but ./rom/DOC dir remains.

Filter the "DOC" dir, at all levels:

```
stcl> ... -filter .../DOC
```

Whole contents are filtered, with "..." matching any number of dir levels.

Filter the ram directory:

```
stcl> ... -filter ram
```

Contents of ram dir filtered, but not the doc file

Filter the RAM and ROM hrefs:

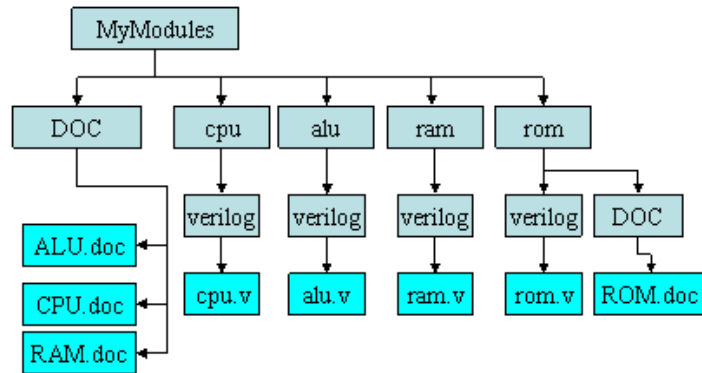
```
stcl> ... -hreffilter R*
```

Whole of RAM/ROM modules filtered, wherever the href is in the module hierarchy

Filter the verilog files, except the rom.v:

```
stcl> ... -filter .../*.v,+.../r*.v,-.../ram.v
```

All .v files removed, then all r\*.v files added back, then ram.v removed



- No "-" needed at front, filters exclude by default
- "-" shown here just for clarity
- No paths allowed in href filters – all or nothing
- Hrefs can only be excluded NOT included
- A different switch for hrefs prevents any clashing of names

Step 11 of 12

View the next step.

## Step 12: Filtering

Read an overview of filtering module data.

Without any filtering, whole module hierarchy

Filter the "DOC" dir, but only the top level:

```
stcl> ... -filter DOC
```

Whole contents are filtered, but ./rom/DOC dir remains.

Filter the "DOC" dir, at all levels:

```
stcl> ... -filter ../DOC
```

Whole contents are filtered, with "..." matching any number of dir levels.

Filter the ram directory:

```
stcl> ... -filter ram
```

Contents of ram dir filtered, but not the doc file

Filter the RAM and ROM hrefs:

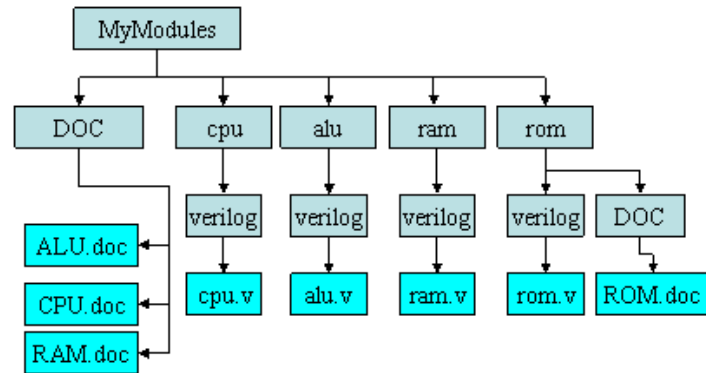
```
stcl> ... -hreffilter R*
```

Whole of RAM/ROM modules filtered, wherever the href is in the module hierarchy

Filter the verilog files, except the rom.v:

```
stcl> ... -filter ../*.v,+../r*.v,-../ram.v
```

All .v files removed, then all r\*.v files added back, then ram.v removed



- No "-" needed at front, filters exclude by default.
- "-" shown here just for clarity
- No paths allowed in href filters – all or nothing
- Hrefs can only be excluded NOT included
- A different switch for hrefs prevents any clashing of names
- Populate a sub-module by addressing on the command line:  
*populate -modulecontext Chip CPU*

Step 12 of 12

## Persistent Populate Filter

### Step 1: Persistent Populate Filter

In this use case, "Workspace LMD" refers to the local metadata. Read an overview of filtering module data.

MyModules

Step 1 of 7

View the next step.

**Step 2: Persistent Populate Filter**

In this use case, "Workspace LMD" refers to the local metadata. Read an overview of filtering module data.

MyModules

Specify a filter with an initial populate:

Step 2 of 7

[View the next step.](#)

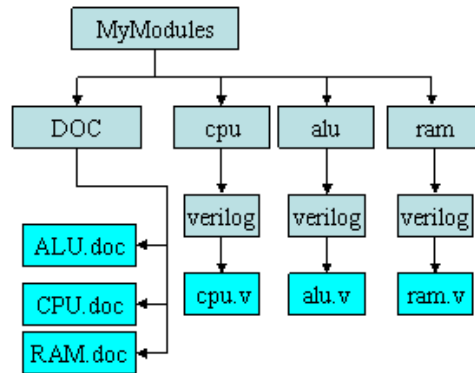
### **Step 3: Persistent Populate Filter**

In this use case, "Workspace LMD" refers to the local metadata. Read an overview of filtering module data.

Workspace LMD

```
Chip
-name: Chip
-hrefilter: ROM
```

```
Specify a filter with an initial populate:
stcl> populate -recursive
-hrefilter ROM \
sync://h:p/Modules/Chip
The ROM is not populated. The filter is stored.
```



**Exact equivalent exists for -filter**

Step 3 of 7

View the next step.

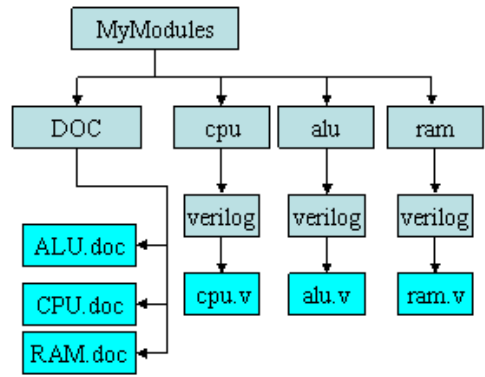
#### Step 4: Persistent Populate Filter

In this use case, "Workspace LMD" refers to the local metadata. Read an overview of filtering module data.

Workspace LMD

```
Chip  
-name: Chip  
-hreffilter: ROM
```

```
Specify a filter with an initial populate:  
stcl> populate -recursive  
-hreffilter ROM \  
sync://h:p/Modules/Chip  
The ROM is not populated. The filter is stored.  
A second populate can filter out some more, but  
the initial filter applies on top:
```



**Exact equivalent exists for -filter**

Step 4 of 7

View the next step.

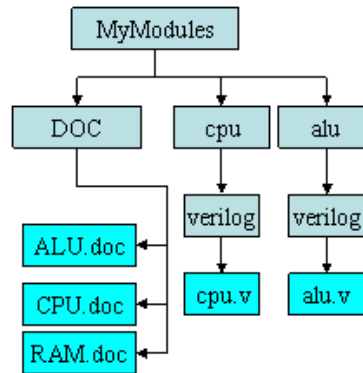
**Step 5: Persistent Populate Filter**

In this use case, "Workspace LMD" refers to the local metadata. Read an overview of filtering module data.

Workspace LMD

```
Chip
-name: Chip
-hrefilter: ROM
```

```
Specify a filter with an initial populate:
stcl> populate -recursive
-hrefilter ROM \
sync://h:p/Modules/Chip
The ROM is not populated. The filter is stored.
A second populate can filter out some more, but
the initial filter applies on top:
stcl> populate -hrefilter RAM Chip
RAM filtered in addition to ROM. New filter not perpetuated
```



Exact equivalent exists for -filter

Step 5 of 7

View the next step.

### Step 6: Persistent Populate Filter

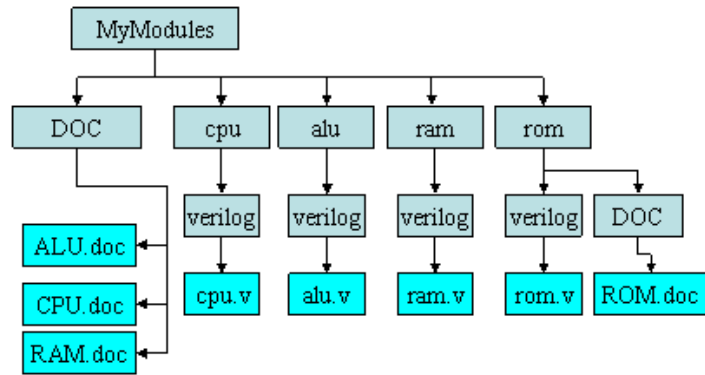
In this use case, "Workspace LMD" refers to the local metadata. Read an overview of filtering module data.

Workspace LMD

```
Chip
-name: Chip
-hrefilter: RAM
```

```
Specify a filter with an initial populate:
stcl> populate -recursive
-hrefilter ROM \
sync://h:p/Modules/Chip
The ROM is not populated. The filter is stored.
A second populate can filter out some more, but
the initial filter applies on top:
stcl> populate -hrefilter RAM Chip
RAM filtered in addition to ROM. New filter not perpetuated

Filter can be displayed, and changed:
stcl> url filter -hrefilter Chip
ROM
stcl> setfilter -hrefilter Chip RAM
Persistent filter changed
```



Exact equivalent exists for -filter

Step 6 of 7

View the next step.

**Step 7: Persistent Populate Filter**

In this use case, "Workspace LMD" refers to the local metadata. Read an overview of filtering module data.



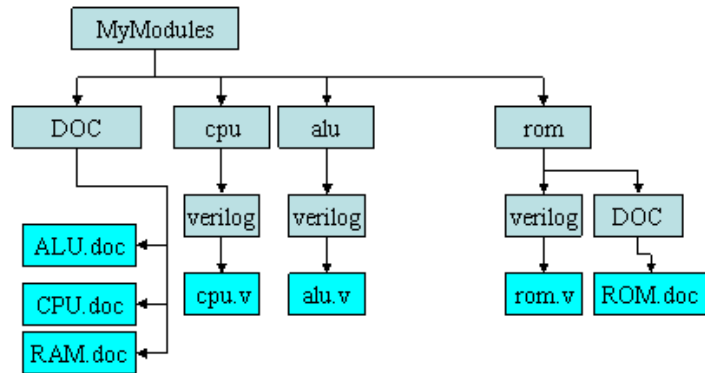
Workspace LMD

```
Chip
-name: Chip
-hrefilter: RAM
```

```
Specify a filter with an initial populate:
stcl> populate -recursive
-hrefilter ROM \
sync://h:p/Modules/Chip
The ROM is not populated. The filter is stored.
A second populate can filter out some more, but
the initial filter applies on top:
stcl> populate -hrefilter RAM Chip
RAM filtered in addition to ROM. New filter not perpetuated

Filter can be displayed, and changed:
stcl> url filter -hrefilter Chip
ROM
stcl> setfilter -hrefilter Chip RAM
Persistent filter changed

Subsequent populate now still filters RAM, but the ROM
is now fetched:
stcl> populate -recursive Chip
```



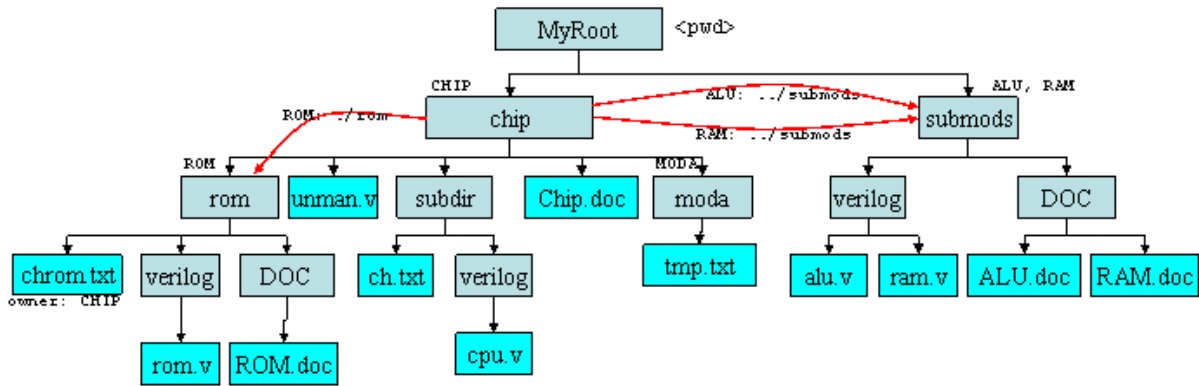
Exact equivalent exists for -filter

Step 7 of 7

## Folder-Centric Operations

### Step 1: Folder-Centric Operations

Read an overview of module recursion.

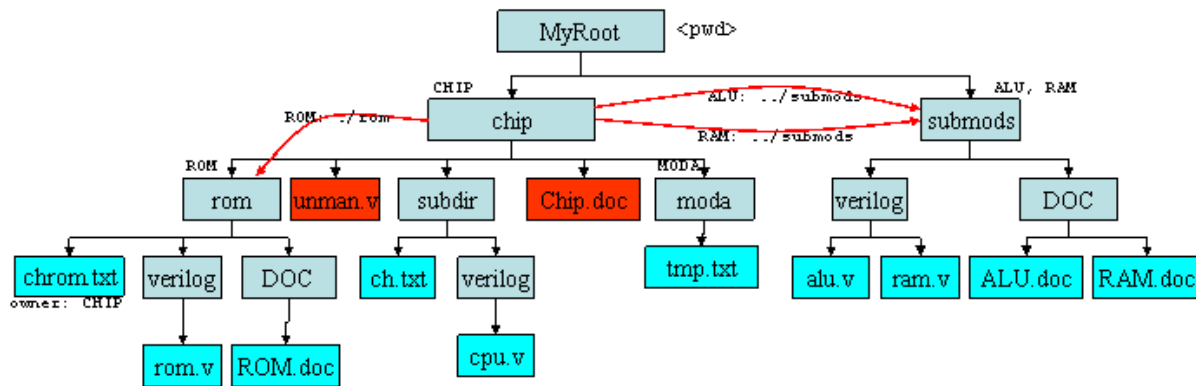


Step 1 of 5

View the next step.

### Step 2: Folder-Centric Operations

Read an overview of module recursion.



<operation> [-norecursive] chip

- Default is -norecursive

- Just the files in this dir -

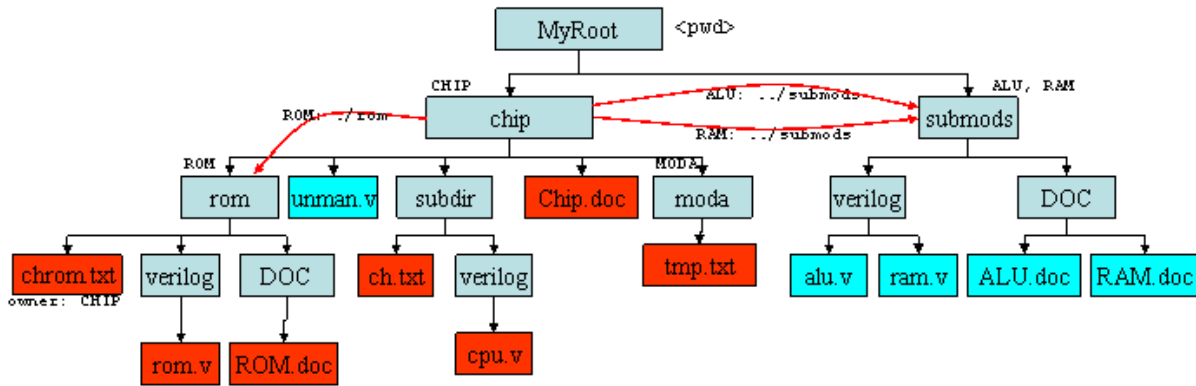
Note: The *ci* command does not operate non-recursively on a folder. See the *ci* command help for details.

Step 2 of 5

View the next step.

### Step 3: Folder-Centric Operations

Read an overview of module recursion.



```
<operation> [-norecursive] chip
•Default is -norecursive
•Just the files in this dir –
  Note: The ci command does not operate non-recursively
  on a folder. See the ci command help for details.
```

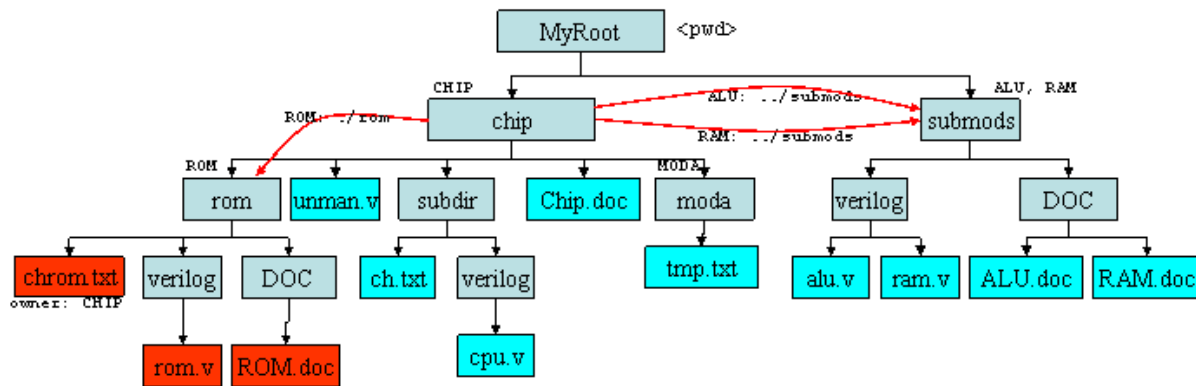
```
<operation> -recursive chip
•This folder, all sub-dirs, ignores module boundaries,
since is a folder operation, so works on all sub-mods
with data placed under here
```

Step 3 of 5

View the next step.

#### Step 4: Folder-Centric Operations

Read an overview of module recursion.



<operation> [-norecursive] chip

- Default is `-norecursive`

- Just the files in this dir –

Note: The `ci` command does not operate non-recursively on a folder. See the `ci` command help for details.

<operation> `-recursive` chip

- This folder, all sub-dirs, ignores module boundaries, since is a folder operation, so works on all sub-mods with data placed under here

<operation> `-recursive` chip/rom

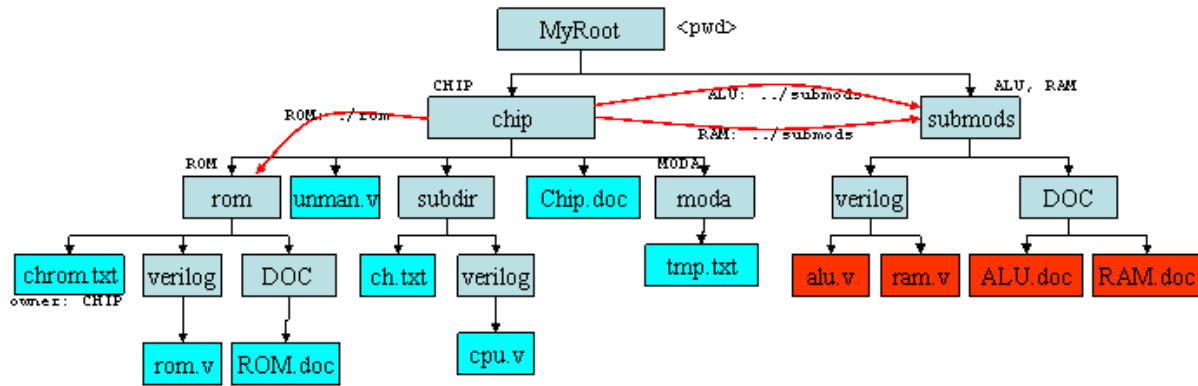
- This folder, all sub-dirs, so picks up `chrom.txt`, so module effectively ignored

Step 4 of 5

View the next step.

### Step 5: Folder-Centric Operations

Read an overview of module recursion.



<operation> [-norecursive] chip  
 •Default is -norecursive  
 •Just the files in this dir –  
 Note: The *ci* command does not operate non-recursively on a folder. See the *ci* command help for details.

<operation> -recursive chip  
 •This folder, all sub-dirs, ignores module boundaries, since is a folder operation, so works on all sub-mods with data placed under here

<operation> -recursive chip/rom  
 •This folder, all sub-dirs, so picks up chrom.txt, so module effectively ignored

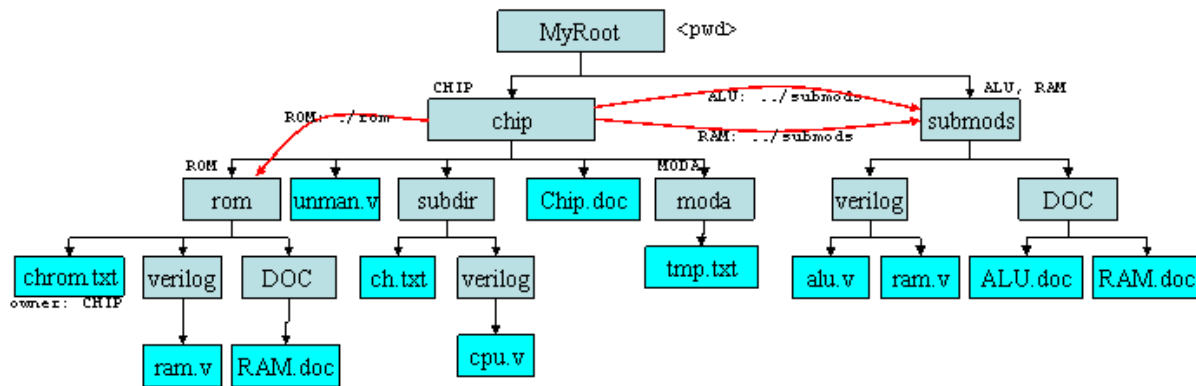
<operation> -recursive submods  
 •This folder, all sub-dirs, so simply picks up everything

Step 5 of 5

## Module-Centric Operations on a Module

### Step 1: Module-Centric Operations on a Module

Read an overview of module recursion.

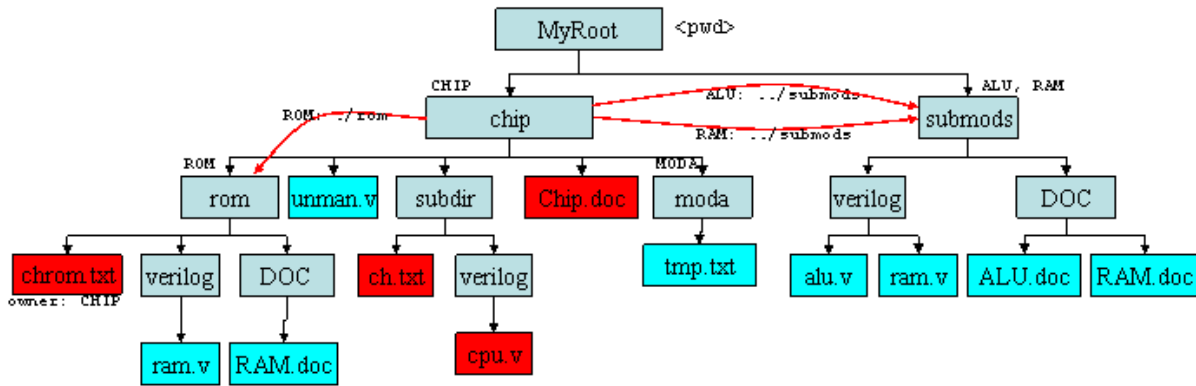


Step 1 of 3

View the next step.

## Step 2: Module-Centric Operations on a Module

Read an overview of module recursion.



<operation> [-norecursive] CHIP

**-Default is -norecursive, operates on all module members**

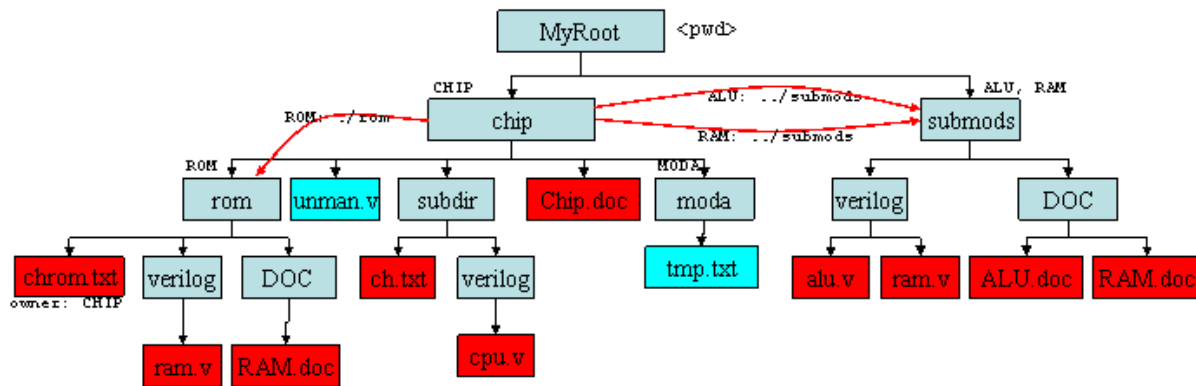
Step 2 of 3

View the next step.

### Step 3: Module-Centric Operations on a Module

Read an overview of module recursion.





<operation> [-norecursive] CHIP

**-Default is `-norecursive`, operates on all module members**

<operation> -recursive CHIP

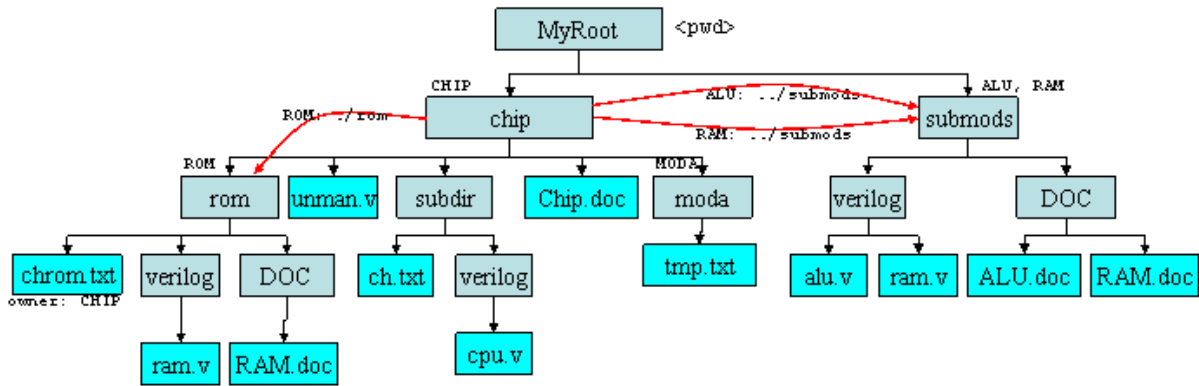
**-Operates across ALL sub-modules**

Step 3 of 3

## Module-Centric Operations on a Subfolder

### Step 1: Module-Centric Operations on a Subfolder

Read an overview of module recursion.

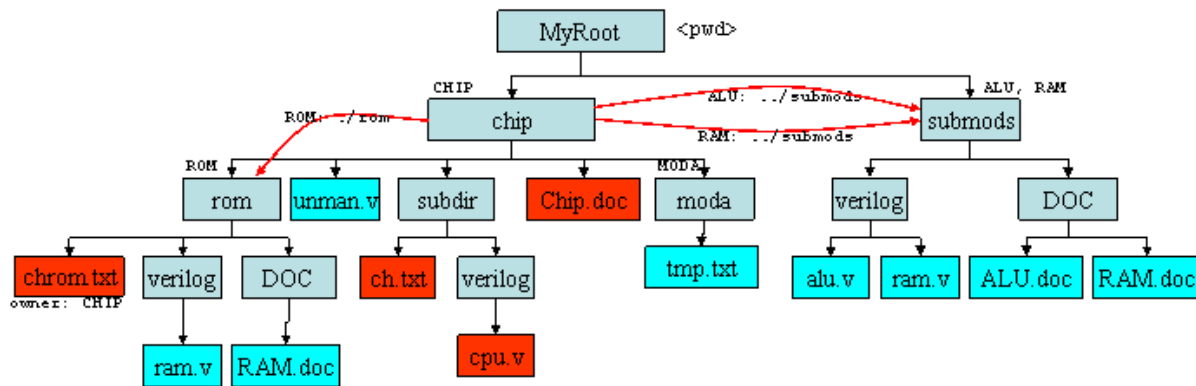


Step 1 of 5

View the next step.

### Step 2: Module-Centric Operations on a Subfolder

Read an overview of module recursion.



```

<operation> -modulecontext CHIP [-norecursive] chip
•Default is -norecursive, argument is a module, filtered
by the specified folder
•All files anywhere under /MyRoot/chip that belong to CHIP

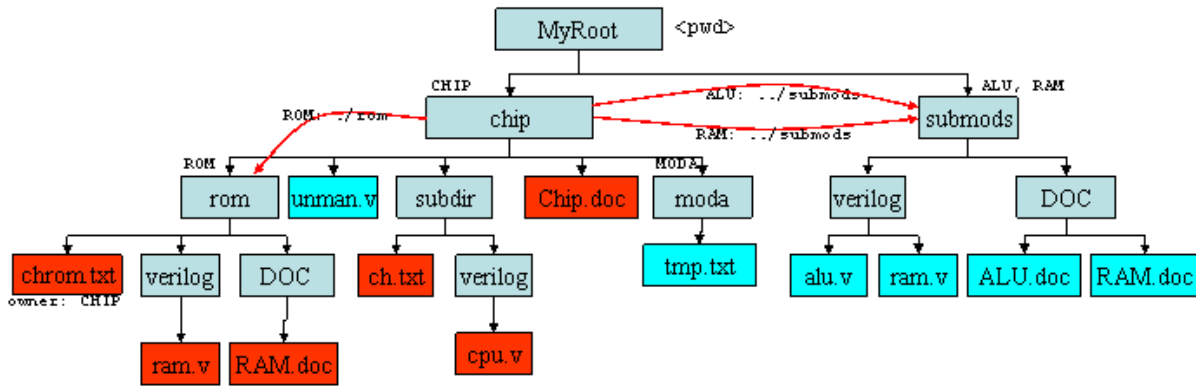
```

Step 2 of 5

View the next step.

### Step 3: Module Centric Operations on a Subfolder

Read an overview of module recursion.



```

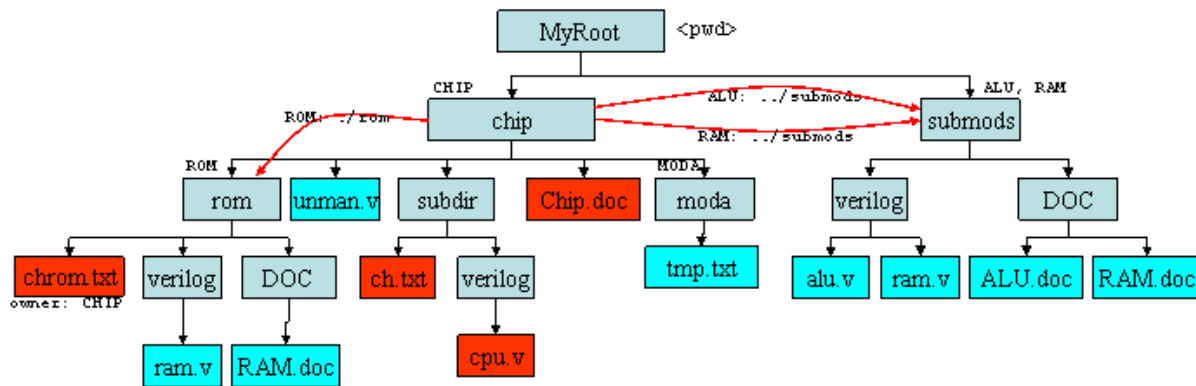
<operation> -modulecontext CHIP [-norecursive] chip
•Default is -norecursive, argument is a module, filtered
by the specified folder
•All files anywhere under /MyRoot/chip that belong to CHIP
<operation> -modulecontext CHIP -recursive chip
•All files under CHIP and its submodules, which are placed
under /MyRoot/chip
    
```

Step 3 of 5

View the next step.

#### Step 4: Module-Centric Operations on a Subfolder

Read an overview of module recursion.



```

<operation> -modulecontext CHIP [-norecursive] chip
•Default is -norecursive, argument is a module, filtered
by the specified folder
•All files anywhere under /MyRoot/chip that belong to CHIP
<operation> -modulecontext CHIP -recursive chip
•All files under CHIP and its submodules, which are placed
under /MyRoot/chip
•To get just the files actually in /MyRoot/chip that belong to CHIP:
  <operation> -modulecontext CHIP chip -filter */**

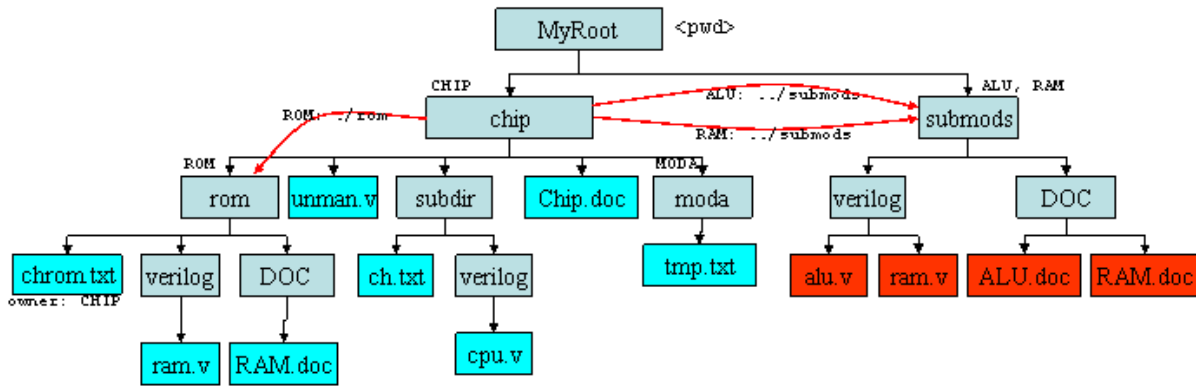
```

Step 4 of 5

View the next step.

### Step 5: Module-Centric Operations on a Subfolder

Read an overview of module recursion.



```
<operation> -modulecontext CHIP [-norecursive] chip
•Default is -norecursive, argument is a module, filtered
  by the specified folder
•All files anywhere under /MyRoot/chip that belong to CHIP
<operation> -modulecontext CHIP -recursive chip
•All files under CHIP and its submodules, which are placed
  under /MyRoot/chip
•To get just the files actually in /MyRoot/chip that belong to CHIP:
  <operation> -modulecontext CHIP chip -filter */**
```

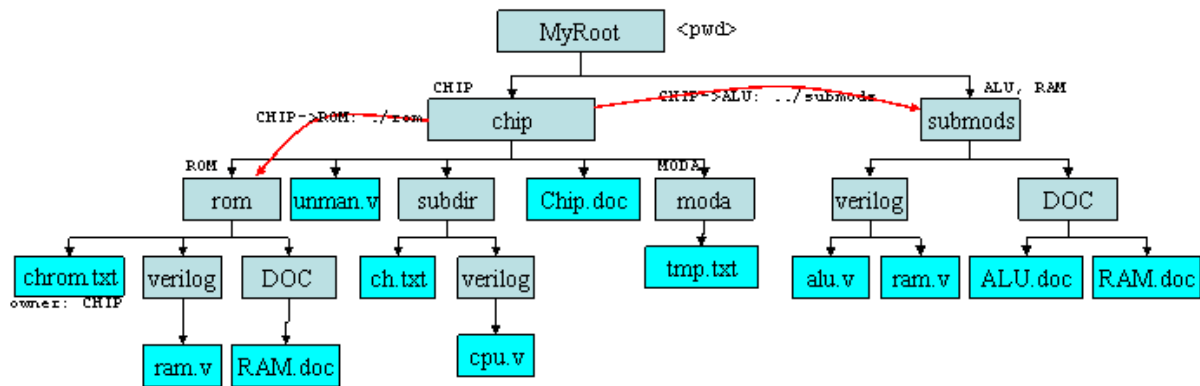
```
<operation> -modulecontext CHIP -recursive
  submods
•All files under CHIP and its submodules which are
  placed under /MyRoot/submods
```

Step 5 of 5

## Module-Centric Operations on an HREF

### Step 1: Module-Centric Operations on an HREF

Read an overview of module recursion.

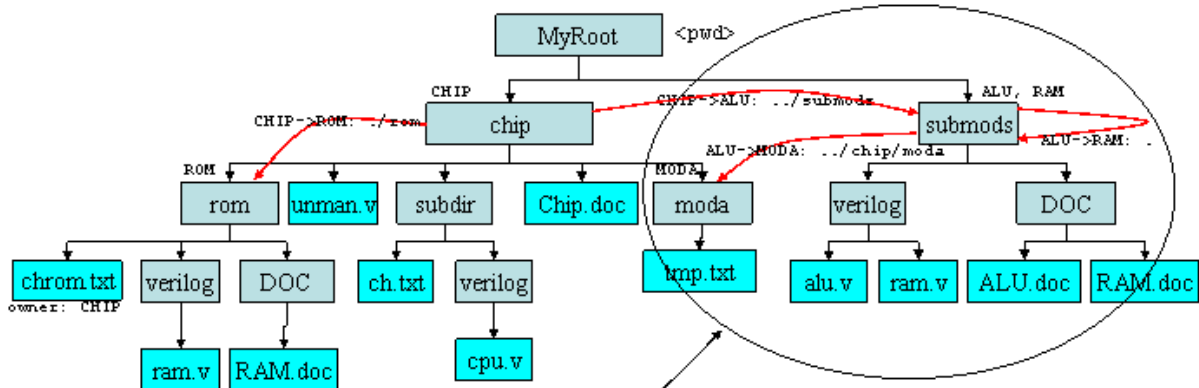


Step 1 of 5

View the next step.

## Step 2: Module-Centric Operations on an HREF

Read an overview of module recursion.



Note change of hierarchy from other examples

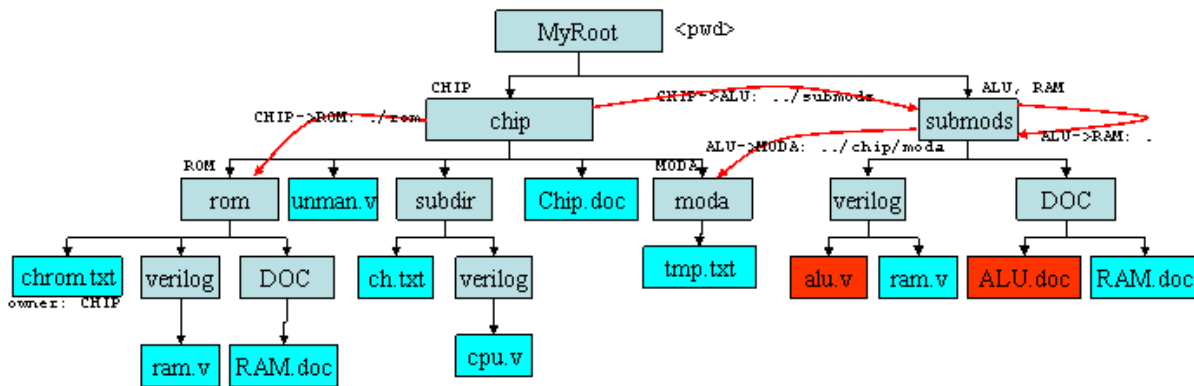
Step 2 of 5

View the next step.

### Step 3: Module-Centric Operations on an HREF

Read an overview of module recursion.





Note change of hierarchy  
from other examples

```
populate -modulecontext CHIP [-norecursive] ALU
```

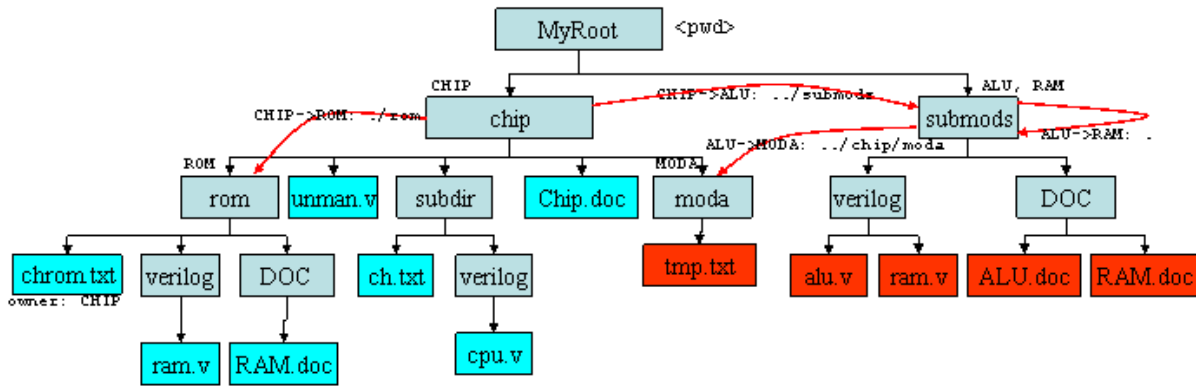
- Default is `-norecursive`, argument is a *module*
- All files in module CHIP that directly belong to submodule ALU
- Bit of a special case: although is `-norecursive`, picks up href and works on that
- `-nomodulerecursive` not appropriate switches to `-norecursive`

Step 3 of 5

View the next step.

#### Step 4: Module-Centric Operations on an HREF

Read an overview of module recursion.



Note change of hierarchy  
from other examples

```
populate -modulecontext CHIP [-norecursive] ALU
```

- Default is `-norecursive`, argument is a *module*
- All files in module CHIP that directly belong to submodule ALU
- Bit of a special case: although is `-norecursive`, picks up href and works on that
- `-nomodulerecursive` not appropriate switches to `-norecursive`

```
populate -modulecontext CHIP -recursive ALU
```

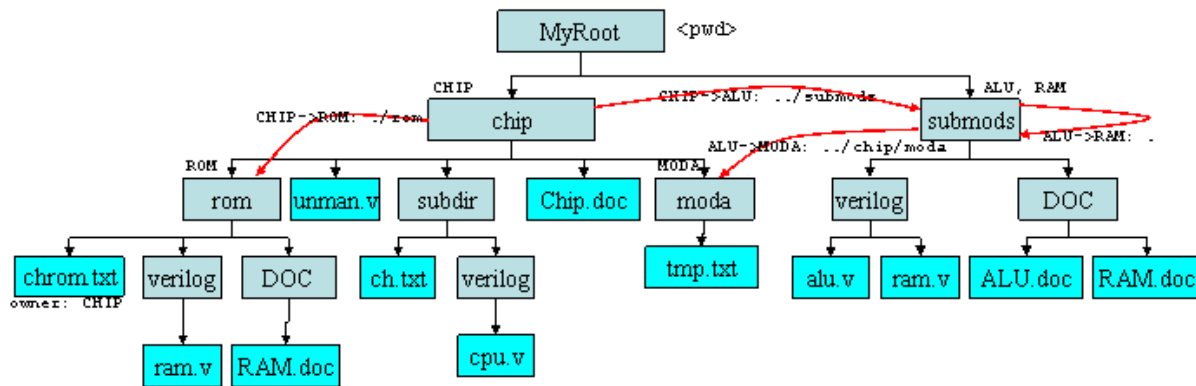
- All files in module CHIP and its sub-modules, which are below ALU in the module hierarchy

Step 4 of 5

View the next step.

### Step 5: Module-Centric Operations on an HREF

Read an overview of module recursion.



Note change of hierarchy  
from other examples

```
populate -modulecontext CHIP [-norecursive] ALU
```

- Default is `-norecursive`, argument is a *module*
- All files in module CHIP that directly belong to submodule ALU
- Bit of a special case: although is `-norecursive`, picks up href and works on that
- `-nomodulerecursive` not appropriate switches to `-norecursive`

```
populate -modulecontext CHIP -recursive ALU
```

- All files in module CHIP and its sub-modules, which are below ALU in the module hierarchy

Step 5 of 5

- HREF contexts are the same as operations on modules**
- Currently, only the populate command operates on HREF's**

## Locking a Module Branch

### Step 1: Locking a Module Branch

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module locking.

## DesignSync Data Manager User's Guide

### Workspace UserA LMD

```
RAM  
-name: RAM  
-selector: Trunk:  
-version: 1.4
```

### Workspace UserB LMD

```
RAM  
-name: RAM  
-selector: Trunk:  
-version: 1.4
```

### Server MD

```
RAM;1.4 (Trunk:Latest)  
-...  
-contents:  
  verilog/ram.v;1.3  
  DOC/RAM.doc;1.2
```

Step 1 of 7

View the next step.

### **Step 2: Locking a Module Branch**

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module locking.

Workspace UserA LMD

```
RAM
-name: RAM
-selector: Trunk:
-version: 1.4
```

Workspace UserB LMD

```
RAM
-name: RAM
-selector: Trunk:
-version: 1.4
```

Server MD

```
RAM;1.4 (Trunk:Latest)
-...
-contents:
  verilog/ram.v;1.3
  DOC/RAM.doc;1.2
```

UserA locks the branch:

```
stcl> hcm lock -comment "abc" RAM
```

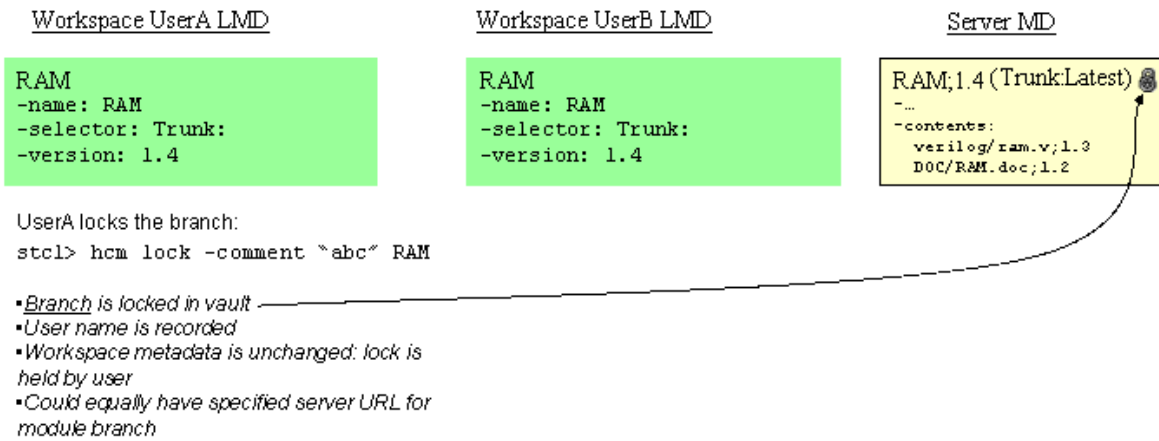
Step 2 of 7

View the next step.

### Step 3: Locking a Module Branch

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module locking.

# DesignSync Data Manager User's Guide

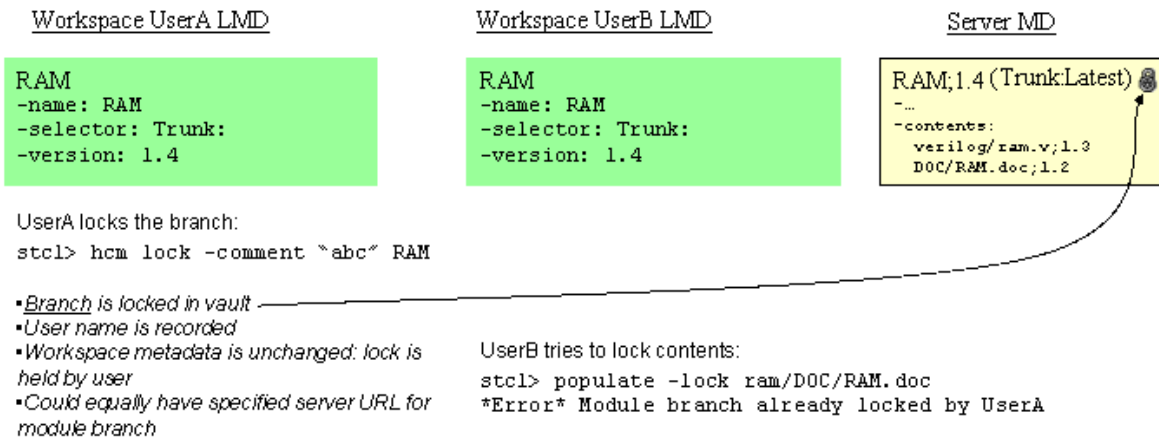


Step 3 of 7

View the next step.

## Step 4: Locking a Module Branch

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module locking.

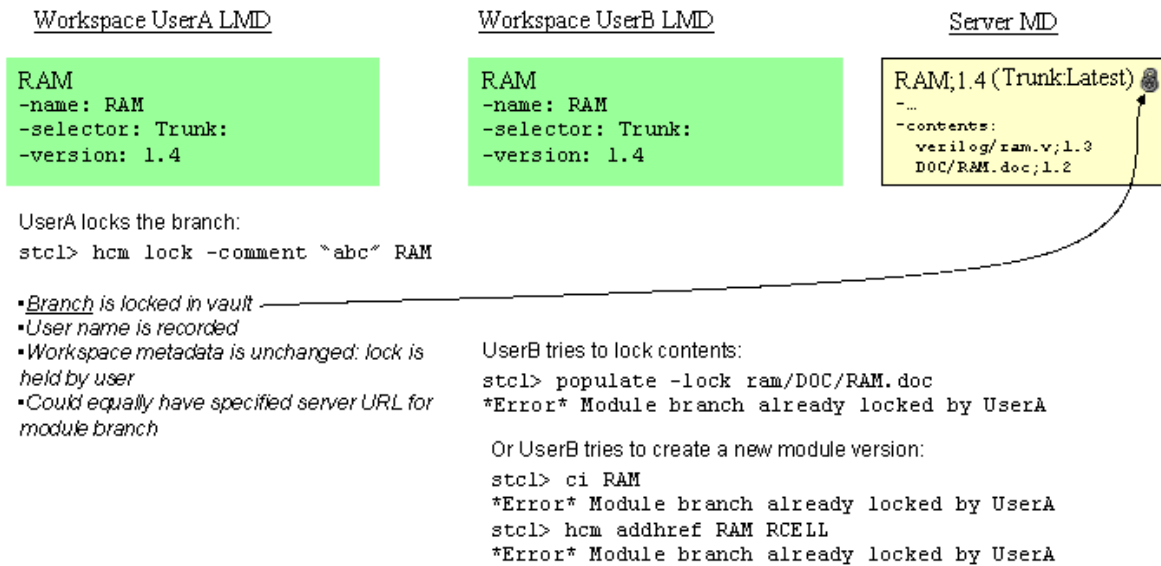


Step 4 of 7

View the next step.

### Step 5: Locking a Module Branch

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module locking.



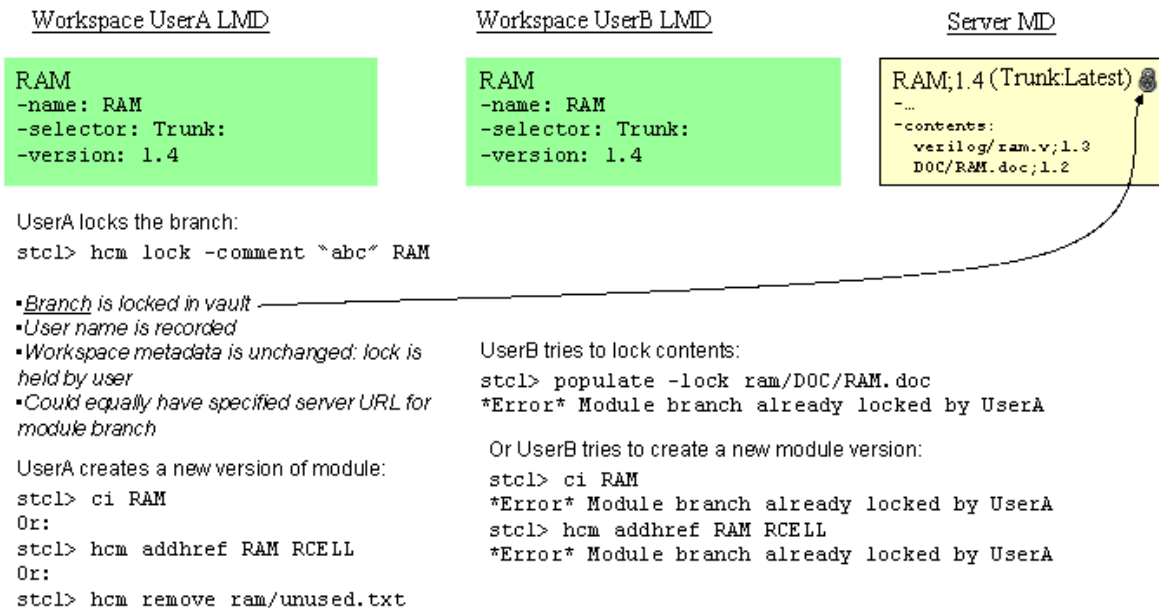
Step 5 of 7

View the next step.

## Step 6: Locking a Module Branch

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module locking.





Step 6 of 7

View the next step.

### Step 7: Locking a Module Branch

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module locking.

Workspace UserA LMD

```
RAM
-name: RAM
-selector: Trunk:
-version: 1.4
```

UserA locks the branch:

```
stcl> hcm lock -comment "abc" RAM
```

- *Branch is locked in vault*
- *User name is recorded*
- *Workspace metadata is unchanged: lock is held by user*
- *Could equally have specified server URL for module branch*

UserA creates a new version of module:

```
stcl> ci RAM
Or:
stcl> hcm addhref RAM RCELL
Or:
stcl> hcm remove ram/unused.txt
```

Operations that creates a new version in the vault:

- *Lock is **not** removed, allowing for multiple version-creating operations, such as add/mhref, move, remove etc.*

Workspace UserB LMD

```
RAM
-name: RAM
-selector: Trunk:
-version: 1.4
```

UserB tries to lock contents:

```
stcl> populate -lock ram/DOC/RAM.doc
*Error* Module branch already locked by UserA
```

Or UserB tries to create a new module version:

```
stcl> ci RAM
*Error* Module branch already locked by UserA
stcl> hcm addhref RAM RCELL
*Error* Module branch already locked by UserA
```

Server MD

```
RAM;1.4
-...
-contents:
  verilog/ram.v;1.3
  DOC/RAM.doc;1.2
```

```
RAM;1.5 (Trunk:Latest)
-...
-contents:
  verilog/ram.v;1.4
  DOC/RAM.doc;1.2
```

Step 7 of 7

*To unlock a module branch, use the "unlock" command with a module server URL.*

## Locking Module Content

### Step 1: Locking Module Content

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module locking.

Workspace UserA LMD

```
RAM
-name: RAM
-selector: Trunk:
-version: 1.4
-content:
  verilog/ram.v;1.3
  DOC/RAM.doc;1.2
```

Workspace UserB LMD

```
RAM
-name: RAM
-selector: Trunk:
-version: 1.4
-content:
  verilog/ram.v;1.3
  DOC/RAM.doc;1.3
```

Server MD

```
RAM;1.4 (Trunk:Latest)
-contents:
  verilog/ram.v;1.3
  DOC/RAM.doc;1.2
```

Step 1 of 12

View the next step.

**Step 2: Locking Module Content**

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module locking.

## Workspace UserA LMD

```
RAM
-name: RAM
-selector: Trunk:
-version: 1.4
-content:
  verilog/ram.v;1.3
  DOC/RAM.doc;1.2
```

UserA locks a file:

```
stcl> populate -lock ram.v
```

## Workspace UserB LMD

```
RAM
-name: RAM
-selector: Trunk:
-version: 1.4
-content:
  verilog/ram.v;1.3
  DOC/RAM.doc;1.3
```

## Server MD

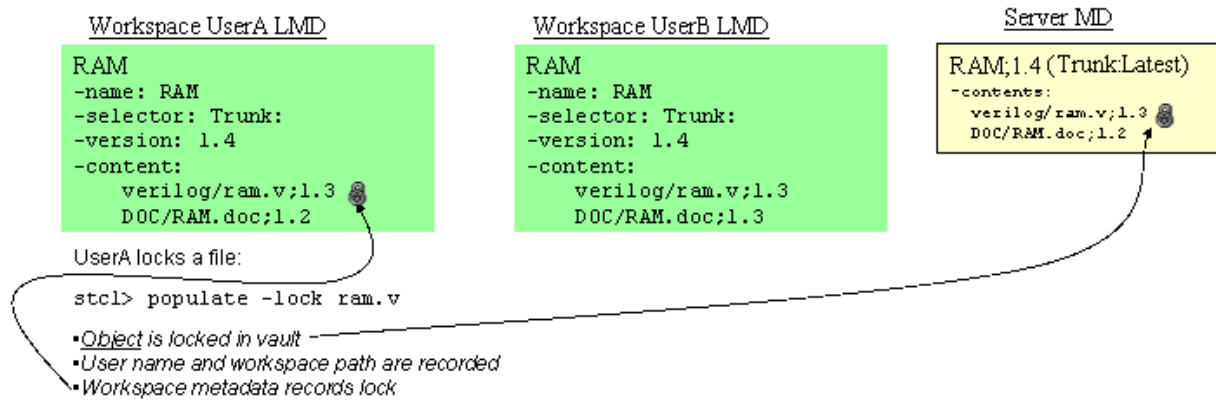
```
RAM;1.4 (Trunk:Latest)
-contents:
  verilog/ram.v;1.3
  DOC/RAM.doc;1.2
```

Step 2 of 12

[View the next step](#)

### **Step 3: Locking Module Content**

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module locking.



Step 3 of 12

[View the next step](#)

#### Step 4: Locking Module Content

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module locking.

# DesignSync Data Manager User's Guide

<u>Workspace UserA LMD</u>	<u>Workspace UserB LMD</u>	<u>Server MD</u>
<pre>RAM -name: RAM -selector: Trunk: -version: 1.4 -content:   verilog/ram.v;1.3 🗝️   DOC/RAM.doc;1.2</pre>	<pre>RAM -name: RAM -selector: Trunk: -version: 1.4 -content:   verilog/ram.v;1.3   DOC/RAM.doc;1.3</pre>	<pre>RAM;1.4 (Trunk:Latest) -contents:   verilog/ram.v;1.3 🗝️   DOC/RAM.doc;1.2</pre>
<p>UserA locks a file:</p> <pre>stcl&gt; populate -lock ram.v</pre> <ul style="list-style-type: none"><li>▪ <i>Object is locked in vault</i></li><li>▪ <i>User name and workspace path are recorded</i></li><li>▪ <i>Workspace metadata records lock</i></li></ul>	<p>UserB tries to lock contents:</p> <pre>stcl&gt; populate -lock -rec RAM *Error* Cannot lock verilog/ram.v, object already locked by UserA in /home/UserA/workspace</pre>	

Step 4 of 12

[View the next step](#)

## Step 5: Locking Module Content

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module locking.

Workspace UserA LMD

```
RAM
-name: RAM
-selector: Trunk:
-version: 1.4
-content:
  verilog/ram.v;1.3
  DOC/RAM.doc;1.2
```

UserA locks a file:

```
stcl> populate -lock ram.v
```

- *Object is locked in vault*
- *User name and workspace path are recorded*
- *Workspace metadata records lock*

Workspace UserB LMD

```
RAM
-name: RAM
-selector: Trunk:
-version: 1.4
-content:
  verilog/ram.v;1.3
  DOC/RAM.doc;1.3
```

UserB tries to lock contents:

```
stcl> populate -lock -rec RAM
```

```
*Error* Cannot lock verilog/ram.v, object already
locked by UserA in /home/UserA/workspace
```

- *Items to be locked overlap with existing locked objects, so whole command fails (atomic operation)*
- *Error would be given for each already locked item*
- *Error indicates the workspace path to the module where this is locked as well as the user name*

Server MD

```
RAM;1.4 (Trunk:Latest)
-contents:
  verilog/ram.v;1.3
  DOC/RAM.doc;1.2
```

Step 5 of 12

View the next step

### Step 6: Locking Module Content

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module locking.

## Workspace UserA LMD

```
RAM
-name: RAM
-selector: Trunk:
-version: 1.4
-content:
  verilog/ram.v;1.3
  DOC/RAM.doc;1.2
```

UserA locks a file:

```
stcl> populate -lock ram.v
```

- *Object is locked in vault*
- *User name and workspace path are recorded*
- *Workspace metadata records lock*

## Workspace UserB LMD

```
RAM
-name: RAM
-selector: Trunk:
-version: 1.4
-content:
  verilog/ram.v;1.3
  DOC/RAM.doc;1.3
```

UserB tries to lock contents:

```
stcl> populate -lock -rec RAM
*Error* Cannot lock verilog/ram.v, object already
locked by UserA in /home/UserA/workspace
  • Items to be locked overlap with existing locked objects,
    so whole command fails (atomic operation)
  • Error would be given for each already locked item
  • Error indicates the workspace path to the module where
    this is locked as well as the user name
```

UserB can still create new module versions:

## Server MD

```
RAM;1.4 (Trunk:Latest)
-contents:
  verilog/ram.v;1.3
  DOC/RAM.doc;1.2
```

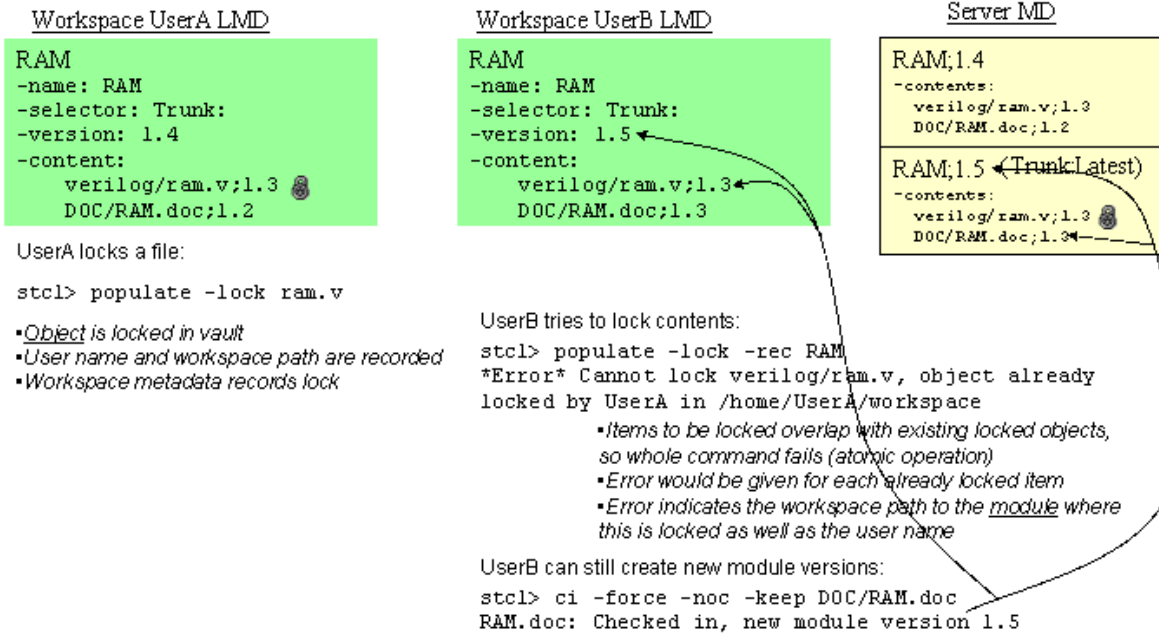
Step 6 of 12

View the next step

## Step 7: Locking Module Content

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module locking.





Step 7 of 12

View the next step

**Step 8: Locking Module Content**

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module locking.

<p style="text-align: center; text-decoration: underline;">Workspace UserA LMD</p> <pre style="background-color: #e0ffe0; padding: 5px;">RAM -name: RAM -selector: Trunk: -version: 1.4 -content:   verilog/ram.v;1.3   DOC/RAM.doc;1.2</pre> <p>UserA locks a file:</p> <pre>stcl&gt; populate -lock ram.v</pre> <ul style="list-style-type: none"> <li>▪ <i>Object is locked in vault</i></li> <li>▪ <i>User name and workspace path are recorded</i></li> <li>▪ <i>Workspace metadata records lock</i></li> </ul>	<p style="text-align: center; text-decoration: underline;">Workspace UserB LMD</p> <pre style="background-color: #e0ffe0; padding: 5px;">RAM -name: RAM -selector: Trunk: -version: 1.5 -content:   verilog/ram.v;1.3   DOC/RAM.doc;1.3</pre> <p>UserB tries to lock contents:</p> <pre>stcl&gt; populate -lock -rec RAM *Error* Cannot lock verilog/ram.v, object already locked by UserA in /home/UserA/workspace</pre> <ul style="list-style-type: none"> <li>▪ <i>Items to be locked overlap with existing locked objects, so whole command fails (atomic operation)</i></li> <li>▪ <i>Error would be given for each already locked item</i></li> <li>▪ <i>Error indicates the workspace path to the <u>module</u> where this is locked as well as the user name</i></li> </ul> <p>UserB can still create new module versions:</p> <pre>stcl&gt; ci -force -noc -keep DOC/RAM.doc RAM.doc: Checked in, new module version 1.5</pre> <ul style="list-style-type: none"> <li>▪ <i>Lock remains on ram.v</i></li> <li>▪ <i>Local module version updated</i></li> </ul>	<p style="text-align: center; text-decoration: underline;">Server MD</p> <table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding: 5px;"> <pre>RAM;1.4 -contents:   verilog/ram.v;1.3   DOC/RAM.doc;1.2</pre> </td> </tr> <tr> <td style="padding: 5px;"> <pre>RAM;1.5 (Trunk:Latest) -contents:   verilog/ram.v;1.3   DOC/RAM.doc;1.3</pre> </td> </tr> </table>	<pre>RAM;1.4 -contents:   verilog/ram.v;1.3   DOC/RAM.doc;1.2</pre>	<pre>RAM;1.5 (Trunk:Latest) -contents:   verilog/ram.v;1.3   DOC/RAM.doc;1.3</pre>
<pre>RAM;1.4 -contents:   verilog/ram.v;1.3   DOC/RAM.doc;1.2</pre>				
<pre>RAM;1.5 (Trunk:Latest) -contents:   verilog/ram.v;1.3   DOC/RAM.doc;1.3</pre>				

Step 8 of 12

View the next step

## Step 9: Locking Module Content

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module locking.

Workspace UserA LMD

```
RAM
-name: RAM
-selector: Trunk:
-version: 1.4
-content:
  verilog/ram.v;1.3
  DOC/RAM.doc;1.2
```

UserA locks a file:

```
stcl> populate -lock ram.v
```

- *Object is locked in vault*
- *User name and workspace path are recorded*
- *Workspace metadata records lock*

UserA checks in file:

```
stcl> touch verilog/ram.v
stcl> ci RAM
```

Workspace UserB LMD

```
RAM
-name: RAM
-selector: Trunk:
-version: 1.4
-content:
  verilog/ram.v;1.3
  DOC/RAM.doc;1.2
```

UserB tries to lock contents:

```
stcl> populate -lock -rec RAM
*Error* Cannot lock verilog/ram.v, object already
locked by UserA in /home/UserA/workspace
  ▪ Items to be locked overlap with existing locked objects,
    so whole command fails (atomic operation)
  ▪ Error would be given for each already locked item
  ▪ Error indicates the workspace path to the module where
    this is locked as well as the user name
```

UserB can still create new module versions:

```
stcl> ci -force -noc -keep DOC/RAM.doc
RAM.doc: Checked in, new module version 1.5
  ▪ Lock remains on ram.v
  ▪ Local module version updated
```

Server MD

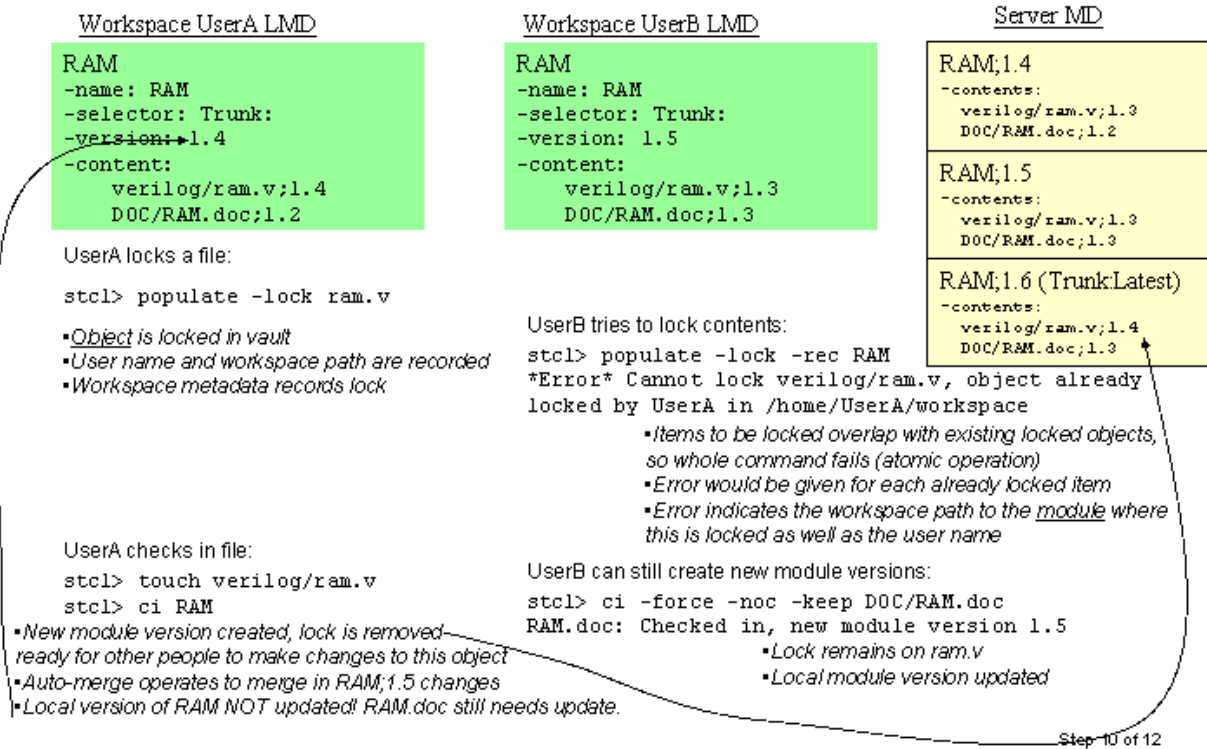
```
RAM;1.4
-contents:
  verilog/ram.v;1.3
  DOC/RAM.doc;1.2
RAM;1.5 (Trunk:Latest)
-contents:
  verilog/ram.v;1.3
  DOC/RAM.doc;1.3
```

Step 9 of 12

View the next step

### Step 10: Locking Module Content

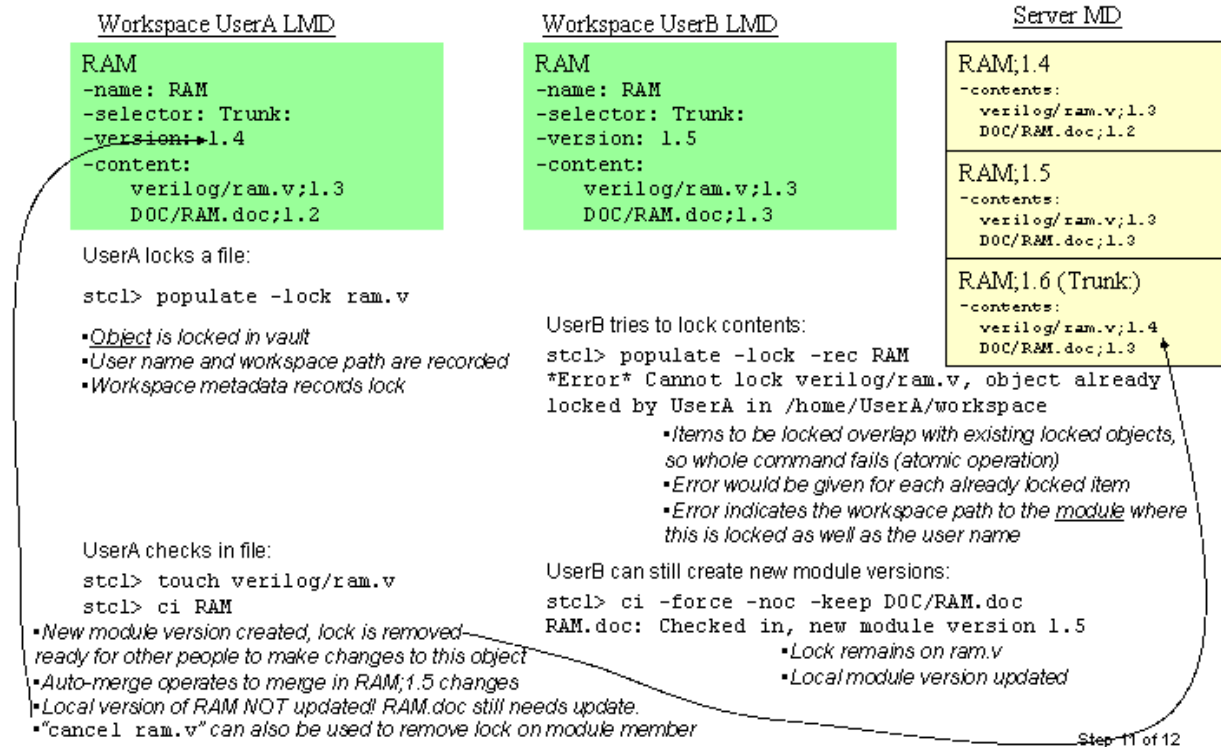
In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module locking.



View the next step

### Step 11: Locking Module Content

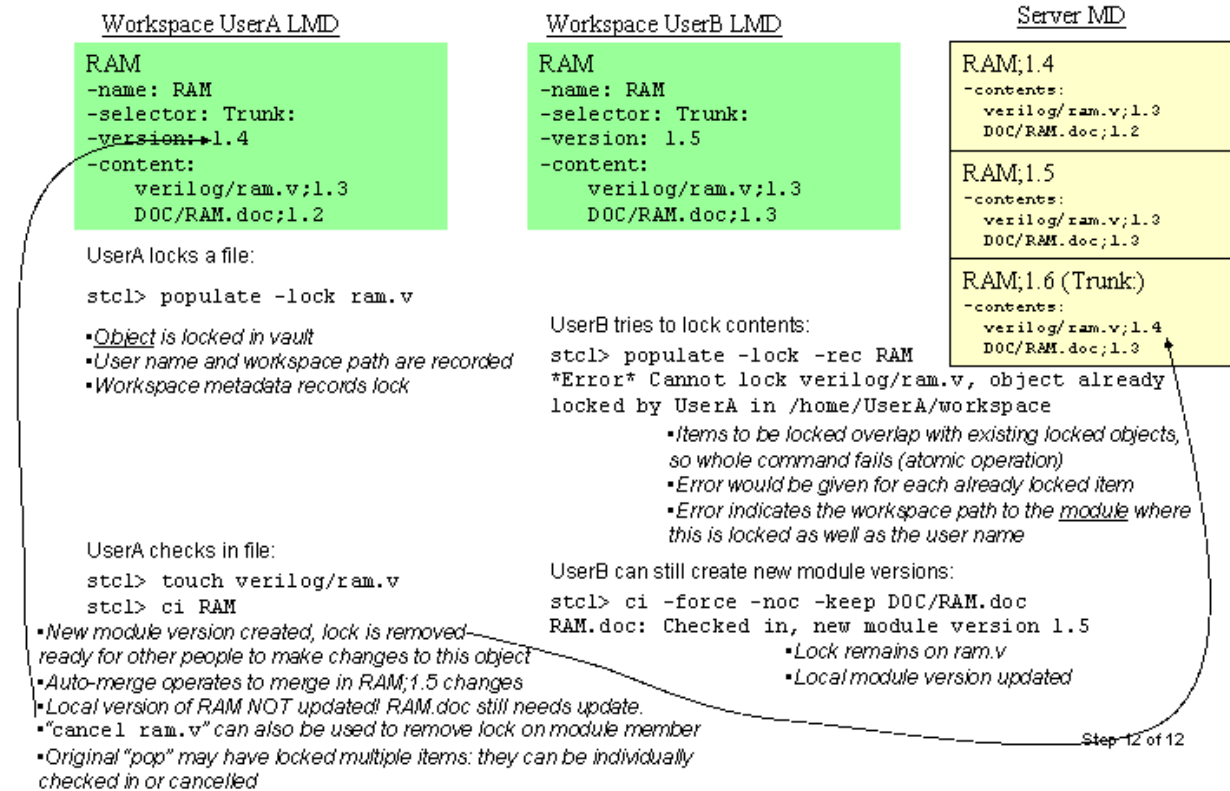
In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module locking.



View the next step

### Step 12: Locking Module Content

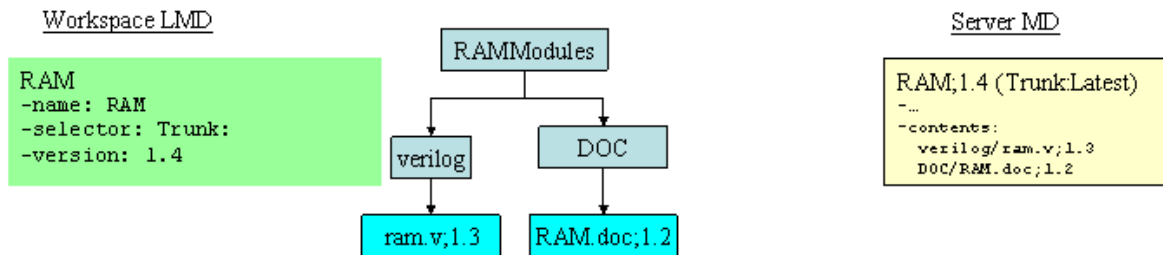
In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module locking.



## Branching a Module

### Step 1: Branching a Module

In this use case, the RAM team is producing a new version of the RAM module, with an "automatic undo" feature. The new version is created as a side-branch of version 1.4. "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module branching.

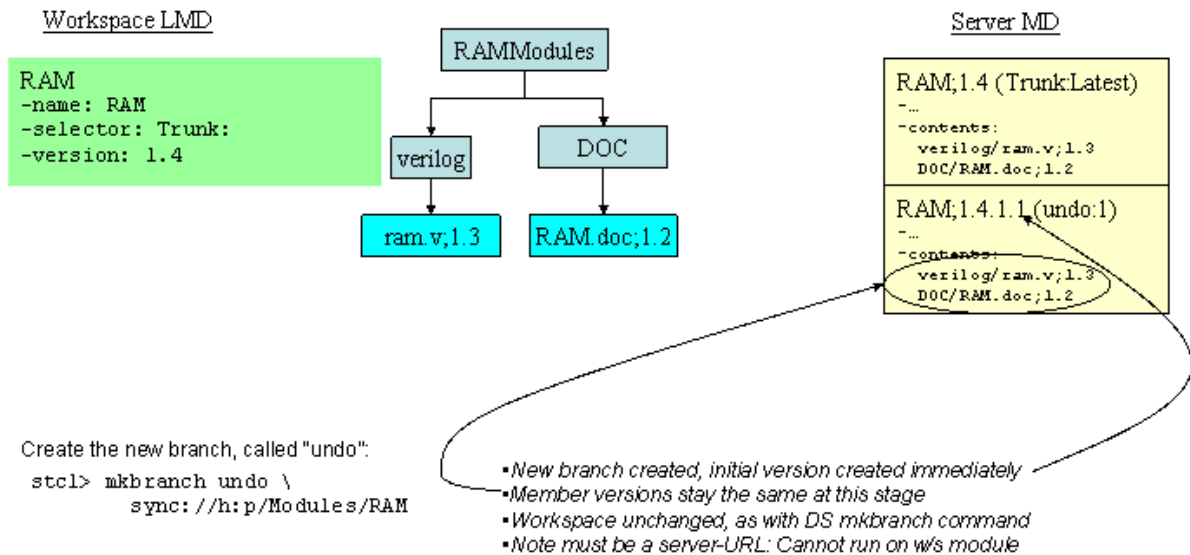


Step 1 of 6

View the next step.

### Step 2: Branching a Module

In this use case, the RAM team is producing a new version of the RAM module, with an "automatic undo" feature. The new version is created as a side-branch of version 1.4. "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module branching.



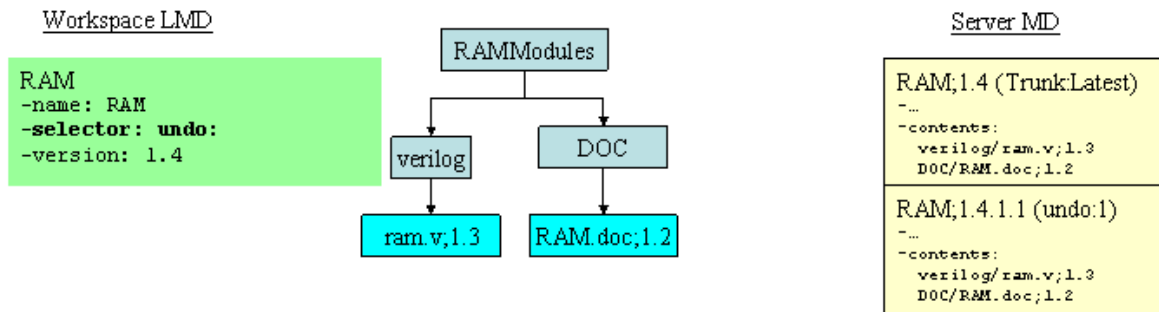
Step 2 of 6

View the next step

### Step 3: Branching a Module

In this use case, the RAM team is producing a new version of the RAM module, with an "automatic undo" feature. The new version is created as a side-branch of version 1.4. "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module branching.





Create the new branch, called "undo":

```
stcl> mkbranch undo \
      sync: //h:p/Modules/RAM
```

Switch to the new branch in the w/s:

```
stcl> setselector undo:Latest RAM
```

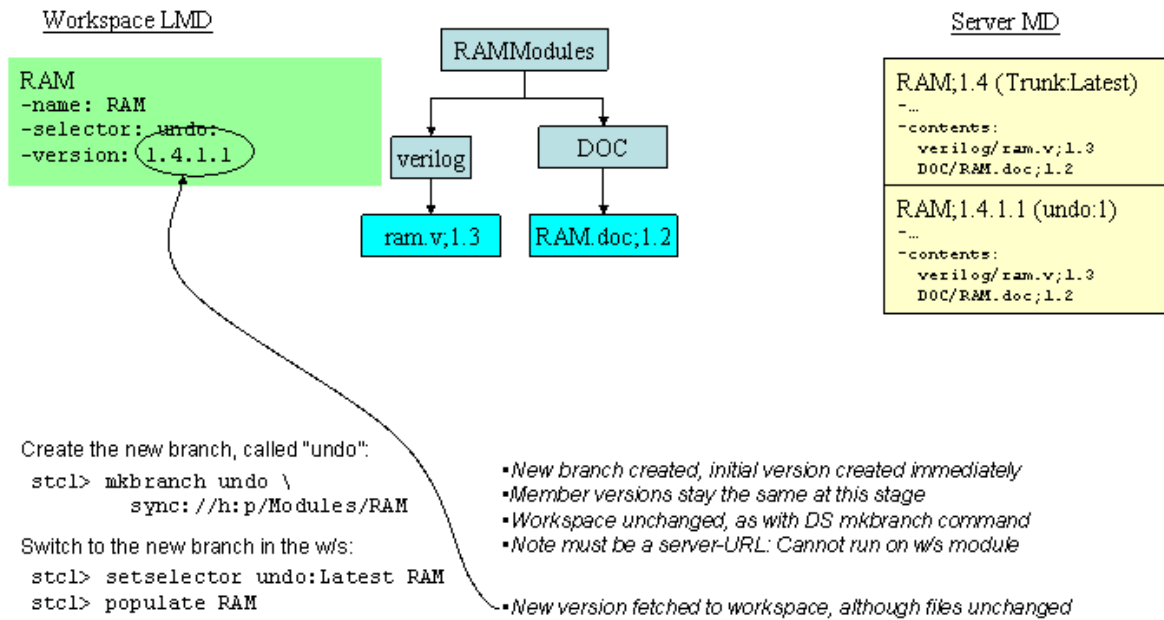
- *New branch created, initial version created immediately*
- *Member versions stay the same at this stage*
- *Workspace unchanged, as with DS mkbranch command*
- *Note must be a server-URL: Cannot run on w/s module*

Step 3 of 6

View the next step

#### Step 4: Branching a Module

In this use case, the RAM team is producing a new version of the RAM module, with an "automatic undo" feature. The new version is created as a side-branch of version 1.4. "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module branching.

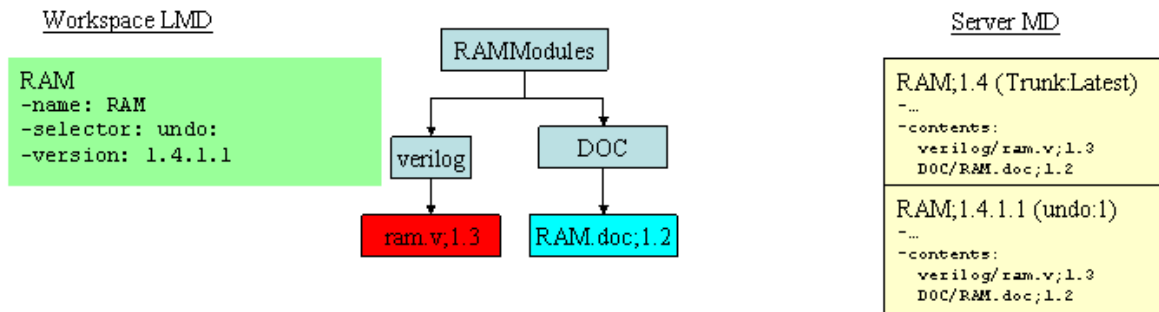


Step 4 of 6

View the next step

### Step 5: Branching a Module

In this use case, the RAM team is producing a new version of the RAM module, with an "automatic undo" feature. The new version is created as a side-branch of version 1.4. "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module branching.



Create the new branch, called "undo":

```
stcl> mkbranch undo \
      sync: //h:p/Modules/RAM
```

Switch to the new branch in the w/s:

```
stcl> setselector undo:Latest RAM
stcl> populate RAM
```

Modify a file and check in:

```
stcl> vi verilog/ram.v
```

- New branch created, initial version created immediately
- Member versions stay the same at this stage
- Workspace unchanged, as with DS mkbranch command
- Note must be a server-URL: Cannot run on w/s module

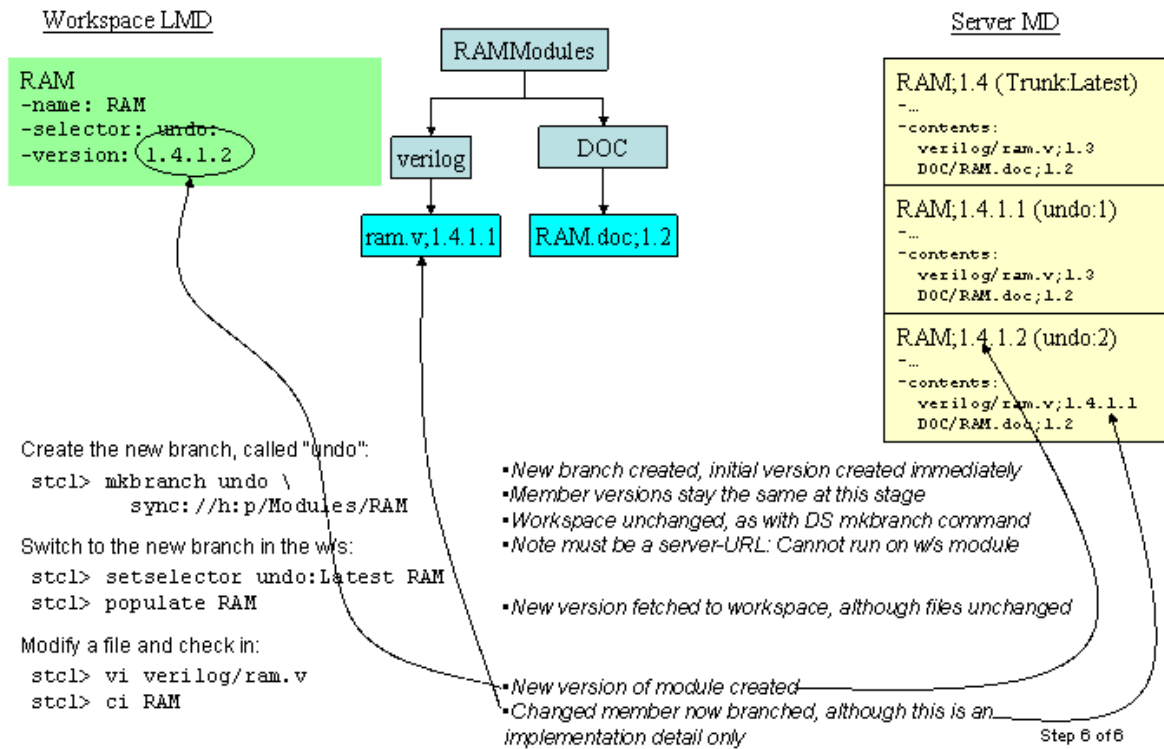
- New version fetched to workspace, although files unchanged

Step 5 of 6

View the next step

### Step 6: Branching a Module

In this use case, the RAM team is producing a new version of the RAM module, with an "automatic undo" feature. The new version is created as a side-branch of version 1.4. "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read an overview of module branching.



## Merging and Modules

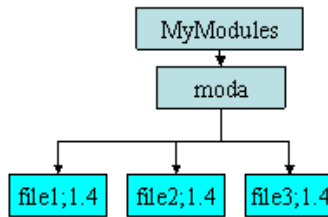
### Auto-Merging Locally Added Files

#### Step 1: Auto-Merging Locally Added Files

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read a description of this use case.

Workspace LMD

```
ModA
-name: ModA
-selector: Trunk:
-version: 1.4
-contents:
  file1;1.4
  file2;1.4
  file3;1.4
```

Server MD

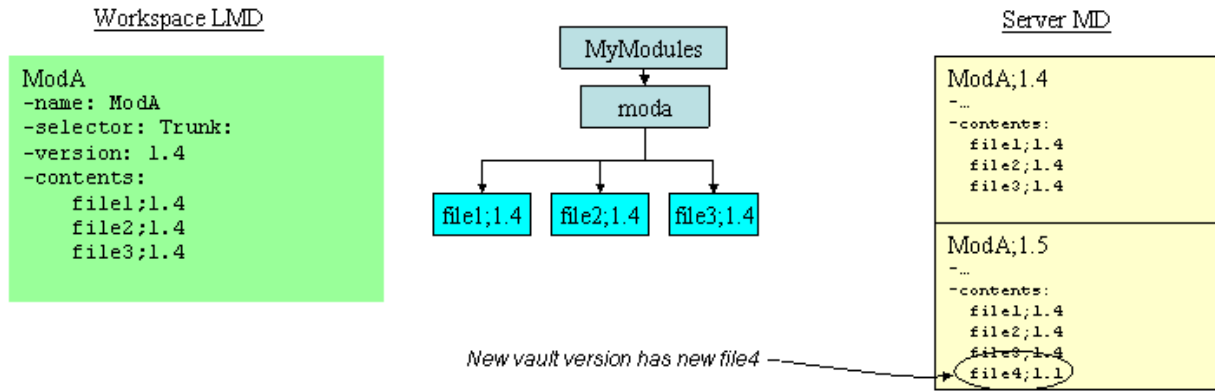
```
ModA;1.4
-...
-contents:
  file1;1.4
  file2;1.4
  file3;1.4
```

Step 1 of 4

View the next step.

**Step 2: Auto-Merging Locally Added Files**

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read a description of this use case.

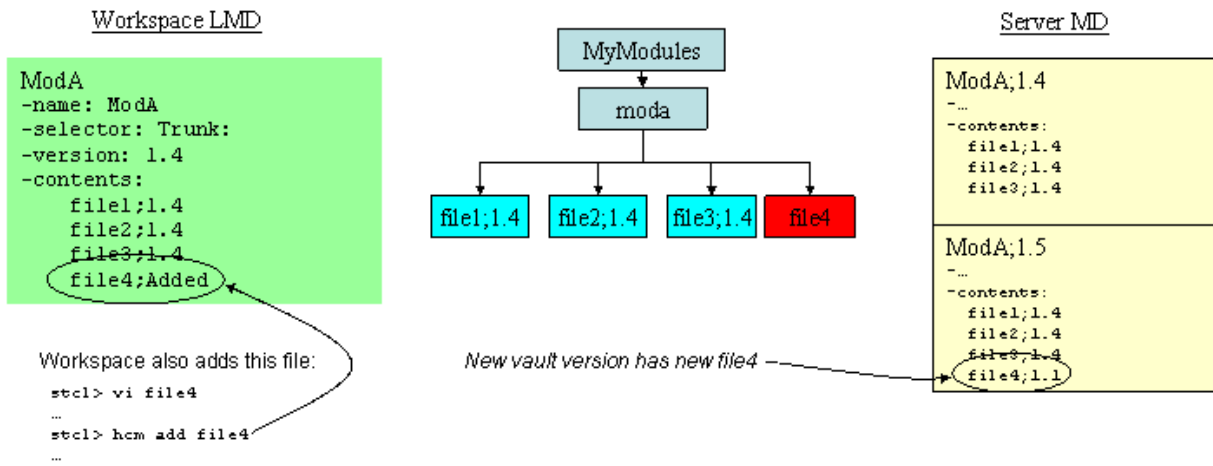


Step 2 of 4

View the next step.

### Step 3: Auto-Merging Locally Added Files

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read a description of this use case.

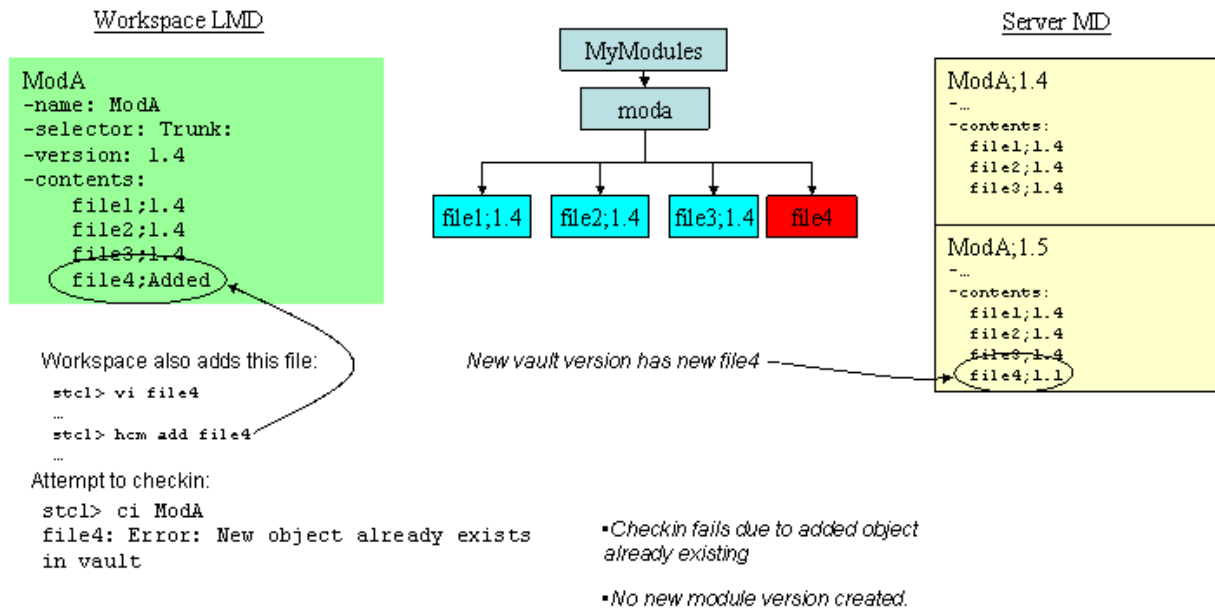


Step 3 of 4

View the next step.

#### Step 4: Auto-Merging Locally Added Files

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read a description of this use case.



Step 4 of 4

## Auto-Merging Locally Modified Files

### Step 1: Auto-Merging Locally Modified Files

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read a description of this use case.

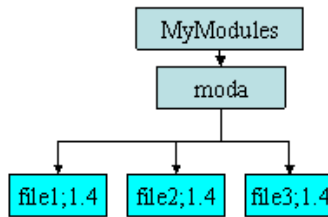


Workspace LMD

```

ModA
-name: ModA
-selector: Trunk:
-version: 1.4
-contents:
  file1;1.4
  file2;1.4
  file3;1.4

```

Server MD

```

ModA;1.4
-...
-contents:
  file1;1.4
  file2;1.4
  file3;1.4

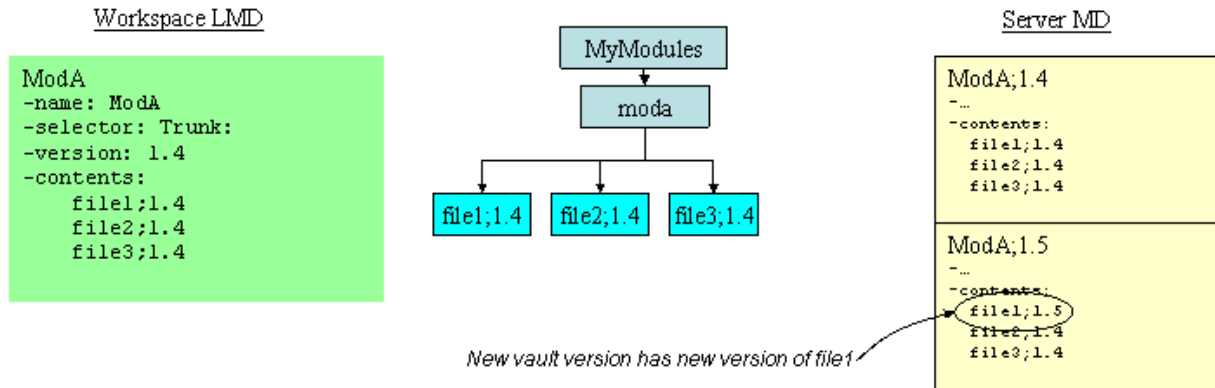
```

Step 1 of 6

View the next step.

**Step 2: Auto-Merging Locally Modified Files**

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read a description of this use case.

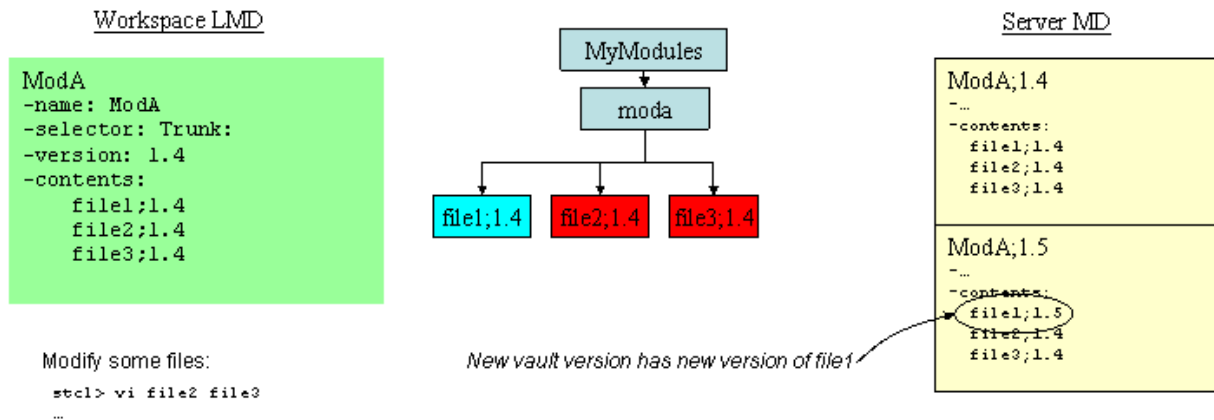


Step 2 of 6

View the next step.

### Step 3: Auto-Merging Locally Modified Files

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read a description of this use case.

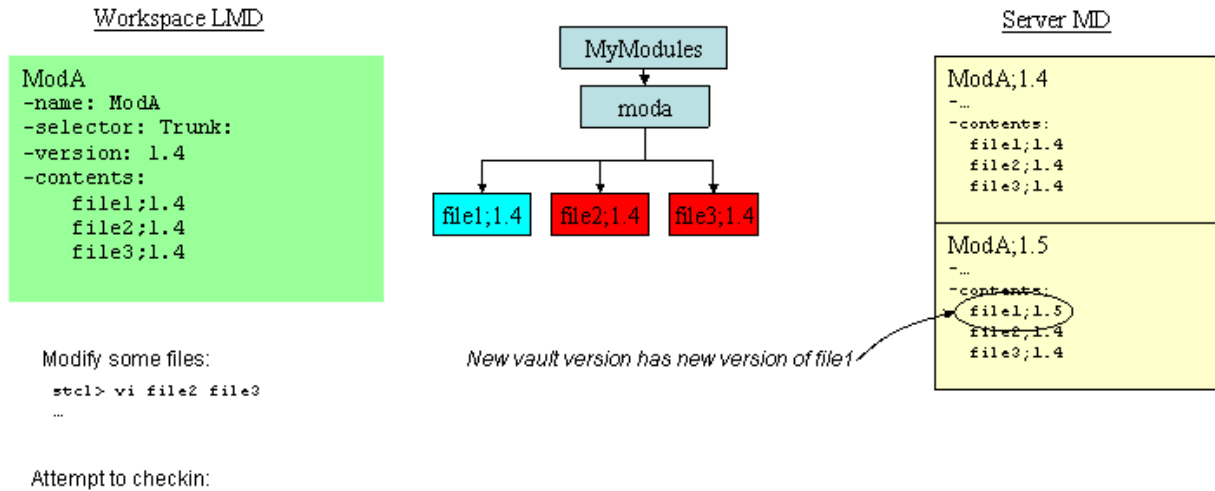


Step 3 of 6

View the next step.

#### Step 4: Auto-Merging Locally Modified Files

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read a description of this use case.

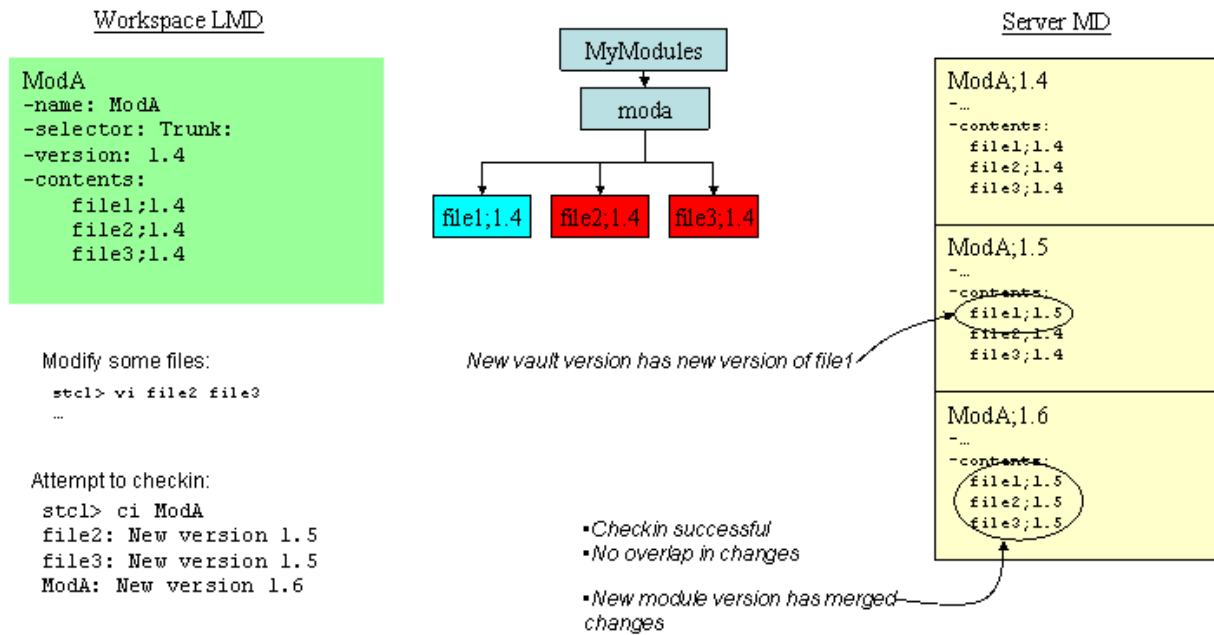


Step 4 of 6

View the next step.

### Step 5: Auto-Merging Locally Modified Files

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read a description of this use case.

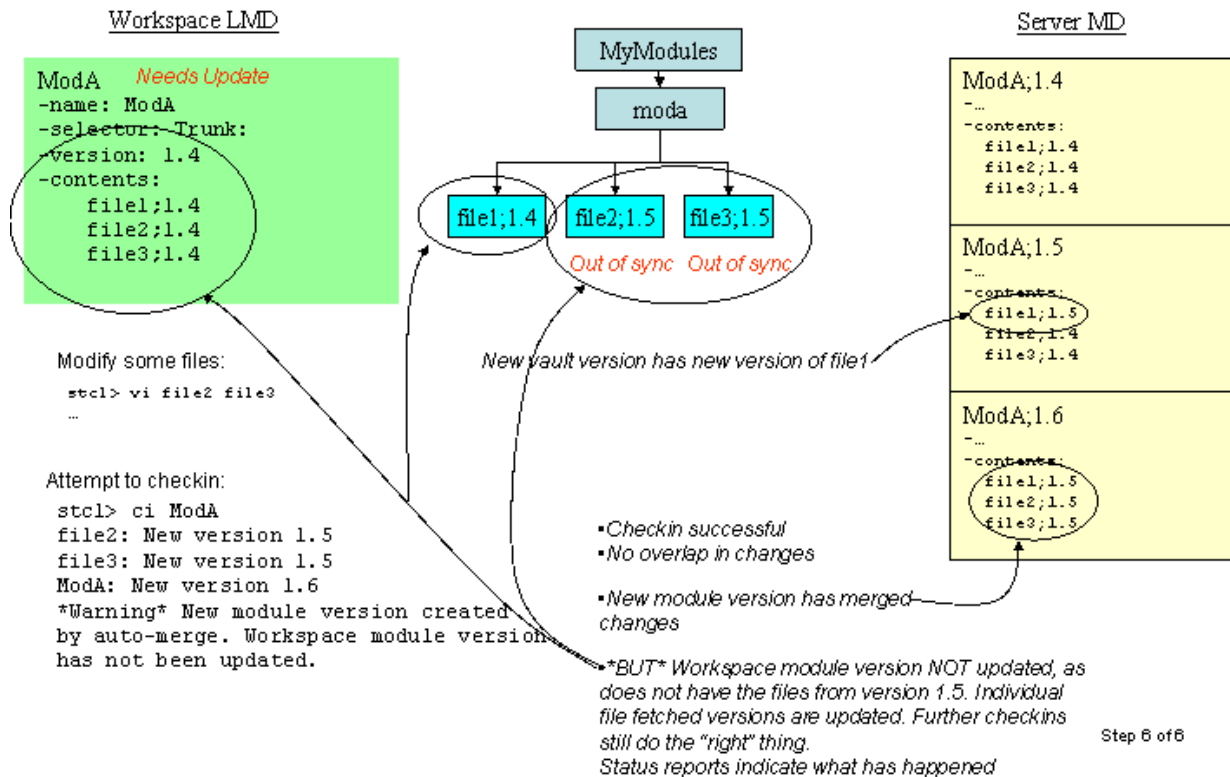


Step 5 of 6

View the next step.

### Step 6: Auto-Merging Locally Modified Files

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read a description of this use case.



## Auto-Merging Locally Modified Files Removed from the Module

### Step 1: Auto-Merging Locally Modified Files Removed from the Module

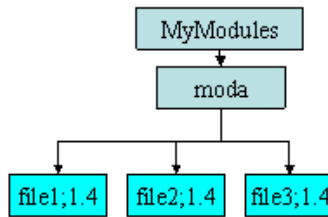
In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read a description of this use case.

Workspace LMD

```

ModA
-name: ModA
-selector: Trunk:
-version: 1.4
-contents:
  file1;1.4
  file2;1.4
  file3;1.4

```

Server MD

```

ModA;1.4
-...
-contents:
  uid 111: file1;1.4
  uid 222: file2;1.4
  uid 333: file3;1.4

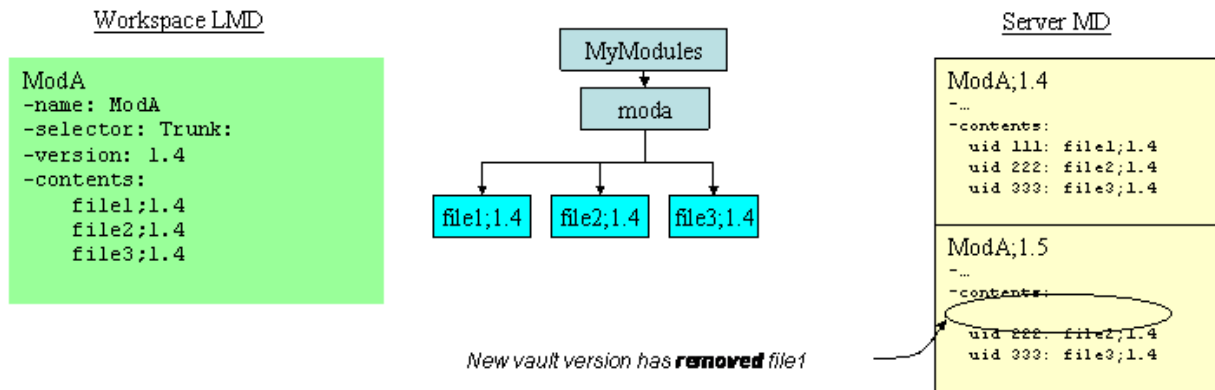
```

Step 1 of 5

View the next step.

**Step 2: Auto-Merging Locally Modified Files Removed from the Module**

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read a description of this use case.



Step 2 of 5

View the next step

### Step 3: Auto-Merging Locally Modified Files Removed from the Module

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read a description of this use case.



Workspace LMD

```

ModA
-name: ModA
-selector: Trunk:
-version: 1.4
-contents:
  file1;1.4
  file2;1.4
  file3;1.4

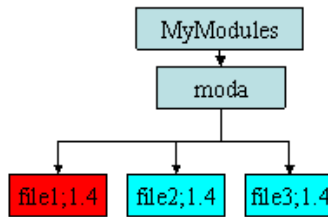
```

Workspace edits **the same** file:

```

stcl> vi file1
...

```



New vault version has **removed** file1

Server MD

```

ModA;1.4
-...
-contents:
  uid 111: file1;1.4
  uid 222: file2;1.4
  uid 333: file3;1.4

```

```

ModA;1.5
-...
-contents:
  uid 222: file2;1.4
  uid 333: file3;1.4

```

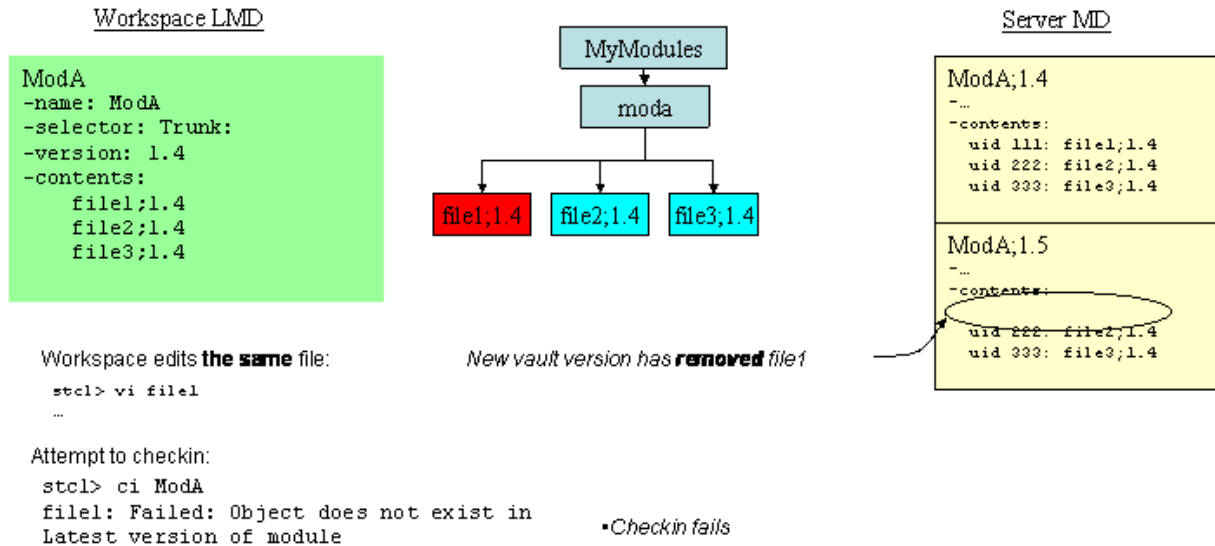
Step 3 of 5

View the next step

#### Step 4: Auto-Merging Locally Modified Files Removed from the Module

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read a description of this use case.

# DesignSync Data Manager User's Guide



Step 4 of 5

View the next step

## Step 5: Auto-Merging Locally Modified Files Removed from the Module

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read a description of this use case.

Workspace LMD

```
ModA
-name: ModA
-selector: Trunk:
-version: 1.4
-contents:
  file1;1.5
  file2;1.4
  file3;1.4
```

Workspace edits **the same** file:

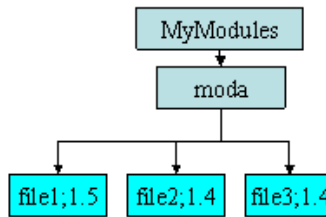
```
stcl> vi file1
...
```

Attempt to checkin:

```
stcl> ci ModA
file1: Failed: Object does not exist in
Latest version of module
```

Attempt to checkin with new:

```
stcl> ci -new moda/file1
file1: New version 1.5
ModA: New version 1.6
*Warning* New module version created
by auto-merge. Workspace module version
has not been updated.
```



New vault version has **removed** file1

Server MD

```
ModA;1.4
-...
-contents:
  uid 111: file1;1.4
  uid 222: file2;1.4
  uid 333: file3;1.4

ModA;1.5
-...
-contents:
  uid 222: file2;1.4
  uid 333: file3;1.4

ModA;1.6
-...
-contents:
  uid 111: file1;1.5
  uid 222: file2;1.4
  uid 333: file3;1.4
```

•Checkin fails

- A "ci-new" can be used to get the object added back
- A "hcm add" is allowed on managed objects as well, for this reason (and should work even if not modified)
- A "skip" is NOT required because the object is not actually in the Latest version.

Step 5 of 5

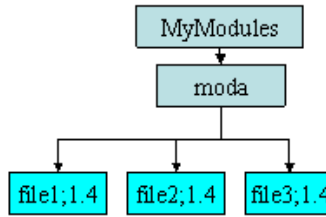
## Auto-Merging Non-Latest Locally Modified Files

### Step 1: Auto-Merging Non-Latest Locally Modified Files

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read a description of this use case.

Workspace LMD

```
ModA
-name: ModA
-selector: Trunk:
-version: 1.4
-contents:
  file1;1.4
  file2;1.4
  file3;1.4
```



Server MD

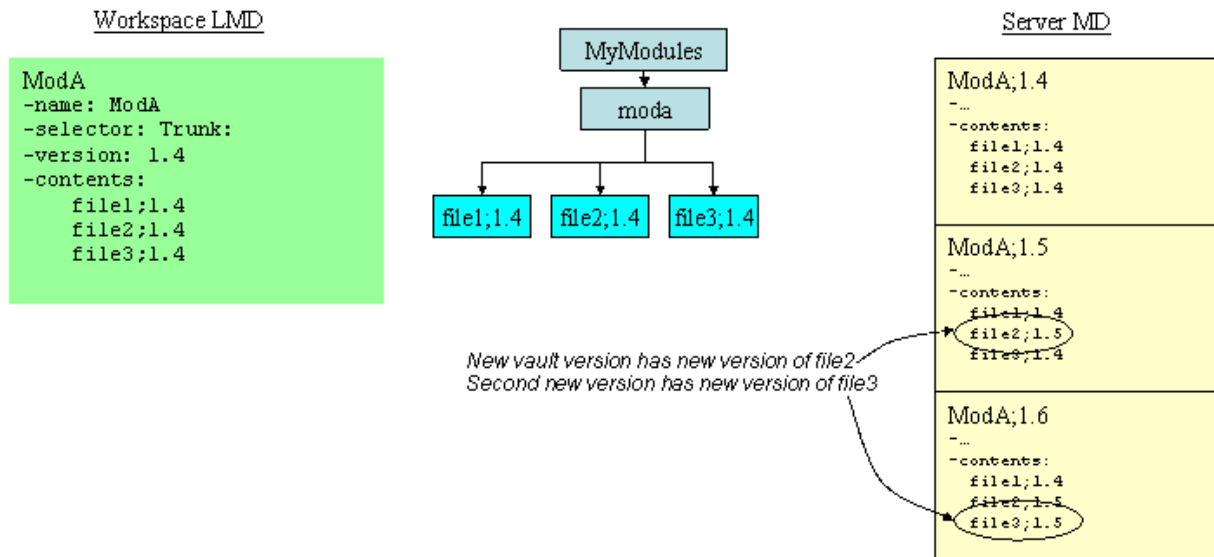
```
ModA;1.4
-...
-contents:
  file1;1.4
  file2;1.4
  file3;1.4
```

Step 1 of 4

View the next step.

**Step 2: Auto-Merging Non-Latest Locally Modified Files**

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read a description of this use case.

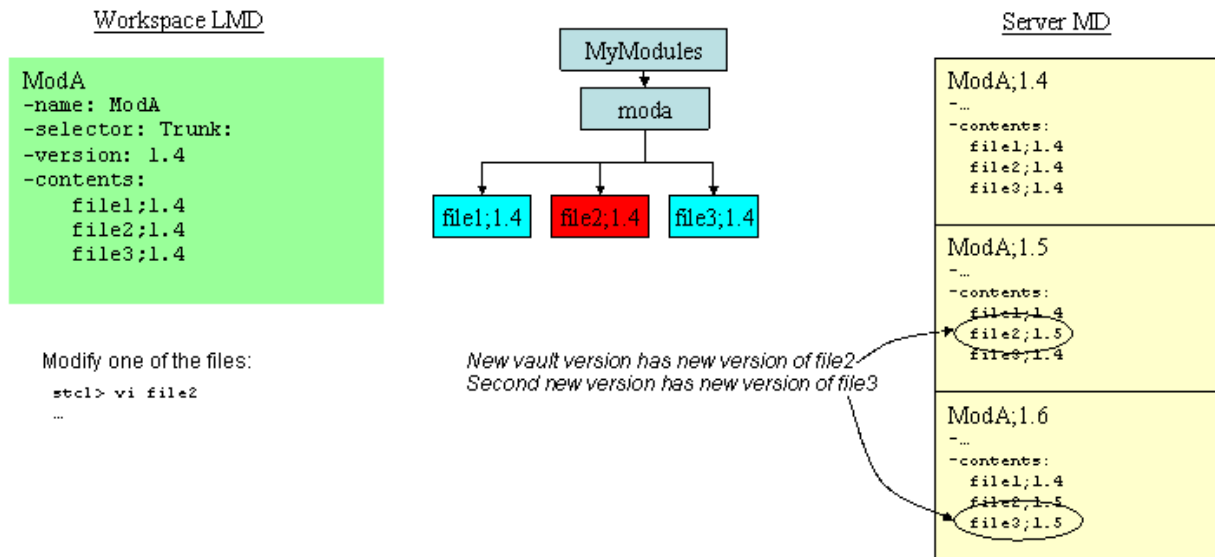


Step 2 of 4

View the next step.

### Step 3: Auto-Merging Non-Latest Locally Modified Files

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read a description of this use case.

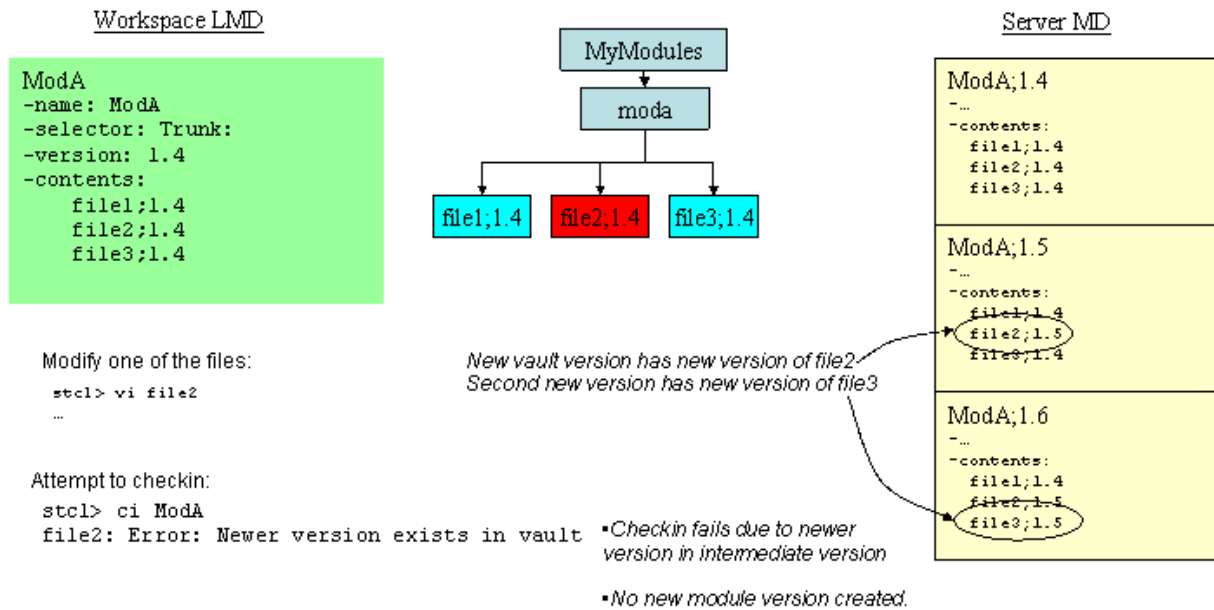


Step 3 of 4

View the next step.

#### Step 4: Auto-Merging Non-Latest Locally Modified Files

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read a description of this use case.



Step 4 of 4

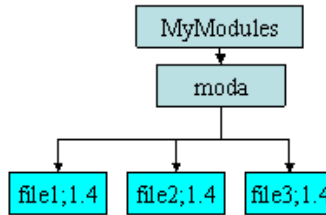
## Auto-Merging Locally Modified Files Renamed in the Module

### Step 1: Auto-Merging Locally Modified Files Renamed in the Module

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read a description of this use case.

Workspace LMD

```
ModA
-name: ModA
-selector: Trunk:
-version: 1.4
-contents:
  uid 111: file1;1.4
  uid 222: file2;1.4
  uid 333: file3;1.4
```



Server MD

```
ModA;1.4
-...
-contents:
  uid 111: file1;1.4
  uid 222: file2;1.4
  uid 333: file3;1.4
```

Step 1 of 4

View the next step.

**Step 2: Auto-Merging Locally Modified Files Renamed in the Module**

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read a description of this use case.

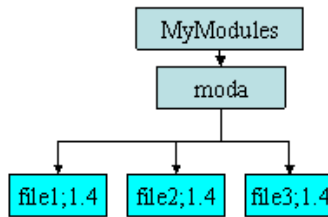


Workspace LMD

```

ModA
-name: ModA
-selector: Trunk:
-version: 1.4
-contents:
  uid 111: file1;1.4
  uid 222: file2;1.4
  uid 333: file3;1.4

```

Server MD

```

ModA;1.4
-...
-contents:
  uid 111: file1;1.4
  uid 222: file2;1.4
  uid 333: file3;1.4

```

```

ModA;1.5
-...
-contents:
  uid 111: file4;1.4
  uid 222: file2;1.4
  uid 333: file3;1.4

```

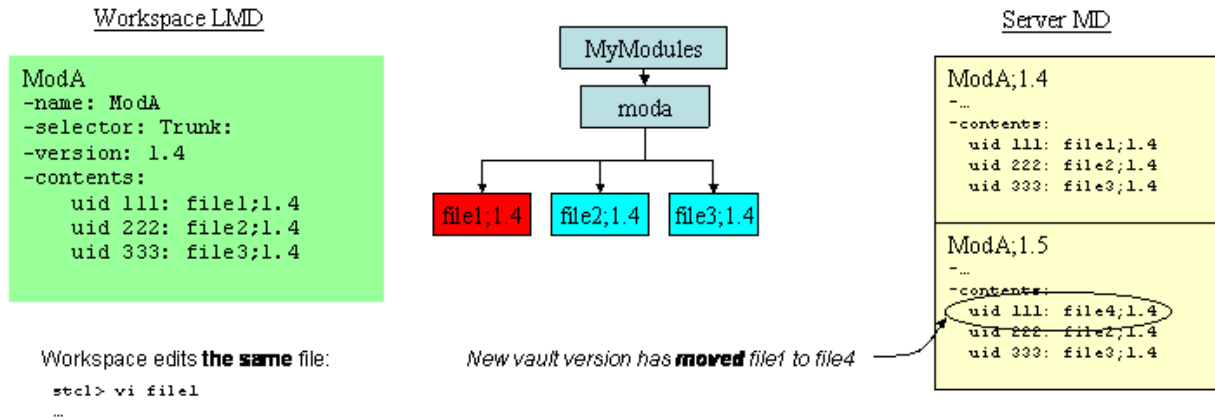
New vault version has **moved** file1 to file4

Step 2 of 4

View the next step

### Step 3: Auto-Merging Locally Modified Files Renamed in the Module

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read a description of this use case.

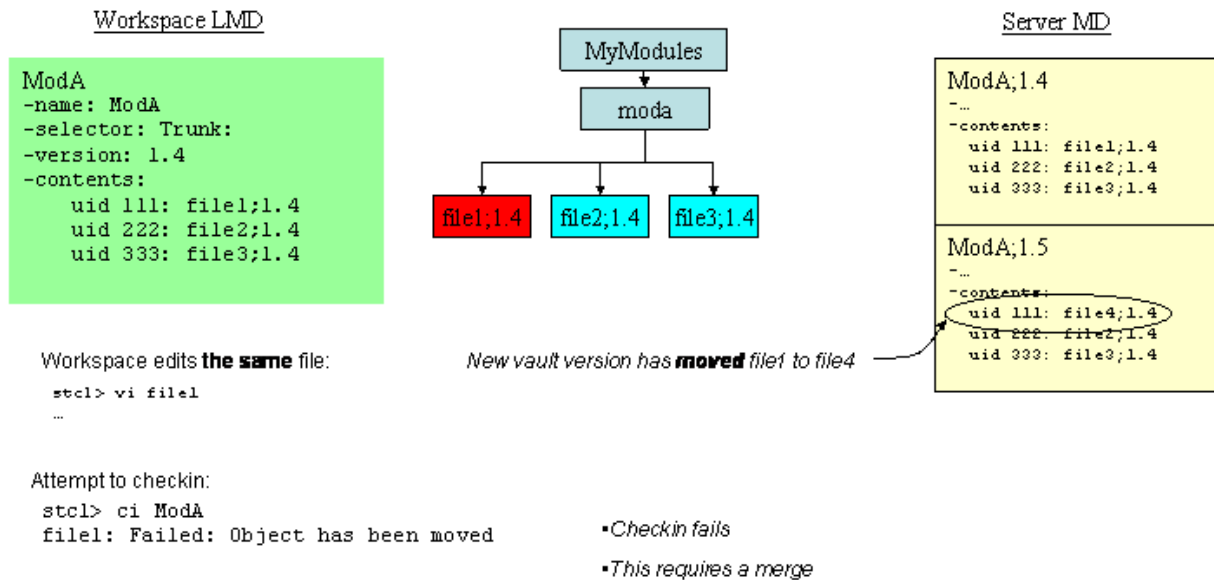


Step 3 of 4

View the next step

#### Step 4: Auto-Merging Locally Modified Files Renamed in the Module

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read a description of this use case.



Step 4 of 4

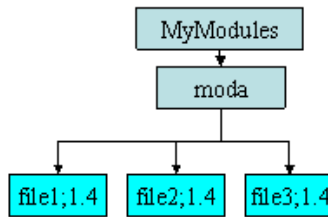
## Auto-Merging Locally Modified Files with Other Files Renamed in the Module

### Step 1: Auto-Merging Locally Modified Files with Other Files Renamed in the Module

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read a description of this use case.

## Workspace LMD

```
ModA
-name: ModA
-selector: Trunk:
-version: 1.4
-contents:
  uid 111: file1;1.4
  uid 222: file2;1.4
  uid 333: file3;1.4
```



## Server MD

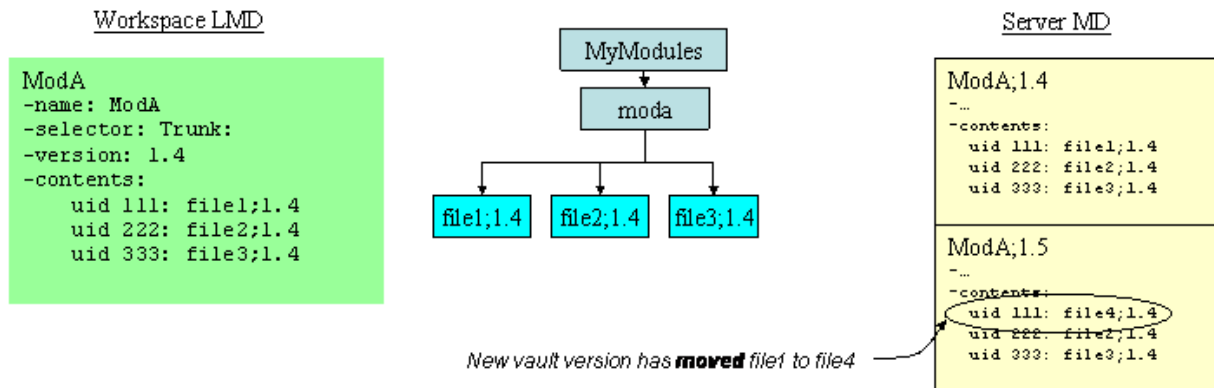
```
ModA;1.4
-...
-contents:
  uid 111: file1;1.4
  uid 222: file2;1.4
  uid 333: file3;1.4
```

Step 1 of 4

View the next step.

### Step 2: Auto-Merging Locally Modified Files with Other Files Renamed in the Module

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read a description of this use case.

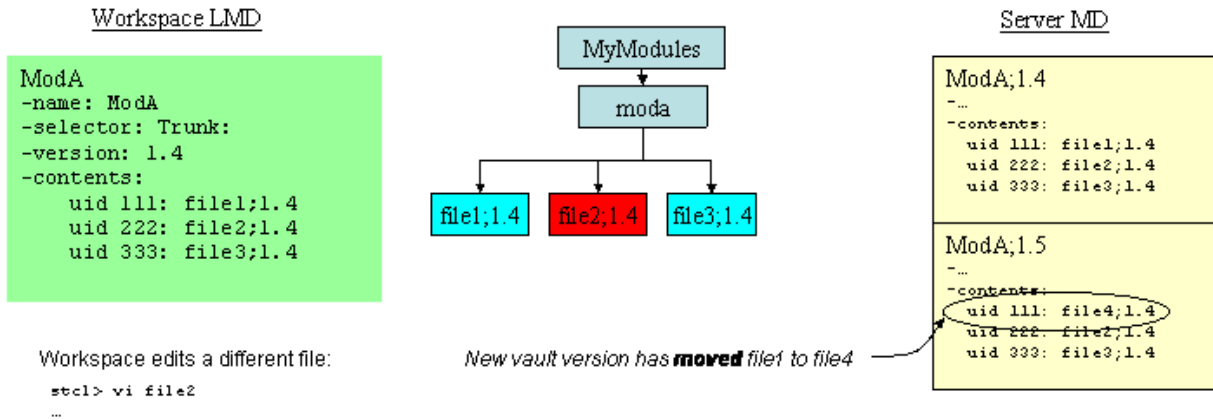


Step 2 of 4

View the next step

**Step 3: Auto-Merging Locally Modified Files with Other Files Renamed in the Module**

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read a description of this use case.

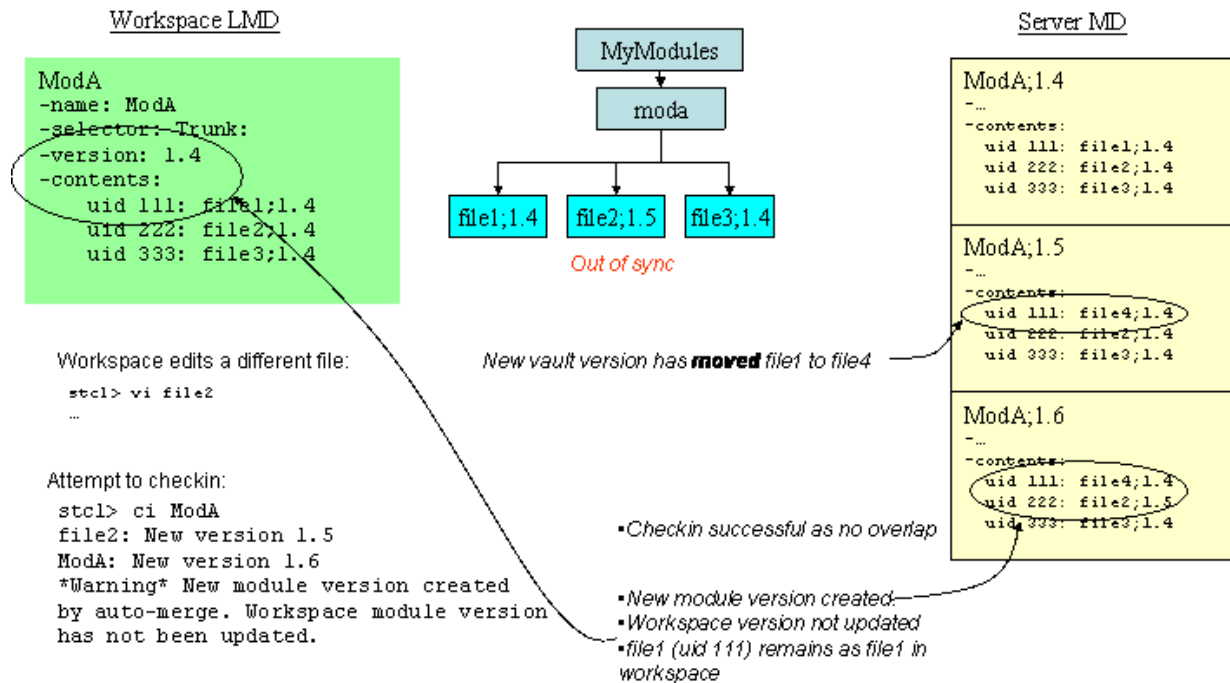


Step 3 of 4

View the next step

#### Step 4: Auto-Merging Locally Modified Files with Other Files Renamed in the Module

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read a description of this use case.

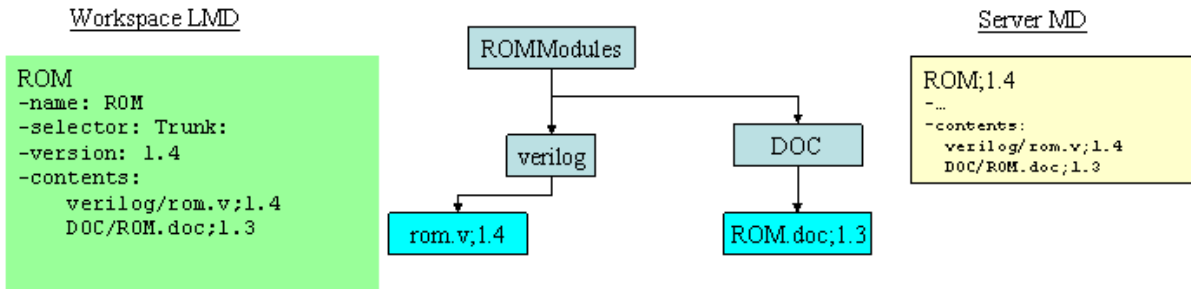


Step 4 of 4

## In-Branch Merging of Locally Added Files

### Step 1: In-Branch Merging of Locally Added Files

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read a description of this use case.



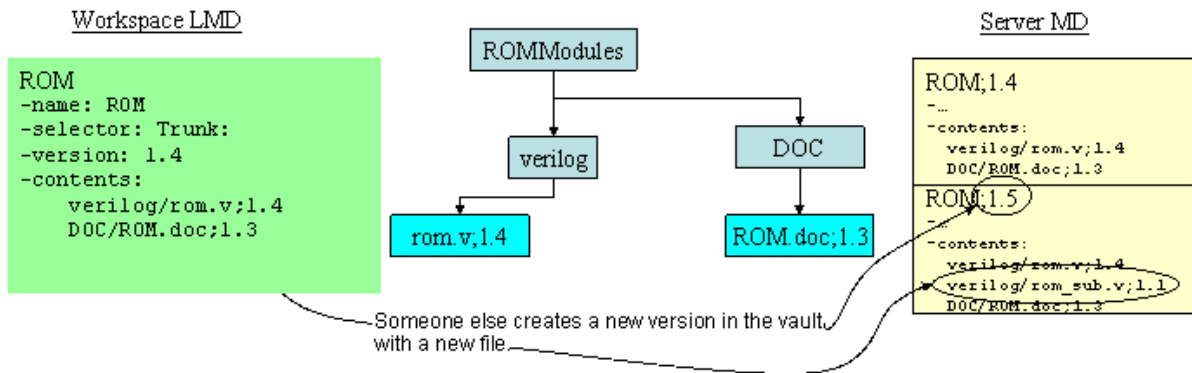
Step 1 of 6

View the next step.

### Step 2: In-Branch Merging of Locally Added Files

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read a description of this use case.



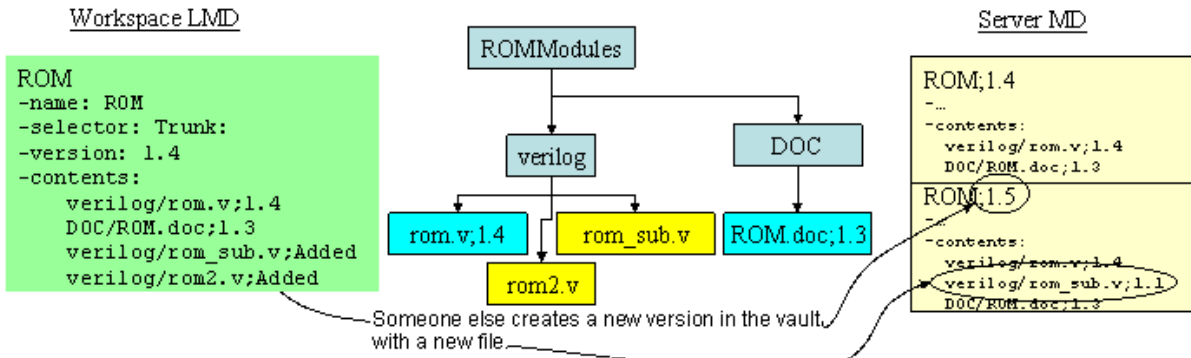


Step 2 of 6

View the next step.

### Step 3: In-Branch Merging of Locally Added Files

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read a description of this use case.



I also create the same file, plus another, and add them:

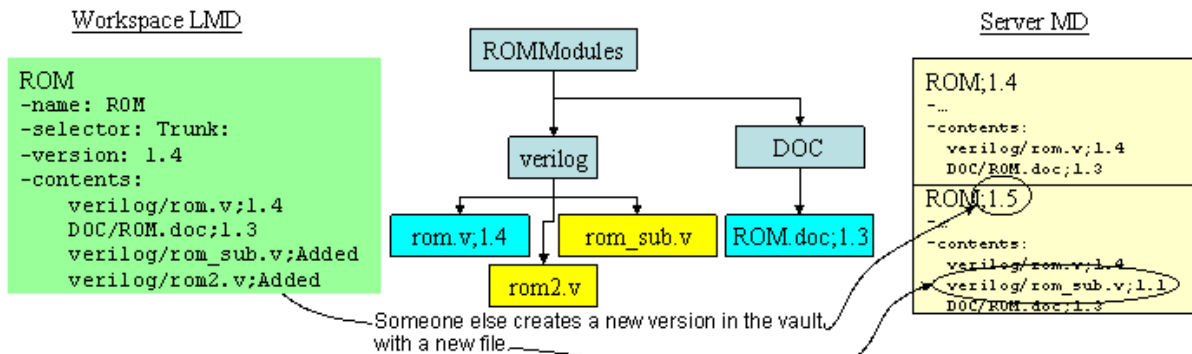
```
stcl> vi verilog/rom_sub.v verilog/rom2.v
stcl> hcm add -recursive verilog
```

Step 3 of 6

View the next step.

#### Step 4: In-Branch Merging of Locally Added Files

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read a description of this use case.



I also create the same file, plus another, and add them:

```
stcl> vi verilog/rom_sub.v verilog/rom2.v
stcl> hcm add -recursive verilog
```

Attempt to populate -merge new module version:

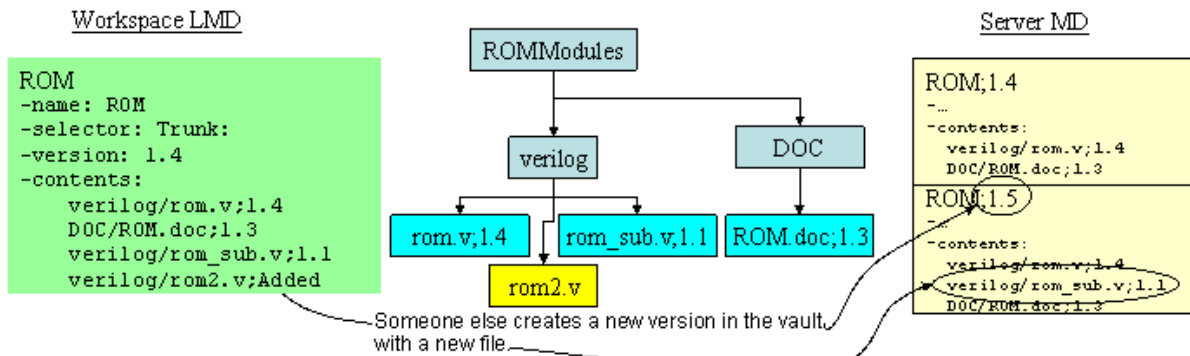
```
stcl> populate -merge ROM
verilog/rom_sub.v: failed, same file added locally. *Fails on added files
```

Step 4 of 6

View the next step.

### Step 5: In-Branch Merging of Locally Added Files

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read a description of this use case.



I also create the same file, plus another, and add them:

```

stcl> vi verilog/rom_sub.v verilog/rom2.v
stcl> hcm add -recursive verilog
  
```

Attempt to populate -merge new module version:

```

stcl> populate -merge ROM
verilog/rom_sub.v: failed, same file added locally.    •Fails on added files
  
```

Delete the local file, and re-populate with merging:

```

stcl> rmfile verilog/rom_sub.v
stcl> populate -merge ROM
verilog/rom_sub.v: Fetched version 1.1
  
```

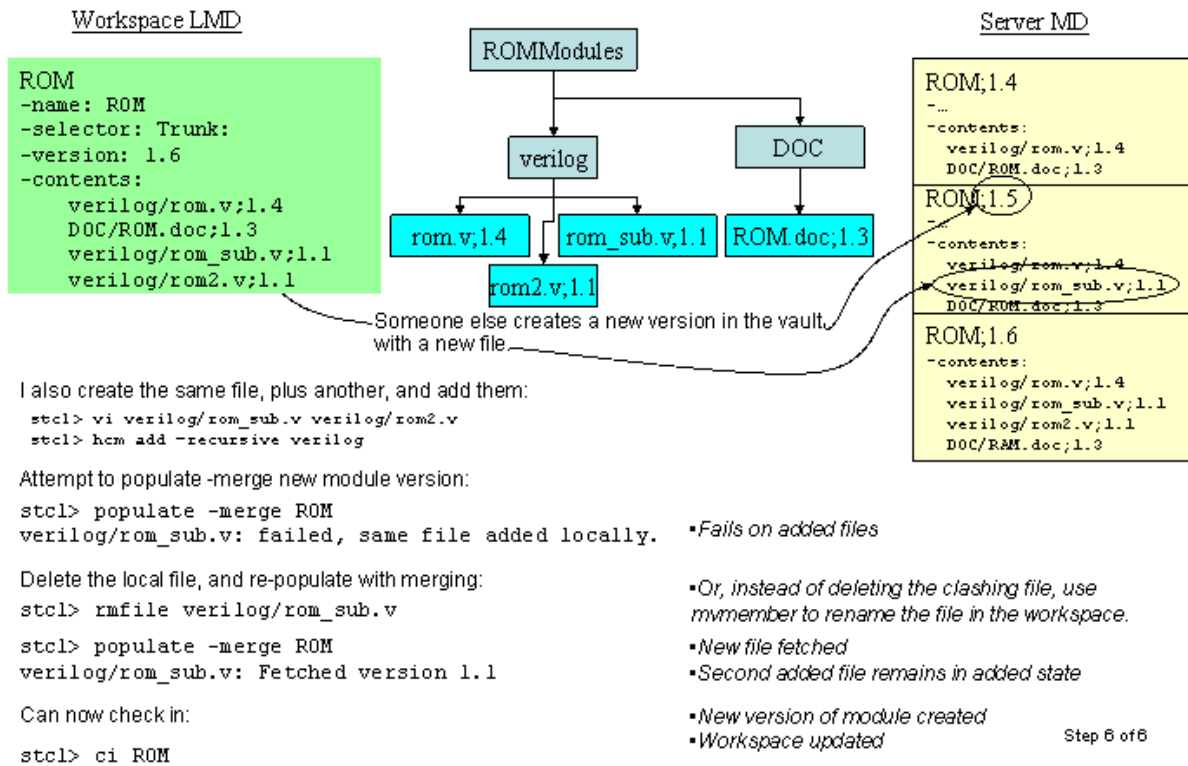
- Or, instead of deleting the clashing file, use mvmember to rename the file in the workspace.
- New file fetched
- Second added file remains in added state

Step 5 of 6

View the next step.

## Step 6: In-Branch Merging of Locally Added Files

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read a description of this use case.

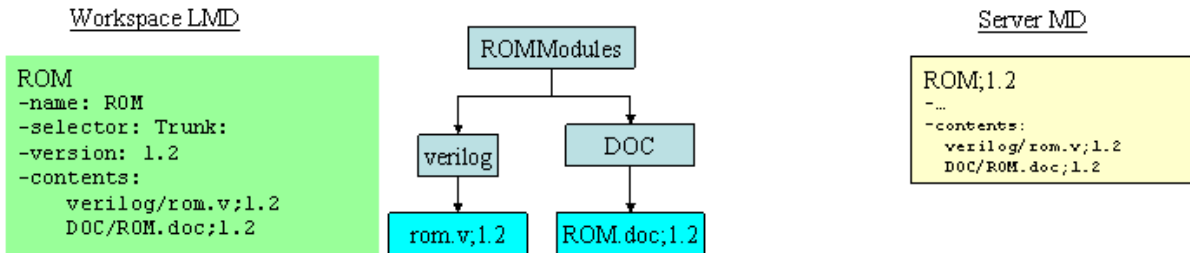


## In-Branch Merging of Locally Modified Files

### Step 1: In-Branch Merging of Locally Modified Files

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read a description of this use case.

# DesignSync Data Manager User's Guide

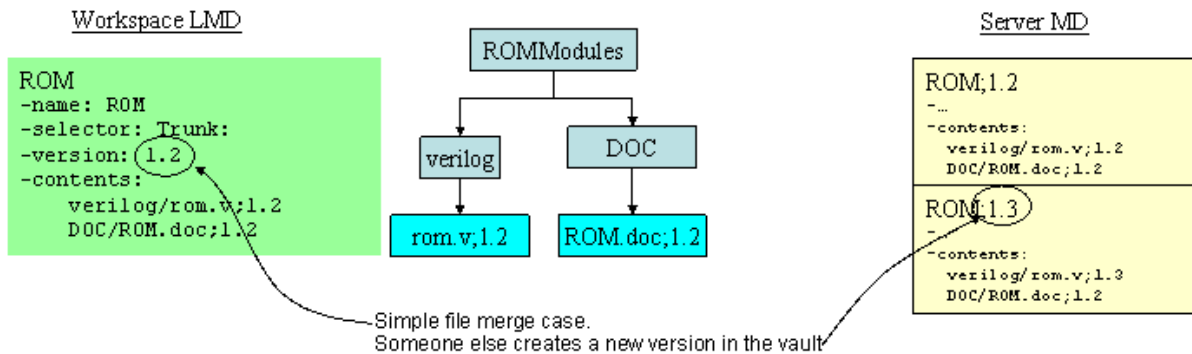


Step 1 of 6

View the next step.

## Step 2: In-Branch Merging of Locally Modified Files

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read a description of this use case.

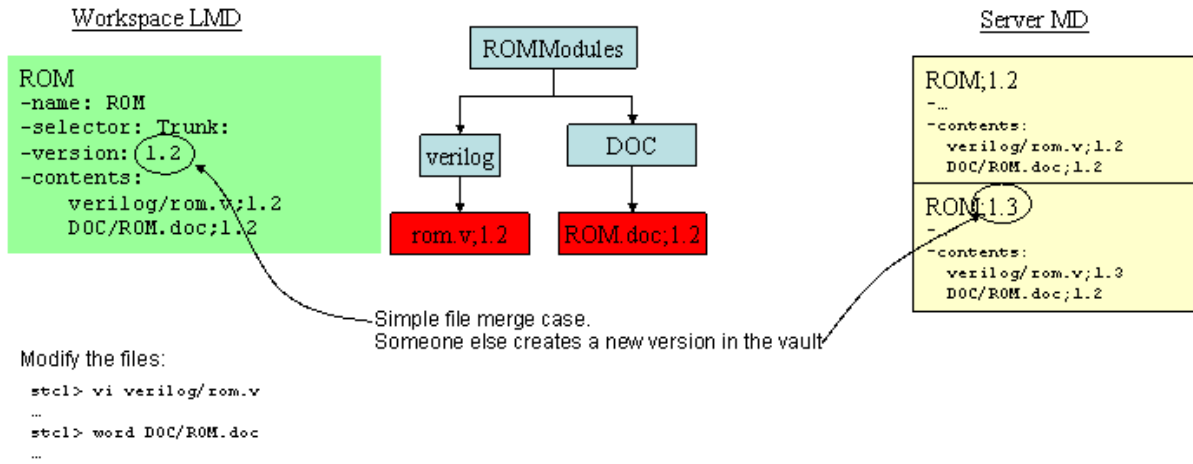


Step 2 of 6

View the next step.

### Step 3: In-Branch Merging of Locally Modified Files

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read a description of this use case.



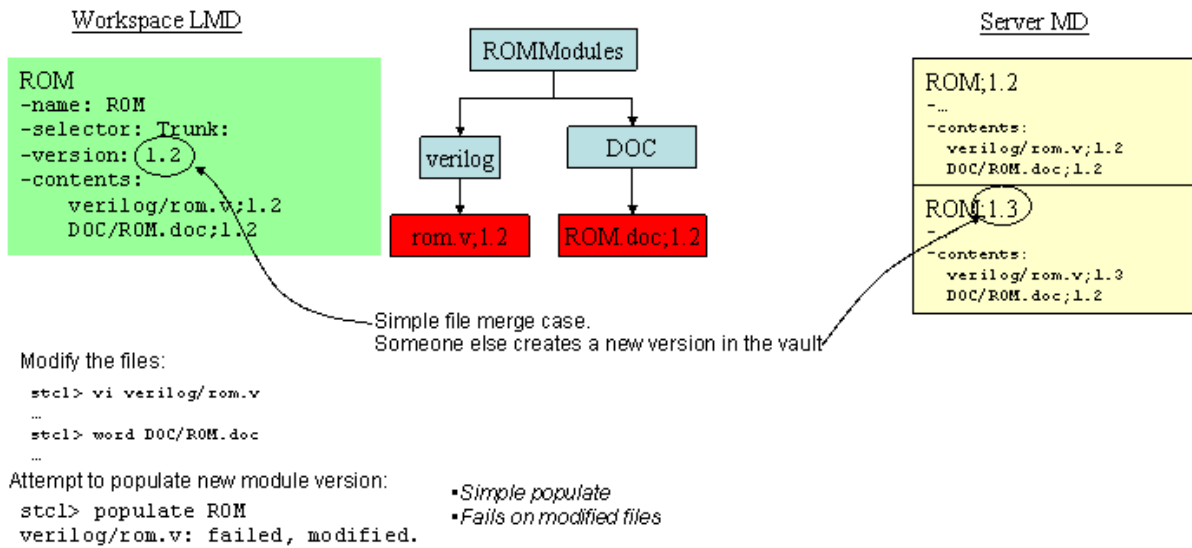
Step 3 of 6

View the next step.

#### Step 4: In-Branch Merging of Locally Modified Files

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read a description of this use case.



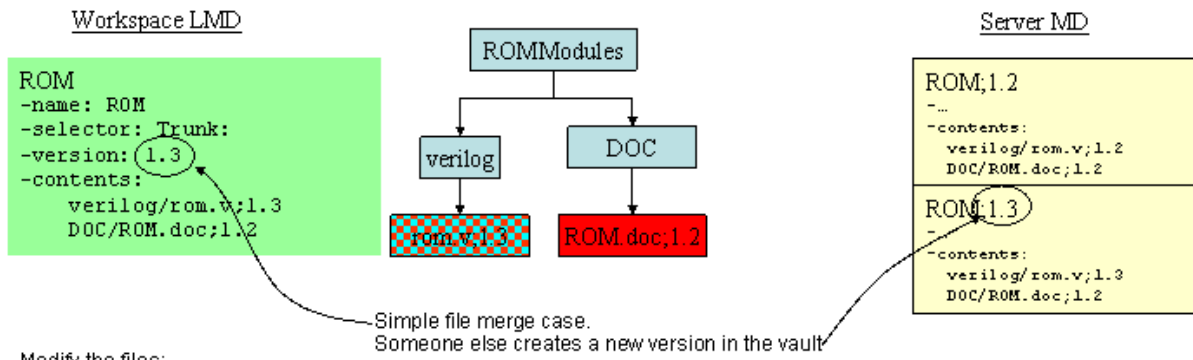


Step 4 of 6

View the next step.

### Step 5: In-Branch Merging of Locally Modified Files

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read a description of this use case.



Modify the files:

```
stcl> vi verilog/rom.v
...
stcl> word DOC/ROM.doc
...
```

Attempt to populate new module version:

```
stcl> populate ROM
verilog/rom.v: failed, modified.
```

- Simple populate
- Fails on modified files

Populate with merging:

```
stcl> populate -merge ROM
verilog/rom.v: changes merged...
```

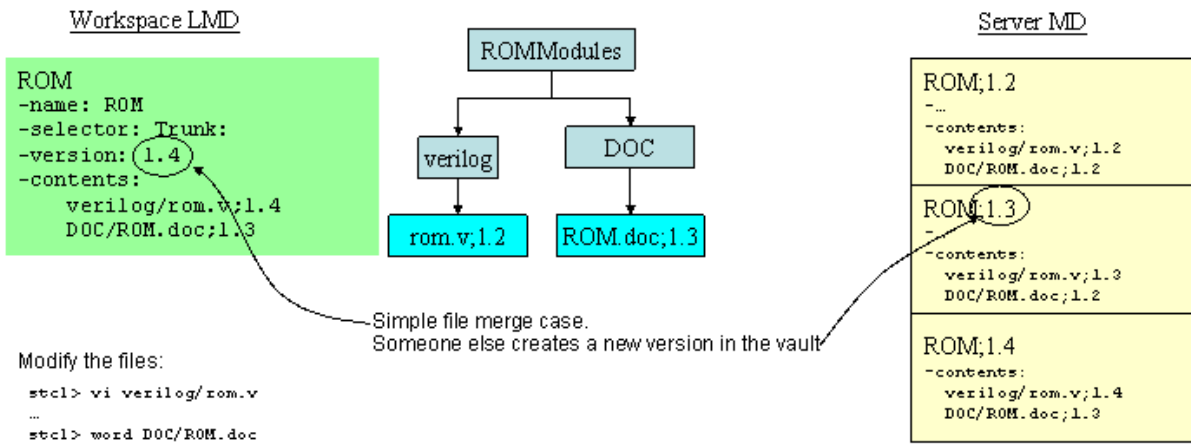
- Changes merged in, may lead to in-file conflicts
- Successful populate, module/file version updated
- rom.v remains modified.

Step 5 of 6

View the next step.

### Step 6: In-Branch Merging of Locally Modified Files

In this use case, "Workspace LMD" refers to the local metadata. "Server MD" refers to the server metadata. Read a description of this use case.



Modify the files:

```
stcl> vi verilog/rom.v
...
stcl> word DOC/ROM.doc
...
```

Attempt to populate new module version:

```
stcl> populate ROM
verilog/rom.v: failed, modified.
```

Populate with merging:

```
stcl> populate -merge ROM
verilog/rom.v: changes merged...
```

Can now check in:

```
stcl> ci ROM
```

Simple file merge case.  
Someone else creates a new version in the vault

- Simple populate
- Fails on modified files

- Changes merged in, may lead to in-file conflicts
- Successful populate, module/file version updated
- rom.v remains modified.

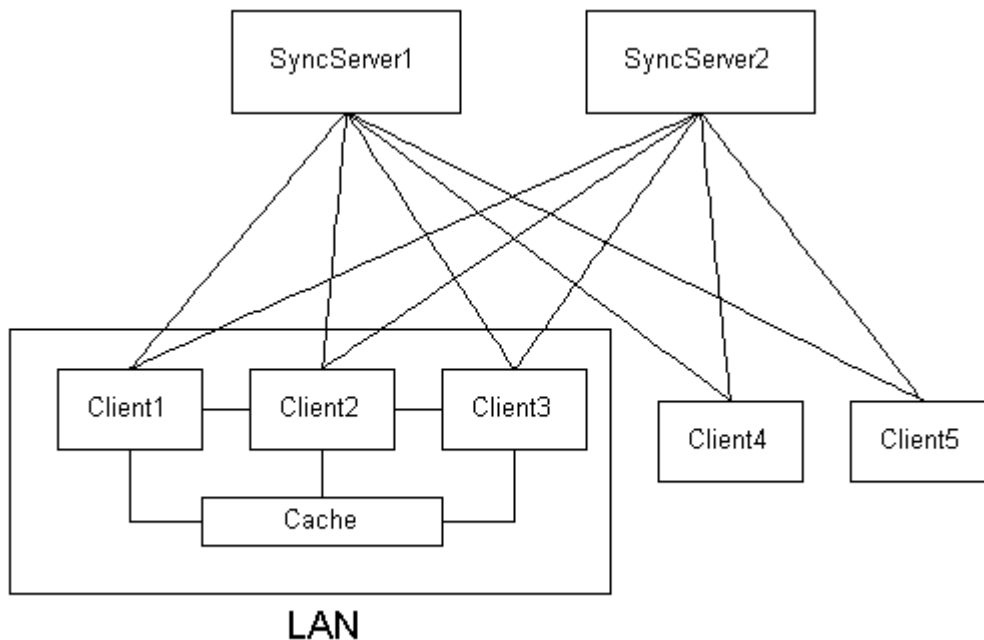
- New version of module created
- Workspace updated

# Reference

## Understanding the DesignSync Architecture

### DesignSync Architecture

DesignSync is a client/server groupware application similar in architecture to WWW browsers and the servers with which they communicate. DesignSync servers, known as SyncServers, manage shared information, control access to design files, and perform administrative functions such as user authentication and user privilege validation. For installations where many or all of the users are on the same LAN with a cross-mounted file system, it may be possible to increase performance through the use of a LAN cache. Also, DesignSync mirrors provide another mechanism for sharing files.



Like WWW servers and browsers, a DesignSync client can communicate with any number of SyncServers, anywhere on a LAN, WAN, or the Internet.

When you request a revision-control operation, the request is passed to the server, where the operation takes place. Once requested, server operations will complete, even when the connection between the client and the server is disconnected before the server has sent confirmation to the client.

While most of the information about a revision-controlled project is stored on the SyncServer, some data is local to the client. This client-side metadata is stored in `.SYNC` folders that reside in the parent folder of any local DesignSync-associated object.

**CAUTION:** DesignSync manages these .SYNC metadata folders; do not directly manipulate these folders or their contents.

#### Related Topics

What Is a SyncServer?

What is a LAN Cache?

Metadata Overview

*DesignSync Data Manager Administrator's Guide: Mirroring Overview*

*DesignSync Data Manager Administrator's Guide: Access Control Overview*

## What Is a SyncServer?

DesignSync is a client/server groupware application similar in architecture to WWW browsers and the servers with which they communicate. DesignSync servers, known as **SyncServers**, manage shared information, control access to design files, and perform administrative functions such as user authentication and user privilege validation.

#### Related Topics

Accessing a SyncServer: User Authentication

DesignSync Data Manager Administrator's Guide: SyncServer List Files

## Object States

You can populate or check out design files into your local workspace in one of five states: locked, unlocked, reference, locked reference, link to the cache, or link to the mirror. You can have any mix of states in the same folder. What state or states you want your files in depends on your design methodology.

State	Description
Locked	<p>A <b>locked</b> object is the original object. When you create a new object, such as a file, the object also is the original.</p> <p>When you check out a file with lock, your work area contains the specified version of the original file.</p>
Unlocked	<p>An <b>unlocked</b> object is a replica of another object. Replication is different from the normal copying process. A "copy," in the usual sense, is an independent object; it is not associated with the object</p>

	<p>from which it was copied. An unlocked copy, on the other hand, retains an association with the original from which it was copied.</p> <p>When you check out a file without lock, your work area contains a replica of the file.</p>
Reference	<p>A <b>reference</b> is an object that is not physically present, but instead points to another object. Although no physical files correspond to a reference, DesignSync metadata keeps track of the revision-control information for the reference. Just as with unlocked copies, the object that the reference points to is the original.</p> <p>When you check an object into the vault and choose to retain a reference in your work area, DesignSync metadata maintains a connection between your working area and the object in the vault.</p> <p>You also can create a locked reference to ensure that the referenced object cannot be updated by another user.</p>
Link (UNIX only)	<p>A <b>link</b> can be either a hard link or a symbolic link to another object. Links are useful when you want to easily access a file without having a copy in your work area - for example, when the file is very large. Links are available only on UNIX systems.</p> <p>You can have links to a shared cache or to a mirror directory.</p> <p><b>Note:</b> The link icon is used for any link in your work area, whether the link is a DesignSync-created link to the cache or mirror or a link to a file or folder that DesignSync did not create. Use the <b>Type</b> field in the List View to determine the revision-control state of a link.</p>

Your project leader can specify the state objects in your work area should be in when they are not being edited (the fetch state). See the *DesignSync Data Manager Administrator's Guide: Default Fetch State* for more information.

### Saving the Setting of an Object's State

You can have DesignSync save the object state settings you specify. During a **checkin**, **checkout**, or **populate** operation, take these steps:

1. In the **Check In**, **Check Out**, or **Populate** dialog box, specify the object state you want.
2. Click **Save Settings**.

Then each time you bring up that dialog box, DesignSync displays the object state as selected. This is the default behavior, even when default fetch state has been defined.

You can also set a default fetch state, which will apply to both command line operations and the DesignSync GUI. For details, see *DesignSync Data Manager Administrator's Guide: Default Fetch State*.

### Related Topics

*DesignSync Data Manager Administrator's Guide: How DesignSync Handles Symbolic Links*

*DesignSync Data Manager Administrator's Guide: Default Fetch State*

## Object Types

The term object most often refers to file system objects such as files or folders (directories). The term object can also be used in a more general sense - for example, to refer to any object identifiable by an Internet Uniform Resource Locator (URL).

The following types of objects are commonly used in DesignSync:

Object Type	Description
Branch	A thread of development that emanated from a version. The branch itself typically contains versions.
Branch point version	A version which is the root of a branch from which other versions emanate.
Category	Provides a means of organizing modules. Specify a virtual path category to group related modules. For example, if you work with two different types of projects, Chip design and CPU design, you can create two categories /Chip and /CPU to store the different modules for each type of project. These paths do not map to actual paths on the server.
Collection	A group of files treated by DesignSync as a single versionable object.
File	File system file.
Folder	File system directory.
Hierarchical Reference	A reference, or connection, from an upper-level module to any of the following object types: a submodule, branch or version, a legacy submodule configuration, a DesignSync vault, or an IP Gear deliverable.
IP Gear deliverable	A package of data that has been uploaded and associated with an IP Gear Catalog Component.
Module	A collection of managed objects that together make up a single

	entity.
View	A named set of rules that defines what module members are filtered in a module workspace. By using a module view, DesignSync administrators can provide a set of common filters available to all members of a project or group.
Module Cache	A folder containing a shared copy of a module.
Link to Mcache	A symbolic link that points to the base directory of the module in the module cache.
Cache	A symbolic link that point to a file cache object.
Cache-hardlinked	A hard link reference that points to a file cache object.
External Module	A reference to an external module.
Vault	Contains files you have checked in and their versions; also contains branches. Vault is also used to mean the default vault associated with a folder (directory).
Version	An immutable snapshot of a file at a particular point in time.

## Object Properties

### Viewing and Setting Properties

The **Properties** dialog box lets you view and set properties for a selected object. You select the object whose properties you want to view, then can either:

- Select **File =>Properties**.
- Click the **Properties** button.
- Right-click on the object and select **Properties** from the context pop-up menu.

The information available from the **Properties** dialog box depends on the type of object you have selected. You can also select multiple objects, but typically less meaningful information is displayed when multiple objects are selected.

Depending on the object selected, one or more of these tabs could display:

- The General tab provides general information.
- The Revision Control tab shows revision status.
- The Modules tab shows module information.
- The Version tab that shows version status.
- The Tag shows all of the branch tags, and all of the version tags, that are associated with the object.
- The Collection tab displays the collection members.



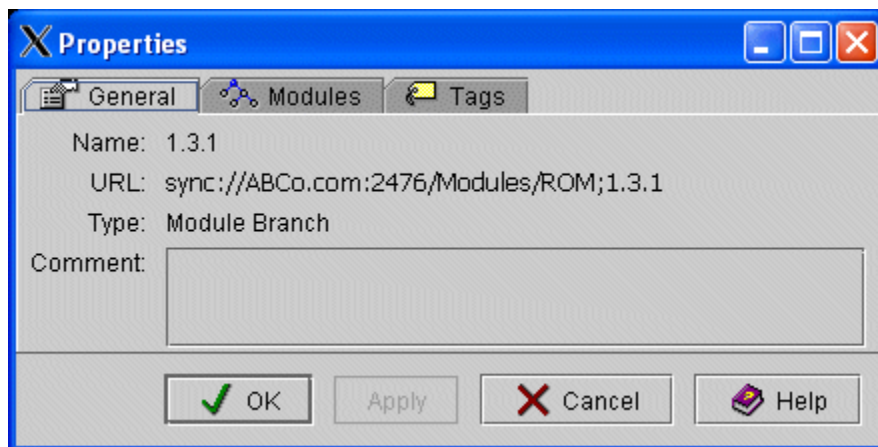
You can also display properties for Public Projects that have been defined for you.

## General Properties

The **General** tab of the **Properties** dialog box provides general information about an object that you have selected.

**Note:** When displayed, the only field you can edit is the **Description** field, where you can enter a description of the object. Any description you enter here is separate from comments associated with revision control operations such as checkin and checkout. It is not propagated to the vault and is only associated with the local file. Other users who access the object from a shared workspace can view this description. Also, this description does not apply to any Potential Checkouts since they do not exist in the workspace.

For managed objects, you can enter revision notes in the **Description** text box. The notes in the Log, if any, also appear in the object's data sheet. The Log is a workspace for you to build check-in comments. When you check out an object, the Log for that object is seeded with your check-out comment. You can add to and edit the Log any number of times while you have the object checked out. When you check in the object, the comment you specify with the check-in command, if any, is appended to the Log to form the complete check-in comment. Therefore, to check in an object without any comment, clear the Log prior to the check-in command. Note that the content of the Log does not contribute to the minimum comment-length requirement, if any. Following the check-in, DesignSync clears the contents of the Log.



**Note:** When viewing the General tab for hierarchical references, you see the following notation for URL: <path>/<ModuleName>#<HierarchicalReferenceName>.

### Related Topics

Revision Control Properties

Module Object Properties

Version Properties

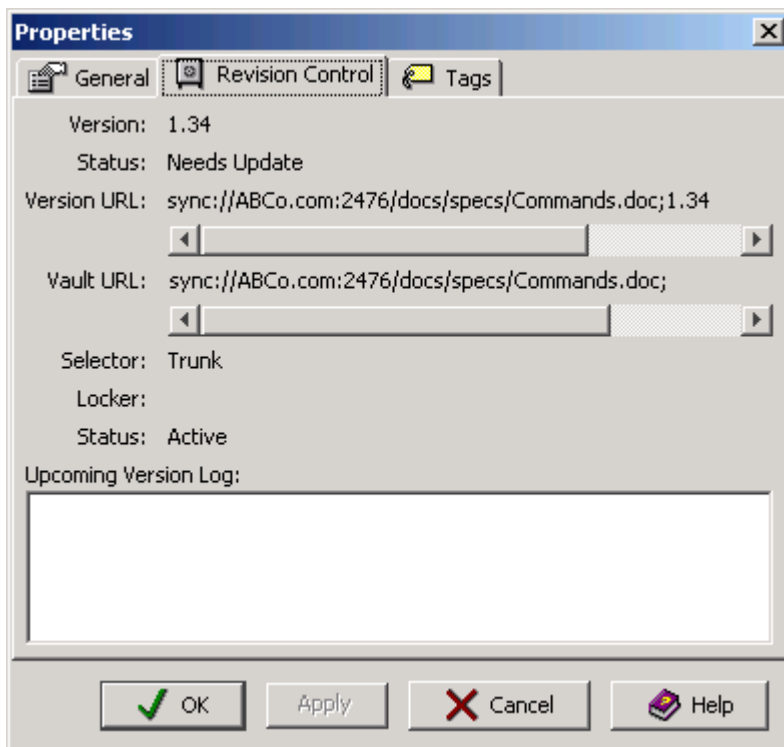
Tag Properties

Displaying Project Properties

Viewing and Setting Properties

## Revision Control Properties

The **Revision Control** tab of the **Properties** dialog box provides revision-control information about an object that you have selected. The information available depends on the object or objects you have selected.



When viewing the properties of a folder, you can initially set the vault location for the folder or change the vault location already associated with the folder. If you cannot see the entire vault location, place your cursor in the field and move it to the right.

### Related Topics

[ENOVIA Synchronicity Command Reference Help: setvault command](#)

[Specifying the Vault Location for a Design Hierarchy](#)

[Changing the Vault for a Design Hierarchy](#)

General Properties

Module Object Properties

Version Properties

Module Object Properties

Version Properties

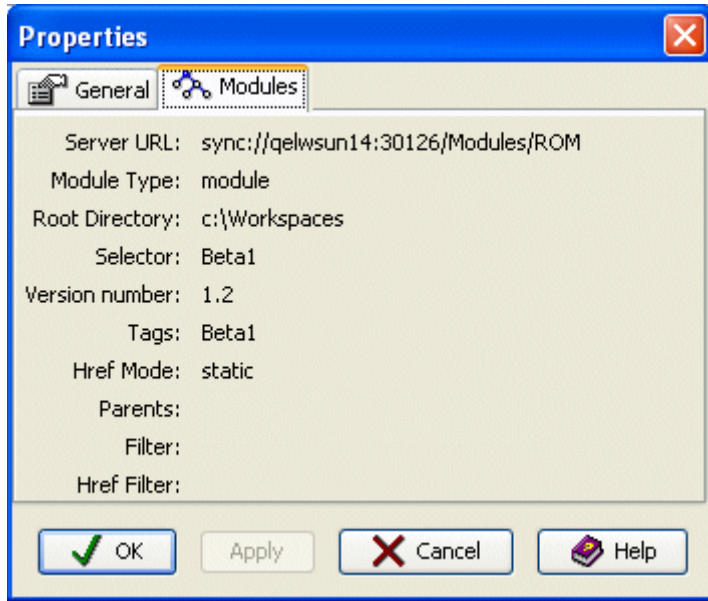
Tag Properties

### Module Objects Properties

The **Modules** tab of the **Properties** dialog box provides revision-control information about modules and modules objects. The information available depends on the object you selected and whether it was selected from the client or the server. You can display module object properties for the following module object types:

Href Node (Server)	Module Node (Server)
Legacy Configuration Node (Server)	Module Version Node (Server)
Legacy Href Node (Server)	Modules Node (Server)
Legacy Module Base Node (Client)	Module Base Node (Client)
Legacy Module Node (Server)	Module Href Node (Client)
Legacy Modules Node (Server)	Module File Member Node (Client and Server)
Module Branch Node (Server)	Module Folder Member Node (Client and Server)
Module Instance Node (Client)	

The Modules tab on the Properties dialog box provides the following information, as appropriate:



<b>Action/Option</b>	<b>Result</b>
<b>Server URL</b>	The module's address on the server.
<b>Module Type</b>	The type of object selected: module, mcacheinst or external.
<b>Root Directory</b>	The location of the workspace root directory on the client.
<b>Selector</b>	The identifying expression used to fetch the module.
<b>Version Number</b>	The module's numeric identifier on the server.
<b>Tags</b>	The module's user-designated identifiers.
<b>Href Mode</b>	The mode associated with the module's hierarchical reference that was used to fetch the module: normal, static or dynamic.
<b>Parents</b>	If the module is hierarchically references from other modules, the names of those modules.
<b>Filter</b>	A specified expression used to identify the exact subset of module members on which the command will operate.
<b>Href Filter</b>	A specified expression used to exclude hierarchical references that will be followed when operating on a module recursively.

Related Topics

General Properties

Revision Control Properties

Displaying Collections

Displaying Project Properties

Version Properties

Tag Properties

## Version Properties

The **Version** tab of the **Properties** dialog box provides the author and revision log information about an object that you have selected in the vault.

### Related Topics

General Properties

RevisionControl Properties

Module Object Properties

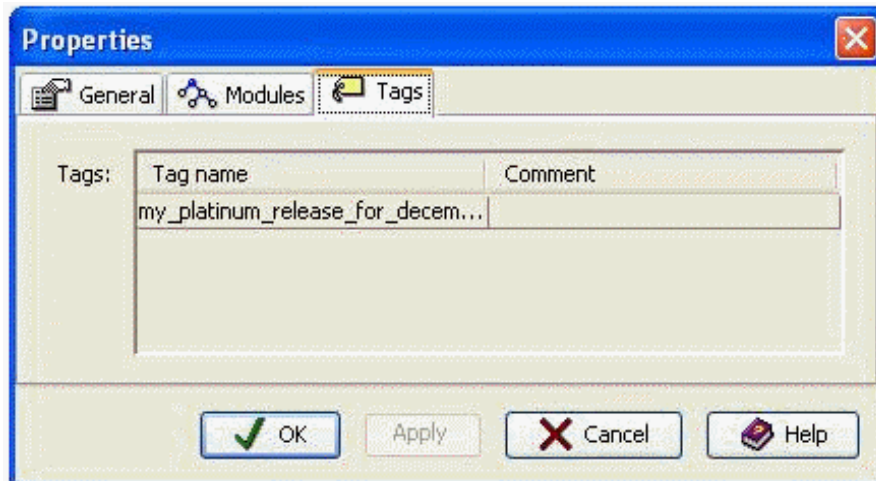
Tag Properties

Displaying Project Properties

Viewing and Setting Properties

## Tags Properties

The **Tags** tab of the **Properties** dialog box shows all of the branch tags, and all of the version tags, that are associated with the object.



If the tag name is too long, you can mouse over to the column splitter and expand the tag name field by grabbing and then moving the column splitter indicator.

### Related Topics

[General Properties](#)

[RevisionControl Properties](#)

[Module Object Properties](#)

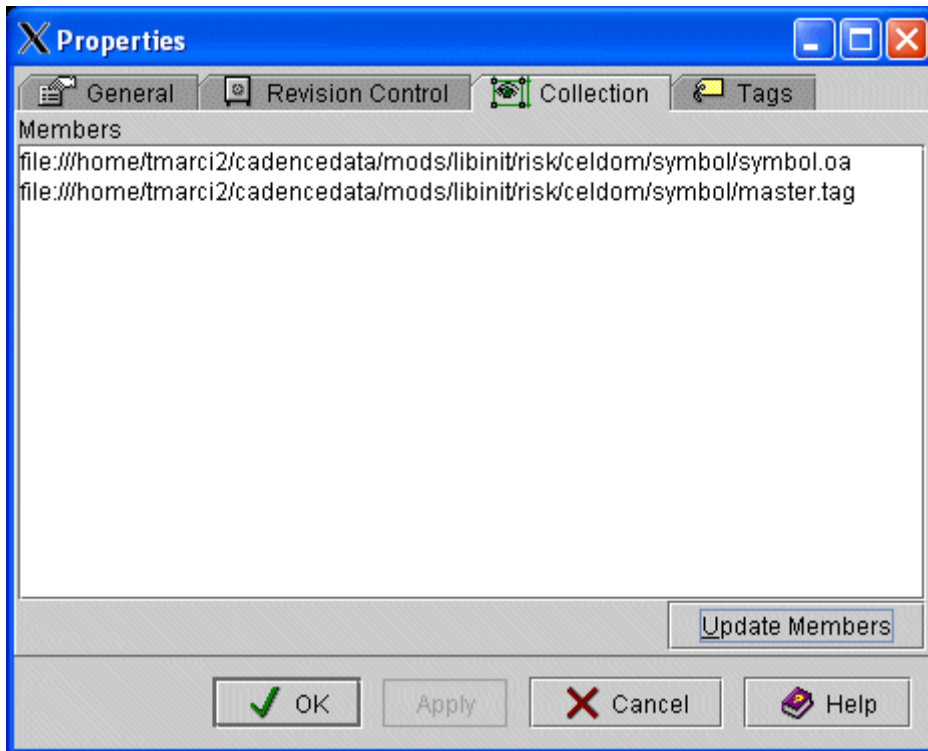
[Version Properties](#)

[Displaying Project Properties](#)

[Viewing and Setting Properties](#)

### Collection Properties

The **Collection** tab of the **Properties** dialog box provides a list of the members of a collection. The **Collection** tab appears when the properties of a View object are displayed.



### Collection Properties Field Descriptions

#### Members

The list of members present in the view. The list displays the full workspace path of the view members.

#### Update Members

Refreshes the list of members.

#### Related Topics

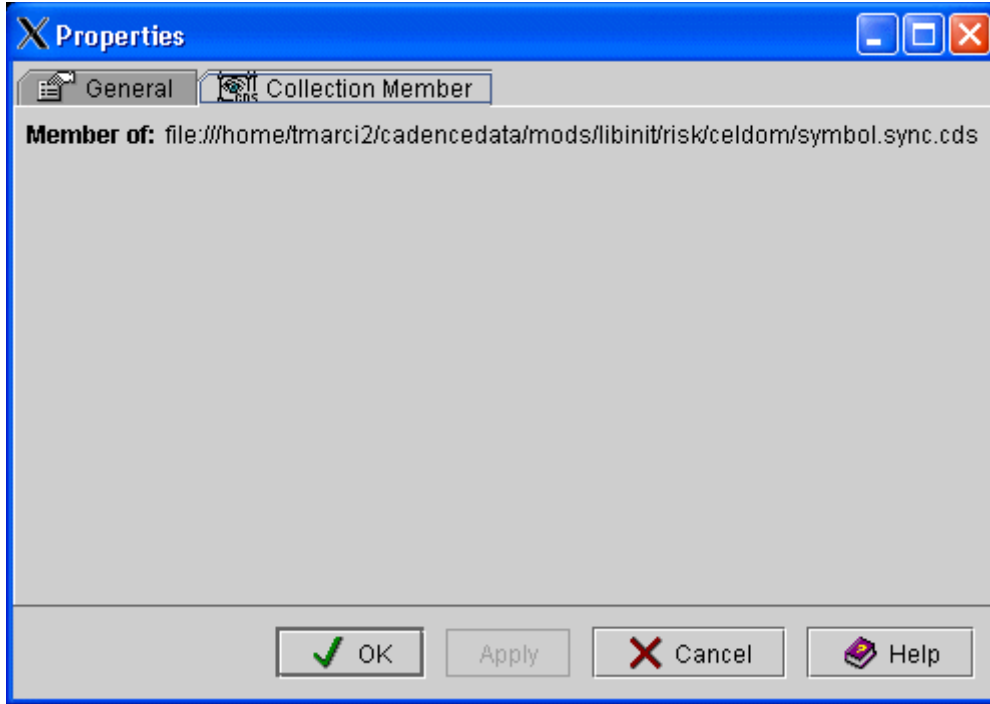
Collections Overview

Displaying Collections

Collection Member Properties

### Collection Member Properties

The **Collection Member** tab of the **Properties** dialog box provides the workspace address of the collection to which the specified member belongs.



**Related Topics**

Collections Overview

Displaying Collections

Collection Properties

**URL Syntax**

The table below shows examples of URL syntax for various types of objects you can access through the SyncServer. The portion **host:port** represents the server's **ip address:port number** or **symbolic DNS address:port number**.

The ellipses (...) in the URLs represent the path to the object ( file, module, folder, vault, version, branch, or branch-point version) being accessed, which should include a leading slash.

Object Type	URL
File	file:///.../file_name  <b>Example:</b>  file:///home/bob/work/mod1/block



Folder	<pre>file:///.../folder_name</pre> <pre>sync://host:port/.../folder_name</pre> <p><b>Example:</b></p> <pre>file:///home/bob/work/mod1</pre> <pre>sync://linus.appco.com:2647/Projects/gemini/block1</pre>
Module	<pre>sync://host:port/Modules/[category/...]module_name</pre> <p><b>Example:</b></p> <pre>sync://linus.appco.com:2647/Modules/ProjectA/ChipModules/ ALU</pre>
Module in the Modules View	<p>For a module instance:</p> <pre>module://module_instance_name/full_path_to_module_base_directory/</pre> <p>For a module root:</p> <pre>module:///full_path_to_module_root_directory/</pre> <p><b>Examples:</b></p> <pre>module://ALU%0/c Workspace/Chip1/ALU</pre> <pre>module:///c Workspace/Chip1</pre>
Module Version	<pre>sync://host:port/Modules/[category/...]module_name;version</pre> <p><b>Example:</b></p> <pre>sync://linus.appco.com:2647/Modules/ProjectA/ChipModules/ ALU;1.4.1</pre>
Version	<pre>sync://host:port/.../vault_name;[branchtag:]version</pre> <p><b>Examples:</b></p> <pre>sync://linus.appco.com:2647/Projects/gemini/block1/top.v;1.3</pre> <pre>sync://linus.appco.com:2647/Projects/gemini/block1/top.v;Gold:Latest</pre>
Branch-Point	<pre>sync://host:port/.../vault_name;1.1</pre>

Version	<p><b>Example:</b></p> <p>sync://linus.appco.com:2647/Projects/gemini/block1/top.v;1.3</p>
Branch	<p>sync://host:port/.../vault_name;1.1.1</p> <p><b>Example:</b></p> <p>sync://linus.appco.com:2647/Projects/gemini/block1/top.v;1.3.1</p>
External Module	<p>sync:///ExternalModule/&lt;external-type&gt;/&lt;external-data&gt;</p>

**Tip:** You can differentiate between a branch numeric and a version numeric by counting the number of segments. A version numeric has an even number of segments like 1.1 or 1.2.1.3. A branch numeric has an odd number like 1.1.1 or 1.2.13.4.

### Reserved Characters

DesignSync object names, including (but not limited to) file-based design objects, collections, module, and category names may only contain printable characters and cannot contain a space, or any of the following symbols:

~ ! ? @ # \$ % ^ & \* ( ) , ; : | ` ' " = [ ] / \ < >

Because these symbols are used for specific purposes in URLs, generated mirror names, and datasheets, and are illegal for modules, you should avoid using them in other DesignSync elements as well to minimize confusion.

Using SyncAdmin, you, or the system or site administrator can restrict the natural path of the object so that neither the object itself, or any folders in the object path can contain these symbols. For more information, see Exclude Lists in the DesignSync Data Manager Administrator's Guide.

## Reserved File Extensions

Do not use the following strings in your URL names, as either files or directory components:

1. `.sgc`  
`.sync`

These strings are reserved for use by DesignSync's Custom Type System. See the DesignSync Custom Type System Programmer's Guide: Custom Type System for more information.

## DesignSync URLs

**URLs** (Uniform Resource Locators) give you a way to identify every object in the computing world. The URL of an object specifies what computer the object is on, where the object is on that computer, and how to access it.

Most URLs you see on the World Wide Web begin with the string **http**. This string represents the protocol portion of the URL. The protocol tells the server receiving the request how to process the URL -- in this case a request for an HTML page residing somewhere on the server's file system.

Instead of the http protocol, DesignSync uses a special protocol, **sync**, to allow it to instruct a **SyncServer** how to distinguish and handle requests for information. Using the sync protocol, DesignSync can address objects such as directories or files, anywhere on the Internet, by specifying a URL with either of the following formats:

**sync://IPAddress:portnum/path\_to\_object**

**sync://host.domain:portnum/path\_to\_object**

For example:

**sync://208.196.5.5:2647/Projects/SCSI\_Proj**

or

**sync://host.mycompany.com:2647/Projects/SCSI\_Proj**

or if you are on the same LAN as "host":

## **sync://host:2647/Projects/SCSI\_Proj**

The default SyncServer port number is 2647. You can omit the port specification if the SyncServer is using this default port number.

All objects can be referenced in the DesignSync clients, and by the DesignSync server as a URL -- not only remote objects, but also objects on the local file system of the computer where you are running the DesignSync client. You address a local object with the **file** protocol, as in the following:

**file:///directory\_path/file\_name**

### **Note:**

DesignSync also supports a **syncs** protocol for communicating with secure (SSL) SyncServer ports. In most cases, DesignSync automatically redirects requests to a cleartext (non-secure) port using the **sync** protocol to the secure port, if one is defined. The default DesignSync secure port number is 2679. Your DesignSync administrator defines what SyncServer ports are available and whether secure communications are required. See Overview of Secure Communications for more information.

### **Related Topics**

URL Syntax

*DesignSync Data Manager Administrator's Guide: Using Secure Communications*

## **Revision Control Status Values**

In the List View Pane, the revision-control status of the objects are listed under status columns.

### **Notes:**

- For collections of data, the Status is reported only for the collection object, and not its member files.
- For module hierarchies with the persistent populate href mode of normal, the DesignSync interface uses the **Change traversal mode with static selector on top level module** set in SyncAdmin to determine which module version is expected in the workspace. For more information, see the Module Hierarchy topic.

<b>Values</b>	<b>Description</b>
Up-to-date	File is currently the correct version for the selector, or that the module member version matches the correct version for the module selector.
Locally Modified	File has been edited since it was fetched and a more recent

	version has not been checked into the branch, or that an add has been performed on a module member that has not been checked into the module.
Needs Merge [<change>]	<p>File has been locally modified and the version is not correct for the current selector.</p> <p>For modules, an additional value indicating the type of change may appear in brackets [ ] after the Needs Merge status value:</p> <ul style="list-style-type: none"> <li>• &lt;Version number&gt; - the version of the member in the module version.</li> <li>• Moved - the module member has a different natural path than the one expected by the module version.</li> <li>• Removed - the module member is not in the module version.</li> <li>• &lt;Version number&gt;,Moved - the module member is both a different version and located at a different natural path than the module version.</li> </ul>
Needs Update [<change>]	<p>Indicates that you have an incorrect version on the given branch.</p> <p>For modules, an additional value indicating the type of change may appear in brackets [ ] after the Needs Update status value:</p> <ul style="list-style-type: none"> <li>• &lt;Version number&gt; - the version of the member in the module version.</li> <li>• Moved - the module member has a different natural path than the one expected by the module version.</li> <li>• Removed - the module member is not in the module version.</li> <li>• &lt;Version number&gt;,Moved - the module member is both a different version and located at a different natural path than the module version.</li> </ul>
Unresolved Conflicts	Indicates that a merge of versions has resulted in conflicts.
Added	Indicates that the object has been added to the module, but has not been checked in.
Added By Merge, Needs Checkin	Indicates that the file was introduced to the work area by a merge or overlay operation and does not exist on the current branch. When you check in the file, the branch is created automatically.
Locally Moved	<p>Module member has moved in the workspace.</p> <p><b>Note:</b> Moved members display with this status even if they have also had their contents modified. There is no separate indication that the contents have changed.</p>

Locally Removed	Module member has been removed from the module.
Remove for merge	An object present in the workspace was removed, either by being removed from the module or by being retired on the branch being merged into the workspace.
Absent	Indicates an object that is unexpectedly absent from the workspace. Typically an object is listed as 'Absent' if it was deleted from the workspace using operating system commands, leaving behind the local metadata.
Unknown	<p>Indicates that the version of the file in the workspace cannot be determined from the local metadata. An object can be Unknown when:</p> <ul style="list-style-type: none"> <li>• The object has been modified locally but is not derived from the version in the vault. For example, when you remove an object and then recreate it. In this case, you can check in the object using the -skip option.</li> <li>• A reference has been retired and then the selector is changed. If you retire an object (without -keep), both the object and its metadata are removed from the local workspace.</li> <li>• A setselector operation then creates a metadata entry, but without a version number.</li> <li>• A broken network connection interrupts the check in of a new object. When the client loses its connection, the check in to the server can succeed, but the client may not receive results back from the server. This situation leaves managed files in the local workspace that do not have corresponding metadata.</li> </ul>
[Retired]	<p>This is not a state of its own; rather it is a prefix to one of the other states. Indicates that the current branch is retired. For example, the status column might contain:</p> <p><code>[Retired] Locally Modified</code></p> <p>Because the <code>[Retired]</code> status is a prefix to other status terms, sorting causes retired items to be grouped either at the beginning or end of the listing, independent of the items' local state.</p>

**Related Topics**

DesignSync Symbols and Icons

**Vaults, Versions, and Branches**

A vault is an object that holds versions of other objects, such as files. For example, if you create a new file and check it in, the checked in file would be identified as version 1.1 in the vault. If you check out 1.1, make changes to it, and check it back in, you create version 1.2. If you check out 1.2, make changes to it, and check it back in, you create version 1.3, and so on. Vault names always end in a semicolon ( ; ), for example: **top.v**; is the vault for **top.v**.

A vault also holds any branches that may be derived from versions of a file, as well as the versions on those branches. Branches allow for parallel development, where multiple design activities on the same design files take place simultaneously.

Folders (directories) are not revision controlled and therefore are not stored in a vault. However, a folder has a default vault associated with it that specifies the vault used, by default, for files within the folder. Using this principle, people often refer to the vault associated with the top-level folder of a revision-controlled project as "The Vault Folder", or simply "The Vault" for the project.

Branches and vaults have owners associated with them. Ownership is important for controlling access to design objects through operating-system protections and DesignSync access controls. For example, the ability to delete a vault can be access-controlled based on the owner of the vault. The owner of a vault is defined as the owner of the design object's main branch. By default, the owner of a branch is the creator of the initial version of the branch unless a different owner has been specified with the setowner command. See the ENOVIA Synchronicity Command Reference: setowner command for more details.

### Related Topics

Viewing the Contents of a Vault

*DesignSync Data Manager Administrator's Guide: Access Control Overview*

Parallel (Multi-Branch) Development

## Introduction to Data Replication

Members of a project team often need read-only access to the same data. DesignSync users on UNIX all have access to the same data, with that data maintained in a common area by DesignSync. If users each fetch local copies of the same data into their own work areas, your project team will require additional disk space per user and additional time for the data to be transferred from the server. Instead, your team can share common data in a cache or a mirror directory.

When users do need their own local copies of data, DesignSync attempts to copy the requested file version from a file cache or a mirror directory. If the requested file version

does not exist in a file cache or a mirror directory, DesignSync fetches the data from the server.

For module data, UNIX users can link to modules in a module cache in addition to linking to file versions in the DesignSync file cache. When linking to file versions in the file cache, the `-share` option is used with the module operation. Depending on the system setup, either hard links or symbolic links are created to the module member version. When linking to a module cache, only one symbolic link is used to link the workspace module base directory to the base directory of this module in the module cache. Both of these methods provide linking to files across the LAN instead of transferring data from the server to optimize performance. Using UNIX links also save disk space.

Legacy modules can be linked to, or copied from, a module cache.

Note: Because of the way links are handled, mirrors, caches, and module caches are not currently supported on Windows.

### Related topics

Using a Module Cache

*DesignSync Data Manager Administrator's Guide: Mirrors Versus Caches*

*DesignSync Data Manager Administrator's Guide: Fetching Files from the Mirror or Cache*

*DesignSync Data Manager Administrator's Guide: Setting up a Module Cache*

## Metadata Overview

When you place objects under revision control, DesignSync manages the objects both in the vault (typically on a server) and in your local (client) work area. Information about the objects that DesignSync manages is called **metadata**. DesignSync maintains both server-side and local metadata. You view the metadata indirectly using the DesignSync graphical interface or DesignSync shells.

Server-side metadata is accessible by all users who have access to the vault, and includes information such as:

- The branches and versions available for a module or an object
- A branch's lock and retired status
- A branch's tags
- A version's tags and log (check-in/check-out comments)



Local metadata is accessible by you, and it may or may not be accessible to other users based on your preferred work style. Local metadata is designed to optimize the performance and disk utilization of common operations and includes information such as:

- The module instance name.
- The state of an object in your local work area (reference, copy, locked copy, link to cache or mirror)
- The object's vault, branch, and version
- The timestamp when you fetched the object
- The persistent selector list
- The version last merged with the object

Local metadata is stored in the workspace module root directory and in `.SYNC` directories on your local file system.

**CAUTION:** DesignSync manages these metadata directories; do not directly manipulate these directories or their contents.

## Local metadata

DesignSync stores the metadata information in the workspace module directory. You must have write access to the module root directory.

**Note:** If your intent is not to share a work area, you should structure your local hierarchy so that multiple users are not required to write into the same module root directory.

If the workspace is shared, DesignSync clients on UNIX platforms must have their UNIX umask set so that the module root directory is created with write access granted to others in the same UNIX group.

### Notes:

- Problems you might experience reading or writing metadata (reported as errors by DesignSync) are typically due to protection or locking issues.
- If two users attempt to perform revision control operations in the same directory, DesignSync forces the second user to wait until the first is complete -- two users cannot modify the metadata at the same time. The second user can interrupt the operation if the wait is too long. For more information, see the ENOVIA Synchronicity Command Reference Help Interrupt (Control-c) section of the Command Reference.

### Related Topics

Controlling Access to Your Local Work Area

### Setting Up a Shared Workspace

*DesignSync Data Manager Administrator's Guide: Troubleshooting Metadata Errors*

## Mirrors

### Mirroring Overview

A mirror exactly mimics the data set defined for your project vault. Mirrors provide an easy way for multiple users to point to the file versions that comprise their project's data. The file versions in the mirror belong to the configuration defined by the project lead. For example, the configuration could be the Latest version of files on the main Trunk branch. A mirror for a development branch may be defined to always contain the file versions on that branch with a specific tag. When the file versions comprising the configuration change, for example, if Latest versions are being mirrored and a new version of a file is checked into the vault, the mirror directory is automatically updated with the new version. Without mirroring, users would need to frequently update their work areas using the `populate` command to reflect the project's current data set. You can find where vault data is being mirrored, and the status of those mirrors. (See the Related Topics below.)

The `setmirror` command associates a workspace with a mirror directory. A mirror will always have accurate metadata because any action that writes to a mirror directory updates the local metadata in the mirror directory. When you use the `setmirror` command to associate a mirror directory, checking in an object will:

- create the new version in the vault,
- update the file in the associated mirror associated, and then
- update the metadata.

Mirror can be updated and administered automatically. See the section Administering Mirrors for details. As of Version 4.2, the legacy Remote Mirror Assurance package is no longer supported.

#### Mirror Attributes

- All actions that write to a mirror directory will update the local metadata in the mirror directory. When looking at a workspace that has objects in the mirror state, a combination of the workspace's and the mirror directory's local metadata will be used to determine the correct version of the objects. This allows you to use the `ls` or `url` command on objects in the workspace to show the correct state of the object.
- Mirrors support all defined configurations.
- When a check-in occurs from a client, it creates a new version in the vault, returns control back to the client, and the client writes the object into the mirror and updates the local metadata in the mirror directory.

- A mirror write through will occur for all fetch states. Regardless of the fetch state, if a mirror write through is done, then the metadata is updated to reflect what was written to the mirror directory.
- No other commands, with or without the `-mirror` option, write through to the mirror. Commands like `populate -mirror` and `cancel -mirror` do not write to the mirror directory. However, the `co -mirror` command writes through to the mirror directory if the correct up-to-date version is not already in the mirror. Most commands only create links from a workspace to the files in the mirror directory.

As a DesignSync administrator, you can:

- Set up a mirror directory and navigate through this mirror knowing that everything is being kept up-to-date.
- Set up your environment (from that LAN where the check-ins occur) to write through to your mirror when checking a new version into the server. You do not have to wait for the mirror update process to update the mirror.

#### Restrictions

- Because mirroring is implemented with UNIX links, mirrors are not supported on Windows platforms.
- The mirror directory and the users accessing it must be on the same LAN.
- Only one process can write to the mirror subdirectory at a time. The system ensures that when you check in a new version, there will be a lock on the mirror subdirectory. The lock is held for the duration of the client check-in from the workspace subdirectory. The system will display a "waiting on metadata lock" message while the system processes the workspace. This may cause a delay if someone is checking in a large amount of objects or large files.
- Mirrors for modules cannot be linked to from a workspace. A module cache should be used instead.

#### Related Topics

General Mirror Topics:

[Mirrors Versus Caches](#)

[Using a Mirror](#)

[ENOVIA Synchronicity Command Reference: mirror wheremirrored](#)

Mirror Administration Topics:

[Administering Mirrors](#)

[Finding Mirrored Data](#)

### Using a Mirror

A mirror exactly mimics the data set defined for your project vault. Mirrors provide an easy way for multiple users to point to the file versions that comprise their project's data. The file versions in the mirror belong to the configuration defined by the project lead.

### Examples

- The configuration might be the Latest version of files on the main Trunk branch. A mirror for a development branch might be defined to always contain the file versions on that branch with a specific tag. When the file versions comprising the configuration change, the mirror directory automatically updates with the new version.
- If Latest versions are being mirrored and a new version of a file is checked into the vault, the mirror directory updates with this new version. Without mirroring, users need to frequently update their work areas using the **populate** command to reflect the project's current data set.

Mirror directories can be treated in the same way as your DesignSync work areas. For example, you can use commands such as the **url** or **ls** commands on mirror directories.

#### Setting Up Your Workspace

Your team leader will have set up a mirror directory for your project. Use the `setmirror` command to associate your workspace with the project's mirror directory. The `setmirror` command does not have a GUI equivalent. See *ENOVIA Synchronicity Command Reference: setmirror help* for more information on this command.

**Note:** you cannot link to a module mirror from a workspace.

All of the workspace's subdirectories automatically inherit the mirror location set for the top level of the workspace. You cannot set a different mirror on a subdirectory from that of its the parent directory.

To determine if your current work area directory is associated with a mirror, use the `url mirror` command. See the *ENOVIA Synchronicity Command Reference: url mirror help* for more information on this command.

**Note:** To resolve the mirror location, DesignSync does not search above the root of a workspace where a `setvault` has been applied.

So if a setvault has been applied to a folder (/Projects/ASIC/alu) and you apply the setmirror command at a higher-level folder (for example, /Projects/ASIC), the setmirror command is ignored at and below the folder where the setvault occurred (/Projects/ASIC/alu). See *ENOVIA Synchronicity Command Reference: setvault help* for more information on this command.

Normally, the path to a mirror is stored exactly as specified by the setmirror command. If your mirror directory is set to an auto-mounted directory, you can set a registry key for DesignSync to resolve the path instead.

See *DesignSync Data Manager Administrator's Guide: DesignSync Client Commands Registry Settings* for more information.

#### Changing the Mirror Directory Associated with Your Workspace

If the mirror directory for your project changes, run the setmirror command from the same directory in which the original setmirror command was run. This command updates the workspace's mirror association, which is inherited by lower level directories.

To correct existing workspace links to mirror files, run the **populate** command with these options:

```
populate -recursive -mirror -unifystate
```

This command corrects the links to point to the mirror directory's new location.

#### Disassociating Your Workspace from a Mirror Directory

If you no longer need to use a mirror directory, you can disassociate your work area directory from the mirror, by using the setmirror command.

#### Using the -mirror Option to Commands

Once you have associated your workspace with a mirror directory, use the -mirror option with populate, **ci**, **co**, and **cancel** commands (or select Keep a link to Latest (mirror) when performing these operations through the DesignSync GUI). You can also specify that -mirror be used by default, if your team leader did not set that for your project. For details, see Object States.

**Note:** You cannot use the populate -mirror command (or select Keep a link to Latest (mirror)) to populate a directory containing a module. In addition, the ci command ignores the -mirror option if you use it when checking in a module.

Having links to files in the mirror directory ensures that you are always referencing the most up-to-date configuration. However, if other mirror users add files to the mirror, they are not automatically exposed to your work area. Therefore, you should periodically populate your work area directory using the **populate -mirror** command.

### Notes:

- When performing the **populate -mirror** operation, DesignSync creates links only if no file or link already exists in your work area directory; DesignSync does not change the state of existing files and links.

To change the state of existing files and links when you populate your working directory, use the **-force** or **-unifystate** option (or select Overwrite local files if they exist or Unify workspace state in the DesignSync GUI), in addition to the **-mirror** option.

**Caution:** Using the **-force** option overwrites any locally modified files.

- You cannot use the populate **-mirror** command to populate a directory containing a module.

### Related Topics

*DesignSync Data Manager Administrator's Guide: Mirroring Overview*

*DesignSync Data Manager Administrator's Guide: Mirrors Versus LAN Caches*

ENOVIA Synchronicity Command Reference Help: setmirror

ENOVIA Synchronicity Command Reference Help: ci

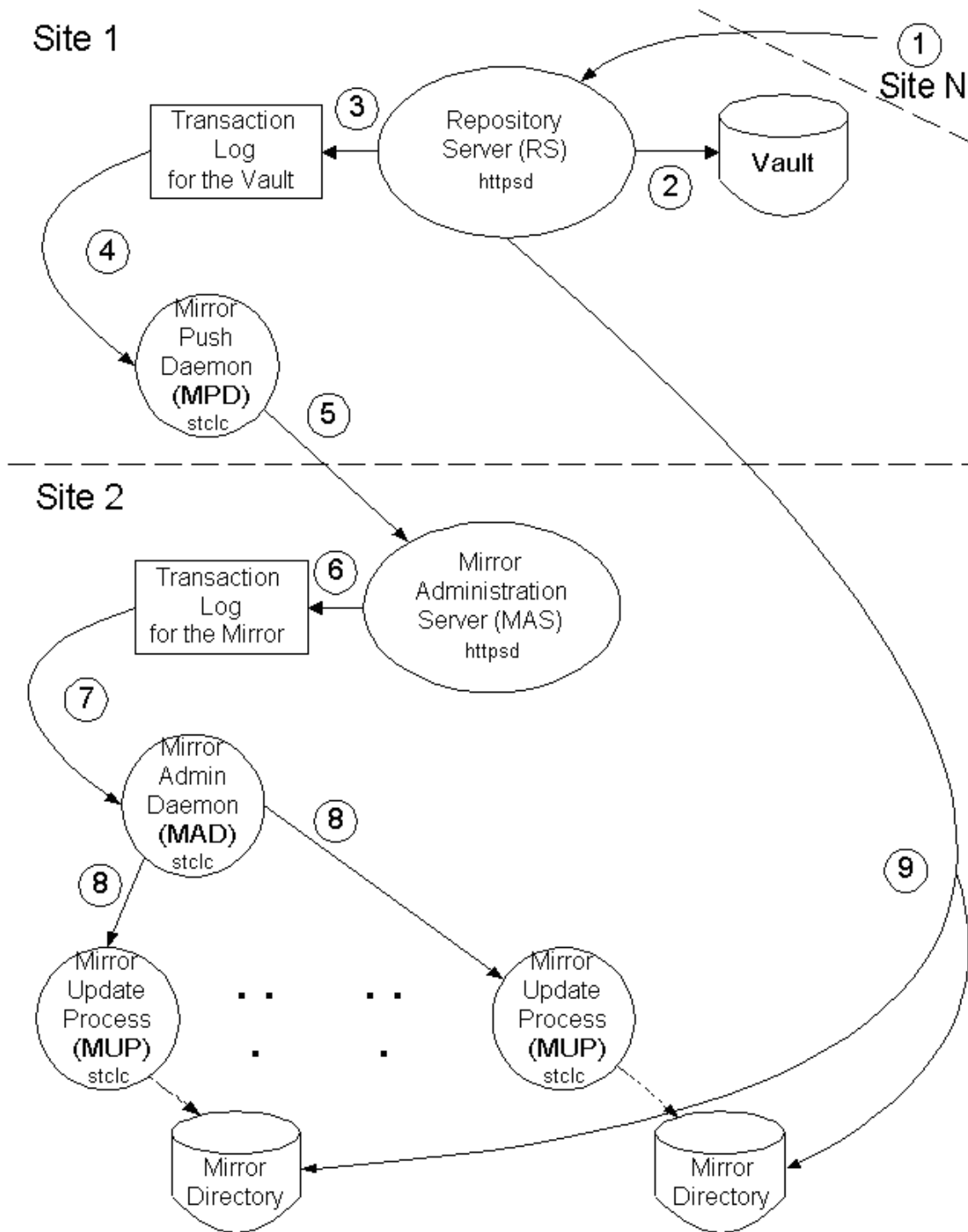
ENOVIA Synchronicity Command Reference Help: populate

ENOVIA Synchronicity Command Reference Help: co

ENOVIA Synchronicity Command Reference Help cancel

### Architecture of the Mirror System

The flow diagram below shows how mirror directories are automatically updated, reacting to a change in the vault being mirrored. Descriptions of the steps follow.



1. A DesignSync client operation, from any site, is issued that will modify the vault data being mirrored. For example, a ci command is run, of a locally modified managed object.
2. A new file version is created in the Repository Server's vault.
3. The Repository Server updates its transaction log with the new version information, recording the change in vault content as vault log entries.

4. The Mirror Push Daemon on the Repository Server reads the transaction log via the Tcl interface, for the "change set".
5. The Mirror Push Daemon pushes the change set via rstcl to each Mirror Administration Server. Only the changes relevant to the mirrors being managed by a particular Mirror Administration Server are pushed to that Mirror Administration Server.
6. The Mirror Administration Server writes the changes to its transaction log as mirror log entries, and then returns control back to the Mirror Push Daemon. From the successful return of the Mirror Administration Server, the Mirror Push Daemon knows that the Mirror Administration Server has received the changes.
7. The Mirror Administration Daemon reads the change set from the mirror's transaction log via the Tcl interface.
8. The Mirror Administration Daemon spawns a Mirror Update Process for each affected mirror that needs to be updated.
9. The mirror directories are updated. When mirroring a non-legacy module, a populate will always be performed, regardless of the DesignSync client command used to modify the module. When mirroring a legacy module or a DesignSync vault, if the originating DesignSync client command was a ci, then each Mirror Update Process performs a co to fetch the new versions that belong in the mirror. If the originating DesignSync client command was some other operation (such as tag, mvfolder, retire, etc.) then each Mirror Update Process performs a populate, to update the mirror. populate is also run by each Mirror Update process if the originating DesignSync client command was a ci that produced new versions of many files. For more information, see the *DesignSync Data Manager Administrator's Guide: Registry Settings for a Mirror Administration Server* for more information.

### Administering Mirrors

DesignSync uses a method for managing mirrors introduced in the 4.1 release of Developer Suite. All DesignSync mirror servers must be minimally running version 4.1 to work properly with this version of DesignSync. If you are setting up mirrors on a pre-4.1 installation, refer to documentation from that release.

You can set up the following types of DesignSync mirrors:

- Standard mirror – Fetches design objects directly from the repository server.
- Primary mirror – Fetches design objects directly from the repository server and serves them to secondary mirrors. A primary mirror must be on a UNIX host that supports hard links. This is not supported for non-legacy modules.
- Secondary mirror – Fetches design objects from a primary mirror instead of directly from the repository server. This is not supported for non-legacy modules.

Secondary mirrors help reduce network traffic at the repository server. A secondary mirror communicates with the Repository Server to determine what



objects need to be updated in the mirror. The secondary mirror fetches the updated contents from the primary mirror instead of from the Repository Server.

Mirrors are managed by a SyncServer. You can use ProjectSync to create and administer two types of DesignSync mirror servers:

- Mirror Administration Server (MAS) – - The SyncServer at a mirror site that manages the mirrors. When objects change in a repository associated with an MAS, the MAS is notified and then updates its affected mirrors. You can create mirrors only on servers with an MAS.
- Repository Server (RS) – - The SyncServer that manages a repository (vault) that is mirrored at one or more mirror sites.

You use the ProjectSync GUI to set up Repository Servers and Mirror Administration Servers and to create and edit mirrors. See ProjectSync Help: Mirror Overview for details on how to set up your mirror servers and your mirrors.

You also can use the mirror commands to work with mirrors. See the mirror command descriptions in the ENOVIA Synchronicity Command Reference for details.

**Note:** As of Version 4.2, Remote Mirror Assurance package and local mirrors are no longer supported.

#### Related Topics

Mirroring Overview

Using a Mirror

Using a Module Cache

*DesignSync Data Manager Administrator's Guide: Administering Mirrors*

*DesignSync Data Manager Administrator's Guide: Mirrors Versus Caches*

*DesignSync Data Manager Administrator's Guide: Fetching Files from the Mirror or Cache*

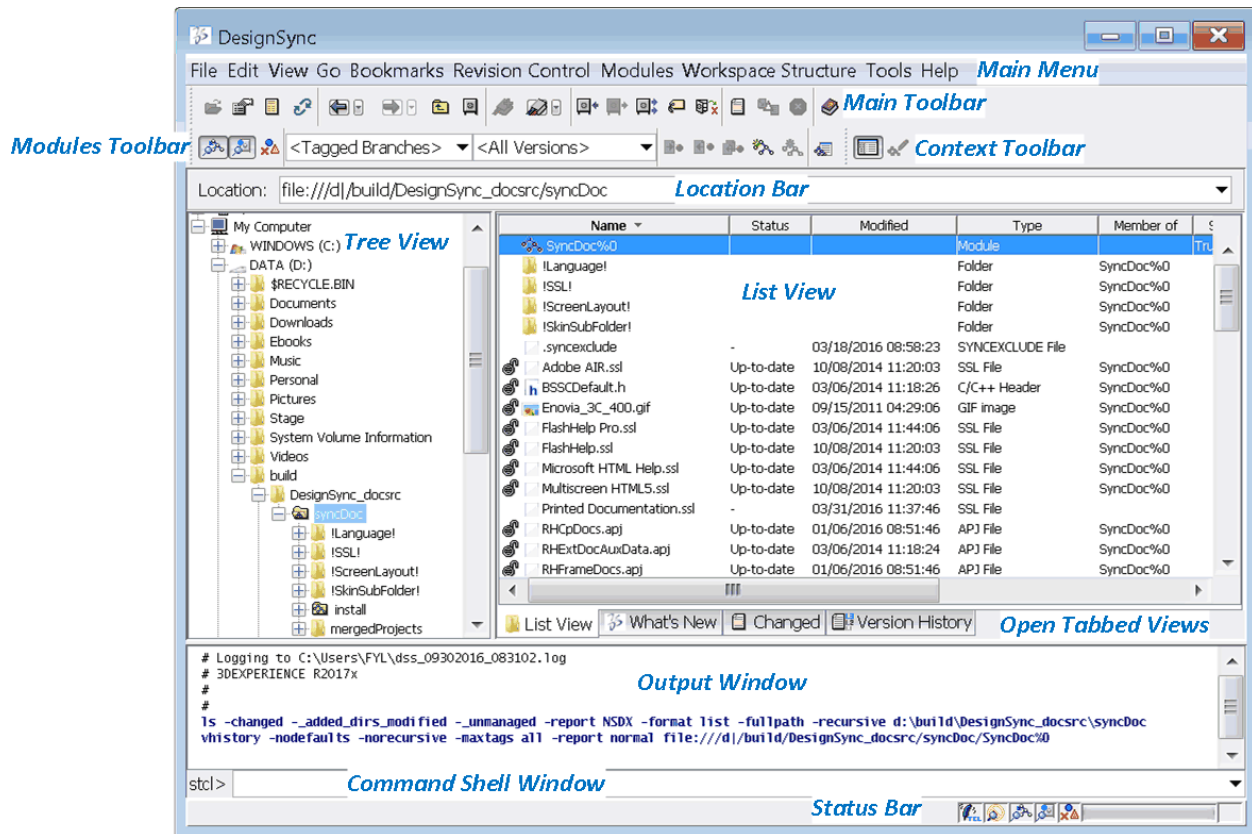
*DesignSync Data Manager Administrator's Guide: Setting up a Module Cache*

*DesignSync Data Manager Administrator's Guide: Setting Up a Mirror Server*

## Understanding the GUI Interface

### Using the Classic DesignSync GUI

There are many elements to the DesignSync graphical user interface (GUI). Click on the label for each region in the following illustration to go to a topic that describes that part of the DesignSync GUI.



In addition to the classic DesignSync GUI, DesignSync provides an integrated browser for examining the workspace structure of modules, and several command-line interfaces. See DesignSync Command-Line Shells for more information.

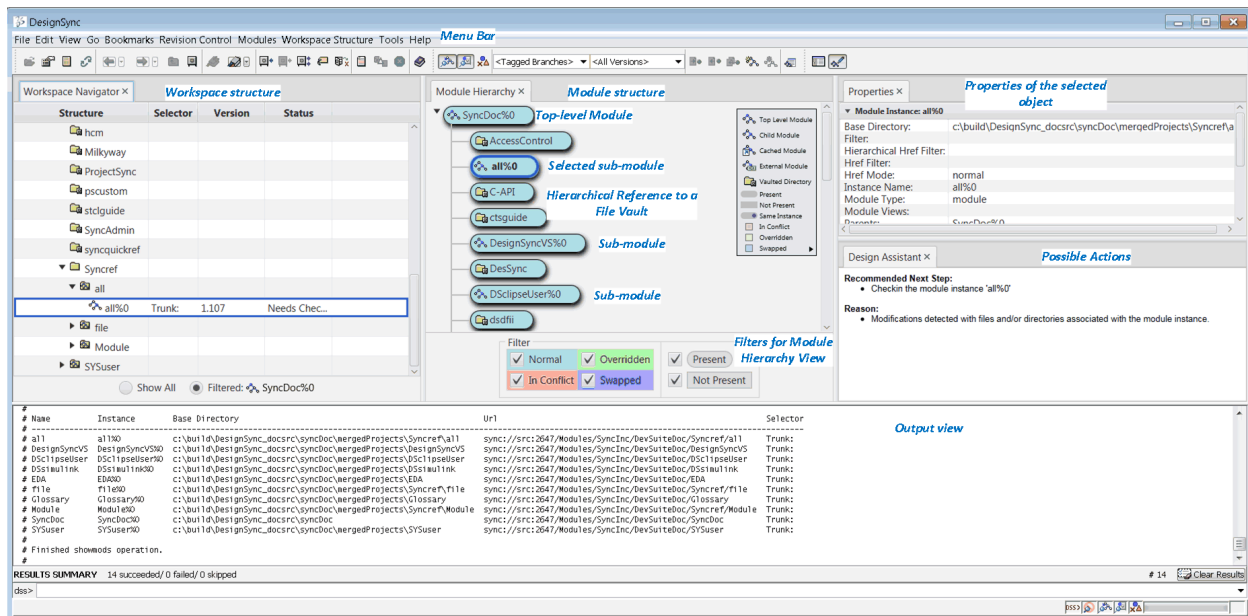
## Using the Workspace Structure Browser

There are many elements within a module that can be hard to visualize. DesignSync provides the Workspace Structure Browser graphical user interface for maintaining and understanding your module hierarchy. The Workspace Structure Browser allows you to:

- Clearly display a module's hierarchy and relate that hierarchy to how it appears in the directory structure.
- Distinctly depict swapped, overriding, and conflicted module instances and how they relate to the other modules in the hierarchy.
- Provide a clear indication of each module instance's status and how that status propagates to its parents.
- Assist the user with guidance of what to do next.

The workspace browser interface is composed of tabbed views that combine to give you a fuller picture of your module workspace and hierarchy. You can have as many or as few of these provided views as desired.




This image shows the default positioning of the views in the Workspace Structure Browser.







## DesignSync Symbols and Icons

### Informational Symbols



These symbols provide information about DesignSync activities.














	"In process" symbols indicate that the selected folder is being processed. (The hourglass cursor also indicates processing, but these symbols tell you specifically which objects are being processed.)
	Failed attempt to connect to a server.
	Informational message.

	File or folder is excluded from the operation you have selected.
	The operation is not allowed on the object you selected. This might appear, for example, if access controls have been defined.
	The server referenced by a project or SyncServers file is accessible.
	A collection object is not properly defined. DesignSync cannot determine the collection's members.

### Object Icons










These icons identify objects in a DesignSync workspace. They may be in either the Tree View Pane or the List View Pane.

 The Internet	The root of all objects in the DesignSync universe. Available under this top-level icon are: <b>SyncServers</b> , <b>My Computer</b> , <b>Projects</b> , and <b>Bookmarks</b> .
 My Computer	The root of the local file system. On a Windows system, expansion of the icon shows icons representing hard drives, removable drives, and so on, as well as the hierarchies under them. (Note: Sort order is case sensitive. Capital letters are sorted before lower case letters.)  On a UNIX system, expansion shows the mount points that you have chosen to display. The default mount point is the location represented by the \$HOME environment variable.

 Projects	<p>A list of DesignSync projects that have been defined. Currently, only Public Projects is available under Projects. Public Projects lists the projects that have been defined by your site administrator or project leader. Click on a project, then use the right-mouse button to display the popup menu:</p> <ul style="list-style-type: none"> <li>• <b>Properties</b> displays the properties (name, vault and cache location, and description) of the project.</li> </ul>
 SyncServers	<p>A list of available SyncServers and vaults that have been defined for you. Expansion of the icon shows three folders: <b>My Servers</b> (personally defined servers/vaults), <b>Site Servers</b> (site-defined servers/vaults), and <b>Enterprise Servers</b> (enterprise-defined servers/vaults). See the SyncServer List Files topic for more information.</p>
	<p>Closed folder.</p>
	<p>Open folder</p>
	<p>Closed module base directory (Folder Explorer)</p>
	<p>Open module base directory (Folder Explorer)</p>
	<p>Closed Module Roots container (Module Explorer)</p>
	<p>Open Module Roots container (Module Explorer)</p>
	<p>Closed module root (Module Explorer)</p>
	<p>Open Module root (Module Explorer)</p>
	<p>Module instance (List view, displayed when you click a module base directory).</p>
	<p>Default workspace object icon. If the system running the client has defined icons for different file types, you will see those icons instead of this one.</p>
	<p>Hierarchical reference.</p>







## Revision Control Object Icons

These icons identify objects in a DesignSync server vault.

	Vault, not open.
	Vault, open.
	A version of a file in a vault.
	DesignSync vault branch.
	Branch-point version.
	Module branch.
	Module version.
	Module member.
	Hierarchical reference.

## Lock Symbols




These symbols provide information about the lock status of DesignSync objects in the workspace. These symbols only appear in the List View Pane.

	Closed lock, meaning that the branch for that version is locked.
	Open lock, meaning that the branch for that version is unlocked.
	Modified closed lock, meaning that you have modified the object since you checked it out with a lock.
	Modified open lock, meaning you have modified an object that you have not checked out with a lock.
	Locked by user, meaning that the object has been locked by someone other than you.
	Modified locked by user, meaning the object has been locked by someone other than you, and you have locally modified your copy of the object.

**Note:** Module members moved in the workspace are always considered modified until the next module version is created on checkin. For more information, see **Moving a module member**.

## State Overlay Symbols

These symbols provide additional information about an object and appear with the object icon.

	A <b>reference</b> is an object that is not physically present, but instead points to another object. Although there are no physical files corresponding to a reference, there is DesignSync metadata that keeps track of the revision-control information for the reference. Just as with replicas, the object that the reference points to is called its original.
	A <b>replica</b> is a copy of another object. The replication process is different from the normal copying process. A "copy," in the usual sense, is an independent object; it does not "remember" the object from which it was copied. A replica, on the other hand, does remember the object from which it was copied.
	In some situations you want to be able to easily access a file without having a copy of it in your work area, especially if the file is very large. This is where a link is useful. A <b>link</b> , or symbolic link, is visible in your local work area (for example, by the <b>ls</b> command), but the file it represents is not actually there. Links are available only on UNIX systems.  DesignSync uses links to implement two sharing methodologies: caches and mirrors.

## Toolbars and Menus

### Using Toolbars

A toolbar is a set of tools represented by icon buttons that are grouped together into an area on the main window of an application. In the DesignSync GUI, there are two main toolbars:

- The **Main Toolbar** is a group of the most frequently used operations. You can customize the toolbar by selecting **Tools =>Options =>GUI Customization =>Main Toolbar**.
- The **Module Toolbar** is a group of the frequently used module operations. You can customize the toolbar by selecting **Tools =>Options =>GUI Customization =>Module Toolbar**.

A brief explanation of each button in a toolbar pops up when you move the mouse pointer over that button (commonly known as a **tool tip**). In addition to the tool tip, a

slightly longer description appears in the status region at the bottom of the DesignSync window.

A toolbar can be either docked or floating. A toolbar is **docked** when it is attached to one side of the DesignSync window. You can dock a toolbar below the Location Bar (or below the Menu Bar if the Location bar is not displayed) or to the left, right, or bottom edge of the DesignSync window. When you drag a toolbar to the edge of the DesignSync window, the toolbar outline snaps into place along the length of the window edge.



A toolbar is **floating** when it is an independent window that is not attached to the DesignSync window. To change the shape of a floating toolbar, move the cursor over any edge until it changes to a double-headed arrow, and then drag the edge of the toolbar.

To move a toolbar, click and hold the left mouse button within the toolbar but not on an icon, then drag the toolbar. If the toolbar is floating, you can also grab the title bar.

**Note:** On UNIX, you cannot dock a floating toolbar by dragging the title bar. You must position the cursor within the window but not over an icon.

### Main Menu Toolbar

The **Main Menu Toolbar** provides access to DesignSync commands. Clicking on an item in the Main Menu produces a drop-down menu containing menu choices. Menu choices either execute immediately, lead you to further choices, or invoke dialog boxes. For example, if you:

- Select a menu choice with ellipses (...), a dialog box displays. For example, if you choose **Revision Control =>  Check In . . .**, the Check In dialog box appears.
- Select a menu choice without ellipses or right arrows, such as ** Go Up One Level**, the command executes immediately.
- Select an item with a right arrow, you are lead to further choices, such **Tools => Reports** leads you to a list of the available reporting tools.

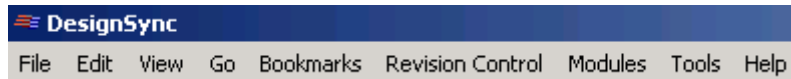
The standard Main Menu Toolbar contains these choices:

- File
- Edit
- View
- Go
- Revision Control
- Modules
- Tools
- Help



- Bookmarks

Click on the menus in the following illustration to go to the topic describing that menu:



## Module Toolbar





By default the Module Toolbar appears under the Main toolbar. You can customize the Module Toolbar to appear:











- On the side of the main toolbar, when your display window is wide enough.
- On either the left side, right side, or bottom of the DesignSync GUI window.
- As a free floating tool bar that you can move to where you want it.

You can choose to display or hide this too bar from the View menu. The Module Toolbar displays these items by default:



See SyncAdmin Help: Customizing the Module Toolbar for more details.

Toolbar Item	Description
	This toggle button shows or hides hierarchical references for workspace listing of modules or module version contents on the server. When hierarchical references are hidden, the hierarchical references button looks like this: 
	This toggle button shows or hides module members on the server. When modules members are hidden, the modules members button looks like this: 

	<p>This toggle button that shows or hides module deltas on the server. When module deltas are hidden, the modules delta button looks like this:</p> 
	<p>This pull-down list allows you to filter module branches displayed. Choices are:</p> <ul style="list-style-type: none"> <li>○ &lt;All Branches&gt;</li> <li>○ &lt;Tagged Branches&gt;</li> <li>○ One of the last five 5 glob expressions as defined in the module branch filter field in the Display Filters dialog box</li> </ul>
	<p>This pull-down list allows you to filter versions displayed. Choices are:</p> <ul style="list-style-type: none"> <li>○ &lt;Tagged Versions&gt;</li> <li>○ &lt;All Versions&gt;</li> <li>○ One of the last five 5 glob expressions as defined in the module version filter field in the Display Filters dialog box</li> </ul>
	<p>This button invokes the Add to Module dialog box.</p>
	<p>This button invokes the Remove from module dialog box.</p>
	<p>This button invokes the Move modules member dialog box.</p>
	<p>This button invokes the Create a hierarchical reference dialog box.</p>
	<p>This button invokes the Create new module dialog box.</p>
	<p>The button adds the Module Hierarchy tab to the display window.</p>

Related Topics

Renaming a Module Member

Creating a Hierarchical Reference

Displaying Module Hierarchy

Creating a New Module

Removing a Member from a Module

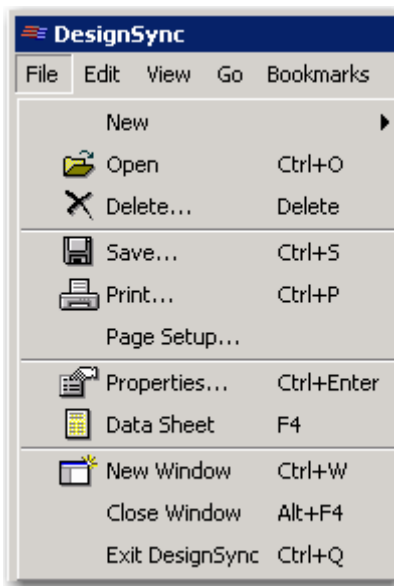
Display Filters

## Context Toolbar

Type topic text here.

## File Menu

The File menu contains the commands that are most relevant to file and folder management. Some of the actions available from the File menu are also available by selecting a file or a folder and right-clicking to display the context menu.



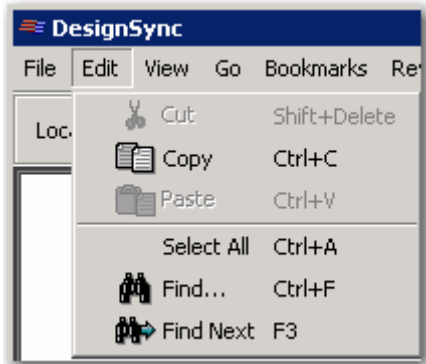
The following actions can be selected from the File menu:

<b>Action</b>	<b>Result</b>
<b>New=&gt;File</b>	Invokes the New File dialog box.
<b>New=&gt;Folder</b>	Invokes the New Folder dialog box.
<b>Open</b>	Opens a session with your default editor and opens the selected file, or opens the selected project so you can view or edit its properties. Your default editor is determined by a registry setting in one of the DesignSync client registry files. You can override the

	<p>installation- or site-wide default editor using the SyncAdmin tool.</p> <p>You can select and open as many files as you choose, but you will be prompted for confirmation if you selected more than four files to open simultaneously.</p> <p><b>Note:</b> If you are using the DesignSync Windows client and you use any application other than the default ASCII editor, DesignSync will not automatically recognize local modifications.</p>
<b>Delete</b>	<p>Deletes any of the following selected objects:</p> <ul style="list-style-type: none"> <li>• A file from your working directory. You cannot delete a file that is a member of a collection object.</li> <li>• An empty folder, either local or on the SyncServer.</li> <li>• A version from a vault.</li> </ul>
<b>Save</b>	<p>Saves information displayed in the View Pane such as a data sheet. You can choose the file name and directory from the dialog box. Collection objects and DesignSync references are not shown. The Save menu item is active for DesSync's Output Region, Text views and HTML views.</p>
<b>Print</b>	<p>Allows you to print information displayed in the View Pane such as a data sheet. You can also highlight text in a View Pane and send it to a printer. If you are using the Windows platform, you can also print a file that you have selected in the List View.</p>
<b>Page Setup</b>	<p>Controls the print parameters of your default printer.</p>
<b>Properties</b>	<p>Invokes the Properties dialog box where you can view or edit the various properties of the selected object. The Properties dialog box will only display when you select one file, folder or module instance in the list or tree view.</p>
<b>Data Sheet</b>	<p>Displays the data sheet for the selected object in a new tabbed view in the View Pane.</p>
<b>New Window</b>	<p>Creates a new instance of DesignSync; the same way you would create a new window in a browser.</p>
<b>Close Window</b>	<p>Closes the current DesignSync window and exits DesignSync if this is the only DesignSync window.</p>
<b>Exit DesignSync</b>	<p>Closes all DesignSync windows and exits DesignSync. If there is more than one DesignSync window currently open, a confirmation box appears and you must confirm the operation. You can also exit DesignSync using the <b>exit</b> command from the command bar. See ENOVIA Synchronicity Command Reference Help: Exit for more information.</p>

## Edit Menu

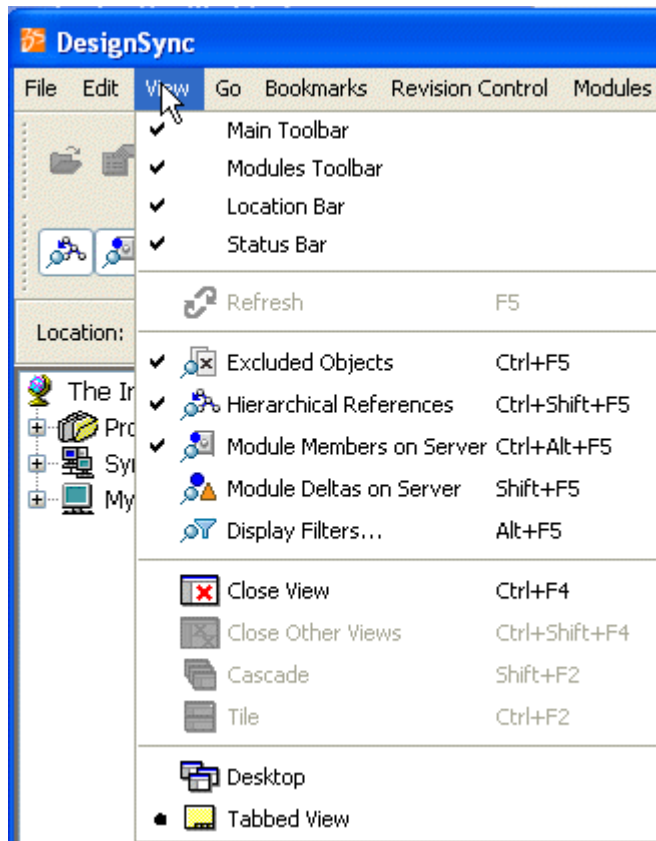
The Edit menu choices allow you to cut, copy, paste, select, and search the text that displays in text output windows such as Data Sheet output, or the output window at the bottom of the screen.



All of the options (except for **Find**) also work on the text in the location bar at the top of the screen.

### View Menu

The View menu contains the commands that control what objects are displayed in the List View. Some of these commands can also be added as Toolbar options to the Main or Module Toolbar.



The following actions or options can be selected from the View Menu:

<b>Action/Option</b>	<b>Result</b>
<b>Main Toolbar</b>	When checked, the Main Toolbar displays. When not checked, the Main Toolbar is hidden.
<b>Module Toolbar</b>	When checked, the Module Toolbar displays. When not checked, the Modules Toolbar is hidden.
<b>Location Bar</b>	When checked, the Location Toolbar displays. When not checked, the Modules Toolbar is hidden.
<b>Status Bar</b>	When checked, the Status Toolbar displays. When not checked, the Status Toolbar is hidden.

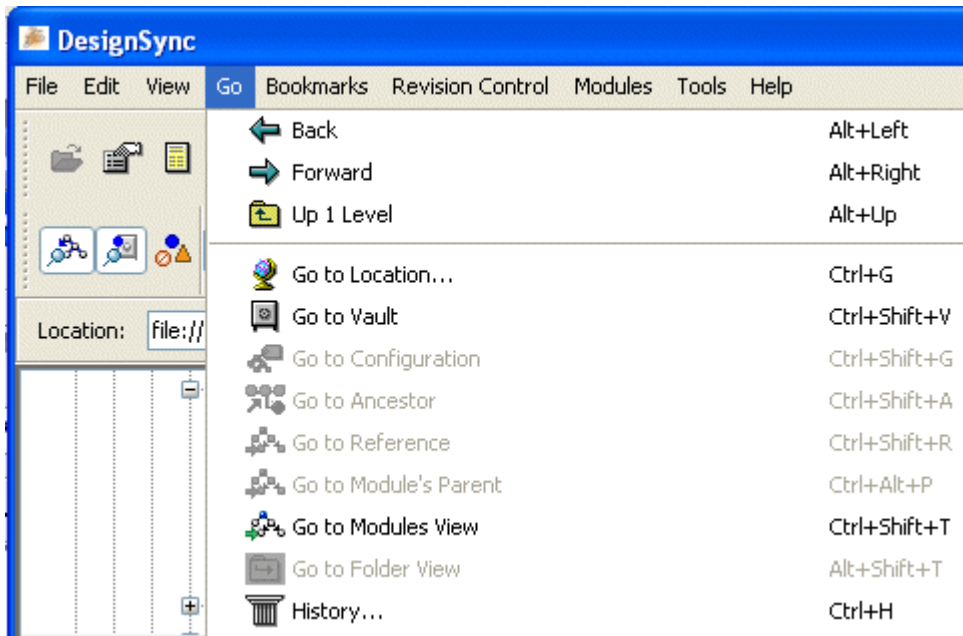
<b>Refresh</b>	<p>Verifies the selected object's current properties and updates the DesignSync display.</p> <p>In the case of container objects (folders, vaults, branches), the child objects are also refreshed.</p> <p>Some DesignSync operations do not update the display automatically because of performance constraints. To ensure that your display is up-to-date, you should refresh an object before operating on it, particularly if the object is shared with other users as they may have changed the state of an object.</p> <p>You can control how often the List View is refreshed by selecting <b>Tools =&gt;Options =&gt;GUI Options</b>. Enter the amount of minutes in the <b>Automatically Refresh after so many minutes</b> field. See the SyncAdmin Help: Options topic for more information.</p>
<b>Excluded Objects</b>	When checked, the Exclude Objects displays in the View Pane. When not checked, the Excluded Objects are hidden.
<b>Hierarchical References</b>	When checked, hierarchical references for workspace listing of modules or module version contents on the server are displayed. When not checked, hierarchical references are hidden.
<b>Module Members on Server</b>	When checked, module members on the server are displayed. When unchecked, the modules members on the server are hidden.
<b>Module Deltas on Server</b>	When checked, module deltas are displayed. When unchecked, the module deltas are hidden.
<b>Display Filters</b>	Invokes the Display Filters dialog box.
<b>Close View</b>	Closes the current view displayed in the View Pane.
<b>Close Other Views</b>	Closes all views except the one you are currently viewing and the List View.
<b>Cascade</b>	Displays the Desktop View windows as cascaded in the View Pane.
<b>Tile</b>	Displays the Desktop View windows as tiled in the View Pane.
<b>Desktop</b>	Sets the View Pane to multiple windows that can be moved around a desktop
<b>Tabbed View</b>	Sets the View Pane to a series of tabbed view with only one view visible at any time.

## Related Topic

SyncAdmin Help: Customizing the View Pane

## Go Menu

The Go menu helps you to quickly navigate to a folder or a location. At the bottom of the Go menu is the **recently visited list**. This list shows up to the last ten (10) locations that have been visited. Select any one of these locations and DesignSync goes to that location.



The following actions or options can be selected from the Go Menu:

<b>Action/Option</b>	<b>Result</b>
<b>Back</b>	Moves you to the previously selected location in the tree view. You can click <b>Back</b> again to move to the prior location. Up to seven locations are remembered.
<b>Forward</b>	Reverses the effect of the last <b>Back</b> operation. You can use <b>Back</b> and <b>Forward</b> to quickly move among several locations.
<b>Up 1 Level</b>	Moves you to the parent folder of the current object. In many cases, this moves you up one folder (directory) level from your current location. Also, the Up One Level button lets you move up the vault hierarchy.
<b>Go to</b>	Invokes the Go to Location dialog box allowing you to enter a path

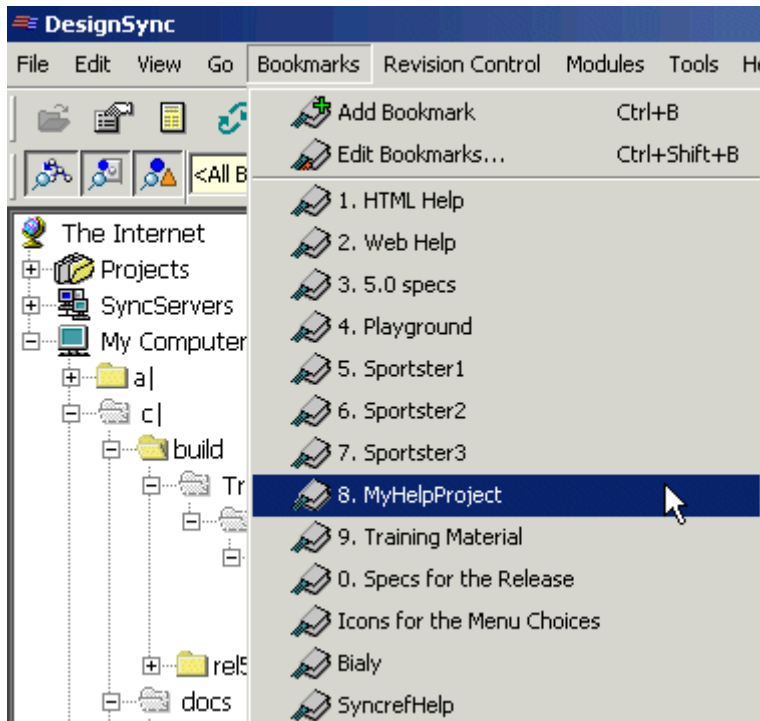


<b>Location</b>	<p>to the file or folder. At this dialog box, you can enter the folder's path or URL in the location field or click <b>Browse Local</b> to navigate to a file or folder on your local machine.</p> <p>By clicking the pull-down arrow to the right of the text field, you can view and select previously visited locations from the location history.</p>
<b>Go to Vault</b>	<p>If the selected object is a module, module member, or hierarchical reference in a module; under revision control; and a vault for it exists, this command displays the module version containing the selected object.</p> <p>If the selected object is a folder (directory), and a vault for it exists, this command displays the contents of the vault where the selected folder is stored.</p> <p>If the selected object is any other type of object, the command displays the versions of the object and the version tags associated with the object.</p> <p><b>Note:</b> This command uses the selector associated with the workspace to display the appropriate branch.</p>
<b>Go to Configuration</b>	<p>This option is supported for legacy modules only.</p> <p>Brings you to the configuration definition on the server for the highlighted object in the workspace.</p>
<b>Go to Ancestor</b>	<p>When a server module branch node is highlighted, invoking the Go to Ancestor command brings you to the ancestor branch point version for the branch selected. You can then browse back through the ancestry of any given version.</p> <p><b>Notes:</b></p> <ul style="list-style-type: none"> <li>• If the desired ancestor is not visible due to the current display filter mode, you will see a dialog box telling you this information.</li> <li>• This action will not be available when branch 1 is selected.</li> <li>• To see descendant information, you will have to get the version history for the given module version.</li> </ul>
<b>Go to Reference</b>	<p>When an hierarchical reference is selected on the client or server, invoking the Go to Reference command brings you to the object referenced by the hierarchical reference.</p> <p>On the client, the referenced object must already be present in the</p>

	<p>workspace for the command to complete.</p> <p>In the Modules view, selecting an hierarchical reference and invoking the Go to Reference command brings you to the object referenced in the Module Explorer.</p>
<b>Go to Module's Parent</b>	<p>If there is a hierarchical reference pointing from one module to another, selecting the target module and invoking the Go to Module's Parent command brings you to the parent module (the module with the hierarchical reference member).</p> <p>If the selected module has than one parent, a dialog box with the list of applicable target parent objects is displayed.</p>
<b>Go to Module Explorer</b>	<p>When working in the Folder view, selecting a module, or a folder that is a member of a module, and invoking the Go to Module Explorer command jumps to the object's representation in the Module Explorer which shows a module-centric perspective of the object.</p> <p>If the selected item is a member of more than one module, a dialog box with the list of applicable modules is displayed. For more information on the Modules Explorer, see Exploring Modules.</p>
<b>Go to Folder Explorer</b>	<p>When working in the Modules Explorer, selecting a module, or a folder that is a member of a module, and invoking the Go to Folder Explorer command jumps to the object's representation in the Folder Explorer which shows a folder-centric perspective of the object.</p>
<b>History</b>	<p>Displays the history dialog box that has a table of all locations visited over the last 30 days as well as the date and time at which each location was last visited. See Reviewing History for more information on this command.</p>

## Bookmarks Menu

The Bookmarks menu lets you add, edit or select bookmarks. Creating a bookmark lets you quickly access modules, directories (folders), vaults, or objects that you access frequently. A list of created bookmarks appears under the Bookmarks commands.



The following actions can be selected from the Bookmarks Menu:

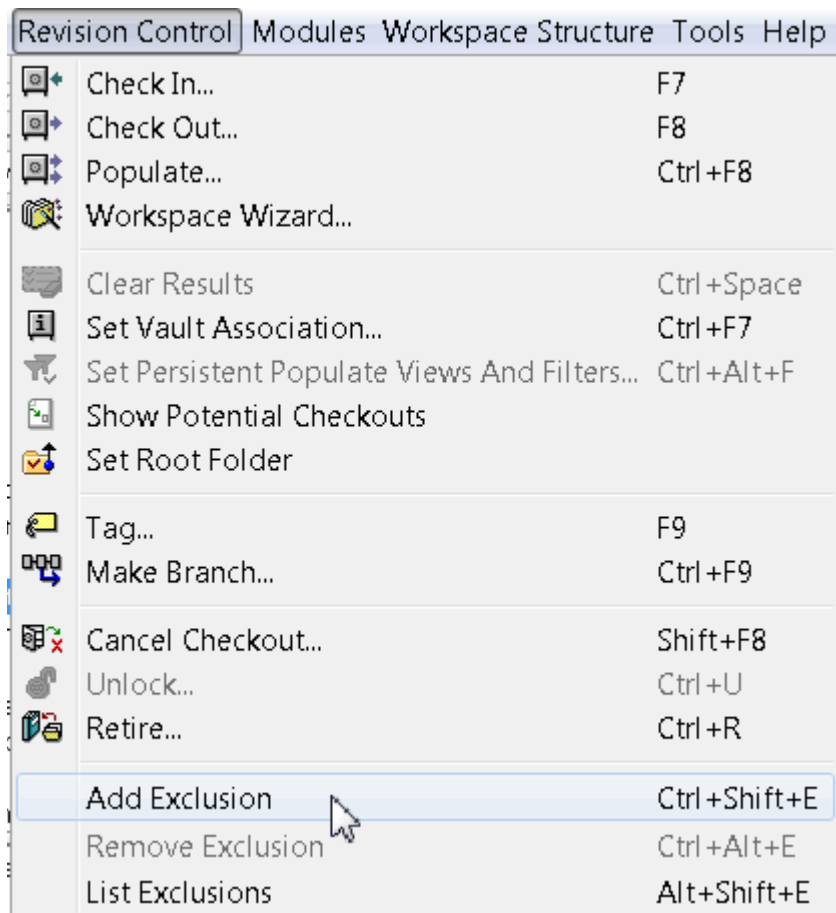
<b>Action</b>	<b>Result</b>
<b>Add Bookmark</b>	Adds the highlighted object to the bottom the bookmarks menu. See Adding, Editing, and Organizing Bookmarks for more information.
<b>Edit</b>	Invokes the Edit Bookmarks dialog box. See Adding, Editing, and Organizing Bookmarks for more information.

#### Related Topic

Defining and Modifying Bookmark Properties

### Revision Control Menu

The Revision Control menu contains the commands that are most relevant to revision control operations. Some of the actions available from the Revision Control menu are also available by selecting a file or a folder and right-clicking to display the context menu.



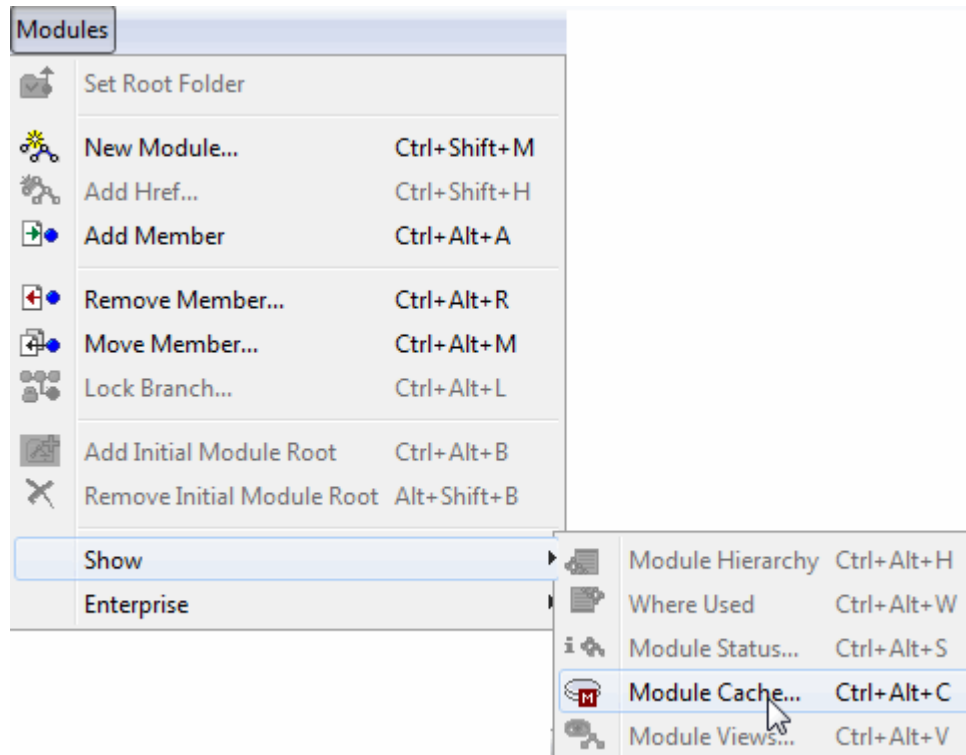
Once you have highlighted a file or folder, the following actions or options can be selected from the Revision control menu:

<b>Action/Option</b>	<b>Result</b>
<b>Check In</b>	Invokes the Check in Dialog box. See Checking In Design Files for more information.
<b>Check Out</b>	Invokes the Check Out Dialog box. See Checking Out Design Files for more information.
<b>Populate</b>	Invokes the Populate Dialog box. See Populating Your Work Area for more information.
<b>Workspace Wizard</b>	Invokes the Workspace Wizard. See Invoking the Workspace Wizard for more information.
<b>Clear Results</b>	Clears the results of the previous action. As part of a Clear Result command, the Result Column and the Summary Bar disappear. See Clearing Results for more information.
<b>Set Vault Association</b>	Invokes the Set Vault Association Dialog box. See Specifying the Vault Location for a Design Hierarchy for more information.

<b>Set Persistent Populate filters</b>	Invokes the Set Persistent populate filters Dialog box. See Setting Persistent Populate Filters for more information.
<b>Show Potential Checkouts</b>	The List View is updated to include all of the objects that exist in the vault but not in the work area. In the Type column, files and collections are shown as Potential Checkout. Folders are shown as Potential Checkout Folder. See Showing Potential Checkouts for more information
<b>Set Root Folder</b>	Sets the specified directory as a workspace root directory. See Setting a Workspace Root for more information.
<b>Tag</b>	Invokes the Tag dialog box. See Tagging Versions and Branches for more information.
<b>Make Branch</b>	Invokes the Make Branch dialog box. See Creating Branches for more information.
<b>Cancel Checkouts</b>	Invokes the Cancel Checkout dialog box. See Canceling a Checkout for more information.
<b>Unlock</b>	Invokes the Unlock dialog box. See Unlocking files for more information.
<b>Retire</b>	Invokes the Retire dialog box. See Retiring Design Data for more information.
<b>Add Exclusion</b>	Adds the selected element(s) an exclusion to all the .syncexclude files in the parent folder of the selected object. If there is not .syncexclude file in the parent folder, DesignSync creates it automatically and adds the exclusion.
<b>Remove Exclusion</b>	Removes the selected element(s) from all the .syncexclude files in the parent folder of the selected object and adds a +exclusion to prevent the element from being excluded by a higher-level .syncexclude file.
<b>List Exclusions</b>	Lists the exclusions from the selected folder and above, thus showing the full hierarchy of exclusions that will be applied to objects in the selected folder in the order that they will be applied. See Viewing Exclusions for more information.

## Modules Menu

The Module menu contains the commands that are most relevant to module operations. Some of the actions available from the modules menu are also available by selecting a file or a folder and right-clicking to display the context menu. Also many of these menu items are available on the Module Toolbar.



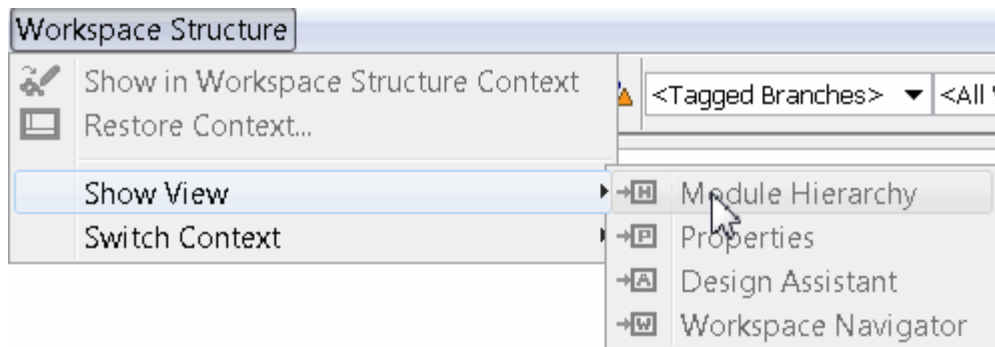
Once you have highlighted a file or folder, the following actions or options can be selected from the Modules menu:

<b>Action/Option</b>	<b>Result</b>
<b>Set Root Folder</b>	Sets the root folder. See Setting a Module root for more information.
<b>New Module</b>	Invokes the Create a new module dialog box. See Creating a Module for more information.
<b>Add Href</b>	Invokes the Create a hierarchical reference dialog box. See Creating a Hierarchical Reference for more information.
<b>Add Member</b>	Invokes the Add to Member dialog box. See Adding a Member to a Module for more information.
<b>Remove Member</b>	Invokes the Remove from module dialog box. See Removing a Member from a Module for more information.
<b>Move Member</b>	Invokes the Move module members dialog box which allows you to move or rename module members. See Moving a module member or Renaming a Module Member for more information.
<b>Lock Branch</b>	Invokes the Lock module branch dialog box. See Locking Module Data for more information.

<b>Add Initial Module Root</b>	If the name of the module root is not in the saved list of module roots, this adds it there.
<b>Remove Initial Module Root</b>	If the name of the module root is in the saved list of module roots, this deletes it.
<b>Show</b>	<p>Display a menu with these choices:</p> <ul style="list-style-type: none"> <li>• Module Hierarchy</li> <li>• Module Where Used</li> <li>• Module Status</li> <li>• Module Cache</li> <li>• Module Views</li> </ul> <p>See these topics for more information.</p>
<b>Enterprise</b>	<p>Displays a menu with these choices:</p> <p>Show Object</p> <p>Synchronize</p>

## Workspace Structure Menu

The Workspace Structure menu contains the commands that launch and assist with working in the Workspace Structure browser.



Once you have highlighted a workspace module, the following actions or options can be selected from the Workspace Structure menu:

Action/Option	Result
Show in Workspace Structure Context	Launches the workspace structure context with the focus on the selected module.

Restore Context	Resets the workspace structure context to the initial view that the context was launched in.
Show View	The workspace structure context features four views that provide additional information about the module and how best to work with the module. Module Hierarchy Properties Design Assistant Workspace Navigator
Switch Context	Switches your DesignSync client view between: Classic - Using the Classic DesignSync GUI. Workspace Structure - Using the Workspace Structure Browser.

## Context Menu

The Context menu pops up when you right-click on an object in the DesignSync View Pane. You can use the selections on this menu to perform most of the common revision-control operations.

Clicking **Edit** opens the selected file using your default editor or to open the selected project so you can view or edit its properties. If you make changes using the default ASCII editor, local modifications are recognized automatically.

**Note:** If you are using the DesignSync Windows client and you use any application other than the default ASCII editor, DesignSync will not automatically recognize local modifications.

**View as Text** displays the contents of the file as text using your default editor. You can change the default ASCII editor using the SyncAdmin tool.

## Location Bar

The **Location Bar** displays the path or URL of the object that you are viewing in the DesignSync window. To go directly to a folder (client-side or server-side), enter the folder's path or URL in the Location Bar. You can also view and select previously visited locations from the location history by clicking the pull-down arrow to the right of the text field.



DesignSync updates the Location Bar as you navigate using the Tree View or List View. However, these locations are not stored in the location history unless you press **Enter** while a location is displayed in the Location Bar.

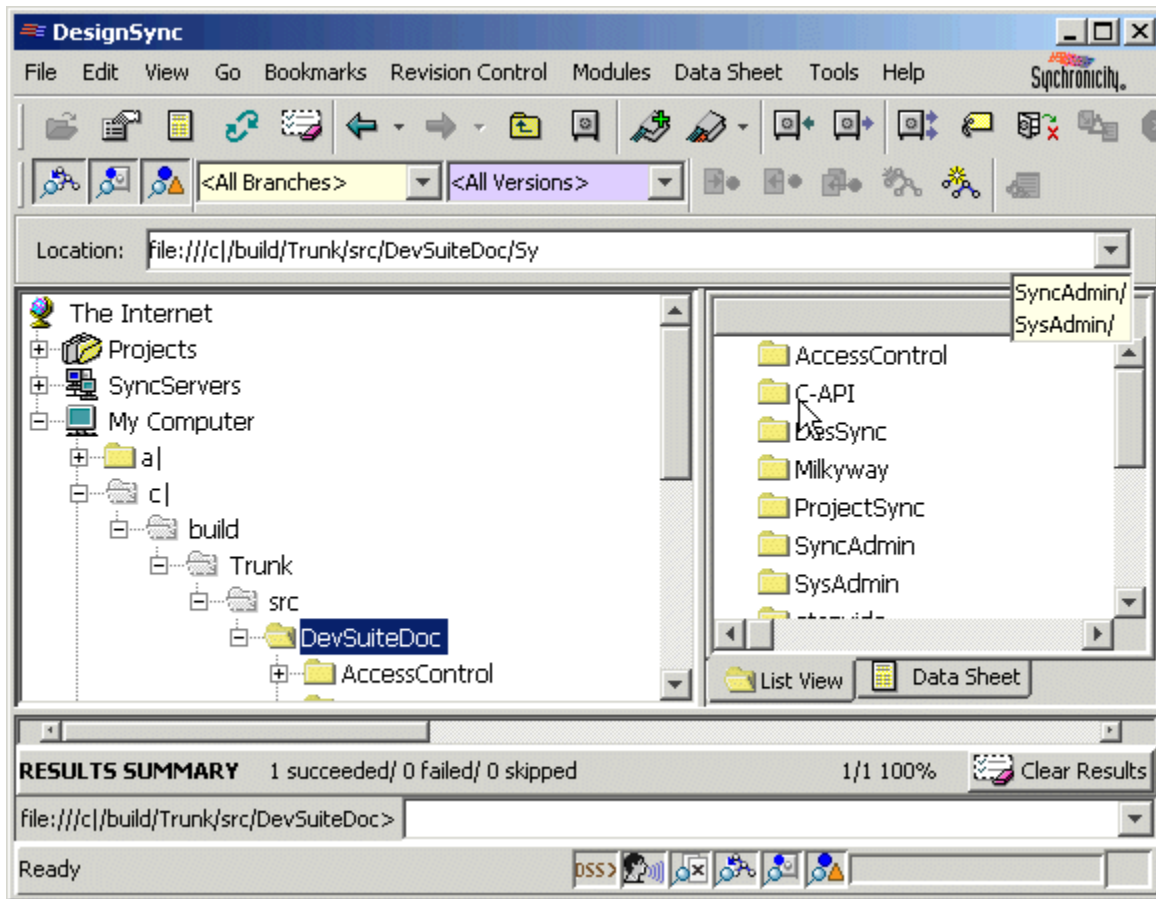
DesignSync does not update the Location Bar when you navigate from the Command Bar using the **scd** (or **cd**) command. The Location Bar is always synchronized with the Tree and List Views, and navigating from the Command Bar does not update the Tree and List Views.

#### File Name and Path Name Completion

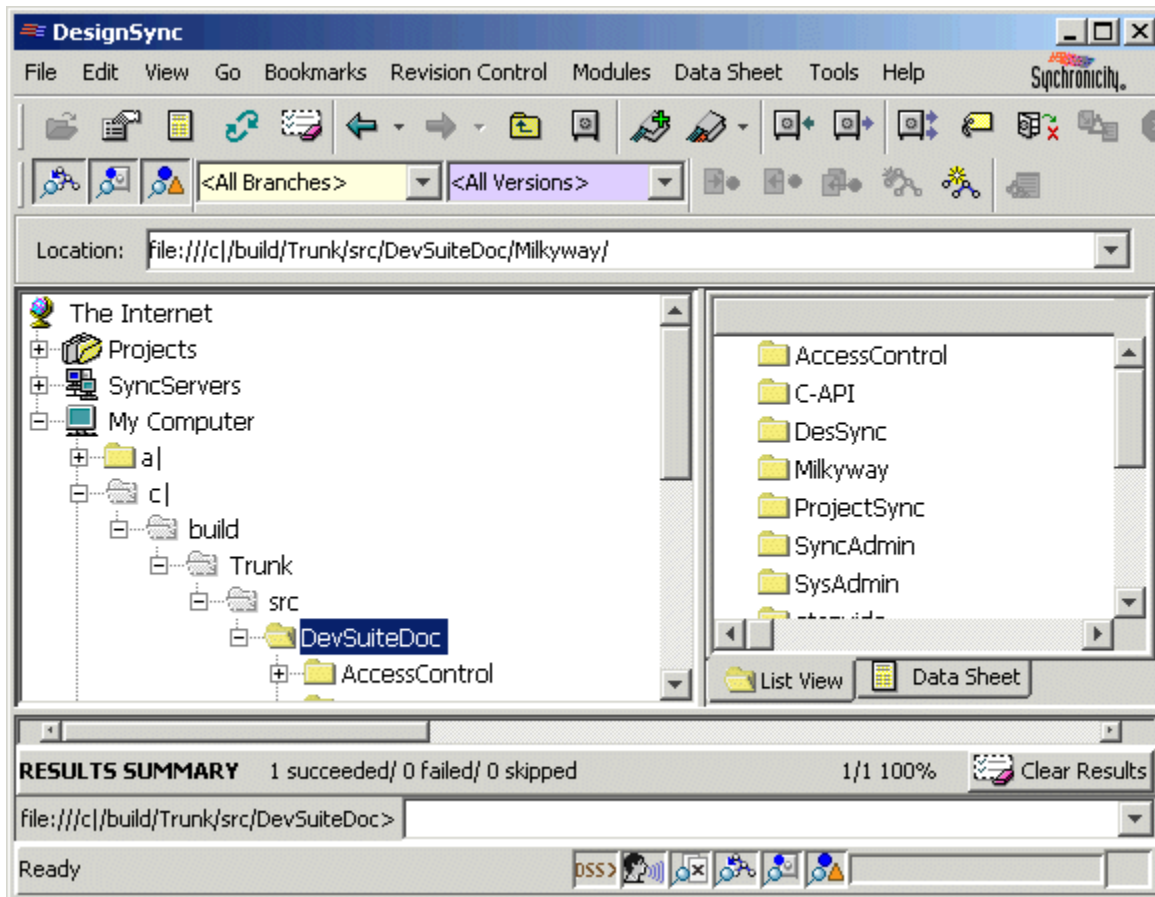
While in the Location Bar, you can press the **<Tab>** key for file name or path name completion.

**Note:** When entering text in the location bar for file/path name completion, remember that the DesignSync application is case sensitive when it comes to the characters entered. A lower case " **y**" entered will not complete file or path names that with start with a capital "**Y**".

In the example below, if you entered **sy** in the location bar and then you pressed the **<Tab>** key, you would get a small pop up list with **SyncAdmin** and **SysAdmin** to choose from since there are two folders that start with "**Sy**".



In the example below, if you entered **m** in the location bar and then you pressed the **<Tab>** key, you would get a file complete because there is only one folder that starts with **M**.



#### Related Topics

[Going to a Location](#)

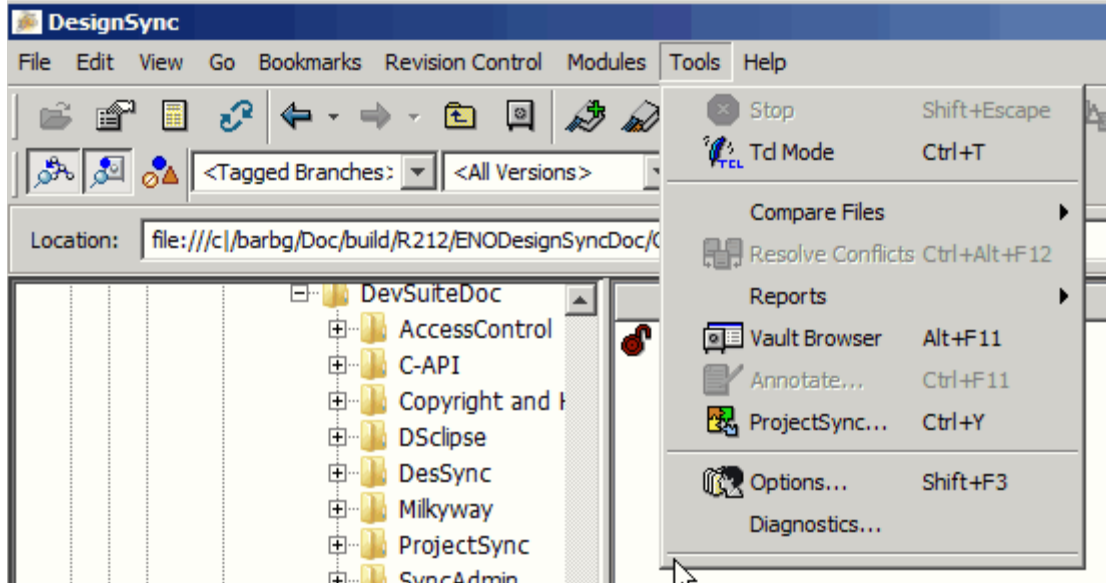
[Go Menu](#)

[ENOVIA Synchronicity Command Reference Help:scd](#)


[ENOVIA Synchronicity Command Reference Help:cd](#)

#### Tools Menu

The Tools Menu contains the list of command and utilities to help you maintain and manage your DesignSync application.



The following actions or options can be selected from the Tool Menu:

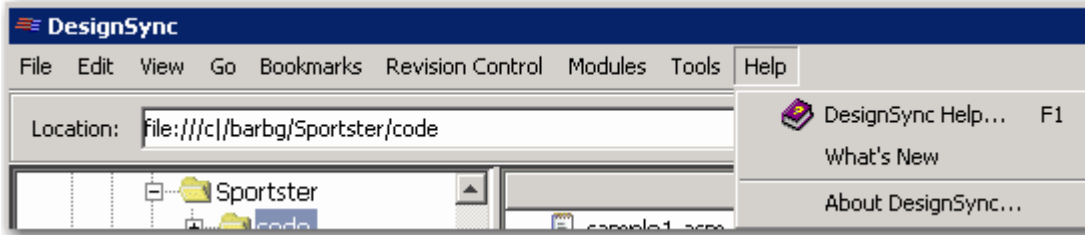
<b>Action/Option</b>	<b>Result</b>
<b>Stop</b>	<p>Interrupts or cancel an operation. If you interrupt a command operating on multiple files, the command completes its operation on the current file before stopping. Generally, you cannot stop or interrupt operations that complete immediately.</p> <p>There is also an icon (  ) for <b>Stop</b> on the toolbar.</p>
<b>Tcl Mode</b>	<p>Toggles the command line mode between dss mode (unchecked) and stcl mode (checked). The default mode is dss. The current mode is displayed on the status bar below the command shell window.</p> <p>You can also right-click in the status bar to choose between dss or stcl mode.</p>
<b>Compare Files</b>	<p>Gives you the following options to compare files:</p> <ul style="list-style-type: none"> <li>• <b>Show Local Modifications</b> – Compares the highlighted object in your work area with the original version that you checked out. This report shows changes made in your work area since the object was checked out.</li> <li>• <b>Compare to Latest</b> – Performs a 3-way diff comparing the highlighted object in your work area with the current version in the vault, using the original version in the vault as the common ancestor</li> <li>• <b>Compare Original to Latest</b> – compares the original version of the highlighted object with the latest version in the</li> </ul>

	<p>vault. This report shows changes made to the vault by others since the object was checked out.</p> <ul style="list-style-type: none"> <li>• <b>Compare to Previous Version</b> - compares the selected version with the previous version in the vault. If this object is the first object on the branch, it compares with the last version before the branch operation. For module members, it compares with the previous member version of the file.</li> <li>• <b>Compare 2 Files</b> – Two files have to be highlighted to have the option available. Compares the two files and displays the comparison.</li> <li>• <b>Advanced Diff</b> – Invokes the Advanced Diff dialog box. See Advanced Diff Options for more information.</li> </ul> <p>See Common Diff Operations for more information on the first four options.</p>
<b>Resolve Conflicts</b>	Helps you resolve conflicts when changes to the same lines in a file have been made by multiple users in multiple workspaces. This menu choice is only enabled when you have performed a merge update into your local workspace, and these updates conflict with your local changes.
<b>Reports</b>	<p>Gives you four Report options to compare files:</p> <ul style="list-style-type: none"> <li>• <b>Version History</b> – Invokes the Version History dialog box. The results of the Version History dialog box displays status information and version history for local objects. See Displaying Version History for more information.</li> <li>• <b>Changed Objects</b> – performs a recursive search of a folder to find all managed changed objects in that workspace or its subdirectories. See Identifying Changed Objects for more information. Note: This option used to be called Modified Objects.</li> <li>• <b>Contents</b> – Invokes the Contents dialog box. See Displaying Contents of Vault Data for more information. When the command is committed by pressing OK, DesignSync returns a list of the vault data.</li> <li>• <b>Compare</b> – Invokes the Compare Workspace/Selectors dialog box. See Compare the Contents of Two Areas for more information. When the command is committed by pressing OK, DesignSync returns a comparison of the contents of two areas.</li> </ul>
<b>Vault Browser</b>	Invokes a graphical representation of the branch and revision history of a managed file.
<b>Annotate</b>	Invokes a line-by-line annotation of a managed text file, pre-pending each line in the file with the version in which it was

	introduced, or last changed; the username of the user who made the change; and the date.
<b>ProjectSync</b>	Launches a new window with the installed version of ProjectSync associated with your workspace.
<b>Options</b>	Invokes the SyncAdmin tool. See the DesignSync Data Manager Administrator's Guide system for more information.
<b>Diagnostics</b>	Invokes the DesignSync Diagnostics dialog box. See the <i>DesignSync Data Manager Administrator's Guide: DesignSync Environment</i> for more information. When the command is committed by pressing OK, DesignSync displays diagnostic information that can help you troubleshoot a problem with your DesignSync environment.

## Help Menu

The Help menu gives you access to the help for the application as well as important information such as summary of new features and contact information.



The following actions or options can be selected from the Help Menu:

<b>Action/Option</b>	<b>Result</b>
<b>DesignSync Help</b>	Invokes the DesignSync help system.
<b>What's New</b>	In the List View window, displays a summary of new DesignSync features. You can choose to show the summary at startup.
<b>About DesignSync</b>	Displays the DesignSync version and ENOVIA contact information.

### Related Topic

Hints for Using help

### Special keystroke operations

The following special keystroke operations are not attached to any menu choices:

Key	Operation
<b>F2</b>	Initiates editing in place in a table or tree.
<b>Alt-F2</b>	Toggle tab/desktop in view pane.
<b>F6</b>	Toggle focus between panes of a splitter bar.
<b>Shift-F6</b>	Set keyboard focus to a splitter bar, so that it can be moved with the arrow keys.
<b>F10</b>	Set keyboard focus to the menu bar.

## Classic Windows and Panes

### View Pane

The View pane displays one or more views showing information about the objects being operated on. The List View, which shows the contents of the folders selected in the Tree View, is always visible. Some menu choices, such as **File =>Data Sheet**, will cause other views to appear.

The view pane can take on two forms:

- In the **tabbed view**, only one view is visible at a time. At the bottom of the view pane are a list of tabs representing the available views; you can switch to a view by clicking on a tab. (If the List View is the only available view, the tabs are not displayed.)

You can also click the right mouse button on the tabs to display a context menu with the following options:

- Close the view (not available for the List View).
- Close all other views except the current view (and the List View).
- Switch to desktop view.
- In the **desktop view**, each view appears as a window that can be moved around a desktop. Each window has a frame with buttons to maximize, minimize, and close the window. You can also click the right mouse button on an empty space in the desktop to display a context menu with the following options:
  - Cascade the windows (arrange them in an overlapping manner).
  - Tile the windows (arrange them in a non-overlapping manner).
  - Switch to tabbed view.

You can switch to the tabbed view by selecting the **View =>Tabbed View** menu choice. You can switch to the desktop view by selecting the **View => Desktop** menu choice. You can also toggle between the two views by pressing the **Alt-F2** key.

You can operate on items inside the view pane by selecting them and choosing from the menu bar, or right-clicking and displaying the context menu.

### Related Topics

DesignSync Symbols and Icons

## Modules Explorer

The Modules Explorer provides a different paradigm for working with modules data in your workspace.

A file system-centric structure of the data may show member files from different modules in a single folder along with any unmanaged files. The Modules Explorer shows a module-centric structure of the module members belonging to a specific module instance in the workspace. The Modules Explorer is displayed in the Tree view under the Module Roots "folder" (revealed when My Computer is expanded). Unless you have added an initial module root from the Modules menu, the Modules Explorer is empty when you log on to the DesignSync GUI.

### Related Topics



Tree View Pane

List View Pane



Exploring Modules

## Tree View Pane

Use the **Tree View** (left pane) to browse any of the following:

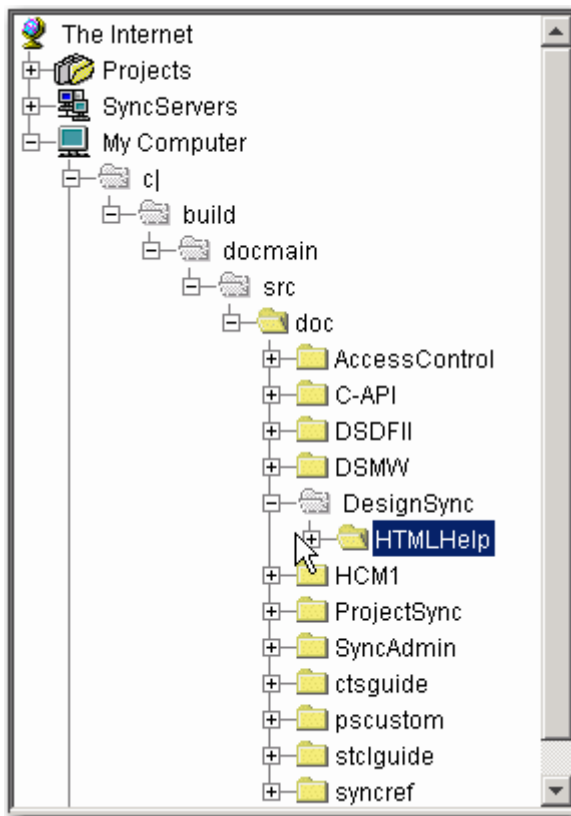
- Folders on your local client system (My Computer)
- Module instances you have created in your workspaces.
  - In the Folder View, these are contained in the module base directory (represented by the  icon).
  - In the Modules view, these are represented by the  icon.
- Module roots
  -



- In the Modules view, these are represented by the  icon. All known roots are kept within the Module roots container, represented by the  icon.
- Available servers (SyncServers)
- Projects

From the Tree View, you can perform the following actions:

- To expand a collapsed item, left click the plus sign (+) or double click on the item.
- To collapse an expanded item, left click the minus sign (-) or double click on the item.
- To select an item, left click on the item. The contents of the item are displayed in the List View pane.
- To operate on an item, right click on the item and choose an operation from the context menu.



- **Note:** A CD drive will display a folder icon if there is no CD in the drive bay.

Gray folders have not been directly traversed to get to a subfolder. In the picture above, the HTMLHelp subfolder folder was reached by selecting a bookmark. Hovering the cursor over a gray folder displays the following message in the

Status Bar: "You have not visited this node. The node will be automatically refreshed if you visit it."

For information on adding a sync server to the sync server list, `sync_servers.txt`, see SyncServer List Files in the *DesignSync Data Manager Administrator's Guide*.

### Related Topics

DesignSync Symbols and Icons

## List View Pane

The **List View** (right pane) gives you a list view of an object that you have selected from the Tree View (left pane). The title of the pane is updated to display the path of the selected object.

The List View has several columns that provide information about the displayed objects, such as the object name, revision-control status, and version. The columns provide the same information that is available from the **ls** command. You can click on the column headings to sort the displayed objects based on that column. See ENOVIA Synchronicity Command Reference Help: **ls** command for more information.

Files excluded from view by exclude files are not displayed by the DesignSync GUI. For more information on exclude files, see Working with Exclude Files.

From the List View, you can do the following:

- To select an item, left click on the item (use the **Ctrl** and **Shift** keys to select multiple items).
- To view or edit a file, double click on the file.
- To expand a folder, double click on the folder.
- To operate on an item, right click on the item and choose an operation from the popup menu.
- To close all of the views except your present view, right click on the List View tab and select Close Other Views.
- To switch from a tabbed view to a Desktop view, right click on the List View tab and select Desktop.

You can choose whether to keep the command shell window synchronized with the tree and list views by selecting **Tools =>Options =>GUI Options =>Options**.

You can specify how frequently information should be refreshed by selecting **Tools =>Options =>GUI Options =>Options**. See the SyncAdmin Help: GUI Options topic for more information.

You can control which columns display by selecting **Tools =>Options =>GUI Options =>Columns**. See the SyncAdmin Help: Customize Columns topic for more information.

You can also resize and change the order of the columns in the List View:

- Click and drag the separator bar to the right of a column header to resize the column.
- Click the column header and drag it to a new position.
- Mouse-over the column header to display tips in the Status bar.

If the display for the Branch and Version Tags fields are truncated due to length, double-click the file in the Branch or Version Tags column (or press **F2**) to switch to a list box view.

You can customize the look and feel of the List View by selecting **Tools =>Options =>GUI Options =>Display**. See the SyncAdmin Help: Customize Display topic for more information. The following table provides a description of each column that could display in the list view.

**Note:** The available options may change depending on what objects you're viewing.

Column Name	Description
Branch	<p>The branch tags of a managed object's current branch, or the persistent selector list for a folder. For example, where an object has two branch tags, the column displays:</p> <pre>Trunk,RelA</pre> <p><b>Notes:</b></p> <ul style="list-style-type: none"> <li>• If the folder belongs to a module, this column is blank. If the folder belongs to a legacy module, it contains the current branch tags or selector list.</li> <li>• The Branch column displays the branch tags of a managed object's current branch, or the persistent selector list for a folder. In cases where a managed object's persistent selector list is different from the parent folder's persistent selector list, the Branch column for the managed object also displays (!). For example:</li> </ul> <pre>(!) Trunk,RelA</pre> <ul style="list-style-type: none"> <li>• The persistent selector lists can disagree if the <b>setselector</b> command has been applied to the managed object. See the ENOVIA Synchronicity</li> </ul>

	<p>Command Reference Help: setselector for more information. Synchronicity does not recommend setting the persistent selector list on individual objects, because populate operations do not obey per-object persistent selector lists</p> <ul style="list-style-type: none"> <li>• Unmanaged items that have a selector individually set in a manner which is different than the parent folder show the -&gt; symbol and the item's selector, in addition to the (!) symbol.</li> <li>• If the Branch value exceeds 250 characters, the value is truncated as indicated by four periods (....). Select an object's displayed branch tags and then double-click or press the [F2] key to display a list of all branch tags on the object.</li> </ul>
Cache URL	The URL location of the cache.
Description (Project/Sync Server)	The description of the ProjectSync project.
Locker	<p>The username of the locker, or is blank if the object is not locked. In addition to the name of the locker, this column displays an asterisk (*) if the object is locked in this location, or displays no asterisk if locked elsewhere. For example:</p> <p><i>Jeff*</i> shows that the file is locked in this location by user Jeff</p> <p><i>Jeff</i> shows that the file is locked elsewhere by user Jeff</p> <p><b>Note:</b> The asterisk lets you sort first by locking user, then by local or remote locks (*).</p>
Member of	Lists the a name of the module or modules in which this object is a member.
Modified/ Modified Time	The time the file was last modified.
Name	The name of the object.
Project URL	The URL location of the ProjectSync project.
Referenced Target	This field is only displayed when the item is a hierarchical reference.
Size	The size of the object, in bytes. For collection objects, the

	Size column displays the total number of member files in the collection. Sizes of server-side files are not available to your DesignSync client and will be displayed as ""-"
Status	The revision-control status of the object. See Status Types for more information.
Type	The object type: File, Folder, Referenced File, Link to File, Link to Folder, Hard-Link, Cached File, Mirrored File, Module Branch, (Module) View, Module Member, Module Folder, and vendor-specific object types. Vendor specific object types include Cadence or Synopsys libraries, cells, and cell views, and custom collection objects.
SubType	The object subtype: Normal, Release, Snapshot. This field viewable when browsing module branches on the server and indicates the branch type. Release branches are immutable and Snapshot branches are content immutable. See Module Snapshots.
Version	<p>The current version number of the object and upcoming version if object is locked or auto-branched. For example:</p> <p>1.3</p> <p>1.7-&gt;1.8</p> <p>The version may also indicate <code>Reference to: 1.3</code> if the item is a reference. Similar output is presented for Mirror and Share links.</p>
Tags	<p>The tags currently on this object's version. This column lists tags in the order of Most-Recent to Oldest (the reverse order of when the tags were added).</p> <p><b>Note:</b> If the Tags column value exceeds 250 characters, the value is truncated as indicated by four periods (...). Select an object's displayed version tags and then double-click or press the [F2] key to display a list of all version tags on the object.</p> <p>For modules, this field displays the "tags of interest" which includes the following:</p> <ul style="list-style-type: none"> <li>• The comma separated tag list in the showtags property for the Module instance.</li> <li>• The comma separated tag list defined in the <code>AlwaysShowTags</code> registry key.</li> </ul>

- Any additional overlaid selectors defined for the workspace. For more information on overlaid selectors, see Module Member Tags.

For more information on defining tags to display in the registry or information on viewing the value of the showtags property, see *ENOVIA Synchronicity DesignSync Data Manager Administrator's Guide: Modules Registry Settings*.

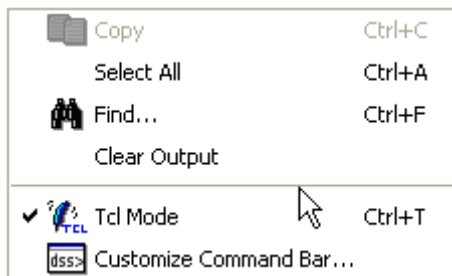
#### Related Topics

DesignSync Symbols and Icons

### Output Window

The **Output Window**, located above the Results Summary window, displays any output produced by commands typed into the Command Shell window and by some graphical operations.

You can drag and drop single lines from the Output Window into the Command Shell Window, You may need to edit the line to remove prefix characters. Right-clicking in the Output Window displays this menu:









from which you can:

- Select and find text in the output
- Clear the output
- Toggle between dss and Tcl (stcl) modes
- SyncAdmin Help: Customize the Command Bar


### Results Display

As soon as an operation begins, the Result Column appears in the List View, and displays the outcomes of the revision control operations.

For example, while populating a project, the Result Column might display:

Name ▾	Result
 fixed_tags_and_movable_tags.htm	 Success - Already fetched, and still unmodified version 1.20
 get_tags_versions.htm	 Success - Fetched version: 1.3
 getting_a_version_history.htm	 Success - Already fetched, and still unmodified version 1.20

At the completion of the populate operation, the Result Bar shows:

<b>RESULTS SUMMARY</b> 2 succeeded/ 0 failed/ 878 skipped	# 880	 Clear Results
---	-------	---

In this populate operation, 2 updated files were fetched into your work area, none of the files failed to populate, and 878 files were unchanged. The number at the right side of the column, **# 880**, indicates the total number of files affected by the operation.

Clicking the **Clear Results** button removes the Result Column from the List View's display.

## Command Shell Window

The **Command Shell Window** is the command bar where you type in commands – as opposed to executing commands through menu selections or Tool Bar buttons:

`stcl>`

Experienced users sometimes prefer typing in a command rather than using the graphical interface. There are two command-line modes that you can select from the **Options** button:

- **dss** (DesignSync shell) – the default shell
- **stcl** (Synchronicity Tcl) – provides access to all DesignSync commands and the Tcl environment

DesignSync displays either `dss>` or `stcl>` in the status area in the lower-left corner of the DesignSync window to indicate which mode you are in. Note that unlike the `dss` and `stcl` shells, the `dss` and `stcl` modes from the graphical interface do not communicate with `syncd`. In this way, the command bar from the graphical interface is more like the `dssc` and `stclc` shells.

You can also select previously entered commands by using the up and down arrow keys or clicking the drop-down arrow to the right of the command line.

You can select the prompt displayed for the command shell window:

- Enter the **prompt -url** command to display the current directory as the prompt.
- Enter the **prompt -default** command to display the **dss>** prompt. This setting is the default.

## DesignSync Data Manager User's Guide

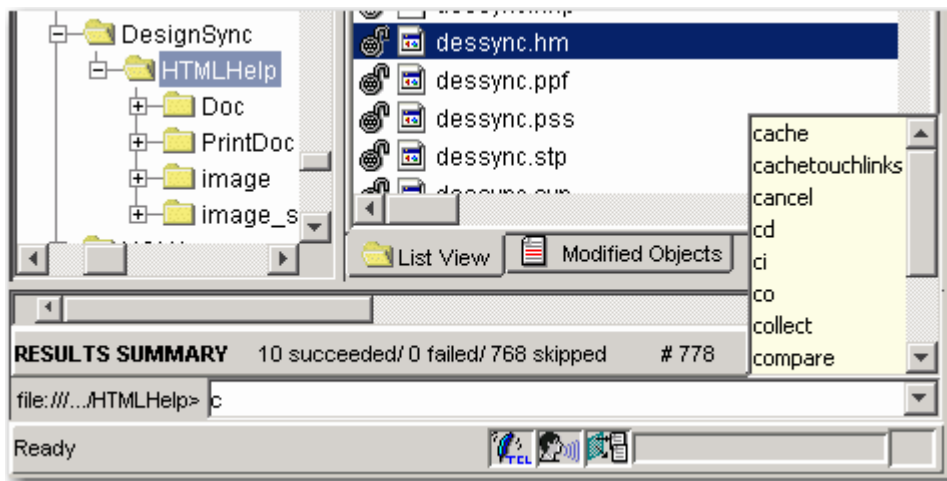
You can choose whether to keep the command shell window synchronized with the tree and list views by selecting **Tools =>Options =>GUI Options =>Options**. See SyncAdmin Help: GUI Options topic for more information.

Right-clicking in the Command Shell Window displays the context sensitive menu shown in the Output Window topic.

You can also execute command-line commands from a DesignSync shell instead of from the graphical interface. DesignSync command-line shells, and the Command Shell window, use the command line defaults system.

### Command Line Completion

In the command bar, press the **<Tab>** key for command line completion. Pressing the **<Tab>** key displays a pop up box with matching recent commands, from which you can select a command to run. In the example below, when you enter the character **c** in the command bar and then press **<Tab>** key, you get a list of all commands that start with the letter **c**.



**Note:** When entering text in the location bar for file/path name completion, remember that the DesignSync application is case sensitive when it comes to the characters entered. A lower case " y" entered will not complete file or path names that with start with a capital "Y".

### Related Topics

[DesignSync Command-Line Shells](#)

[Command Line Defaults System](#)

[ENOVIA Synchronicity Command Reference: Command defaults](#)

### Status Bar



The status bar displays status messages during DesignSync operations and also allows you to set display options.

Right-click in the status bar to toggle between dss and Tcl mode. The default mode is dss. To change it to Tcl (stcl) mode, select **Tools =>Tcl Mode**. A check mark next to **Tcl Mode** means you are in stcl mode; otherwise, you are in dss mode. DesignSync displays the mode (DSS or STCL) in the status area next to the command shell window.

You can also right-click on the icons on the right side of the status area to toggle between options such as:

- Toggle between dss and Tcl mode
- Toggle between various display modes for the View Pane including: Show Excluded Objects, Show Hierarchical References, and others.

#### Related Topics

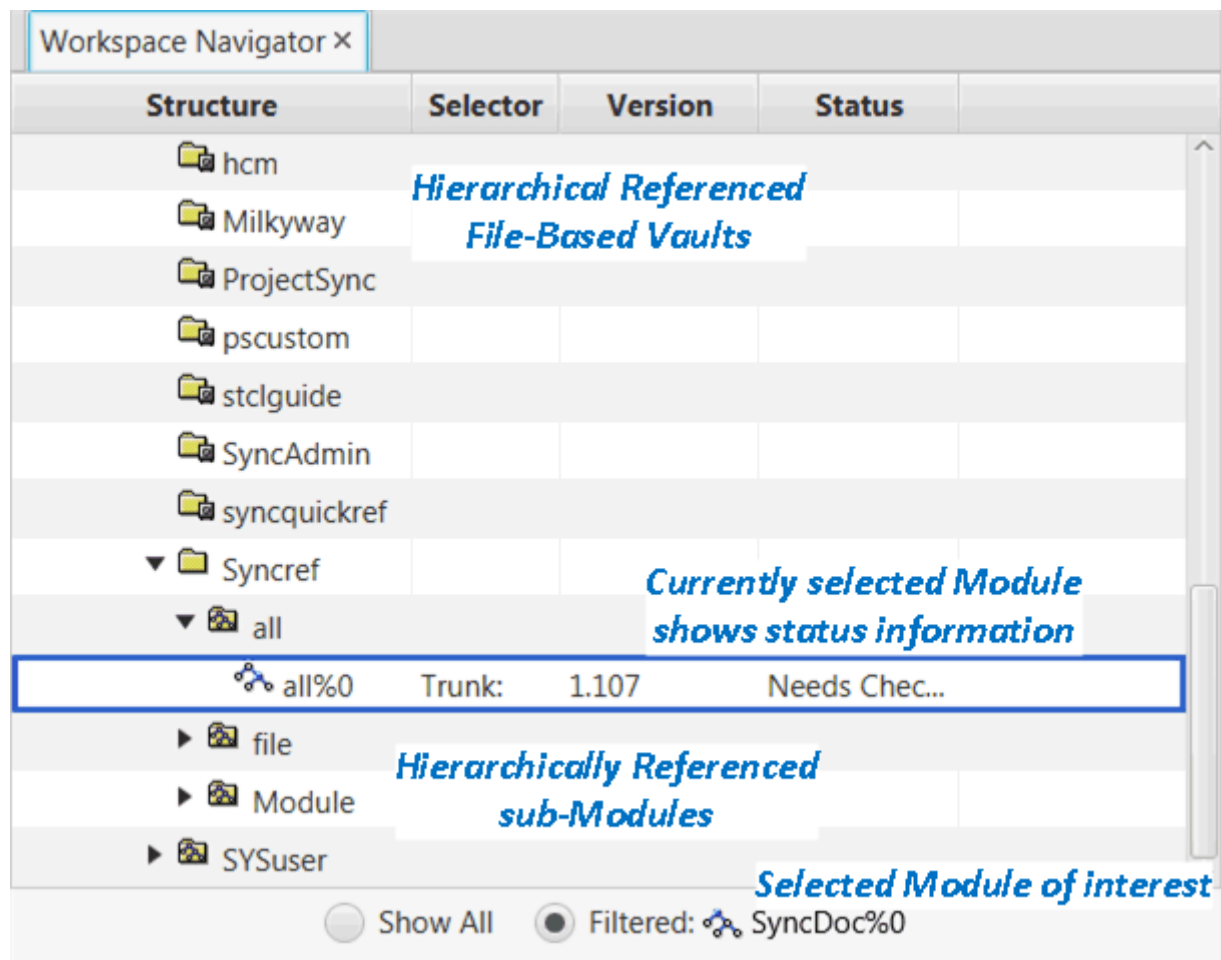
DesignSync Command-Line Shells

## Workspace Structure Browser Windows and Views

### Workspace Navigator

The Workspace Navigator view in the Workspace Structure Browser context shows the layout of module instances and their hierarchies in the file system, beginning with the user's workspace root directory. For Modules and sub-modules, it also shows selector, version, and status information. For hierarchical references to other object types, such as IP, file-based vaults, external modules, or legacy modules, it shows the equivalent workspace directory with no status information.

**Click on the fields in the following illustration for information.**



### Workspace Navigator View Descriptions

#### Workspace Navigator View

The Workspace Navigator view displays the workspace structure on the local disk, beginning from the workspace root and continuing through the local disk structure. Modules that are not populated in the workspace are not visible in the Workspace Navigator pane. The display reflects the way the module is populated. If the module is populated in peer-structure, you will see all the hierarchically linked modules in a peer structure. This example is a cone structure with a top-level module populated at the workspace module root, and sub-modules and hierarchically referenced file-based vaults populated beneath it.

If you select an object in the Workspace Navigator view, you can launch the context menu to perform object-specific Workspace Navigator View Actions.

#### Filter

If a workspace root has multiple top-level module instances, the workspace navigator shows multiple module hierarchies in the workspace or can be adjusted to filter only the module hierarchy of interest.

- Show all - Show all modules beneath the workspace root.
- Filtered: - Show only the hierarchy related to the module selected when the Workspace Structure Browser was launched or the specified module of interest, if a module of interest was specified. The interface displays the workspace instance name of the selected module so you always know what module are you viewing.

### Related Topics

Using the Workspace Structure Browser

Workspace Navigator View Actions

Module Hierarchy

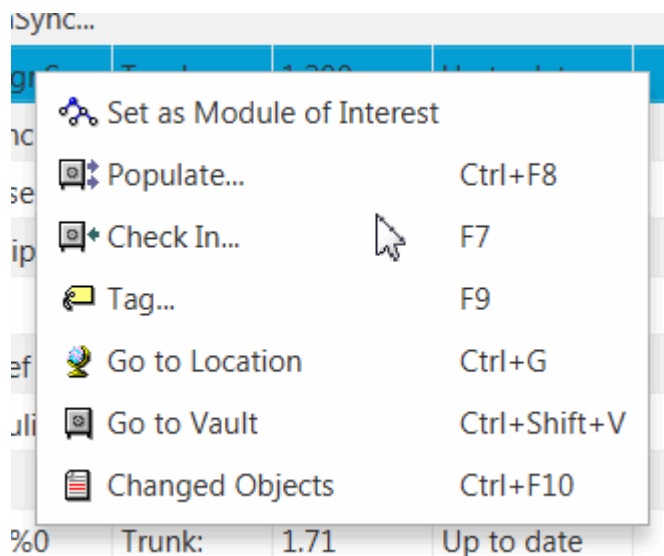
Properties

Design Assistant

### Workspace Navigator View Actions

The specific actions available for a selected option in the Workspace Navigator depends on the type of object being selected. Now all actions are available for all types of objects.

**Click on the fields in the following illustration for information.**



<b>Action/Option</b>	<b>Applicable To:</b>	<b>Result</b>
<b>Collapse/Expand</b>	Folder	Expands or collapses the folder being viewed.
<b>Expand All</b>	Folder	Expand all folders beneath the selected folder.
<b>Set as Module of Interest</b>	Module External Module	Set the selected module or external module as the object of interest. All operations in the Workspace Structure Browser revolve around the selected module of interest. When you Restore Context, the context will center around the module selected as the module of interest.
<b>Populate</b>	Module External Module	Launches the populate dialog.
<b>Check in</b>	Module	Checks in the specified module.
<b>Tag</b>	Module External Module	Tags the specified module or external module.
<b>Go to Location</b>	Module External Module Cached Module File-based Vault Folder	Switch to Classic Context viewing the selected object in the workspace, with the object selected in the Tree View Pane.and the module members of the objects displayed in the List View Pane.
<b>Go to Vault</b>	Module File-based Vault Folder	Switch to Classic Context viewing the selected object on the server, with the module version selected in the Tree View Pane.and the module members of the version displayed in the List View Pane.  <b>Note:</b> This command uses the selector associated with the workspace to display the appropriate branch.
<b>Changed Objects</b>	Module Folder	Launches the Changed Objects Report with the selected object as tree root for the changed object report.

**Related Topics**

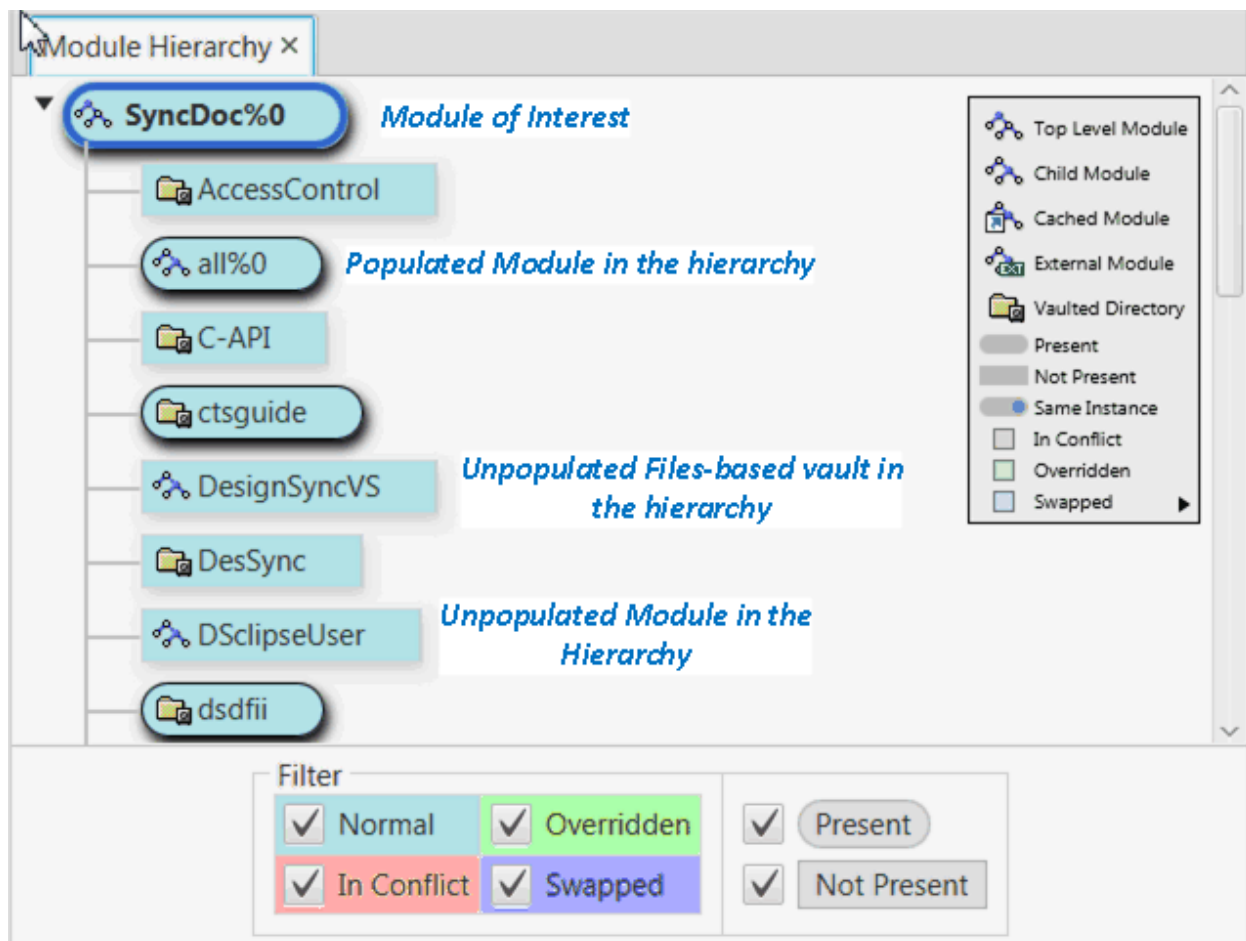
Using the Workspace Structure Browser

Workspace Navigator View

## Module Hierarchy View

The Module Hierarchy view in the Workspace Structure Browser context shows graphically the module hierarchy of the module of interest. If the workspace structure browser was not launched with a module selected (for instance, if you had a folder selected), there is no default module of interest, and you must designate a module as the module of interest in order to view the module hierarchy. The module hierarchy is based on the server view of the module, showing the hierarchical references between the modules. This view shows all hierarchical references of the module, regardless of whether the complete module is populated in the workspace.

Click on the fields in the following illustration for information.



## Module Hierarchy View Descriptions

Module Hierarchy View

The Module Hierarchy pane displays the module structure, beginning from the module of interest, and following the hierarchy. Modules that are not populated in the workspace are displayed with a square box as shown above. The display reflects the way the module is defined.

If you select an object in the Module Hierarchy view, you can launch the context menu to perform object-specific Module Hierarchy View Actions.

### Filter

The filter can be used to focus the module hierarchy display according to the following module statuses:

- Normal - show all modules that are populated normally.
- In Conflict - shows conflicting hierarchical references; multiple references to different versions of the same module. When this option is not selected, conflicting hrefs are not highlighted.
- Overridden - shows both the populated and the overridden hierarchical references. When this option is not selected, the hrefs that are overridden by other hrefs are not displayed.
- Swapped - shows the workspace version of modules that are swapped. When this option is not selected, the version displayed is indicated by the hierarchical reference, not the workspace version.
- Present - Shows the modules populated in the workspace.
- Not Present - Shows the modules not populated in the workspace.

### Related Topics

[Using the Workspace Structure Browser](#)

[Workspace Navigator View](#)

[Properties View](#)










[Design Assistant View](#)

### Module Hierarchy View Actions

The specific actions available for a selected option in the Workspace Navigator depends on the type of object being selected. Now all actions are available for all types of objects.

**Click on the fields in the following illustration for information.**

---

 Collapse	
 Expand All	
 Populate...	Ctrl+F8
 Check In...	F7
 Tag...	F9
 Add Href...	Ctrl+Shift+H
 Go to Location	Ctrl+G
 Go to Vault	Ctrl+Shift+V
 Changed Objects	Ctrl+F10

---

**Note:** All actions are inactive for hierarchically referenced objects that are not populated in the workspace.

<b>Action/Option</b>	<b>Applicable To:</b>	<b>Result</b>
<b>Collapse/Expand</b>	Module	Expands or collapses the selected module. A module can only be expanded or collapsed if it has children.
<b>Expand All</b>	Module	Expand all levels of hierarchy for the selected module.
<b>Populate</b>	Module External Module	Launches the populate dialog.
<b>Check in</b>	Module	Checks in the specified module.
<b>Tag</b>	Module External Module	Tags the specified module or external module.
<b>Go to Location</b>	Module External Module Cached Module File-based Vault	Switch to Classic Context viewing the selected object in the workspace, with the object selected in the Tree View Pane.and the module members of the objects displayed in the List View Pane.
<b>Go to Vault</b>	Module File-based Vault	Switch to Classic Context viewing the selected object on the server, with the module version selected in the Tree View Pane.and the module

		members of the version displayed in the List View Pane.  <b>Note:</b> This command uses the selector associated with the workspace to display the appropriate branch.
<b>Changed Objects</b>	Module File-based Vault	Launches the Changed Objects Report with the selected object as tree root for the changed object report.

**Related Topics**

Using the Workspace Structure Browser

Workspace Navigator View

Module Hierarchy View

Properties View

Design Assistant View

**Properties View**

Properties view displays information about a selected object such as a module, folder, version, or vault. To display the object properties, select an object in either the Workspace Navigator View or the Module Hierarchy View.

The information that is displayed depends on the object you selected. For example, for a module, the properties include: the module instance name, the base directory, Server URL, filters, href filters, views, href mode, module type, parents, module root, selector, module status, module swap status, version, tags, top-module status (whether the module is a top-module). For a file-based vault, the properties include: the directory name, vault URL, configuration type, path, and selector.

**Related Topics**

Using the Workspace Structure Browser

Workspace Navigator View

Module Hierarchy View

Design Assistant View

**Design Assistant View**



The Design Assistant monitors the selected objects in the workspace browser and determines the state of the selected object and what possible actions can be taken and why those actions are applicable. For example, the Design Assistant can provide guidance to resolving conflicting hierarchical references, determining whether your module is current or out-of-date, providing information about how to update your hrefs, etc.

If the state of an object is unknown, the Design Assistant will display a warning explaining the unknown status and providing suggestions for locating and fixing the problem.

The Design Assistant is intended as a starting point for diagnosing the work required to bring your workspace to an "up-to-date" status.

#### **Related Topics**

Workspace Navigator View

Module Hierarchy View

Properties View

## **DesignSync Shells**

### **DesignSync Command Line Shells**

In addition to DesSync, the DesignSync graphical user interface, DesignSync has four command-line client shells: `dss` (DesignSync shell), `dssc` (concurrent `dss`), `stcl` (Synchronicity Tcl), and `stclc` (concurrent `stcl`). You might use a command-line shell instead of the graphical interface if you:

- Are more comfortable with command-line interfaces
- Need to perform operations not available from the graphical interface
- Do not want to incur the overhead of running a graphical interface
- Want to create scripts to automate tasks or to perform operations in batch mode

Note that DesSync's command bar provides both Tcl and non-Tcl command-line modes so that you can perform command-line operations from the graphical interface.

#### **Related Topics**

Comparing the DesignSync Shells

Invoking a DesignSync Shell

Using DesignSync Commands in OS Shell Scripts

Creating DesignSync Scripts

Command Line Defaults System

Command Line Editing

ENOVIA Synchronicity Command Reference Help

## Comparing the DesignSync Shells

The primary differences of the four command-line clients are outlined below:

Shell	Uses syncd	Includes Tcl
dss	yes	no
dssc	no	no
stcl	yes	yes
stclc	no	yes

See ENOVIA Synchronicity Command Reference Help for more information on these command line clients.

### syncd versus no syncd

The dss and stcl clients communicate with a DesignSync server (SyncServer) through syncd. The syncd process can manage multiple dss/stcl requests per user, allowing one user to run parallel dss/stcl sessions. However, syncd handles requests serially, which can cause operations from one dss/stcl session to be blocked while operations from another session execute. It is therefore recommended that you use the concurrent shells: dssc and stclc. The dssc and stclc clients do not use syncd; they communicate directly with a SyncServer. The dssc and stclc clients also have the advantage of supporting more robust command-line editing than the dss and stcl shells. See Command-Line Editing for details. The stcl shell also does not support DesignSync command abbreviations as do the dss/dssc/stclc shells.

The only advantage of dss/stcl over dssc/stclc is that dss/stcl start-up time when a syncd is already running is less than dssc/stclc start-up time. If you frequently run DesignSync commands from your OS shell using the form:

```
% dss <command>
```

```
% stclc -e "<command>"
```

instead of staying within the shell, consider using dss/stcl.

## Tcl versus no Tcl

The stcl and stclc clients include Tcl, a powerful interpreted command language that provides programming constructs (such as variables, conditionals, and loops) in addition to all the DesignSync commands available from dss/dssc. Use the stcl/stclc shells when you need the scripting constructs of Tcl; otherwise, use dssc/dssc, which provide a simpler command environment. With stcl/stclc:

- You must use double quotes around objects that contain a semicolon (;), such as vaults, branches, and versions. The semicolon is the Tcl (and therefore stcl/stclc) command separator.
- You must specify the `-exp` option to execute a DesignSync command from the OS shell. Because stclc is primarily a scripting shell, an argument specified without `-exp` is assumed a script. With dss/dssc, the syntax for specifying a single command is simpler.

For more information on Tcl, including documentation for Tcl commands, visit the Tcl Web page:

<http://www.tcltk.com>

You can determine the version of Tcl included in your DesignSync installation's stcl interpreter by using the Tcl `info tclversion` and `info patchlevel` commands within an stcl/stclc client shell.

## Related Topics

DesignSync Command Line Shells

Invoking a DesignSync Shell

Command Line Editing

ENOVIA Synchronicity Command Reference Help: dss Command

ENOVIA Synchronicity Command Reference Help: dssc Command

ENOVIA Synchronicity Command Reference Help: stcl Command

ENOVIA Synchronicity Command Reference Help: stclc Command

## Invoking a DesignSync Shell

To invoke a DesignSync command-line shell, type one of the following commands from your UNIX or Windows shell:

## DesignSync Data Manager User's Guide

```
dss
```

```
dssc
```

```
stcl
```

```
stclc
```

On Windows platforms, you can also invoke DesignSync shells from the Windows **Start** menu.

For `dss` and `dssc`, your prompt changes to **dss>** to indicate that you are in the `dss/dssc` shell. For `stcl/stclc`, the prompt is **stcl>**. You can now enter any of DesignSync's command-line commands, and if you are in an `stcl/stclc` shell, you can also use any Tcl construct.

The following example shows the invocation of `dssc` from a UNIX shell:

```
% dssc
dss>
```

To exit any DesignSync shell, type `exit`.

```
dss> exit
%
```

You can also enter DesignSync commands directly from the UNIX or Windows shell without remaining in the DesignSync shell. This capability is useful for executing occasional DesignSync commands, and for embedding DesignSync commands in OS shell scripts. To execute DesignSync commands without remaining in the shell, prefix the DesignSync command with **dss** or **dssc --** for example:

```
% dssc ls sync://myserver.myco.com:2647/Projects
<output from the DesignSync ls command>
%
```

For the `stcl/stclc` shells, you prefix the DesignSync or Tcl commands with **stcl** or **stclc**, and you must use the `-exp` option to indicate that you are executing a command, not running a script -- for example:

```
% stclc -exp "ls sync://myserver.myco.com:2647/Projects"
<output from the DesignSync ls command>
%
```

### Note:

The dss and stcl shells require a **syncd** daemon process. If syncd is not already running, it starts up automatically a few seconds after dss or stcl is invoked. You can also start syncd manually from a UNIX or Windows shell by using the **syncdadmin** command with the **start** argument. "syncdadmin start" starts syncd only if syncd is not already running -- only one syncd process can be running for a given user on a given machine.

## Related Topics

DesignSync Command-Line Shells

Comparing the DesignSync Shells

Command-Line Editing

Using DesignSync Commands in OS Shell Scripts

Creating DesignSync Scripts

Running Scripts

ENOVIA Synchronicity Command Reference Help

## Command Line Editing

Command-line editing including key shortcuts, command completion, and filename completion are supported for some DesignSync client shells. The same command-line editing support is provided for both Windows and UNIX clients; however, the command-line editing support varies depending upon the DesignSync client: stcl, stclc, dss, or dssc.

Note: The DesignSync GUI (DesSync) has a built-in command line interface with its own shortcuts and element completion. For information on the GUI command line, see Command Shell Window.

The following table lists the types of command-line editing support for each of these clients.

Type of Editing	stclc Shell	dssc Shell	stcl Shell	dss Shell
control key shortcuts	Supported	Supported	Adopts key bindings of platform	Adopts key bindings of platform
arrow key	Supported	Supported	Adopts key	Adopts key

shortcuts			bindings of platform	bindings of platform
command abbreviations	Supported	Supported	Not Supported	Supported
command completion	Supported	Supported	Not Supported	Not Supported
filename completion	Supported	Supported	Not Supported	Not Supported
Tcl history command and '! expansion	Supported	Not Supported	Not Supported	Not Supported

### Key Bindings

The following table describes the key bindings supported for the stcl and dssc shells.

Behavior	Control Keys	Special Keys
Forward one character	Control-f	Right arrow
Back one character	Control-b	Left arrow
Beginning of line	Control-a	Home (only supported for Windows platforms)
End of line	Control-e	End (only supported for Windows platforms)
Kill rest of line	Control-k	
Kill line	Control-u	<p>Esc</p> <p><b>Note:</b> The Esc key instead invokes vi command mode if your &lt;EDITOR&gt; environment variable is set to vi or your ~/.inputrc file contains the line 'set editing-mode vi'. Note also that the DesignSync GUI default editor setting does not affect the behavior of the Esc key.</p>

Delete character	Control-d  (Deletes character to the right of cursor) <b>Note:</b> Control-d exits shell if entered on an empty line.	Delete  (Deletes character to the left of cursor) <b>Note:</b> Control-d exits shell if entered on an empty line.
Previous command from command history	Control-p	Up arrow
Next command from command history	Control-n	Down arrow
Exit stcl/dssc shell	Control-d	

## Command and Filename Completion

The stcl and dssc shells support command and filename completion.

**Note:** Filename completion is supported for files, folders, and module workspace instance names because filename completion uses local file system objects. Filename completion is not supported for other DesignSync object types, such as Cadence view objects, DesignSync references, and server-side objects such as vaults or modules.

To use command and filename completion:

1. Type a partial command and press the **<Tab>** key.
2. If the command is unique, the command displays and you can enter options using option completion:

To view a list of the command's options, press '-' and then the **<Tab>** key. To complete a partial option, press '-' followed by the partial option and the Tab key. If the partial option is unique, the option displays. If the partial option is not unique, choose from the options listed by typing to the next unique character and pressing the **<Tab>** key.

**Note:** Option completion is supported for DesignSync commands, not for general Tcl commands.

3. If there are multiple matches for a command completion, select a command from the list of matched commands displayed by typing to the next unique character and pressing the **<Tab>** key again.
4. If your command line includes a filename, you can use filename completion:

Type the partial filename and press the Tab key. If the partial filename is unique, the filename displays. If the partial filename is not unique, select a filename from

the list of matched filenames displayed by typing to the next unique character and pressing the **<Tab>** key again.

5. Once you have completed the command line using command, option, or filename completion, you can edit the command line if necessary, then press Enter to invoke the command.

If no matches are found for command, option, or filename completion, the message "(no matches)" displays.

**Examples of Command, Option, and Filename Completion**

<b>If you type</b>	<b>The completion is</b>
pu<Tab>	puts
rm<Tab>	rmfile          rmfolder rmvault        rmversion
rmfi<Tab>	rmfile
ci -r<Tab>	recursive    reference    retain
ci -rec<Tab>	ci -recursive
ls <Tab>	.SYNC        Sub          x.v
ls S<Tab>	ls Sub
notetype <Tab>	create                  delete getdescription        rename

**Command History**

The stlc and dssc shells save a history of up to 100 commands so that you can select a previous command using one of these methods:

- To view previous commands, select Control-p or the Up arrow repeatedly until you see the command you wish to reinvoke. To invoke the command, press Enter. You can also edit the command line before reinvoking the command.
- To move forward through the command history after viewing previous commands, select Control-n or the Down arrow.
- In an stlc shell, you can use the Tcl `history` command to list the command history. Each command is preceded by a number. To reinvoke a command, type `!#` where `#` is the number of the command. To reinvoke a command, you can also type `!substring` where `substring` is an abbreviation of a previous command.



The command history is saved between sessions, so you can use these commands to navigate through the command history of your last session.

### Command History Search

In addition to moving forward and backward through command history, you can search the command history for previous commands:

1. Select Control-r to enter incremental search mode.

Incremental search mode is indicated by a '?' prompt.

2. Select from the following choices and enter the selection at the '?' prompt to search command history:

A character	Characters you type are added to the search string displayed to the left of the '?' prompt. As you type, the command history is searched for the current search string. If a match is found, the match displays to the right of the prompt. If no match is found, the area to the right of the prompt remains empty.
Enter	Invoke the command found by the incremental search mechanism and return to normal mode.
Esc	Terminate the search and return to normal mode.
Control-r	Set the direction of the search to reverse mode and continue searching. Repeat Control-r to find multiple previous commands that match the search string.
Control-s	Set the direction of the search to forward mode and continue searching. Repeat Control-s to find multiple following commands that match the search string. <b>Note:</b> Control-s is only applicable if you have already entered reverse mode using Control-r.
Backspace	To delete characters in the current search string, press the Backspace key. As you delete characters, the command history is searched for the current search string. <b>Note:</b> If you delete all of the characters in the search string, the search is set to the end of command history. If you are in forward mode (set using Control-s), no matches can be found because you are at the end of command history.

### Related Topics

DesignSync Command Line Shells

Comparing the DesignSync Shells

Using DesignSync Commands in OS Shell Scripts

Creating DesignSync Scripts

### Working with Command Aliases

You can create DesignSync command aliases, either from the command line or within scripts, to define your own DesignSync commands. For example, you can create an alias as follows:

```
dss> alias -args 1 go scd $1 && ls
```

Once defined, typing `go` with a folder argument has the effect of entering the command `scd <folder>` followed by `ls`. You cannot use the **alias** command to redefine the behavior of built-in DesignSync commands. For example, you could not have specified `scd` instead of `go` in this example.

Arguments to an alias are substituted for the **\$1** through **\$n** placeholders in the alias definition. In `stcl` mode, you must surround the argument placeholders with curly braces to ensure that they are not interpreted as Tcl variables. For example:

```
stcl> alias -args 1 go scd {$1} && ls
```

Alias definitions remember the mode, `dss` or `stcl`, in which they were defined. For example, if while in `stcl` mode you create an alias containing Tcl constructs, the alias will always execute in `stcl` mode even if you change to `dss` mode. The following alias contains Tcl constructs and demonstrates some of the details of `stcl` syntax:

```
alias -args 1 -- checkin if \[file exists \"$1 \] \{ci -new -noc  
  \"$1 \}
```

Things to note in this example:

- The `--` option ensures that the `-new` and `-noc` checkin options are not treated as options to the `alias` command.
- Backslashes are required for `[` and `]` to prevent the expression from being interpreted during alias definition.
- Backslashes are required for `{` and `}` to avoid substitution of the second `$1` placeholder.
- There must be a space after both occurrences of `\$1`, otherwise the alias command sees `"$1]"` (or `"$1}"`), which it does not recognize as `$1`, so no substitution takes place.

Other variations of the `alias` command include:

- `alias -delete <alias_name>`

Deletes the alias.

- `alias -temporary <alias_name> <alias_definition>`

Creates the alias, but does not store it permanently. If you exit DesignSync and come back, the alias is no longer defined.

- `alias -list`

Lists all available aliases.

## Related Topics

ENOVIA Synchronicity Command Reference Help: alias command

# Configuring the DesignSync Interface

## Configuring DesignSync

The DesignSync Administrator (SyncAdmin) tool is a graphical user interface that lets users easily configure the DesignSync client for individual use. See the ENOVIA Synchronicity Administrator (SyncAdmin) Help for more information on the SyncAdmin tool. The SyncAdmin tool is part of the DesignSync software distribution and is available on both UNIX and Windows platforms.

- To start SyncAdmin on Windows:

**Start =>Programs =>ENOVIA Synchronicity DesignSync V6R2011x => SyncAdmin**

- To start SyncAdmin on UNIX:

```
% SyncAdmin
```

SyncAdmin maintains your user preferences in a registry file.

**Note:** In order for changes made to your user registry file to take effect, you must restart your DesignSync client (exit and restart your DesSync, dssc, or stlc session, or if you are using dss or stcl, use the **syncdadmin** command to restart syncd). An alternative to restarting the DesignSync client is to use the **sregistry reset** command to re-read the registry files.

The DesignSync GUI's **Tools =>Options** menu invokes SyncAdmin, which you can use to set additional preferences pertaining to the DesignSync GUI.

You can also use environment variables to configure your DesignSync environment.

### Related Topics

Command Line Defaults System

*DesignSync Data Manager Administrator's Guide: Using Environment Variables*

*ENOVIA Synchronicity Command Reference Help: syncdadmin command*

*ENOVIA Synchronicity Command Reference Help: sregistry reset*

## Controlling Access to Your Local Work Area

Depending on the sharing methodology employed on your project, you may or may not want other users to be able to browse or even modify DesignSync objects in your work area. Access is based on two factors:

- The ability of other users to obtain the proper metadata lock
- Your operating system directory and file protections

For other users to browse objects in your work area, they must obtain a read lock on the associated metadata (`. SYNC`) directory. The ability to obtain a read or write lock on a file depends on the type of permissions set when the file is created. How DesignSync creates files depends on the operating system platform on which it runs:

- On the Windows platform, DesignSync creates files with open permissions. This level of permission lets anyone obtain metadata locks.
- On UNIX platforms, DesignSync creates files with permissions based on your umask. If users want to modify your work area, they must obtain a write lock, and therefore your umask must allow write access.

Once the proper lock is obtained, the directory- and file-level protections determine what access users have to objects in your local work area.

### Related Topics

Metadata Overview

Setting Up a Shared Workspace

## Setting Up a Shared Work Area

**Note:** Setting up a shared work area is a task that a UNIX system administrator performs.

To set up a shared work area you first must set up the team's UNIX environment for sharing by setting appropriate UNIX permissions.

DesignSync follows UNIX permissions such that:

- If User1 has UNIX permission to read User2's files, then User1 sees User2's managed objects as being under revision control.
- If User1 has UNIX write permissions for User2's files, then user1 can perform revision control operations on User2's managed objects.

You can set up a shared work area for either of the following situations:

- Individual users can still lock objects for revision control operations.

Shared work area can be set up such that locks are on an individual basis. In this model, individual users can lock objects for revision control operations. Also, only the UNIX owner of a file can checkout files with a lock (**co -lock**) to modify them and subsequently perform check in or cancel operations. The other team members who share the same work area have only read permissions and cannot perform any revision control operations mentioned above.

- Locks are shared; individuals can switch lock ownership.

Shared work area can be set up such that locks are shared and individuals can switch lock ownership. In this model, user1 can checked out a file with a lock (**co -lock**), user2 can edit the file, then switch the ownership of the lock to User2 and check in the object.

### **To set up a shared work area where locks can be applied on an individual basis:**

As system administrator, take the following steps:

1. Set up the work area so that members of a team have write permission into the same work area.

For example, on the work area directory, use the UNIX change mode command (`chmod -R 775`). This command sets the permissions to let the owner and the group read, write, and execute files but denies write permission for others.

2. Set each team member's user mask by adding a `umask` entry to each member's `.cshrc` file.

For example, to each user's `.cshrc` file add `umask 002`. This setting specifies that files be created with read and write permission for the owner and group but turns off write permission for others.

## To set up a shared work area where locks are shared:

As system or DesignSync administrator, take the following steps:

1. Start SyncAdmin and choose **Change 'Site' settings**.
2. Click the Client Triggers tab and create a client-side trigger. Define the trigger to:
  - Fire when a team member performs a checkout command (**co**)
  - Fire when a lock is present on the object (isLock 1)
  - Fire after the object is checked out (postObject)
  - Run a script that changes the object's permissions from the default (644, which is rw-r--r--) to 664 (rw-rw-r--)

When you enable the trigger, the next restart of a DesignSync client loads the trigger for the client. The trigger is then available to all users within the installation.

If User1 checks out an object with a lock, the trigger fires. The trigger runs the commands, which change the object's permissions. User2 then has UNIX permission to edit the object that User1 checked out.

3. Change the access control restriction on the **switchlocker** command to allow users access to the command.

The **switchlocker** command changes the current owner of a lock on an object. The default AccessControl files included with DesignSync deny access to this command for all users. However, as a system or DesignSync administrator, you can modify the AccessControl file to allow users to access to the command. For more information, see DesignSync Action Definitions.

For example, suppose you modify the AccessControl file to allow lock switching among team members. If User1 checks out the file with a lock, User2 can use the **switchlocker** command to change an object's lock owner from User1 to User2. User 2 can then check in the modified file.

See the ENOVIA Synchronicity Command Reference Help: switchlocker command for more information and examples of its use.

### Related Topics

Controlling Access to Your Local Work Area

## Moving a Work Area

There may be situations in which you want to relocate the project data of a work area to another UNIX file structure location.

**Note:** Moving a module work area is not recommended. To use a different module work area, check in any modified files or locked files using the existing work area, then populate the new work area. If there are unmanaged files in the original work area, you can copy or move them to the new location. For more information on creating a work area see, [Populating Your Work Area](#).

You may want to move an existing work area to a new work area, but you want to preserve the work area's association with the vault. Or you may want to move a work area, set up an association with another server/vault, and perhaps create new project data.

In either situation, you use the DesignSync **setvault** command after you relocate the work area.

### The Role of setvault in Moving a Work Area

The **setvault** command associates a single directory with a vault location. DesignSync stores this assignment as an entry in the file `../.SYNC/Contents`. For example, suppose you have the following directory structure:

```
db/  
  
db/pci/  
  
db/pci/rtl/  
  
db/pci/rtl/pci.v  
  
db/pci/rtl/pcicntl.v
```

If you run the **setvault** command on the `db/pci` directory, DesignSync stores the vault association information in the file `db/.SYNC/Contents`.

When files are passed to and from the vault, DesignSync stores the association of a file with a vault in the `../.SYNC/Contents` file. However, DesignSync does not associate a subdirectory with the vault unless you run the **setvault** command directly on that subdirectory. So in the example directory structure, the subdirectory `db/pci/rtl/` is not associated with the vault by a `../.SYNC/Contents` entry until you run the **setvault** command on the `db/pci/rtl/` subdirectory.

**Note:** If **setvault** is not run on a subdirectory, the subdirectory's vault association is inherited from its parent.

See the ENOVIA Synchronicity Command Reference Help: `setvault` command for information on the syntax and options of this command.

## Moving a Work Area Yet Preserving Its Vault Association

Because the `.SYNC/Contents` file for your work area resides at the level above your work area in the hierarchy, the file does not get moved with the work area directories and files. To move a work area, yet preserve its association with the vault:

- Move the top-level work area directory. (Only the top-level directory needs to be moved; the rest of the hierarchy follows the top-level directory, thus preserving the current structure of the work area.)
- Recreate the top-level `.SYNC/Contents` entry by associating the relocated work area with the vault.

**Caution:** To avoid corrupting metadata, you should never edit `.SYNC/` directory contents by hand, .

For example, suppose you move all files and directories of our example work area from the location `db/pci` to a new UNIX file structure location, `projects/pci`. The vault association for the `pci` work area, however, does not get moved, because it is stored in the file `db/.SYNC/Contents`, which resides at a level higher than the `pci/` directory. Without this file in the new location, the `pci/` directory has no server/vault association. In addition, all the subdirectories have no association with the vault, since they inherit it from the `pci/` directory.

The only exception is where **setvault** has been run on a subdirectory to explicitly set up an association between that subdirectory and the vault.

To associate the work area in the new location with the same vault as the old work area:

1. Change directory to the work area root directory (the `pci/` directory in our example).
2. From this directory, execute the DesignSync command **setvault** and specify the URL of the vault. For example:

```
dss> setvault  
sync://adagio.myco.com:2647/Projects/Sportster .
```

DesignSync creates the `.SYNC/Contents` file at the level above the work area work directory (`projects/.SYNC/Contents` in our example).

## Moving a Work Area and Creating New Project Data

When you move a work area, the files in the work area are still associated with the old server/vault location. To associate the relocated work area with a different vault, use the DesignSync **setvault** command with the **-recursive** option. Using the **-recursive** option causes the `setvault` operation to modify the server/vault association value for all files in the hierarchy as well as for the root directory. Then when the files are used in a transfer



to or from the new server/vault location, they have their association set to follow that chosen at the root directory.

To move a work area and make a new vault association:

1. Move the top-level work area directory. (Only the top-level directory needs to be moved; the rest of the hierarchy follows the top-level directory, thus preserving the current structure of the work area.)
2. Change directory to the work area root directory (the `pci/` directory in our example).
3. Use the DesignSync **setvault** command with the **-recursive** option and specify the URL of the vault. For example:

```
dss> setvault -rec
sync://adagio.myco.com:2647/Projects/Sportster .
```

DesignSync associates the relocated work area with the new server/vault. Now you can check in new project data.

### Related Topics

[Metadata Overview](#)

[Setting Up a Shared Workspace](#)

## UNIX Permissions of Work Areas and Vaults

When populating a workspace, all subfolders inherit the parent folder's UNIX permissions.

When an object is fetched `-lock`, the object is always writable by the file's UNIX owner. All other workspace UNIX permissions, of both locked and local file copies, depend on:

- Your umask setting
- Whether the workspace is shared
- The default preference for how local (unlocked) copies of files are fetched into the workspace
- The UNIX permissions on the vault file
- Cache and mirror UNIX permissions

These factors are explained below.

### How your umask Affects Local Permissions

Your UNIX umask controls whether read and execute permissions for UNIX group and UNIX everyone are set, when a file is fetched into your workspace. The UNIX write bit is never set for UNIX group or UNIX everyone, regardless of your umask.

### **How Shared Workspaces Affect Local File Permissions**

Users' umask settings must allow others in their UNIX group to access shared data. The shared work area must also be created with the appropriate UNIX directory permissions. If users are to share DesignSync locks, additional set up steps are required, such as setting the UNIX write bit for the UNIX group. See *Setting Up a Shared Work Area* for details.

### **How the Default Fetch Preference Affects Fetched Copies**

If the DesignSync client's default fetch preference is to "fetch read-only", the file's UNIX owner cannot write to unlocked copies that are fetched. If the default fetch preference is not "fetch read-only", The file's UNIX owner will be able to write to unlocked copies that are fetched. For more information on how to modify the default behavior, see *DesignSync Data Manager Administrator's Guide: General Options*.

### **How UNIX Permissions on Vault Files are Managed**

UNIX permissions on vault files are always read-only. They are never writable, not even for the server owner. The UNIX read and execute permissions for UNIX owner, group and everyone are set every time a new file version is checked in. When a new file version is checked in, the UNIX read and execute permissions of the local workspace file being checked in are applied to the vault file. The UNIX read and execute permissions on a vault file reflect the UNIX read and execute permissions of the last version checked in, from the workspace in which that last version was checked in.

When a file is fetched from the vault, the workspace file's UNIX read and execute permissions are set to those of the vault. A user's umask can cause the fetched file's UNIX permissions to be further restricted, but a user's umask cannot cause more open UNIX permissions on the fetched file. For example, if a vault file does not have execute permissions for UNIX everyone, when the file is fetched, the workspace file will not have execute permissions for UNIX everyone, regardless of the user's umask.

### **How Cache and Mirror UNIX Permissions Affect Local File Permissions**

The default `-from local` performance optimization copies files from a LAN cache or mirror directory into your workspace, instead of fetching the file from the server. If the cache or mirror environment was not set up correctly, the local file copies fetched into the cache or mirror will have incorrect UNIX permissions. Consequently, if you use the `-from local` option to copy the cache file or mirror file into your workspace, the file will also have incorrect UNIX permissions. See *DesignSync Data Manager Administrator's Guide: Fetching Files from the Mirror or Cache* for details.

## Command Line Defaults System

The command line defaults system only pertains to the command line interface. Underlying commands that are automatically invoked by the DesignSync GUI's dialog boxes, reports, etc., do not use the command line defaults system. The **Save Settings** button in DesignSync GUI dialog boxes does not interact with the command line defaults system.

The DesignSync GUI's **Command Shell Window**, into which a user types commands, does use the command line defaults system. For an overview of the command line defaults system, see the ENOVIA Synchronicity Command Reference Help: command defaults command line topic.

### Related Topic

Command Shell Window

## Working with Scripts

### DesignSync Scripts

You can include DesignSync commands in scripts that you invoke from any of the DesignSync clients or from your operating system. The following sections focus on creating and running scripts containing DesignSync commands:

- Using DesignSync Commands in OS Shell Scripts
- Creating DesignSync Scripts
- Running Scripts
- Running a Script at Startup

You can also use DesignSync commands in Tcl scripts by using the stcl language, a combination of DesignSync, ProjectSync, and Tcl commands. If your scripts are Tcl scripts that include DesignSync or ProjectSync commands, see the ENOVIA Synchronicity stcl Programmer's Guide for guidance in developing, setting up, and running both client and server stcl scripts.

### Using DesignSync Commands in OS Shell Scripts

You can embed DesignSync and Tcl commands in operating-system (OS) shell scripts. For single commands, include the line as though you were invoking it directly from the OS shell. In the following example, the c-shell script called `lockNotify.csh` takes a directory and your user name as arguments, searches for all files locked by you, then sends you an email with the list of locked files:

## DesignSync Data Manager User's Guide

```
#!/bin/csh
set dir=$1
set user=$2

# Find files locked by the user in the specified directory
#
dssc ls -recursive -path -locked $dir | grep $user >
/tmp/$user.list

# Notify the user
#
mail -s "Reminder: You have locked files" $user <
/tmp/$user.list
```

Note that all the commands in this script are OS commands except for `ls`, which is the DesignSync `ls` command and must therefore be preceded by `dssc`. You could then run this script from your OS shell:

```
% lockNotify.csh . goss
```

If you have a block of DesignSync or Tcl commands that you want to include in your script, you can enter the DesignSync shell, issue several commands, then exit from the shell, as shown:

```
#!/bin/csh

# - Go to the local working directory
# - Specify which files to check-in and tag
# - Check in selected files from the local working directory
# to the shared cache. Links are created from
# the work area to the files in the cache.
# - Tag these files, which will be used by test engineers,
moving
# the tag from a previous version if necessary
dssc<<EOF
scd $HOME/synth
select hdl*
ci -comment "ready for testing" -share
tag -replace ready_for_testing
exit
EOF
```

### Related Topics

[DesignSync Command-Line Shells](#)

[Invoking a DesignSync Shell](#)

## Creating DesignSync Scripts

### Creating DesignSync Scripts

You can create scripts containing DesignSync commands in two ways:

- By hand, using any text editor. The following is an example of a script created by hand:

```
# Go to the local working directory
scd $HOME/ synth

#
# Specify which files to check-in and tag
#
select hdl*

#
# Check in selected files from the local working directory
# to the shared cache. This creates links from the working
# directory to the files in the cache.
#
ci -comment "ready for testing" -share
```

- ```
#
# Tag these files, which will be used by test engineers
# Move the tag from a previous version if necessary
#
tag -replace ready_for_testing

#
# Exit dssc
#
exit
```
- Using DesignSync's logging capability to create a log file of a DesignSync session that you can later run to reproduce the session. The DesignSync **log** command controls logging options, including the name and location of the log file, and whether just commands are logged or also command output (which is commented out in the log file). Project leaders can set up site-side logging preferences from the SyncAdmin tool.

If you want to also include Tcl commands in your scripts, see the ENOVIA Synchronicity stcl Programmer's Guide.

### Related Topics

DesignSync Command-Line Shells

Using DesignSync Commands in OS Shell Scripts

Running Scripts

ENOVIA Synchronicity Command Reference: log Command

## Running Scripts

You can run a DesignSync script in several ways:

- From any DesignSync shell, use the **run** command:

```
dss> run <filename>
```

If the extension of the file is **.tcl**, the script is run in stcl mode regardless of your current shell. Otherwise, the script is run as a dss script (no Tcl constructs allowed).

If you do not specify a path for the script file, DesignSync looks in the default log directory. By default, the default log directory is the directory from which you invoked DesignSync. You can change the default log directory by using either of the following DesignSync command-line commands:

```
dss> run -defaultdir <path>  
dss> log -defaultdir <path>
```

Note that project leaders can set up site-side logging preferences from the SyncAdmin tool.

- From an stcl or stclc shell, you can use the Tcl `source` command to run your script. Running the script with `source` or `run` has the same effect.

```
stcl> source <filename>
```

- From your OS shell, use stcl or stclc without the `-exp` option to run the script:

```
% stclc <filename>
```

- DesignSync can run a script automatically when you invoke a DesignSync client. See [Running a Script at Startup](#).

**Note:**

To run server-side scripts -- scripts that reside on and are executed by a SyncServer as opposed to a DesignSync client -- use the **rstcl** command.

**Related Topics**

DesignSync Command-Line Shells

Creating DesignSync Scripts

Running a Script at Startup

ENOVIA Synchronicity Command Reference Help: run Command

ENOVIA Synchronicity Command Reference Help: rstcl Command

ENOVIA Synchronicity stcl Programmer's Guide

**Running a Script at Startup**

You can have DesignSync run a startup script when you invoke any DesignSync client. In SyncAdmin, select **General =>Startup** to specify a startup script. The script can contain DesignSync and Tcl commands. By default, the script is named `dsinit.dss`, but you can choose any name and path for the script. DesignSync interprets the specified script file as a dss script unless the file has a `.tcl` extension, in which case DesignSync interprets it as an stcl script. If you do not specify a path for the script, DesignSync searches for the script in these directories in the following order:

1. `$$SYNC_USER_CFGDIR` (by default, resolves to `<HOME>/synchronicity` on UNIX and `%AppData%\Synchronicity` on Windows)
2. `$$SYNC_SITE_CUSTOM` (resolves to `<SYNC_CUSTOM_DIR>/site` by default)
3. `$$SYNC_ENT_CUSTOM` (resolves to `<SYNC_CUSTOM_DIR>/enterprise` by default)

DesignSync searches for the script in the `$$SYNC_SITE_CUSTOM` directory only if the `$$SYNC_USER_CFGDIR` directory does not contain the script.

Examples of tasks you might include in a startup script are:

- `scd` to a particular directory
- Perform an incremental populate (so you always have the latest versions of files)

Your system administrator or project leader can also prepare startup scripts for DesignSync to source at startup. In this way, your site or project team can share the same aliases. See Autoloading Tcl Procedures for more information.

### Related Topics

*DesignSync Data Manager Administrator's Guide: Startup Options*

DesignSync Command-Line Shells

Running Scripts

Creating DesignSync Scripts

*DesignSync Data Manager Administrator's Guide: Autoloading Tcl Procedures*

*DesignSync Data Manager Administrator's Guide: Using Environment Variables*

## Improving Efficiency Using Caches and Mirrors

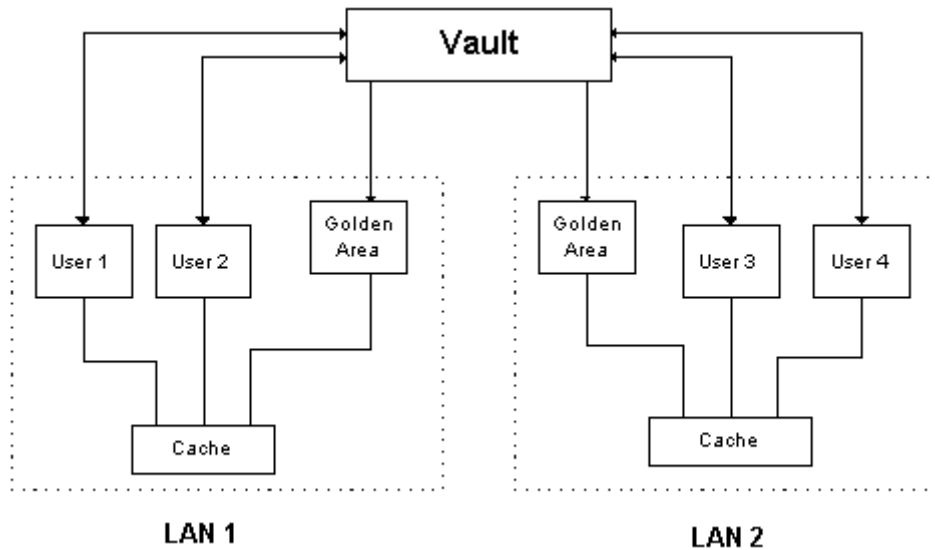
### Mirrors Versus Caches

Mirrors and caches are discussed in the *DesignSync Data Manager Administrator's Guide*, in the topic Mirrors Versus Caches.

### What is a File Cache?

DesignSync provides the ability for engineers located in widely dispersed locations to access design data stored in a central vault. In large projects, engineers within one location are typically grouped together on a local area network (LAN). In the illustration, users 1, 2, 3, and 4 all work on the same project, checking design data into and out of the vault. Users 1 and 2 are on a common LAN, as are users 3 and 4. At various milestones along the way, a golden configuration can be defined in the vault by assigning a tag (such as "Gold") to a collection of file versions that work together satisfactorily. Each LAN can periodically populate its golden area with the golden configuration.





A cache reduces the amount of file duplication among the users on a LAN. A cache comes into use, for example, when you issue a check out command with the **-share** option (populate, check-in, check-out, and cancel operations all have share options). The object you are checking out is put into the file cache, if it is not already there because of a previous check out. The object is not put into your own work area. Instead, a symbolic link or hard link to it is put into your working directory, visible through the UNIX **ls** command.

#### Notes:

- Objects in the cache have DesignSync-generated filenames. DesignSync manages the cached files, so you do not need to deal with these meaningless names. However, the links in your work area to the files in the cache use these meaningless names, for example:

```
$ ls -l top.v
```

```
top.v -> /home/tgoss/sync_cache/s64/s64a06945af4f7c9b-1.1-982002979-f216a783154b31cc
```

For more information about these cache link files, see [How DesignSync Manages Caches](#).

- If your team is using a cache methodology, your project leader may have defined a default fetch state of 'share'. This default fetch state eliminates the need for specifying the **-share** option each time you perform a revision-control operation.
- Because caching is implemented using links, only UNIX systems can take advantage of this capability.

#### Related Topics

### Why Use a File Cache?

*DesignSync Data Manager Administrator's Guide: Setting up a LAN Cache*

*DesignSync Data Manager Administrator's Guide: Mirrors Versus LAN Caches*

## Why Use a File Cache?

The purpose of using a cache is to save both disk space on the LAN and to save time for individual fetches of updated files.

A cache is useful when, for example, the members of a team populate their simulation directories. Each user first designates the server vault location that is to act as the source for the simulation data by using the **setvault** command. Then, when the first user issues a **populate -recursive -share** command, the vault folder hierarchy is copied to the cache directory and the links are created in the user's simulation directory that point to the corresponding cache files. The cache file names are encoded in a way that allows the client to decrypt the source vault location from which the file originated. For populate operations that are subsequently executed by other users, the client does not copy any file to the cache that is already present. It must only establish a link to the existing file.

After a user's simulation directory is populated with links to the cache, any files that need to be individually modified by the user can be checked out with the **-lock** option. This operation will check out a copy of the file from the server vault into the user's simulation directory and then delete the link to the cache copy. Now the user can make modifications to the real file to diverge the simulation behavior from that produced by the standard vault simulation. However, the bulk of the directory tree is still shared in a read-only mode with the other project members. The cache copy of a file is removed automatically when the final user removes the last link.

### Related Topics

What is a LAN Cache?

## What is a Module Cache?

A module cache (mcache) can be thought of as a shared workspace. The DesignSync administrator or Project manager can create module caches to contain the common tools, modules, resource libraries, SITaR baselines, etc. required by the entire development team. By populating a module cache link into their workspaces, they populate a single DesignSync object, rather than the full contents of the module.

For UNIX workspaces, which link to the module in the mcache, this reduces populate time, load on the SyncServer, and saves disk space. For Windows workspaces, which

do not link to a module cache, but can copy from the cache, it reduces load on the SyncServer during the populate and updates.

A project leader fetches modules into a module cache, preferably in "share" mode, to utilize DesignSync's file caching. When users populate module data, they can specify, using the Module Cache Mode option, whether to link to modules in the module cache, copy the files from the module cache to their local system, or fetch modules from the server. When the module cache is populated, it is assigned an instance name. This allows the team leader to maintain the module cache as any other module. The default module cache path, and the default module cache mode, can be defined in the registry by the team leader or by an individual user.

When users populate module cache links, the symbolic link created is identified as a "Link to mcache" object type and assigned a module instance to allow the user to maintain the mcache link as any other module.

Module caches that contain non-legacy modules must designate the topmost directory of the module cache as a workspace root directory. Use the 'setroot' command to define the topmost directory as a root and then proceed to populate the DesignSync modules into the module cache.

Legacy modules can be linked to, or copied from, a module cache. The module cache mode of "copy" only applies to legacy modules. Attempting to copy a non-legacy module from a module cache fetches the module from the server.

#### **Related topics**

Using a Module Cache

Procedure for Creating a Module Cache Link

*DesignSync Data Manager Administrator's Guide*: Procedure for setting up a Module Cache

*DesignSync Data Manager Administrator's Guide*: Module Options

## **Mirroring Overview**

A mirror exactly mimics the data set defined for your project vault. Mirrors provide an easy way for multiple users to point to the file versions that comprise their project's data. The file versions in the mirror belong to the configuration defined by the project lead. For example, the configuration could be the Latest version of files on the main Trunk branch. A mirror for a development branch may be defined to always contain the file versions on that branch with a specific tag. When the file versions comprising the configuration change, for example, if Latest versions are being mirrored and a new version of a file is checked into the vault, the mirror directory is automatically updated

with the new version. Without mirroring, users would need to frequently update their work areas using the `populate` command to reflect the project's current data set. You can find where vault data is being mirrored, and the status of those mirrors. (See the Related Topics below.)

The `setmirror` command associates a workspace with a mirror directory. A mirror will always have accurate metadata because any action that writes to a mirror directory updates the local metadata in the mirror directory. When you use the `setmirror` command to associate a mirror directory, checking in an object will:

- create the new version in the vault,
- update the file in the associated mirror associated, and then
- update the metadata.

Mirror can be updated and administered automatically. See the section Administering Mirrors for details. As of Version 4.2, the legacy Remote Mirror Assurance package is no longer supported.

### Mirror Attributes

- All actions that write to a mirror directory will update the local metadata in the mirror directory. When looking at a workspace that has objects in the mirror state, a combination of the workspace's and the mirror directory's local metadata will be used to determine the correct version of the objects. This allows you to use the `ls` or `url` command on objects in the workspace to show the correct state of the object.
- Mirrors support all defined configurations.
- When a check-in occurs from a client, it creates a new version in the vault, returns control back to the client, and the client writes the object into the mirror and updates the local metadata in the mirror directory.
- A mirror write through will occur for all fetch states. Regardless of the fetch state, if a mirror write through is done, then the metadata is updated to reflect what was written to the mirror directory.
- No other commands, with or without the `-mirror` option, write through to the mirror. Commands like `populate -mirror` and `cancel -mirror` do not write to the mirror directory. However, the `co -mirror` command writes through to the mirror directory if the correct up-to-date version is not already in the mirror. Most commands only create links from a workspace to the files in the mirror directory.

As a DesignSync administrator, you can:

- Set up a mirror directory and navigate through this mirror knowing that everything is being kept up-to-date.

- Set up your environment (from that LAN where the check-ins occur) to write through to your mirror when checking a new version into the server. You do not have to wait for the mirror update process to update the mirror.

## Restrictions

- Because mirroring is implemented with UNIX links, mirrors are not supported on Windows platforms.
- The mirror directory and the users accessing it must be on the same LAN.
- Only one process can write to the mirror subdirectory at a time. The system ensures that when you check in a new version, there will be a lock on the mirror subdirectory. The lock is held for the duration of the client check-in from the workspace subdirectory. The system will display a "waiting on metadata lock" message while the system processes the workspace. This may cause a delay if someone is checking in a large amount of objects or large files.
- Mirrors for modules cannot be linked to from a workspace. A module cache should be used instead.

## Related Topics

### General Mirror Topics:

[Mirrors Versus Caches](#)

[Using a Mirror](#)

[ENOVIA Synchronicity Command Reference: mirror wheremirrored](#)

### Mirror Administration Topics:

[Administering Mirrors](#)

[Finding Mirrored Data](#)

## Locking, Branching, and Merging

### What Is Merging?

Merging combines changes made on two branches, or within a branch. Branching without merging has limited usability – two developments streams diverge forever and can never be reconciled.

There are several important notions related to the general concept of merging that are equally applicable to file merging and module merging:

Merge Conflicts

Two-Way Merge

Three-Way Merge

Merge Edges

Module merging has additional factors to consider because instead of merging specific files, you are merging a set of changes. For more information on module merging, see [Merging Module Data](#).

### Related Topics

[Merging Module Data](#)

[Using the Merging Work Style](#)

[Parallel \(Multi-Branch\) Development](#)

[Merge Conflicts](#)

[Two-Way Merge](#)

[Three-Way Merge](#)

[Merge Edges](#)

## Locking and Merging Work Styles

### Locking and Merging Work Styles

DesignSync supports the two common design-management (DM) work styles: locking and merging (non-locking).

- **Locking Style**

With the locking style, you always check out with a lock an object that you plan to edit. DesignSync locks the branch associated with the version you are checking out, prohibiting other team members from creating new versions on that branch. You, the holder of the lock, reserve the right to create the next version on that branch. Other team members can fetch (check out without a lock) the object, but no one else should make changes while you hold the lock.

For example, Barbara checks out with a lock version 1.3 (the Latest version) of a file. Jack then tries to check out with a lock the same file, but the checkout fails and DesignSync reports that the file is already locked. Jack contacts Barbara to let her know that he needs to edit the file. Barbara finishes her changes and checks the file in, creating version 1.4. Jack can now check out version 1.4 with a lock, make his changes, then check in the file to create version 1.5.

Note that in multibranch environments, each branch of an object can be independently locked and unlocked. Therefore, different team members can modify the same object on different branches even when using the locking work style.

- **Merging Style**

With the merging style, more than one person can fetch the same object with the intention of editing the object. The first person to check the object back in creates the next version. The other person must first merge the changes from this new Latest version into his or her local copy, manually resolve any merge conflicts, then check in the merged object.

For example, Barbara and Jack both check out version 1.3 (the Latest version) of a file. Jack checks in his changes first, creating version 1.4. To check in her changes, Barbara must first merge the new Latest version (1.4) into the modified version in her work area. She manually resolves any conflicts between her changes and Jack's changes. She then checks the merged version in, creating version 1.5.

Note that DesignSync also supports merging across branches. See [Parallel \(Multi-Branch\) Development](#) for information.

Which work style is appropriate for your team depends on various factors:

- **Comfort with design management**

Teams new to design management often use the locking style. The locking style avoids merges, which can be intimidating to novice users. As projects get bigger and teams gain more experience, teams tend to shift to the merging style to facilitate extensive sharing of files.

- **Frequency of sharing objects**

Cases where there is typically one team member modifying a given object at a time are well suited to the locking style. If multiple users commonly want to simultaneously modify a given object, then the merging style is appropriate.

- **Data format**

- Merges require that design data be ASCII (text), as opposed to binary. If your team is managing binary data, you must use the locking work style, at least for the binary data.

DesignSync also lets you mix the two work styles, although doing so is not generally recommended. For example, your team is using the merging style, but you check out with a lock an object that you want no one else to modify.

### Related Topics

Using the Locking Work Style

Using the Merging Work Style

## Using the Locking Work Style

The following operations define the locking work style:

- To check out an object for editing, lock the object:
  - From the **Check Out** dialog box, select **Locked copies**.
  - From the **Populate** dialog box, select **Locked copies**
  - When using the **co** or **populate** commands, use the `-lock` option.
- To check in an object while continuing making changes, retain the lock:
  - From the **Check In** dialog box, select **Locked copies**.
  - When using the **ci** command, use the `-lock` option.
- To cancel your check out when you have an object locked but have decided not to make changes, select **Revision Control => Cancel Checkout** or use the **cancel** command. Note that to control the state of the object (local copy, link to the cache, link to the mirror, reference) after the checkout, you must use the `cancel` command; canceling from the graphical interface always uses your default fetch state if defined, otherwise leaves a local copy.
- To remove a lock held by another user, select **Revision Control => Unlock** or use the **unlock** command. Note that removing another user's lock is not a typical operation and one that is often access controlled. Consult your team leader.
- If you have already made changes to an object, you can still obtain a lock for the object by using the **co** command with the `-lock` option. DesignSync lets you acquire the lock without fetching the object and overwriting your work. If another user has checked in changes to the object or has acquired a lock on the object, DesignSync does not let you lock the object. You can also obtain the lock by using the **Check Out** dialog box and selecting **Locked copies**.
- If you regularly regenerate a large file but you want to lock the file before regenerating, you can save time by using **co** with the `-lock` and `-reference` options. You save time because DesignSync creates a reference to the file instead of fetching the previously generated file. You can also obtain a locked reference by using the **Check Out** dialog box and selecting **Locked references**.
- To find out where an object is locked, you can use the `showlocks` command.



Your project leader can use access controls to enforce a locking style by limiting check-in operations to locked objects. See the ENOVIA Synchronicity Access Control Guide: Access Control Scripting for a sample script to enforce a locking model. The ability to perform a merge can also be restricted; see the access control definition.

#### Related Topics

Locking and Merging Work Styles

Using the Merging Work Style

ENOVIA Synchronicity Command Reference: co

ENOVIA Synchronicity Command Reference: populate

ENOVIA Synchronicity Command Reference: ci

ENOVIA Synchronicity Command Reference: cancel

ENOVIA Synchronicity Command Reference: unlock

## Merge Conflicts

Whenever you merge, whether the merge is between two versions on the same branch (see Using the Merging Work Style) or between branches (see Parallel (Multi-Branch) Development), there may be merge conflicts. Merge conflicts can also arise when changes in the two branches being merged are incompatible, such as when a file is renamed to different locations on the two branches. Merge conflicts occur when different changes were made to the same region of the two versions that are being merged. DesignSync cannot automatically determine which changes are the correct ones; you must resolve the conflicts manually.

When working with modules, it is not necessary to resolve all conflicts before creating the next module version. You can check in, or perform other operations that create new module versions, as long as the objects being operated on are not the ones that are in conflict.

### Note:

DesignSync records "merge edges" – information about what versions participated in the merge -- with the new version resulting from the merge. DesignSync uses merge edges in future calculations of closest common ancestors instead of always going back to the original ancestor. This capability relieves you from having to resolve the same merge conflicts during future merges. See Merge Edges for more information.

## Resolving Merge Conflicts

## DesignSync Data Manager User's Guide

A conflict is presented in a textual or graphical format as two options. You resolve the conflicts either one or at a time, or collectively by selecting the appropriate version.

DesignSync alerts you to conflicts during the merge. Conflicts are identified in the Changed Object Browser, by the Status field of the List View and from the **Is** command. You can also use the **url inconflict** command.

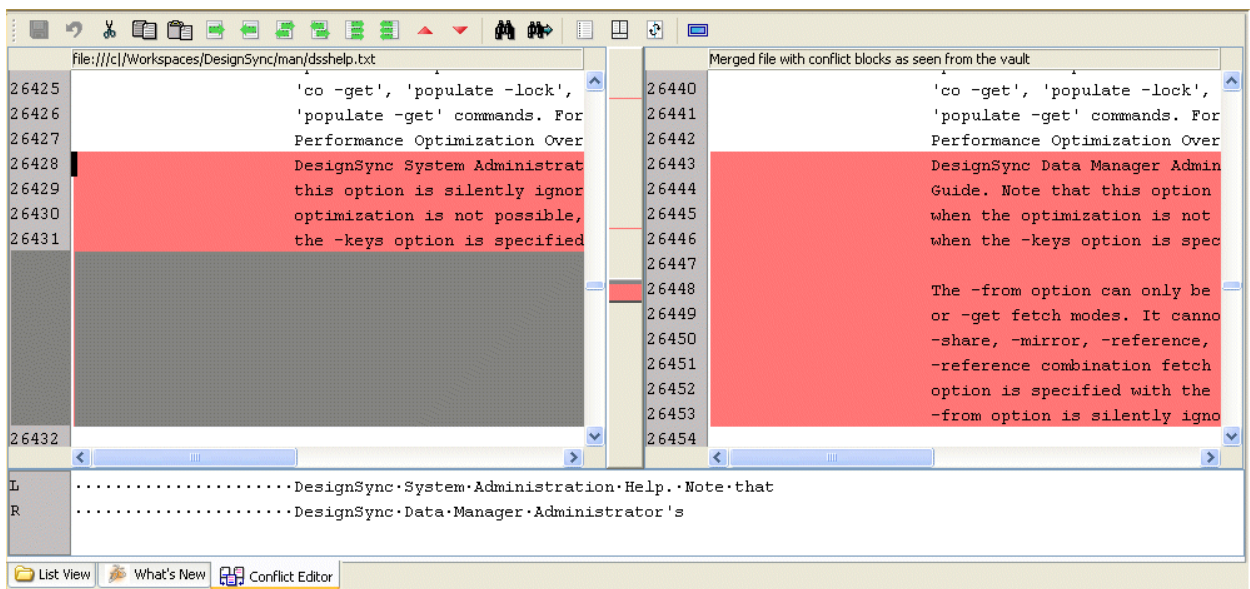
When you merge a file, the conflicts are indicated in the file text with a conflict delimiter (exactly 7 less-than, greater-than, or equal signs starting in column 1) and the version number to indicate what text is present in each version:

```
<<<<<<< versionID
Lines from Latest version (same-branch merge) or overlaid
version
Lines from locally modified version
>>>>>>>
```

DesignSync considers the conflicts resolved when the file no longer contains any of the conflict delimiters.

To invoke the Conflict Editor, select the Resolve Conflicts action from the context menu of an object that's identified as **In Conflict** by the Changed Object Browser. Or select an in conflict file and run **Tools => Resolve Conflicts**.

In the Merge Conflict Editor, the conflicts are indicated by highlighted text.



DesignSync considers the conflicts resolved when the file no longer contains any of the conflict delimiters. When you resolve a conflict, the Merge Conflict Editor removes the

conflict indicators for you and changes the highlight color to the resolved conflict color, light pink by default.

**Note:** The highlighted text uses the color specified for conflict resolution with SyncAdmin, in the **For multi-window Diff viewers/editors** section.

#### Related Topics

Two-Way Merge

Three-Way Merge

Conflict Handling

Identifying Changed Objects

Locking and Merging Work Styles

Merge Conflict Editor

Using the Merging Work Style

Parallel (Multi-Branch) Development

ENOVIA Synchronicity Command Reference: Is

ENOVIA Synchronicity Command Reference: url inconflict

ENOVIA Synchronicity Command Reference: populate

## Selecting Versions and Branches

### Selecting Versions and Branches

Some revision-control operations, such as checkout and populate, operate on versions of design objects (files or collections). Other operations, such as checkin and retire, operate on branches. One operation, tagging, can operate on both versions and branches. How DesignSync determines what version or branch to operate on varies by operation but can include the following:

- The current branch or version of the object in your work area

For operations such as retiring, unlocking, and tagging design objects, the default behavior is to operate on the current version or branch of objects in your work area.

- The branch or version you explicitly specify using a selector or selector list

Many command-line commands have a **-branch** or **-version** option so that you can specify the branch or version on which to operate. The DesignSync graphical user interface (GUI) lets you specify a branch or version using **Version** and **Branch** fields where appropriate. For example, the **Check Out**, and **Populate** and **Tag** dialog boxes have **Version** fields. The **Check In**, **Unlock**, **Retire**, and **Tag** dialog boxes have **Branch** fields.

- The branch or version identified by the object's persistent selector list
- Every object has a persistent selector - a selector or selector list that is stored in local metadata (or inherited from the parent folder) and is used by some commands in the absence of an explicit branch or version selector list. By default, the persistent selector list is **Trunk** (shorthand for Trunk:Latest), which is the default tag name for branch 1. Check-in, check-out, populate, and import operations obey the persistent selector.

### Note:

Using CONFIG statements in `sync_project.txt` files, you can map a configuration to a single selector or a selector list. See Using Vault References for Design Reuse for more information.

### Specifying Branches and Versions Using Fields or Vault URLs

Some DesignSync commands (for example, **mkbranch**, **retire**, **tag**, and **unlock**) provide two ways to specify a particular version or branch for the command to operate on:

- Use the **Branch** or **Version** fields (or **-branch** or **-version** command-line options). For example:

```
stcl> tag -recursive alpha -version baseline ASIC/top/alu
```

DesignSync tags all objects that have the `baseline` version tag in the `alu` folder and all its subfolders.

```
stcl> unlock -branch RelA:Latest -recursive  
sync://apollo:2647/Projects/Sportster/code
```

DesignSync unlocks branch `RelA:Latest` for all objects in the `code` folder and all its subfolders.

- Specify a branch or version selector with the object's URL.

```
dss> retire
"sync://apollo:2647/Projects/Sportster/top/top.v;Main:Latest"
```

DesignSync retires the version of `top.v` that has the branch selector `Main:Latest`. **Note:** Enclose the vault URL between quotation marks (") in the `stcl` shell if the string contains a semicolon (;).

If you enter a value in the **Branch** or **Version** field (or use a **-branch** or **-version** option) and you specify a vault object URL that includes a branch or version selector, the vault object URL selector takes precedence. For example, suppose you enter:

```
stcl> tag -version alpha beta
"sync://apollo:2647/Projects/ASIC/top/top.v;1.2"
```

DesignSync ignores the `alpha` version selector and tags version `1.2` of `top.v`.

#### Related Topics

[What Are Selectors?](#)

[What Are Selector Lists?](#)

[What Are Persistent Selector Lists](#)

[Selector Formats](#)

## What Are Selectors?

A selector is an expression that identifies a branch and version of a managed object. For example, the version selector `'gold'`, the branch selector `'Rel2:Latest'`, the version number `'1.4'`, and the reserved keyword `'Latest'` are all selectors.

### Static Selectors Versus Dynamic Selectors

Static selectors denote a set of objects whose contents are fixed. These fixed objects might constitute a 'release' of the group of objects. Static selectors include version selectors such as `gold` and branch selectors with fixed versions, such as `Rel2:gold`. The objects denoted by a static selector do not change with subsequent checkins. Changes made to module workspaces populated with a static selector cannot be checked in.

Dynamic selectors denote a set of objects whose contents are not fixed. A branch selector such as `Rel2:Latest` is a dynamic selector because the objects denoted by the selector change; a new `Latest` version is created on the `Rel2` branch with each subsequent checkin.

### How Branch and Version Selectors Are Resolved

Branch tags and version tags share the same name space. To distinguish version selectors from branch selectors, you append `:<versiontag>` to the branch name; for example, `Gold:Latest` is a valid branch selector. You can leave off the `Latest` keyword as shorthand; for example, `Gold:` is equivalent to `Gold:Latest`. The selector `Trunk` is also a valid branch selector. `Trunk` is a shorthand selector for `Trunk:Latest`.

You cannot assign the same tag name to both a version and a branch of the same object. For example, a file called `top.v` cannot have both a version tagged `Gold` and a branch tagged `Gold`. However, `top.v` can have a version tagged `Gold` while another file, `alu.v`, can have a branch tagged `Gold`.

Consider adopting a consistent naming convention for branch and version tags to reduce confusion. For example, you might have a policy that branch tags always begin with an initial uppercase letter (`Rel2.1`, for example) whereas version tags do not (`gold`, for example).

If the selector identifies a version, DesignSync resolves the selector to both the object's version number and branch number. For example, if version 1.2.1.3 is tagged `Gold`, DesignSync resolves `Gold` as both version 1.2.1.3 and branch 1.2.1. A version selector only resolves if the object has a version tag of the same name; it does not resolve if the tag is a branch tag.

If the selector identifies a branch, DesignSync resolves the selector to both that branch and the `Latest` version on that branch. If branch 1.2.4 has branch tag `Rel2`, DesignSync resolves `Rel2:Latest` as both branch 1.2.4 and the `Latest` version on that branch (say, 1.2.4.5). This behavior is important because some commands (such as `Check Out`) operate on a version, some (such as `Check In`) operate on a branch, and others (such as `Tag`) operate on either a version or branch. If the tag cannot be resolved as a branch, DesignSync searches for a version of the same name, determines which branch the version is on, and resolves to the `Latest` version on that branch. For example, suppose an object, `netlist.txt`, has a version tagged `beta` on its 1.2.4 branch. If the selector is `beta`, DesignSync first searches for a `beta` branch. Finding no `beta` branch, DesignSync searches for a `beta` version. DesignSync finds the `beta` version, determines its branch, 1.2.4, and resolves to the `Latest` version on the 1.2.4 branch.

A selector can also specify both a branch and a version, for example `Rel2:gold`. This selector resolves if there is a branch `Rel2` and if a version tagged `gold` exists on the `Rel2` branch.

A selector might not match any branch or version of a given object. For example, a file may not have a branch or version tagged `Gold`. Because selectors can fail, it is common to specify selector lists.

**Note: To resolve a selector, DesignSync does not search above the root of a workspace where a setvault has been applied. Thus, if a folder has no selector or persistent selector set, DesignSync searches up the hierarchy only as far as the first folder that has a vault association.**

#### Related Topics

Selecting Versions and Branches

What Are Selector Lists?

What Are Persistent Selector Lists

Selector Formats

## What Are Selector Lists?

You can combine selectors into a selector list, a comma-separated list of selectors. Selector lists are processed differently for module workspaces than they are for files-based workspaces. This topic discusses how files-based selector lists are processed.

Selector lists for files-based workspaces resolve to a single selector. Selector lists for module-based workspace combine selectors to create a composite, or blended workspace. For more information on module-based selector list processing, see [Module Member Tags](#).

No white space between items is allowed. Examples of selector lists are:

```
gold,silver,bronze,Trunk:Latest
```

```
auto(Test),Main:Latest
```

```
Dev2.1:Latest,Rel2.1:Latest,Trunk
```

Selector lists are used by commands that fetch objects (`co`, `populate`, and `import`). They provide a search order for identifying and retrieving versions. DesignSync operates on each element of a selector list in the same way it operates on an individual selector. If the first selector in the list does not resolve to a version, then DesignSync looks at the next selector in the list. The first matching version is used. The command fails if none of the selectors in the list resolves to a version. In the case of tags, DesignSync first looks for a branch with the specified tag, and if found, resolves to the Latest version on that branch. Otherwise, DesignSync looks for a version with that tag.

For example, if you want to populate with "the most stable configuration", you might define your selector list as `green, yellow, red, Trunk`, where your team's development methodology is to use version tags of `green`, `yellow`, and `red` to indicate decreasing levels of stability. With this selector list, DesignSync retrieves the first of the following versions it locates:

1. The version tagged `green`.
2. The version tagged `yellow`.
3. The version tagged `red`.
4. The `Latest` version on the branch tagged `Trunk`. `Trunk` (shorthand for `Trunk:Latest`) is the default tag name for branch 1.

If your team is using a branching methodology and you want to populate with "the most stable configuration", you might define your selector list instead as `Green:Latest, Yellow:Latest, Red:Latest, Trunk`. With this selector list, DesignSync retrieves the first of the following versions it locates:

1. The `Latest` version on the branch tagged `Green`.
2. The `Latest` version on the same branch as the version tagged `Green`.
3. The `Latest` version on the branch tagged `Yellow`.
4. The `Latest` version on the same branch as the version tagged `Yellow`.
5. The `Latest` version on the branch tagged `Red`.
6. The `Latest` version on the same branch as the version tagged `Red`.
7. The `Latest` version on the branch tagged `Trunk`.

Selector lists are a powerful mechanism, but can also add complexity. To avoid accidental checkins to unintended branches when you have complicated selector lists, the `'ci'` and `'co -lock'` commands consider only the first selector in a selector list. If the first selector resolves to a branch, the operation continues, otherwise the operation fails.

**Note: Using `CONFIG` statements in `sync_project.txt` files, you can map a configuration to a single selector or a selector list. See [Using Vault References for Design Reuse](#) for information.**

### Related Topics

[Selecting Versions and Branches](#)

[What Are Selectors?](#)

[What Are Persistent Selector Lists](#)

[Selector Formats](#)

## What Are Persistent Selector Lists



Some operations (check in, check out, populate, and import) support persistent selector lists. Persistent selector lists specify what branch or version a command operates on in the absence of an explicit branch or version selector list (**Version** or **Branch** fields in the graphical interface, **-version** or **-branch** command-line options). Commands that do not obey the persistent selector list typically operate on the current version or current branch of the object in your work area.

You explicitly set the persistent selector list, typically on an entire work area, using the **setselector** command. You can also set the selector at the same time you set the vault with the **setvault** command. DesignSync also sets the persistent selector list in the following cases:

- When populating a module with the **-version** option, DesignSync sets the persistent selector to specified version.
- When populating a configuration-mapped folder. This behavior is a performance optimization. See the **populate** help topic for details.
- When populating or checking out with the **-overlay** option and an object exists on the overlay branch but not on the branch being overlaid (the work area branch). DesignSync may augment the object's persistent selector list with the **Auto()** selector so that the object is automatically branched when checked in. See the **-overlay** option to the **populate** or **co** command for details.

A persistent selector list is stored in an object's local metadata, or it is inherited from its parent folder. If a persistent selector list has not been defined, the default is "Trunk".

#### Notes:

- Selector lists are not supported if you map configurations using REFERENCES in `sync_project.txt` files.
- Selector lists are processed differently for module workspaces than files based workspaces. for more information, see [What Are Selector Lists?](#).

#### Related Topics

[Selecting Versions and Branches](#)

[What Are Selectors?](#)

[What Are Selector Lists?](#)

[Selector Formats](#)

[ENOVIA Synchronicity Command Reference: setselector](#)

[ENOVIA Synchronicity Command Reference: setvault](#)

[ENOVIA Synchronicity Command Reference: populate](#)

ENOVIA Synchronicity Command Reference: co

**Selector Formats**

All DesignSync revision control operations use selectors to determine what version of an object to operate on. The selector can specify the branch of the object, the version of the object on the branch, or both.

Additionally, each workspace has a persistent selector that determines which version, by default, is populated into the workspace. You set the persistent selector for the workspace during the populate of your work area, or by explicitly setting the persistent selector.

There are two types of selectors: static and dynamic.

- Static selectors always resolve to a fixed version and the version they resolve to does not change. Changes to objects in static selector workspaces cannot be checked in.
- A dynamic selector refers to a version that can change. Changes to objects in dynamic selector workspaces can be checked in.

A selector can have one of several formats:

| Selector Type | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|---------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <number>      | <p>A branch or version number. Branch and version numbers are also known as "dot numerics". Using branch or version numbers as selectors is typically less convenient than using tags or date-based selectors.</p> <p>A version &lt;number&gt; selector is a static selector; the objects denoted by the version &lt;number&gt; selector are fixed. A branch &lt;number&gt; selector is a dynamic selector; the objects denoted by the branch &lt;number&gt; selector change upon subsequent checkins.</p> <p><b>Examples</b></p> <p>Version &lt;number&gt; selectors: 1.1, 1.3.2.3</p> <p>Branch &lt;number&gt; selectors: 1, 1.3.4, 1.1.1</p> |

`<versiontag>`

A version selector. If you specify a version selector, DesignSync resolves the selector to both the object's version number and branch number. For more details, see [What Are Selectors: How Branch and Version Selectors Are Resolved](#).

A `<versiontag>` selector is a static selector; the objects denoted by the `<versiontag>` selector are fixed.

A given tag name might be applied to a branch or to a version (but never both at the same time for the same object).

Branch selectors use the syntax `<branchtag>:<versiontag>`, for example, `Rel2:Latest`, to differentiate them from version selectors.

If you specify a version selector during the check-in of a new object, the object is created, by default, on the Trunk branch.

If you instead intend to check the object into a different branch, be sure to specify a branch selector rather than a version selector.

### Examples

gold

alpha

`<branchtag>:Latest`

A branch selector that specifies the most recent version on the branch. A given tag name might be applied to a branch or to a version (but never both at the same time for the same object). To specify a branch selector, append `:<versiontag>`, in this case, `:Latest`, to the branch tag name, for example, `Rel2.1:Latest`. You can leave off the `Latest` keyword as shorthand. For example, `Rel2.1:` is equivalent to `Rel2.1:Latest`.

A `<branchtag>:Latest` (or `<branchtag>:`) selector is a dynamic selector; the objects denoted by this selector change upon subsequent checkins.

If `<branchtag>` cannot be resolved as a branch tag, DesignSync searches for a version tag of that name and resolves to the Latest version on that version's branch. For more details, see [What Are Selectors: How Branch and Version Selectors Are Resolved](#).

The tag `Trunk` (shorthand for `Trunk:Latest`) has special significance; it is the default tag name for branch 1.

See [Using Latest and Date\(\) Selectors](#) for more details.

### Examples

`Trunk` - Branch 1 default (shorthand for `Trunk:Latest`)

`Rel2.1:Latest` - Branch selector of most recent Rel2.1 version

`Rel2.1:` - Shorthand for `Rel2.1:Latest`

`<branchtag>:<versiontag>`

A specific version on a specific branch. The `<branchtag>` and `<versiontag>` values are themselves selectors. This format is often used with `Date()` and `Latest` selectors to identify a version on a particular branch.

To specify a specific branch and version, the selector must contain both the branch and version. `:<versiontag>` is illegal. `<branchtag>:` resolves to the `Latest` version on the specified branch.

A selector such as `Trunk:gold` is valid and indicates a version tagged `gold` only if it is on a branch called `Trunk`; otherwise, the selector fails.

A selector of the form `Gold:Latest` looks for a branch tagged `Gold`, and if found, fetches the `Latest` version on that branch. If a `Gold` branch is not found, `DesignSync` looks for a version tagged `Gold`, and if found, retrieves the `Latest` version on the branch that the `Gold` version is on.

Selectors of the form

`Gold:Date(<date>)` behave similarly. See [Using Latest and Date\(\) Selectors](#) for more details.

Unlike the dynamic

`<branchtag>:Latest` selector, a `<branchtag>:<versiontag>` selector is a static selector; the objects denoted by the `<branchtag>:<versiontag>` selector are fixed.

### Examples

`Rel2:alpha`

`Trunk:Date(yesterday)`

`<branchtag>:Date (<date>)`

The most recent version on the specified branch that was created on or before the specified date. The `Date` keyword is case insensitive.

A `<branchtag>:Date (<date>)` selector is a static selector; the versions closest to the date do not change.

**Note:** If the `Date` selector resolves to a date in the future, the selector is equivalent to `Latest` and the selector is dynamic until the future date is reached, at which point the version becomes fixed and the selector is static.

You specify the `Date` selector as follows:

`<branchtag>:Date (<date>)` where `<branchtag>` is a branch tag. If `<branchtag>` cannot be resolved as a branch tag, DesignSync searches for a version tag of that name and resolves to the most recent version created on or before the specified date on that version's branch.

See the `Date Formats` for details on how to specify dates. See `Using Latest and Date() Selectors` for more information about specifying date selectors.

### Examples

`Trunk:Date (yesterday)` - Resolves to the last version checked in yesterday on the `Trunk` branch

`gold:date (4/11/00)` - If no branch is named `gold`, but there is a version selector `gold`, DesignSync resolves this selector to the last version checked in on or before 4/11/00 on the branch containing the `gold` version

`Rel2:Date (today)` - Resolves to the

last version checked in today on the Rel12  
branch

`VaultDate (<date>)`

The most recent version on any branch that was created on or before the specified date. The `VaultDate` keyword is case insensitive. Like the `Date` selector, the `VaultDate` specification can accept a branch tag, in the format:

```
<branchtag>:VaultDate (<date>)
```

where `<branchtag>`: is optional.

See [Date Formats](#) for details on how to specify dates.

A `VaultDate (<date>)` selector is typically a static selector; the objects denoted by this selector, once determined by the date, do not change.

**Note:** If the `VaultDate` selector resolves to a date in the future, the selector is equivalent to `Latest` and the selector is dynamic until the future date is reached, at which point the version becomes fixed and the selector is static.

### Examples

```
VaultDate (yesterday)
```

```
VaultDate (4/11/00)
```

```
Rel40:VaultDate (today)
```



Auto(<tag>)

Used for auto-branching, which creates branches on an as-needed basis as opposed to branching an entire project. This methodology is useful for "what if" scenarios. In general, Auto(<tag>) is equivalent to just <tag>. The 'Auto' is significant only for operations that can create branches (ci, co -lock, populate -lock). The Auto keyword is case insensitive. An Auto(<tag>) selector is a dynamic selector.

### Examples

```
Auto (Dev)
```

```
auto (Rel2.1_p1)
```

### Notes:

- The value supplied to the auto-branch selector must be a branch or version name, not a branch selector. Auto (Golden: ) and Auto (Golden: Latest) are illegal selectors.
- The current version is always the branch-point version when Auto () creates a new branch.
- If present, Auto (<tag>) must be the first selector in a selector list.

`<tag>,[<tag>,....]<tag>`

When a list of selectors is specified, the operation processing varies depending on whether it is operating in a module or files-based vault.

When the selector list is processed for a module, the last tag selector is designated as the main selector and module members matching the tags are overlaid sequentially beginning with the next-to-last selector and finishing with the first selector on the line. For more information on working with a module using a selector list, see Module Member Tags.

When the selector list is processed for a files-based vault, the operation is performed with the objects that match the first matching tag, until all the selectors have been processed.. This is used for integrating a fixed set of module members into an editable development environment.

Using the selector list can recurse through a directory hierarchy according to the command it is used with, however it does not recurse through a static href, or a hierarchical reference to a legacy module, external module, or file based vault. The selector list is silently ignored when applied recursively to these sub-module types.

If you use a selector list during an initial populate, the module manifest defined by the selector is used as the persistent selector for the workspace.

To change the persistent selector, populate with the desired new selector.

### Using Latest and Date() Selectors

- `<branchtag>:Latest` is equivalent to `<branchtag>:Date(<a_date_in_the_future>)`.

- When used with the **setselector** command or as part of a selector list argument (more than one selector) to the `-version` option, `Latest` and `Date()` must be qualified with a branch:

```
<branchtag>:Latest
```

```
<branchtag>:Date(<date>)
```

- For backward compatibility, DesignSync supports selectors of the form `<version_number>:Latest` and `<version_number>:Date(<date>)`. DesignSync uses the branch of the specified version, and then applies the `Latest` or `Date()` selector. For example, `1.2:Latest` resolves to `1:Latest`, and `1.3.2.1:Date(yesterday)` resolves to `1.3.2:Date(yesterday)`.
- When used as the only selector to a `-version` option, DesignSync augments the selector with the persistent selector list. For example, if the persistent selector list is `Gold:,Trunk` and you specify `co -version Latest`, the selector list used for the operation is `Gold:Latest,Trunk:Latest`; the persistent selector list remains `Gold:,Trunk` after the operation.

**Exception:** When you check in an object whose branch you have locked (having done a checkout or populate with the `-lock` option), the date selector augments the current branch, not the persistent selector list. You typically want to remain working on the locked branch even if the persistent selector list has changed.

These restrictions are required to avoid ambiguity about which branch a `Latest` or `Date()` selector applies to.

#### Related Topics

Selecting Versions and Branches

What Are Selectors?

What Are Selector Lists?

What Are Persistent Selector Lists

Date Formats

ENOVIA Synchronicity Command Reference: `setselector`

## Date Formats

This section describes how you can specify dates when using `Date(<date>)` and `VaultDate(<date>)` selectors. DesignSync uses a public-domain date parser that supports a wide range of date and time specifications. The parser is the same one used

## DesignSync Data Manager User's Guide

by the Gnu family of tools. Visit a Gnu website for a complete specification. This section documents the more common formats.

**Note:** If the date specification contains spaces, you must surround the entire selector list with double quotes. For example: "Gold: Date(last tuesday),Trunk"

### Year

You can specify the year using 2 or 4 digits. DesignSync interprets 2-digit year specifications between 00 and 69, inclusive, as 2000 to 2069, and specifications between 70 and 99, inclusive, as 1970 to 1999. If you omit the year, the default is the current year.

### Month

You can specify the month as a number (1 through 12), or using the following names and abbreviations:

January Jan Jan.  
February Feb Feb.  
March Mar Mar.  
April Apr Apr.  
May May May.  
June Jun Jun.  
July Jul Jul.  
August Aug Aug.  
September Sep Sep. Sept Sept.  
October Oct Oct.  
November Nov Nov.  
December Dec Dec.

Note that September is the only month for which a 4-letter abbreviation is valid.

If you omit the month, the default is the current month.

### Day

You can specify days of the week in full or with abbreviations:

Sunday Sun Sun.  
Monday Mon Mon.  
Tuesday Tue Tue. Tues Tues.  
Wednesday Wed Wed. Wednes Wednes.  
Thursday Thu Thu. Thurs Thurs.  
Friday Fri Fri.  
Saturday Sat Sat.

You can add words such as "last" or "next" before a day of the week to specify a date other than the nearest day of the same name. For example:

Thursday -- Specifies the most recent past Thursday, or today, if today is Thursday.

next Thursday -- Specifies one week after the most recent Thursday (includes the current day if today is Thursday).

last Thursday -- Specifies one week before the most recent Thursday (includes the current day if today is Thursday).

If you omit the day, the default is the current day.

Note that a comma after a day of the week item is ignored.

## Time

You specify the time of the day as hour:minute:second, where hour is a number between 0 and 23, minute is a number between 0 and 59, and second is a number between 0 and 59.

Any portion not specified defaults to "0", so a date specification of 03/04/00 defaults to a time of 00:00:00, which is the start of the day (end of the previous day).

## Examples

All of the following examples specify the same calendar date:

### Note:

The preferred order in the U.S. may be ambiguous compared to other countries usage of DD-MM-YY if the number of either the month or the day is less than 10. For example, a date such as 9/12/00 means September 12, 2000 in the U.S. but December 9, 2000 in many other countries.

|            |                                                                         |
|------------|-------------------------------------------------------------------------|
| 2000-09-24 | ISO 8601.                                                               |
| 00-09-24   | 00 indicates year 2000.                                                 |
| 00-9-24    | Leading zeros are not required. For example, "9" is equivalent to "09". |
| 09/24/00   | U.S. preferred order. See previous note.                                |
| 24-sep-00  | Three-letter month abbreviations are allowed.                           |
| 24sep00    | Hyphen and slashes are not required delimiters.                         |

|                |                                                                                                                                                                              |
|----------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 23 sep 00      | Spaces are permitted, but the entire selector list must be placed within double quotes.                                                                                      |
| 20:02:00       | 2 minutes after 20 (8pm, local time).                                                                                                                                        |
| 20:02          | 2 minutes after 20 (8pm, local time), zero seconds implied.                                                                                                                  |
| 8:02pm         | 2 minutes after 8pm, local time.                                                                                                                                             |
| 20:02-0500     | 2 minutes after 8pm Eastern U.S. Time. The time is Eastern U.S. time because -0500 means 5 hours behind UTC (Coordinated Universal Time, also known as Greenwich Mean Time). |
| 01/24/00 20:02 | 2 minutes after 8pm on January 24th 2000. Contains spaces, so the entire selector list must be placed within double quotes.                                                  |

**Related Topics**

Selecting Versions and Branches

What Are Selectors?

Selector Formats

## Parallel (Multi-Branch) Development

### Using the Merging Work Style

The following operations define the merging work style:

- To check out an object for editing, fetch an unlocked copy:
  - From the Populate dialog box, select **Unlocked copies**.
  - From the Check Out dialog box, select Unlocked copies.
  - When using the **co** or **populate** commands, use the `-get` option.
- To check in your changes:
- 1. Determine whether you must first merge. The Status column in the List View or from the **ls -report status** command displays "Needs Merge" if a newer version of the object has been checked in, in which case you must merge. If no merge is required, go to step 4.
  2. Perform a merge:
    - From the Check Out dialog box, select **Unlocked copies** and **Merge with workspace**.

- From the Populate dialog box, select **Unlocked copies** and **Merge with workspace**.
  - When using the **co** or **populate** commands, use the `-merge` option.
3. Resolve merge conflicts, if any.
  4. Check in the merged object using the Check In dialog box or **ci** command selecting whichever state option you want, except lock.

Your project leader can use access controls to enforce a merging style by denying all check-out operations that request a lock. See the access controls documentation for additional information, particularly the access control definition.

#### Related Topics

Locking and Merging Work Styles

Using the Locking Work Style

ENOVIA Synchronicity Command Reference: `co`

ENOVIA Synchronicity Command Reference: `populate`

ENOVIA Synchronicity Command Reference: `ci`

ENOVIA Synchronicity Command Reference: `ls`

## Parallel (Multi-Branch) Development

Design processes, for both software and hardware projects, are complex. There are typically many places where tasks can be performed in parallel. DesignSync uses the design management features of branching and merging to implement parallel design processes. Parallel development, when properly managed, can result in:

- **Increased quality** by isolating new feature development
- **Increased productivity** by increasing the bandwidth of a design team and allowing multiple tasks to be managed on multiple branches
- **Increased flexibility** by allowing decomposition and experimentation of features separate from the main line of development

Branches can have different purposes depending on your project's development methodology:

- Feature Branch

Each new project feature is isolated on its own branch, minimizing the risk of the feature negatively impacting other parts of the design. This approach assumes that you are starting from a stable, high-quality branch (baseline). As features

mature, they are merged back into the baseline. It may make sense to group several related features on the same branch, reducing the number of branches needed for the project. See Feature or Subproject Branches for an example of a feature branch lifecycle.

- Release Branch

Each branch is associated with a release milestone. For example, as the 2.1 version of a product nears its release, a branch is created off the main development branch. Final bug fixes and qualification can take place on the 2.1 branch as new development for a follow-on release takes place on the main branch. Two styles of release branching can be used: central main and cascading main. In the central main approach, all side branches eventually merge back to the main branch. In the cascading main approach, a side branch can itself become the main development branch.

- Policy Branch

A policy branch supports a new design policy, such as a new tool or a new version of an existing tool being incorporated into the design flow. The new policy or methodology is developed on a side branch so as not to disrupt other development on the main branch. When the policy has been implemented, the policy branch becomes the main development branch. See Policy Branches for an example of a policy branch lifecycle. SITaR, the Submit, Integrate, Test and Release methodology, uses this development style. For more information on SITaR, see Overview of SITaR Workflow.

- Variant Branch

A variant branch supports the investigation of different approaches to the same design. Work done on one variant does not affect other variants. The variant branches never merge. Instead, the variants are developed in parallel until a variant is no longer needed, at which time the variant branch is abandoned.

- Platform Branch

A platform branch supports a specific hardware platform or a process technology. For a software project, this model would facilitate porting the source code to multiple platforms. In chip design, this model might apply to supporting multiple process technologies for the same design.

Creating branches is a simple process. When the branches are in use, however, managing multiple active branches can be difficult. Study your design process and determine where parallel development might benefit you. You should clearly weigh the costs in time and effort of adopting a parallel design process:



- Up-front design, training, and resources

A project manager must define the branching structures and nomenclature to assist in the naming of branches. The project manager must communicate clearly the branch structure, and designers must be trained in the design process to avoid making changes on the wrong branch.

- Process maturity

If your design team has not defined its processes, it is difficult to suddenly mandate a highly structured parallel development process. A parallel design process is refined over many projects. The tools used for parallel development need to conform to your design process; the tool should not dictate a design process that you must follow.

- Branch management and administration

Dedicated project managers are often required to monitor the number of branches and avoid unnecessary and confusing branch creation, to create and merge branches, and to mentor users about the design process.

#### Related Topics

Creating Branches

Other Branch Operations

Feature or Subproject Branches

Policy Branches

Autobranching: Exploring "What If" Scenarios

Merging Module Data

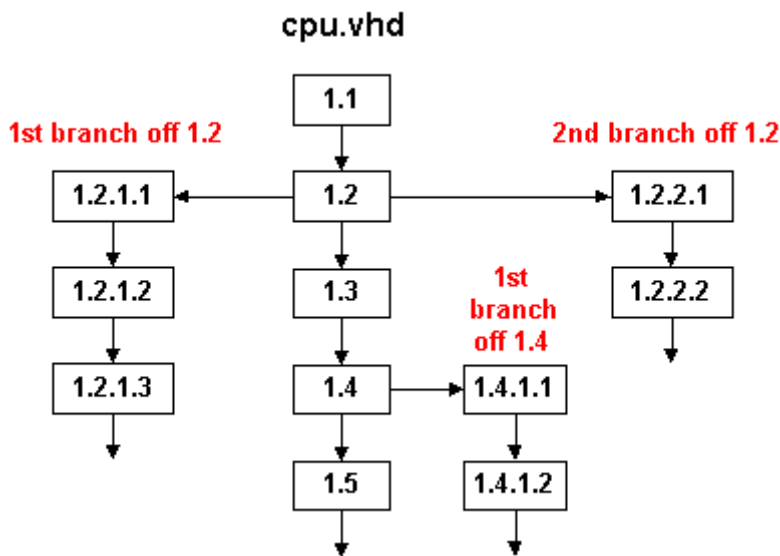
### Methods for Creating Branches

DesignSync recommends and supports two methods for creating branches: project branching and autobranching.

**Project branching**, also known as formal or explicit branching, involves creating a new branch for every object in a project. When there is a need for a new branch, a single person, typically a release engineer or project manager, uses the `mkbranch` command to branch all design objects at the same time. If you are using the GUI, use the Revision Control => Make Branch dialog. Users can then create a new work area for the new branch. Project branching is typically used to implement the branch types discussed in the Parallel Development topic.

**Autobranching**, also known as informal or implicit branching, involves creating a new branch for individual objects on an as-needed basis. Autobranching is typically used by a single or small group of developers who need to try "what if" scenarios, where the life of the branch is known to be limited and will affect only a small portion of the objects in the project. Autobranching is implemented by individual developers using the **Auto(<branchTag>)** selector. When the developer checks in or checks out with a lock on an object, a new branch is created if the branch does not already exist. If the branch already exists, then operations take place on that auto-created branch. See [Autobranching:Exploring "What If" Scenarios](#).

The following diagram shows a file that has several branches with versions on each branch:



DesignSync does not limit you to one branch off a given version. For example, `cpu.vhd` version 1.2 has two branches off it.

Note that referring to a branch by its branch number is not recommended. DesignSync requires that a branch tag be associated with a branch when it is created, whether using `mkbranch` or `Auto()`. Use branch tags to identify branches. A branch can have more than one branch tag.

#### Related Topics

[Parallel \(Multi-Branch\) Development](#)

[Other Branch Operations](#)

### Other Branch Operations

In addition to creating branches, DesignSync supports the following branch operations:

- To view what branches an objects has, use Go to Vault from the graphical interface or the **vhistory** command.
- To add, move, or delete branch tags, use the **tag** command.
- To lock a branch, check the branch out with a lock using Check Out or Populate from the graphical interface or the **co** or **populate** command.
- To merge two versions on the same branch, check out the Latest version of an object with the merge option using Check Out or Populate from the graphical interface or the **co** or **populate** command. See the Using the Merging Work Style topic for details.
- To merge across branches, use the **co** or **populate** command with the `-merge` and `-overlay` options.
- To unlock a branch you have locked in your work area due to a check-out operation, use Check In from the graphical interface or the **ci** command to check in your changes and release the lock, or use Cancel Checkout from the graphical interface or the **cancel** command to cancel your checkout. To unlock a branch locked by another user or locked by you when you no longer have the object in your work area, use Unlock from the graphical interface or the **unlock** command.
- To retire a non-module branch, use Retire from the graphical interface or the **retire** command. You cannot retire a module branch. Retiring obsolete branches discourages users from making change to files on the branch. You can also make the branch harder to access by removing the branch tags as described in Tagging Versions and Branches from the graphical inferace or the tag command.
- To create or remove merge edges used when a module is merged across branches, use the **mkedge** or **rmedge** commands.
- To discourage changes to a non-module branch (not a module version, branch, or member) that's become obsolete, you can use the retire command to retire the DesignSync branch or use the tag command to remove all the branch tags from the branch.
- To completely remove a non-module branch (not a module version, branch or member), use the **purge** command.

Note: DesignSync does not support deleting or retiring module branches.

#### Related Topics

ENOVIA Synchronicity Command Reference: vhistory

ENOVIA Synchronicity Command Reference: tag

ENOVIA Synchronicity Command Reference: co

ENOVIA Synchronicity Command Reference: populate

ENOVIA Synchronicity Command Reference: ci

ENOVIA Synchronicity Command Reference: ls

ENOVIA Synchronicity Command Reference: cancel

ENOVIA Synchronicity Command Reference: unlock

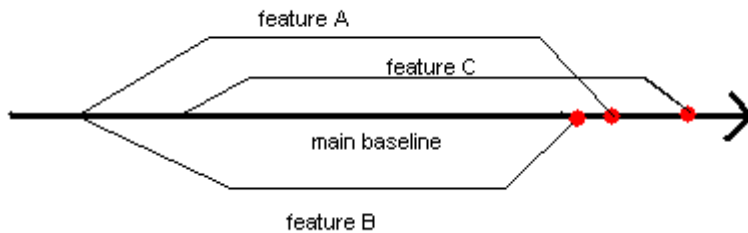
ENOVIA Synchronicity Command Reference: retire

ENOVIA Synchronicity Command Reference: purge

### Example Branching Scenarios

#### Feature or Subproject Branches

Feature or subproject (a set of related features) branches are used for feature isolation. The main development branch, or baseline, remains relatively stable as each feature or subproject is developed on a separate branch. When a feature is complete, or at least stable, the feature branch is merged back to the baseline. Multiple feature branches may be active in parallel, but are isolated from each other in order to simplify the development cycle for each feature.



In this diagram, features A and B are developed on separate side branches of the main project baseline. Development on the main baseline continues, mostly for bug fixes or minor features that will not compromise the stability of the baseline. Development for feature C started later than features A and B, but also takes place on a separate branch. The dots (red if viewed online) in the diagram show where each feature is merged back into the baseline once that feature's isolated development and testing is completed. After the merge, testing on the baseline evaluates the compatibility of the feature with the rest of the baseline. Changes necessary to correct post-merge problems with the feature are performed on the baseline.

Isolating features from the main baseline makes it easy to abandon a feature that is no longer desired or that will not be completed in time for the next product release. You do not merge from the feature branch to the baseline until the feature's future is certain.

The lifecycle of a feature branch is:

1. The project manager creates the branch.

2. The team members create work areas for the feature branch and begin their development.
3. When the feature is completed, the project manager collapses the feature branch back into the baseline.
4. Team members begin working on another branch in a different work area.

### Creating a Feature Branch (Project Manager)

Creating a feature branch is typically the role for a single project manager or release engineer.

1. Create a work area for the main baseline if you do not already have one. For example, if the project vault is  
`sync://myhost.mycom.com/Projects/Xproject,`

the baseline is the `Trunk` branch, and the local work area you want to create is

```
/home/relmgr/Projects/Xproject/Trunk
```

```
dss> scd /home/relmgr/Projects/Xproject
```

```
dss> mkfolder Trunk
```

```
dss> scd Trunk
```

```
dss> setvault  
sync://myhost.mycom.com/Projects/Xproject@Trunk .
```

The `@Trunk` syntax is a shortcut for the **setselector** command to set the persistent selector list for the work area. `Trunk` (shorthand for `Trunk:Latest`) is DesignSync's default persistent selector list, so unless a different persistent selector list has been defined on a parent folder, the `@Trunk` can be omitted in this example. However, if your baseline is any branch other than `Trunk`, you must set the persistent selector list.

The directory structure of having branch directories (`Trunk`) under project directories (`Xproject`) under some top-level projects directory (`Projects`) is only one of many possible hierarchies. Another reasonable organization is to have project directories under branch directories. This organization might facilitate branch-related operations that span projects.

2. Create DesignSync references to the versions from the baseline from which the feature branch will be created. DesignSync references do not exist on disk, so do not require the transfer of data. For example, if you are branching from the `Latest` versions on `Trunk`:

## DesignSync Data Manager User's Guide

```
dss> populate -recursive -reference
```

The `populate` command obeys the work area's persistent selector list and populates from Trunk, and by default DesignSync fetches the Latest versions. If you are branching from versions other than Latest, use the `-version` option to the `populate` command to specify a selector such as a version tag or `Date()` selector.

3. Tag the branch-point versions. While not required, tagging the branch points can be valuable for later troubleshooting activities. Use a meaningful tag name, such as appending "-bp" (branch point) to the name of the branch you are creating. For example, if you are creating two branches, `devA` and `devB`, you would apply two branch tags:

```
dss> tag -recursive devA-bp .
```

```
dss> tag -recursive devB-bp .
```

4. Create the branches:

```
dss> mkbranch -recursive devA .
```

```
dss> mkbranch -recursive devB .
```

### Working on the New Branch (Team Members)

Once the project manager creates the feature branch, team members can set up work areas and begin working on the new branch. For example, team members working on feature A create a work area for the `devA` branch:

1. Create a new work area for the `devA` branch:

```
dss> scd /home/goss/Projects/Xproject
```

```
dss> mkfolder devA
```

```
dss> scd devA
```

```
dss> setvault
```

```
sync://myhost.mycom.com/Projects/Xproject@devA:Latest .
```

By setting the persistent selector list (`@devA:Latest`) on the `devA` folder, `populate`, `check-in`, and `check-out` operations will take place on branch `devA` automatically. Notice that to specify a branch selector, you append the version to the branch tag name, in this case the branch tag is `devA` to which you append `:Latest`.

2. Populate the new work area:

```
dss> populate -get -recursive
```

DesignSync fetches the Latest versions from devA. If no new version of a given object has been created on the devA branch, then DesignSync fetches the branch-point version from the baseline (Trunk).

3. You can now check out, make edits, and check in files to develop feature A. On any given branch, the team can use the locking or merging work style. For example, the team uses the merging work style. You want to make changes to top.v, so you fetch the Latest version, make changes, merge in changes made by another team member, resolve any merge conflicts, then check in the merged version:

```
dss> co -get -comment "Addressing defect #4545" top.v
```

```
[Edit the file.]
```

```
dss> ci -keep top.v
```

```
[The checkin fails because someone has checked in a newer version.]
```

```
dss> co -merge top.v
```

```
[Resolve any merge conflicts.]
```

```
dss> ci -keep top.v
```

You should also periodically populate to pick up changes made by other team members working on the devA branch. If you are using the locking model, then run:

```
populate -get -recursive
```

If you are using the merge model, then run:

```
populate -merge -recursive
```

4. Changes made to other branches do not affect the branch you are working on. However, there may be times when you want to pick up an important change from another branch. In such cases, you can merge individual files or groups of files into your work area. For example, if a developer fixes a problem in the file `alu.vhd` on the Trunk branch, and that bug fix is needed for feature A development, then you can merge that file into the feature A development stream. From the devA work area:

## DesignSync Data Manager User's Guide

```
dss> co -merge -overlay Trunk alu.vhd
```

```
[Resolve any merge conflicts]
```

```
dss> ci -comment "Picked up fix from Trunk" alu.vhd
```

Other team members working on the devA branch will pick up the fix the next time they populate or check out `alu.vhd`. These types of individual merges may need to be coordinated through a team leader.

### Collapsing the Feature Branch (Project Manager)

When the feature branch has reached a level of maturity where it no longer makes sense to maintain a separate branch, the project manager can merge from the feature branch to the baseline.

1. Alert all users of the feature branch (devA) that the merge is taking place and have them check in their final changes.
2. Restrict checkins for the duration of the merge operation. Having activity on either branch complicates the process. To restrict checkins, define an access control:

```
access allow Checkin only users $ProjectManager
```

Remember to have the SyncServer re-read the access-control files (`access reset`) after defining this new access control.

3. Update the Trunk work area with the Latest versions of design files:

```
dss> scd /home/realmgr/Projects/Xproject/Trunk
```

```
dss> populate -get -recursive
```

4. Tag the merge-point versions on the baseline. As with tagging the branch-point versions, this step is optional but recommended. Use a meaningful tag name, such as appending "-mp" (merge point) to the branch name:

```
dss> tag -recursive devA-mp .
```

5. Perform an overlay merge from the feature branch (devA) to the baseline (Trunk):

```
dss> populate -recursive -merge -overlay devA:Latest
```



The merge can produce new files from the devA branch that did not exist on Trunk, files that merge without conflicts, and files where you need to resolve merge conflicts manually.

6. Resolve any merge conflicts.
7. Check in the merged results:

```
dss> ci -recursive -comment "Merged from devA" .
```

Any object that was introduced on the feature branch (did not exist on the baseline prior to the merge) is automatically branched so that the object can be checked into the baseline.

8. For completeness, you may want to tag the final versions on the devA branch to indicate the versions that participated in the merge:

```
stcl> tag -recursive -version devA final [url vault .]
```

This command tags the Latest versions on the devA branch. As you would expect, the command fails for any object in your Trunk work area that does not have a devA branch.

**Note:** To use the programming variable, you must be in stcl mode.

9. Optionally, for non-module objects, retire the feature branch to prevent accidental checkins to the branch and indicate that the branch is no longer active:

```
dss> retire -recursive -branch devA *
```

10. Run regression tests or any other checks to ensure that the baseline is stable and ready for development to continue. Fix any problems that are identified on baseline branch (Trunk).
11. Re-open the vault by removing the access control statement that denied checkins (and remember `access reset`).

### Starting Work on a Different Branch (Team Members)

Once the feature branch is no longer active, team members should start working on a different branch -- the baseline branch or another feature branch. Developers should leave the feature-branch work area intact and create a new work area for future development. DesignSync supports re-using work areas (`setselector -recursive` to change the persistent selector list, then populate with a new configuration), but doing so introduces risks such as losing files that were never checked into a branch, or checking into the wrong branch. If disk space is a concern, consider archiving (compressed tar files or tape backups) and deleting old work areas.

### Related Topics

### Parallel (Multi-Branch) Development

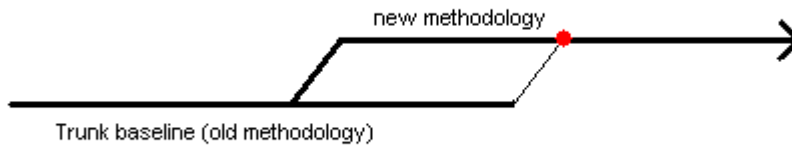
ENOVIA Synchronicity Command Reference: setselector

ENOVIA Synchronicity Command Reference: populate

ENOVIA Synchronicity Command Reference: retire

### Policy Branches

This section describes the lifecycle of a policy branch, which is most effective when introducing major changes in a design flow. This policy change could be moving to a new compiler for a software project, changing to a new vendor of cell libraries in an ASIC project, introducing a new simulation tool and syntax, or a design overhaul. A designated team works on a branch in order not to interfere with continued development on the main branch. Once the new policy branch is well defined and stable, it becomes the main development branch.



The lifecycle of a policy branch is similar to that of a feature branch. The primary difference is that the policy branch eventually becomes the main development branch, whereas the feature branch is merged back to the main branch.

The lifecycle of a policy branch is:

1. The project manager creates the branch.
2. The team members create work areas for the policy branch and begin their development.
3. When the policy change has been implemented and the policy branch is stable, the project manager merges the old baseline into the new policy branch.
4. All team members continue development on the new baseline.

### Creating a Policy Branch (Project Manager)

Creating a policy branch is typically the role for a single project manager or release engineer.

1. Create a work area for the main baseline if you do not already have one. For example, if the project vault is

```
sync://myhost.mycom.com/Projects/Xproject, the baseline is the
Trunk branch, and the local work area you want to create is
/home/relmgr/Projects/Xproject/Trunk:
```

```
dss> scd /home/relmgr/Projects/Xproject
```

```
dss> mkfolder Trunk
```

```
dss> scd Trunk
```

```
dss> setvault
sync://myhost.mycom.com/Projects/Xproject@Trunk .
```

The `@Trunk` syntax is a shortcut for the **setselector** command to set the persistent selector list for the work area. `Trunk` (shorthand for branch `Trunk:Latest`) is DesignSync's default persistent selector list, so unless a different persistent selector list has been defined on a parent folder, the `@Trunk` can be omitted in this example. However, if your baseline is any branch other than `Trunk`, you must set the persistent selector list.

The directory structure of having branch directories (`Trunk`) under project directories (`Xproject`) under some top-level projects directory (`Projects`) is only one of many possible hierarchies. Another reasonable organization is to have project directories under branch directories. This organization might facilitate branch-related operations that span projects.

2. Update your work area to contain the versions from the baseline from which the policy branch will be created. For example, if you are branching from the `Latest` versions on `Trunk`:

```
dss> populate -recursive -get
```

The **populate** command obeys the work area's persistent selector list so populates from the `Trunk` branch, and by default fetches the `Latest` versions. If you are branching from versions other than `Latest`, use the `-version` option to the `populate` command specify a selector such as a version tag or `Date()` selector.

3. Tag the branch-point versions. While not required, tagging the branch points can be valuable for later troubleshooting activities. Use a meaningful tag name, such as appending `"-bp"` (branch point) to the name of the branch you are creating. For example, if the policy branch is `NewCompiler`:

```
dss> tag -recursive NewCompiler-bp .
```

4. Create the branch:

## DesignSync Data Manager User's Guide

```
dss> mkbranch -recursive NewCompiler .
```

### Working on the New Branch (Team Members)

Once the project manager creates the policy branch, team members involved in the policy development can set up work areas and begin working on the new branch.

1. Create a new work area for the NewCompiler branch:

```
dss> scd /home/goss/Projects/Xproject
```

```
dss> mkfolder NewCompiler
```

```
dss> scd NewCompiler
```

```
dss> setvault
```

```
sync://myhost.mycom.com/Projects/Xproject@NewCompiler:Latest .
```

By setting the persistent selector list (@NewCompiler:Latest) on the NewCompiler folder, populate, check-in, and check-out operations will take place on branch NewCompiler automatically. Notice that to specify a branch selector, you append the version to the branch tag name, in this case the branch tag is NewCompiler to which you append :Latest.

2. Populate the new work area:

```
dss> populate -get -recursive
```

DesignSync fetches the Latest versions from NewCompiler. If no new version of a given object has been created on the NewCompiler branch, then DesignSync fetches the branch-point version from the baseline (Trunk).

3. You can now check out, make edits, and check in files to develop the new policy. On any given branch, the team can use the locking or merging work style. For example, the team uses the merging work style. You want to make changes to top.v, so you fetch the Latest version, make changes, merge in changes made by another team member, resolve any merge conflicts, then check in the merged version:

```
dss> co -get -comment "Addressing defect #4545" top.v
```

```
[Edit the file.]
```

```
dss> ci -keep top.v
```

[The checkin fails because someone has checked in a newer version.]

```
dss> co -merge top.v
```

[Resolve any merge conflicts.]

```
dss> ci -keep top.v
```

You should also periodically populate to pick up changes made by other team members working on the NewCompiler branch.

4. Changes made to other branches do not affect the branch you are working on. However, there may be times when you want to pick up an important change from another branch. In such cases, you can merge individual files or groups of files into your work area. For example, if a developer fixes a problem in the file `alu.vhd` on the Trunk branch, and that bug fix is needed for the new compiler work, then you can merge that file into the NewCompiler branch. From the NewCompiler work area:

```
dss> co -merge -overlay Trunk alu.vhd
```

[Resolve any merge conflicts]

```
dss> ci -comment "Picked up fix from Trunk" alu.vhd
```

Other team members working on the NewCompiler branch will pick up the fix the next time they populate or check out `alu.vhd`. These types of individual merges may need to be coordinated through a team leader.

### Switching to the New Baseline (Project Manager)

When the policy development is complete or at least stable, the project manager merges the old baseline into the policy branch so that the policy branch becomes the new baseline.

1. Alert all users that the merge is taking place and have them check in their final changes to both Trunk and NewCompiler.
2. Restrict checkins for the duration of the merge operation. Having activity on either branch complicates the process. To restrict checkins, define an access control:

```
access allow Checkin only users $ProjectManager
```

Remember to have the SyncServer re-read the access-control files (`access reset`) after defining this new access control.

3. Update the NewCompiler work area with the Latest versions of design files:

```
dss> scd /home/relmgr/Projects/Xproject/NewCompiler
```

```
dss> populate -get -recursive
```

4. Tag the merge-point versions on the NewCompiler branch. As with tagging the branch-point versions, this step is optional but recommended. Use a meaningful tag name, such as appending "-mp" (merge point) to the branch name:

```
dss> tag -recursive NewCompiler-mp .
```

5. Perform an overlay merge from the old baseline (Trunk) to the policy branch (NewCompiler).

```
dss> populate -recursive -merge -overlay Trunk
```

The merge can produce new files from Trunk that did not exist when NewCompiler was branched, files that merge without conflicts, and files where you need to resolve merge conflicts manually.

6. Resolve any merge conflicts. Because NewCompiler will be the baseline going forward, merge conflicts are generally resolved in favor of the changes on the NewCompiler branch.
7. Check in the merged results:

```
dss> ci -recursive -comment "Merged from Trunk" .
```

Any object that was introduced on the baseline after the policy branch was created (did not exist on the policy branch prior to the merge) is automatically branched so that the object can be checked into the policy branch.

8. For completeness, you may want to tag the final versions on the Trunk branch to indicate the versions that participated in the merge:

```
dss> tag -recursive -version Trunk final .
```

This command tags the Latest versions on the Trunk branch. As you would expect, the command fails for any object in your NewCompiler work area that does not have a Trunk branch.

9. Going forward, NewCompiler is the main baseline, so you might consider applying a new branch tag to the old Trunk and moving the Trunk branch tag to the NewCompiler branch.

```
dss> tag -recursive TrunkPriorToNewCompiler -branch Trunk .
```

```
dss> tag -recursive -replace Trunk -branch NewCompiler .
```

10. Optionally retire the old baseline branch to prevent accidental checkins to the branch and to generally flag the branch as no longer active:

```
dss> retire -branch TrunkPriorToNewCompiler *
```

Note that the **retire** command does not have a recursive option, so you must execute this command in every folder in the hierarchy, or create a script to traverse the hierarchy for you.

11. Run regression tests or any other checks to ensure that the new baseline is stable and ready for development to continue. Fix any problems that are identified on the NewCompiler baseline branch, which is now also tagged Trunk.
12. Re-open the vault by removing the access control statement that denied checkins (and remember `access reset`).

### Starting Work on the New Baseline (Team Members)

Team members who were not working on the policy branch must now create work areas for the policy branch, which is now the new Trunk baseline. Developers should leave the previous Trunk work areas intact and create new work areas for future development. DesignSync supports re-using work areas (`setselector -recursive` to change the persistent selector list, then populate with a new configuration), but doing so introduces risks such as losing files that were never checked into a branch, or checking into the wrong branch. If disk space is a concern, consider archiving (compressed tar files or tape backups) and deleting old work areas.

Team members who were working on the policy branch should change their persistent selector list from NewCompiler to Trunk:

```
dss> setselector -recursive Trunk .
```

This operation does not re-use a work area, but reflects the fact that the branch associated with the current work area is now identified as Trunk.

### Related Topics

Parallel (Multi-Branch) Development

ENOVIA Synchronicity Command Reference: `setselector`

ENOVIA Synchronicity Command Reference: `populate`

ENOVIA Synchronicity Command Reference: `retire`

### Autobranching: Exploring "What If" Scenarios

Autobranching allows individuals or small teams to try out new ideas in the context of the work in progress without impacting the baseline development. This informal approach to branching places the responsibility for branching and merging on individual team members instead of the project manager who is responsible for formal branching activities. Unlike formal project branching, where all the objects in a project are branched at the same time, autobranching creates branches only for objects that change.

Autobranching is most effective when the number of people working on the branch, the number of project files affected, and the duration of the effort are small. The larger the effort, the more likely it would benefit from formal branching.

DesignSync implements autobranching using the `Auto(<tag>)` selector. See [Selector Formats](#) for details.

Assume that you and a few other team members have an idea for a bug fix for "Yproject" that you are not sure will work out. Your main development branch is `Rel2.1` (a release branch), and you will test your bug fix using an autobranch called `DevFix`.

1. You (and each team member working on `DevFix`) create a `DevFix` work area:

```
dss> scd /home/goss/Projects/Yproject
dss> mkfolder DevFix
dss> scd DevFix
dss> setvault sync://myhost.mycom.com/Projects/Yproject .
dss> setselector Auto(DevFix),Rel2.1:Latest .
```

To set the persistent selector list, you could have used the `@Auto(DevFix),Rel2.1:Latest` syntax to the **setvault** command as a shortcut for the **setselector** command.

The `Auto(DevFix)` selector indicates that you are autobranching to the `DevFix` branch. The second selector in the persistent selector list, `Rel2.1:Latest`, indicates that `Rel2.1` is your baseline branch from which `DevFix` branches are created as needed.

2. Populate the work area:

```
dss> populate -get -recursive
```



DesignSync fetches Latest versions of files with a DevFix branch (probably none at this point, unless one of your DevFix team members already autobranched an object), then fetches from Rel2.1.

3. You and your team members perform check-out, check-in, populate, and other operations as you normally would. Because of autobranching:
  - DesignSync automatically creates a DevFix branch when you check in or check out with a lock an object that does not have a DevFix branch.
  - For fetch operations DesignSync first tries to fetch from the DevFix branch and failing that fetches from Rel2.1.

You and the DevFix team are effectively working on two branches with one branch serving as the baseline while your new development takes place on the autobranch . Because only you and the other team members working on this experimental fix have work areas that reference the DevFix branch other team members working on Rel2.1 do not see the DevFix development.

4. There may be times when the DevFix team is making changes to the same files that are being modified on the baseline. If the changes on the baseline are needed on DevFix, you can merge from Rel2.1. For example, to pick up changes to `reg8.vhd` and `reg16.vhd`:

```
dss> co -merge -overlay Rel2.1:Latest reg8.vhd reg16.vhd
```

As with any merge, you may need to resolve merge conflicts, and then you can check in the merged version to the DevFix branch:

```
dss> ci -comment "Merged changes done on Rel2.1" reg8.vhd
reg16.vhd
```

5. Eventually, DevFix development needs to be merged back to the baseline, unless the fix was unsuccessful, in which case this merge step is skipped. To merge the DevFix branch back to the Rel2.1 baseline, a designated user would go to a fully populated Rel2.1 work area (the persistent selector list set to `Rel2.1:Latest`, not `Auto (DevFix) , Rel2.1:Latest`), then merge:

```
dss> scd /home/goss/Projects/Yproject/Rel2.1
```

```
dss> url selector .
```

```
Rel2.1:Latest
```

```
dss> populate -get -recursive
```

```
dss> populate -merge -overlay DevFix:Latest -recursive
```

Resolve any merge conflicts, and then the merged files can be checked into Rel2.1:

```
dss> ci -comment "Merged from DevFix" -recursive .
```

The checkin fails for any object that was introduced on the autobranch because the object does not exist on the baseline branch. For these objects, you need to create the branch and then check in. For example, if `test.asm` and `test.mem` were introduced on DevFix:

```
dss> mkbranch Rel2.1 test.asm test.mem
```

```
dss> ci -comment "Introduced on DevFix" test.asm test.mem
```

6. At this point, the autobranch development was a success and the branch was merged back to the baseline, or the autobranch development was not successful. Either way, it is time to abandon the branch -- future development takes place on the baseline branch (Rel2.1). You can optionally retire the DevFix branch to prevent accidental checkins to that branch:

```
dss> retire -branch DevFix *
```

Note that the `retire` command does not have a recursive option, so you must execute this command in every folder in the hierarchy, or create a script to traverse the hierarchy for you.

DesignSync supports reusing work areas (`setselector -recursive` to change the persistent selector list, then populate with a new configuration), so in the autobranch scenario, the same work area could have been used before, during, and after the DevFix development. However, reusing work areas introduces risks such as losing files that were never checked into a branch, or checking into the wrong branch.

**Tip:** Define a work area for each development branch. If disk space is a concern, consider archiving (compressed tar files or tape backups) and deleting old work areas.

### Related Topics

Parallel (Multi-Branch) Development

ENOVIA Synchronicity Command Reference: `setvault`

ENOVIA Synchronicity Command Reference: `retire`

ENOVIA Synchronicity Command Reference: `setselector`

ENOVIA Synchronicity Command Reference: `populate`

## Working with Legacy Modules

### How DesignSync Handles Legacy Modules

Starting with the DesignSync Developer Suite version 5.0, the structure of modules was enhanced to provide greater functionality and speed. However, to use the new functionality,, legacy modules from previous versions of DesignSync require an upgrade.

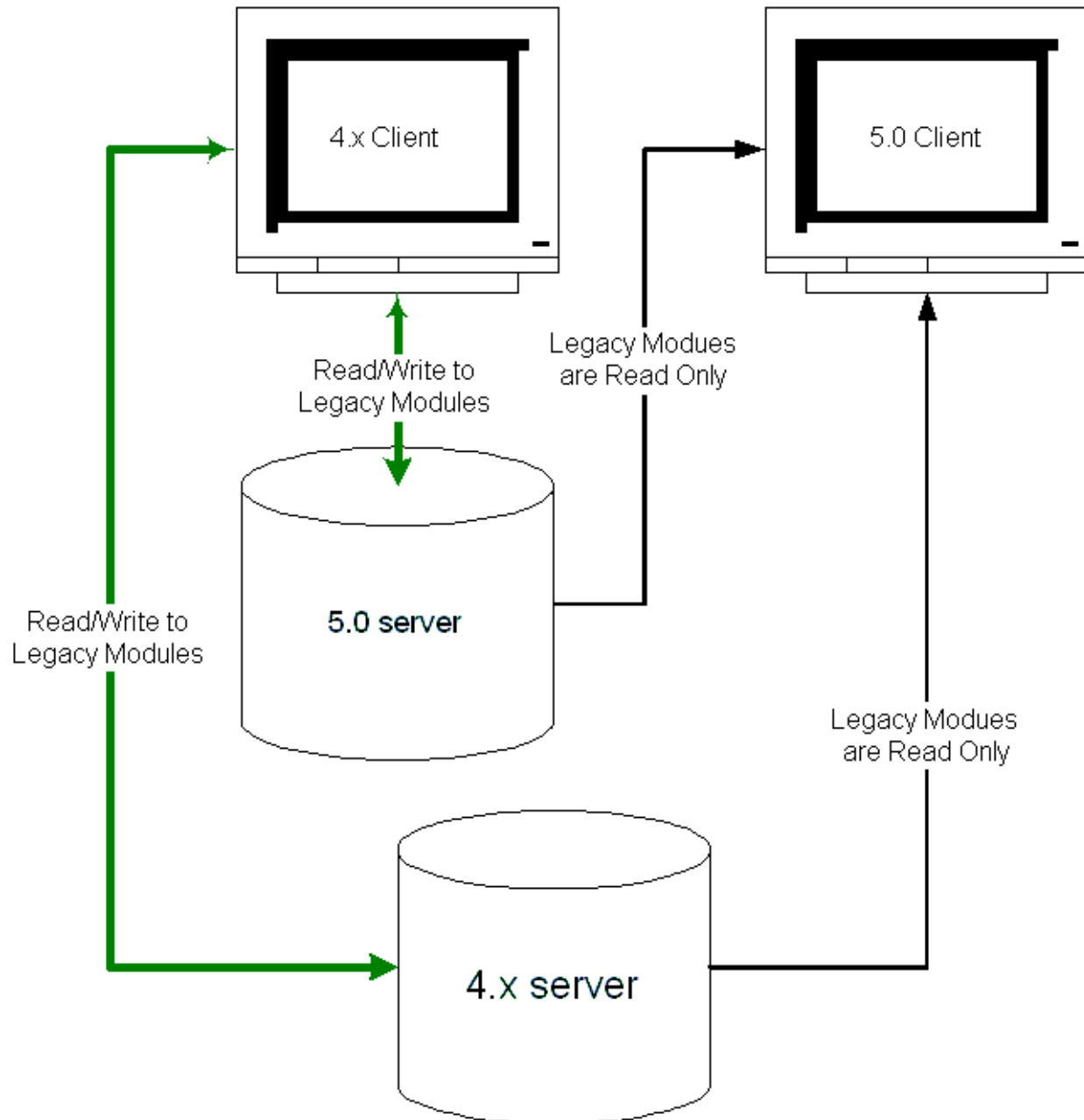
Note: If you cannot update to take advantage of the benefits of the new modules structure, you may use the "hcm legacy module mode" which provides the ability to modify as well as view and reference legacy modules. For more information, see the *ENOVIA Synchronicity DesignSync Data Manager HCM User's Guide*.

To take advantage of the new module functionality, all 4.x data must be upgraded with the `upgrade` command.

Because there is an upgrade required to access the new functionality, you might find yourself working in an environment that requires you to use both new modules and legacy modules during the transition period. To allow users to continue to be productive during the upgrade process from the DesignSync 4.x architecture to the DesignSync 5.0 architecture, users can use a 4.x client to modify legacy modules, or if you have Legacy Module Mode enabled, you can use the current DesignSync version.

**Important:** If you are using Legacy Module mode and modern modules, you cannot modify them in the same workspaces. Users must have separate workspace for editing work. This is discussed more fully in *ENOVIA Synchronicity DesignSync Data Manager HCM User's Guide*. The rest of this topic assumes that you are not using legacy module mode.

There are some restrictions in the DesignSync client that control how you can use legacy modules. How DesignSync handles legacy modules in a mixed 4.x/5.0 server and client installation is shown in the graphic below:



**Note:** A user of a 5.x client can not write to a legacy module. The module can only be populated in a read-only mode.

#### Related Topics

*ENOVIA Synchronicity Command Reference Help: upgrade*

*ENOVIA Synchronicity DesignSync Data Manager HCM User's Guide.*

### Upgrading Legacy Modules

In this version of DesignSync, you can create modules to manage your design data. Legacy modules, or those modules created with the `mkmod` command in DesignSync versions prior to 5.0, are read-only from this DesignSync client. To take advantage of the functionality of new modules, and use your legacy modules in interactive mode, you can upgrade your legacy modules using the `upgrade` command.

Legacy modules can be upgraded on an as-needed basis. Modules that have not been upgraded are still available to users in read only mode.

Legacy modules use configurations to associate versions of objects together into a collective "module" unit. These configurations are described in the `sync_project.txt` file under the legacy module's vault folder. The `upgrade` command uses this `sync_project.txt` file as a guide for upgrading the legacy module's configurations into the new module. Other branch or selector configurations might be defined in additional `sync_project.txt` files within the legacy module vault hierarchy.

**Important:** Before you upgrade a legacy module, you must check in any modifications and release any locks.

The command allows you to specify the name of the new module as well as a category path. For example, if your legacy module is located on the server at some level below the `/Projects` area on the server, you may use the `-category` option in conjunction with the `-name` option on the `upgrade` command to define the same directory structure as your original project vault.

For example, if the URL of the original legacy module is:

```
sync://granite:2647/Projects/Memory/ROM
```

Then you may use the following `upgrade` command line:

```
hcm upgrade sync://granite:2647/Projects/Memory/ROM -category  
NewMemory -name ROM1
```

The URL of the upgraded module object would be:

```
sync://granite:2647/Modules/NewMemory/ROM1
```

Custom access controls are not upgraded along with the module. If custom access controls are desired for the new module, they can be added either before or after the module is upgraded. After an upgrade completes, the server's administrator is notified by email that the module was upgraded, and the custom access controls were not carried forward to the new module.

Access controls restricting access to the legacy module are added by the upgrade process, while it is in progress, to prevent modifications to the legacy module. The legacy module access controls remain in place after the upgrade has completed to prevent accidental upgrades to the legacy module.

### Notes:

- All ProjectSync notes associated with the legacy module and its configurations are also associated with the new module after the upgrade is complete. However, any email subscriptions associated with elements from the legacy module vault hierarchy are not modified and must be added manually by the users. If DesignSync is configured to create a RevisionControl note for upgrade, and email notifications is enabled, the upgrade process sends an email to all users currently subscribed to activity on the legacy module vault folder informing them of the vault's upgrade to a module. The users should then update their subscriptions.
- It is also important to note that this upgrade occurs on the server only. Any existing workspaces will continue to point to the existing legacy module configuration. The new module resulting from the upgrade will need to be populated into a new workspace.

An upgrade log is created on the server to allow the user to track the progress of the upgrade. This log remains after the upgrade is complete and can be viewed at the following URL:

```
http://<host>:<port>/syncserver/upgrade/upgrade_<category>_<name>.html
```

### Notes:

- All slashes in the `<category>` field will be replaced with underscores.
- If server customizations have been imported to a different server after a module upgrade, the upgrade log may not appear at the URL provided for monitoring the upgrade. If the URL is not found, you may still access the upgrade log from the ModuleUpgrade sub-directory. For more information on the ModuleUpgrade sub-directory, see the `upgrade` command documentation in the *ENOVIA Synchronicity Command Reference*.

## Types of Objects Created During the `hcm upgrade` Command

The following types of objects are created as a result of performing an upgrade on a legacy module.

### Module:

A module with the specified name and category path.

**Module branches:**

- A default module branch is created and is assigned an immutable module branch tag of “Trunk”.
- If any release configurations are defined for the legacy module, a module branch is created from module version 1.1 and a mutable module branch tag named “Releases” is added to that branch. This branch will contain each release configuration (as a module version) defined for the legacy module.
- A module branch is created from module version 1.1 for every branch configuration and selector configuration. The configuration name is added to the module branch as a mutable module branch tag. The first module version is created on the module branch and the member versions that are part of the configuration are added to the module version.

**Module versions:**

- Module version 1.1 is created to include all members matching the “Trunk:Latest” selector. If no files resolve to this selector, the module version is empty. If a Trunk configuration is defined to be mapped to a different selector, the upgrade process issues a warning noting that the Trunk branch will contain the members matching the Trunk:Latest selector and not the selector defined in the `sync_project.txt` file.
- Each module branch is created with an initial module version comprised of member versions that were part of the legacy module configuration.
- A new module version is created on the “Releases” branch for each release configuration with all member versions that are part of that release. The module versions will be created in chronological order by release.

**Module branch tags:**

For each configuration that is exactly equal to the Latest on the legacy module’s branch, a mutable module branch tag with the same name is added to the module branch created for the configuration.

For example, if your legacy module’s `sync_project.txt` file contains the entry “CONFIG Silver Main:Latest”, a module branch named “Silver” is created and the mutable module branch tag “Main” is added to the branch, as long as a Main configuration is not also defined.

**Module version tags:**

- An immutable module version tag is created for every release configuration, using the release name as the tag name, and added to the module version that was created for the release configuration.
- A mutable module version tag is created for every legacy module alias configuration, using the alias name as the tag name and adding the tag to the module version (previously created for the release configuration) to which the alias refers.

- For each configuration that is exactly equal to a version tag, a mutable module version tag with the same name will be added to the first module version on the branch that was created for the configuration.

For example, if your legacy module's `sync_project.txt` file contains the entry "CONFIG Gold Alpha", a module branch named "Gold" is created and the mutable module version tag named "Alpha" is added to the first module version on the "Gold" branch.

### Hierarchical References:

- Legacy module hrefs are associated with a configuration of a legacy module. During the upgrade, the configurations will be converted to either a module branch (for branch and selector configurations) or a module version (for release or alias configurations). The legacy module href is converted to a new module href and added to the module branch (the first module version on the branch) or to the module version that was created for this configuration.
- The href will refer to another legacy module, a DesignSync vault folder or an IP Gear deliverable.

By default, or if the `-maphrefs` command option was specified, the upgrade attempts to determine whether the target of the href has already been upgraded to a new module. If so, the href will point to the new module rather than the legacy module with the new module's branch selector or module version selector recorded as the dynamic hrefmode value and the new module version's number recorded as the static hrefmode value.

An href can only be mapped if the target's configuration was converted to a module branch or module version. For example, if a target configuration was not defined in the legacy module's `sync_project.txt` file, the target configuration is not transferred to the new module since the mapping process does not have a branch or version in which to place this href. An error is issued during the mapping process and the upgrade terminates without creating the new module.

If you want the mapping errors treated as warnings instead, use the `-nomaperror` option to transfer the original href. Mapping errors are issued if a href's target was upgraded but the new module does not exist, if there is no matching branch or version for the href's target configuration, or if the href target's server is not accessible.

**Tip:** If you plan on upgrading an entire legacy module hierarchy, you should do so in a bottom-up fashion, performing the top-module upgrade last, so the hrefs map to the newly upgraded modules.

After the upgrade is complete, you may migrate any of the tags that did not get transferred to the new module with the `migratetag` command. Any existing selector list, except date selectors, can be specified with `migratetag` to create a new module



branch later. The supported selectors, such as version tags, or branch selector can also be combined to create a new module branch.

Please refer to the `upgrade` or `migratetag` command topics in the *ENOVIA Synchronicity Command Reference* for more detailed information.

### An Example of Upgrading a Legacy Module

The following scenario illustrates the procedure for upgrading an existing legacy module to a module.

1. Amy, the designer of an ALU design, ALU, is managing her design through the use of a legacy module, which now needs to be upgraded to use the new module structure available in DesignSync 5.0.

The legacy module, ALU, located at `sync://granite:2647/Projects/ALU` has the following structure in its associated `sync_project.txt` file:

```
NAME ALU
DESCRIPTION HCM module
OWNER amy
COMPONENT FALSE
CONFIG Platinum Trunk amy *
CONFIG_DESC HCM configuration
CONFIG_First_Release_Of_Platinum First_Release_Of_Platinum-
1165686270--R amy *
CONFIG_DESC HCM release
CONFIG_alias_of_platinum First_Release_Of_Platinum-
1165686270--R amy *
CONFIG_DESC HCM alias
```

The legacy module also contains one hierarchical reference to a DesignSync vault folder on the Platinum configuration to:

```
sync://granite:2647/Projects/Development/vault1
```

2. Amy contacts her project manager, who upgrades this legacy module by as follows:

```
dss> hcm upgrade sync://granite:2647/Projects/ALU -name ALU
```

As a result of the upgrade, a new module is created at the following location:

```
sync://granite:2647/Modules/ALU
```

The new module has the following contents:

- Module branch 1, tagged “Trunk”. This module branch contains one module version, 1.1.
- Module branch 1.1.1, tagged “Releases”. This module branch contains one module version, 1.1.1.1 tagged with “First\_Release\_Of\_Platinum” and “alias\_of\_platinum”.
- Module branch 1.1.2, tagged “Platinum”. This module branch contains one module version, 1.1.2.1. This module version contains a hierarchical reference named “vault1” with a reference target of:

```
sync://granite:2647/Projects/Development/vault1
```

### Related Topics

ENOVIA Synchronicity Command Reference: upgrade

ENOVIA Synchronicity Command Reference: hcm migratetag

## Upgrading DesignSync Vaults

In DesignSync 5.0, you can create modules to manage your design data. To use these new capabilities with existing DesignSync data, you must upgrade your existing DesignSync vault hierarchies to modules with the `upgrade` command.

**Important:** Before you upgrade a legacy module, you must check in any modifications and release any locks.

The `upgrade` command uses the `sync_project.txt` file as a guide for upgrading the relevant configurations into the new module.

**Note:** If a design configuration exists for your vault directory but has not been defined in ProjectSync's `sync_project.txt` file, the design configuration is not propagated to the new module. You should verify that the configurations you want to upgrade have an entry in this file. To add new design configurations to ProjectSync's `sync_project.txt`, either hand edit the file or use ProjectSync's Configuration panel.

The command allows you to specify the name of the new module as well as a category path. For example, if the vault directory you want to upgrade is located on the server at some level below the `/Projects` area on the server, you may use the `-category` option in conjunction with the `-name` option on the `upgrade` command to define the same directory structure as your original project vault.

For example, if the original project is:

```
sync://granite:2647/Projects/ProjectA/one/two/ROM
```

Then you may use the following `upgrade` command line:

```
upgrade sync://granite:2647/Projects/ProjectA/one/two/ROM -
category ProjectA/one/two -name ROM
```

The URL of the upgraded module object would be:

```
sync://granite:2647/Modules/ProjectA/one/two/ROM
```

Custom access controls are not upgraded along with the DesignSync vault hierarchy. If custom access controls are desired for the new module, they can be added either before or after the module is upgraded. After an upgrade completes, the server's administrator is notified by email that the vault hierarchy was upgraded, and the custom access controls were not carried forward to the new module.

The original vaults and data remain unchanged and unaffected by the hcm upgrade procedure. However, during the upgrade, access controls restricting write access to the DesignSync vault hierarchy are added to prevent modifications to the original vaults. The access controls on the original vaults remain in place after the upgrade has completed to prevent accidental updates to the original vaults.

#### Notes:

- All ProjectSync notes associated with the vault directories will also be associated with the new module after the upgrade is complete. However, email subscriptions on the vault are not modified and must be upgraded manually by the users. Users currently subscribed to activity on the vault will receive an email informing them of the vault's upgrade to a module and the need to update their subscriptions. The upgrade process informs this subset of users by email.
- All ProjectSync notes associated with an object in the original vault hierarchy are also associated with the new module after the upgrade is complete. However, any email subscriptions associated with the original vault hierarchy are not modified and must be added manually by the users. If DesignSync is configured to create a RevisionControl note for upgrade, and email notifications is enabled, the upgrade process sends an email to all users currently subscribed to activity on the vault folder informing them of the vault's upgrade to a module. The users should then update their subscriptions.
- This upgrade occurs on the server only. Any existing workspaces will continue to point to the existing DesignSync vault. The new module resulting from the upgrade needs to be populated into a new workspace.

An upgrade log is made available on the server to allow the user to track the progress of the upgrade. This log remains after the upgrade is complete and is located at:

```
http://<host>:<port>/syncserver/upgrade/
upgrade_<category>_<name>.html
```

#### Notes:

## DesignSync Data Manager User's Guide

- All slashes in the `<category>` field will be replaced with underscores.
- If server customizations have been imported to a different server after a module upgrade, the upgrade log may not appear at the URL provided for monitoring the upgrade. If the URL is not found, you may still access the upgrade log from the ModuleUpgrade sub-directory. For more information on the ModuleUpgrade sub-directory, see the *ENOVIA Synchronicity Command Reference*: `upgrade` help.

The following types of objects are created as a result of performing an upgrade on a DesignSync vault directory:

### Module:

A module with the specified name and category path.

### Module branches:

- A default module branch is created and is assigned an immutable module branch tag of “Trunk”.
- For each configuration, a module branch is created from module version 1.1 and tagged with the configuration name as a mutable module branch tag.

### Module versions:

- Module version 1.1 is created to include all members matching the “Trunk:Latest” selector. If no files resolve to this selector, the module version will be empty. If a Trunk configuration is defined to be mapped to a different selector, the upgrade process issues a warning noting that the Trunk branch will contain the members matching the Trunk:Latest selector and not the selector defined in the `sync_project.txt` file.
- Each module branch is created with an initial module version, comprised of member versions that were part of the configuration.

### Module branch tags:

In addition to the mutable module branch tag that is created for each configuration name, other branch tags are created as follows. For each configuration that is exactly equal to the Latest on the vault's branch, a mutable module branch tag with the same name is added to the module branch created for the configuration.

For example, if your `sync_project.txt` file contains the entry “CONFIG Silver Main:Latest”, a module branch named “Silver” is created and the mutable module branch tag “Main” is added to the branch.

### Module version tags:

For each configuration that is exactly equal to a version tag on a vault's version, a mutable module version tag with the same name will be added to the first module version on the branch that was created for the configuration.

For example, if your `sync_project.txt` file contains the entry “CONFIG Gold Alpha”, a module branch named “Gold” is created and the mutable module version tag named “Alpha” is added to the first module version on the “Gold” branch.

#### Hierarchical References:

If your vault hierarchy contains a `sync_project.txt` file with REFERENCE statements that point to another vault directory, these REFERENCES are created in the module as hierarchical references.

By default, or if the `-maphrefs` command option was specified, the upgrade attempts to determine whether the target of the href has already been upgraded to a new module. If so, the href will point to the new module rather than the original target with the new module’s branch selector or module version selector recorded as the dynamic hrefmode value and the new module version’s number recorded as the static hrefmode value.

An href can only be mapped if the target’s configuration was converted to a module branch or module version. For example, if a target configuration was not defined in a `sync_project.txt` file in the original vault hierarchy, the target configuration is not transferred to the new module since the mapping process does not have a branch or version in which to place this href. An error is issued during the mapping process and the upgrade terminates without creating the new module.

If you want the mapping errors treated as warnings instead, use the `-nomaperror` option to transfer the original href. Mapping errors are issued if a href’s target was upgraded but the new module does not exist, if there is no matching branch or version for the href’s target configuration or if the href target’s server is not accessible.

**Tip:** If you plan on upgrading an entire design hierarchy, a vault hierarchy and any referenced vault hierarchies, you should do so in a bottom-up fashion, so that the REFERENCES will map to the newly upgraded module.

After the upgrade is complete, you may migrate any of the tags that did not get transferred to the new module with the `migratetag` command. Any existing selector list, except date selectors, can be specified with `migratetag` to create a new module branch later. The supported selectors, such as version tags, or branch selector can also be combined to create a new module branch.

See the **upgrade** or `migratetag` command topics in the *ENOVIA Synchronicity Command Reference* for more detailed information.

#### An Example of Upgrading a DesignSync Vault

The following scenario illustrates how an existing simple vault directory (a vault directory structure with no vault REFERENCES) is upgraded to a module.

1. Amy, the designer of an ALU design, alu8, has previously placed a design under revision control.

```
dss> setvault sync://granite:2647/Projects/alu8 /home/amy/Projects/myalu8
dss> ci -recursive -new -keep -comment "8-bit ALU" /home/amy/Projects/myalu8
dss> tag -recursive baseline /home/amy/Projects/myalu8
dss> mkbranch -recursive Golden -version Trunk
sync://granite:2647/Projects/alu8
```

2. Amy contacts her project manager, who creates a new project with an associated sync\_project.txt file on the ProjectSync server. Joe also creates a new configuration for the project representing the “Golden” branch.

The sync\_project.txt file on the server contains the following information for this project:

```
NAME alu8
DESCRIPTION Project alu8
OWNER amy
COMPONENT FALSE
CONFIG Golden Golden:Latest amy *
CONFIG_DESC Golden
```

3. Amy then continues to work on her design on the “Golden” branch. At this point, Amy feels that her design might be better managed using a module.
4. Amy contacts her project manager, who upgrades her vault directory to a module.

```
dss> hcm upgrade sync://granite:2647/Projects/alu8 -name
ALU8
```

As a result of the upgrade, a new module is created at the following location:

```
sync://granite:2647/Modules/ALU8
```

The new module contains two branches, Trunk and Golden, and module versions are created appropriately.

### Related Topics

ENOVIA Synchronicity Command Reference: upgrade

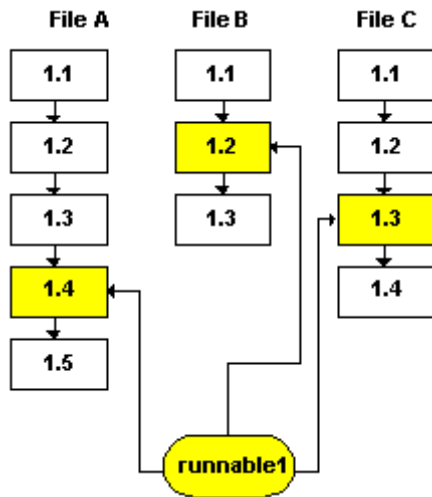
ENOVIA Synchronicity Command Reference: migratetag

## Managing Legacy Configurations and REFERENCES

### Managing Non-HCM Configurations

## What Is a Design Configuration?

A typical design is made up of many files. Each file can be worked on by an individual designer or several designers. At some point in time, all the files arrive at a stage where they work successfully together. For example, suppose that a design is made up of files A, B, and C. Version 1.4 of file A, version 1.2 of file B, and version 1.3 of file C work successfully together. These versions make up a "design configuration." To keep a record of this configuration, you can label them with an identifying tag -- "runnable1", for example.



If you or some other designer or group wishes to check out this configuration, you can perform the checkout using the tag. This one operation checks out all the file versions belonging to the configuration. You do not have to remember that you need version 1.4 of file A, version 1.2 of file B, and so on.

**Note:** This is a similar paradigm to the modules-based SITaR methodology. The Design Configuration Methodology mimics the native SITaR functionality in a non-modules based environment.

## Related Topics

Overview of SITaR Workflow

Creating a Design Configuration

Tracking Development with Design Configurations

Creating Releases

Fixed Tags and Movable Tags

## Creating a Design Configuration

To create a design configuration, first select the items (files or folders) you want to have in the configuration, then click the **Tag** button on the toolbar.

### Related Topics

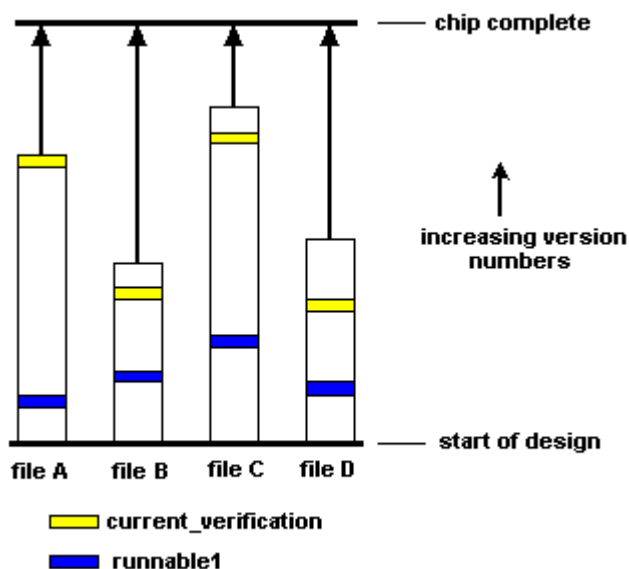
Tagging Versions and Branches

ENOVIA Synchronicity Command Reference: tag

### Tracking Development with Design Configurations

You can think of design configurations as snapshots along the path of development. Whenever development reaches a stable state, you take a snapshot of it by creating a design configuration. Each configuration should be identified by a name that makes sense. For example, when the design first reaches a runnable state, the design team can create a design configuration tagged "runnable1". Later, when development reaches a stage ready for verification, the team can create a design configuration tagged "current\_verification" and hand it off to the verification team. The verification engineers then check out the configuration and run verification procedures on it.

Notice in the figure below that the files making up the chip design vary in their version numbers; file C has had many more revisions than file B. Thus, configuration tags are necessary to specify which versions work together to form a functional chip.



### Mapping Configurations for Design Reuse

Designers can import modules from other design projects into their designs by mapping design configurations. In this way, designers can reuse modules from other projects.



Teams can develop modules independently, releasing modules on a schedule independent of the overall chip release schedule.

To map configurations, team members create a REFERENCE that maps a module to an implementation of the module using ProjectSync (in a sync\_project.txt file). See Using Vault REFERENCES for Design Reuse for a design example.

**Note:** When mapped configurations are configured in a mirroring environment, your mirrors should be set to always update, regardless of whether the specific configuration in the mirror is updated. For more information, see DesignSync Data Manager Administrator's Guide: Check tags for mirror update.

## Related Topics

### Using Vault REFERENCES for Design Reuse

#### Creating Releases

You create a release by creating a design configuration.

For internal releases, label the design configuration with tags such as "runnable1", "ready\_for\_simulation", "Bronze", "Silver", and so on. The tags should be meaningful. For example, "ready\_for\_simulation" can be a tag for a configuration that the design team wants to hand over to the simulation team. Note that tag names cannot contain spaces or periods ( . ).

When development and testing are complete, "Gold" is a typical tag for the configuration that represents the releasable product.

A file version can have more than one tag, since it can be a member of different configurations.

#### Using Tags

There are design configurations that you want to keep throughout the development process. For example, you may have a configuration that you are keeping as a possible starting point for a different line of development in the future. This tag is considered a "fixed" tag. The objects marked by the fixed tag move out of circulation. You do not use the fixed tag again in the project. You can designate a fixed tag as "immutable" when you create the tag. Immutable tags can never be moved. If you designate the tag as "mutable," it can be moved by a user with appropriate access rights, even though it is logically designated as a fixed tag.

On the other hand, there may be configurations that you use only temporarily -- for example, a configuration you use to show customers what a product looks like at the current state of development. Such a tag might be "currentDemo". A week later, you are no longer interested in that configuration, because you have achieved a more advanced

configuration to use for the demo. In such a case, you would tag the more recent design configuration with the already-used "currentDemo" tag. In effect, you have thrown away the older design configuration information (even though those versions still exist). The tag is considered a "movable" one.

The logical tag types fixed or movable are used to describe ways in which you can use design configuration tags, but contain no special attributes to identify them. The responsibility of maintaining them properly rests on the design team. The physical tag types, immutable, and mutable force a more strict level of control by designating whether the tag can be moved.

### Related Topics

Tagging Versions and Branches

ENOVIA Synchronicity Command Reference: tag

## Using Vault REFERENCES for Design Reuse

### Using Vault REFERENCES for Design Reuse

DesignSync eases development of a chip that relies on

- Reused modules from other projects, or
- Modules developed independently of the chip

Designers can release modules on one schedule, while the chip that uses them follows an entirely different release schedule.

The following scenario illustrates how a module is distributed and used by team members. The design projects are stored in server-side vault folders on a SyncServer. Team members use vault REFERENCES to import modules from other design projects into their own designs. A REFERENCE is a pointer you create in a ProjectSync project (in a `sync_project.txt` file) to map configurations.

**Note:** This design reuse flow shows how module clients and producers can share design modules. If your design team uses vault REFERENCES to share modules, your team needs to have a consistent methodology governing how clients and producers create and share information about design configurations.

This flow assumes a basic understanding of the DesignSync configuration management tasks; see *Using DesignSync: An Overview* to gain this understanding.

1. Amy, the designer of an ALU design, `alu8`, places the design under revision control:

```
dss> setvault sync://granite:2647/Projects/alu8
/home/amy/Projects/myalu8
```

```
dss> ci -recursive -new -keep -comment "8-bit ALU"
/home/amy/Projects/myalu8
```

2. Amy iteratively modifies and tests her design until she is satisfied that she has a clean version. She creates a "baseline" configuration:

```
dss> tag -recursive baseline /home/amy/Projects/myalu8
```

3. Amy informs her team coordinator that she has a stable version of her alu8 tagged "baseline" for her team members to import into their designs.
4. Joe is the team coordinator who is creating an area that will later become the beta release of the Asic\_zr2 project. Joe places the Asic\_zr2 project under revision control and applies the "beta" configuration tag:

```
dss> setvault sync://marble:2647/Projects/Asic_zr2
/home/joe/Asic_zr2
```

```
dss> ci -recursive -new -keep -comment "rev2 of zr proj"
/home/joe/Asic_zr2
```

```
dss> tag -recursive beta /home/joe/Asic_zr2
```

5. The Asic\_zr2 uses the alu8 module Amy is designing, as well as other modules being designed by other team members. In order to use these modules, Joe will create REFERENCES within subfolders of his Asic\_zr2 project to point to these modules. Joe uses ProjectSync to create a project and its associated **sync\_project.txt** file referencing Amy's module. The **sync\_project.txt** file contains a REFERENCE to Amy's alu8 module and a CONFIG mapping statement to ensure that team members import the correct version of the alu8. Joe could also create the **sync\_project.txt** file using a text editor.

**Note:** These steps require write access to the SyncServer vault. Usually the owner of the SyncServer process performs these steps.

- a. Within his Asic\_zr2 project on his marble:2647 SyncServer, Joe creates a placeholder folder that will contain a REFERENCE to Amy's alu8 project .

```
$ mkdir
<SYNC_DIR>/../syncdata/marble/2647/server_vault/Projects/
Asic_zr2/alu
```

The path to the vault folder shown here is a typical UNIX path to a vault folder. A vault folder path on a default Windows installation has a different structure, for example C:\syncdata\server\_vault\Projects\Asic\_zr2\alu.

**Note:** If Joe were creating the **sync\_project.txt** file by hand, he would create the file in this folder.

- b. Joe creates a top-level ProjectSync project and its associated **sync\_project.txt** file for the Asic\_zr2 project:

Joe selects the **Project Create** menu item in the left panel of the ProjectSync window and enters the following settings:

**Project Name:** Asic\_zr2

**Description:** The new version of the Asic\_zr chip.

**Owner:** Joe Michaels

**Purpose:** ◦ This project will contain files under revision control.

**Vault Path:** <Leave blank; Vault Path defaults to the project's vault.>

**Initial Configuration:** <Leave blank.>

- c. Joe creates a ProjectSync project within the alu folder of the Asic\_zr2 project. This step creates a **sync\_project.txt** file containing the REFERENCE to Amy's alu8:

Joe again selects the **Project Create** menu and enters the following settings:

**Project Name:** Asic\_zr2/alu

**Description:** References the alu8 developed by Amy

**Owner:** Joe Michaels

**Purpose:** ◦ This project will contain files under revision control.

**Vault Path:** sync://granite:2647/Projects/alu8

**Initial Configuration:** beta

**Note:** See the ProjectSync documentation for step-by-step instructions on creating projects.

Joe selects the **Create Project** button at the bottom of the ProjectSync Create New Project window. The Create Project Configuration window appears.

- d. Next, Joe maps the "beta" configuration used in his Asic\_zr2 project to the "baseline" configuration tag name Amy has applied internally to her alu8 design, using these settings:

**Project Name:** Asic\_zr2/alu

**Configuration Name:** beta

**Configuration Description:** Baseline version of alu8

**Owner:** Joe Michaels

**Vault Tag:** baseline

**Team Members:** <Joe selects Amy Jones, Paul Winslow, Zachary Jackson, and Dana Conti from the list of Available Users.>

Joe creates the configuration by selecting the **Create Configuration** button at the bottom of the ProjectSync Create Project Configuration window.

ProjectSync generates the following **sync\_project.txt** file in the <SYNC\_DIR>/../syncdata/ marble/2647/server\_vault/Projects/Asic\_zr2/alu folder:

```
NAME alu
REFERENCE sync://granite:2647/Projects/alu8
DESCRIPTION References the alu8 developed by Amy
OWNER joe
COMPONENT FALSE
CONFIG beta baseline joe amy paulw zach dana joe
CONFIG_DESC Baseline version of alu8
```

The CONFIG entry in the **sync\_project.txt** file maps the "beta" configuration -- the configuration name used by the Asic\_zr2 project that will import Amy's alu8 -- to the "baseline" tag. The COMPONENT FALSE entry refers to the Purpose field in the ProjectSync Create New Project window. This entry indicates that the project will be used to manage files under revision control and not merely as a tool for organizing ProjectSync notes. See the ProjectSync documentation for more information.

**To add additional CONFIG mapping statements**, Joe can select the **Configuration** menu item in the left panel of the ProjectSync window or hand-edit the **sync\_project.txt** file.

6. After the **sync\_project.txt** file is in place, Joe or any other team member on the Asic\_zr2 project can populate to get the Asic\_zr2 design, including the alu8 module's files:

```
dss> populate -recursive -get -version beta -dir
/home/joe/Asic_zr2
```

Joe has already made the vault association using the setvault command in an earlier step. When DesignSync encounters the Projects/Asic\_zr2/alu vault folder during the populate operation, it reads the **sync\_project.txt** file. The **sync\_project.txt** file directs it to populate with the "baseline" version of the alu8 files from the sync://granite:2647/Projects/alu8 vault folder.

### Mapping a Configuration to a Selector List

DesignSync also allows mapping of a configuration to more than one selector. To map a configuration name to a selector list, Joe can use ProjectSync to create a configuration specifying A as the **Configuration Name** and gold,silver,bronze as the **Vault Tag**. For example:

```
CONFIG A gold,silver,bronze
```

**Note:** Selector names should be separated by commas, with no white space between names.

Then Joe can populate his work area with configuration A, for example:

```
dssc> scd /home/joe/Asic_rz2

dssc> setselector A

dssc> populate -get -recursive
```

DesignSync sets the persistent selector for the Asic\_zr2 folder to gold,silver,bronze. When Joe checks in a file, DesignSync resolves the gold selector to a version and checks in that version. If the gold selector in the list does not resolve to a version, then DesignSync looks at the next selector in the list.

### Related Topics

[What Is a Design Configuration?](#)

## Mapping Configurations for Design Reuse

## REFERENCES and Revision Control Commands

## REFERENCE Chaining

### REFERENCES and Revision Control Commands

The following details about how DesignSync revision control commands handle REFERENCES will help your design team use REFERENCES in `sync_project.txt` files to import modules. For an overview of how your team can use REFERENCES to import modules, see [Using Vault REFERENCES for Design Reuse](#).

#### REFERENCES and the populate Command

When you populate a configuration-mapped folder (either directly or through a recursive populate operation) and the selector you specify is mapped, DesignSync sets the persistent selector list for that folder to the mapped value.

In the example in [Using Vault REFERENCES for Design Reuse](#), the alu folder's persistent selector list is set to "baseline" even though the configuration specified during the populate operation is "beta". Storing the mapped configuration is a performance optimization. If a change is made to the configuration mapping -- for example, if the beta tag for the alu design is changed to map to "newbaseline" instead of "baseline" -- you must re-populate the work area using the non-mapped configuration (beta). Re-populating updates the persistent selector list for the alu folder from "baseline" to "newbaseline".

#### REFERENCES and the tag Command

### How DesignSync Applies a Tag to a Configuration-Mapped Folder

When you apply a tag recursively to a configuration that has been mapped using a REFERENCE, DesignSync compares the version specified for the tag operation to the configuration defined in the REFERENCE.

**Note:** You specify a version by using the **-version** option of the **tag** command or the **Version Selector** field of the DesignSync **Tag** dialog. If no version is specified for the tag operation used, DesignSync uses the selector for the folder.

If the configuration name matches the version specified for the tag operation, DesignSync:

- Skips the referenced folder in its tagging operation and displays a message about the skip. By skipping referenced vaults, the tagging operation respects module boundaries so that only module owners can tag modules.

## DesignSync Data Manager User's Guide

- Creates a CONFIG statement to the REFERENCE in `sync_project.txt` of the mapped vault folder.

The newly created CONFIG statement maps the configuration named by the tag command to the resolved configuration of the referenced vault. The new CONFIG statement in effect, adds a new "tag" to the referenced vault. It also allows DesignSync to skip the tagging of referenced vaults but retain the referenced vault for subsequent populate commands.

For example, given the configuration mapping example in [Using Vault REFERENCES for Design Reuse](#), a team member, Zach, wants to create an "alpha" configuration of the entire `Asic_zr2` project. He sets the selector to the beta configuration, populates his work area with that configuration, checks in his changes, and then recursively tags his entire `Asic_zr2` project work area as "alpha":

```
dssc> scd Asic_zr2
dssc> setvault sync://marble:2647/Projects/Asic_zr2
dssc> setselector beta .
dssc> populate -recursive -get
dssc> co -lock -comment "adding power-up state vector"
top/decoder/decoder.v
dssc> ci -keep -comment "added power-up state vector"
/home/zach/Asic_zr2/top/decoder/decoder.v
dssc> tag -rec alpha .
```

When the recursive `tag` command reaches the referenced alu module, DesignSync:

- Checks the folder's `sync_project.txt` file. In our example, this file contains the configuration statement:

```
CONFIG beta baseline joe amy paulw zach dana joe
```

- Since the configuration name (beta) matches the version specified for the tag, DesignSync skips the referenced folder in its tagging operation and displays a message about the skip. For example:

```
Tagging: Asic_zr2/alu : Skipping (Added Config "alpha"
mapping to "baseline")
```

- Adds the following line to the REFERENCE in the `sync_project.txt` file for the vault folder. The REFERENCE for the example now contains:

```
NAME: ALU
REFERENCE sync://granite:2647/Projects/alu8
DESCRIPTION References the alu8 developed by Amy
OWNER joe
```



```
COMPONENT FALSE
CONFIG beta baseline joe amy paulw zach dana joe
CONFIG_DESC Baseline version of alu8
CONFIG alpha baseline zach *
```

When Zach populates the alpha version of his Asic\_zr2 project, he picks up Amy's baseline alu8 files.

If the configuration name in the REFERENCE does not match the version specified for the tag operation, DesignSync descends into the mapped vault folder and applies the tag to the files there.

If the configuration name matches the version specified for the tag operation but that configuration name maps to a dynamic tag (for example, CONFIG beta Trunk zach \*), DesignSync descends into the mapped vault folder and applies the tag to the files there.

If the REFERENCE configuration that maps to the referenced vault's configuration does not exist, DesignSync does not generate the new CONFIG statement. For example, the REFERENCE in the alu folder contains the following CONFIG statement:

```
CONFIG beta baseline joe amy paulw zach dana joe
```

If the beta configuration does not exist, no tagging is required.

If the tag command does not include the `-version` option, the newly added REFERENCE maps the Trunk configuration to the configuration of the referenced vault, in this case:

```
CONFIG Trunk baseline zach *
```

### How DesignSync Deletes a Tag from a Configuration-Mapped Folder

If the tag specified for deletion matches the configuration name in the `sync_project.txt` file of a mapped vault folder, DesignSync deletes the CONFIG statement. For example, suppose Zach decides to delete the alpha configuration he created (in our example):

```
dssc> tag -delete alpha -recursive .
```

When the tag delete operation encounters the configuration-mapped vault folder, (alu), DesignSync compares the alpha tag to the configuration name(s) in the REFERENCE and finds that they match. DesignSync deletes the CONFIG statement:

```
CONFIG alpha baseline zach *
```

## DesignSync Data Manager User's Guide

leaving the original configuration definition:

```
CONFIG beta baseline joe amy paulw zach dana joe
```

If no alpha configuration is defined, DesignSync descends into the alu folder and removes the alpha tag from the files there.

### How DesignSync Replaces a Tag on a Configuration-Mapped Folder

When a tag replacement operation reaches a vault folder that is mapped to another configuration (with a CONFIG statement), DesignSync:

- Compares the tag specified for replacement with the configuration names defined in CONFIG statements in the `sync_project.txt` file for the mapped folder.
- If the names match, DesignSync replaces the existing CONFIG statement with a new one that maps the configuration name to the new configuration tag.

To continue the example of the `Asic_zr2` project, suppose Amy, the alu designer, updates her design files. She then checks them in to the `alu8` vault folder and tags them `newbaseline`. The `Asic_rz2` project leader (Joe) wants the to include Amy's latest files in alu folder for the project. He modifies the `sync_project.txt` file for the alu folder to change the CONFIG statement to map to the "newbaseline" configuration tag instead of "baseline".

The CONFIG statement now includes:

```
CONFIG beta newbaseline joe amy paulw zach dana joe
CONFIG alpha baseline zach *
```

Suppose that Zach wants the alpha configuration for the alu design to map to Amy's latest files (tagged "newbaseline") as well. Zach sets the selector for his work area to "beta" and populates the work area, which fetches the beta configuration. He then replaces the alpha tag:

```
dssc> tag -recursive -replace alpha .
```

When the tag delete operation encounters the configuration-mapped vault folder, (`alu`), DesignSync determines that the alpha tag matches the alpha configuration name in the CONFIG statement of the `sync_project.txt` file. DesignSync replaces the old configuration definition:

```
CONFIG alpha baseline zach *
```

with the new alpha configuration definition:

```
CONFIG alpha newbaseline zach *
```

## Related Topics

Using Vault REFERENCES for Design Reuse

REFERENCE Chaining

### REFERENCE Chaining

Team members use vault REFERENCES to import modules from other design projects into their own designs. A REFERENCE is a pointer you create in a ProjectSync project (in a `sync_project.txt` file) to map configurations.

**Note:** REFERENCES are not supported if your revision control commands use selector lists.

If a REFERENCE points to another project that also has a REFERENCE in its **sync\_project.txt** file, DesignSync follows that REFERENCE, and so on until an actual vault is located. This is called **REFERENCE chaining**. The maximum number of REFERENCES that DesignSync follows is 10.

DesignSync attempts to limit the number of times it traverses a REFERENCE chain to once per operation sequence (for example, checking in or out an entire design hierarchy). However, the greater the number of REFERENCES in the chain, the greater the performance degradation.

### Using REFERENCE Chaining to Move a Vault Folder

To change the location of a vault folder, you can chain REFERENCES:

1. Move the vault.

See [Moving Vaults](#) for instructions.

2. Leave a REFERENCE pointing to the new vault location.
3. At some later date, remove the REFERENCE.

Until that time, users accessing the old location will be making an extra server hop to access the vault.

## Collections

### Collections Overview

Collections are groups of files that together define a design object. For example, a schematic may consist of dozens or even hundreds of files within any number of folders.

You operate on the design object as a single entity while letting your design tools manage the object at the file level.

Within DesignSync, a collection is a single revision-controllable object; you do not perform revision-control operations directly on individual collection files. When you check out a collection object, DesignSync checks out each file contained in the collection. If you then edit one of the files, you check back in the collection object, not just the changed file. This creates a new version of the collection object and each of the collection members. DesignSync recognizes that only one file in the collection has been modified and stores only the required change information.

DesignSync supports the following collection types:

- Cadence Design Systems cell view collection, as part of DesignSync's general support for Cadence design libraries. See *Cadence Design Objects Overview* for more information.
- Synopsys Custom Compiler cell view collection, as part of DesignSync's general support for Synopsys design libraries. For more information, see the *ENOVIA Synchronicity DesignSync Data Manager for Custom Designer User's Guide*.
- Custom generic collection, a collection object defined by a Custom Type Package (CTP). See *Custom Type Package Collections Overview* for more information.

## Displaying Collections

After your DesignSync administrator or project leader has enabled DesignSync to recognize a collection type, you can use DesignSync to display information about the collection and its members.

As with any other object type, you can use DesignSync **List View** to display information about the collection object, including revision control information. To get the same information from a command shell, use the **ls** command.

### Displaying a Collection's Members

#### From List View:

1. From **List View**, select the collection object.
2. Select **File => Properties => Collection**.

You cannot edit the listing to add or remove members. What determines the members of a collection depends on the type of collection object. For example, a function available from the Cadence software determines the members of a Cadence view and a Custom Type Package (CTP) determines the members of a custom collection.

#### Note:

- To have DesignSync recompute what files are members of the collection, click **Update Members**. For example, if you have made changes to the contents of your collection, you can click **Update Members** to see a list of current members.
- If DesignSync cannot determine the members of the collection object, an error message is displayed at the top of the member list. An error is also displayed when you use the **Is** command to display a collection.

### From a DesignSync command shell:

Use the **url members** command and specify the collection object.

### Identifying the Collection to which a Member Object Belongs

#### From List View:

1. From **List View**, select the member object.
2. Select **File => Properties => Collection Member**.

#### From a command shell:

1. Change directory to the folder containing the member object.
2. Use the DesignSync **Is** command with its **-report OX** option. Using this option displays each object's object type and owner (the collection to which the object belongs).

#### Related Topics

[Cadence Design Objects Overview](#)

[ENOVIA Synchronicity Command Reference: Is](#)

[ENOVIA Synchronicity Command Reference: url members](#)

[Custom Type Package Collection Overview](#)

## Cadence Collections

### Cadence Design Objects Overview

#### Note:

Cadence library/cell/cell view recognition is not supported on Windows platforms.

Many Cadence Design Systems (CDS) tools operate on design data that is organized as libraries, cells, and cell views.






- A **library** is a collection of design objects, called cells. For example, you might have IC libraries from different vendors from which you build your design. On your file system, a library is a folder that contains technology files and a folder for each cell in the library.
- A **cell** is an individual building block of a chip or system. For example, a TTL library might have cells called AND2, AND3, and NOR2. A cell has one or more cell views that are different representations of the cell. On your file system, a cell is a folder under the library folder.
- A **cell view** is a specific representation of a cell. For example, a NAND2 cell may have four cell views: Verilog description, symbol, schematic, and layout. On your file system, a cell view is a folder containing several files that together define the cell view.

DesignSync chooses the files that make up a Cadence cell view collection based on the type of the cell view, as specified in the Cadence data registry file. To display collection members from **List View**, select the cell view folder. DesignSync displays the folder's contents and identifies members in the Type column.

DesignSync 4.x clients can work with Cadence cell view data that was created by DesignSync 5.0+ clients in compatibility mode. When the DesignSync 5.0+ client is in this mode, the cell view folder contains a housekeeping file called **.<view>.sync.cds.synccmd**, as well as the cell view files used in the design.

The **.<view>.sync.cds.synccmd** file is used by DesignSync 4.x clients to manage the collection object. DesignSync 5.0+ clients do not use this file, so compatibility mode creates the housekeeping file in order for DesignSync 4.x clients to correctly recognize the DesignSync 5.0+ created views.

The following example shows the contents of the layout view of the cell called mux2 on a DesignSync client when DesignSync is in compatibility mode:

| file:///home/tgoss/Cadence/cell_design/master/mux2/layout                                                    |                     |                  |      |        |        |  |
|--------------------------------------------------------------------------------------------------------------|---------------------|------------------|------|--------|--------|--|
| Name                                                                                                         | Type                | Modified         | Size | Status | Locker |  |
|  .layout.sync.cds.synccmd | Cadence View Member | 06/14/2000 16:28 | 95   | -      | -      |  |
|  layout.cdb               | Cadence View Member | 06/14/2000 16:28 | 5664 | -      | -      |  |
|  master.tag               | Cadence View Member | 06/14/2000 16:28 | 39   | -      | -      |  |
|  pc.db                    | Cadence View Member | 06/14/2000 16:28 | 55   | -      | -      |  |
|  prop.xx                  | Cadence View Member | 06/14/2000 16:28 | 248  | -      | -      |  |

**Note:** If the Cadence-supplied function cannot determine the collection members, a **CAI** error message is displayed at the top of the member list. The same error is displayed when you use the DesignSync **ls** command to display a collection. See How DesignSync Recognizes Cadence Data for details.

You never operate directly on the files that comprise a cell view; your design tools manage the files. DesignSync manages Cadence cell views as collection objects -- groups of files managed as a single revision-controlled object. By default, DesignSync does not perform revision control on any files in a cell view folder that are not members of a Cadence collection. You can, however, explicitly select non-member files for revision control (see Managing Non-Collection Objects).

In addition to DesignSync's support for Cadence design objects, ENOVIA Synchronicity's DesignSync® DFII product integrates many of DesignSync's design management capabilities directly into Cadence's Design Framework II (DFII) environment. See DesignSync Data Manager DFII User's Guide for more information.

**Note: Do not use the DesignSync reference state when working with Cadence data unless it is a locked reference (which you can use when regenerating design data). Cadence object recognition in DesignSync and DesignSync DFII will not work properly when files are in the reference state.**

#### Related Topics

[Enabling Cadence Object Recognition](#)

[How DesignSync Recognizes Cadence Data](#)

[How DesignSync Manages Cadence Objects](#)

[DesignSync DFII Help: DesignSync DFII Design Management Overview](#)

## Enabling Cadence Object Recognition

Support for Cadence objects is enabled when you configure DesignSync DFII during the installation of DesignSync software.

To enable (or disable) recognition after installation, use the Third Party Integration panel of the SyncAdmin tool. **Note:** A COP (Cadence Object Processing) license must be installed in order to use DesignSync to manage these objects. See The License File for an example.

#### Related Topics

[Cadence Design Objects Overview](#)

[SyncAdmin Help: Third Party Integration Options](#)

## How DesignSync Recognizes Cadence Data

DesignSync uses a combination of methods to determine if a folder (directory) is a Cadence library. Once a library is recognized, any folder within the library is a cell, and any folder within the cell is potentially a cell view. DesignSync then uses Cadence-supplied routines to determine the co-managed set of files that constitute a cell view.

DesignSync determines if a folder is a library as follows:

1. DesignSync constructs a list of libraries defined in `cds.lib` files. If a folder matches a library definition in this list, then the folder is a library.

DesignSync calls Cadence-supplied routines (from the Cadence CAI library) to locate `cds.lib` files in your Cadence search path. DesignSync then augments that list with `cds.lib` files found in the search path defined using the **addcdslib** command.

**Note:**

The directory from which a tool is invoked is included in the Cadence search path. Therefore, you should invoke DesignSync from the same directory as you invoke your Cadence DFII applications to ensure that the same libraries are recognized.

2. If a folder has a `cdsinfo.tag` file containing an uncommented `CDSLIBRARY` property, then DesignSync recognizes the folder as a Cadence library.

**Note:**

- DesignSync uses this method of library recognition only when `cds.lib`-based recognition fails to identify a folder as a library.
- The `CDSLIBRARY` property is only valid within a library-level `cdsinfo.tag` file and is not inherited from any other `cdsinfo.tag` found in the Cadence search path.
- A physical copy of `cdsinfo.tag` must remain in the folder for library recognition to function properly. Therefore, any of the following situations could cause Cadence object recognition to fail:
  - You perform an operation that removes `cdsinfo.tag`, such as populating with the force option a configuration that does not include `cdsinfo.tag`.
  - You associate a vault with a folder below the library level in your design hierarchy. DesignSync does not recognize the cell views as part of your Cadence library because the `cdsinfo.tag` file is at the library level, above where the vault is set.
  - You check Cadence design objects into a vault that is mirrored before checking in the `cdsinfo.tag` file. The cell views are not recognized in the mirror directory.



- You specify the reference or locked reference state while performing a revision-control operation on `cdsinfo.tag`. Note that the DesignSync `populate` command never leaves `cdsinfo.tag` in the reference or locked reference state.

Because DesignSync relies on Cadence routines to identify Cadence libraries and to determine a cell view's co-managed set of files, you must have the Cadence executables directories in your `PATH` environment variable (so that DesignSync can locate the `cds_root` executable). For example, you might have the following lines in your `.cshrc` file:

```
# The Cadence tools installation is /usr/cds
setenv PATH /usr/cds/tools/bin:/usr/cds/tools/dfII/bin:${PATH}
```

Any errors reported by the Cadence data registry (for example, if you have errors in your `cds.lib` file) are reported to DesignSync as **CAI** errors. These errors appear either in the operating-system shell from which you invoked DesignSync or in your DesignSync window, depending on when the error was encountered.

#### Related Topics

[Cadence Design Objects Overview](#)

[How DesignSync Manages Cadence Objects](#)

[ENOVIA Synchronicity Command Reference: addcdslib](#)

[DesignSync DFII Help: DesignSync DFII Design Management Overview](#)







## How DesignSync Manages Cadence Objects


In most ways, DesignSync treats Cadence objects like any other object under revision control. However, DesignSync depicts Cadence libraries and their contents with Cadence-specific icons and terminology:

- When you click on a directory containing a Cadence library in the Tree View, the List View shows the library folder icon and the library name. **Cadence Library** is listed as the Type.
- When you click on a library name in the Tree View, the List View shows the files in the library, cell folder icons, and the cell names. **Cadence Cell** is listed as the Type of the cell folders.
- When you click on a cell name in the Tree View, the List View shows the cell view icon and the collection-object name assigned to the cell view folder. The Type column shows Cadence View for the collection object.

Because DesignSync regards each cell view as a single revision-controllable object, DesignSync assigns a collection-object name to each cell view and its contents. DesignSync uses the collection-object name to manage the cell view folder and its contents. This name takes the form **<name>.sync.cds**, where **<name>** corresponds to the name of the cell view folder.

The following illustration shows how DesignSync displays the contents of the cell **mux2** in the Cadence library **master** in the List View.

| Name ▾                                                                                               | Type                | Member of | Relative Path |
|------------------------------------------------------------------------------------------------------|---------------------|-----------|---------------|
|  layout             | Cadence View Folder |           |               |
|  layout.sync.cds    | Cadence View        |           |               |
|  schematic          | Cadence View Folder |           |               |
|  schematic.sync.cds | Cadence View        |           |               |
|  symbol             | Cadence View Folder |           |               |
|  symbol.sync.cds    | Cadence View        |           |               |



**Note:** If the cell view icon is a red **X** instead of the **CDS** symbol, then the data registry did not run properly. See [How DesignSync Recognizes Cadence Data](#) for details.

### Operating on Cadence Data

You can use the DesignSync graphical interface to operate on Cadence data. For instance, to check out or check in view objects. If you specify a collection member as the object to be operated on, DesignSync skips the object and warns that the object is not versionable. If DesignSync attempts to operate on a collection member specified implicitly (through the use of wildcards or a recursive operation), DesignSync silently skips the object. You can change this behavior by using the SyncAdmin Map operations on collection members to owner setting. If you select this setting and DesignSync attempts to operate on a collection member during a revision control operation, DesignSync determines the member's owner collection and operates on the collection as a whole.

### Related Topics

[Cadence Design Objects Overview](#)

## Managing Non-Collection Objects

DesignSync does not perform recursive operations on objects in a cell view folder that are not members of a Cadence collection. However, you can view these non-member objects and perform other operations on them using DesignSync.

For example, if you specify a recursive check out of a cell, DesignSync checks out the view objects but does not traverse into the cell view folders and check out non-member files. However, you can use DesignSync to navigate into the cell view folder and check out non-collection members that it contains. You also can list non-member files in cell view folders, delete them, get their properties, and so on. This behavior is similar to Library Manager's approach to also-managed files.

You can change this behavior using a setting in your registry. If you want your revision-control operations to traverse into cell view folders and act on non-collection members by default, edit your registry file as described in *DesignSync Data Manager Administrator's Guide: Vendor Objects Registry Settings*.

#### Related Topics

Cadence Design Objects Overview

Collections Overview

*DesignSync Data Manager Administrator's Guide: Registry Settings for Vendor Objects*

## Custom Type Package Collections

### Custom Type Package Collections Overview

The Custom Type System (CTS) is a programming interface used to customize DesignSync to manage your unique design data. Using this interface, you can define a Custom Type Package (CTP), which is a generic collection or a group of data files that you want DesignSync to treat as an abstract object. Then you can use DesignSync to check in, check out, and tag this abstract object, called a custom generic collection, as a single object. DesignSync safeguards your data by preventing users from checking in the constituent parts of the custom collection. Instead, users have to operate on the collection as a whole.

#### Note:

Do not use the DesignSync reference state when working with custom generic collections.

#### Enabling CTP Object Recognition

To enable DesignSync to recognize a CTP collection object, you install the CTP within the DesignSync custom hierarchy. When users invoke a DesignSync client, the DesignSync Custom Type System registers the CTP so that revision control operations recognize the collection types defined in your CTP. For more information, see *Installing Custom Type Packages in the Custom Type System Programming Guide*.

## How DesignSync Recognizes CTP Data

DesignSync recognizes CTP data when you install the CTP within the DesignSync custom hierarchy. After installation, when users invoke a DesignSync client, the DesignSync Custom Type System registers the CTP so that revision control operations recognize the collection types defined in your CTP.

DesignSync displays the following information about custom collection objects:

- CTP custom collection objects have names follow the form: **<object>.sgc.<collectiontype>**, for example, `symbol.sgc.mytool`. The `.sgc` extension indicates the object is a custom generic object. If an icon for the custom collection object type has been defined in the CTP type catalog, List View displays the icon.
- Each object of a custom type has its object type displayed in the **Type** column of both List View and the output of the **ls -report OX** command. If an icon for the custom collection object type has been defined in the CTP type catalog, List View displays the icon.

Here is an example display in DesignSync List View:

| Name         | Result                                  | Type              | Size    | Modified            |
|--------------|-----------------------------------------|-------------------|---------|---------------------|
| README       | ✓ Success - New version: 1.1 on New ... | File              | 592     | 07/16/2004 11:21:17 |
| a.html       |                                         | a Test Member     | 0       | 07/16/2004 10:15:10 |
| a.prop       | ✓ Success - New version: 1.1 on New ... | File              | 0       | 07/16/2004 10:15:10 |
| a.sgc.tst    | ✓ Success - New version: 1.1 on New ... | a Test collection | 2 files |                     |
| a.txt        |                                         | a Test Member     | 0       | 07/16/2004 10:15:10 |
| b.prop       | ✓ Success - New version: 1.1 on New ... | File              | 0       | 07/16/2004 10:15:10 |
| b.txt        | ✓ Success - New version: 1.1 on New ... | File              | 0       | 07/16/2004 10:15:10 |
| c.html       | ✓ Success - New version: 1.1 on New ... | File              | 0       | 07/16/2004 10:15:10 |
| d.html       |                                         | d Test Member     | 0       | 07/16/2004 10:15:10 |
| d.prop       | ✓ Success - New version: 1.1 on New ... | File              | 0       | 07/16/2004 10:15:10 |
| d.sgc.tst    | ✓ Success - New version: 1.1 on New ... | d Test collection | 2 files |                     |
| d.txt        |                                         | d Test Member     | 0       | 07/16/2004 10:15:10 |
| f.html       |                                         | f Test Member     | 0       | 07/16/2004 10:15:10 |
| f.notamember | ✓ Success - New version: 1.1 on New ... | File              | 0       | 07/16/2004 10:15:10 |
| f.prop       | ✓ Success - New version: 1.1 on New ... | File              | 0       | 07/16/2004 10:15:10 |

If DesignSync detects a problem with an installed CTP, DesignSync prevents the check-in of the CTP data objects. In this case, DesignSync displays an error message indicating why the check-in operation failed. If you encounter this type of error when attempting to check in CTP data, contact your CTP developer. For more information on CTP data recognition, see DesignSync Recognition of Custom Type Packages in the DesignSync Custom Type System Programming Guide.

## Integration with ENOVIA Program Central

### Using the ENOVIA Semiconductor Accelerator for DesignSync Central

ENOVIA Semiconductor Accelerator for Team Collaboration is a complete revision management solution. It provides issue tracking and product life cycle management capabilities for the DesignSync data using ENOVIA Live Collaboration.

The DSFA architecture contained within Team Collaboration enables communication between DesignSync and ENOVIA Live Collaboration allowing virtual teams to share PLM documents or DesignSync data across the platforms, and providing management reports and collaborative discussion threads that link to DesignSync data.

If you have been using ProjectSync, you can convert your ProjectSync notes into Team Collaboration issues and discussions. For more information, see the *ENOVIA DesignSync Central Migration Toolkit Guide*.

### Using the ENOVIA Semiconductor Accelerator for IP Management

DesignSync Central is a complete IP management solution. It provides linked IP management from the IP stored in DesignSync to the IP management infrastructure in ENOVIA Library Central.

The DSFA architecture contained within IP Management enables two-way communication between DesignSync and ENOVIA Live Collaboration allowing trackable design reuse, and cross-team collaboration.

If you have been using IP Gear, you can convert your IP into ENOVIA Semiconductor Accelerator for IP Management IP. For more information, see the *ENOVIA Synchronicity IP Management Migration Toolkit Guide*.

# User Interface

## Performing GUI operations

### Selecting Objects

There are several ways to select objects, such as files or folders:

- **Graphically**

In the DesignSync window, select an object by left-clicking on it. To select a group of objects, hold down the **Ctrl** key and left-click the individual objects. To select a range of objects, left-click on the object at the beginning of the range, then hold down the **Shift** key and left-click on the object at the end of the range.

- **Command line**

Most commands that operate on objects require you to specify the objects. In some cases the objects do not have to be specified -- for example, in the following recursive checkin, which checks in all files in the `myasic` directory and in any directories under it:

```
ci -recursive /users/joe/myasic
```

You can also create DesignSync select lists to operate on objects. See the ENOVIA Synchronicity Command Reference Help: select command for more information.

### Going to a Location

There are two different methods to specify a location:

1. Selecting **Go** from the menu.
2. Entering the folder's path or URL in the Location Bar.

#### Go Menu Option

To go directly to a folder, select **Go =>Go to Location** from the menu. In the Go to Location dialog box, enter the folder's path or URL in the location field. Click **Browse Local** to navigate to a file or folder on your local machine.

View and select previously visited locations from the location history by clicking the pull-down arrow to the right of the text field.

#### Location Bar Option

To go directly to a folder (client-side or server-side), enter the folder's path or URL in the Location Bar. You can also view and select previously visited locations from the location history by clicking the pull-down arrow to the right of the text field.

DesignSync updates the Location Bar as you navigate using the Tree View or List View. However, these locations are not stored in the location history unless you press Enter while a location is displayed in the Location Bar.

DesignSync does not update the Location Bar when you navigate from the Command Bar using the **scd** (or **cd**) command. The Location Bar is always synchronized with the Tree and List Views, and navigating from the Command Bar does not update the Tree and List Views.

### Related Topics

Go Menu

ENOVIA Synchronicity Command Reference: scd

ENOVIA Synchronicity Command Reference: cd)

## Navigating the Tree View

The Tree view provides two views for working with modules that reside on your local client system:

- The familiar Folder Explorer.
- The Module Explorer, located in the Modules Roots folder that is displayed when My Computer is expanded.

Navigate between these two views by clicking a module (or an item in the module's directory structure) to highlight it and indicating the target view from the Go menu.

| <b><i>Module location<br/>Tree view</i></b> | <b><i>Command</i></b> | <b><i>Resulting Location in Tree View</i></b>                |
|---------------------------------------------|-----------------------|--------------------------------------------------------------|
| Folder Explorer                             | Go to Module Explorer | Module instance within a module root directory.              |
| Module Explorer                             | Go to Folder Explorer | The module base directory that contains the module instance. |

If more than one module satisfies the navigation command, the Select Module Context dialog box displays with a drop down list of possible target modules.

### Related Topics

Module Explorer

Tree View Pane

List View Pane

## Adding, Editing, and Organizing Bookmarks

### Adding a Bookmark

To bookmark an object:

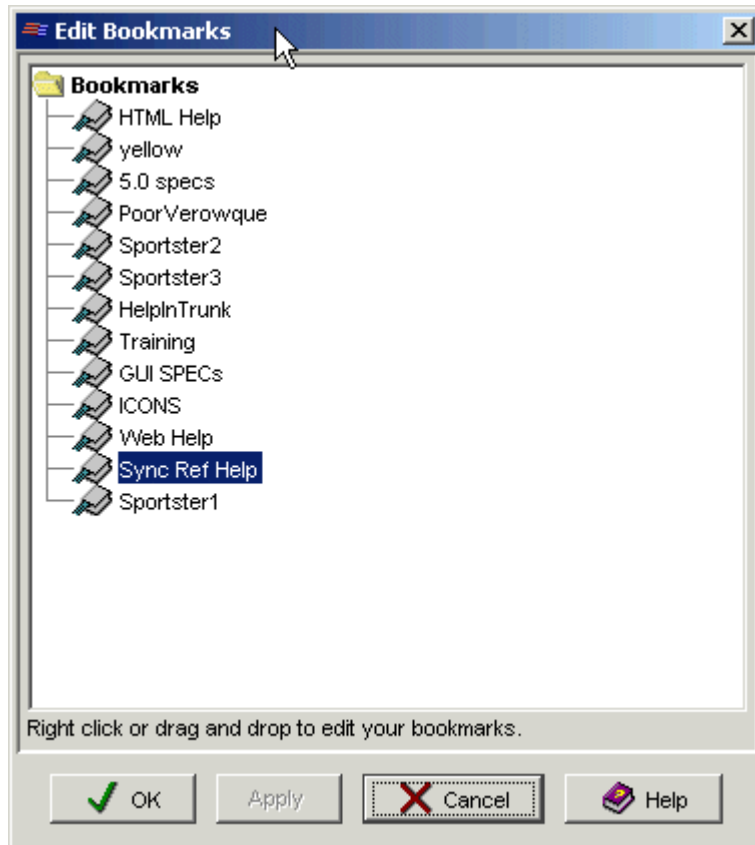
- Select the object and then select **Bookmarks =>Add Bookmark**.
- Select the object and then click **Ctrl+B**

With either of these methods, your current location is added to the bookmark list. The object's bookmark is represented by a folder with a bookmark on it. The bookmark persists between invocations of DesignSync.

### Editing and Organizing Bookmarks

To edit or organize your bookmarks, select from the Main menu, **Bookmarks =>Edit Bookmarks**. This displays the Edit Bookmarks dialog box.





To change the order of your bookmarks, select one or more bookmarks and then drag them to their new position.

To modify a bookmark, highlight a bookmark and click the right mouse button. This displays a context menu with the following choices:

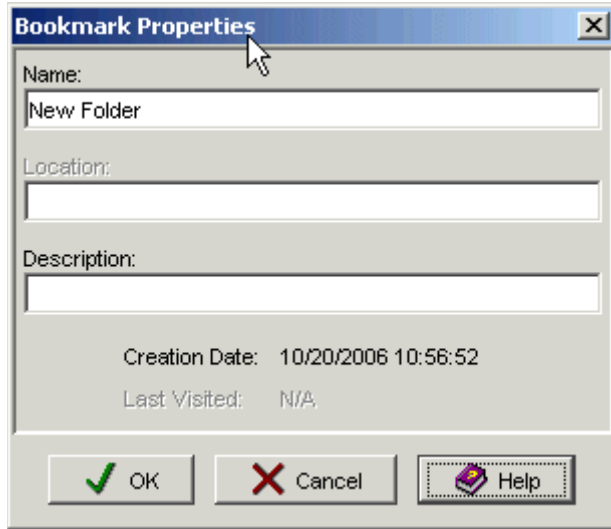
- **Visit** takes you to the bookmark's location. You can also visit a bookmark by double clicking on it.
- **New Bookmark** creates a new bookmark. It then displays the bookmark properties dialog box so you can define the bookmark.
- **New Folder** creates a new folder for bookmarks. It then displays the bookmark properties dialog box so you can define the folder.
- **Rename** allows you to change the name of the bookmark or folder. You can also rename a bookmark or folder by selecting it and pressing **F2**. Note that renaming a bookmark changes its name, but not its location.
- **Delete** deletes the selected bookmarks or folders. You can also delete bookmarks and folders by pressing the Delete key.
- **Properties** displays the bookmark properties dialog box, so you can change the bookmark's or folder's properties.

#### Related Topics

[Adding a Vault to Bookmarks](#)

## Defining and Modifying Bookmark Properties

You can modify the properties of a bookmark or bookmark folder with the bookmark properties dialog box.



This dialog box allows you to set the following properties:

- The **name** of a bookmark or folder is the text that appears in the bookmark menu.
- The **location** of a bookmark is the location that the bookmark goes to when you select it from the bookmark menu. Bookmark folders do not have a location.
- The **description** of a bookmark appears in the status bar when you move your mouse over the bookmark's menu choice.
- The **creation date** is the date that the bookmark or folder was created. Bookmarks created in earlier versions of DesignSync do not have a creation date. You cannot modify the creation date.
- The **last visited** date is the last date that the bookmark was visited by selecting it from the bookmark menu. Bookmark folders do not have a last visited date; you cannot modify the last visited date.

For example, you might create a bookmark with the following properties:

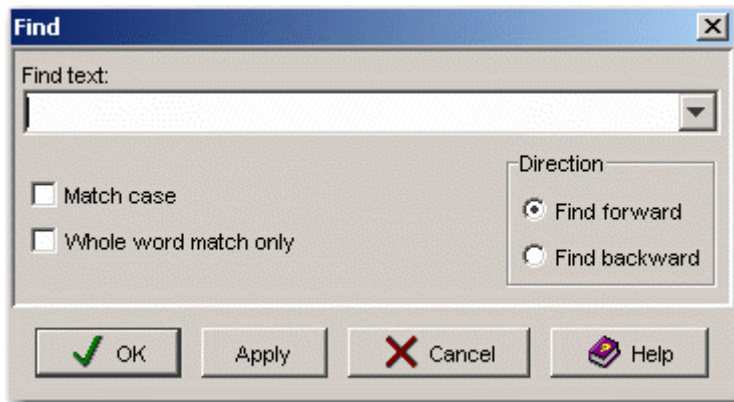
|                    |                                       |
|--------------------|---------------------------------------|
| <b>Name</b>        | ALU75 Workspace                       |
| <b>Location</b>    | file://home/aurora/Projects/ALU75/src |
| <b>Description</b> | Workspace for the ALU75 project.      |

## Searching for Text

You can search the Data Sheet, HTML or text report output for any text that you choose. These include:

- diff results
- report results
- annotate results
- custom tools that generate text reports

Select **Edit => Find...** or right-click in the region to search and select **Find...** from the context menu. This displays the **Find** dialog box:



**Find text** Enter the text to find. You can also click on the pull-down menu to choose from previously searched text.

**Match case** Select if you want to find text only with the exact case specified. If not selected, the search is case insensitive by default.

**Whole word match only** Select if you want to only find text that matches an entire word. If not selected, which is the default, matching text that is embedded within a word is returned.

**Direction** Select whether to search forward or backward from the current cursor position.

Click **OK** to initiate the search.

To search for the next occurrence of the last text searched for, select **Edit => Find Next**, or right-click in the region to search and select **Find Next** from the context menu.

## Reviewing History

The history dialog box allows you to review the history of previously visited locations. To display the history dialog box, select **Go => History**.

The history dialog box displays a table of all locations visited over the last 30 days, and the date and time at which each was last visited. You can sort the table by location or by time by clicking on the table header.

You can select entries in the history table by clicking on them. Use Ctrl and Shift to select multiple entries in the table.

To operate on the locations listed in the history table, right click the mouse button. This displays a context menu with the following entries:

- **Visit** takes you to the selected location. You can also visit a location by double clicking on it.
- **Add Bookmark** creates a bookmark for the selected location.
- **Delete** deletes the selected location from the history list. You can also delete a location by pressing the Delete key.
- **Clear History** removes all entries from the display.
- **History Properties** displays the Customize History dialog box.

To close the history dialog box, click Close.

## Using Data Sheets

A DesignSync data sheet displays information about a selected object such as a file, folder, version, or vault. To display a data sheet, select an object, then do one of the following:

- Select **File=>Data sheet**.
- Select **Data Sheet** from the right mouse button's drop-down menu.
- Click the **Data Sheet** toolbar button.
- Press **F4**.

DesignSync displays the information in a new window in the View Pane.

The information that is displayed by a data sheet depends on the object you selected. For example, the data sheet for a file in your working folder contains information such as lock status, modification status, version number, associated tags, attached notes, and the ongoing log (see the description of Revision Log for details on the ongoing log). For more information on using and navigating data sheets, see *ENOVIA Synchronicity ProjectSync User's Guide: Displaying Data Sheets*.

### Related Topics

*ENOVIA Synchronicity Command Reference: datasheet Command*

*ENOVIA Synchronicity Command Reference: vhistory command*

*ENOVIA Synchronicity ProjectSync User's Guide: Displaying Data Sheets*

Revision Control Properties

## Setting the Verbosity of the Output Window

Many revision control operations, particularly those that can operate recursively over an entire directory hierarchy, can generate a large amount of output in the output window. Changing the verbosity setting can make it easier to locate the important messages in the output.

Each command that allows controlling of the verbosity command has a report option on the dialog box used to run the command.

If logging is turned on, the log file contains all output regardless of the verbosity mode being used.




## Viewing the Results of an Operation

While a revision control operation is performed, the results of the operation are displayed in the main DesignSync window. There are three places in the window where you can find information about the progress of the operation: the Output Window, the Result column of the list view, and the summary bar.

### Result Column

As soon as an operation begins, the result column appears in the list view. As objects are operated on, the results appear in the appropriate rows of the list column.

For files and collections, the result column contains a message containing the result of the operation on that particular object. The column also displays one of the following icons:

|                                                                                     |                                        |
|-------------------------------------------------------------------------------------|----------------------------------------|
|  | The operation succeeded on this object |
|  | The operation failed on this object.   |
|  | This object was skipped                |

For folders, the result column summarizes the results of the objects it contains. For example, if 5 objects within the folder were operated on successfully, the result column contains the message "5 succeeded."

The result column for a folder also contains one or more icons indicating the results of the objects it contains. For example, if all of the objects in a folder succeeded, the success icon is displayed; if some succeeded and others were skipped, it contains both the success and skip icons.

To see exactly what happened to each object in a folder, double-click on the folder in the list view to expand it, and examine the result column of those objects.

### Summary Bar

The summary bar appears in between the Output Window and the Command Bar. It provides a summary of the operation as it progresses.

The summary bar provides a total of all objects that have succeeded, failed, and been skipped.

For some operations, it is possible to determine how many objects will be operated on before the operation takes place. In this case, the summary bar provides a percent complete indication. If a total cannot be determined in advance, the summary bar provides a total count of objects operated on.

It is not possible to determine a total in advance for the following operations:

- The **populate** command
- The **tag** command
- Any operations entered into the command bar that use the **-recursive** flag.

### Clearing the Results

When you are done examining the results of an operation, you may clear the results by pressing the **Clear Results** button on the summary bar, or by selecting the **Revision Control => Clear Results** menu choice.

When the results are cleared, the Result Column and the Summary Bar disappear.

To clear the operation's output, right-click in the Output Window to display its context menu. Select **Clear Output** to clear the Output Window.

## Common Interface Topics

### Comment Field

Enter notes for this operation here.

- Check-out comments are added to the Revision Log, which is used as part of the future check-in comments.

- Check-in comments are appended to any comments you have in your Revision Log (**File=>Properties**). Check-in comments become part of the version history.

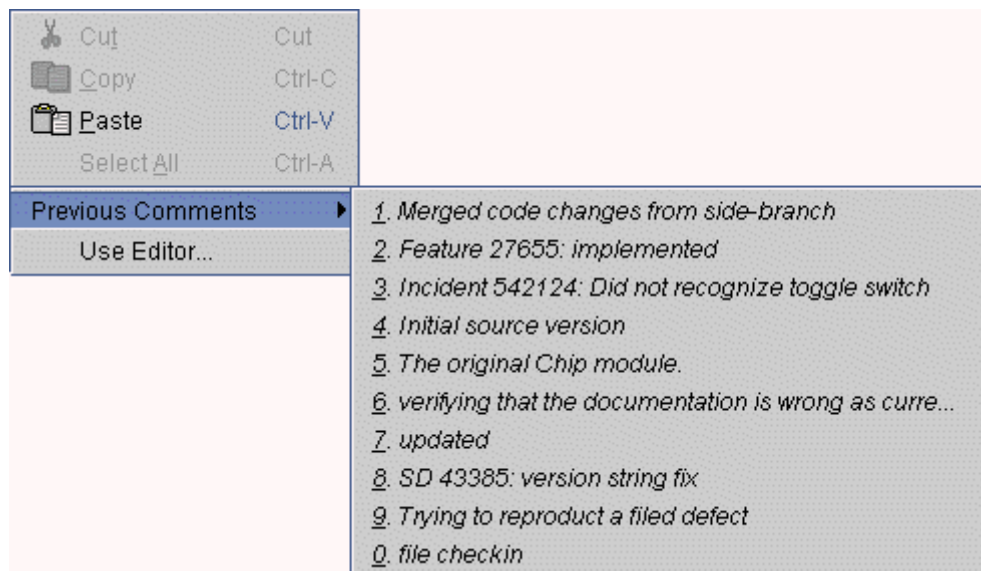
Depending on the DesignSync methodology your team adopts, your project leader may require that every check-in have a comment of a given length. If there is a minimum comment length defined with SyncAdmin, a tool-tip window will display in the lower-right portion of the Comment field. The tool tip will show you how many characters you have entered and the minimum character length of the comment. When you reach the minimum length, the tool tip will close.

**Note:** This does not display a tip for minimum comments length defined with Access Controls.

- Tag and Branch comments, which are only available for module data, become part of the version history.
- Comments specified when creating a new module are reported by the `showmods` command as explained in the `showmods` command description: Understanding the Output, and displayed in the List View when viewing modules on a server.
- Comments support multibyte characters (UTF-8 compliant) both within the DesignSync GUI interface and submitted within comment files.

Right click in the Comments field to display the context menu.

Click on the fields in the following illustration for information.



From here, you can choose to cut, copy, paste, or select all of your text.

### Previous Comments

Choose from comments you entered for previous check-in and check-out operations. If there are no previous comments, then the Previous Comments choice is present but unavailable. A maximum of 10 previous comments are stored, after which older comments drop off the list. Only comments consisting of fewer than 2000 characters are stored. The first 50 characters of a comment are shown in the **Previous Comments** list for selection.

Upon selecting a previous comment, the text from the selected previous comment is entered into the Comment box.

### Use Editor

Open the default editor to prepare your comments. When you save and exit from your editor, your comments will be added to the Comment field. Quitting from your editor without saving leaves the Comment field empty.

### Related Topics

Displaying Version History

ENOVIA Synchronicity Command Reference Help: showmods

### Exclude Field

Enter here the kinds of files or directories you want to exclude from the operation. Wildcards are allowed. Separate items with commas. For example, to exclude all log files from the operation, you would specify **\*.log** .

Click on the button to the right of the **Exclude** text field to bring up a list of common exclude patterns (`tmp,* .o,* .obj,* .bak,* ~,* .log,* .db`). Selecting a pattern will append the pattern to the current contents of the **Exclude** text field.

Do not specify paths in your arguments to **Exclude** . Before operating on each object (such as during a recursive operation), DesignSync compares the object's leaf name (path stripped off) to the items in the **Exclude** field to see if there is a match. Because the object's path is not considered, it will not match any item in the exclude list specified with a path. For example, if you specify **bin/\*.exe** in the **Exclude** field, you will not successfully exclude `bin/test.exe` or any other `*.exe` file. You need to instead specify **\*.exe** (or **test.exe** if you want to exclude only `test.exe`). This means, however, that you cannot exclude a specific instance of a file or folder -- you exclude all matching files and folders.



For details of how the **Exclude** value is used by a particular command, see the `exclude` option description for the command invoked by the graphical interface.

See the SyncAdmin Help: Exclude Lists topic for information about global exclude lists.

## Filter Field

Apply the specified expression, to identify the exact subset of objects on which the command will operate. Specify comma-separated object expressions, where an object expression takes the form:

```
[+|-]<path_expression>
```

where:

- `+/-` indicates whether a particular object expression indicates items to be included or excluded. The default is `-`, excluding items based on the `<path_expression>`
- `<path_expression>` is an extended glob expression that specifies object paths. An extended glob expression is a standard glob-style expression, but extended to allow the use of the `"..."` syntax to mean "match any number of directory levels".

For example, the expression `top/.../lib/*.v` matches any `*.v` file in a directory path that starts with `top`, then has any number of levels (zero or more), ending in a `lib` sub-directory.

**Note:** A glob expression of `*` for a `<path_expression>` is not always the same as `.../*`. For an exclude (`-*` or `*`), the `*` would match the folders in the top level directory, and therefore everything down the hierarchy. But for an include (`+*`), the `*` would only match the top level directories, and not match the lower level items, causing the lower level items to remain excluded.

Path expression matches are performed against the relative path of the objects from the starting point of the command.

If a match is performed against a full `sync: URL`, then a `"..."` value will match the `sync://<host>:<port>` at the start of the URL. For example, `.../Chip` matches `sync://<host>:<port>/Modules/Chip`.

Ordinarily, a command starts with everything included. But if the first entry in the **Filter field** is an include (`+<path_expression>`), then it is assumed that the intention is to start with everything excluded, as if the expression had started with `-.../*`. This makes it simple to write expressions that only include some items.

**Notes:**

- The Exclude field is applied in addition to the **Filter field**.
- When Populating Your Work Area, see Setting Persistent Populate Filters for how Populate uses the **Filter field**.
- The filter command is available for all modules-based operations, but may not apply to all non-module based operations.

## Force Overwrite of Local Modifications Option

By default, DesignSync does not overwrite locally modified files. Use this option to force DesignSync to overwrite locally modified files. Your locally modified copies of files will be replaced according to the fetch mode selected. For example, unlocked copies, or links to cached files.

## Href Filter Field

Apply the specified expression to filter out the hierarchical references followed, when operating on a module recursively. Specify comma-separated object expressions, where an object expression can take one of following forms:

- Simple href filter: `<href_expression>`
- Hierarchical href filter:  
`[/]<Module_expression>[/<Module_expression>...]<href_expression>`

where the `<module_expression>` is a module or href name.

where the `<href_expression>` is a simple glob expression.

A simple href filter is a simple leaf name or the name of the href (specified when the hierarchical reference was added); you cannot specify a path. DesignSync matches the specified href filter against hrefs anywhere in the hierarchy. Thus, DesignSync excludes all hrefs of this leaf name; you cannot exclude a unique instance of the href.

A hierarchical href filter specifies a path and a leaf submodule or hrefname, for example JRE/BIN excludes the BIN submodule only if it is in beneath JRE in the hierarchy.

**Note:** You can use wildcards with both types of hreffilter, however, if a wildcard is used as the lone character in hierarchial href, it only matches a single level, for example: "JRE/\*/BIN" would match a hierarchy like "JRE/SUB/BIN" but would not match "JRE/BIN" or "JRE/SUB/SUB2/BIN".

You can specify both forms of hreffilter within the same operation, however when doing a populate, you can only specify a hierarchical hreffilter for an initial populate. When applying a hierarchical hreffilter, you must specify the Recursive option.

**Tip:** If you need to add or change hierarchical hreffilters for an existing workspace, use either the Setting Persistent Populate Views and Filters or the setfilter command.

For more information on understanding hreffiltering and hierarchical href filtering, see Href and Hierarchical Href Filtering.

**Notes:**

- This field is only available when operating on module data.
- When Populating Your Work Area, see Setting Persistent Populate Filters for how Populate uses the **Href filter field**.

## Keys Field

You can control keyword substitution by selecting one of the following options from the **Keyword Substitution** drop-down menu:

- **Update values and keep keys** expands keyword values and retains the keywords in the file (default option). For example: \$Revision 1.4 \$
- **Update values and remove keys** expands keyword values but removes keys from the file. This option is not recommended when you check out files for editing. If you edit and then check in the files, future keyword updates are impossible, because the value without the keyword is interpreted as regular text. For example, 1.4.
- **Remove values and keep keys** keeps the keywords but removes keyword values. This option is useful if you want to ignore differences in keyword expansion, such as when you are comparing two different versions of a file. For example, \$Revision: 1.9 \$
- **Do not update** keeps exactly the same keywords and values as were present at checkin.

By default, revision control keywords in your ASCII (but not binary) files are expanded during a checkout or populate operation. You can also expand keywords in local copies of files that you leave in your working directory during a checkin.

If you perform this operation using a mirror or cache directory, keywords are automatically expanded in the file that remains in the cache or mirror directory and the keyword itself remains in the file -- as if the **Update values and keep keys** option were used.

**Related Topics**

Revision Control Keywords Overview

## Local Versions Field

**Note:** This option only affects objects of a collection defined by the Custom Type Package (CTP). This option does not affect objects that are not part of a collection or collections that do not have local versions.

When it fetches an object, the populate operation first removes from your workspace any local version that is unmodified. (To remove a local version containing modified data, specify **Force overwrite of local modifications**.) Then the populate operation fetches the object from the vault (with the local version number it had at the time of checkin).

The **Local Versions** option specifies the action that the populate operation takes with modified local versions in your workspace (other than the current, or highest numbered, local version). (DesignSync considers a local version to be modified if it contains modified members or if it is not the local version originally fetched from the vault when the collection object was checked out or populated to your workspace.)

Specify the **Local Versions** option with one of the following values:

- **Save local versions.** If your workspace contains a local version other than the local version being fetched, the populate operation saves the local version for later retrieval. For more information on retrieving local versions that were saved, see the ENOVIA Synchronicity Command Reference: `localversion restore` command.
- **Delete local versions.** If your workspace contains a local version other than the local version being fetched, the populate operation deletes the local version from your workspace.
- **Fail if local versions exist.** If your workspace contains an object with a local version number equal to or higher than the local version being fetched, the populate operation fails. This is the default action. A DesignSync administrator can change this default setting. For more information, see SyncAdmin Help: Command Defaults.

**Note:** If your workspace contains an object with local version numbers lower than the local version being fetched and if these local versions are not in the DesignSync vault, the populate operation saves them. This behavior occurs even when you specify **Fail if local versions exist**.

## Module Context Field

This field is only available when a module folder is being operated on. Specifying a module context enables the operation to be run on a workspace folder that is below multiple modules, or on a sub-folder of a module on a server.

When viewing the module folder in the Folder Explorer, the default module context value is shown as an empty string. You can type the server URL of a module, or select an existing client module to restrict the scope of the operation. You can also select from

among the module instances for the folder being operated on, which are listed alphabetically in the pull-down. If more than one folder is being operated on, you can only select from the available module instances (you cannot type in a value).

For dialog boxes in which a `Browse...` button appears next the **Module context** field, you can use the `Browse...` button to navigate to and select a module on a server.

When viewing the module folder in the Module Explorer, the module context value is the selected folder's parent module. If the selected object is the base folder itself, then that is the module context value.

When performing a checkin operation with the **Allow Checkin of New items** option selected, the module context default value is `<Auto-Detect>` which indicates that DesignSync uses smart module detection to determine the target module for the new module member. For more information, see [Understanding Smart Module Detection](#).

## Module Views Field

This field is only active when a module is being operated on. You can type the name of the module view or query the server for a list of defined views.

The Module Views field is active:

- during the initial populate of a module in the workspace.
- when you are changing the persistent view or filter for a module in a workspace.

To query the server for a list of module views, click the down arrow to bring up the **Get views...** option. You can then select a view from the list.

To clear view set on a workspace, you can type `None` or select it from the list.

You can select multiple views by separating the view names with a comma. (For example, `DOC, RTL`)

### Notes:

- When you query the server for module view information, the system displays a "Getting module views" screen. You can stop the process by clicking the **Stop** button during the query.
- If there are multiple module views with the same name in a module hierarchy, the list will only display a single unique view name; the first view definition reached as the query traverses the hierarchy from the selected object (child) up the hierarchical tree.
- When doing a Compare operation (**Reports | Compare**), the Module Views field is only active when a selector is provided.

## Populate Log

Because populate operations can be long and complex, you may want to specify a log file to contain only the output of the populate command to store for later reference. This is particularly useful in cases where you perform a complex populate operation, such as merging a set of module changes from a different branch into workspace.

The log file name can be specified as one of the populate options or saved as a default value in the DesignSync GUI.

If the specified log file exists, the populate output automatically appends to the end of it, preserving earlier populate information.

### Example of populate log

```
Logging populate command output to: /home/rsmith/popmerge.log
```

```
Beginning populate operation at Thu Jul 19 03:05:46 PM EDT
2007...
```

```
Populating objects in Module          ROM%1
Base Directory /home/rsmith/MyModules/rom
Without href recursion
```

```
Fetching contents from selector 'Silver:', module version
'1.6.1.2'
```

```
Merging with Version: 1.6.1.2
Common Ancestor is Version: 1.6
```

```
=====
=====
Step 1: Identifying items to be merged and conflict situations
=====
=====
```

```
/romMain.c : No merge required.
/rom.v : No merge required.
/rom.c : No merge required.
/romSub.c : No merge required.
/doc/rom.doc : No merge required.
```

```
=====
=====
Step 2: Transferring data for any items to be fetched into the
workspace
=====
=====
```

No files to fetch.

```
=====
=====
Step 3: Merging file contents as required into the workspace
=====
=====
```

Beginning Check out operation...

Checking out: rom/rom.doc : Success - Version  
1.1.1.1 has replaced version 1.1.

Checkout operation finished.

```
=====
=====
Step 4: Updating files fetched into the workspace
=====
=====
```

ROM%1 : Version of module in workspace not updated (Due to  
overlay operation).

```
=====
=====
Step 5: Comparing hrefs for the workspace version and merge
version:
=====
=====
```

No hrefs present in workspace version  
No hrefs present in merge version

Finished populate of Module ROM%1 with base directory  
/home/rsmith/MyModules/rom

Finished populate operation.

### Related Topics

Populating Your Work Area

## Recursion Option

For a DesignSync folder, recursively operate on its contents. The set of objects operated on may be reduced by use of the Exclude field. By default, only the contents of the selected folder are operated on.

For a module, follow any hierarchical references and recursively operate on their contents. The set of objects operated on may be reduced by use of the Exclude field, the Filter field and/or the Href filter field. By default, only the contents of the selected module are operated on.

For a module folder, recursively operate on its contents. The Module context field determines the set of objects operated on. The set of objects operated on may be further reduced by use of the Exclude field and the Filter field. By default, only the contents of the selected folder are operated on.

## Retain Timestamp Field

Use this option if you want to retain the "last modified" time of when the file version was checked into the vault.

This option is meaningful only when working with physical copies, as is the case when you specify the **Unlocked copies** or **Locked copies** options. DesignSync automatically uses this "keep last modified time" behavior when linking to files in a mirror or cache directory; files in the mirror/cache retain their original timestamps. However, links in your work area to the cache/mirror have timestamps of when the links were created. If you specify the **References to versions** or the **Locked references** options, no object is created in your work area, so there is no timestamp information at all.

**Note:** If an object is checked into to the vault and the setting of the **Retain Timestamp** option is the only difference between the version in the vault and your local copy, DesignSync does not include the object in checkout operations.

DesignSync operations follow the SyncAdmin registry setting for Retain last-modification timestamps. By default, this setting is not enabled; therefore, the timestamp of the local object is the time of the check-in, check-out, or cancel operation. To change the default setting, your Synchronicity administrator can use the SyncAdmin tool. For information, see SyncAdmin Help: Command Defaults.

## Suggested Branches, Versions, and Tags



Dialog boxes that ask you to specify a version, branch, or tag provide a pull-down menu of suggested entries. These entries include:

- Any tags specified by the Project Leader through the Sync Administration application. See SyncAdmin Help: Tag Options.
- Standard selectors, such as **Trunk** and **Latest**, and special specifiers, such as **Date()**. These only appear when appropriate.
- Up to three options that query the server and add the results to the selection list:
  - **Get branches** queries the server for a list of all branches of the selected object.
  - **Get versions/tags** queries the server for a list of all versions or tags of the selected object. This option only appears when appropriate and when a single object is selected.
  - **Get selectors** gets a list of all selectors appropriate for the current directory. This only works correctly if the current directory, or its associated vault, has the form **sync://host:port/Projects/projectname/....** And if *projectname* is a ProjectSync project that has configurations defined.

**Note:** When you query the server for branches, versions, tags, or selector information, the system displays a "Getting vault information" screen. You can stop the process by clicking the **Stop** button during the query.

## Trigger arguments

Trigger arguments are passed to any triggers that have been set up for the operation. Consult your project leader for information about any triggers that are in use and how they use arguments.

## Command Invocation

The DesignSync command that will be invoked by the graphical interface is shown at the bottom of the dialog box. As fields are selected in the dialog box, the command options corresponding to each field in the dialog box are added or removed from the command invocation.

See the ENOVIA Synchronicity Command Reference for definitions of command options. The description of a command notes if the command is subject to access control.

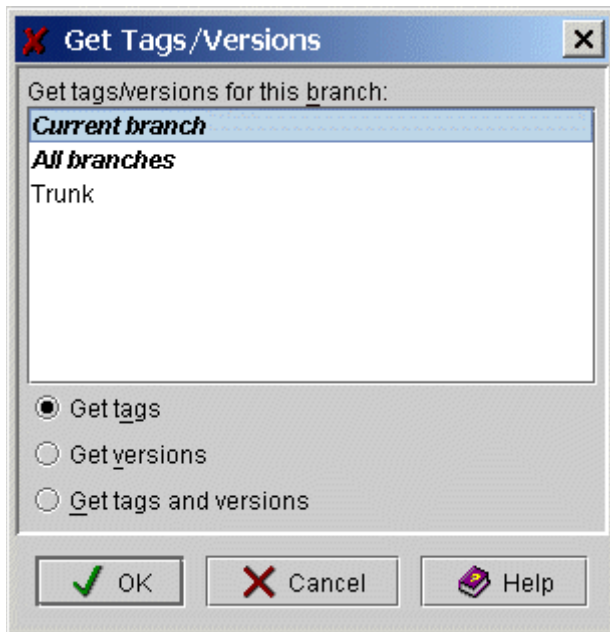
**Note:** The command line defaults system only pertains to the command line interface. Underlying commands that are automatically invoked by the DesignSync graphical interface do not use the command line defaults system.

## Command Buttons

| Button        | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|---------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Save Settings | <p>Click this button to save the option settings that you have selected. The saved settings are displayed the next time you bring up the dialog box, and the settings persist from one DesignSync invocation to the next.</p> <p>Most saved settings apply only to future invocations of the same operation. However, some saved settings, such as for <b>Exclude Filter</b> and <b>Key Substitution</b>, apply to all operations that support the options. You typically want to apply some options consistently across all commands. For example, if you exclude *.log files during checkins, you likely want to exclude *.log files for all operations.</p> |
| OK            | When you click on the <b>OK</b> button, your settings are executed and the dialog closes.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| Cancel        | When you click on the <b>Cancel</b> button, the dialog closes without executing any of the settings in the dialog.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| Help          | This button invokes help information for the dialog. You can also invoke help by pressing <b>F1</b> at any time.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |

## Get Tags/Versions

The Get Tags/Versions dialog allows you retrieve from the vault the available tags and versions applied to the object.



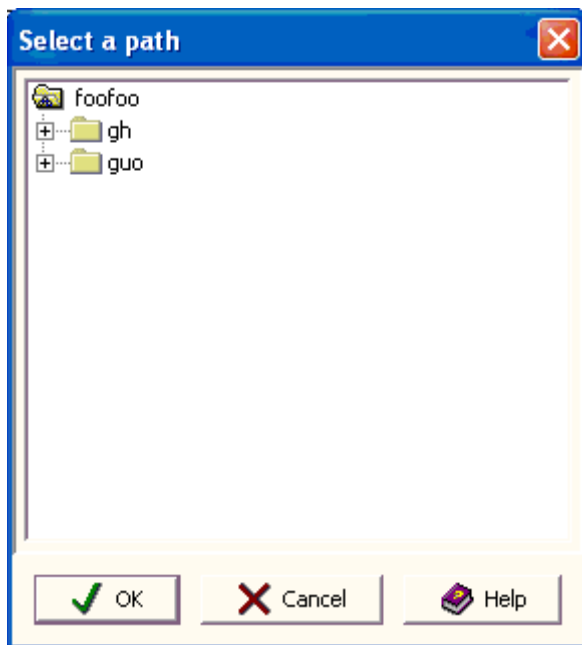
### To display available tags and versions:

1. From the **Get tags/versions for this branch** field, select the branch for which you want to display tags or versions. You can choose the current branch, all branches, or branches that are defined in the vault (for example, Trunk).
2. Select an option to specify whether you want display tags, versions, or both tags and versions.
3. Click **OK**. DesignSync displays a list of available tags and versions in the pulldown menu of the Version field.

**Note:** The tags displayed by the Get Tags/Versions dialog are "tags of interest" for the selected object(s). Tags of interest include any module snapshot tags and any tags set to display by the `AlwaysShowTags` registry key. For more information on defining tags to display in the registry, see *ENOVIA Synchronicity DesignSync Data Manager Administrator's Guide: Modules Registry Settings*.

## Select a path

If you are moving a module member and use the Browse button to select the new path name, the **Select a Path** dialog box appears.



### To select a path with the Select a path dialog box:

1. Navigate through the folder until you get to the path you want.
2. Click **OK**. The new path for the module member appears in the **New path** field of the **Move module members** dialog box.

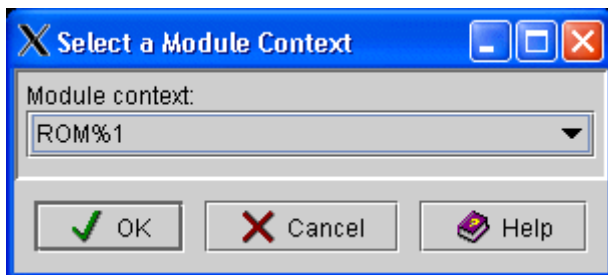
## Select Module Context

The Select Module Context dialog displays when an object needs a module context assigned to it. This usually happens when the objects can be associated with more than one module, for example, when adding an object to a module in a workspace that contains more than one module.

You may see this menu when performing any of the following tasks:

- Adding objects to a module
- Viewing a module's data hierarchy
- Navigating to the module vault on the server

Click on the fields in the following illustration for information.



### Module Context

Select from the available module instances. The choices are listed in alphabetical order. Only the listed module instances can be applied to the selected objects.

When you are performing an operation that creates a new module object, such as **Checkin** with the **Allow check in of new items** option or **Add**, you have an additional default option of <Auto-detect> which uses the DesignSync smart module detection to determine the target module for the objects. For more information on using smart module detection, see Understanding Smart Module Detection.

### Related Topics

[Adding a Member to a Module](#)

[Checking in Design Data](#)

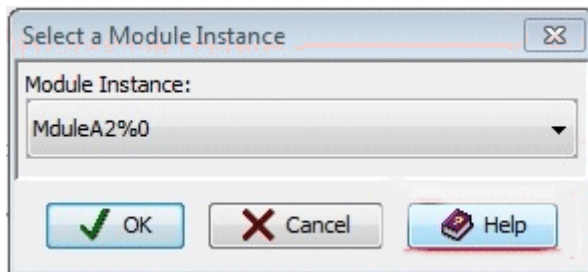
[Displaying Module Hierarchy](#)

[Go Menu](#)

## Select Module Instance

The Select Module Instance dialog displays when you have more than one possible instance of a module you want to see. This usually happens when there is more than one module instance version in the module base directory.

**Click on the fields in the following illustration for information.**



### Module Instance

Select the desired 'module instance from the drop-down list.

### Related Topics

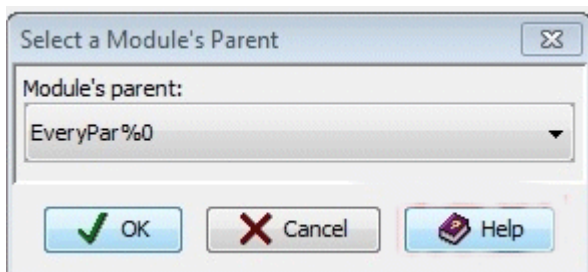
Module Explorer

## Select Parent Module

The Select Parent Module dialog displays when navigating within the workspace to a parent module results in more than one parent module candidate.

You may see this menu when working in the workspace with hierarchical module structures featuring multiple modules populated in the same module root.

**Click on the fields in the following illustration for information.**



### Module's Parent

Select the desired parent module from the drop-down list.

## Related Topics

Module Explorer

Go Menu

## Select Vault URL Browser

Facilitates entering a vault URL by using the vault associated with a public project (as defined by your project leader) or an entry in your site or local **SyncServer lists**. You can expand the listed project or server folders to select subfolders if desired.

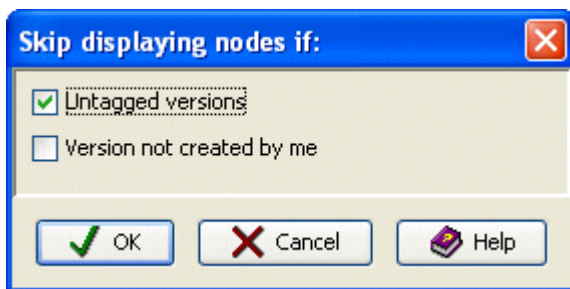
## Filter Interesting Dialog

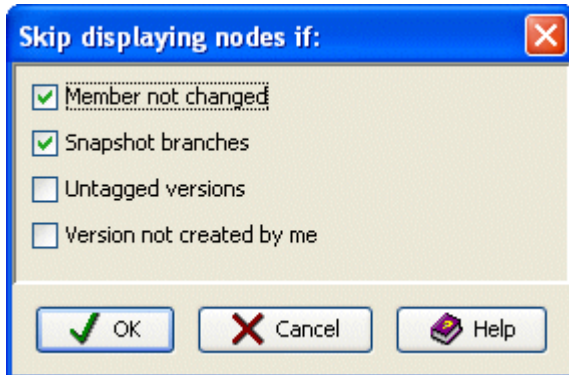
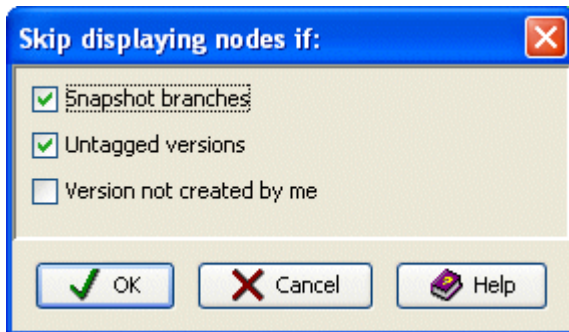
Using the Filter interesting objects option allows you to control which objects are displayed when using the Vault Browser with the Show Interesting mode selected. Checking any option removes a version from the Vault Browser display if they do not meet any other inclusion criteria.

**Note:** Versions or branches excluded by selecting filter conditions are shown if there is a visible sub-graph rooted at that version or branch.

The filter interesting dialog changes depending on what type of object you have selected in the vault browser. The following illustrations shows file-based vaults, modules vaults, and module member vaults.

Click on the fields in the following illustrations for information.





### Untagged Versions

Check to remove any versions that are not explicitly tagged from the vault browser display.

### Version not created by me

Check to remove any versions that were not created by the user who is running the vault browser.

### Snapshot branches

Check to remove any versions that are on a snapshot branch. This is applicable only to module and module member versions.

### Member not changed

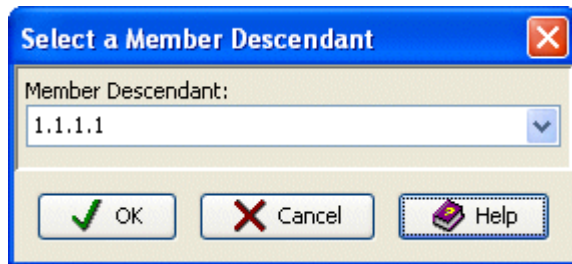
Check to remove any versions that are unmodified. This is only applicable to module member versions.

## Related Topics

Vault Browser Tools

## Select a Member Descendant

Within the Vault Browser, you can navigate to one of module versions containing descendants of the selected member version. If there is more than one descendant for a member, DesignSync displays a dialog box for choosing a descendant being displayed:



### Member Descendant

Select the desired member descendant from the drop-down list.

## Related Topics

Vault Browser Actions





# Index

## A

### Annotate 333

- actions 335

- highlighting results 337

### Auto-Merging 189

- locally added files 429

- locally modified files 430

  - non-latest 432

  - removed from module 431

  - renamed in module 433

  - with other renamed files in module 434

## B

### Bookmarks

- adding 821

- adding a vault 51

- deleting 821

- editing 821, 823

- organizing 821

- properties 823

### Branch 651

- autobranching 766

- container 364, 374

- creating 124, 766

- feature or subproject 769

- modules 216

- operations 767

- parallel development 764

- policy 775

- sub-module 364, 374

- tagging 127

- unlocking 136

## C

### Cache

- file cache 733, 735

- module cache 735

- versus mirrors 733

### Cadence Objects 810

- enabling recognition 812

- managing 814

- recognizing 812

### Check In 108

- comments 36, 827

- files 108

- object state 36
- Checkout 100
  - canceling 105
  - files 100
  - potential 293
- Collection 808
  - custom type package (CTP) 816
  - displaying 809
  - managing non-collection objects 815
  - member properties 644
  - properties 643
- Command
  - aliases 719
  - buttons 838
  - invocation 838
  - line defaults system 728
  - line editing 714
- Comparing Objects 314, 319
- Configurations 796
  - creating 796
  - mapping 797
  - selecting 29
- Custom Type Package (CTP) 816
  - DesignSync recognition 817
- D
  - Data Sheets 825
  - Date Format 760
  - Deleting 252
  - DesignSync 1
    - architecture 633
    - command shell 391
    - command-line shells 710
    - configuring 720
    - GUI 379, 662
    - introduction 2
    - list view pane 695
    - output window 699
    - special keystroke operations 691
    - symbols and icons 664
    - tree view pane 693
    - URLs 648
- Diff
  - advanced 274
  - graphical diff utility 288
  - output 280
  - revised diff format 285

## DesignSync Data Manager User's Guide

- simple operations 273

dss 700, 710

- compared to other shells 711

- invoking 712

- running scripts 731

dssc 700, 710

- compared to other shells 711

- invoking 712

- running scripts 731

E

Edit

- command line 714

- menu 673

- sub-module 356

Edit-in-place

- overview 233

Enterprise Development

- displaying 313

External Modules 228

- URL syntax 645

F

Field

- excluding 829

- filter 830

- href filter 831

- keys 832

- local versions 832

- module context 833

- module views 834

- retain timestamp 837

File

- checking in 108

- checking out 100

- creating 241

- deleting 253

- excluding 35

- moving and renaming 242

- populating 53, 79

- retiring 267

- selecting 819

Filter

- display filters 331

- module data 196

- persistent populate filters 74

Folders

- creating 250

- deleting 255
- moving and renaming 252
- versioning 215

G

- General Properties 638
- Go Menu 677
- GUI 662

H

- Help
  - contacting ENOVIA 6
  - menu 691
- Hierarchical References
  - adding to module 409, 411
  - creating 158
  - deleting 165
  - filter field 831
- Hierarchy
  - specifying location 45
- History 824

L

- Legacy Modules
  - handling 784
  - upgrading 785

Library

- cadence 810

Location

- bar 685, 819

Locking

- module 204
- module data 166
- work style 739, 741

Login 19

M

- Merge Edge 11
- Merging 6, 738
  - conflict editor 13
  - conflicts 7, 742
  - module data 219
  - modules 224
  - three-way merge 9
  - two-way merge 9
  - work style 739, 763

Metadata 653

Mirrors 655, 736

- administering 661
- architecture of the system 659

permissions 72

specifying 31

using 69, 657

versus caches 733

### Module 181

adding content to 121, 155, 242

branching 216

creating 152

deleting 170

displaying status 306

external modules

adding an href to 228

instance names in the workspace 73,  
97, 149, 167

locking 204

members

moving 245, 840

removing 163, 270

tagging 230

merging 219, 224

recursion 200

resolving structure conflicts 176

rolling back 168

where used 309

Module Cache 735

deleting module cache link 174

displaying 301

using 238

### Module Context

defining and enabling 370

field 833

locating 363

selecting 841

### Module Hierarchy 206

creating 409, 411

populating 40

### Module Views 192

## O

### Objects

comparing 273, 274

modified 293

retiring 267

selecting 819

states 30, 95, 634

types 636

unmanaged 293

## P

### Parallel Development

- overview 764

### Persistent Selectors 749

### Populate 53, 79

- results 826

- snapshots 230

### Projects 21

- properties 23

- public 23

- setting up a work area 21

### Properties

- collection member 644

- collections 643

- displaying project 23

- general 638

- module objects 640

- revision control 639

- tag 642

- version 642

- viewing and setting 637

## R

### REFERENCE

- chaining 808

- REFERENCEs and revision control commands 804

### Registry Files 720

### Releases

- creating 798

### Revision Control 4

- keywords 143

- using 144

- menu 680

- properties 639

## S

### Scripts

- creating 730

- DesignSync 728

- OS shell scripts 728

- running 731, 732

### Selectors 746

- dynamic 746, 751

- formats 751

- selector lists 748

- specifying 29

- static 746, 751

### SITaR Designer 355

## DesignSync Data Manager User's Guide

- branching a sub-module 358
- creating a SITaR sub-module 371
- creating a workspace 355
- editing a sub-module 356
- submitting a sub-module for integration 357
- synchronizing with the baseline 356, 357
- SITaR Integrator 359
  - branching 364, 374
  - creating a SITaR container module 369
  - creating a SITaR sub-module 371
  - creating an initial baseline release 372
  - creating workspace 360
  - integrating 362
  - integration workspace 359
  - locating submissions 361
  - releasing a baseline 364
  - selecting sub-modules 361
  - testing release candidates 362
  - workflow overview 360
- SITaR Overview
  - branching 377
- designer role 355
- integrator role 359
- module structure 375
- SITaR environment variables 365
- workflow 374
- stcl 700
  - compared to other shells 711
  - invoking 712
  - running scripts 728, 731
- stclc 700
  - compared to other shells 711
  - invoking 712
  - running scripts 731
- Swap 233
- SyncAdmin 720
- syncd 711
- SyncServer 634
  - authentication 19
- T
- Tag
  - branches 127
  - fixed and movable 798
  - properties 642



- retrieving from the vault 839
- snapshots 230
- using 798
- versions 127

Triggers

- arguments 838

U

URL 648

- syntax 645

User Authentication 19

V

Vault Browser 340

- actions 344
- finding objects 350
- selecting URL 843
- tools 346

Vaults 651

- adding to bookmarks 51
- browsing for a file or project 51
- changing for a hierarchy of files 73
- deleting 261
- location 45
- permissions 726
- upgrading 791
- vault data 295, 298
- viewing contents 51

Versions 651

- deleting 264
- history 325
- properties 642
- selecting 744
- suggested 837
- tagging 127
- viewing in the vault 51

View

- field 834
- persistent view 39, 74
- view panel 309

W

Where Used 309

Work Area

- controlling access 721
- moving 723
- permissions 726
- populating 53, 79

Workspace

## DesignSync Data Manager User's Guide

recreating developer's workspace 363

selecting a workspace 28, 34, 40

selecting workspace files for new  
modules 43

setting a root 73, 167