SPARCworks/TeamWare Solutions Guide

SunPro
A Sun Microsystems, Inc. Business
2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

Part No.: TWSG-0194/1K
Revision: January 1994

Contents

Preface

This initial release of the *SPARCworks/TeamWare Solutions Guide* is just a beginning — it is anticipated that this guide will be updated periodically to include new topics.

This initial release is comprised of three scenario based topics designed to help users take full advantage of SPARCworks/TeamWare™ features. The topics are:

- *Workspace Hierarchy Strategies for Software Development and Release*
- *Deleting Files From CodeManager Workspaces*
- *Remote Software Development Using CodeManager*

It is assumed that readers of this guide have some software development experience and understand basic SPARCworks/TeamWare concepts.

For further information about the SPARCworks/TeamWare tools please refer to these manuals on-line through AnswerBook on the SunPro product CD:

- *CodeManager User's Guide*
- *VersionTool and CheckPoint User's Guide*
- *Merging Source Files* (FileMerge)
- *ParallelMake User's Guide*

Notational Conventions

The following table describes the notational conventions and symbols used in this guide.

Table P-1 Notational Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	Names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `system% You have mail.`
`AaBbCc123`	User input, contrasted with on-screen computer output	`system%` **`su`** `password:`
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type **`rm`** *filename*.
AaBbCc123	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in *User's Guide*. These are called *class* options. You *must* be root to do this.
♦	A single-step procedure	♦ **Click on the Apply button.**
Code samples are included in boxes and may display the following:		
`%`	C shell prompt	`example%`
`$`	Bourne shell prompt	`example$`
`#`	Superuser prompt, either shell	`system#`

TW Topic:

Workspace Hierarchy Strategies for Software Development and Release

The way you organize CodeManager workspaces into development hierarchies affects how software changes move through your development groups and how you manage software releases.

This topic discusses workspace hierarchy strategies that have been developed at a large software development site that uses CodeManager.

This discussion is intended to present you with ideas and concepts that you can adapt to your own unique development environment. Perhaps the most significant underlying concept presented in this discussion is that CodeManager workspace hierarchies are *flexible*. CodeManager enables you easily reconfigure hierarchies as your project, organization, and experience with CodeManager evolve.

Scenario

A company that produces software to control assembly line processes recently purchased CodeManager to help it manage its large software development process. This company markets two products — one controls large, heavy industrial processes and another controls electronics assembly lines. The industrial product employs approximately 100 developers and the electronics product employs about 75 developers.

After analyzing their development and release processes, company management recognized that their primary goals for using CodeManager were to:

- Increase developer access to software changes. As changes are made to various parts of the product, make them quickly available to developers whose work is affected by those changes.

- Maximize developer access to integration areas by keeping software changes accessible to developers during the integration process. Integration consists of merging the changes made by individual developers and then, later, integrating those changes with changes made by other groups of developers.
- Increase the stability of integration and release areas.
- Facilitate concurrent development of multiple software releases.
- Move features between releases.
- Manage unanticipated releases.

The company realized that a key to accomplishing these goals was to effectively organize its workspace hierarchies.

Implementation

The following sections describe the workspace hierarchy strategies that company management developed that enabled them to meet their goals.

Workspace Hierarchy Levels

The company decided that to best organize both development groups (the Industrial Group and the Electronics Group), the development workspace hierarchies must be at least three levels deep. These levels are:

- Development level
- Integration level
- Release level

Note – Your workspace hierarchies may require fewer levels or more levels than this company decided was optimal for it. If your project consists of a very large number of developers, you can add a second sub-integration level in order to reduce the number of developers that put back work to each integration workspace. If, on the other hand your project is not large, you can omit the integration level entirely and have developers put back work directly to the release workspace. The principles described in this section apply in either case.

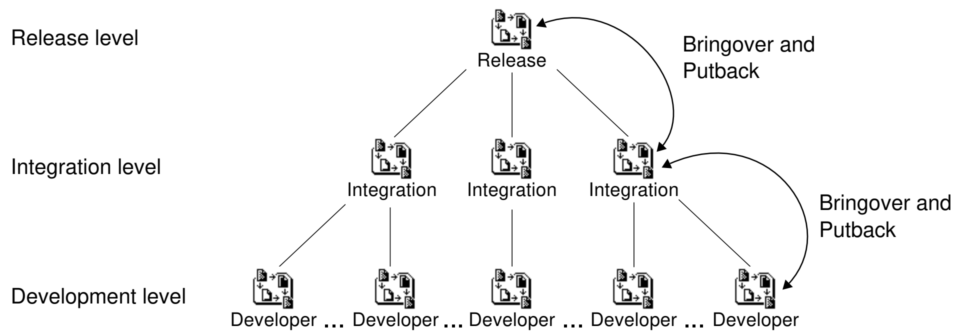


Figure 1 Workspace Hierarchy Levels

Development Level

This is the level in which the code development work is done. Generally, each developer has her own workspace. Note that a developer who works on more than one distinct task might have more than one workspace.

In a typical development cycle, the developer:

- Brings over changes from the integration level
- Merges the changes into her own work
- Makes additional changes of her own
- Tests the changes
- Puts the merged changes back to the integration workspace

Development workspaces generally contain only the files required for the developer to build and test her work.

Integration Level

Integration workspaces are used to combine the work of a number of developers. Generally, integration workspaces are allocated based on functionality, for example the group responsible for a large library and the group responsible for product's I/O drivers are assigned their own integration workspaces. Developers working in these functional areas share their changes by putting their changes back to their integration workspace and by bringing over changes made by other developers from these workspaces.

Integration workspaces generally contain only the files required to build and test a given portion of the product.

Release Level

The release level is comprised of a single workspace at the top of the hierarchy. After work done by developers in functional groups is integrated in the integration workspaces, it is put back to this workspace for final product release integration, testing and dissemination to the other integration workspaces. Each release is generated from a child workspaces of the release workspace that are created specially for this purpose.

Release workspaces contain all the files that comprise the product.

Optimizing Integration

Company management decided that one of the most crucial steps in the development process is the integration of developers' work in both the integration workspaces and in the release workspace. The strategies discussed in this section maintain the consistency and availability of integration workspaces.

If integrators work directly in integration workspaces, the workspaces must be locked against putback and bringover transactions from developers in order to insure consistency. Locking integration workspaces interrupts the development process by preventing developers from exchanging code.

Likewise, when the work from integration workspaces is integrated into a release workspace, the release workspace also goes through a period of instability. During the time it takes to incorporate changes, work cannot be putback to, or brought down from the release workspace.

The development groups arrived at two strategies to address these issues. The two strategies are very similar, they differ slightly in the way integration is managed. Both strategies mitigate the disruption caused by the integration process.

The first strategy was adopted by the Electronics Group. Their product is highly interdependent — the entire product must be integrated as a whole. The second strategy was adopted by the Industrial Group. Their product is more modular — the different functional areas can be integrated separately.

Strategy 1

Because the Electronics Group's product is highly interdependent and must be integrated as a whole, this strategy utilizes a *main integration* workspace in which the entire electronics product is integrated.

Integration does not actually take place in the individual integration workspaces, instead the code is brought over from the individual integration workspaces to the main integration workspace where the work is integrated, tested, and then put back to the release workspace.

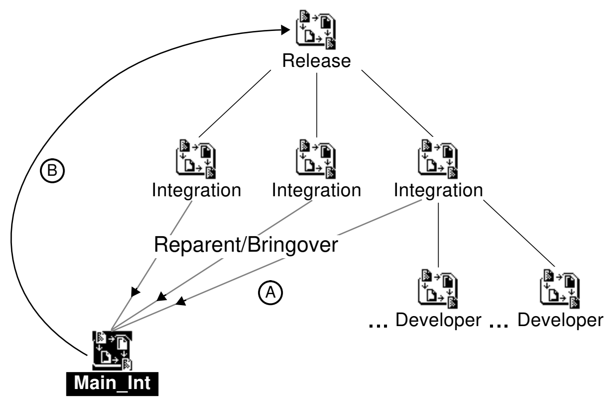


Figure 2 Using a Main Integration Workspace (Strategy 1)

The following steps outline how changes are integrated and moved into the Release workspace.

1. **A Main Integration workspace is created.**
2. **At specified development milestones, the Main Integration workspace is reparented (in turn) to each Integration workspace and all changes are brought over into it (A).**
3. **The changes are integrated, tested and stabilized entirely in the Main Integration workspace.**

4. **After the integration is complete and stable, the contents are put back to the Release workspace where it becomes available to the Integration workspaces, and ultimately to the developers (B).**[1]

This strategy accomplishes the following goals:

- It isolates the disruption caused by the integration process from the main development environment.

  By using the bringover transaction to transfer changes from the integration workspaces into the Main Integration workspace, all conflicts are isolated in the Main Integration workspace. If the putback transaction is used to transfer changes directly from integration workspaces to the Release workspace, conflicts are resolved in directly in the integration workspaces.
- Developers can continue to put back and bring over changes without disrupting the integration process.
- Changes are not introduced to the Release workspace until they have been stabilized in the Main integration workspace.

Strategy 2

The Industrial Group employed a variation of Strategy 1. Their source base is organized so that sub-components of the product (contained in integration workspaces) can be integrated separately or in groups. An advantage of this strategy is that changes made by developers can be disseminated throughout the hierarchy as they become ready, rather than waiting for the entire project to be ready as is the case in Strategy 1. Final product integration is postponed until all changes are present in the release workspace.

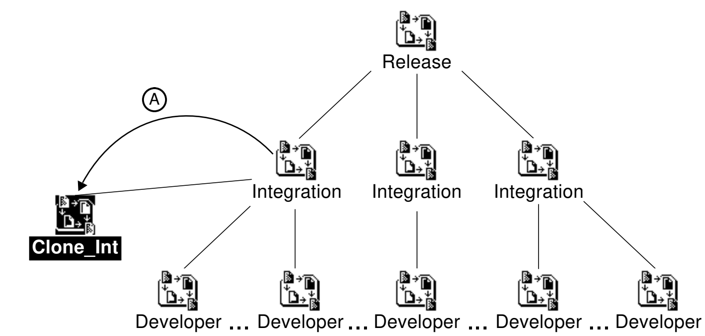
Figure 7 describes Strategy 2.

---

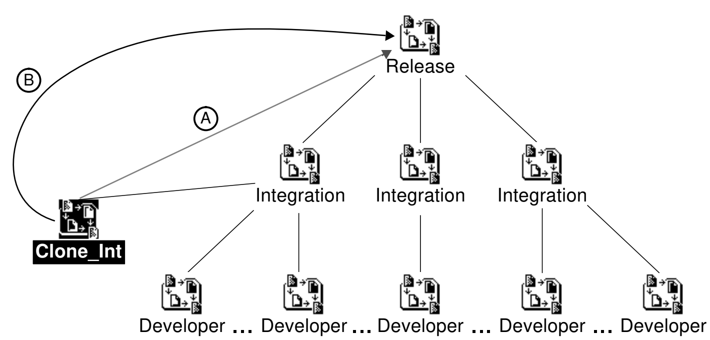
1. Instead of using the putback transaction to move the contents of the Main Integration workspace to the Release workspace, you could simply replace the Release workspace with the Main Integration workspace. An advantage of using the putback transaction is that you insure that nothing has been inadvertently changed in the Release workspace since the last update.

1. At specified development milestones, each Integration workspace (or group of Integration workspaces) uses the bringover create transaction to create a child workspace in which to do integration work (A).

   The child workspace is a *clone* of the Integration workspace.



2. After the integration process is complete, the cloned integration workspace is reparented to the Release workspace (A) and its changes are put back to the Release workspace (B).



3. When all of the Integration workspaces have put their changes back, the Release workspace is cloned using the Bringover transaction (A). Final integration takes place in the cloned workspace.

   By integrating in a clone of the Release workspace, the Release workspace remains open to bringover transactions and the development process is not disrupted during integration.

4. After the changes are integrated and tested, the work is put back from the cloned release workspace to the original Release workspace (B).

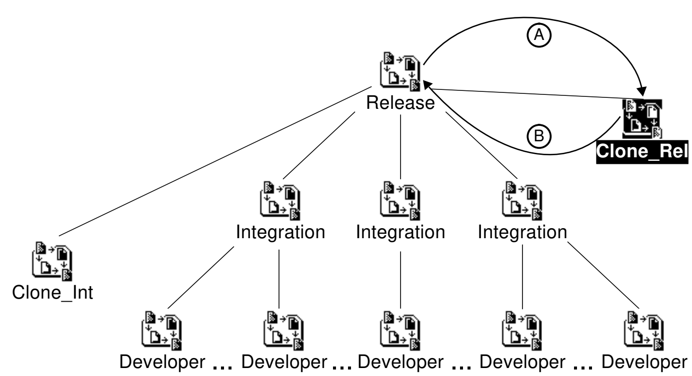


Figure 3 Using Integration Workspace Clones (Strategy 2)

Peer-to-Peer Updates

After the company began using CodeManager, some developers discovered that it was sometimes helpful to reparent and use the bringover transaction to *directly* update each other's workspaces.

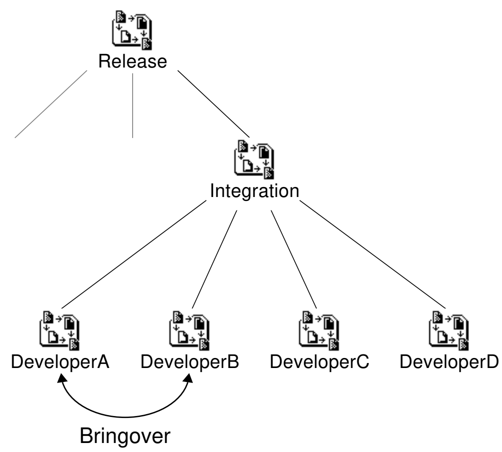


Figure 4 Peer-to-peer Updates Using the Bringover Transaction

This peer-to-peer update technique was found to be useful when two or three developers are working closely on a specific part of a project. They are typically part of an integration group that contains other developers who are working in other areas. It is sometimes efficient and convenient for closely coupled developers to share their work directly with each other and not through the normal workspace hierarchy.

Development groups have found this strategy to have these *advantages*:

- Developers exchange information efficiently and quickly.
- Developers can share unstable, experimental code without going through the common integration workspace where it can inadvertently be brought over by other developers.

They have also found that the strategy can have these *disadvantages*:

- When it is time for the developers to put their code back to the integration workspace, along with their own changes, they put back changes made by the other developer. This requires a developer to merge code that he hasn't written.
- Code that *should* be shared with other developers is sometimes not available to them as quickly as it would be if the updates had been made through the common integration workspace.

These co-developers use the bringover transaction to exchange work directly between their workspaces. This is best accomplished by reparenting "on-the-fly" as part of the bringover transaction (specify an alternate parent in the Transactions Window or use the `-p` option on the command-line).

Note – Generally, the putback transaction is not used in this manner because it is best for a developer not to change the contents of another developer's workspace.

Concurrent Development

The company often finds itself in the position of developing multiple product releases concurrently. For example, development of Release 2 of the industrial product began well before development of Release 1 was complete.

The company employs a couple of strategies to assist it with concurrent development. The major goals of these strategies are:

- To ensure that successive releases are based on the same consistent set of sources.
- To ensure that all features developed for Release 1 are also included in Release 2.
- To facilitate the porting of features to Release 2 that were intended for Release 1 but were not completed in time.

The *cascade model* was found most effective when the source bases for Release 1 and Release 2 are closely related. The *reparenting model* was developed for cases when the Release 1 and Release 2 sources bases differ significantly and also for moving discrete features between releases.

Cascade Model

The company noticed that in most circumstances Release 2 is a superset of Release 1 — Release 2 contains all of the features of Release 1, plus bugfixes and new features. When this relationship exists, a cascading hierarchy of development hierarchies has been found to be very effective. In the cascade model the new Release workspace is a child of the previous release's Release workspace.

Linking development hierarchies in this manner has a been found to have a couple of advantages:

- As the Release 1 source base changes, the changes can easily be brought over for integration in Release 2.
- When development hierarchies are related in this manner they are displayed together in the `codemgrtool` workspace graph pane and their relationships can be easily determined using the CodeManager `workspace children` and `parent` commands.

Figure 5 shows the relationship between Release 1 and Release 2 in the cascade model.

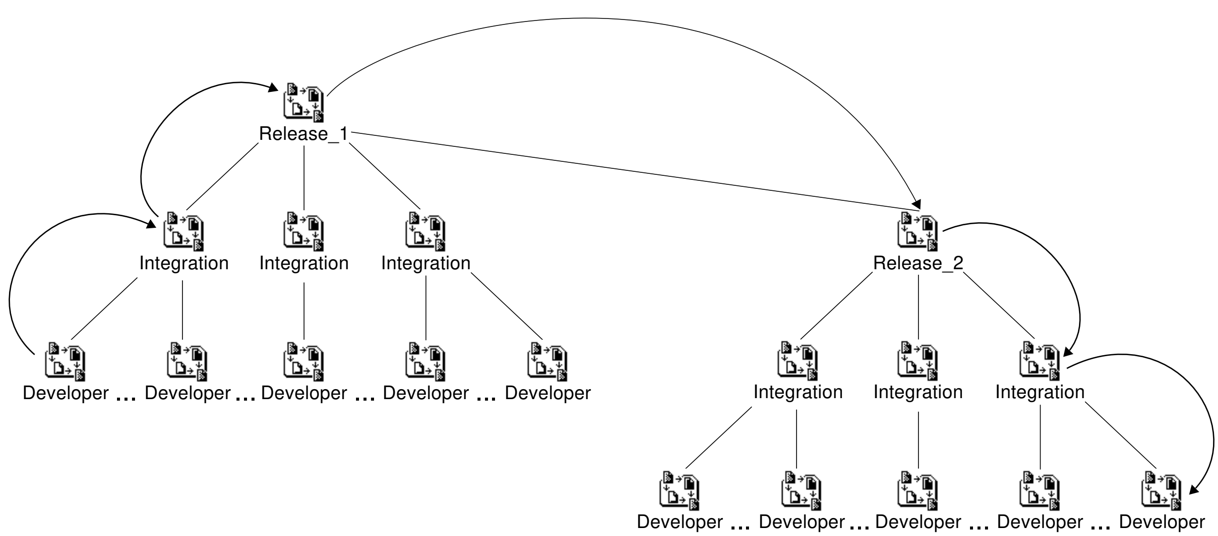


Figure 5 Updating Release 2 from Release 1 (Cascade Model)

Note that changes are *always* brought down *from* the older release *to* the newer release. In this case, as new work is integrated in Release_1 it is brought over to Release_2.

Note – Since you have to integrate the features from Release_1 into Release_2, you probably want to consider bringing the work over into a clone (child) of the Release_2 workspace for integration rather than directly into Release_2. Refer to “Optimizing Integration” on page 4 for details.

Using a clone workspace provides two advantages:
- The Release_2 workspace remains open for bringovers during integration
- Unwanted changes made during the integration can be backed out by reverting to the contents of Release_2

This cascade model can be extended to include a cascade of multiple consecutive release hierarchies.

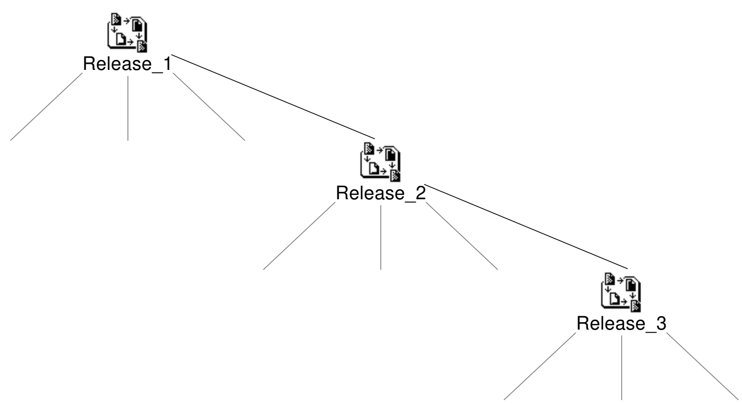


Figure 6 A Three-release Cascade

Reparenting Model

The development groups found that in some situations it is not practical to transfer changes between hierarchies through its release workspaces. In these cases they decided that it is better to propagate discrete features directly from the development level of a release hierarchy to the development or integration level of another. Consider the following cases:

- A feature intended for Release 1 cannot be included in that release due to time-to-market considerations, but *will* be included in Release 2. The feature's development workspace can be reparented directly to the Release 2 hierarchy. The company calls this process "late binding of a feature to a release."
- The source base for Release 2 diverges so significantly from Release 1 that Release 2 cannot be a child of Release 1. If a Release 1 developer implements a feature that happens to be compatible with Release 2, the development workspace in which the feature is developed can be cloned and reparented directly from Release 1 to the Release 2 development hierarchy.
- A Release 2 developer finds and fixes a bug that is also applicable to Release 1. That developer's workspace can be cloned and reparented to the Release 1 integration workspace, or directly to the workspace of a developer working in the analogous area in the Release 1 hierarchy.

Note – Whenever you transfer work between workspaces it is extremely important that you guard against inadvertently contaminating the reparented workspace with changes that you do not intend to transfer. One way to help prevent this type of contamination is to carefully select and limit the files that you transfer.

Figure 7 shows the steps involved in moving a feature between development hierarchies.

1. The CodeManager reparenting command is used to change the FeatureX workspace's parent from the Release 1 hierarchy to Release 2.

   Note: If you do not plan to delete FeatureX from Release 1, you should use the bringover create transaction to create an identical (clone) copy of the FeatureX workspace, then reparent the *clone* workspace to the Release 2 hierarchy.

2. The bringover update transaction is used to synchronize the newly reparented FeatureX workspace with the work done in Release 2.

   Note: Contents of FeatureX are now merged with work done for Release 2 and should not be reparented back to the Release 1 hierarchy.

3. After the new features have been merged and tested with the work done in the Release 2 hierarchy, the feature is put back to the integration workspace.

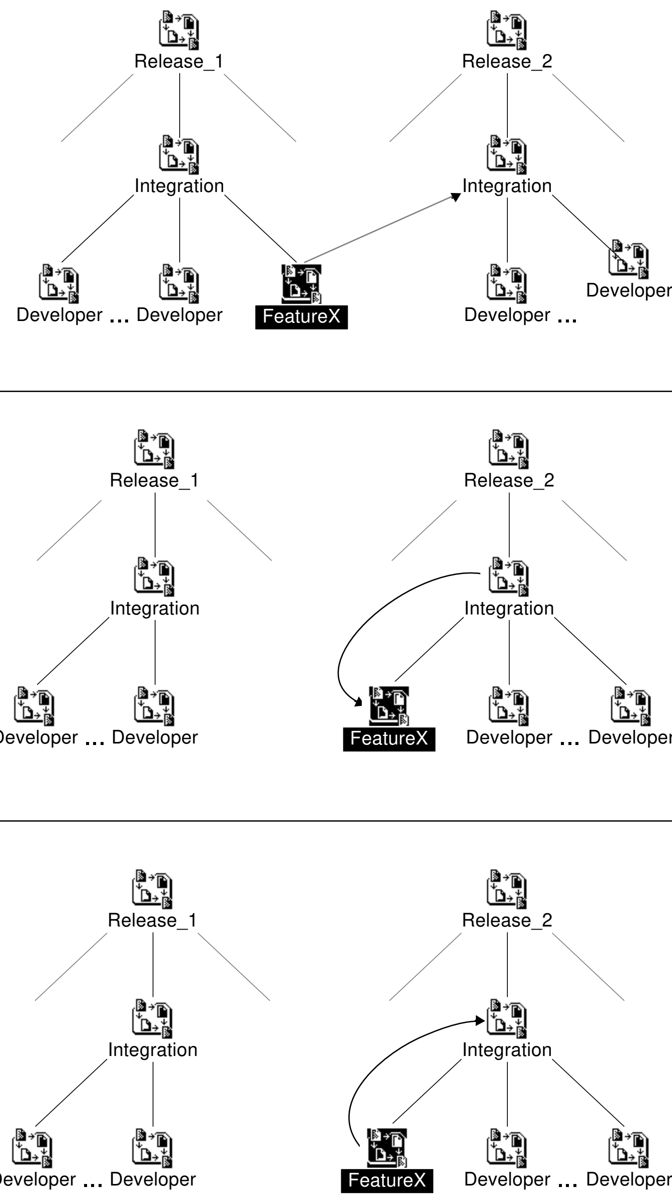


Figure 7 Late Binding of a Feature to a Release

Unanticipated Releases

In addition to regularly scheduled releases, the company finds itself generating releases that were not initially planned. These unanticipated releases generally fall into two categories:

- Minor releases
- Patch releases

The company developed the following strategies to accommodate these releases. Both are variations on the cascade and reparenting models discussed in the previous sections.

Minor Releases

A minor release is one that contains the entire product, much like a regularly scheduled release. The company denotes these types of releases with numbering schemes such as "1.1" or "1.0.1." They have found that it works well to insert these releases into the product release cascade.[1]

The following steps insert a minor release into the release cascade.

1. Release cascades are described in "Cascade Model" on page 10.

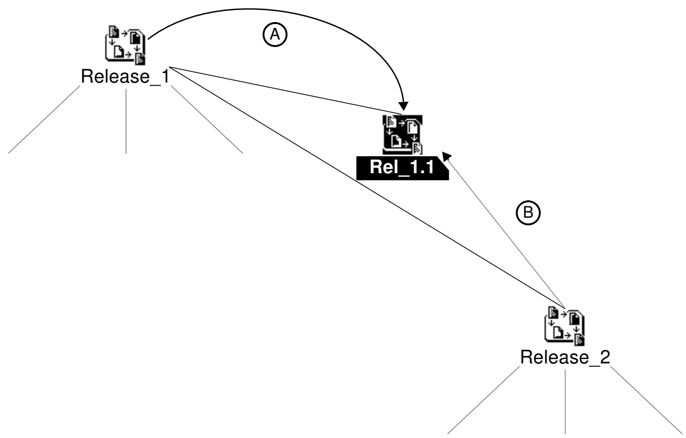


Figure 8 Inserting a Minor Release into the Release Cascade

1. **A child of the Release 1 release workspace is created using the bringover create transaction (A).**
   A new development hierarchy is set up under the Release_1.1 workspace in which to develop Release 1.1.

2. **The CodeManager reparenting command is used to reparent Release_2 to Release_1.1 (B).**

3. **The Release 2 workspace is periodically updated with changes made for Release 1.1 (C).**

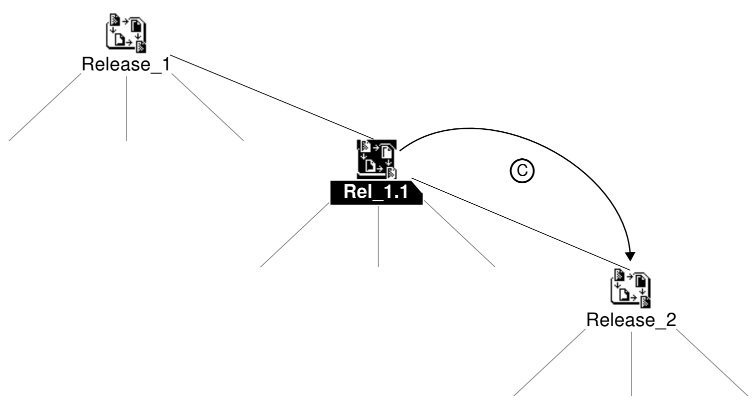


Figure 9 Updating Release 2 from Release 1.1

By inserting these minor releases into the cascade:

- It is easy to propagate the changes down the line to all releases being supported
- The hierarchies are displayed together in the `codemgrtool` workspace graph pane and their relationships can be determined using the CodeManager `workspace children` and `parent` commands.

Patch Releases

A patch release is one in which only a small portion of the product is released in order to correct a defect in the product. The company has found that this type of change usually best propagated to other releases using the reparenting model.[1]

For example, a serious bug is found in Release 1 of the company's industrial product and the company decides to produce a patch that can be distributed to customers that encounter the problem.

Figure 10 shows a patch release workspace inserted into a release cascade.

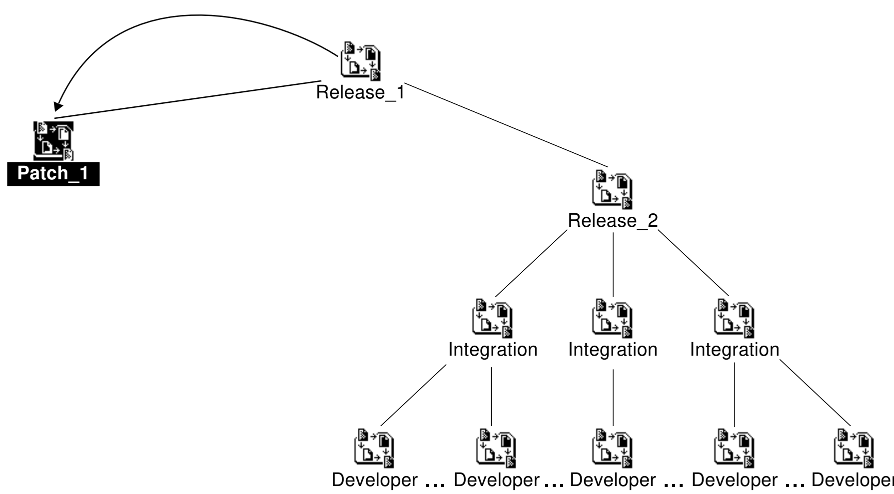


Figure 10 Creating a Patch Release

1. This model is described in "Reparenting Model" on page 11.

The following steps create a Release 1 patch release workspace and incorporate the patch into the Release 2 development hierarchy:

1. **The bringover create transaction is used to create a child workspace (Patch_1) of Release_1 (Figure 10).**

2. **A development hierarchy is created with Patch_1 as the top-level release workspace.**
   The size and depth of this hierarchy depends on the number of developers assigned to work on the patch release. Generally, patch development hierarchies are relatively small.

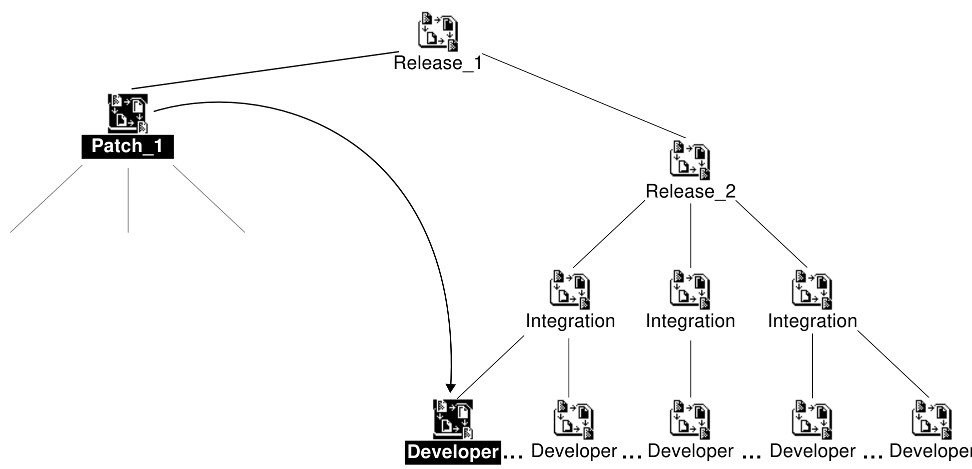


Figure 11 Reparenting the Patch Workspace

3. **After the patch is developed, tested and released, a developer from the Release 2 development hierarchy uses the bringover transaction to integrate the patch into his workspace in the Release 2 hierarchy.**
   This is best accomplished by reparenting "on-the-fly" as part of the bringover transaction (specify an alternate parent in the Transactions Window or use the `-p` option on the command-line).

Notes

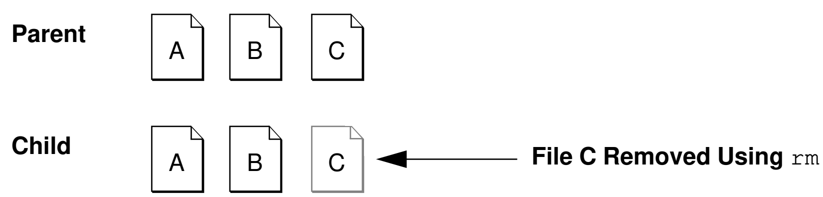
- The process described above is sometimes reversed. Bugs are discovered and fixed in the Release 2 hierarchy and ported *back* to Release 1 as a patch. In this case the clone is made from the Release 2 hierarchy and reparented to the Patch_1 workspace.
- Whenever you transfer work between workspaces it is extremely important that you guard against inadvertently contaminating the reparented workspace with changes that you do not intend to transfer. One way to help prevent this type of contamination is to carefully select and limit the files that you transfer.
- The contents of the patch workspaces are never put back to the Release 1 workspace. Putting the changes back to the release workspace would change the contents of the release archive.
- Over their lifetime, releases often end up with multiple patches. When this occurs you can either make each patch release a separate child workspace of the Release_1 workspace, or you can start a cascade of patch release workspaces like those described for release workspaces in "Cascade Model" on page 10. The cascade model is most useful if the patches you release are cumulative — where the most recent patch contains everything released in all of the previous patches.

TW Topic:

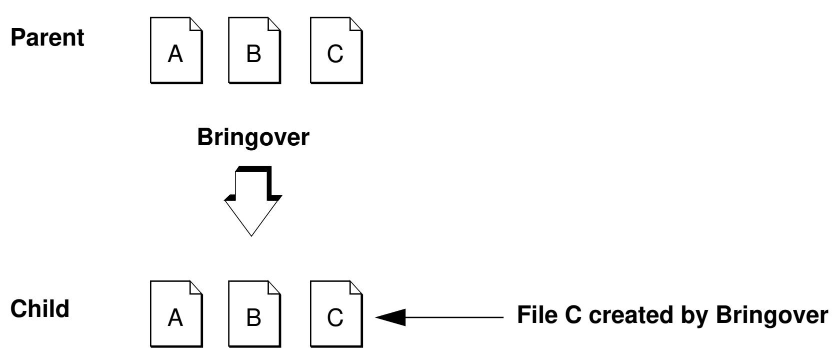
Deleting Files From CodeManager Workspaces

Deleting files from a CodeManager workspace is a little trickier than it first appears. Deleting a file from a workspace with the `rm` command causes CodeManager to think (during Bringover and Putback transactions) that the file has been newly created in the workspace's child or parent.

Take for instance, the following example. The file "C" is removed from the child workspace using the `rm` command.



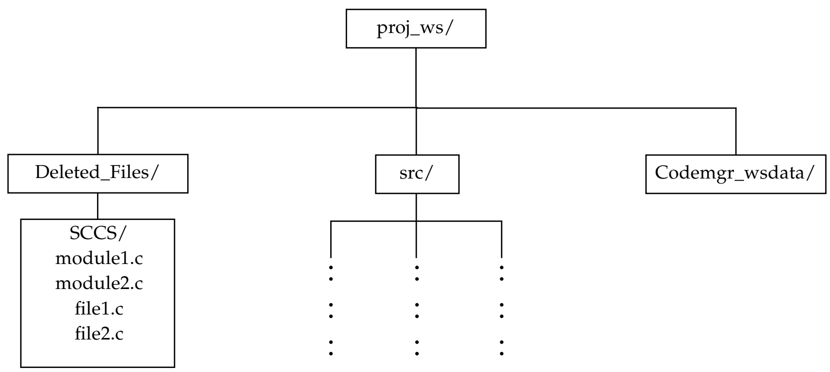
Later, the Bringover Update transaction is used to update the child. CodeManager examines the two workspaces and determines that the file "C" exists in the parent and not in the child — following the usual CodeManager rules, it creates "C" in the child.



The recommended method for "deleting" files in workspaces is to rename them out of the way using a convention agreed upon by everyone working on the project. Two possible methods are described below.

1. **Move deleted files to a "deletion" directory.**
   Move (rename) files to an agreed upon directory that will be present in all the project's workspaces. For example:

```
example% mv proj_ws/src/gui/module.c proj_ws/Deleted_Files/module.c.06.24.93
example% mv proj_ws/src/gui/SCCS/s.module.c proj_ws/Deleted_Files/SCCS/s.module.c.06.24.93
```



Experience has shown that it is important to ensure that the names of the files you move into this directory are made unique in some way so that deletions do not bump into each other. For example, if you delete files named `/ws/tape/driver/s.io.c` and `/ws/disk/driver/s.io.c`, by moving both to `/ws/Deleted_files/s.io.c`, CodeManager only propagates the change of the last file moved.

Two ways you can prevent this problem are:

- Append the date of the deletion to the file name (as in the example above)
- Move the full workspace-relative path name of the file beneath the `Deleted_Files` directory.

Note – Remember to move *both* the SCCS history files and the g-files.

Using this method:

- CodeManager moves the "deleted" files completely out of the source path.
- CodeManager does not recreate the file.
- CodeManager propagates the change throughout the workspace hierarchy as a rename, automatically "deleting" the file in all workspaces.
- The file remains available to later reconstruct releases for which it was a part (for example, if it was part of a checkpoint — see *VersionTool and CheckPoint User's Guide* for more information about checkpoints)

2. **Move deleted files to a special name in the same directory.**
Rename files you wish to "delete" so that they begin with some agreed upon prefix; for example the `.del-` prefix.

```
example% mv module.c .del-module.c
example% mv SCCS/s.module.c SCCS/s..del-module.c
```

This method provides the benefits mentioned in Step 1, but keeps the file in the directory in which it originally resided.

TW Topic:

Remote Software Development Using CodeManager

Software projects are sometimes developed by groups in which all the developers cannot share a common file system. Developers in these groups may be separated geographically and/or electronically. This separation can further complicate the already complex task of coordinating the work of development groups. CodeManager is exceptionally well suited to assist groups to coordinate the flow of code developed concurrently under these circumstances.

This topic discusses how a group of developers at a U.S. software company successfully use CodeManager to coordinate concurrent compiler development between the United States and Russia.

Scenario

The U.S. company has contracted with a group of Russian computer scientists in Moscow to assist with compiler development. Most new development is done at the company's U.S. facility, while the Russian group focuses largely on fixing bugs. As a result of this arrangement, both groups are often working simultaneously on the same files.

The compiler development group decided that CodeManager provides the means by which they can keep work at both sites synchronized and manage the transfer of files and merging of changes made simultaneously at both sites.

Implementation

This section describes the process the compiler group uses to coordinate remote development between the U.S. and Russian sites. The first section discusses how the group decided on a remote development strategy and the second section discusses tactics they considered for transferring data between sites.

The Remote Development Process

After analyzing the situation in great detail, the group came to the following conclusions. These conclusions formed the basis of their final remote development process.

Maintain Equivalent Source Bases At Both Sites

In order to coordinate their work the group realized that it was necessary for both sites to contain complete source hierarchies and for the hierarchies to be kept as synchronized as possible. It was recognized that it is important to update each site with changes made by the other site as frequently as possible.

Use Bridge Workspaces for Data Transfer

In order to make it quicker and easier for developers to access the changes made by the other site, the group decided that each local site treat the other (remote) site as if it is (hierarchically) another local developer. To accomplish this they created *Bridge* workspaces at each site at the same hierarchical level as the other developers. Bridge workspaces are regular CodeManager workspaces that are dedicated exclusively to transferring data between the sites. Bridge workspaces are used to:

- Bring over updates from the Integration workspace in preparation for shipping them to the other site
- Put back updates to the Integration workspace from the other site

The primary goal is to keep both Bridge workspaces synchronized.

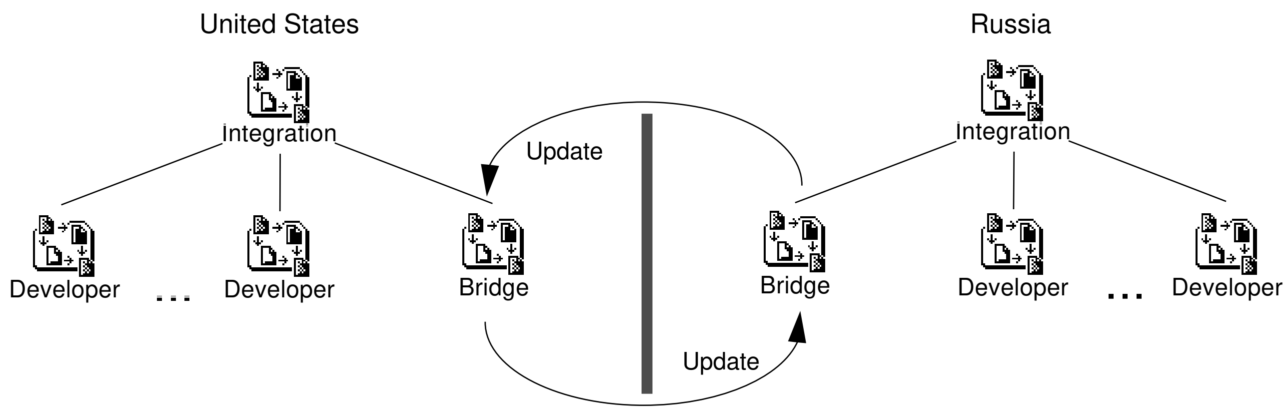


Figure 12 Workspace Hierarchies with Bridge Workspaces

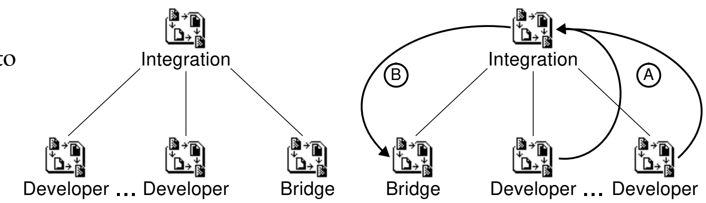
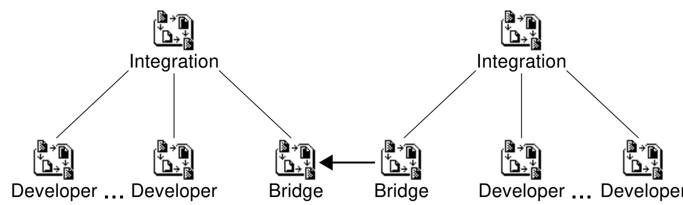
Update Each Site Sequentially

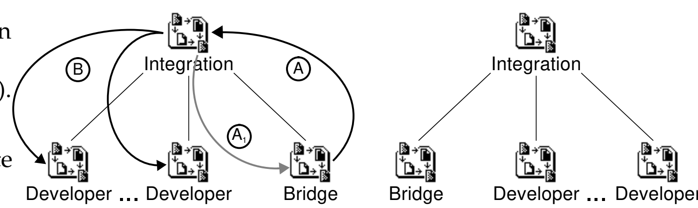
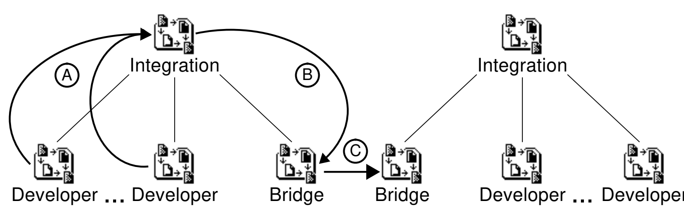
For the sake of reliability, the group realized that data must move between the sites *sequentially* — only one site is updated at a time. At first glance, it seems seductively more efficient to have both sites ship their updates to the other site at the same time. Upon further analysis it was determined that this method does not insure that changes are propagated correctly.

To maintain consistency, CodeManager enforces a single-writer model — only one transaction that alters data can proceed in a workspace at a time. Concurrency is achieved by the developers working concurrently in their own workspaces and then updating a central integration workspace sequentially, one at a time. From CodeManager's perspective, both Bridge workspaces in this model function together as a *single* virtual workspace.

If both sites execute bringover transactions to their Bridge workspaces simultaneously, it is equivalent to executing two simultaneous bringover transactions from two different workspaces into a single workspace. This breaks the CodeManager single-writer paradigm; under normal (non-remote) circumstances CodeManager does not permit this to happen.

<table>
<tr><th></th><th>U.S.</th><th>Russia</th></tr>
<tr><td>1. Russian developers put back changes to the Integration workspace (A). These changes are then brought over into the Bridge workspace (B).

Note: Throughout this process, development work continues at both sites. These figures highlight only the specific transactions involved in remote transfers.</td><td colspan="2">
</td></tr>
<tr><td>2. The Bridge workspace is copied to tape and shipped to the U.S. By one of the methods discussed in “Transferring File Changes Between Sites” on page 27, the data from the Russian Bridge workspace is incorporated into the U.S. Bridge workspace.</td><td colspan="2">
</td></tr>
<tr><td>3. The changes are put back to the U.S. Integration workspace (A). If conflicting changes have been made in the Integration workspace, the conflicting files are first brought over to the Bridge workspace to be resolved (A_1).

The U.S. developers bring over the changes from the Integration workspace into their workspaces (B), integrate the changes and add changes of their own.</td><td colspan="2">
</td></tr>
<tr><td>4. The U.S. developers put their changes back to the Integration workspace (A). The changes are then brought over into the U.S. Bridge workspace (B). The Bridge workspace is copied to tape and shipped to Russia (C). By one of the methods discussed in “Transferring File Changes Between Sites” on page 27, the data from the U.S. Bridge workspace is incorporated into the Russian Bridge workspace.</td><td colspan="2">
</td></tr>
<tr><td>5. The changes are put back to the Russian Integration workspace (A). If conflicting changes have been made in the Integration workspace, the conflicting files are first brought over to the Bridge workspace to be resolved (A_1).

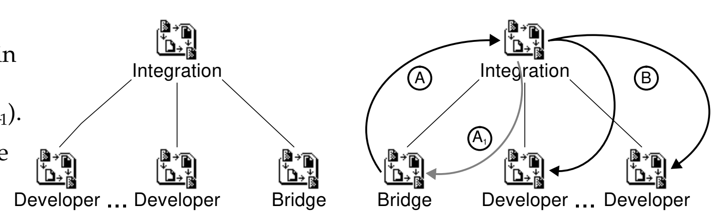
The Russian developers bring over the changes from the Integration workspace into their workspaces (B), integrate the changes and add changes of their own.</td><td colspan="2">
</td></tr>
</table>

Figure 13 A Remote Update Cycle

Consider the typical update cycle pictured in Figure 13. At the conclusion of Step 2 and Step 4 the Bridge workspaces at both sites *must be identical*. The putback transaction in Step 3 (A) is exactly the same as if it were executed directly from the Russian Bridge workspace and the putback transaction in Step 5 (A) is exactly the same as if it were executed directly from the U.S. Bridge workspace — from a CodeManager perspective, both Bridge workspaces function together as a single workspace.

Transferring File Changes Between Sites

After the group determined the fundamental model they would use, they had to decide how they intended to transfer the updates between the two sites. After analyzing the situation they realized that there is an inherent trade-off — the *less* information they transferred between sites, the *more* additional overhead was required.

After careful evaluation they determined that they had the following three options.

Table 1 Data Transfer Options

Option	Description
1	Transfer the entire Bridge workspace between the sites
2	Transfer a workspace that contains only the files that have changed (files that are updated, newly created or renamed)
3	Transfer only: • Differences (created using the `diff` command) created in the SCCS history files that have changed • The entire SCCS histories of files that were newly created • A list of the names of files that have been renamed • A workspace "checksum" created using the CheckPoint command that is used to verify that the two Bridge workspaces are identical after each update

The group decided that it was crucial for them to be able to transfer data via email — this eliminated Option 1. They felt it important to reduce as much as possible the amount of data they send between the sites, even at the added cost of additional administrative work at both sites — for this reason they chose Option 3.

The following three sections describe the three different options in some detail and present advantages and disadvantages of each.

Option 1

Transferring the *entire* Bridge workspace makes the process virtually the same as if the Russian site's Bridge workspace was actually a child of the U.S. Integration workspace (and vice versa). With this option, except for the transit delay, CodeManager is used normally.

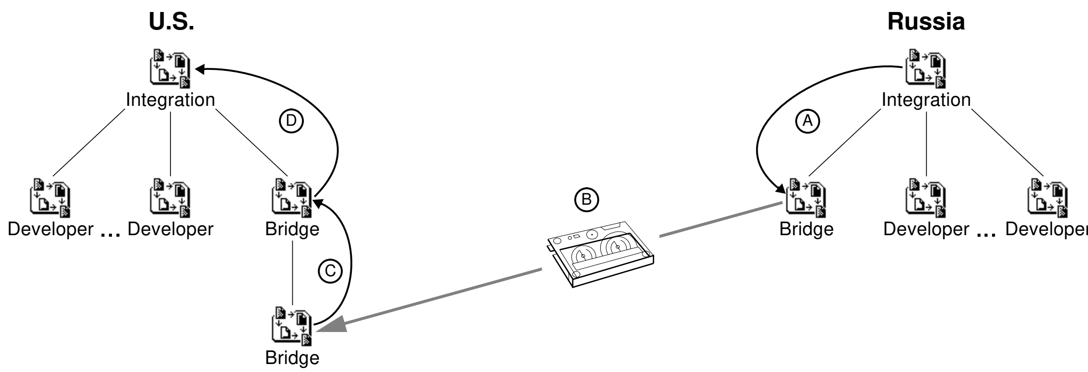


Figure 14 Option 1: Transferring the Entire Bridge Workspace

1. **The Russian Bridge workspace is updated from the Integration workspace (A), copied to tape and shipped to the U.S. site (B).**

Note – As mentioned in the previous section it is *crucial* that the two Bridge workspaces stay synchronized. After it has been updated with changes from the Integration workspace, the Russian Bridge workspace must not be altered until it is in turn updated with changes from U.S. The CodeManager access control facility can be used to lock the workspace against any changes.

---

2. **The update is received at the U.S. site and the Russian Bridge workspace is extracted.**

3. **Using the CodeManager reparenting facility, the newly arrived Russian Bridge workspace is reparented to the U.S. Bridge workspace.**[1]
   Note that this step is necessary even if the names of the workspaces are the same at both sites because the fully qualified path names of the parent workspaces are probably different both sites. CodeManager stores the names of a workspace's parent and child workspaces as fully qualified path names.

4. **The U.S. Bridge workspace is updated with the work of Russian developers using the putback transaction (C).**
   At this point the U.S. Bridge workspace is identical to the current Russian Bridge workspace. After the update is complete, the copy of the Russian workspace can be deleted.

5. **The information in the newly updated Bridge workspace is putback into the U.S. Integration workspace (D).**
   The information is now available to the U.S. developers.

6. **Reverse the previous steps to update the Russian site with changes made in the U.S.**

Option 2

Transfer a workspace that contains only the recent updates to the Bridge workspace.

The *advantage* of this method over Option 1 is that you can greatly reduce the amount of data you transfer between sites. This reduction in data size might enable you to transfer updates via email, saving time and expense over other methods such as shipping tapes.

The *disadvantage* of this method over Option 1 is that more human intervention is required. A few additional steps are added to the process and people are required to do more work.

---

1. You can save a step in the process by simply replacing the U.S. Bridge workspace with the Russian Bridge workspace, and then reparenting the Russian Bridge workspace directly to the Integration workspace. However, by circumventing the putback transaction, CodeManager cannot protect you from losing any data that may have been inadvertently copied to the U.S. Bridge workspace. In addition, the CodeManager history facility will not record the transactions for later inspection.

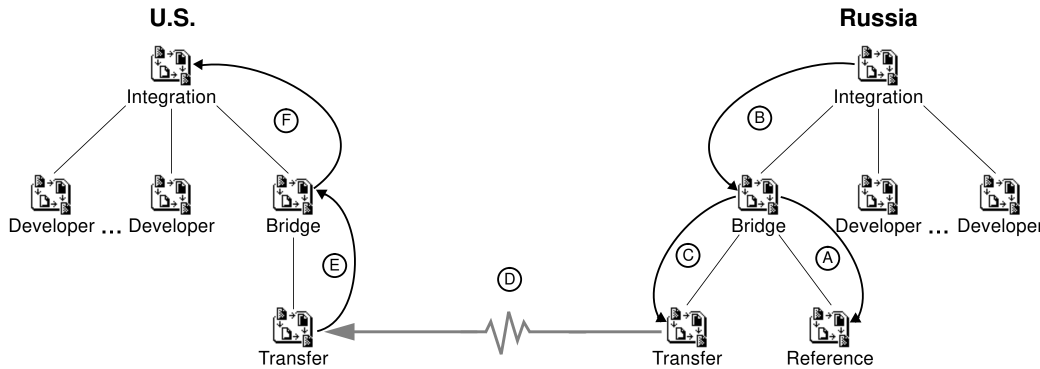


Figure 15 Transferring Updated, Created or Renamed Files (Option 2)

1. **A Reference workspace is created as a child of the Russian Bridge workspace (A).**
   This workspace is used to preserve the original state of the Bridge workspace.

2. **The Russian Bridge workspace is updated from the Integration workspace (B).**

3. **The Bringover transaction (with the Preview option selected) is used to compare the Bridge workspace to the Reference workspace.**
   This produces a list of files that have been updated, created or renamed.

4. **The files that are listed as updated, created or renamed are brought over into a newly created Transfer workspace (C).**
   The Transfer workspace can be compressed and encoded using standard UNIX utilities such as `compress`, `uuencode` and `tar`. The resulting file can then be sent via email to the U.S. site.

5. **The Transfer workspace is emailed to the U.S. where it is uncompressed (D).**

6. **The Transfer workspace is reparented to the U.S. Bridge workspace.**
   At this point the contents of the U.S. Bridge workspace are still identical to the Russian Bridge workspace *before* its most recent update.

7. **The contents of the Transfer workspace are put back into the Bridge workspace (E).**
   At this point the U.S. Bridge workspace is identical to the current Russian Bridge workspace. The Transfer workspace can be deleted after the update is complete.

8. **The contents of the U.S. Bridge workspace are put back to the Integration workspace (F).**

Option 3

It is possible to reduce even further the amount of data transferred between sites. There are however, costs for this reduction:

- The process is significantly more complicated
- More human intervention is required
- The potential for data corruption increases

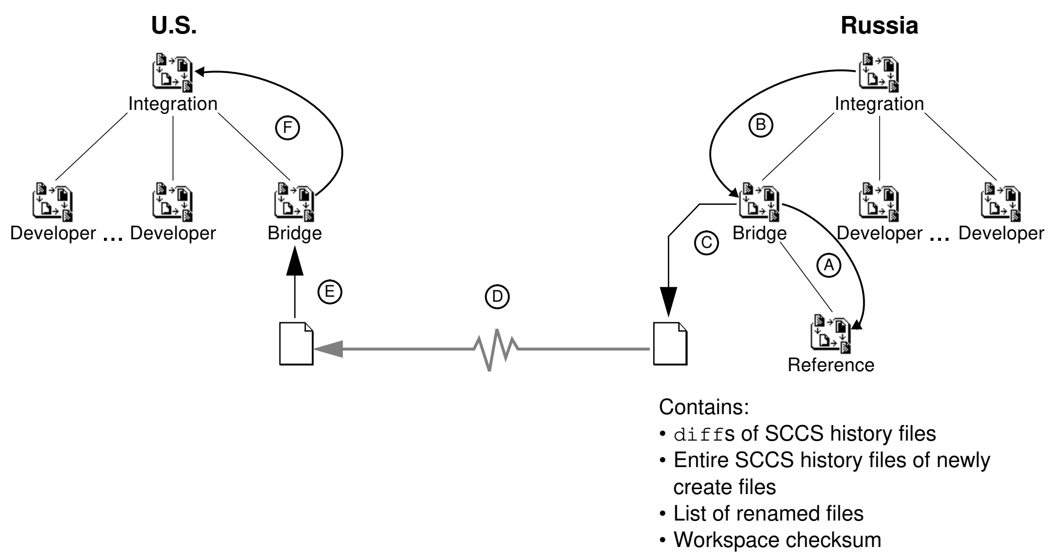


Figure 16 Transferring only diffs of Changed SCCS History Files (Option 3)

1. **A Reference workspace is created as a child of the Russian Bridge workspace (A).**
   This workspace is used to preserve the original state of the Bridge workspace.

2. **The Russian Bridge workspace is updated from the Integration workspace (B).**

3. **The Bringover transaction (with the Preview option selected) is used to compare the Bridge workspace to the Reference workspace.**
   This produces a list of files that have been updated, created or renamed.

4. **The SPARCworks/TeamWare CheckPoint utility is used to create a "checksum" from the Russian Bridge workspace.**
   This checksum will be used at the U.S. site to confirm that the U.S. and Russian Bridge workspaces are identical after they are updated.

5. **The `diff` utility is used to determine the differences between SCCS history files in the Bridge workspace and the Reference workspace.**
   The SCCS history file of each file that has been updated is compared to its previous state in the Reference workspace. The output of the `diff` utility will be applied (using the `patch` utility) to the same files in the U.S. Bridge workspace to recreate the current state of the Russian Bridge workspace.

6. **The data required to make the U.S. Bridge workspace identical to the Russian Bridge workspace is packaged into an email file (compressed, encoded, etc.) (C) and mailed to the U.S. site (D).**
   The information required for this process includes:
   - The `diff` results for all updated files
   - The entire SCCS history files of any newly created files
   - A list of all files that have been renamed
   - The workspace checksum created in Step 4

7. **The U.S. Bridge workspace is updated using the information sent from Russia (E).**
   - As required, files are renamed using the `mv` command so that they correctly match the new names in the Russian Bridge workspace
   - The `patch` utility is used to apply the `diff` output to each updated file
   - History files of newly created files are copied into the appropriate directories

8. **The updated U.S. Bridge workspace is verified to be identical to the Russian Bridge workspace.**
   CheckPoint is used to create a checkpoint file from the U.S. Bridge workspace. The U.S. checkpoint file is compared to the Russian checkpoint file, if they match, then the update was performed correctly.

9. **The contents of the U.S. Bridge workspace are put back to the Integration workspace (F).**

Final Notes

- As in the case of most CodeManager use, frequent, disciplined updating has been found to be helpful. The more out of sync the two sites become, the more conflicts must be merged. This group has found that by updating frequently, conflict problems can be resolved with occasional email communication.
- Remember that it is crucial that the Bridge workspaces at both sites stay synchronized.
  - After a Bridge workspace is updated with changes from the *Integration* workspace, that Bridge workspace must not be altered until it is in turn updated with changes from the other site.
  - After being updated from the *other site*, the Bridge workspaces at both sites must be identical.
- You can use CodeManager's Access Control facility to ensure that the contents of the Bridge workspaces are not inadvertently changed between update cycles.
- You can used CodeManager's Notification facility to automatically inform another site via email when changes of interest are made to the source base. This can alert developers to expect changes in areas in which they are working.
- CodeManager does not use time information to determine whether files have changed. This means that time zone changes and slight discrepancies between system clocks do not effect CodeManager's ability to determine whether files have changed.