Sun WorkShop TeamWare: Solutions Guide

THE NETWORK IS THE COMPUTER™

A Sun Microsystems, Inc. Business
2550 Garcia Avenue
Mountain View, CA 94043 USA
415 960-1300 fax 415 969-9131

Part No.: 802-5954-10
Revision: A, December 1996

Please
Recycle

Contents

Preface

The *Sun WorkShop TeamWare: Solutions Guide* is an in-depth case study containing eight scenario-based topics that help users take full advantage of Sun WorkShop TeamWare™ features.

The case study describes how the Sun WorkShop TeamWare tools are used by Sun in the development of the Solaris™ operating system:

- *Managing SunOS Development With TeamWare*

The scenario-based topics are:
- *Workspace Hierarchy Strategies for Software Development and Release*
- *Remote Software Development Using the Configuring Tool*
- *Strategies for Deploying Configuring in Software Development Organizations*
- *Getting the Most From the Configuring Tool Notifications*
- *Deleting Files From the Configuring Tool Workspaces*
- *Using Makefiles With TeamWare*
- *Using the Configuring Tool File List Programs*
- *Configuring and the File System — Outlying Files and Security Issues*

Readers of this guide are expected to have some software development experience and understand basic Sun WorkShop TeamWare concepts.

For further information about the Sun WorkShop TeamWare tools please refer to the following manuals:

- *Sun WorkShop TeamWare User's Guide*

Typographic Conventions

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

Table P-1 Typographic 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	`system%`
`$`	Bourne shell prompt	`system$`
`#`	Superuser prompt, either shell	`system#`

TW *Topic:*

Managing SunOS Development With TeamWare

This case study describes how the Sun WorkShop TeamWare™ code management software is being used in a large software development project. The study is taken from the experience of a Sun Microsystems® business unit, in its ongoing development of the SunOS™ operating system component of the Solaris® operating environment.

Project Description

The source base for the SunOS project is large — more than one million lines of code contained in over twelve thousand files. The staffing for the project is correspondingly large, with more than 100 developers working at sites across California and Colorado.

Management recognized during the project's planning stages that developers would need parallel access to the most current source — the classic one-developer-at-a-time access allowed by SCCS (the Source Code Control System) was too limiting for their needs. As a more capable alternative, they adopted Configuring, the TeamWare component that manages groups of files for groups of developers.

Configuring provides an isolated workspace for each developer and automatically records the change history of each source file. By judiciously setting up their hierarchy of Configuring workspaces, the group allows each developer access to the newest stable source code at any time. If two developers edit copies of the same file, Configuring detects the differences and presents them graphically in the Merging tool. One or both of the developers merge the changes into a single file that is then returned to the common source base.

Five integration engineers coordinate and maintain the Configuring workspace hierarchy for the project. Only a fraction of their time is spent on Configuring issues, however. Their additional duties include:

- Building the current SunOS release and packaging it for delivery to Release Engineering
- Managing earlier SunOS releases, including the build and delivery of patches
- Maintaining the makefile hierarchy used to build the project

The integration engineering staff generally agrees that TeamWare tools have reduced time-to-market for the SunOS product by lowering the overhead required to manage source code during the product's development phase. The tools are also being used to maintain earlier product releases.

Case Study Overview

This case study details the following aspects of the strategy used by the SunOS group in adopting and deploying Configuring and the other TeamWare tools:

- **Organizing Configuring workspaces** — The organization of the Configuring workspace hierarchy affects the building and testing of executables at the developer, integration, and release levels.
- **Automating builds within Configuring** — The directory structure within workspaces and the organization of makefiles both affect the ability of the `make` utility to smoothly automate the compilation and linking of programs. The SunOS group makes use of custom scripts and File List Programs (FLPs) to set environment variables for makefiles and to bring header files and libraries into workspaces automatically.
- **Coordinating network and operating system features with Configuring** — The SunOS group takes advantage of operating system features to enhance Configuring source file access control and to facilitate changing to workspace directories. They also use the Configuring notification feature to mail policy statements to developers.

Basic Configuring Concepts

This section defines five Configuring terms that will help you understand the case study.

Workspace — A directory (and its subdirectories) that has been designated for use by Configuring. Configuring identifies a workspace by the presence of a subdirectory named `Codemgr_wsdata`.

Workspace hierarchy — An organization of parent and child workspaces, much like a standard directory hierarchy. Each workspace can have a *parent* and one or more *children*, in the same way that a directory has a parent directory and subdirectories.

Putback — A transaction between a child workspace and its parent, in which changes to files from the child are copied ("put back") into the parent.

Bringover — A transaction between a child workspace and its parent, in which changes to files from the parent are copied ("brought over") into the child.

Reparent — An operation that changes the parent of a child workspace.

Organizing the Workspace Hierarchy

SunOS development is an ongoing project, so the workspace hierarchy was organized to allow for successive, sometimes overlapping, releases. Central to the organization is the concept of "the train," a constantly evolving top-level workspace that always represents the current group development effort. The train analogy results from the way product-release workspaces are generated, and is explored more fully on pages 14-16.

Beneath the top-level workspace, called `train`, is a group of sibling integration workspaces (see Figure 1). Developers bring files into their private workspaces from these integration workspaces. They also put back to the integration workspaces, never to `train` directly. Because the integration workspaces restrict access to the release-level workspace, the SunOS group calls them *gates*. In each group, a senior-level developer is designated as the *gatekeeper*. The gatekeeper is responsible for maintaining the group's gate workspace, making sure that the source it contains is without error before it is put back to the release-level workspace.

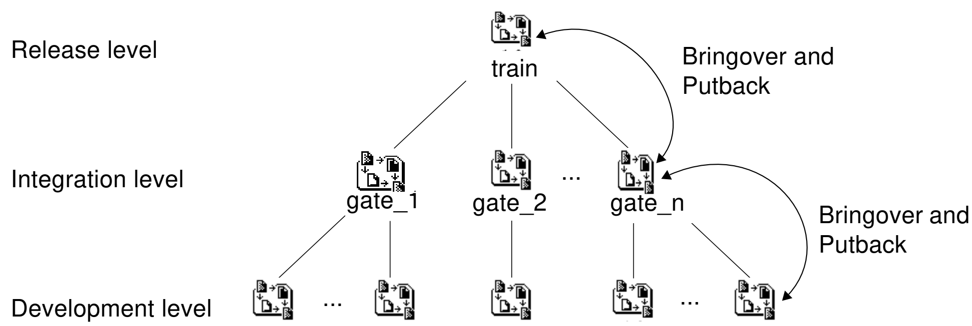


Figure 1 Basic Workspace Hierarchy

Most integration workspaces are organized functionally: developers working on a related group of commands or libraries transact with a single integration workspace. Other integration workspaces represent physical organization: developers working at a remote site transact with a single integration workspace, which puts back to `train` overnight. (Remote integration workspaces communicate with `train` through the NFS® distributed file system by means of a dedicated communications link, which slows down transactions somewhat.)

The organization shown in Figure 1 is an ideal; the actual hierarchy at any given time may look quite different because the SunOS group usually reparents workspaces during building and testing, as described in the following sections. The ability to reparent workspaces results in a great deal of flexibility. The hierarchy is a statement of group policy, not a rigid organizational structure, and can easily be revised.[1]

Building and Testing Executables

Each developer in the SunOS development group is expected to build (compile and link) and test the new code he writes. Custom File List Programs (FLPs) and hierarchical makefile structures help ensure that every development workspace builds with the most recent header files and libraries. After a

1. Developers working in close collaboration can even reparent to each other's workspaces in order to solve a problem jointly, then put back to the integration workspace when the problem is solved.

successful build and test, the developer puts back his new source files into the integration workspace. If a conflict occurs because another developer's changes were put back while the first developer was working, the first developer resolves the conflicts with the Merging tool, and the resulting source files are once again compiled, linked, and tested.

A large software development project is almost always organized into functional modules, and the SunOS project is no exception. Because of this modularity, the code that has been put back to an integration workspace cannot be guaranteed to build and test successfully even though each developer tests new code in his own workspace. While a development workspace usually tests one bug fix at a time, an integration workspace can accumulate dozens or even hundreds of putbacks between each build-test cycle. To make certain that the interaction between all these changes is correct, a senior developer is given responsibility for building and testing at the integration workspace level. Only when the testing is completed is the new code put back into the release workspace to become part of the product.

Building in a parent workspace, whether it be integration or release, requires that no child workspaces put back during the build. If putbacks were allowed, the source code base could change in the midst of compilation, almost assuring errors. This requirement clashes with the goal of letting each developer transact with his integration workspace at any time — a goal that increases productivity and decreases wait time for developers. Configuring solves the problem by allowing reparenting of workspaces, as discussed in the next section.

Isolating Test Workspaces

The SunOS development group creates identical copies ("clones") of integration workspaces in which to build and test. After a successful test of the software, reparenting is generally used to put back changes to the release workspace. Depending on the demands of the moment, testing is done in a clone of one gate or in a temporary composite workspace formed by cloning one gate, then reparenting the clone to a second gate and updating it with files from the second gate. This section describes both approaches.

Testing in a Single Integration Workspace

Consider the gatekeeper who is responsible for maintaining the integration workspace `gate_1`. The workspace has accumulated enough changes to warrant putting them back into `train`. Before putting back, however, the gatekeeper must do three things:

1. Bring over any new source files from `train` that have been put back from other integration workspaces. This step also involves resolving any conflicts that may result from the bringover.

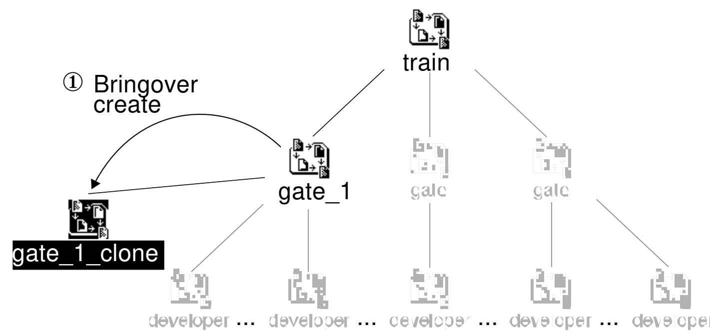
2. Compile and link the source code in `gate_1`.

3. Test the resulting executables.

If the gatekeeper brings over directly from `train` into `gate_1`, she is likely to encounter conflicts. As she resolves these conflicts (by merging source files), she will be forced to lock `gate_1` to keep developers from putting back while she works.

Instead of locking the gate workspace, the SunOS group's gatekeepers use a cloning technique that allows developers access to the gate while they resolve conflicts and fix bugs in the source base. The following figure (Figure 2) shows the steps the gatekeeper would perform to integrate `gate_1` with the release workspace `train`.

Create a clone of `gate_1` with a bringover create command (①).



Reparent to `train` (②), and then bring over into the clone (③).

After the gatekeeper resolves the conflicts that result from bringing over updates from `train`, `gate_1_clone` is no longer an exact image of `gate_1` — it has been modified by the most recent changes other integration workspaces made in `train`.

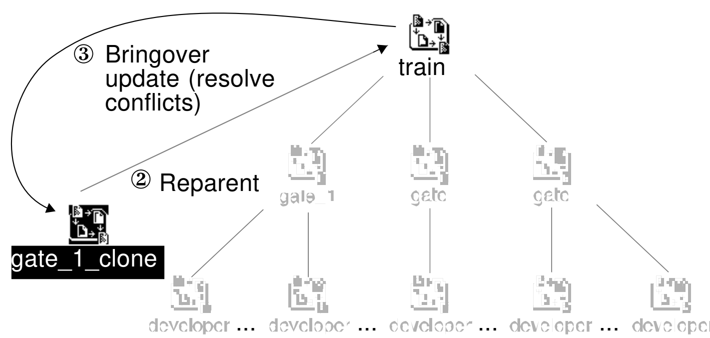


Figure 2 Integrating an Integration Workspace Into a Release Workspace

The gatekeeper must now build and test the source code in `gate_1_clone` to make sure it is correct (④). Building in `gate_1_clone` requires locking it against putbacks, but this constraint inconveniences no one because `gate_1_clone` has no children.

The build and test may reveal inconsistencies and bugs that will have to be fixed. As the gatekeeper fixes the errors, the contents of `gate_1_clone` diverge from `gate_1`.

When the gatekeeper is satisfied with the results of her testing, she puts back from `gate_1_clone` to `train` (⑤) rather than reparenting and putting back to `gate_1`.

When changes are put back from the clone directly into `train`, they become available immediately to all other integration workspaces, including `gate_1`. A bringover from `train` to `gate_1` brings `gate_1` up to date (⑥).

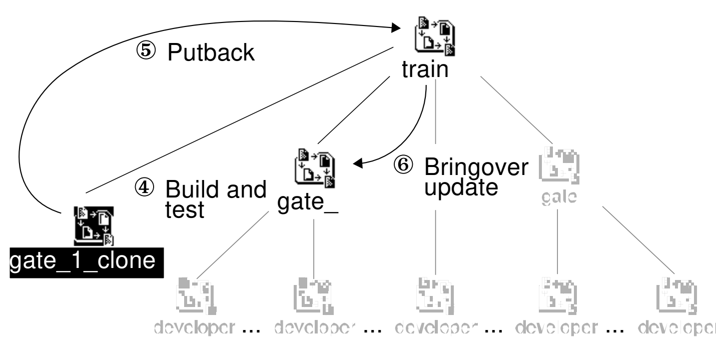


Figure 2 Integrating an Integration Workspace Into a Release Workspace (Continued)

Note that in the day or two the gatekeeper spent building and testing in `gate_1_clone`, developers may have been putting new changes into `gate_1`. The best method for synchronizing `gate_1` and `gate_1_clone` in that event involves three steps, as described in Figure 3.

Reparent `gate_1_clone` to `gate_1` (⑥).

Bring over to update `gate_1_clone` (⑦), resolving conflicts as required.

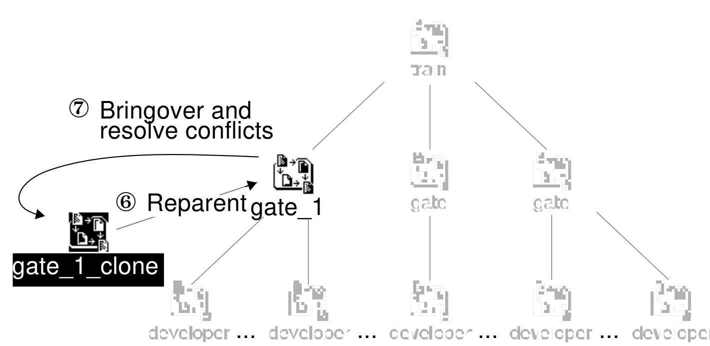


Figure 3 Synchronizing the Integration Environment With its Clone

Put back to `gate_1`, (⑧).

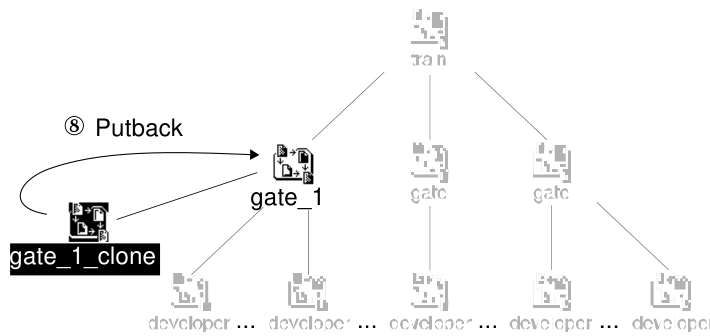


Figure 3 Synchronizing the Integration Environment With its Clone (Continued)

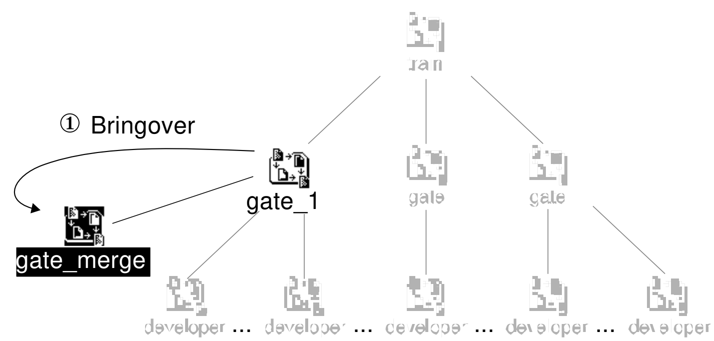
The cloning technique just outlined allows development to move forward at the same time a large, complex workspace is being tested and integrated into the release level. It's disadvantage comes when a developer puts back a large functional module into the integration workspace while testing is going on in the clone. In such a case, the final bringover and putback (⑦ and ⑧, Figure 3) can result in many conflicts and a great deal of work for the gatekeeper. These cases can be avoided by polling developers about their work in progress and scheduling the build and test operations for lull times following a developer milestone putback event.

Testing in Merged Workspaces

With a few additional steps, the process discussed in the last section can be applied to merging several gates, building and testing, and then reparenting and putting back to the release workspace. If the gates do not contain a lot of new code, gatekeepers can save time by merging the gates before building and testing them: instead of building and testing a clone of each gate, the gatekeepers build and test once for all the merged workspaces.

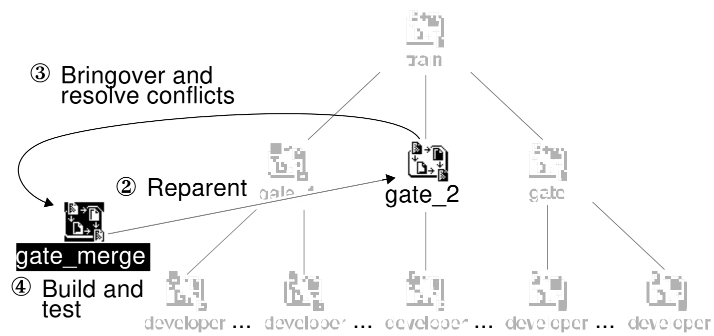
Figure 4 shows the process for two integration workspaces.

A gatekeeper creates a clone of `gate_1` by bringing over into a new workspace called `gate_merge` (①).



The gatekeeper reparents to `gate_2` (②) and brings over changes, which may create conflicts that must be resolved in `gate_merge` (③).

After the conflicts are resolved, the gatekeeper builds and tests in `gate_merge` (④).



When testing is done, the gatekeeper reparents to `train` (⑤) and puts back into the release workspace (⑥).

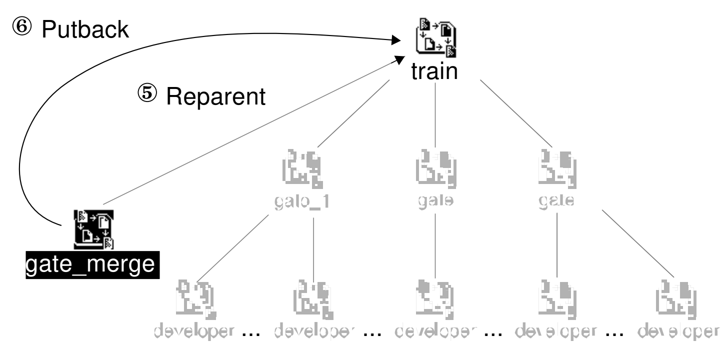


Figure 4 Merging Two Workspaces and Putting Back to the Release Level

To synchronize `gate_1` and `gate_2` with `gate_merge`, the gatekeeper could bring over from `train`. However, that bringover could create conflicts that would need to be resolved in `gate_1` and `gate_2`, which would lock out developers from those integration workspaces. A better approach is separately to reparent and put back from `gate_merge` into `gate_1` (⑦) and `gate_2` (⑧).

If the putbacks are blocked, the gatekeeper can perform new bringovers and resolve conflicts in `gate_merge`, leaving the integration workspaces free for developer transactions.

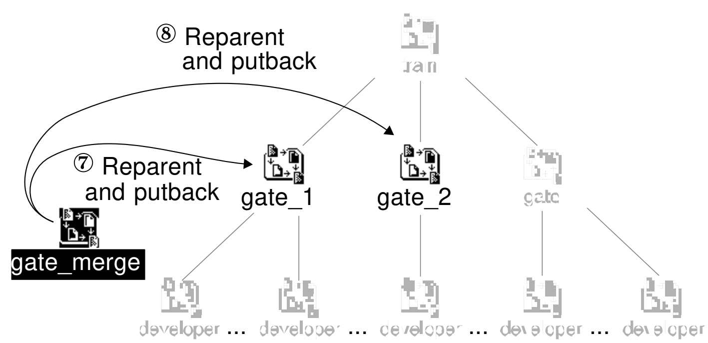


Figure 4 Merging Two Workspaces and Putting Back to the Release Level (Continued)

Clearly, more than two integration workspaces can be merged in the manner just described in Figure 4. Projects of lesser complexity may extend workspace merging to include all integration workspaces in the project, periodically merging, building, testing, and putting back to the release level. See the TW Topic "Workspace Hierarchy Strategies for Software Development and Release on page 37" for a discussion of this approach.

Managing Release-Level Workspaces

Many of the considerations that apply to integration-level workspaces also apply to the release level — providing transaction access for child workspaces during build and test cycles, for example. Additional issues at the release level involve providing for alpha, beta, and customer releases, multiple simultaneous customer releases, and maintenance (patch) releases.

As a workspace nears release, the group relies on a senior developer who is assigned to the workspace. This person decides when to merge changes into the workspace and when testing has been completed satisfactorily. The position is one of considerable responsibility. Integrators at the release level are aided by Configuring and network features that control access to source files and email notifications when workspace transactions occur.

Posting Notifications

Configuring can send email messages when an interworkspace transaction occurs (bringover, putback) or when a workspace is deleted, moved, or reparented. Integrators at the release level set up these notifications so that they are alerted whenever an interesting transaction occurs in workspaces for which they have responsibility.

The SunOS group arranged to have a special email message sent to team members who bring over files from release-level workspaces. The message is a policy statement outlining the restrictions on putting back files to the release level and explaining the release-gateway-developer workspace hierarchy. This message helps ensure that new developers do not bring over source directly from the release level into their private workspaces, from which they will not be allowed to put back to the release level.

Controlling Access to Source Files

Controlling access to workspaces is important, particularly at the release level. Configuring, through the `access_control` file in the `Codemgr_wsdata` directory, allows only specified users and net groups to perform transactions with a workspace (bringovers and putbacks) and operations on a workspace (reparentings, deletions, and so on).

Although workspace access can be controlled through Configuring commands, there is no special access control for workspaces at the file system level. Configuring does not limit the ability of a user to change directories to the workspace across the NFS distributed file system, check out a file from SCCS, edit it, and check it back in. This possibility is especially worrisome at the release workspace level, where a developer might accidentally check in a file.

To overcome this problem, the SunOS group selectively sets the permissions of the directories in its workspaces. For example, the `Codemgr_wsdata` directory is writable by anyone because workspace transactions require the ability to write to a locks file in that directory. However, the `usr` directory, which is the root of the source base, is read-only on most file systems so that it cannot be written to casually. It is read-write to only one machine, which developers must log into remotely (`rlogin`) in order to put back. The machine itself can be password protected from remote logins by unauthorized users, but more important is the fact that qualified developers must perform the extra step of logging into the machine before putting back. This extra step protects from accidental source file checkins by ensuring a conscious step on the part of developers before they put back source files.

Building and Testing at the Release Level

At the release level, building and testing is performed in a clone workspace just as it is at the integration level. However, procedures at the release level are more formalized because risks to the code base are higher. Figure 5 illustrates the process. By cloning a workspace (①) for test activity, the release workspace itself is free for transactions with integration workspaces.

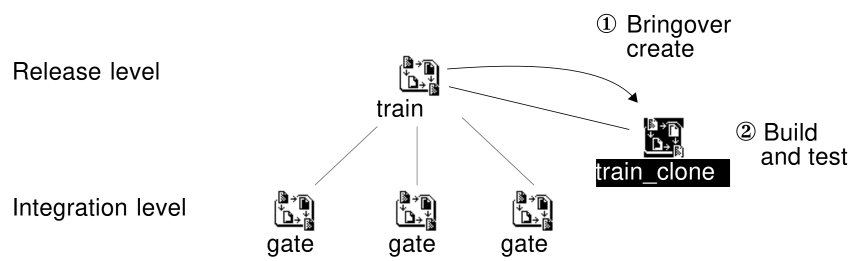


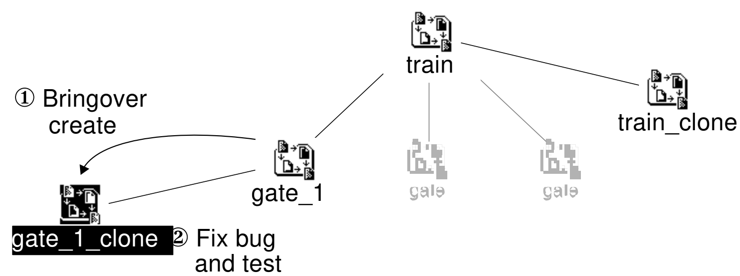
Figure 5 Release Level Building and Testing

The clone workspace is write-only — it never puts back to the release workspace. If a bug arises during testing, it is evaluated in the clone workspace and narrowed down to an area of functionality corresponding to an integration workspace. The bug is fixed at the integration level and put back to the release level.

A typical bug fix scenario is shown in Figure 6.

Here, a bug has been identified as having originated in the gate_1 workspace. A clone is made of `gate_1` (¨) so that `gate_1` itself can be free for developer transactions.

The bug is fixed and tested in `gate_1_clone` (②).



The clone is reparented to `train`, and the new code is put back to the release level (③).

From release level, the new code is brought over into `train_clone` (④) where it is tested again (⑤).

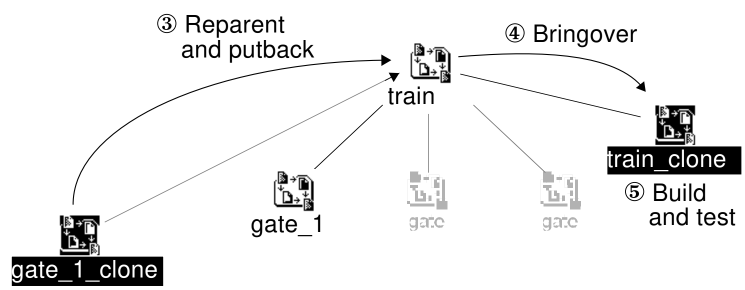


Figure 6 Release-level Bug Fixing

Because the SunOS operating system is compiled for more than one architecture (SPARC® and Intel® x86 architectures at this writing), a build clone is created for each architecture.

Managing Test Releases

If the `train` clone shown in Figure 6 is created for a test release, it can be named `Alpha` or `Beta` and maintained for the period of the test release. If it is created only to establish a stable release workspace, it can be deleted after testing is completed.

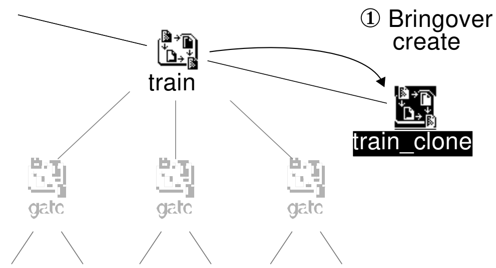
Managing Multiple Customer Releases

The SunOS group publishes an ongoing set of releases, handling at least two releases concurrently. As one release enters beta testing and First Customer Ship (FCS), development and alpha testing begins on the next release. A modified cascade workspace structure is used to support the releases (see the TW Topic “Workspace Hierarchy Strategies for Software Development and Release on page 37” for a more general discussion of workspace hierarchies).

The release-level workspace `train` is always the top of the current development hierarchy. For each release, `train` is stabilized and cloned. The old `train` workspace is renamed to designate its release number and the clone is renamed `train`. Active integration workspaces are reparented to the new `train`, and development continues on the next release. Figure 7 illustrates the process for Release 3.0.

Before Release 3.0

Release-level workspace `train` is cloned and named `train_clone` (①).



After Release 3.0

Workspace `train` is renamed `Release_3.0`, and workspace `train_clone` is renamed `train` (②). The integration-level children of `Release_3.0` are reparented to `train` (③).
An alternative reparenting technique, for hierarchies with many integration workspaces, is to name the `train` clone `Release_3.0`, then reparent `Release_3.0` to `Release_2.0` and `train` to `Release_3.0`.

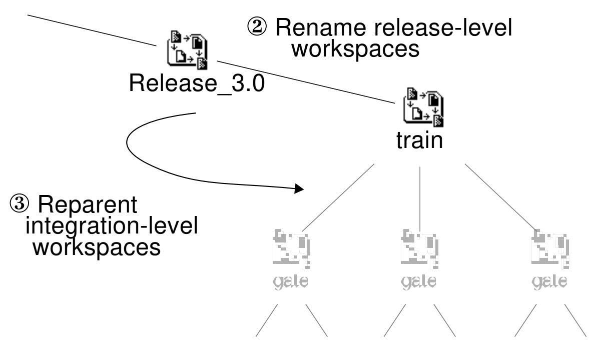


Figure 7 Leaving a Release Behind While the `train` Moves On

By reparenting all the existing development and integration workspaces to the new `train`, any work in progress that was not finished for the last release (the development of a new feature, for example) will automatically be applied to the next release.[1]

The analogy to a train is apt when the workspace hierarchy is viewed over time. Each release begins as a snapshot of the current `train`, and a string of releases takes on the appearance of a series of boxcars left behind by `train`, which proceeds along the track. From a developer's perspective, "the train" is always the active development workspace with the most recent code, always moving on to the next release (Figure 8).

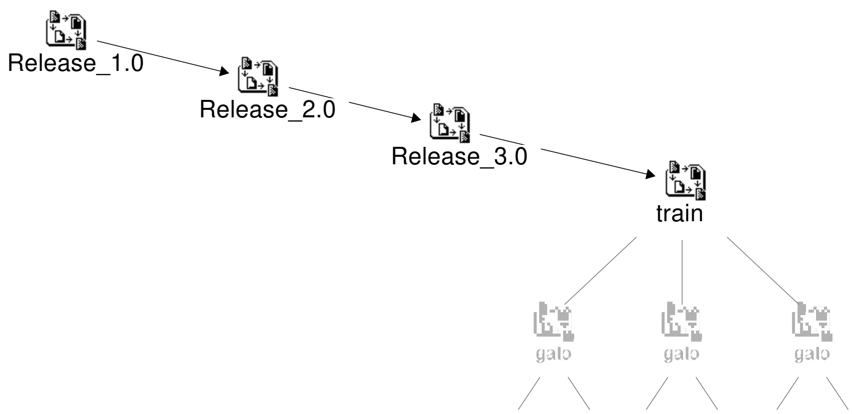


Figure 8 A String of Releases Left Behind by `train`

Managing Maintenance Releases

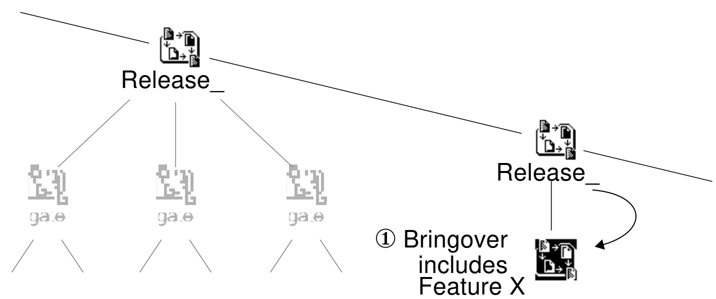
Of course, each release does not remain frozen in time. As bugs are reported against a particular release, child workspaces are created in which to fix the bug. Moreover, a bug fix applied to a shipping release may also be applied to later release workspaces, including `train`. This is the normal case for the SunOS project because each succeeding release is a superset of the previous release.

Note that the arrows between the release workspaces in Figure 8 show a one-way information flow. Bug fixes from earlier releases are brought over into later releases, but no code is ever put back into an earlier release. The major reason for this formality is to avoid introducing incompatible features from a recent release into older releases.

1. And, if it is not finished for the next release, it can be applied to the one after that.

In a similar vein, a child of a late release is never allowed to reparent into the hierarchy of an earlier release because of the risk of putting back a late feature into an earlier release. Figure 9 illustrates what can go wrong.

A senior developer creates a child of Release 3.0 in order to fix a bug (①).
Some of the code he brings over implements Feature X, unknown to him. Feature X has been developed for Release 3.0, but is not intended for Release 2.0.
He fixes the bug, which is entirely unrelated to Feature X, and decides to apply his bug fix to Release 2.0.



He reparents to an integration workspace under Release 2.0 (②) and puts back his changes, inadvertently putting back Feature X at the same time. The next time the integration workspace puts back to Release 2.0, it will include Feature X in the files it puts back.

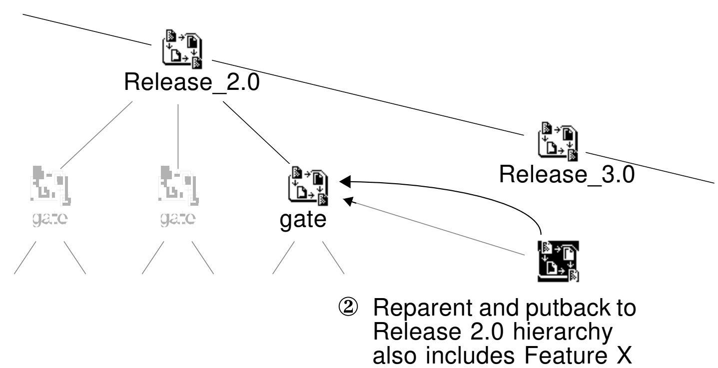


Figure 9 Problems in Reparenting From a Recent Release to an Earlier Release

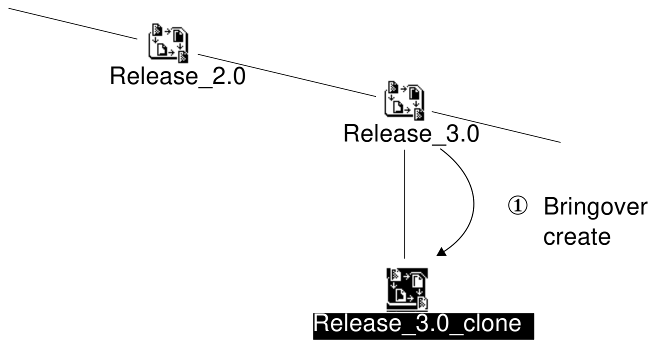
The inadvertent contamination of earlier releases illustrated in Figure 9 is difficult to guard against and can be difficult to fix. The best defense is to strongly discourage such reparenting, as the SunOS group does. Instead, they fix the bug in the earliest release to which it applies and bring the changes forward along the release chain.

Bringing Changes Into a Recent Release

When changes are brought over from an early release workspace into a later release, they are almost always brought over into a clone of the later release workspace (not into the later release directly). For example, when the person responsible for release-level integration wants to bring changes from the Release 2.0 workspace into Release 3.0, she creates a clone of Release 3.0, reparents to Release 2.0, and brings over the changes into the clone (Figure 10). After resolving conflicts and thoroughly testing the changes in the Release 3.0 clone, she puts back to the Release 3.0 workspace.

If the build reveals a major problem, the bug is investigated in the clone and narrowed down to one area of functionality in one of the release workspaces. The bug is fixed in the integration workspace where the functionality originated, and the change is put back into the release workspace. The change is then brought over into the build clone, and build and test begins again.

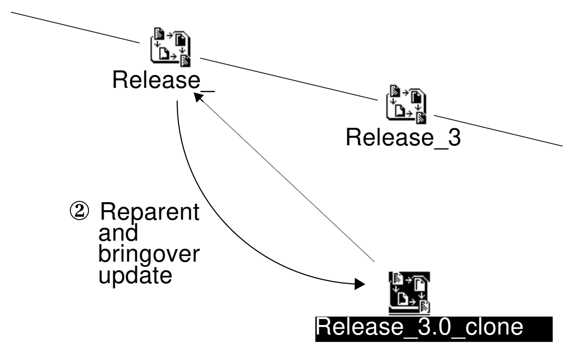
To bring Release 2.0 changes into Release 3.0, a release-level integrator first creates a clone of Release 3.0 (①).



`Release_3.0_clone` is reparented to `Release 2.0`, and a bringover update is performed on `Release_3.0_clone` (②).

If the bringover creates conflicts, they are resolved in `Release_3.0_clone`.

After the bringover, `Release_3.0_clone` becomes a merge of Release 2.0 and Release 3.0.



After building and testing the source in `Release_3.0_clone`, the clone is reparented to `Release_3.0` and changes are put back.

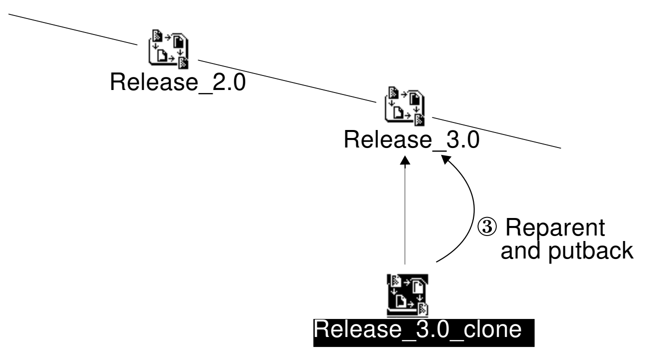


Figure 10 Bringing Changes from Release 2.0 into Release 3.0

By merging Releases 2.0 and 3.0 in a clone of Release 3.0, the release-level integrator leaves Release 3.0 free for putbacks from its child bug-fix workspaces. The penalty for allowing these putbacks is that the putback from the clone into Release 3.0 (③) may be blocked. In that case, the integrator must bring over updates from Release 3.0 into the clone, resolve conflicts, build, test, and put back again.

When the early release workspaces are active with bug fixes, the SunOS group merges changes into the current release every two or three days because the group is concerned about bringing bug fixes into recent releases as quickly as possible. And, as with all such merging, the more time that elapses between merges, the more difficult the merge becomes.

Automating Builds Within Configuring

Maintaining logical, consistent source and makefile hierarchies is important to any large project, whether or not it uses Configuring. Like most large projects, the SunOS project organizes its source files into directories along functional lines. The makefile hierarchy follows the source file organization: each directory contains one default makefile, and each makefile may include other makefiles from parent directories.

Staff engineers established guidelines that encourage developers to observe certain conventions when writing makefiles. For example, developers are requested to use designated environment variables in their makefiles and to include a master makefile that defines system-wide make targets.

To smoothly automate the build process in any workspace, SunOS integration engineers used two mechanisms to provide Configuring with insight into the source and makefile structures: they wrote an initialization script called `ws`, and they provided custom File List Programs (FLPs).

Organizing the Workspace Directory Structure

The root of each Configuring workspace is defined by the environment variable `CODEMGR_WS`. This variable contains the absolute path name of the workspace, and can be thought of as the relative root of the workspace. Beneath this relative root are three top-level directories:

- `Codemgr_wsdata`, which is found at the top of every Configuring workspace
- `usr`, which is the root of the source base directory tree

- `proto`, a prototype area that contains files used during the linking stage of a build and where new executables are installed

The `proto` directory has no makefiles because no builds are performed directly in it or its subdirectories.

Organizing the Source Directory Structure

Source files are contained in a directory hierarchy under `$CODEMGR_WS/usr/src`. However, `/usr/src` is never referred to directly by any makefile; instead, it is the value of the `SRC` environment variable, which is used instead. The full path name of the top-level source tree in an active workspace is therefore `$CODEMGR_WS/$SRC`. This indirection allows the entire source tree to be moved without altering any makefile or script.

Figure 11 shows part of the top-level source tree. Besides directories for commands, device drivers, header files, and so on, the top-level directory contains a makefile that is used to build the entire product, and a master makefile that is included in other makefiles in the source tree.

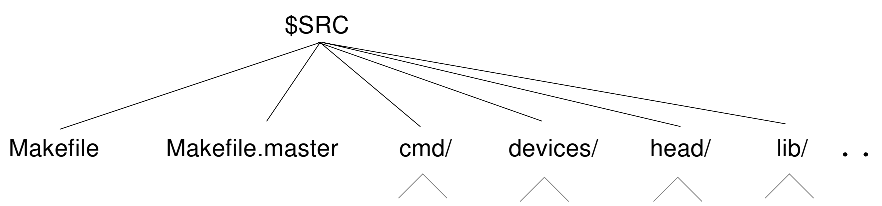


Figure 11 Partial Directory Tree for SunOS Source Hierarchy

Coordinating Makefiles

The SunOS group had the following goals in mind when it wrote its guidelines for writing makefiles:

- The makefiles have a common internal structure to make debugging easy. For example, all makefiles contain the `all` and `install` targets.
- Each makefile includes a higher-level makefile to make sure important variables are inherited. The highest-level makefile, `Makefile.master`, is included in all makefiles.

- Builds are as simple as possible, and incremental builds are supported.
- It is easy to detect if and where a build fails.
- The makefiles are optimized to be used with DMake.

An important guideline that realizes these goals is to have makefiles share as much code and as many rules as possible. Because all makefiles include `Makefile.master`, any change there (a change to the value of the `CFLAGS` variable, for example) will affect all other makefiles.

The following example shows how the policy of including parent makefiles helps keep each makefile simple. The example shows the build makefile for the `ls` command. It includes two higher-level makefiles, `Makefile.cmd` and `Makefile.targ`.

`Makefile.cmd` contains variables and rules common to the builds of all commands in the `$SRC/cmd` directory. It includes the top-level makefile, `Makefile.master`. The second included makefile, `Makefile.targ`, specifies system-wide install targets.

```
#
#ident"@(#)Makefile1.792/12/15 SMI"
#
# Copyright (c) 1992 by Sun Microsystems, Inc.
#
PROG= ls
include ../Makefile.cmd
LDLIBS += -lgen -lw -lintl
.KEEP_STATE:
all: $(PROG)
install: all $(ROOTPROG)
clean:
lint:lint_PROG
include ../Makefile.targ
```

Figure 12 Makefile for the `ls` Command

Using DMake

Large software projects typically consist of multiple independent modules that can be built in parallel. DMake supports concurrent processing of targets on a single machine; this concurrence can significantly reduce the time required to build a large project.

Table 1 shows typical times required to build the SunOS source base on three different machines. On a SPARCstation™ 2 system, the use of DMake improves performance by over 30 percent. Larger improvements are realized on higher-performance machines.

Table 1 Build Times for SunOS Operating System, Make *vs*. DMake

Build Machine	**Make**	**DMake**	**Percent Improvement**
SPARCstation 2	15 hours	10 hours	33
SPARCstation 10	10 hours	4.5 hours	55
SPARCcenter™ 2000	7 hours	1 hour	86

The `ws` *Utility*

The SunOS group wrote the `ws` utility to automatically initialize the build environment within a Configuring workspace. It configures an environment to build the SunOS executables, sets environment variables for the workspace, and spawns a shell for the environment that has been set up.

During a build, `make` relies on makefiles to define search paths for the proper header files, libraries, and installation directories. Makefiles construct these search paths by absorbing environment variables. For the makefiles to work properly, each developer must set several environment variables. Setting them manually is an error-prone chore for the developer, a chore that is eliminated by `ws`.

Header and library files undergoing development are installed in a workspace directory named `proto`. When `ws` configures the environment, it sets environment variables that define the `proto` directory to be used during the build. If the build involves installing files, `ws` also defines a (perhaps different) `proto` directory as the install target.

The ws utility was written as a shell script, which makes it both portable and easy to modify.

Using ws

Consider the developer responsible for the ls command. Only a few source files are required to build the ls executable, and these are the files she brings over into her workspace when she begins work (Figure 13). CODEMGR_WS is the environment variable that contains the absolute path name of the current workspace.

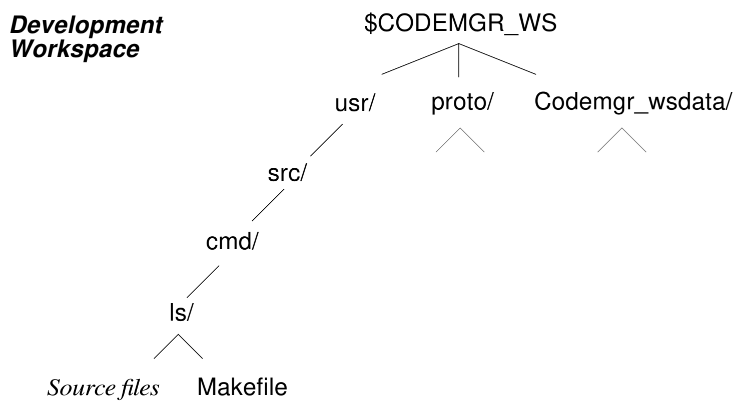


Figure 13 Directory Structure for ls Development Workspace

When the developer builds the ls executable, she does *not* want to use the native system library and header files. Instead, she wants to use the project-specific files that have been installed in proto. If proto is empty or does not exist in her workspace, she must direct make to search for the directory in ancestor workspaces during the build. The ws utility automates setting of these search paths to ensure that a build always uses the proper libraries and headers.

By setting four environment variables (named PROTO1, PROTO2, PROTO3, and TERMPROTO), the developer can set compiler flags to control the build search paths. For the ls case, she might want to search for header files and libraries first in the proto directory in her development workspace, then in the integration workspace used by the library group, then in her development workspace's parent (integration) workspace, and finally in the release workspace. She would set her PROTO variables to search as shown in Figure 14.

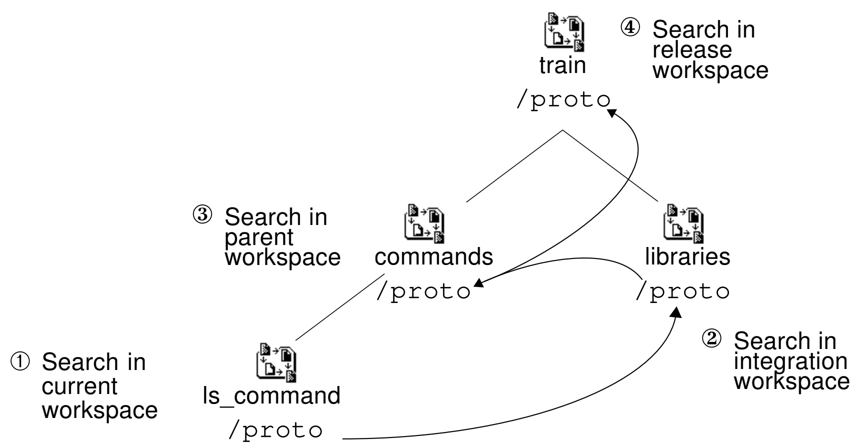


Figure 14 Example Search Path for Files in `proto` Directory

The search path shown in Figure 14 might provide the developer with the latest versions of the libraries — those produced by the libraries group — but might not provide her with the most stable versions. Those versions would presumably be in the release workspace. Or, if she and her fellow developers in the commands group decide to build with the same libraries, she would want those libraries to be in her parent workspace ("commands" in Figure 14).

If the necessary files cannot be found in any `proto` directory, the search can default to the native file system libraries, issue a warning message, or both. Whatever search sequence she decides on, `ws` provides an easy way to specify it.

The following fragment from the `ws` script first shows the search for a `protodefs` file, which the user can edit to set up her `PROTO` environment variables. If no `protodefs` file exists, the `PROTO` variables are set to default values. The script then constructs the `ENVCPPFLAGS` and `ENVLDLIBS` environment variables from the `PROTO` variables. The `ENVCPPFLAGS` and `ENVLDLIBS` variables are absorbed by makefiles to direct the compiling and linking of the program.

Figure 15 Part of the ws Script That Sets Environment Variables

```
...
CM_DATA=Codemgr_wsdata
wsosdir=$CODEMGR_WS/$CM_DATA/sunos
protofile=$wsosdir/protodefs

if [ ! -f $protofile ]; then
    if [ ! -w $CODEMGR_WS/$CM_DATA ]; then
        #
        # The workspace doesn't have a protodefs file
        # and this script cannot create one.
        # Tell user and use /tmp instead.
        #
        echo "Unable to create the proto defaults file
                ($protofile)."

        # Create a protodefs file in in /tmp
        wsosdir=/tmp
        protofile=$wsosdir/protodefs
    fi

    if [ ! -d $wsosdir ]; then
        mkdir $wsosdir
    fi

    cat << PROTOFILE_EoF > $protofile
#!/bin/sh
#
#   Set default proto areas for this workspace
#   NOTE: This file was initially automatically generated.
#
#   Feel free to edit this file.  If this file is removed
#   it will be rebuilt containing default values.
#
#   The variable CODEMGR_WS is available to this script.
#
#   PROTO1 is the first proto area searched
#   and is typically set to a proto area associated
#   with the workspace.  The ROOT environment variable
#   is set to the same as PROTO1.  If you will be doing
#   make installs, this proto area needs to be writable.
#
#   PROTO2 and PROTO3 are set to proto areas to search
#   before the search proceeds to the local machine
```

Figure 15 Part of the `ws` Script That Sets Environment Variables (Continued)

```
#   or the proto area specified by TERMPROTO.
#
#   TERMPROTO (if specified) is the last place searched.
#   If TERMPROTO is not specified the search will end
#   at the local machine.
#

PROTO1=\$CODEMGR_WS/proto

if [ -f "\$CODEMGR_WS/Codemgr_wsdata/parent" ]; then
    #
    # If this workspace has a Configuring parent,
    # then set PROTO2 to point to the parent's
    # proto space.
    #
    parent=\`workspace parent \$CODEMGR_WS\`
    if [ "\$parent" != "" ]; then
       PROTO2=\$parent/proto
    fi
fi
PROTOFILE_EoF

fi

. $protofile

#   The next line means you don't have to type make -e
#   each time you build

MAKEFLAGS=e; export MAKEFLAGS

#
#   Set up the CPPFLAGS and LDLIBS environment variables
#
MACH=`uname -p`
ROOT=/proto/root_${MACH}              # default

ENVCPPFLAGS1=
ENVCPPFLAGS2=
ENVCPPFLAGS3=
ENVCPPFLAGS4=
ENVLDLIBS1=
ENVLDLIBS2=
ENVLDLIBS3=
```

Figure 15 Part of the ws Script That Sets Environment Variables (Continued)

```
if [ "$PROTO1" != "" ]; then         # first proto area specifed
    ROOT=$PROTO1
    ENVCPPFLAGS1=-I$ROOT/usr/include
    export ENVCPPFLAGS1
    ENVLDLIBS1="-L$ROOT/usr/ccs/lib -L$ROOT/usr/lib"
    export ENVLDLIBS1

    if [ "$PROTO2" != "" ]; then
                                          # second proto area
specifed
        ENVCPPFLAGS2=-I$PROTO2/usr/include
        export ENVCPPFLAGS2
        ENVLDLIBS2="-L$PROTO2/usr/ccs/lib -L$PROTO2/usr/lib"
        export ENVLDLIBS2

        if [ "$PROTO3" != "" ]; then
                                          # third proto area specifed
            ENVCPPFLAGS3=-I$PROTO3/usr/include
            export ENVCPPFLAGS3
            ENVLDLIBS3="-L$PROTO3/usr/ccs/lib \
                        -L$PROTO3/usr/lib"
            export ENVLDLIBS3
        fi
    fi
fi

export ROOT

if [ "$TERMPROTO" != "" ]; then# fallback area specifed
    ENVCPPFLAGS4="-Y I,$TERMPROTO/usr/include"
    export ENVCPPFLAGS4
    ENVLDLIBS3="$ENVLDLIBS3 -Y
P,$TERMPROTO/usr/ccs/lib:$TERMPROTO/usr/lib"
    export ENVLDLIBS3
fi
...
```

Shielding Developers From Unstable Files

If a developer sets up his build search path to refer to a copy of a library or header file located in another workspace's `proto` directory, he is exposed to the changes other developers make to the file. Any change in the file will be immediately visible the next time he performs a build. However, there are times when a developer does not want to see such changes — when he is diagnosing a bug, for example. A developer can do one of two things to shield himself from changes:

- He can use the file in the `proto` directory of the release workspace during a build. Files there are not updated as often as those in other workspaces, and are typically well tested.
- He can bring over a complete copy of the SunOS source base and build his own `proto` directory to use during builds. This option gives him full control of the `proto` directory used during a build.

Other Custom Utilities

The SunOS group wrote several other scripts to simplify manipulation of source files controlled by Configuring. The most interesting are `sccsrm` and `sccsmv`.

`sccsrm`

Files in workspaces controlled by Configuring can be renamed but not deleted. The reason for this seeming paradox is obvious if you consider that a file that has been deleted from one workspace will reappear when it is put back from a child workspace where it has not been deleted. By renaming a file to a hidden file name — `.del-filename`, for example — the file appears to be deleted, and the rename propagates to other workspaces, where the file will also appear to be deleted.

The `sccsrm` script initially renames a file to a `.del*` name in the current directory; another script is run periodically to move `.del*` files to an area in a directory hierarchy parallel to (and with the same structure as) the source hierarchy. See the TW Topic "Deleting Files From the Configuring Tool Workspaces on page 89" for details on this strategy.

Because users are not required to remember the path name of the directory that stores deleted files when they rename a file they want to delete, the `sccsrm` utility helps avoid mistakes. Figure 16 shows the entire script for `sccsrm.`

Figure 16 Script for the `sccsrm` Utility

```
#!/usr/bin/sh
USAGE="usage: sccsrm [-f] <filename> ..."
#
#ident "@(#)sccsrm            1.4        93/03/30 SMI"
#
# This script is to be used to remove files from any CodeManager
# workspace. It will do this by moving the specified file,
# and its corresponding s-dot file, to a .del-<file>-`date`
# format.
#
# The only way to remove files under the CodeManager is
# through the rename mechanism - it is not enough to
# simply `rm' the file.
#


message() {
    if [ ${F_FLAG} -eq 0 ]; then
    echo "$*"
    fi
}

#
# LC_ALL=C is set so that the this script will work no matter
# which localization you have installed on your machine. Some
# localizations can cause the output of `date' and other commands
# to vary.
#
LC_ALL="C"; export LC_ALL

date=`/usr/bin/date +%h-%d-%y`
F_FLAG=0


#
# Parse options...
#
set -- `getopt f $*`
if [ $? != 0 ]; then
 echo $USAGE
 exit 2
fi
```

```
 (Continued)


for i in $*
do
 case $i in
 -f) F_FLAG=1; shift;;
 --) shift; break;;
 esac
done

if [ $# -eq 0 ]; then
 message $USAGE
 exit 1
fi

#
# Process s-dot files.
#
for file in $*
do
 new_file="${file}-${date}"
 #
 # If there is a deleted file of the same name, append the pid
 # to the name.
 if [ -f SCCS/s..del-${new_file} -o -d .del-${new_file} ]; then
 new_file="${new_file}.$$"
 fi
 if [ -f SCCS/s.$file ]; then
 if [ -f SCCS/p.${file} ]; then
 if [ ${F_FLAG} -eq 0 ]; then
    echo "warning: ${file} is checked out for editing, \
    all edits will be lost - continue (y/n)"
    read ans
    while [ `expr $ans : "^[YyNn]"` -eq 0 ]
    do
    echo "warning: ${file} is checked out for editing, \
    all edits will be lost - continue (y/n)"
    read ans
    done
    else
    ans="y"
    fi
    if [ `expr $ans : "^[Yy]"` -eq 1 ]; then
 rm -f SCCS/p.${file}
```

```
(Continued)
   rm -f ${file}
   else
   continue
   fi
fi
if [ -f ${file} ]; then
mv ${file} .del-${new_file}
fi
mv SCCS/s.${file} SCCS/s..del-${new_file}
elif [ -d ${file} -a ${file} != "SCCS" ]; then
mv ${file} .del-${new_file}
else
message "${file}: not an SCCS file"
fi
done
```

sccsmv

Because moving SCCS-controlled files is so similar to deleting them in the Configuring environment, the utility `sccsmv` was written as a companion to `sccsrm`.

`sccsmv` renames files the same way `sccsrm` does, but with `sccsmv` a user can specify the new name of the file. In practical terms, this utility merely saves the extra step of moving both the g-file (sometimes called the *clear file*) and the SCCS history file when performing a rename. However, because Configuring acts only on SCCS history files, errors are eliminated by moving them automatically whenever a g-file name is given to `sccsmv`.

Using File List Programs

When a developer brings over a file into her workspace, she explicitly specifies the file she wants to move. When she specifies a directory, a default File List Program (FLP) sees to it that all the files in the directory are brought over. The default FLP accomplishes this feat by listing the contents of the directory and writing them to `stdout`, from which the bringover and putback commands read them. Users can replace the default FLP with a custom FLP that writes any file names they want to `stdout`, enabling them to bring over (and put back) files without naming them explicitly.

The SunOS group has found custom FLPs to be an easy way to ensure that all the files needed to build an executable are available in a development workspace.

Default File List Program

Once again, consider the developer responsible for the `ls` command. With the default FLP (named `def.dir.flp`) in force, the developer would explicitly bring over the `ls` and `proto` directories, resulting in a development workspace directory structure like the one shown in Figure 17. (The `Codemgr_wsdata` directory is present in every workspace.)

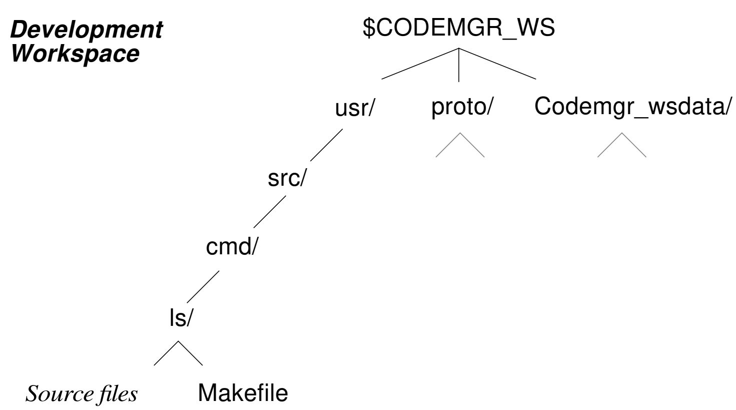


Figure 17 Development Workspace for the `ls` Command (Default FLP)

One shortcoming of the workspace shown in Figure 17 is obvious when you consider the makefile for `ls` shown on page 22. That makefile includes two other makefiles, `Makefile.cmd` and `Makefile.install` from the `cmd` directory. `Makefile.cmd`, in turn, includes `Makefile.master`. None of these files were brought over by the default FLP, so an attempt to build the `ls` executable will fail in the workspace shown in Figure 17.

A less obvious problem is that other files in the source base may need to be built before `ls`, as determined by the `ls` makefile. If such files exist, they must be brought over explicitly when using the default FLP.

Custom File List Program

The SunOS group has customized the default FLP so that it executes scripts that identify the files that are required for a specific build. The required files are listed by scripts named `inc.flp` and `req.flp`, which are stored in the same directory as the source files required for the build.

Using the custom FLP, the developer specifies only the `/usr/src/cmd/ls` directory in his bringover command. The custom FLP looks in the directory and lists its contents recursively just as the default FLP does. In addition, it looks for a script named `inc.flp` or `req.flp`. If one or both of these scripts is among the contents of the directory, it is executed. The executed scripts list additional files required to build the executable, and the names are read by the bringover command. In the case of `ls`, the scripts would name the `proto` directory and the makefiles required for the build. Figure 18 shows the resulting workspace.

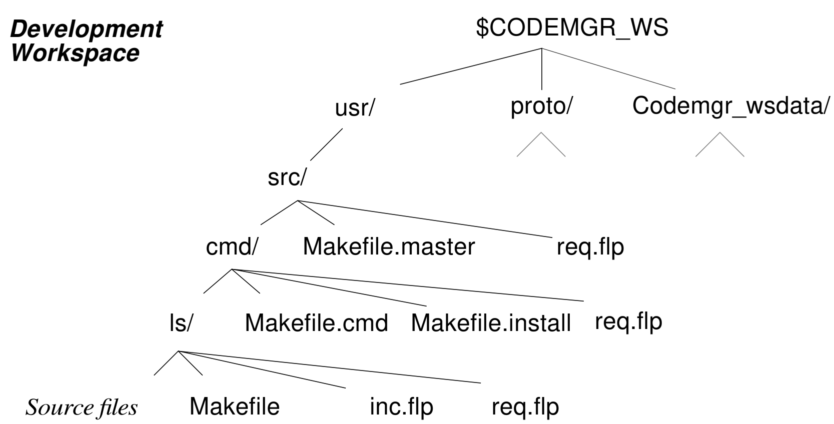


Figure 18 Development Workspace for the `ls` Command (Custom FLP)

The `inc.flp` script lists files that do not reside under the current directory but are required in order to build the current directory. Header (include) files are the best examples of such files. The `req.flp` script lists other files that are not included but are nevertheless required to build `ls` — most importantly, makefiles. The `inc.flp` script is associated with the source directory structure, while the `req.flp` script is associated with the makefile structure.

When a developer brings over `/usr/src/cmd/ls`, the custom FLP starts in the current directory and searches for the `req.flp` and `inc.flp` scripts. It searches for `req.flp` in ancestor directories and for `inc.flp` in descendant directories, executing the scripts when it finds them (Figure 19). The `req.flp` script lists files that are required for any build in a subdirectory. For example, the `req.flp` script in the directory `/usr/src/cmd` writes out `Makefile.cmd` and `Makefile.install`, which are used to build all commands.

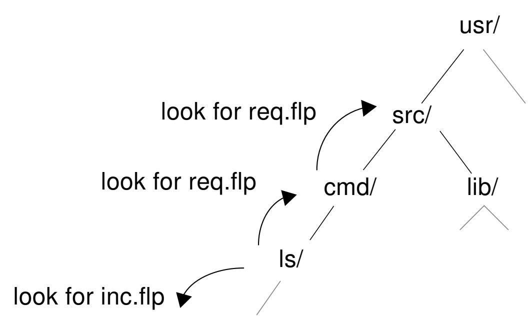


Figure 19 The Custom FLP Searches Directories for `req.flp` and `inc.flp`

Similarly, when the custom FLP executes the `req.flp` script in the `/usr/src` directory, it writes out the name `Makefile.master`, the makefile that is used to build every executable in the source base.

If the developer brings over `/usr/src/lib/lib.c`, the custom FLP encounters a different `req.flp`. The `req.flp` in `/usr/src/lib` writes out `Makefile.lib`, which is required to build every library. `Makefile.master` is brought over when the custom FLP executes the `req.flp` in the ancestor directory `/usr/src`.

The `inc.flp` script performs a similar function for files elsewhere in the file system. When a build depends on files in the brought-over directory, its subdirectories, or in distant directories, `inc.flp` provides the file names to the bringover command.

The custom FLP and the scripts it executes help automate complicated Configuring transactions. However, the scripts themselves are static descriptions of the dependencies of build targets and must be edited when the directory structure or build target dependencies change.

Registering Public Workspaces With NIS

The SunOS group takes advantage the automounter and the Network Information Service (NIS), both standard parts of the Solaris operating environment, to reduce the amount of information developers must remember daily.

The automounter (`automount`(8)) is a daemon that automatically mounts an NFS directory as needed. It monitors attempts to access directories that are associated with an NIS *automount map*, along with any directories or files that reside under them. When a user attempts to access a file or directory, the daemon mounts the required NFS file system.

By registering the SunOS public workspaces (integration and release-level) in the automount map, the SunOS group is able to assign them abstract, net-wide names. The abstract names relieve users from having to remember the true path name of workspaces that must be accessed often.

For example, a workspace physically located at
`server:/export/here/there/everywhere/sunos1`
is normally accessed by the automounter with the command
`cd /net/server/export/here/there/everywhere/sunos1`
In contrast, after a map name for the workspace has been entered in the automounter map, SunOS developers can access the same workspace with the command
`cd /workspace/sunos1`

If the physical location of a workspace must be changed because of disk space limitations or other reasons, users do not need to learn the new path name. The new physical location is entered into the automounter map, but the abstract name does not change.

TW Topic:

Workspace Hierarchy Strategies for Software Development and Release

The way you organize the Configuring tool 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 Configuring.

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 Configuring workspace hierarchies are *flexible*. Configuring enables you easily reconfigure hierarchies as your project, organization, and experience with Configuring evolve.

Scenario

A company that produces software to control assembly line processes recently purchased Configuring 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 Configuring 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 developed by company management 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
- Integration
- Release

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.

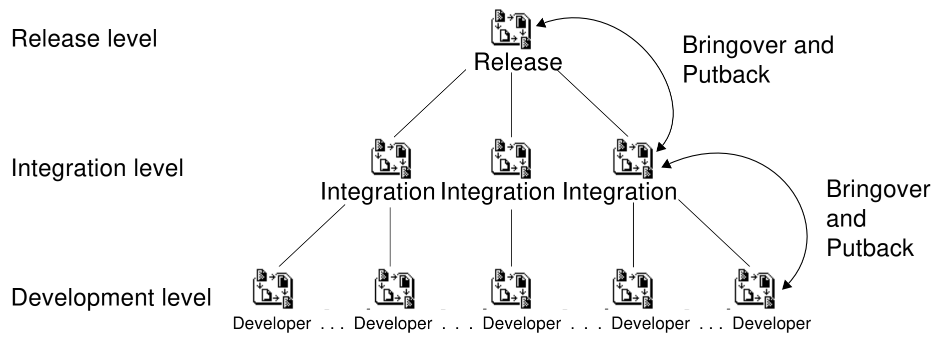


Figure 1 Workspace Hierarchy Levels

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
- 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 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 ensure 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, but 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 illustrates this process.

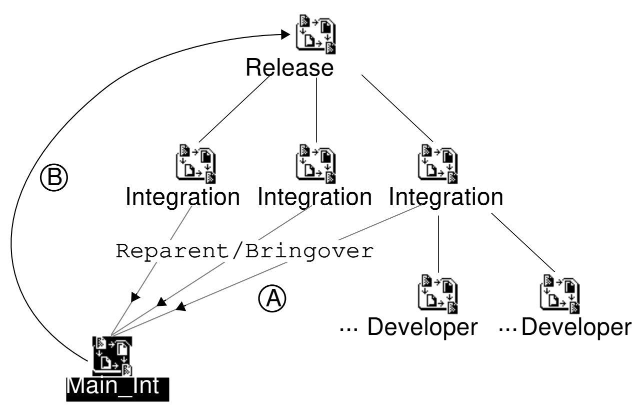


Figure 2 Using a Main Integration Workspace (Strategy 1)

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 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.

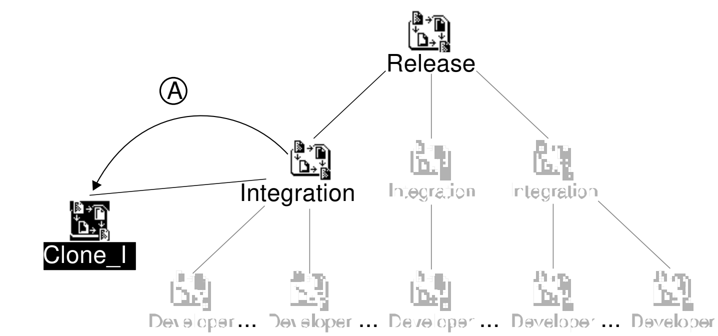
Strategy 2

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.

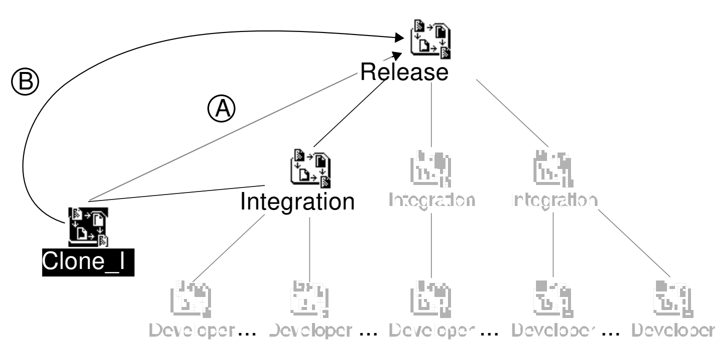
Figure 3 describes Strategy 2.

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.

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.



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).



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.

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

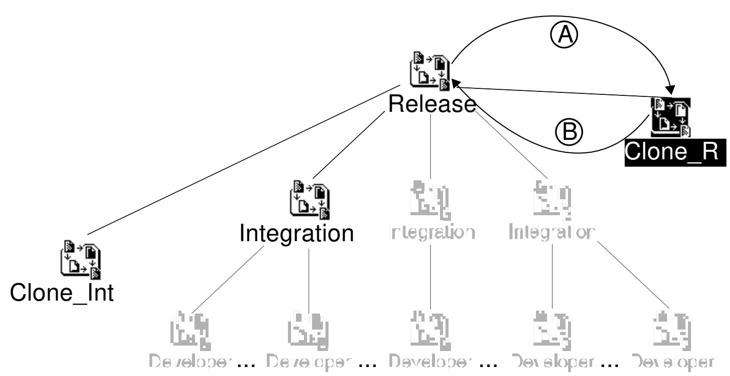


Figure 3 Using Integration Workspace Clones (Strategy 2)

Peer-to-Peer Updates

After the company began using Configuring, some developers discovered that it was sometimes helpful to reparent and use the bringover transaction to *directly* update each other's workspaces. This is shown in Figure 4.

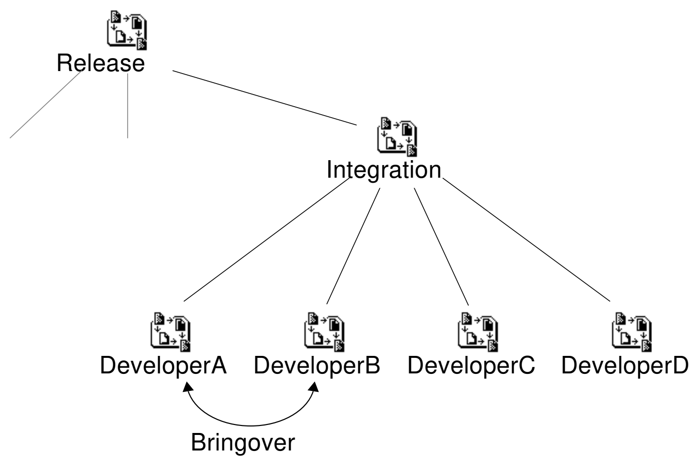


Figure 4 Peer-to-Peer Updates Using the Bringover Transaction

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, they put back changes made by the other developer along with their own changes. This requires developers to merge code they haven'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 that a developer not 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
- Ensure that all features developed for Release 1 are also included in Release 2
- 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 was 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 was 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 `twconfig` workspace graph pane and their relationships can be easily determined using the Configuring `workspace children` and `parent` commands.

Figure 5 shows the relationship between Release 1 and Release 2 in the Cascade Model.

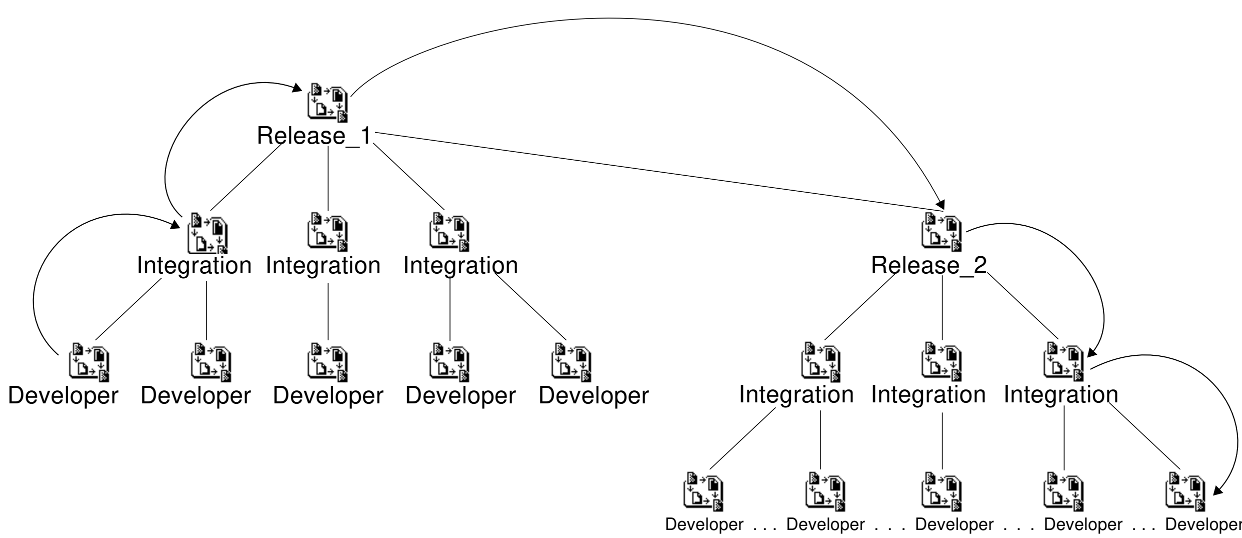


Figure 5 Updating Release 2 from Release 1 (Cascade Model)

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 40 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.

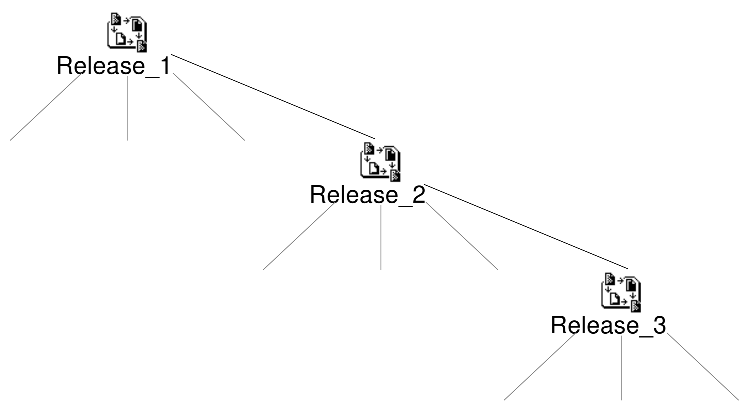


Figure 6 A Three-Release Cascade

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:

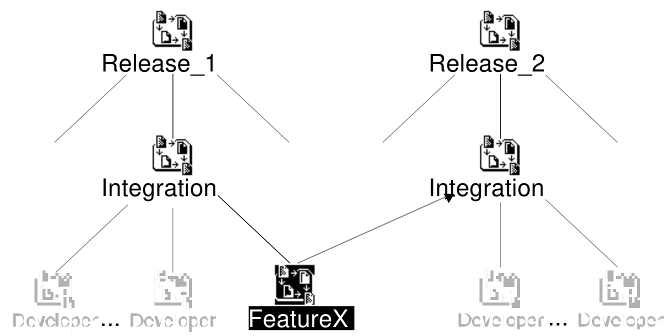
- A feature intended for Release 1 cannot be included in that release due to time-to-market considerations, but *will* be included in Release 2. The feature's development workspace can be reparented directly to the Release 2 hierarchy. The company calls this process "late binding of a feature to a release."
- 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.

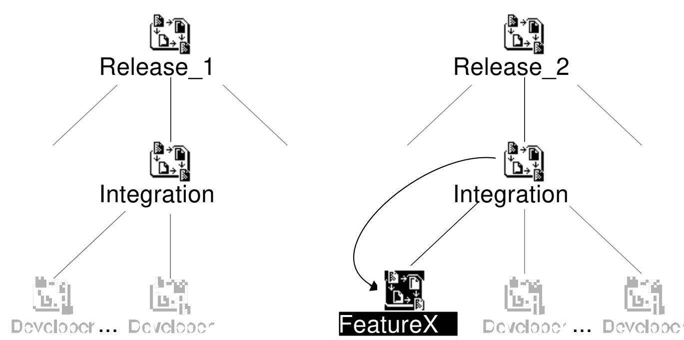
The Configuring reparenting command is used to change the FeatureX workspace’s parent from the Release 1 hierarchy to Release 2.

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.



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.



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.

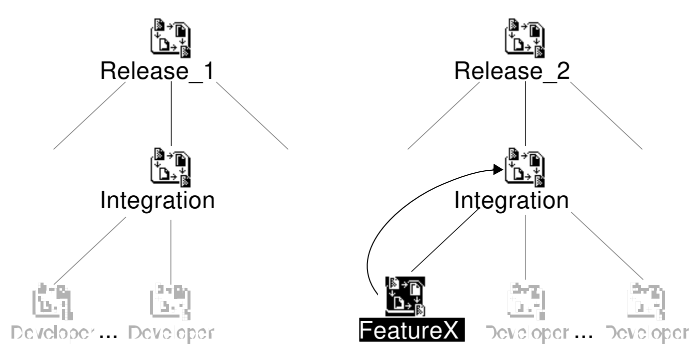


Figure 7 Late Binding of a Feature to a Release

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 46.

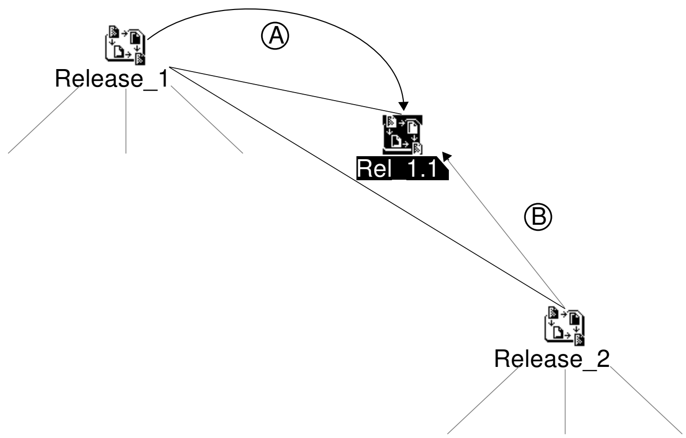


Figure 8 Inserting a Minor Release Into the Release Cascade

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 Configuring 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).**

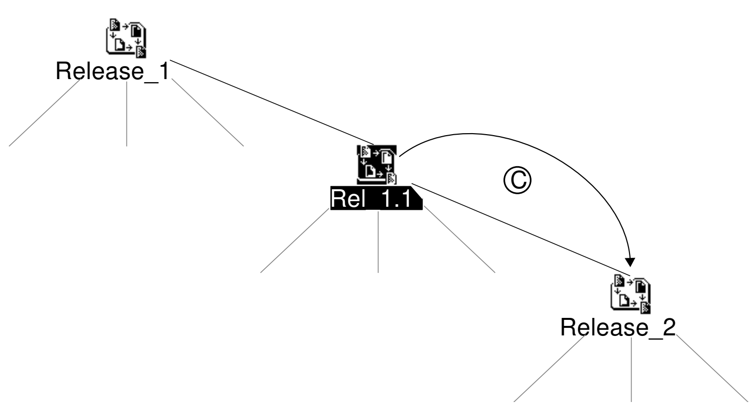


Figure 9 Updating Release 2 From Release 1.1

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 `twconfig` workspace graph pane and their relationships can be determined using the Configuring `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 who encounter the problem.

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

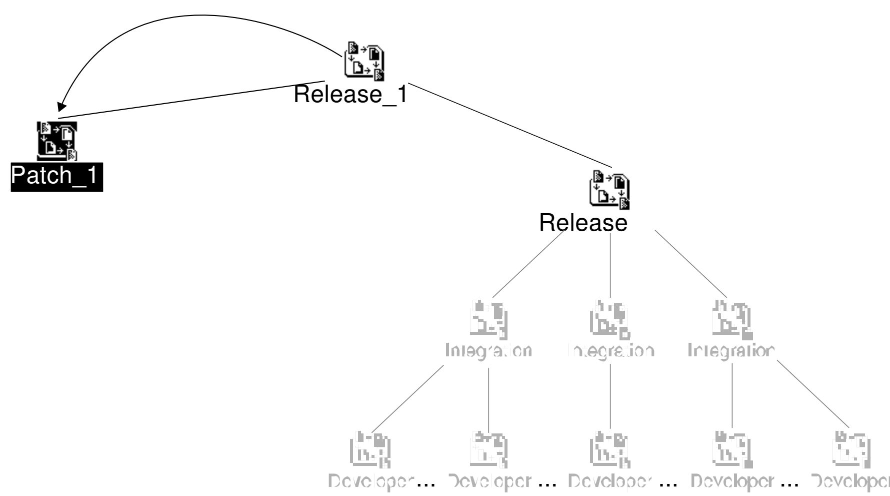


Figure 10 Creating a Patch Release

1. This model is described in "Reparenting Model" on page 47.

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.

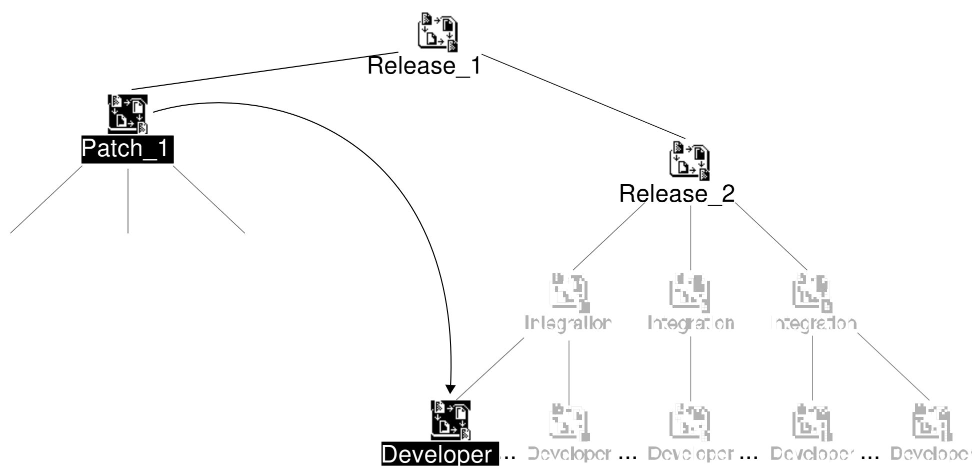


Figure 11 Reparenting the Patch Workspace

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).

Some Notes About Patch Releases

- 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 46. 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:

Remote Software Development Using the Configuring Tool

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. Configuring tool 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 Configuring 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 Configuring 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. They 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 were (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 Configuring 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.

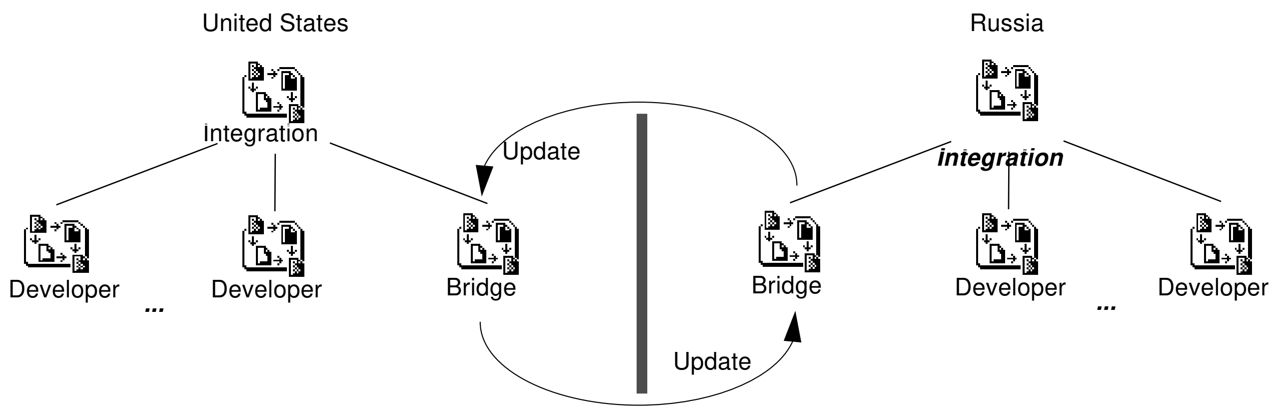


Figure 12 Workspace Hierarchies With Bridge Workspaces

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 seemed seductively more efficient to have both sites ship their updates to the other site at the same time. Upon further analysis they determined that this method does not ensure that changes are propagated correctly.

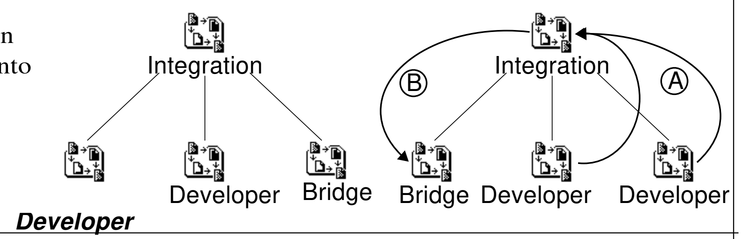
To maintain consistency, Configuring enforces a single-writer model — only one transaction that alters data can proceed in a workspace at a time. Concurrency is achieved by the developers working concurrently in their own workspaces and then updating a central integration workspace sequentially, one at a time. From Configuring'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 Configuring single-writer paradigm; under normal (non-remote) circumstances Configuring does not permit this to happen.

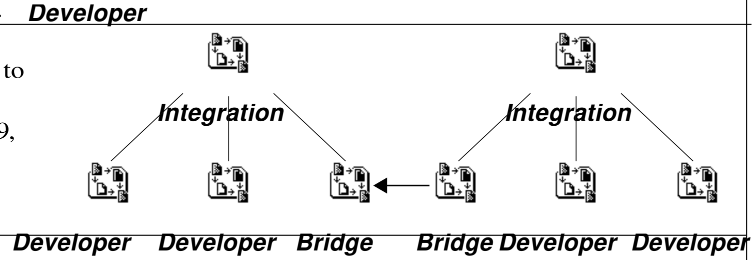
	U.S.	Russia

Russian developers put back changes to the Integration workspace (A). These changes are then brought over into the Bridge workspace (B).

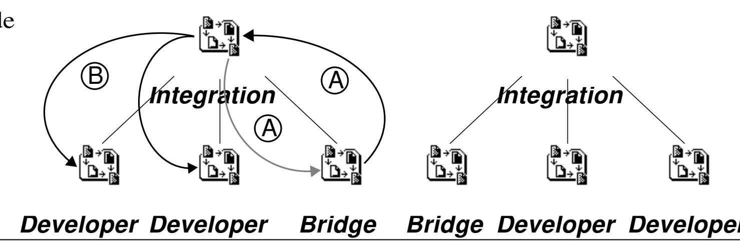
Note: Throughout this process, development work continues at both sites. These figures highlight only the specific transactions involved in remote transfers.



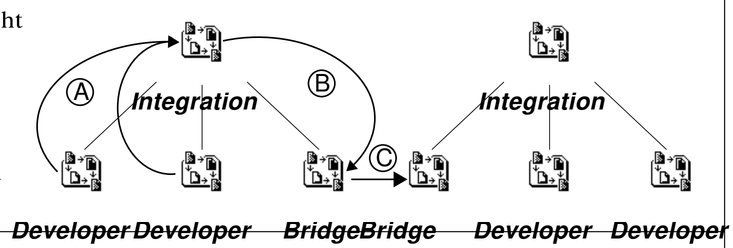
The Bridge workspace is copied to tape and shipped to the U.S. By one of the methods discussed in “Transferring File Changes Between Sites” on page 59, the data from the Russian Bridge workspace is incorporated into the U.S. Bridge workspace.



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.



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 59, the data from the U.S. Bridge workspace is incorporated into the Russian Bridge workspace.



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.

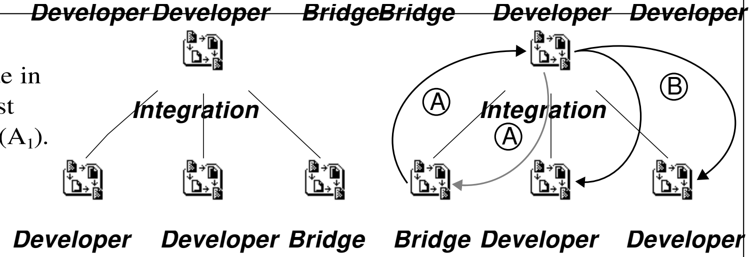


Figure 13 A Remote Update Cycle

Consider the typical update cycle pictured in Figure 13. At the conclusion of Step and Step the Bridge workspaces at both sites *must be identical*. The putback transaction in Step (A) is exactly the same as if it were executed directly from the Russian Bridge workspace and the putback transaction in Step (A) is exactly the same as if it were executed directly from the U.S. Bridge workspace — from a Configuring 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 had been renamed A workspace "checksum" created using the Freezepointing 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, Configuring is used normally.

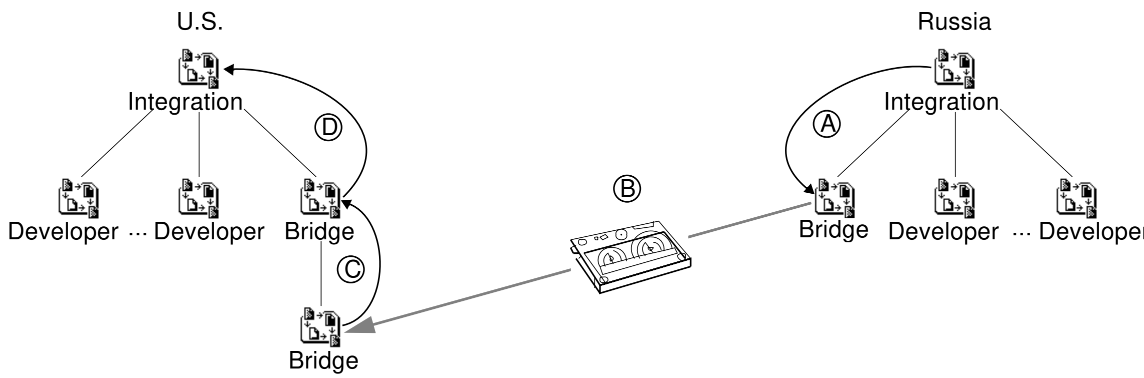


Figure 14 Transferring the Entire Bridge Workspace (Option 1)

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 Configuring 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 Configuring 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. Configuring 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, Configuring cannot protect you from losing any data that may have been inadvertently copied to the U.S. bridge workspace. In addition, the Configuring history facility will not record the transactions for later inspection.

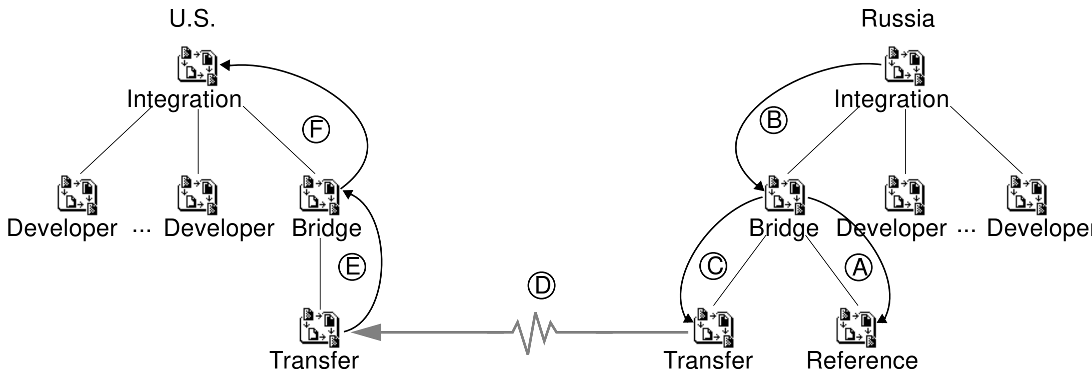


Figure 15 Transferring Updated, Created or Renamed Files (Option 2)

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.

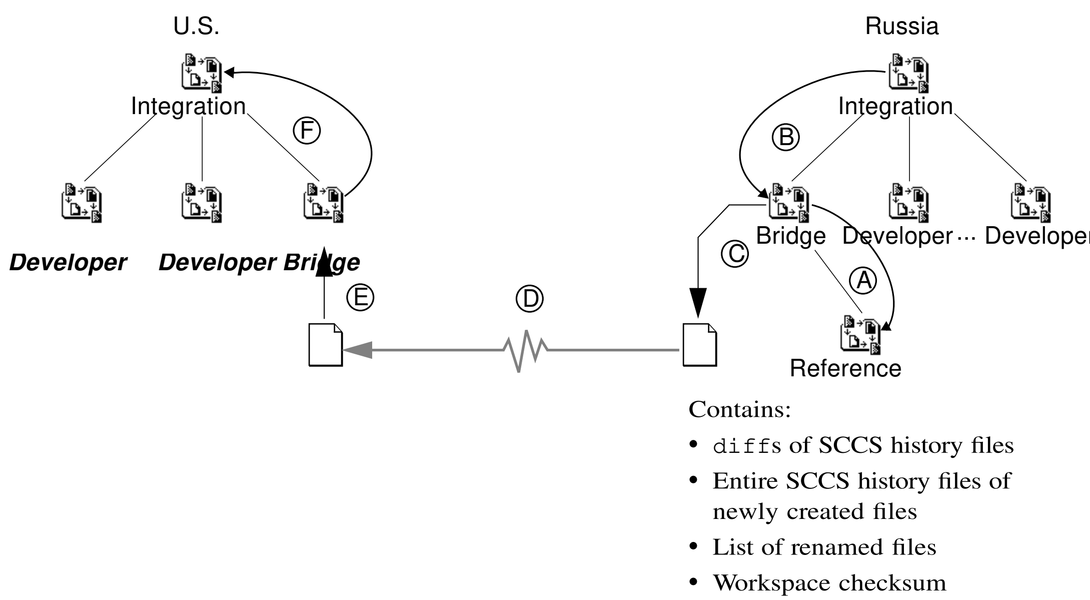


Figure 16 Transferring Only `diffs` of Changed SCCS History Files (Option 3)

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 TeamWare Freezepointing utility is used to create a "checksum" from the Russian Bridge workspace.**
   This checksum is 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 is 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.**
   Freezepointing is used to create a freezepoint file from the U.S. Bridge workspace. The U.S. freezepoint file is compared to the Russian freezepoint 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 Configuring 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 Configuring's Access Control facility to ensure that the contents of the Bridge workspaces are not inadvertently changed between update cycles.
- You can used Configuring's Notification facility to automatically inform another site by 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.
- Configuring 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 Configuring's ability to determine whether files have changed.

TW Topic:

Strategies for Deploying Configuring in Software Development Organizations

After you decide how you want to use Sun WorkShop TeamWare to help manage your development process, you have to deploy that solution throughout your organization. It is a good idea to think about how you will deploy your solution even as you devise and implement it. This topic describes how a software development organization efficiently and effectively deployed its solution without disrupting release schedules.

Scenario

A large software development company recently decided to use the TeamWare Configuring tool to help manage its software development process. The company has two products: voice recognition software and card reading software. The voice recognition project employs 25 developers, and the card reading project employs 20 developers. A group of 10 other developers produces common code used by both product groups.

Management analyzed the development organization and determined ways to use Configuring to enhance the code management process. They then devised a transition strategy to deploy Configuring to the development groups.

Major goals for the transition strategy:

- Prototype the company's planned use of Configuring. Management wanted to try out the workspace hierarchy strategy they devised for a small group before deploying it to the entire organization.
- Gracefully incorporate Configuring into active development projects. Management wanted to impact current development schedules as little as possible.

Implementation

The following sections describe the company's:

- Pre-Configuring source file organization
- Configuring solution
- Transition plan that incrementally adopts Configuring into the development organization

Before Configuring

The company's pre-Configuring source hierarchy was structured so that each development group worked in its own subdirectory under a common source directory. SCCS was used for version control. Figure 17 shows the layout of the source hierarchy.

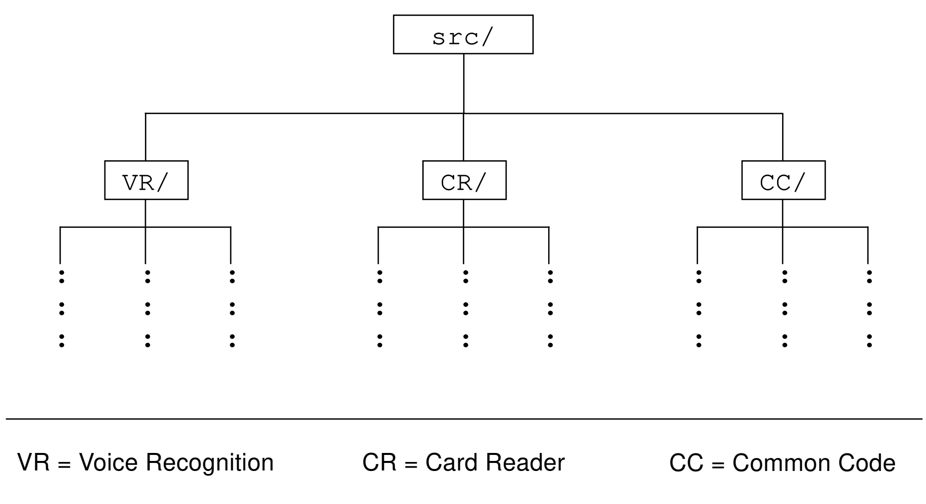


Figure 17 Source File Hierarchy

Developers worked directly in the main project directories containing the files they were concerned with. Often the developers checked files out from SCCS and then copied them to their own work areas, returning them to the project directories when they checked them back in.

This method of development made it difficult to coordinate the work of developers who shared common and interdependent files. Using SCCS in this manner provided only serial access to common files, one developer at a time — this process created a productivity bottleneck. To get around the bottleneck, developers sometimes made copies of already checked-out files and copied them to their private work areas where they made changes. Changes were extremely difficult to track when the code was merged together.

The Configuring Solution

After analyzing its products, source base, and development organization, the company decided that a three-level workspace hierarchy was the best solution. The three-level hierarchy provides:

- A top-level release workspace. This workspace contains all of the product source files.
- An integration workspace for each functional group. Senior developers from each group work in these workspaces to integrate changes made by developers. The workspaces contain only the relevant sources for each group.
- A workspace for each developer or task. Development workspaces contain only the files that developers need to do their work.

See the TW Topic “Workspace Hierarchy Strategies for Software Development and Release on page 37” for details about workspace hierarchy strategies.

Figure 18 shows the final hierarchy and the types of files contained at each level.

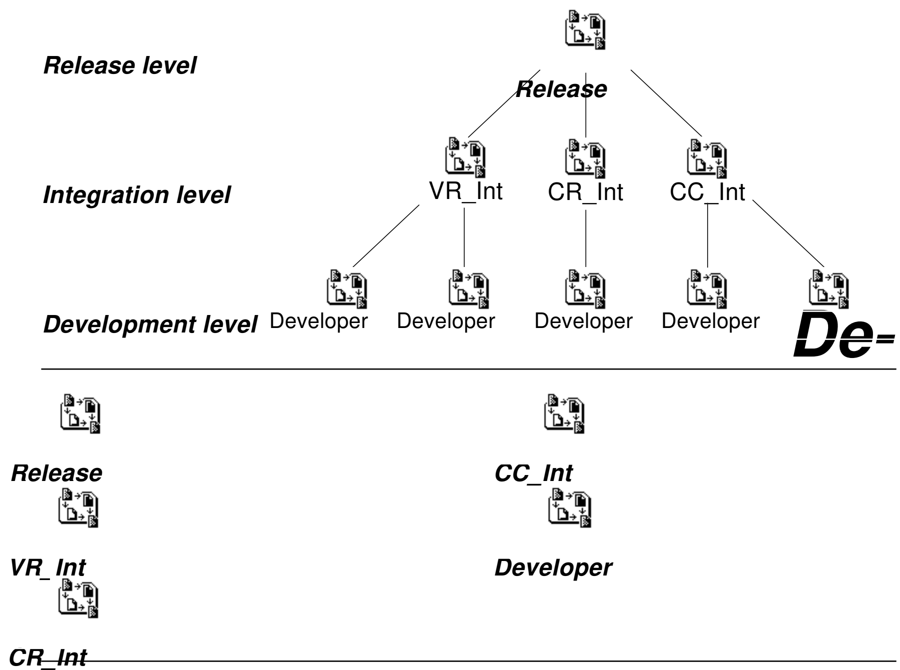


Figure 18 Configuring Workspace Hierarchy

As you can see by comparing the file system hierarchy (Figure 17) with the workspace hierarchy (Figure 18), the workspace hierarchy is analogous to the file system hierarchy. In addition, this hierarchy strongly reflects the company's engineering organization.

A major advantage of this workspace hierarchy is that it lends itself very well to the company's incremental transition plan. That plan is described in the next section.

The Transition Plan

The company's goal was to incrementally adopt Configuring into the development organization. The strategy was to prototype the solution in a smaller group in order to cause minimal disruption to current product development schedules.

The company decided on the following transition plan:

1. Initially, deploy the Configuring solution only in the CC group. This group was chosen for prototyping because:
   - The CC group was the smallest of the three groups.
   - The code produced by this group was used by both of the other projects. Therefore, the larger projects would be exposed to some Configuring concepts during the transition period.
2. Allow the VR and CR groups to work uninterrupted in their usual manner during the initial deployment.
3. After testing the solution, smoothly deploy the Configuring solution to the entire development organization as schedules permitted.

The transition plan is shown in Figure 19 on page 72.

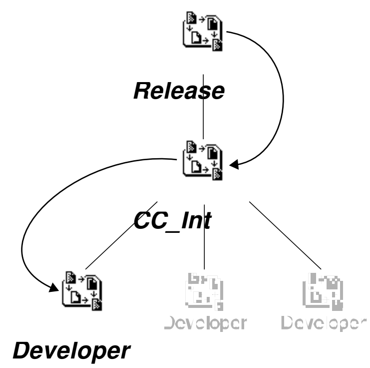
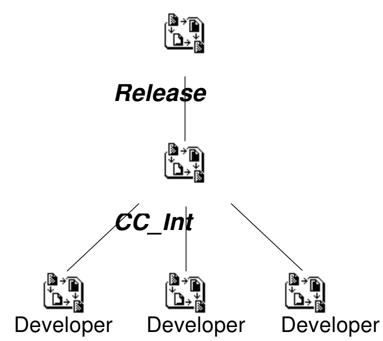
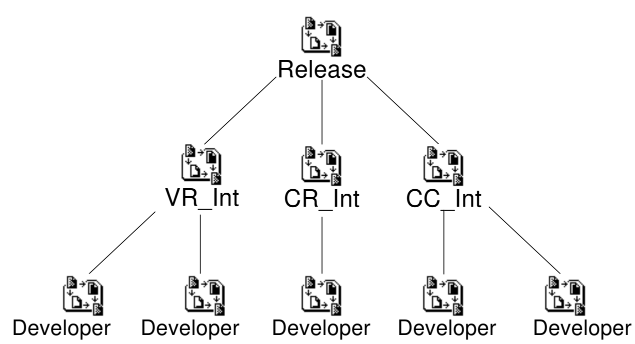
A workspace hierarchy is created: The existing project source hierarchy becomes the release workspace ①. The source developed by the CC group is brought over to create an integration workspace ②. Developer workspaces are created as required ③.	
Development continues: The VR and CR groups work directly in the Release workspace using their traditional methods. The CC group begins development in workspaces, using Configuring commands to transfer changes between the developer workspaces and the integration workspace, and between the integration workspace and the release workspace. Product releases are created from the Release workspace.	
The VR and CR groups convert to Configuring workspaces when: The company decides that the Configuring process is working well in the CC group. VR and CR product development is at a convenient juncture (typically, just after a release).	

Figure 19 The Transition Plan

Notes

- The only difference between a Configuring workspace and a standard directory is that the workspace contains a `Codemgr_wsdata` directory. Therefore, the CR and VR groups' use of the Release workspace as their source directory was totally transparent to them; for their purposes the Release workspace was just like their traditional project source directory.
- All developers in the CC group had their own workspaces. Periodically, changes made in those workspaces were integrated in the CC integration workspace and those changes were then integrated in the release workspace. Conversely, changes made by the CR and VR developers in the release workspace were brought down into the integration workspace, and then to the individual developers' workspaces. See the TW Topic Workspace Hierarchy Strategies for Software Development and Release on page 37" for more information about workspace hierarchies.

TW Topic:

Getting the Most From the Configuring Tool Notifications

the notification feature of the Configuring tool automatically sends email when a workspace-related command (such as a putback) is executed. Users have expanded on this feature to perform a variety of tasks. This topic describes some of these tasks and how they can be implemented.

A complete script that takes advantage of the notification feature is also presented, beginning on page 82. The script can be adapted to a variety of uses.

Typical Use of Notification

As an example of the way notifications are typically used, consider the user `chip`, who wishes to be notified whenever the file `foo.cc` is put back from a development workspace into an integration workspace. To register the event with the Configuring notification feature, `chip` can use either the graphical user interface or a text editor to place the following entry in the `Codemgr_wsdata/notification` file in the integration workspace. This entry causes Configuring to send `chip` a message when anyone performs a putback of `foo.cc` that changes `foo.cc` in the parent:

```
chip@monk putback-to
BEGIN
foo.cc
END
```

This basic use of the notification feature is valuable in itself, but many sites have found more creative uses for notification. The key to exploiting the notification feature is recognizing that it calls `mail`, a flexible program with capabilities that are often overlooked. For example, `mail` can send messages not only to users but to other programs through a *pipe*, the operating system feature that allows the output from one program to become the direct input to another.

The Ins and Outs of `mail`

The `mail` program can send messages to aliases as well as to users. If you place the name of an alias in the `notification` file of a workspace, a mail message is sent to all users listed in the alias. By substituting a pipe symbol (|) and a program name for a user name, the contents of the message are passed to the program instead of mailed to a user.

As an example, assume that user `chip` wants to compile a list of the names of users who bring over his file (`foo.c`) into their workspaces. He could place the following lines in the `notification` file, either with the Configuring GUI or directly with a text editor:

```
foo_alias@monk bringover-from
BEGIN
foo.cc
END
```

These entries cause a message to be sent to `foo_alias` whenever `foo.cc` is brought over into another workspace. If `foo_alias` is defined as follows in the `/etc/aliases` file:

```
foo_alias:chip@monk, | /home/chip/bin/foo_log.sh
```

then user `chip` receives the notification message, and it is also piped to `chip`'s private program `foo_log.sh`. In this case, `foo_log.sh` is a script that extracts the `Date:` and `User:` lines from the mail message and appends them to a log file. Because these lines contain the date the message was sent and the name of the user who performed the bringover, the log file accumulates a concise history of who brought over this important file.

Mailing a Policy Statement

Real-world examples of custom notification scripts are common among Sun WorkShop TeamWare sites. One example, taken from the annals of SunOS development, is described briefly on page 12. In that case, a notification is placed in top-level workspaces that causes a policy statement to be sent to users who bring over files for the first time. The policy statement details the responsibilities of users who edit files from top-level workspaces and alerts new users that are not able to put back files to top-level workspaces without special permission.

A mail alias provides a straightforward way to automate sending the policy statement. First, the person responsible for the top-level workspace places the required line in the `Codemgr_wsdata/notification` file of the workspace. For example, the following line, entered without `BEGIN` and `END` keywords, causes a message to be sent to `newuser_alias` whenever someone brings a file over from the workspace:

```
newuser_alias@mach bringover-from
```

A line is then placed in the `/etc/aliases` file on `mach` to define the alias. The following line causes a message that is mailed to the alias to be piped to the script `newuser.sh`. The script is designed to parse the mail messages and take the desired action.

```
newuser_alias: | newuser.sh
```

When a new user brings over a file from the top-level workspace into a development workspace, the following occurs:

1. Configuring checks the `notification` file to determine whether or not to send a mail message in response to the bringover event. The `notification` file instructs Configuring to send a message to `newuser_alias`.

2. Configuring launches the `mail` program to send a message to `newuser_alias`, one member of which is `|newuser.sh`. The mail message is routed through a pipe to `newuser.sh`.

3. The `newuser.sh` script parses the mail message and extracts the name of the user who performed the bringover.

4. The script reads a list, searching for the user name. If the name is in the list, then the user is not a new user, and the script quits.

5. If the user name is not in the list, then the user is a new user, and the script adds the user name to the list.

6. The script launches `mail` to send a policy statement to the new user and quits. The `mail` program sends the policy statement to the new user in a mail message.

Other Uses of Notifications

The last section described one use of a script that parses mail messages; that is, to send a message to a user whose name does not appear in a list of users. Some other possible uses of message-parsing scripts are to:

- Mail messages only to users who perform certain activities such as reparenting workspaces
- Log notification messages categorized by user, date, activity, or workspace (parent or child)
- Maintain statistics on how often certain events such as putbacks occur in specific workspaces
- Change the subject line of the notification message, add commentary, and forward the message to interested parties or save it to a log file. For example, the script could parse the `Parent workspace:` lines to determine the project or release level to which the event applied. On that basis, the script could change the `Subject:` line to provide more detail before forwarding the message.
- Search the message for a bug identifier, and log or forward the message based on the bug identification number
- Schedule (following the putback of a source file) in the parent workspace an overnight build and automatic test of executables that depend on the source file.

The next section provides details of how to accomplish these goals.

Parsing Mail Messages

This section describes ways to write scripts that parse mail messages from Configuring, including an example script you can modify for almost any use. The examples are not the only ways to accomplish their stated goals, and may not be the most elegant ways. They are presented here because they have been proven to work.

Typical Mail Message

The mail message shown in Figure 20 is typical of those sent by the Configuring notifier. This message was sent in response to the putback of three files from the child workspace `develop_1` to the parent workspace `integrate_1` by user `jdoe`. This message is used as the example for the rest of this topic.

```
From jdoe Thu Jan 26 13:01 PDT 1995
Date: Thu, 26 Jan 1995 13:01:50 +0800
From: jdoe (Jan Doe)
To: script_alias@conundrum
Subject: Code Manager notification

Event: putback-to
Parent workspace: integrate_1
    (holymoly:/export/home/holymoly/jdoe/src/ws/integrate_1)
Child workspace: develop_1
    (holymoly:/export/home/holymoly/jdoe/src/ws/develop_1)
User: jdoe

Comment:
    In my attempt to fix BUG-12, I fixed BUG-121 instead!

Files:
create: file_1
create: file_2
create: file_3
```

Figure 20 Typical Notification Mail Message

Copying the Message

Before attempting to parse a mail message, most scripts make a copy of the message. For example, when a script receives its input from `stdin`, the following line in the script stores a temporary copy of the incoming message in `/tmp`:

```
cat > /tmp/incoming_message
```

The message can then be parsed or appended to a more permanent log file.

Parsing Techniques

Most scripts that parse mail messages make heavy use of filters and utilities such as `grep`(1) (or `egrep`(1)), `sed`(1), `awk`(1), and `tr`(1). Consult the man pages for complete information.

Checking Message Validity

Messages from the Configuring notifier contain the string "`Code Manager notification`" in their subject lines. A script may check this line to make sure the message was indeed sent to the script by the notifier. The following lines perform this check.:

```
SUBJECT=`egrep -i '^Subject:' /tmp/incoming_message \
             | head -1 | awk -F: '{print $2}'`
if [ "$SUBJECT" != " Code Manager notification" ]; then
    Process misdirected message …
    exit
fi
Continue processing …
```

The first part of the statement,

```
egrep -i '^Subject:' /tmp/incoming_message
```

finds every line in the message that begins with the string `Subject:`.
The command

```
| head -1
```

outputs only the first of these lines. This step is a safety mechanism to ensure that messages forwarded to the alias (which in general may contain more than one `Subject:` line) are handled properly. Finally, the `awk` command

```
| awk -F: '{print $2}'
```

defines the colon (:) as the field separator in the record (line) and outputs the second field (the string " `Code Manager notification`" if the message was sent by Configuring). This is the value placed in the variable `SUBJECT`.

Extracting the User Name

The following line extracts the user name from the stored copy of the message and places its value in the script variable USER:

```
USER =`egrep -i '^User:' /tmp/incoming_message \
               head -1 | awk -F: '{print $2}' | awk '{print $1}'`
```

Extracting the Event

The following line extracts the event from the message and places its value in the script variable EVENT:

```
EVENT=`egrep -i '^Event:' /tmp/incoming_message \
                   | head -1 | awk -F: '{print $2}'`
```

Other information can be extracted with similar commands. For example, the date can be extracted by searching for `'^Date:'`.

Example Script

Some of the constructs described in the last section are used in the Bourne shell script shown in Figure 21. Refer to the comments in the script for an explanation of its operation.

Configuring notification mail sent to the alias `script_alias@conundrum` is piped to the script. The script is designed to examine each message sent by Configuring in response to putback events. If the putback is related to a bug fix, group policy requires developers to name the bug with an identifying number in the comment that is included in the message. Bug ID numbers are of the form BUG-*number*. The script searches each message for these ID numbers and, if the numbers appear in a database file, forwards the message to interested managers.

Interested managers are identified in a database file named `bug.dbase`. The file is a list of ordered pairs of bug IDs and user names. For example, the entry

```
BUG-12 nod@naptime
```

identifies `nod@naptime` as the manager interested in `BUG-12`.

If a bug ID that is not in the bug database appears in the message, the message is forwarded for clarification to the user who performed the putback.

Figure 21 Example Script for Detecting Bug IDs in Mail Files.

```
#!/bin/sh -
#
# Name: bugcheck.sh
#   This script checks an incoming Configuring notification mail
#   message to determine whether or not it contains bug ID numbers
of
#   the form BUG-<number> in the putback comments message.
#   If it does, the bug IDs are extracted and compared
#   to a database that contains current bugs and the mail
#   address of the manager(s) responsible for each bug ID.
#   If the bug ID is found in the database, the bug ID is
#   added to the original Configuring notification message,
#   which is then forwarded to the responsible manager(s).
#   If the bug ID is not in the database, the message is forwarded
#   the user who entered the comment.
#
#   The format of lines in the bug ID/manager list is as follows:
```

(Continued)

```
#   BUG-bug_id_number manager_name@machine_name
#
#   The prefix "BUG-" must appear at the beginning of the bug ID.
#   The bug ID and manager name can be separated by tabs or spaces.
#   Only one bug ID and manager can appear on each line in the
database,
#   but the same bug ID or manager can appear more than once.

# Define variables.

BUGPREFIX=BUG-# String prefix for bug ID.
BDB=/home/host/bug.dbase# Path name of bug id/manager database.
WD=/tmp/bugcheck.dir# Temporary working directory.
NDB=$WD/ndb$$# New bug database with tabs stripped.
IM=$WD/im$$ # Temp file for incoming message.
RT=$WD/rt$$ # Temp file for returned message.
BF=$WD/bf$$# Temp file of message bug IDs.

# Create the temporary working directory.

mkdir -p $WD

# Store the incoming message in a temporary file.

cat > $IM

# Change all spaces, tabs, and punctuation (except "-") in message
# to newlines in order to place each word in the message
(including
# the bug IDs) on a separate line.
# Delete duplicates, filter for bug IDs, and store bug IDs in $BF.

cat $IM | \
tr `\001'-'\054''\056'-'\057''\072'-'\100''\133'-'\140'
'\173'-'\177' `\012' | \
sort | uniq | grep "$BUGPREFIX" > $BF

# Change tabs to spaces and make a copy of bug list.

cat $BDB | tr -s `\011' `\040' > $NDB

# If the bug IDs from the message are in the bug database,
```

(Continued)

```
# construct a new SUBJECT line and send the original notification


# message to the registered manager.
# If a bug ID from the message is not in the bug database,
# forward the message (with commentary) to the user
# who executed the putback.

for BUGID in `cat $BF`
do
    MGR=`grep $BUGID\ $NDB | awk `{print $2}'`
    if [ "$MGR" ] ;
    then
    mail -s "Configuring putback notification ($BUGID)" \
    $MGR < $IM
    else
    echo " The putback comment in your message refers to" > $RT
    echo " a bug with ID number $BUGID." >> $RT
    echo " This bug ID is not in the active bug database." >> $RT
    echo " Make sure you have entered the correct bug ID." >> $RT
    echo "" >> $RT
    echo "" >> $RT
    echo "---------Begin Included Message---------------" >> $RT
    cat $IM >> $RT
    echo "" >> $RT
    echo "" >> $RT
    echo "---------End Included Message----------------" >> $RT
    USER=`egrep -i `^User:' $IM \
     | head -1 | awk -F: `{print $2}' | awk `{print $1}'`
    cat $RT | \
    mail -s "Invalid bug ID ($BUGID) in putback comment" $USER
    fi
done

# Clean up and exit

rm -r $WD
exit
```

Example Output

Consider a `bug.dbase` file that contains the following lines (note the absence of `BUG-121`).

```
BUG-12      nod@naptime
BUG-3456    torq@inquisition
BUG-6789    hoover@fbi
```

When this file is used with the mail message shown in Figure 20, the script sends two mail messages. The first message is mailed to `nod@naptime` as a result of being listed in the `bug.dbase` file on the same line as `BUG-12`. The second is sent to `jdoe`, the user who initiated the putback, because he mentioned `BUG-121` in his comments, which is not in the `bug.dbase` file.

The reference to `BUG-12` generates the message shown in Figure 22.

Figure 22 Mail to Interested Manager About `BUG-12`

```
From jdoe Thu Jan 26 13:05 PDT 1995
To: nod@naptime
Subject: Configuring putback notification (BUG-12)
Content-Type: text
Content-Length: 532

From jdoe Thu Jan 26 13:01 PDT 1995
Date: Thu, 26 Jan 1995 13:01:50 +0800
From: jdoe (Jan Doe)
To: script_alias@conundrum
Subject: Code Manager notification

Event: putback-to
Parent workspace: integrate_1
 (holymoly:/export/home/holymoly/jdoe/src/ws/integrate_1)
Child workspace: develop_1
 (holymoly:/export/home/holymoly/jdoe/src/ws/develop_1)
User: jdoe

Comment:
In my attempt to fix BUG-12, I fixed BUG-121 instead!

Files:

update: foo
update: bar
update: foobar
```

The reference to nonexistent `BUG-121` generates the message shown in Figure 23.

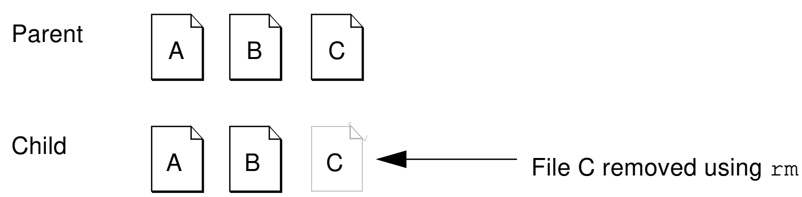
```
From jdoe Thu Jan 26 13:05 PDT 1995
To: jdoe
Subject: Invalid bug ID (BUG-121) in putback comment
Content-Type: text
Content-Length: 820

 The putback comment in your message refers to
 a bug with ID number BUG-121.
 This bug ID is not in the active bug database.
 Make sure you have entered the correct bug ID.

------------Begin Included Message--------------------

From jdoe Thu Jan 26 13:01 PDT 1995
Date: Thu, 26 Jan 1995 13:01:50 +0800
From: jdoe (Jan Doe)
To: script_alias@conundrum
Subject: Code Manager notification

Event: putback-to
Parent workspace: integrate_1
 (holymoly:/export/home/holymoly/jdoe/src/ws/integrate_1)
Child workspace: develop_1
 (holymoly:/export/home/holymoly/jdoe/src/ws/develop_1)
User: jdoe

Comment:
In my attempt to fix BUG-12, I fixed BUG-121 instead!

Files:
update: foo
update: bar
update: foobar

------------End Included Message----------------------
```

Figure 23 Returned Message Due to Nonexistent Bug ID

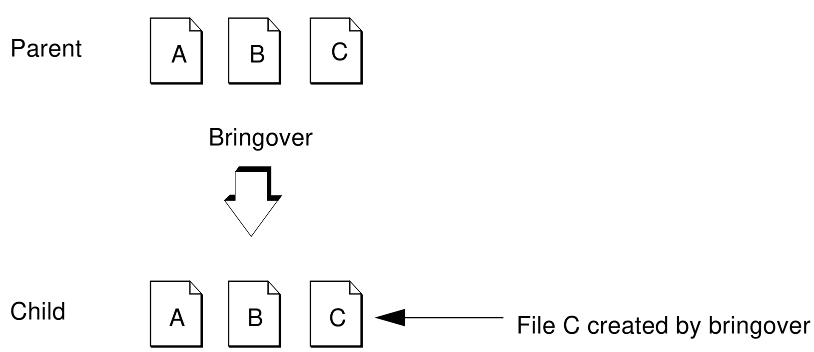
Deleting Files From the Configuring Tool Workspaces

Deleting files from a Configuring workspace is a little trickier than it first appears. Deleting a file from a workspace with the `rm` command causes Configuring to behave (during bringover and putback transactions) as if 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.



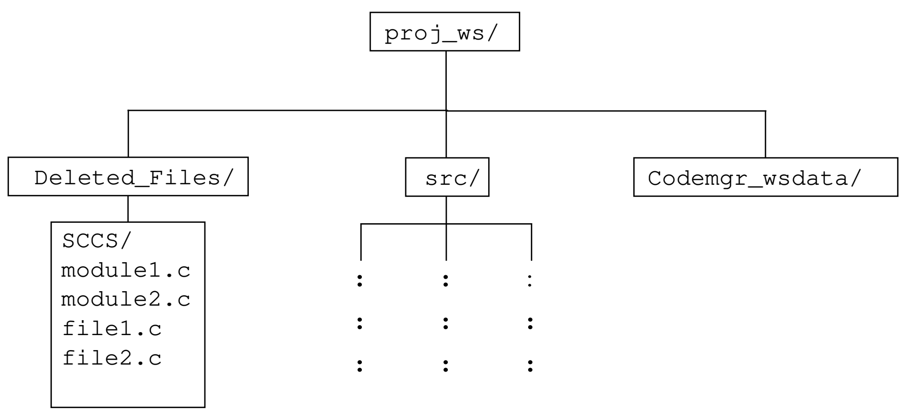
Later, the bringover update transaction is used to update the child. Configuring examines the two workspaces and determines that the file "C" exists in the parent and not in the child — following the usual Configuring rules, it creates "C" in the child.



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
```



   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`, Configuring 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, the Configuring tool:

- Moves the “deleted” files completely out of the source path
- Does not recreate the file
- 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 freezepoint — see *Versioning and Freezepointing User’s Guide* for more information about freezepoints[1])

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.

1. The name “CheckPoint” has been changed to “FreezePoint.”

TW Topic:

Using Makefiles With TeamWare

This topic describes how to write makefiles that take full advantage of Configuring and DMake, the `make` utility that is bundled with the Sun WorkShop TeamWare release. Both Configuring and DMake are designed to be compatible with existing makefiles and methods.

Configuring does not depend on any specific release of `make` or on any other system modeler, but your site's makefile hierarchy may affect the way you organize your Configuring workspaces. In addition, poorly written makefiles may keep you from realizing the full potential of DMake.

Source and Makefile Hierarchies

In order to build and test successfully in a development workspace, a developer must bring over all the files required for the build. In most cases, one or more makefiles are among the files required for the build. Finding the correct makefiles may not be obvious if they reside in directories far from the local build directory, which is the case for many large projects.

The following sections describe the different ways three software development groups structured their makefile hierarchies. The makefile organizations range from casual to formal, and each organization has different implications for Configuring. In each case, the source directory tree was structured along modular lines, with one module or deliverable in each subdirectory of the tree.

Self-Contained Makefiles

The first case describes the Bantam project. The project was small and the number of developers assigned to it were few. The Bantam group decided to use a makefile and source tree organization in which the makefile in each directory was self-contained. This strategy was regarded as sufficient because the project contained few modules and its source directory tree was simple.

Also, because the group was small, coordination between developers was easy. When necessary, preprocessor, compiler, and linker flags were set globally for each build with a few environment variables.

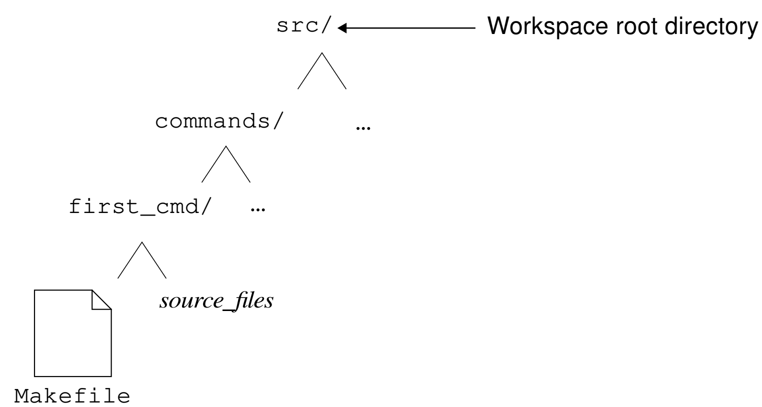


Figure 24 Source and Makefile Organization with Self-Contained Makefiles

The simplicity of this organization had several advantages. First, Configuring bringovers were obvious because all source, including the makefile required for the build, was present at each directory level — a bringover of all files in a directory was sufficient to build successfully. Second, when makefiles were not allowed to grow too large, they were easy to debug because they contained no included files.

A major disadvantage of this organization was that minor unforeseen changes to the build (a new location for include files, for example) required a change to each makefile. Also, style consistency in makefiles was difficult to enforce, which resulted in makefiles that were difficult to maintain.

A final disadvantage became apparent as the Bantam project matured. Over time, the project became larger than first envisioned, and the source tree eventually required so many large makefiles that understanding and maintaining them became difficult.

Direct Inclusion of Makefiles

The second case describes the Midpoint project, which was larger than the Bantam project and required more formal organization. The Midpoint group settled on a strategy of including selected high-level makefiles in low-level makefiles; the top-level makefiles set flags and macros that were used throughout each build. By setting these flags in top-level makefiles, the group was able to centrally control such parameters as the installation location of targets, the locations of special libraries and include files, and the optimization level of compilers.

Some low-level makefiles included makefiles from intermediate directories in the source tree. For example, one family of commands had so much in common that the makefile from a common parent directory was included in the makefile for each command. This case is shown in Figure 25, where `cmd.Makefile` is included in the makefile for the `first_cmd` module.

The group had to be careful not to include high-level makefiles more than once in lower-level makefiles. For example, Figure 25 shows `master.Makefile` being included in the lowest level makefile. It should not be included in an intermediate makefile such as `cmd.Makefile` that is also included at the lowest level, although such multiple inclusions rarely lead to fatal errors.

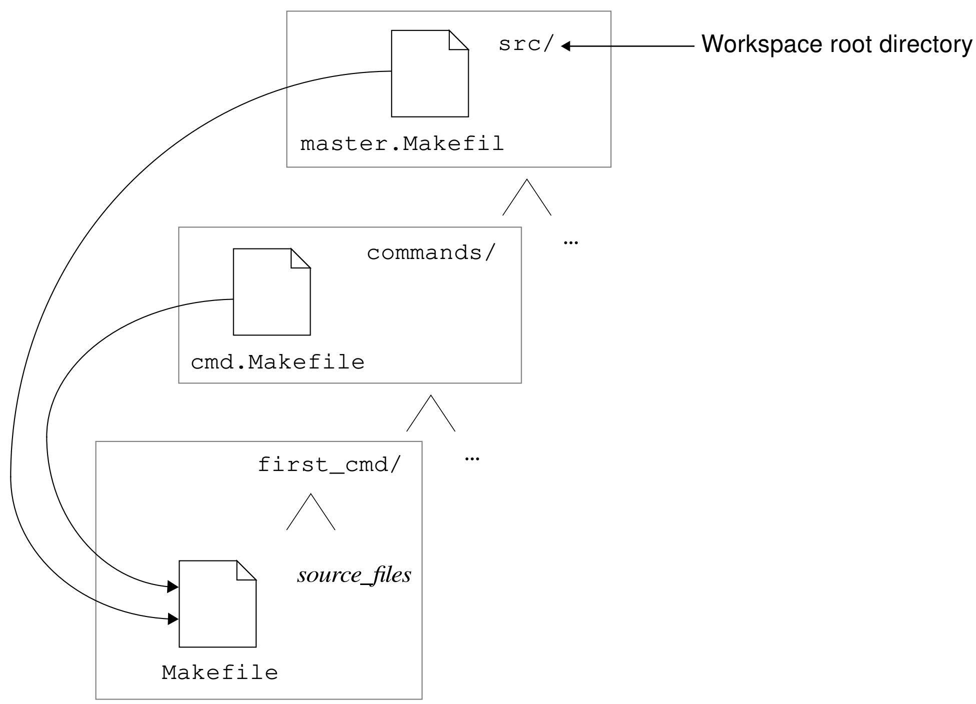


Figure 25 High-Level Makefiles Included Directly in Low-Level Makefiles

The major advantage of this organization was that the Midpoint group could set flags, macros, and directories common to all makefiles at the highest level; a single change in `master.Makefile` applied to all the makefiles in the hierarchy. Makefile style was easier to enforce — in fact, it had to be enforced, at least to the degree that all makefiles used the included top-level makefiles the same way. Individual makefiles were smaller in the Midpoint project than in the Bantam project, but debugging makefiles became more difficult in some cases because of the inclusions.

When the Midpoint group adopted Configuring, another disadvantage of the makefile organization became apparent: a bringover of any module required that the included makefiles also be brought over. The group handled the difficulty by customizing the default File List Program (FLP) so that it brought over not only the contents of the source directory but also automatically brought over the included makefiles. The FLP list of included makefiles had to be updated manually when a new high-level makefile was included in the makefile for a low-level module.

Nested Inclusion of Makefiles

The last case describes the experience of the Hefty project, which was large and required the most formal strategy in its makefile organization. The Hefty group decided to require that each makefile include the makefile immediately above it in the source tree. The most general makefiles were at the top level. Descending the source tree, each subdirectory contained a makefile with increasingly specific build instructions. The formal organization ensured that high-level makefiles were included only once in lower-level makefiles.

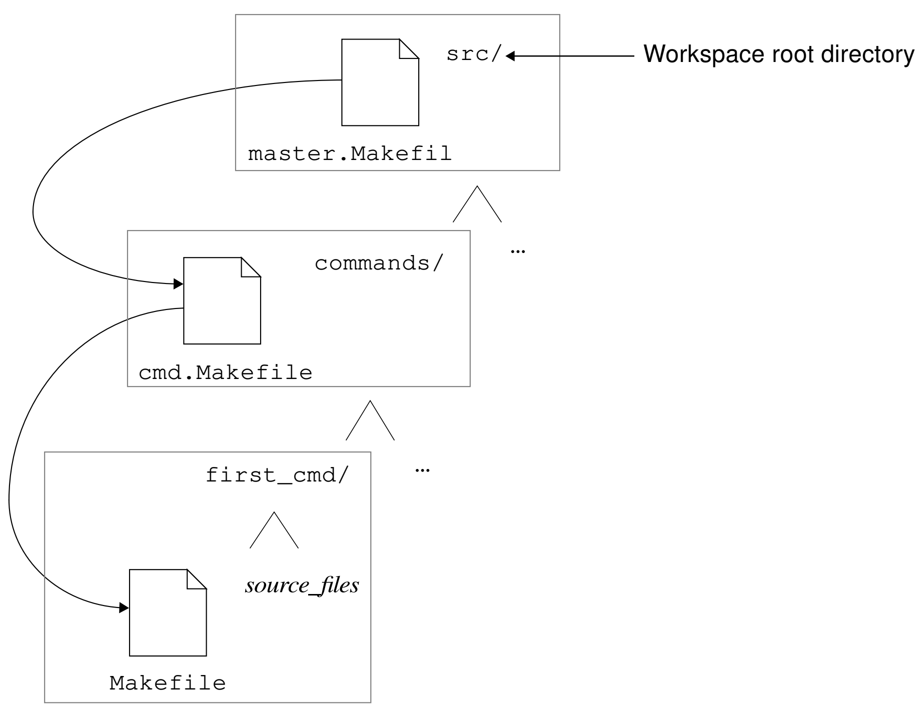


Figure 26 Nested Makefiles — Each Makefile Includes the Makefile Above It

As with the strategy adopted by the Midpoint group, an advantage of the Hefty makefile organization was that makefile style was easy to enforce. Another advantage was that each individual makefile was small. However, because of the large number of nested includes, understanding and debugging makefiles at the lowest levels sometimes proved difficult.

When the Hefty group adopted Configuring, they recognized that each developer had to be sure to bring over the entire list of included makefiles in order to build successfully. The listing of makefiles was automated by a custom FLP and script. This makefile organization and use of the custom FLP is similar to that used by the SunOS™ group. See "Using File List Programs" on page 32 for a description of that group's experience.

More About Configuring and Makefiles

When Configuring copies source files between workspaces in bringover and putback transactions, it relies on SCCS to identify, control, and track the files. In fact, Configuring operates exclusively on SCCS files, so the decision about what files to place under SCCS control is important.

Projects that use `make` to manage builds usually treat makefiles as source files, placing them under SCCS control and copying them between workspaces during bringovers and putbacks.

Makefile Contents

A makefile contains a list of derived modules (usually files), called *targets*, and the modules required to build them (called *dependencies*). A target may be the final result of a long series of compilations (a *top-level* target) or an intermediate result (*secondary* target) used to build a top-level target. Typical secondary targets are the object (`.o`) files produced by C compilers. A makefile may also contain explicit rules describing how a target is to be built from its dependencies.

Makefiles as Derived Source

One form of secondary target is the *derived source* file, a source file that is generated automatically near the beginning of a build. At some sites, makefiles or parts of makefiles are written automatically at the beginning of the build process by a makefile generator and are themselves derived source.

Some insights into generated makefiles can be gained from the experience of the Concoct project development group. This group inherited a build environment that included generated makefiles. When the group adopted Configuring, they needed to find a way to enforce consistency in their generated makefiles.

Generated Makefiles

As the Concoct group reviewed the original rationale for using makefile generators, they realized that the main reason was to read the contents of one or more directories, identify the source files in them, and explicitly list those file names in the generated makefile. For example, one makefile generator scanned the build directory for C++ source files of the form `*.cc`, built a list, and wrote a makefile that included the list explicitly.

This approach relieved developers from hand editing makefiles whenever they created or renamed a source file. However, when a generated makefile was placed under SCCS control, the following permission conflict resulted: subsequent builds tried to overwrite the makefile even though it was read-only as a consequence of being under SCCS control. This condition created a problem because a makefile that is not under SCCS control is not put back into the parent workspace following a build.

Controlling Generated Makefiles

The Concoct group actually used three different makefile generators:

- *Generator 1* — `imake`, a compiled binary that read a template, a set of `cpp` macro functions, and an input file in each build directory. Using these text files as instructions, `imake` generated makefiles.
- *Generator 2* — A script that performed the same function as `imake`, but used different input files.
- *Generator 3* — A script that generated makefiles without reading input instructions. The script simply listed the build directory contents and treated all listed files as source.

The group realized that it could control its generated makefiles indirectly by placing the inputs to the makefile generators (or the generators themselves if they were scripts and not compiled binaries) under SCCS control, as illustrated in Figure 27.

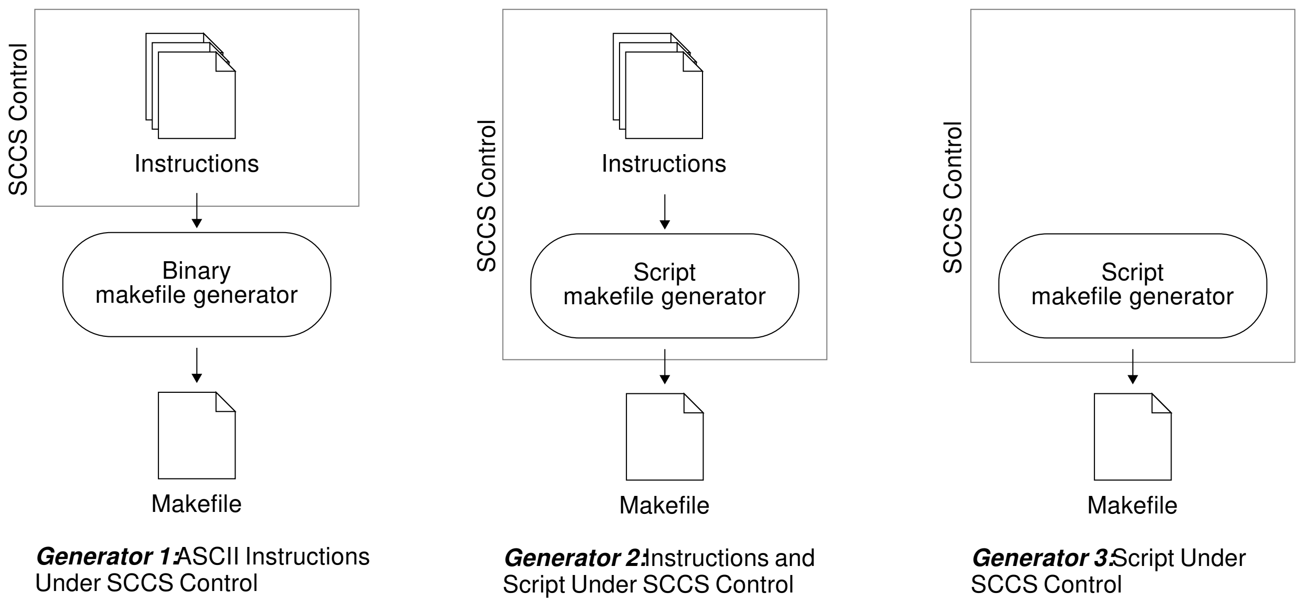


Figure 27 Putting Makefile Generators Under SCCS Control

In each case, only true source (either the input to the makefile generator or the generator itself) and not derived source (the generated makefiles) were under SCCS control. As a result, new makefiles were built consistently and the information used to generate makefiles was propagated between workspaces.

The group agreed on a policy to ensure that makefile consistency would not be compromised:

- Each build would create generated makefiles anew, and the generated makefiles would not be edited by hand after being built.
- If a generated makefile had to be edited, an automated way to edit it would be provided. The script that automated the editing would be placed under SCCS control and treated as source.
- Generated makefiles would be deleted after each build (with the `make clean` pseudotarget) along with other secondary targets such as object (`*.o`) and library (`*.a`) files.

Devguide as Makefile Generator

The Concoct group produces software for the XView™ environment and uses the Open Windows™ Developer's Guide (Devguide) to develop the Graphical User Interface (GUI) for its product. Devguide generates a makefile as part of its output, and so may be considered a makefile generator. Devguide consists of a GUI Design Editor and a set of postprocessors that convert the output of the editor into C or C++ source code for use with a variety of GUI libraries and toolkits.

The GUI Design Editor provides a graphical way for developers to design user interfaces without writing code. After the interface has been designed on the screen, the result is saved to a GIL (Guide Interchange Language) file, an intermediate text-based representation of the interface. The GIL file is a source file — the GUI Design Editor that creates it is analogous to the text editor a developer might use to write an ordinary source file. Because the Concoct group's executables are targeted for an XView application, they run the postprocessor GXV (Guide-to-X View) on the GIL file to generate XView source in the form of a header file, two source files, and a makefile. These derived sources are used by `make` to build the desired executable.

The use of GXV (or one of the other Devguide postprocessors) presents the same challenge to source code control as a makefile generator — the files actually used by `make` to build the executable are derived sources. Because they are derived during the build process, they should not be placed under SCCS control. The GIL file created by the GUI Design Editor, however, can be considered true source because it is generated by the developer prior to the build process. As true source, the GIL file can be placed under SCCS control, and Configuring will bring it over and put it back along with other source.

The first step in the Concoct group's automated build process is to run the postprocessor GXV to regenerate the derived source. Once created, the derived source is never hand edited and is deleted along with other intermediate targets following the build.

In deciding not to hand edit derived source, the group avoided the following complications: if the derived source were to be hand edited, it would have to be placed under SCCS control (to make it eligible for Configuring bringovers). In order to allow the derived source to be rederived, the makefile would have to be enhanced to check out the derived source (making it writable), the hand edits would have to be merged into the checked out file, and the result checked back in to SCCS. Only then could the project build begin.

Makefiles Using Built-In `make` *Features*

Following a major release milestone, the Concoct group decided to rewrite its entire code base. They took advantage of the occasion to rewrite their makefiles as well in order to eliminate their makefile generators (with the exception of Devguide). They discovered that they could achieve the same goals by writing makefiles that exploited the built-in features of `make` (dynamic macros, pattern-matching and implicit rules, conditional macro assignment, and so on). These makefiles could be placed under SCCS control and propagated between workspaces as true source.

In designing its makefile hierarchy, the Concoct group adopted the makefile inclusion strategy used by the Midpoint group (shown in Figure 25 on page 96). In this strategy, one or more top-level makefiles are included in each makefile in the directory hierarchy.

The example makefiles are not explained in great detail. Refer to the `make` man page and the manuals *SunOS 5.0 Programming Utilities* and *SunOS 5.0 User's Guide* for more information on `make` and makefiles.

Master Makefile

The master makefile shown in Figure 28 defines several macros (`LDLIBS`, `CC`, `CPPFLAGS`, and so on). It also sets the special-function target `.KEEP_STATE` for all makefiles. Because the master makefile is included in all other makefiles, these definitions apply throughout each build.

```
# master.Makefile, to be included in all
# subordinate makefiles.

.KEEP_STATE:

LDLIBS = -L/usr/lib
CC = cc
CPPFLAGS = -DSUN5_x
CFLAGS = -g -xs
LINK.c = ${CC} ${CFLAGS} ${CPPFLAGS} ${LDFLAGS}
```

Figure 28 Top-Level Makefile for the Concoct Project

Low-Level Makefile

A typical low-level makefile for the Concoct project is shown in Figure 29. This makefile is used to build an executable named `cogitate`.

Comments in the file explain the constructs of its targets, dependencies, and rules. Note that the makefile automatically absorbs any new C source files (files with a `.c` suffix in their names) that may be introduced in the build directory. This capability, implemented with a pattern replacement macro, helps eliminate the need for a makefile generator.

```
# First, include the master makefile.
# This low-level makefile assumes that master.Makefile
# is two directories above the current directory.

TOP = ../..
include ${TOP}/master.Makefile

# The next macro definition uses command
# substitution to list all C source files in
# the current directory.

CSRCS:sh = ls *.c

# The next pattern replacement macro creates
# a list of object file names, based on the
# source file names.

COBJS = ${CSRCS:%.c=%.o}

# The next rule uses a dynamic macro:
# $@ represents the name of the current target, in
# this case "cogitate." The rule links all object
# files and produces an executable named "cogitate".

cogitate : ${COBJS}
    ${LINK.c} -o $@ ${COBJS} ${LDLIBS}

# Next, the targets and dependencies are listed by
# means of a pattern matching rule.
# The last build rule uses a dynamic macro:
# $< is the name of a dependency file.

%.o: %.c
    ${CC} ${CFLAGS} ${CPPFLAGS} -c $<
```

Figure 29 Low-Level Makefile for Building the `cogitate` Executable

Using Makefiles With DMake

DMake, the make utility bundled with Sun WorkShop TeamWare, can build several targets simultaneously in parallel processes. This new capability contrasts with earlier versions of make, which build targets one at a time in the sequence in which they appear in the makefile. DMake is a syntactic superset of the standard Sun™ version of make. It supports all the syntax of standard make and works with existing makefiles.

Improving Performance With DMake

Most build processes are I/O bound, which means that they are limited by the speed at which data can be read and written from mass storage devices, not by CPU performance. Therefore, building targets in parallel results in performance increases on all machines, even single-processor ones, but the greatest improvements are seen in the new classes of multiprocessor servers and workstations.

Listing Dependencies Explicitly in Makefiles

Most makefiles that were written for earlier versions of make will work with DMake. When problems occur, they usually result from a reliance on the order in which targets appear in a makefile to establish the build order. These problems can be avoided by explicitly listing each target's dependencies rather than relying on the build of another target to bring the dependencies up to date. For example, consider the following makefile fragment, which the Concoct group inherited from an early version of the project:

```
all:prog1 prog2
prog1: prog1.o aux.o
    $(LINK.c) prog1.o aux.o -o prog1
prog2: prog2.o
    $(LINK.c) prog2.o aux.o -o prog2
```

When built in serial, the target aux.o was built as a dependent of prog1 and was up to date for the build of prog2. However, when built in parallel, the link of prog2 sometimes began before aux.o had been built, and was therefore incorrect. The group corrected the dependency list for the prog2 target to read:

```
prog2: prog2.0 aux.o
```

Controlling DMake With Special Targets

Other cases of the implicit ordering of dependencies were more difficult for the Concoct group to identify and correct. The *ParallelMake User's Guide* discusses these cases in detail. In the end, the group used the following special-function targets to handle their most difficult problems:

- `.WAIT` — When inserted into a dependency list, the special-function target `.WAIT` causes DMake to complete the processing of all prior dependents in the list before proceeding with the following dependents. For example,

  ```
  all: hdrs .WAIT libs functions
  ```

  causes DMake to wait until the `hdrs` dependency is built before building `libs` and `functions`.

- `.NO_PARALLEL:` — The target `.NO_PARALLEL:` causes a list of targets to be built in serial rather than in parallel. For example,

  ```
  .NO_PARALLEL: prog1 prog2
  ```

  When all targets must be built serially, `.NO_PARALLEL:` can be specified without arguments. If there are exceptions that can be built in parallel, they can be identified with the `.PARALLEL:` special target.

- `.PARALLEL:` — When `.NO_PARALLEL:` is used without arguments, `.PARALLEL:` is used to identify targets that can be built in parallel. For example,

  ```
  .NO_PARALLEL:
  .PARALLEL: prog3 prog4
  ```

  causes all targets to be built serially with the exception of `prog3` and `prog4`, which can be built in parallel.

Using DMake Serially

Finally, DMake can be forced to behave serially for all targets by using the `-R` option. While this option guarantees compatibility with working existing makefiles, it sacrifices the improved performance possible with DMake.

TW ***Topic:***

Using the Configuring Tool File List Programs

The default behavior of the Configuring tool bringover and putback transactions is to search the directories you specify for SCCS files and act on them. This default behavior (expanding specified directories into the SCCS files they contain) is the result of a script known as the *default directory file list program*, named `def.dir.flp`. This part of Configuring was purposely designed to be accessible to users so they could customize its behavior if the need arose.

The action of `def.dir.flp` can be changed by providing your own file list program (FLP), the function of which is to list the files you want to bring over or put back on `stdout`. The FLP is called by the Configuring transaction, which reads the list of file names and acts on them. Most FLPs are scripts, but any executable can be run as an FLP.

This topic describes how to use FLPs to full advantage.

Levels of FLP Use

All Configuring users use FLPs, whether they know it or not. By running a bringover transaction and specifying directory names, for example a user causes `def.dir.flp` to run in each specified directory in both the parent and child workspaces. In the form in which `def.dir.flp` is shipped, it prints on `stdout` the names of all SCCS files in the directories and their subdirectories. The bringover transaction acts on the union of all listed files.

Because most projects take advantage of the hierarchical file system to organize their projects into directories, this default behavior is usually all that is required.

Using Directory FLPs

The build process in some directories may require a file that is not under SCCS control or that is contained in the workspace but lies outside a specified directory. In these cases, users can place their own FLP, called `dir.flp`, in the directory. The `dir.flp` can simply echo the names of required files on `stdout`, or it can use an algorithm to produce the list of file names. For example, a directory FLP might parse the directory's makefile and list target dependencies.

The `def.dir.flp` looks in each specified directory for a `dir.flp`, and runs `dir.flp` if it exists. This is the most basic level of FLP customization.

Using CODEMGR_DIR_FLP

You can also specify an FLP by setting the environment variable `CODEMGR_DIR_FLP` to the name of an FLP you want to use. The `def.dir.flp` checks to see if this variable is set, and if it is, it runs the FLP that is the variable's value. This is a more general level of customization because the FLP is used automatically by every transaction between the workspaces, regardless of directory.

Specifying an FLP Directly

Another way to execute an FLP is to specify it directly on the command line or in the graphical user interface. When you specify an FLP in a bringover or putback transaction, the FLP runs once in the root directory of each workspace. Any directories you also specify are treated independently in the standard fashion, by running `def.dir.flp` in each of them. For example, consider the following command, executed in a child workspace:

```
tutorial% bringover –f ~/aux.flp dir1 dir2
```

This command causes the FLP `aux.flp` (in this case, contained in the user's home directory) to be run in the root directories of the parent and child workspaces. The `aux.flp` can produce its list of file names in any fashion, but often it simply echoes a list of file names contained within the FLP itself. The `bringover` command treats the file names produced by `aux.flp` as though they had been specified on the command line along with `dir1` and `dir2`.

After `aux.flp` runs, the `bringover` command treats the specified directories in the standard fashion: it changes to each directory in turn in each workspace and runs `def.dir.flp`. The directories can be specified relative to the workspace roots, as are `dir1` and `dir2`, or specified as full path names (in which case Configuring computes the relative path names). If the directories lie outside of either workspace, they are ignored.

Replacing the Default Directory FLP

The highest level of customization is to replace `def.dir.flp` with an executable of your own. When you customize `def.dir.flp`, your changes affect every bringover and putback at your site. This level of customization is powerful and imposes a corresponding amount of responsibility on the person who implements it. The customization itself, however, is quite easy to accomplish.

FLP Execution Context

One limitation of FLPs arises from the fact that they are called automatically by Configuring. Because FLPs are not executed from the command line, you cannot pass values to them by means of command-line arguments. An easy workaround is to put values in environment variables that are then used by the FLP.

The last section mentioned the CODEMGR_DIR_FLP environment variable, which you can set to the name of an FLP to use with all bringover and putback commands. When a Configuring transaction starts, it sets several other environment variables for use by FLPs. These variables are summarized in Table 0-1. You can, of course, set other environment variables for use by your own custom FLPs.

Table 2 Environment Variables for Use with FLPs

Environment Variable	Meaning
CODEMGR_CMD	Contains the name of the Configuring transaction being run.
CODEMGR_WS_CHILD	Contains the name of the child workspace for this Configuring transaction.
CODEMGR_WS_DEST	Contains the name of the destination (to) workspace for this Configuring transaction.
CODEMGR_WS_PARENT	Contains the name of the parent workspace for this Configuring transaction.
CODEMGR_WS_SRC	Contains the name of the source (from) workspace for this Configuring transaction.
CODEMGR_WS_ROOT	The bringover and putback transactions execute FLPs in both the parent and child workspaces. This variable contains the name of the workspace in which the FLP is currently executing.

The Default Directory FLP

Figure 30 shows the `def.dir.flp` that ships with every version of Configuring, omitting some environment-checking statements to improve clarity. The `def.dir.flp` checks for a setting for `CODEMGR_DIR_FLP`, then for the existence of a dir.flp in the current directory. The def.dir.flp runs these other FLPs, if they exist. Finally, it uses find to print all SCCS files in the current directory and its subdirectories to stdout.

```
#! /bin/sh
# Default directory file list program (FLP).
# Decides which FLP to run.
#
# Look first for a shell environment variable named
# CODEMGR_DIR_FLP. If it exists, assume it names
# an FLP and run it.
#
# Next look for a file named "dir.flp" in the current
# directory.  If it exists, run it.
#
# Finally, look for files under SCCS control.
#
#
if [ -n "$CODEMGR_DIR_FLP" ]; then
    $CODEMGR_DIR_FLP
elif [ -f dir.flp ]; then
    ./dir.flp

else
    #
    # Recursively find all the files in SCCS subdirectories
    # below the current directory
    #
    find `pwd` -name 's.*' -print | grep '/SCCS/s\.'
    #
    # If find doesn't find anything it will exit with value 1
    # and bringover and putback will stop.  Always exit 0 here.
    #
    exit 0
fi
```

Figure 30 The Configuring Standard Default Directory FLP, def.dir.flp

Notice that the find command invoked in the def.dir.flp uses an expansion of `pwd` rather than '.' to represent the current working directory. The reason for this usage is that find may run in directories below the workspace root. The `'.'` notation represents each current working directory where `find` runs, but `` `pwd` `` produces a *full path name* to each of these directories. The full path name encompasses the workspace root, which Configuring can identify. Because each search must start at the workspace root, `` `pwd` `` gives Configuring a useful path name.

The `def.dir.flp` lists the full path names of the `s.*` files. The Configuring transactions use the full path names to compute path names relative to the workspace root. An FLP can also list the relative path names of files, in which case Configuring interprets them as relative to the workspace root. However, if a file name written by an FLP is not under either workspace named in the transaction, Configuring ignores it.

Figure 31 illustrates the flow control during a transaction as the `bringover` command executes FLPs. The diagram is somewhat simplified in that does not account for such features as checking the `Codemgr_wsdata/args` file in each workspace for recently used arguments.

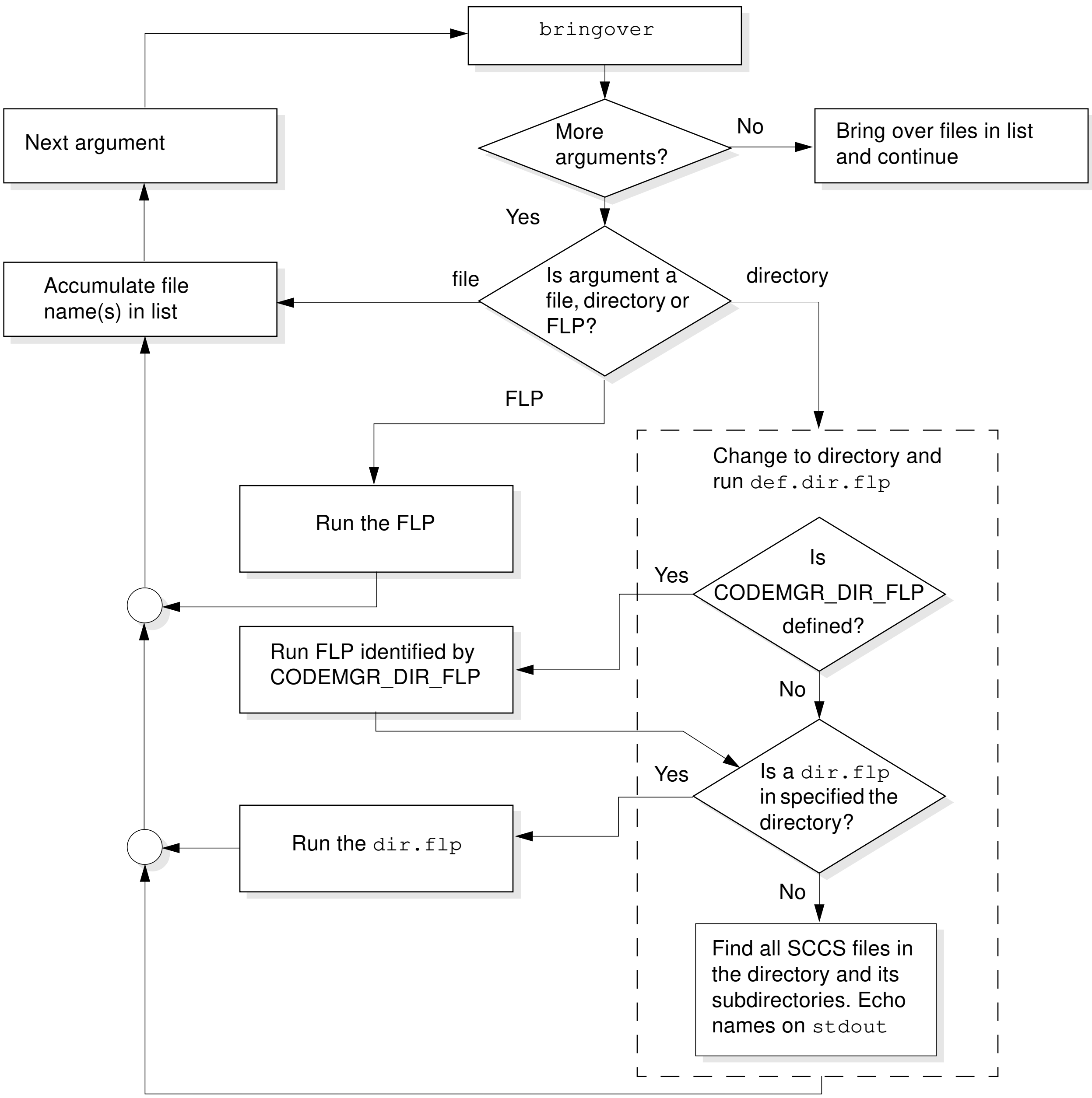


Figure 31 FLP Control Flow During a Bringover Transaction

Directory File List Programs

A directory FLP (dir.flp) is typically used to bring over files that lie above the current working directory in the workspace directory tree.

Figure 32 shows a dir.flp, called by the standard def.dir.flp. It lists the name of a makefile, Makefile.cmd, in the bringover or putback transaction for the doit directory, where dir.flp is stored. Makefile.cmd is included in the local makefiles used in the builds for development of all of this project's commands.

Figure 32 A Directory FLP: dir.flp (Continued)

```
#! /bin/sh
# Directory file list program (dir.flp).
#
# Report the master makefile required by
# every makefile in the project
#
echo "$CODEMGR_WS_ROOT/usr/src/cmd/Makefile.cmd"
#
# Recursively find all the files in SCCS subdirectories
# below the current directory
#
find `pwd` -name 's.*' -print | grep '/SCCS/s\.'
#
# If find doesn't find anything it will exit with value 1
# and bringover and putback will stop.  Always exit 0 here.
#
exit 0
```

Note that control does not return to def.dir.flp after dir.flp exits. Therefore, a find statement identical to the one in the def.dir.flp has been included to search for source files under SCCS control.

Figure 33 shows the position of Makefile.cmd in the workspace directory hierarchy. When a bringover command specifies the doit directory, Makefile.cmd is brought over automatically.

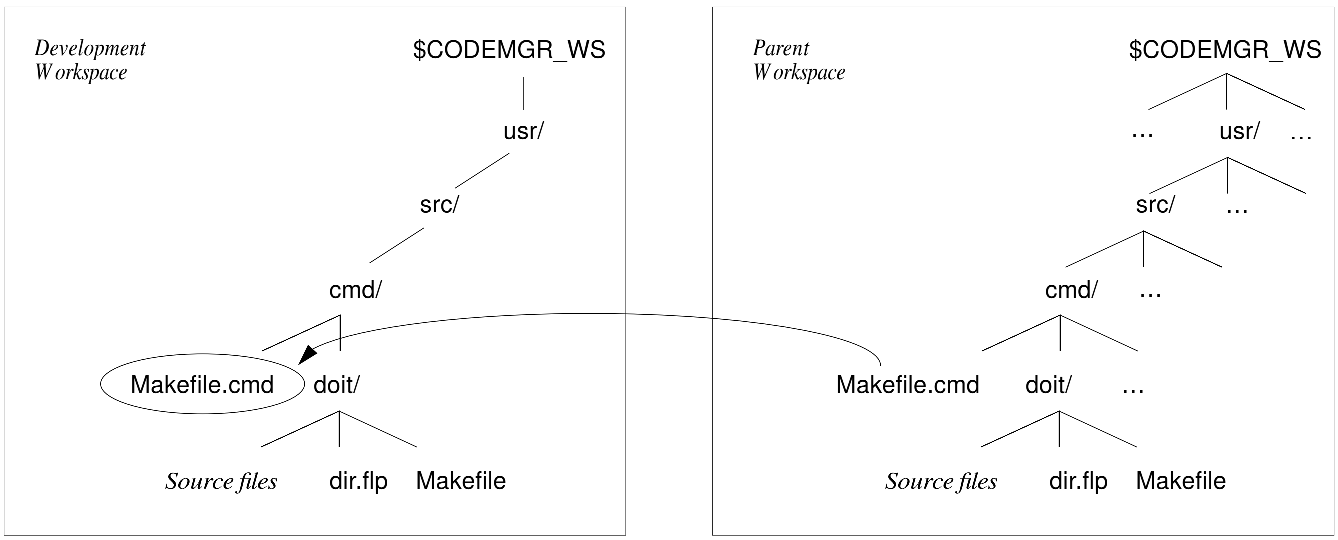


Figure 33 A `dir.flp` Brings a Makefile into a Development Workspace

Directory FLPs are useful for bringing special files into specific directories without having to list them manually during each bringover or putback. However, to achieve a global effect, you can modify the `def.dir.flp`.

Customizing the Default Directory FLP

Some projects require that certain files (typically makefiles and header files) be brought over into every workspace where a build will take place. By customizing the `def.dir.flp`, you can globally specify the files to be moved in bringover and putback transactions. This section presents the experiences of two development groups who produced custom default directory FLPs. The examples illustrate different approaches to listing files not reported by the standard `def.dir.flp`.

Finding Auxiliary Files with `def.dir.flp`

Development Group A needed to bring makefiles into each development workspace automatically. In their project structure, each build directory contains a local makefile that includes other makefiles from higher in the directory tree. All workspaces require one master makefile that is kept at the top of the workspace hierarchy, and each directory needs additional makefiles to completely describe local builds. The group decided not to write an individual FLP (`dir.flp`) for each directory. Instead, they maintain lists of the required makefiles in each directory and assign the task of echoing those lists to a custom `def.dir.flp`.

The custom `def.dir.flp` developed by Group A is shown in Figure 35. It is a C-shell script. In each of the project's development directories is a file, called `inc.fl`, that lists files that must be included in a bringover in order to build the executables in the directory. The file names can be absolute path names or names relative to the directory in which the `inc.fl` file resides. The `inc.fl` files are not directory FLPs or even scripts. They merely contain lists of files, although they may also contain comments preceded by "#" symbols.

Figure 35 adds two functions to the standard `def.dir.flp`:

- First, it searches the directory tree for `inc.fl` files and accumulates their absolute path names in a temporary file. The script then computes the absolute path names of the files listed in the `inc.fl` files and echoes them on `stdout`.
- Second, it searches for makefiles under SCCS control in the directory tree between the current working directory and the workspace root directory and echoes the names of these files on `stdout`.

An example of the effect of the custom `def.dir.flp` is shown in Figure 34. When a user brings over files into a workspace for building the `ls` command, the script finds two makefiles in the directory tree above the directory `ls` and brings them over into the workspace. It also finds an included file, `custom.h`, which was listed in `inc.fl`, and brings it over. In addition, the script brings over all the SCCS-controlled source files in and below the `ls` directory, just as the standard `def.dir.flp` does.

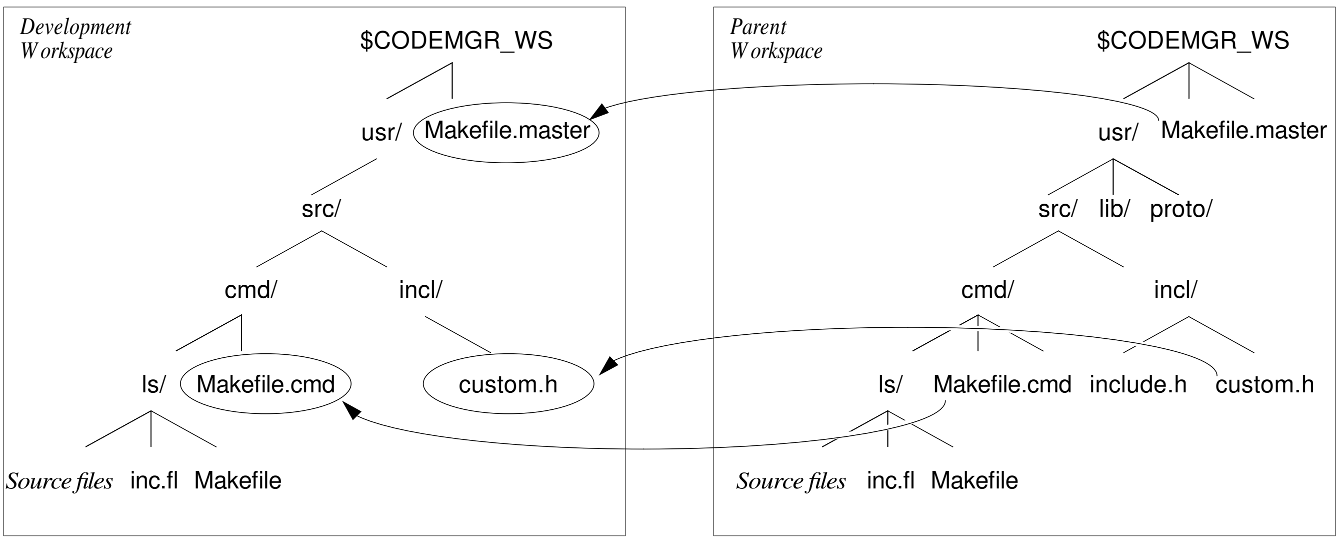


Figure 34 A Custom `def.dir.flp` Brings Outlying Files into Each Development Directory

Note that Figure 35 provides another example of passing a value into an FLP with an environment variable. During the processing of the `inc.fl` files, the script replaces instances of the string "`$CODEMGR_WS_ROOT`" by the value of the environment variable of the same name. This technique can be applied to any environment variable.

```
#! /bin/csh -f
#
# This customized default directory FLP performs the following
# functions:
#
# 1.    Does the same thing as the standard def.dir.flp, that is,
#       looks for files under SCCS control in and below the
#       current working directory.
# 2.    Also reports the contents of any file named 'inc.fl'.
#       Relative paths in 'd/inc.fl' are considered relative
#       to directory 'd'.  Instances of the string
#       "$CODEMGR_WS_ROOT" are replaced by the value
```

Figure 35 A C-Shell Custom `def.dir.flp`

```
#       of the environment variable of the same name.
#       Comments preceded by '#' are ignored.
# 3.    Also reports makefiles under SCCS in the directories
#       between the current working directory an the workspace
#       root directory.
#
#       Recursively find all the files in SCCS subdirectories
#       below the current directory and accumulate path names of
#       inc.fl files in a temp file.
#
set exit_status = 0
cat /dev/null > /tmp/inc.fl$$
find `pwd` -name 's.*' -print -o \
    -name inc.fl -exec /bin/sh -c 'echo $0 >> \
    '/tmp/inc.fl$$ {} \; | grep '/SCCS/s\.'
#
# For each inc.fl file listed in /tmp/inc.fl$$, write it to
# stdout, processing the output as follows:
#   1.  Delete lines which contain only a comment. For example,
#           # This is a comment line
#   2.  Strip comments off the end of lines. For example,
#           /ws/Makefile   # Top level makefile
#   3.  Replace instances of '$CODEMGR_WS_ROOT' with environment
#       variable $CODEMGR_WS_ROOT. For example,
#           $CODEMGR_WS_ROOT/usr/src/uts/README
#   4.  Fully qualify relative pathnames with respect to the
#       files' containing directory. For example:
#               README
#           might be converted to:
#               /ws/usr/src/project/README
#
if (`wc -l < /tmp/inc.fl$$` != 0) then
    # If the CODEMGR_WS_ROOT env is not set, set the
    # SHELL variable by the same name so that the sed
    # command can be executed. The leading "/" prevents
    # this script from considering it a relative pathname.
```

Figure 35 A C-Shell Custom `def.dir.flp` (Continued)

```
    if ( ! $?CODEMGR_WS_ROOT ) then
        set basename = `basename $0`
        /bin/sh -c \
        "echo '$basename':' warning - CODEMGR_WS_ROOT not set'\
            >&2"
        set CODEMGR_WS_ROOT = '/CODEMGR_WS_ROOT'
    endif
    foreach f ( `cat /tmp/inc.fl$$` )
        sed -e '/[ ]*#/d' -e 's/#.*//' \
            -e 's#^\$CODEMGR_WS_ROOT#'$CODEMGR_WS_ROOT'#' \
            -e 's#^[^/]#'`dirname $f`'/&#' $f
    end
endif
#
# Clean up.
#
rm -f /tmp/inc.fl$$
#
# Report any makefiles under SCCS in the directory between the
# current working directory and the workspace root directory.
#
unset done
while ( ! $?done )
    # If the working directory has a Codemgr_wsdata directory,
    # then the working directory is the root of the workspace.
    # Because the workspace root might be missed, check
    # for the working directory '/'.  Even if the ws root dir
    # is missed and some SCCS files that are not in the workspace
    # are reported, it won't cause any problems because
    # Code Manager ignores such files.
    #
    if ( -d Codemgr_wsdata || (`pwd` == '/') ) then
        set done
    else
        cd ..
        if ( -d SCCS ) then
            # Use the same find as is used above
            # to find SCCS-controlled makefiles.
            find `pwd`/SCCS -name 's.*' -print | \
                grep '/SCCS/s\.*akefile*'
        endif
    endif
```

Figure 35 A C-Shell Custom `def.dir.flp` (Continued)

```
end

#
# Always exit 0 here so that Code Manager
# won't conclude that the FLP failed.
#
exit (0)
```

Figure 35 A C-Shell Custom `def.dir.flp (Continued)`

Finding Auxiliary Files With Secondary Scripts

The next example is taken from the experience of Development Group B. Like Group A, Group B also needed to list makefiles in the directory tree above each specified directory. In addition, they wanted to identify and transfer specific files below each specified directory that might not be under SCCS control. Group B also decided against devoting a `dir.flp` in each directory to this task. Instead, they wrote a set of auxiliary scripts that are called by their custom `def.dir.flp` to identify these files. This arrangement allowed them to preserve the flexibility of placing a `dir.flp` in a specific directory if circumstances required.

The FLP shown in Figure 37 finds included files by descending the directory tree and calling a secondary script named `inc.flp` in each directory. Each `inc.flp` echoes the names of files that must be included in its directory in order to build successfully. The `def.dir.flp` performs a similar search up the directory tree, searching for and executing scripts named `req.flp`. Each `req.flp` echoes the names of required files such as makefiles that may reside above the specified working directory.

Using this custom `def.dir.flp`, the developer needs to specify only one directory, such as `/usr/src/cmd/ls`, in his bringover transaction. The `def.dir.flp` looks in the directory and recursively lists its SCCS-controlled files just as the standard `def.dir.flp` does. In addition, it looks for scripts named `inc.flp` or `req.flp` in each directory. If the scripts are in the directory, they are executed. The custom `def.dir.flp` then searches for `req.flp` in ancestor directories and for `inc.flp` in descendant directories, executing the scripts when it finds them (Figure 36).

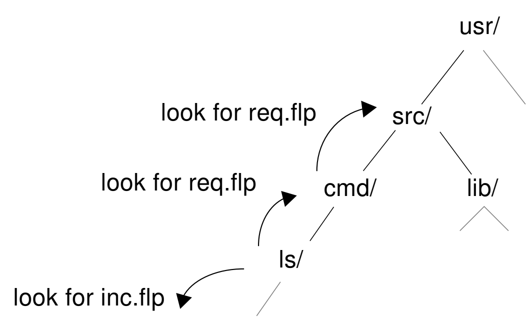


Figure 36 The Custom `def.dir.flp` Searches For and Executes Secondary Scripts

The custom `def.dir.flp` adapts to secondary scripts written in any shell language. If the secondary FLP (`inc.flp`, `req.flp`, `dir.flp`, or a script defined by `CODEMGR_DIR_FLP`) is written in Bourne shell, it is in-lined directly to take advantage of the functions defined in the custom `def.dir.flp`. If the FLP is written in another shell language, it is executed by its native interpreter.

Figure 37 A Bourne-Shell Custom `def.dir.flp`

```
#! /bin/sh
#
# Default directory file list program (FLP),
# modified by Group B.
#
# Look first for a shell environment variable named
CODEMGR_DIR_FLP.
# If it is exists, assume it names an FLP and run it.
#
# Next look for a file named "dir.flp" in the current
# directory.  If that exists, run it.
#
# Finally, do the standard thing - look for files under SCCS.
#
# find_files(pat,dirs...)
#
# pat      = pattern to pass to find
# dirs     = space-separated list of directories for find to
visit
```

Figure 37 A Bourne-Shell Custom `def.dir.flp` (Continued)

```
find_files() {

    pat=$1
    shift

    for dir in $*
    do
            if [ -d $CODEMGR_WS_ROOT/$dir ]; then
                find $CODEMGR_WS_ROOT/$dir -name "$pat" -print | \
                grep "/SCCS/s."
            fi
    done

} # find_files()

echo_file() {
    #
    # Check to make sure a file exists, if it does then
    # echo it out.
    #
    if [ -f $CODEMGR_WS_ROOT/$1 ]; then
            echo $CODEMGR_WS_ROOT/$1
    fi
} # echo_file()

# exec_file(script)
#
# script =
#   full path name to script to be executed
#           or
#   relative path name to script to be executed
#        (relative to root of workspace)
#
# exec_file will execute the script pointed to by 'script'.
# It will do this by interpreting the #! notation at the top
# of the file to determine what 'interpreter' to run
# for the script.
#
# Note: If the ((script == /bin/sh) || (script == /usr/bin/sh))
# then it is actually in-lined with the
#         . ${script}
# notation.  This gives the additional functionality
```

Figure 37 A Bourne-Shell Custom def.dir.flp (Continued)

```
# of being able to inherit functions defined by the def.dir.flp
# script (this file).
#
exec_file() {
    FILE=$1
    if [ `expr "$FILE" : '^/'` = "0" ]; then
            #
            # If not absolute path, then prepend $CODEMGR_WS_ROOT
to
            # path.
            #
            FILE=$CODEMGR_WS_ROOT/$FILE
    fi
    MAGIC=`head -1 $FILE`
    if [ `expr "$MAGIC" : '^#!'` = "0" ]; then
            #
            # No #! notation - assume this is a SHELL script
            #
            SHELL="/bin/sh"
    else
            #
            # Set SHELL to string after #!
            #
            SHELL=`echo $MAGIC | sed -e "s/^\#\![ \t]*//"`
    fi
    if [ "(" `expr $SHELL : '^/bin/sh'` = "0" ")" -a \
            "(" `expr $SHELL : '^/usr/bin/sh'` = "0" ")" ]; then
            #
            # Not Bourne shell, so execute the interpreter itself
            #
            $SHELL $FILE
    else
            #
            # This is a Bourne script, so just in-line it.  This
lets
            # us take advantage of the find_file routine already
in
            # this script.
            #
            . $FILE
    fi
    #
    # If a script was executed, echo it so it is transferred.
```

Figure 37 A Bourne-Shell Custom `def.dir.flp` (Continued)

```
    #
    echo $FILE

} # exec_file


dodir() {

    cd $1
    for i in * .*
    do

    case $i in

    \* | \. | \.\.)
            ;;

    inc.flp)
            exec_file $1/$i
            ;;

    SCCS)
            sccs_list=`echo $i/s.*`
            if [ -d $i -a ! -h $i -a "$sccs_list" != "$i/"'s.*' ]
;\
            then
            for j in $i/s.*
            do
                echo $1/$j
            done
            fi
            ;;

    *)
            if [ -d $i -a ! -h $i ] ; then
                dodir $1/$i
                cd $1
            fi
            ;;
    esac

    done
} #dodir
```

Figure 37 A Bourne-Shell Custom `def.dir.flp` (Continued)

```
if [ -n "$CODEMGR_DIR_FLP" ]; then
    $CODEMGR_DIR_FLP

elif [ -f dir.flp ]; then
    ./dir.flp
else
    PWD=`pwd`
    #
    # The following couple of lines are to translate a
    # CODEMGR_WS_ROOT path from a /net mount to a local
    # directory if it is actually a local disk mount.
    #
    cd $CODEMGR_WS_ROOT
    CODEMGR_WS_ROOT=`/usr/bin/pwd`
    cd $PWD
    #
    # Find and execute all req.flp's above current directory.
    #

    lcd=$PWD
    while [  `expr $lcd : \.\*$CODEMGR_WS_ROOT\$ ` = "0" ]
    do
            if [ -f $lcd/req.flp ]; then
                exec_file $lcd/req.flp
            fi
            lcd=`dirname $lcd`
    done #while

    if [ -f $lcd/req.flp ]; then
            exec_file $lcd/req.flp
    fi

    dodir $PWD

    #
    # If find doesn't find anything it will exit with value
    # 1 and bringover and putback will stop.
    # Always exit 0 here.
    #
    exit 0
fi
```

For a view of this custom `def.dir.flp` in a project setting, see the description of the custom FLP in the TW Topic Chapter , “Managing SunOS Development With TeamWare” on page 34.

The custom FLPs shown in these examples are static descriptions of the dependencies of build targets. They or their secondary files (`inc.fl` in Figure 35, `inc.flp` and `req.flp` in Figure 37) must be edited when the directory structure or build target dependencies change. Nonetheless, they go a long way toward relieving the individual developer of annoying details when auxiliary files must be included in a workspace to enable a build.

When to Use Custom FLPs

Custom FLPs are convenient and simple to use. They are useful when users must remember to transfer special workspace files that are kept outside a specified directory (such as header files, makefiles, and libraries) into a workspace. Because FLPs are executables, they are not limited to statically echoing of a list of file names. For example, an FLP could parse a makefile to identify source files that must be brought into a child workspace for a build.

The versatility of FLPs might tempt a user to use them for tasks for which they are not entirely suited. For example, some builds require the use of shared libraries or standard header files that are not kept in Configuring workspaces. Because workspaces are directories, a user can establish soft links to these files from his development workspace. Or, for performance reasons, he may want to copy the files directly into the workspace. In either event, FLPs should not be used to accomplish these tasks.

Soft links are best described by makefile rules so that `make` can establish them at the beginning of a build. The `make` command can guarantee that linked binaries are up to date, and describing the links in makefiles produces a build structure that works without the presence of Configuring.

FLPs are also not well suited to copying files directly into a workspace when the files are outside of either workspace named in a bringover or putback transaction. Copying such files is best accomplished by wrapping the `bringover` and `putback` commands in executable shell scripts.[1] The wrappers can copy the necessary files before they execute the bringover or putback. The advantage of separating the copying activity from the default FLP is that `def.dir.flp` may be called several times during a bringover or putback, causing wasteful overwrites of copied files.

1. The wrapping technique, which substitutes a shell script for the `bringover` and `putback` commands, works smoothly with the graphical user interface (GUI) because the GUI issues these commands in response to user selections in the GUI.

TW Topic: Configuring and the File System — Outlying Files and Security Issues

The Sun WorkShop TeamWare Configuring tool identifies a workspace as a directory that contains a local subdirectory named `Codemgr_wsdata`. In all other respects, a Configuring workspace is a directory like any other in the file system.

Because a workspace is a directory, it and the files it contains are governed by the same ownership and permissions settings that govern other directories. Therefore, a workspace is subject to the effects of commands from both Configuring and the operating system. While this arrangement provides a high degree of flexibility, it introduces the potential for circumventing the automation and security features built into Configuring.

This topic describes ways you may want to take advantage of the directory nature of workspaces and ways you can protect workspaces from inadvertent file system manipulation.

Copying and Linking Outlying Files

Some builds require the use of files that are not under SCCS control. Such files may be derived files or source files that are not a part of the development project. Examples of such files are shared libraries and standard header files. Because these files are not under SCCS control (or are not in the project's release workspace), they are not copied between workspaces with Configuring transaction commands. Such files are sometimes called *outlying* files.

There are two common ways to incorporate outlying files into your build automatically: with symbolic links or by copying the files directly into a development workspace.

Makefiles

A developer can establish symbolic links to outlying files from his development workspace. Symbolic links are best described by makefile rules so that `make` can establish them at the beginning of a build. The `make` command can guarantee that linked binaries are up to date, and describing the links in makefiles produces a build structure that works without the presence of Configuring. The makefile rule shown in Figure 38 is an example.

```
...
# Ensure that the symbolic link speclib.a
# is made to the file /usr/lib/speclib.a.

speclib.a : /usr/lib/speclib.a
    ln -f -s /usr/lib/speclib.a speclib.a
...
```

To improve performance, a developer may want to copy a file directly into the workspace instead of establishing a symbolic link. Once again, this action can be conveniently performed in a makefile, as shown in the fragment of Figure 39.

```
...
# Ensure that the file /usr/lib/speclib.a
# is copied into the current working directory

speclib.a : /usr/lib/speclib.a
    cp -p /usr/lib/speclib.a speclib.a
...
```

Wrapping Configuring Commands

Another approach is to wrap the `bringover` and `putback` commands in executable shell scripts. The wrappers can copy the necessary files before issuing the bringover or putback, effectively automating the process. Wrappers are typically linked symbolically to the command name (`bringover` or `putback`), and the symbolic link is placed in the execution directory in place of the original executable. The original executable is moved to another directory, from which it is called by the wrapper.

The simple wrapper shown in Figure 40 copies the file `/usr/lib/speclib.a` into the current working directory, then issues the `bringover` command from the `/cmgr` directory.

```
#!/bin/sh -
#
# Name: .wrapper_bringover
#   This script intercepts a bringover command, copies a
#   a file into the current working directory, then issues
#   the bringover.
#
progname=`basename $0`
filename="/usr/lib/speclib.a"

cp -p $filename .
echo "$filename has been copied to the current working
directory."

#  Issue original command, if it can be found.

if [ -f $progname ]
then
        eval exec /cmgr/$progname $opts '"$@"'
else
        echo >&2 "Sorry, /cmgr/$progname was not found."
        exit 1
fi

exit
```

Wrapper scripts can also be used to establish symbolic links in the current working directory. They work equally well from the command line or the graphical user interface (GUI), because the GUI itself performs bringover and putback transactions by issuing commands in a shell.

Security and Access Control

User accessibility to workspaces is controlled by the `Codemgr_wsdata/access_control` file in each workspace. Entries in this file determine which users can initiate bringover and putback transactions, and who can delete, reparent, and move the workspace. Control at this level is adequate to prevent unauthorized access to workspaces by users who execute Configuring commands. Users still have access to workspace directories with operating system commands, however.

For example, a user who possesses the correct passwords and permissions can change directories to a workspace, check out a file from the Source Code Control System (SCCS), edit it, and check it back in. Such an action short-circuits Configuring's file tracking system and is especially worrisome at the release workspace level, where a developer might mistakenly check in a file without the authorization of the release-level integration engineer.

One development group addressed the problem by selectively setting the permissions of the exported directories in its workspaces. Permissions were set as follows:

- Because Configuring transactions must write to a locks file in the `Codemgr_wsdata` directory, that directory was set to be writable by anyone.
- The `usr` directory, which was the root of the source file hierarchy, was exported read-only to all but one file system so that it could not be written to casually.
- The `usr` directory was exported read-write to only one machine, which developers had to log into remotely (`rlogin`) before they could put back. That machine was protected by password from remote logins by unauthorized users.

Of course, no system of controls is really adequate to protect files from authorized users with malicious intent. The real value of the arrangement just described was that qualified users had to perform the extra step of logging into the machine before putting back. This extra step protected from unintentional source file checkins by ensuring a conscious step on the part of developers before they put back source files.

Index

Symbols

A

B

C

D