
Perforce 97.3
Graphical User Interface
User's Manual

Manual 97.3.gg1 (Beta Edition)

November 24, 1997

This manual copyright 1997 PERFORCE Software.
All rights reserved.

PERFORCE software and documentation is available from <http://www.perforce.com/>. You may download and use PERFORCE programs, but you may not sell or redistribute them. You may download and print the documentation, but you may not sell or redistribute it. You may not modify or attempt to reverse engineer the programs.

PERFORCE programs and documents are available from our Web site *as is*. No warranty or support is provided. Warranties and support, along with higher capacity servers, are sold by PERFORCE Software.

PERFORCE Software assumes no responsibility or liability for any errors or inaccuracies that may appear in this book.

By downloading and using our programs and documents you agree to these terms.

PERFORCE and *Inter-File Branching* are trademarks of PERFORCE Software. PERFORCE software includes software developed by the University of California, Berkeley and its contributors.

All other brands or product names are trademarks or registered trademarks of their respective companies or organizations.

The remainder of this column is left blank for your use. You may utilize it to store such written, inscribed, drawn, painted, copied, drafted, or duplicated material as you see fit, or for any other use, providing the use does not violate the laws, ordinances, statutes, regulations, edicts, canons, or decrees of your country, state, territory, kingdom, province, county, city, or other municipality. PERFORCE Software will assume no liability, responsibility, debt, risk or other obligation for any defamatory, libelous, pejorative, unlawful, slanderous, or otherwise illegal material appearing below this paragraph in this column on this page.

Table of Contents

<i>PREFACE</i>	<i>About This Manual</i> 7
	Margin Note Icons 7
	The Example Set 7
	Menu Commands and Shortcut Menus 8
	Please Give Us Feedback 8
<i>CHAPTER 1</i>	<i>PERFORCE Concepts</i> 9
	PERFORCE Architecture 9
	Moving Files Between the Clients and the Server 10
	File Conflicts 10
	Labeling Groups of Files 11
	Branching Files 11
	Job Tracking 11
	Change Review and Daemons 12
	Protections 12
<i>CHAPTER 2</i>	<i>The P4WIN Window</i> 13
	The P4WIN Panes 14
	<i>The Depot Pane</i> 14
	<i>The Object Pane</i> 14
	<i>The Status Pane</i> 15
<i>CHAPTER 3</i>	<i>Connecting to the p4d Server</i> 16
	Verifying the Connection to the p4d Server 16
	Telling P4 Where P4D is 17
<i>CHAPTER 4</i>	<i>Depots, Clients, Files and Changelists: Quick Start</i> 18
	Underlying Concepts 18
	<i>File Configurations Used in the Examples</i> 18

Setting Up a Client Workspace	19
<i>Naming the Client Workspace</i>	19
<i>Describing the Client Workspace to the PERFORCE Server</i>	19
<i>Editing an Existing Client Specification</i>	21
<i>Deleting an Existing Client Specification</i>	21
Copying Files from the Workspace to the Depot	21
<i>Adding Files to the Depot</i>	22
<i>Editing Depot Files</i>	24
<i>Deleting Files From the Depot</i>	24
Retrieving Files from the Depot into a Workspace	24
Reverting Files to their Unopened States	26
Basic Reporting	26

CHAPTER 5

Depots, Clients, and Changelists: More Details **28**

Description of the Client Workspace	28
Mapping the Depot to the Client Workspace	29
<i>Using Views</i>	30
<i>Wildcards in Views</i>	31
<i>Types of Mappings</i>	31
PERFORCE Syntax for File Names Within Views	33
Name and String Limitations	34
<i>File Names</i>	34
<i>Descriptions</i>	34
<i>Depot and Client Names</i>	34
Changelists	34
<i>Creating Numbered Changelists</i>	35
<i>Moving Files between Changelists</i>	35
<i>Automatic Creation of Numbered Changelists</i>	35
<i>Changelists May Be Renumbered upon Submission</i>	36
<i>Deleting Changelists</i>	36
<i>Viewing Submitted Changelists</i>	36
Accessing Older File Revisions	36
File Types	37
Depot Pane Options	38

CHAPTER 6

PERFORCE *Basics:* *Resolving File Conflicts* 40

RCS Format: How PERFORCE Stores File Revisions 40

Only the Differences Between Revisions are Stored 41

Use of 'diff' to Determine File Revision Differences 42

Scheduling Resolves of Conflicting Files 42

Resolving Conflicting Files 43

Interactive File Resolution 43

Automatic File Resolution 48

Resolving Binary Files 48

Locking Files to Minimize File Conflicts 49

Preventing Multiple Resolves with File Locking 49

Resolves and Branching 50

CHAPTER 7

Labels 51

Why Not Just Use Change Numbers? 51

Viewing Labels 52

Creating a Label 52

Adding and Changing

Files Listed in a Label 53

Preventing Accidental Overwrites of
a Label's Contents 54

Retrieving File Revisions from a Label
into a Client Workspace 54

Matching the Client Workspace to the Label 54

*Retrieving a Subset of a Label's File Revisions
Into the Client Workspace* 54

Deleting Labels 55

CHAPTER 8

Branching 56

What is Branching? 56

When to Create a Branch 56

Viewing Branches 57

Branching's First Action:

Creating a Branch 57

Step 1: Create the branch view 58

Step 2: Include the Branched Files in the Client View 59

Step 3:

Use Integrate

*to Create the Target Files
in the Client Workspace* 60

Step 4:

Submit the Changelist

*to Create the Files
in the Depot* 61

Working With Branched Files	61
Branching's Second Action: Propagating Changes from One Codeline to the Other	61
<i>Propagating Changes from Branched Files to the Original Files</i>	62
Deleting Branches	62
How Integrate Works	62
<i>Integrate's Definitions of yours, theirs, and base</i>	62
<i>The Integration Algorithm</i>	63
<i>Integrate's Actions</i>	63
Additional Command-Line Functionality	64

CHAPTER 9

Job Tracking 65

Viewing Jobs	65
Creating and Editing Jobs	66
Linking Jobs to Changelists, and Changing a Job's Status	67
<i>Automatically Performed Functions</i>	67
<i>Controlling Which Jobs Appear in Changelists</i>	68
<i>Manually Associating Jobs with Changelists</i>	68
<i>Arbitrarily Changing a Job's Status</i>	69
Deleting Jobs	69

About This Manual

This is the *PERFORCE 97.3 Graphical User Interface User's Guide*. It is meant for P4WIN users who have never used PERFORCE before; experienced PERFORCE command line users will find the *P4 to P4WIN Translation Guide* to be more useful. This manual teaches the use of PERFORCE's Windows GUI interface, P4WIN; the command line is discussed in this manual only to point out differences between P4 and P4WIN. For information on our command line interface, P4, please see the *Command Line User's Guide*, which is available at our Web site.

Margin Note Icons

This manual makes use of notes in the left margin to supply additional information. The icons accompanying these notes have the following meanings:

p4

Highlights an important difference between P4WIN and P4.



A cross-reference to other material in this manual.



A concrete example of the material discussed.



A note of general interest.



This note is rather important!

The Example Set

We have attempted to develop a uniform example set for use with this manual. All of the examples use the source code for `elm`, a popular UNIX mail program. We selected the `elm` source code for a number of reasons:

- `elm` is widely used, and many PERFORCE users will be familiar with the program. If they are not, they at least understand what it does.

- The source code is stored in well-organized subdirectories, which allow us to demonstrate certain capabilities of PERFORCE.
- The source code for Elm is widely available; users of this manual can download Elm and try the examples as they're encountered.

Links to the Elm source code can be found at

<http://www.myxa.com/elm.html>

We are using the Elm source with the kind permission of Sydney Weinstein and Bill Pemberton of the USENET Community Trust.

<p>Disclaimer: To the best of our knowledge, the Elm team has never used PERFORCE for source management. As far as we know, they never heard of PERFORCE until they received our email asking for permission to use their code in our manual. No implication that the Elm team uses or endorses PERFORCE is intended; none should be inferred.</p>

Menu Commands and Shortcut Menus

In this manual, references to menu commands appear as *Menu>Command*. For example, the **Lock** command in the **File** menu is referred to as **File>Lock**. Submenus are indicated by extending this scheme to multiple levels; for example, the **Sync to Head Revision** submenu of the **Sync/Remove** command of the **File** menu would appear as **File>Sync/Remove>Sync To Head Revision**.

For every menu command, a corresponding shortcut menu command is available. These shortcut menus, which are accessed by right-clicking on objects within windows, are not explicitly mentioned in the manual unless there is some important difference between the menu command and the shortcut menu version of the same command.

Please Give Us Feedback

This is the first release of this particular manual, and we're very interested in receiving opinions on it from our users. In particular, we'd like to hear from users who have never used PERFORCE before. Does this guide teach the topic well? Are there any glaring errors? Are the explanations clear, or are the exemplifications obfuscated by this enchiridion? Please let us know what you think; we can be reached at manual@perforce.com.



You don't need to read this chapter if you don't want to.

All the material discussed here is also covered in the 'how-to' chapters, which comprise the rest of the manual.

This chapter is provided as a guide to what PERFORCE does, without the details of how to do it.

PERFORCE facilitates the sharing of files among multiple users. It is a software configuration management tool, but software configuration management (SCM) is defined in many different ways, depending on who is giving the definition. SCM has been described as providing version control, file sharing, release management, defect tracking, build management, and a few other things. It's worth looking at exactly what PERFORCE does and doesn't do:

- PERFORCE offers version control: multiple revisions of the same file are stored, and older revisions are always accessible.
- PERFORCE provides facilities for concurrent development; multiple users can edit their own copies of the same file.
- Some release management facilities are offered; PERFORCE will track which revisions of which files are part of a particular release.
- Bugs and system improvement requests can be tracked from entry to fix; this is known as defect tracking.
- PERFORCE supplies some lifecycle management functionality; files can be kept in release branches, development branches, or in any sort of needed file set.
- Change review functionality is provided by PERFORCE; this allows users to be notified by email when particular files are changed.
- Although a build management tool is not built into PERFORCE, we do offer a companion freeware product called "JAM - Make(1) Redux". JAM and PERFORCE meet at the file system; source files managed by PERFORCE are easily built by JAM.

PERFORCE excels at all file management functions. Although PERFORCE was built to manage source files, it can manage any sort of on-line documents. It can be used to store revisions of a manual, to manage Web pages, or to store old versions of operating system administration files. Its branching functionality, which allows copies of files to evolve separately from the files they were copied from, is unparalleled in the industry. And PERFORCE is extremely fast.

PERFORCE Architecture

PERFORCE has a client/server architecture, in which many computers, called *clients*, are connected to one central machine, the *server*. Each user works on a client; at their command, files they've been editing are transferred to and from the server. The clients communicate with the server via TCP/IP.

The PERFORCE clients may be distributed around a local area network, wide area network, dialup network, or any combination of these. There can also be PERFORCE clients on the same host as the server.

Three programs do the bulk of PERFORCE's work:

- The P4D program is run on the PERFORCE server. It manages the shared file repository, and keeps track of users, clients, protections, and other PERFORCE metadata.
P4D must be run on a UNIX or Windows/NT host.
- The P4 command-line program is run on PERFORCE clients. It sends the users' requests to the P4D server program for processing, and communicates with p4d via TCP/IP.
P4 client programs can be run on many platforms, including UNIX, Windows, VMS, Macintosh, BeOS, and Next hosts.
- P4WIN, the subject of this manual, is similar to P4, except it is controlled via a graphical user interface rather than by typing commands. It handles most of the tasks that P4 can do, but does not run P4's administrative functions.
P4WIN runs only on Windows 95 and Windows/NT.

Moving Files Between the Clients and the Server

Users create, edit, and delete files in their own directories on the clients; these directories are called *client workspaces*. PERFORCE commands are used to move files to and from a shared file repository on the server known as the *depot*. PERFORCE users can retrieve files from the depot into their own client workspaces, where they can be read, edited, and resubmitted to the depot for other users to access. When a new *revision* of a file is stored in the depot, the old revisions are kept, and are still accessible.



The details of changelists, and basic PERFORCE usage, are discussed in chapters 4 and 5.

Files that have been edited within a client workspace are sent to the depot via a *changelist*, which is a list of files, and instructions that tell the depot what to do with those files. For example, one file might have been changed in the client workspace, another added, and another deleted. These file changes might be sent to the depot in a single changelist, which is processed *atomically*: either all the changes are made to the depot at once, or none of them are. This allows problem fixes that span multiple files to be updated in the depot at exactly the same time.

Each client workspace has its own *client view*, which determines which files in the depot can be accessed by that client workspace. One client workspace might be able to access all the files in the depot; another client workspace might access only a single file. The PERFORCE server is responsible for tracking the state of the client workspace; PERFORCE knows which files a client workspace has, where they are, and which files have write permission turned on.



Resolving file conflicts is the topic of Chapter 6.

File Conflicts

When two users edit the same file, it is possible for their changes to conflict. For example, suppose two users copy the same file from the depot into their workspaces, and each edits his copy of the file in different ways. The first user sends his version of the file back to the

depot; subsequently, the second user tries to do the same thing. If PERFORCE were to unquestioningly accept the second user's file into the depot, the first user's changes would not be included in the latest revision of the file (known as the *head revision*).

When a file conflict is detected, PERFORCE allows the user experiencing the conflict to perform a *resolve* of the conflicting files. The resolve process allows the user to decide what needs to be done: should his file overwrite the other user's? Should his own file be thrown away? Or should the two conflicting files be merged into one? At the user's request, PERFORCE will perform a *three-way merge* between the two conflicting files and the single file that both were based on. This process generates a *merge* file from the conflicting files: the merge file contains all the changes from both conflicting versions, and this file can be edited and then submitted to the depot.

Labeling Groups of Files



Chapter 7 discusses labels.

It is often useful to mark a particular set of file revisions for later access. For examples, the release engineers might want to keep a list of all the file revisions that comprise a particular release of their program. This list of files can be assigned a single mnemonic name, like `release2.0.1`; this name is a *label* for the user-determined list of files. At any subsequent time, the label can be used to copy the old file revisions into a client workspace.

Branching Files



The workings of Inter-File Branching is covered in Chapter 8.

Thus far, it has been assumed that all changes of files happen linearly. But this is not always the case: suppose that one source file needs to evolve in two separate directions; perhaps one set of upcoming changes will allow the program to run under VMS, and another set will make it a Mac program. Clearly, two separately evolving copies of the same files are necessary.

PERFORCE's *Inter-File Branching*TM mechanism allows any set of files to be copied within the depot. By default, the new file set, or *codeline*, evolves separately from the original files, but changes in either codeline can be propagated to the other.

We're particularly proud of PERFORCE's branching mechanism. Most SCM systems allow some form of branching, but PERFORCE's is particularly flexible and elegant.

Job Tracking



You'll learn how to do job tracking in Chapter 9.

Job is a generic term for a plain-text description of some change that needs to be made to the source code. A job might be a bug description, like "the system crashes when I press `return`", or it might be a system improvement request, like "please make the program run faster."

Whereas a job represents work that is intended to be performed, a changelist represents work actually done. PERFORCE's job tracking mechanism allows jobs to be linked to the changelists that implement the work requested by the job. A job can later be looked up to determine if and when it was fixed, which file revisions implemented the fix, and who fixed it.

PERFORCE's job tracking mechanism does not implement all functionality normally supplied by full-scale defect tracking systems. Its simple functionality can be used as is, or it can be integrated with a full-scale job tracking system with a scripting language such as Perl.

Change Review and Daemons

PERFORCE's *change review* mechanism allows users to receive email notifying them when particular files have been updated in the depot. The files that a particular user receives notification on is determined by that user. Change review is implemented by an external Perl program, or *daemon*, and can be recoded by a knowledgeable user, allowing change review functionality to be customized.

P4

Change review administration and protections are both administrative functions, and must be handled from the p4 command line interface. Please see the PERFORCE Command Line User's Guide for more information.

Protections

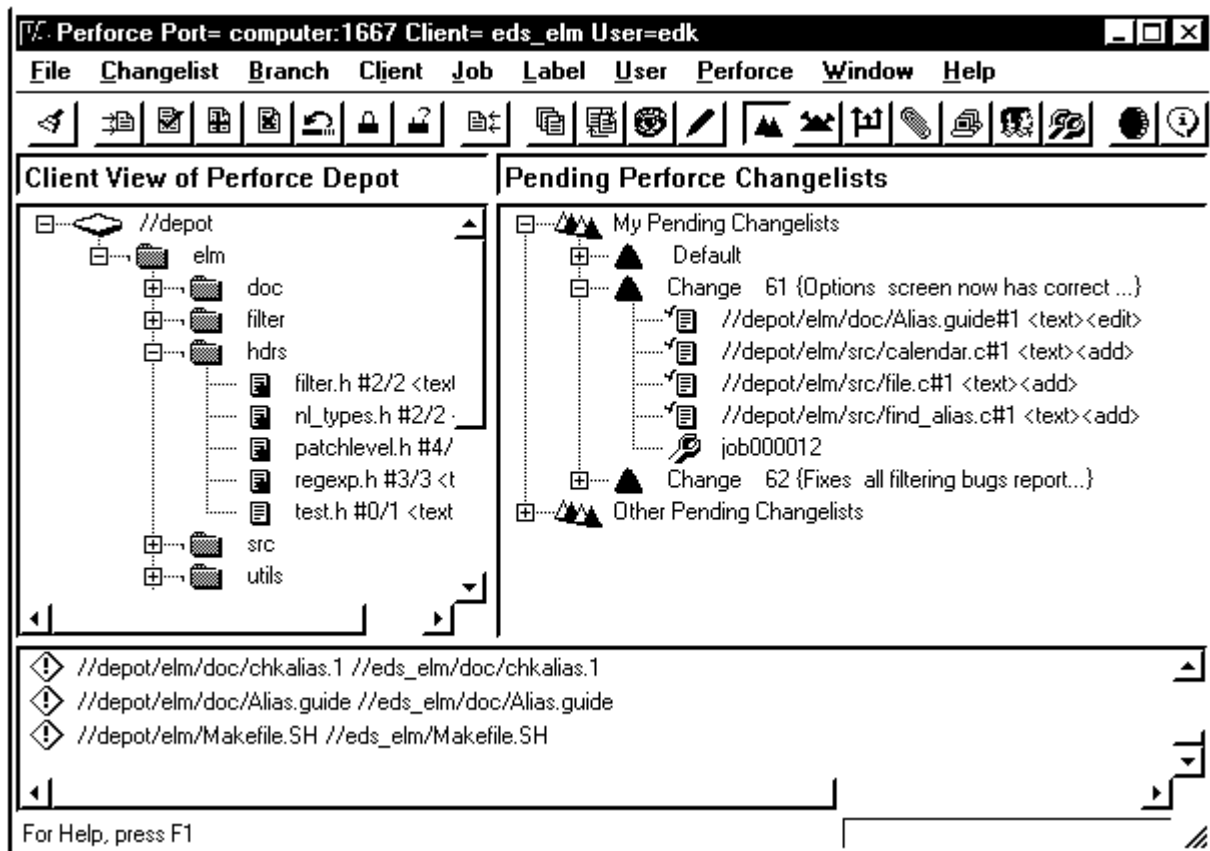
PERFORCE provides a protection scheme to prevent unauthorized or inadvertent access to the depot. The protection mechanism determines exactly which PERFORCE commands are allowed to be run by any particular client.

Permissions are granted or denied based on the user's username and IP address. Since PERFORCE usernames are easily changed, protections at the user level provide safety, not security. Protections at the IP address level are as secure as the host itself.

The P4WIN Window

All work in P4WIN is done in the P4WIN window. This window provides a graphical representation of all files and activities managed by a PERFORCE server.

The p4win window has this appearance:

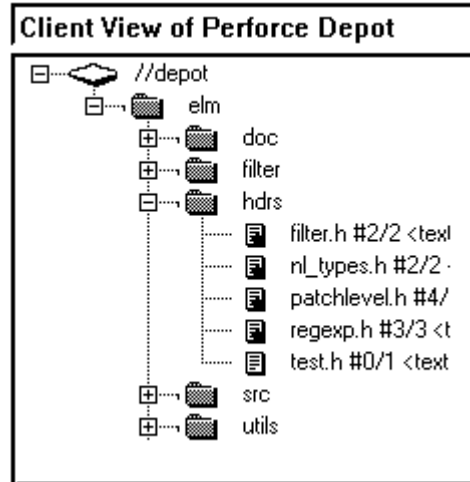


It is divided into three primary panes: two side-by-side graphical panes at the top, and a pane that displays text messages near the bottom. These panes may be resized by dragging the split bars.

The P4WIN Panes

The Depot Pane

The pane at the upper-left of the window is the *depot pane*. It displays files stored in the current P4D server, and has this appearance:



Items in this pane may be manipulated similarly to the way they're used in Windows Explorer: folders may be unfolded by clicking on the ; multiple files and folders can be selected contiguously with the shift key, or non-contiguously by using the control key.

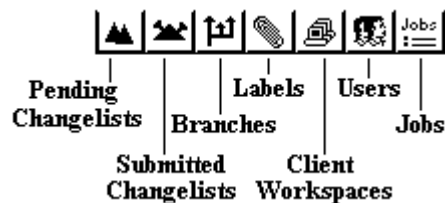
A multiple selection may be dragged, but it differs from Windows Explorer in one key respect: after making the multiple selection, the modifier key must be held down while performing the drag. If it is not, only the last file clicked on will be dragged.

The meaning of the items in the depot pane is explained in the next two chapters.

The Object Pane

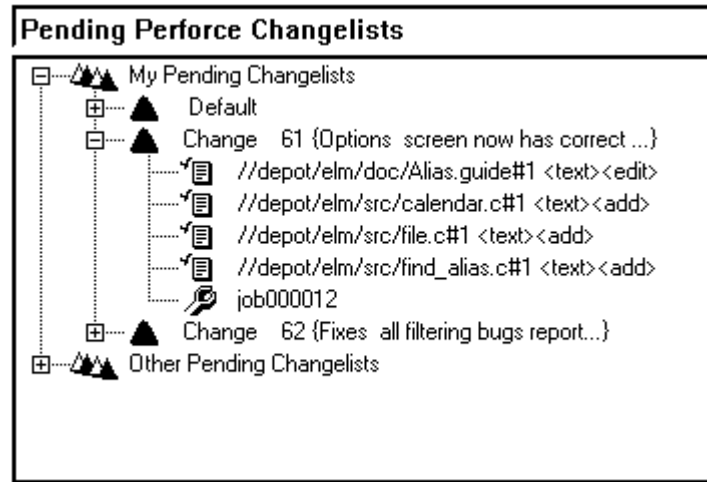
The rightmost pane's display changes depending on what the user has selected to appear. Collectively, the pane is referred to as the *object pane*, but it's usually referred to by its current contents.

The contents of this pane is controlled by seven buttons in the toolbar. The buttons and panes displayed are:



Alternatively, each pane may be displayed by choosing a corresponding menuitem. For example, the branches pane can be displayed either by clicking on the branches button, or by choosing the **Activate Branches Window** command in the **Branch** menu. The remainder of this manual refers only to buttons to change panes, but the corresponding menu item is always available.

These seven panes fall into two categories. The first category consists of one of the above panes, the pending changelists pane. It looks like this:



Although the contents of this pane are somewhat different than those of the depot pane, it follows the same rules as the depot pane and Windows Explorer. The contents of this pane are explained in the next two chapters

The second category of panes is made up by the remaining six panes (the submitted changelists, branches, labels, client workspaces, labels, and jobs panes). These panes display lists of objects. The branches pane, for example, looks like this:

Branch	Date	Description
branch	1997/11/04	Created by barklay.
branch_DOS	1997/11/22	Branch of release 1 code for DOS
release1	1997/11/22	Branch from mainline for first code

Any of the columns can be sorted on by clicking in the header for that column. The current sort column is indicated by a red triangle; the branch listing above is currently sorted on the Branch column.

Columns may be resized by dragging the split bars that separate the columns.

The Status Pane

The pane at the bottom is the status pane. It displays messages from P4WIN and from the P4D server.

Connecting to the p4d Server



This chapter assumes that both the p4d and p4 programs have been installed. xxx”just do it” or cross-ref to instructions?xxx

PERFORCE uses a client/server architecture. Files are created and edited by users on their own client hosts; these files are transferred to and from a shared file repository located on a PERFORCE server. Every running PERFORCE system uses a single server and can have many clients.

Three programs do the bulk of PERFORCE’s work:


- The p4d program is run on the PERFORCE server. It manages the shared file repository, and keeps track of users, clients, protections, and other PERFORCE metadata.
- p4win is run on those Windows PERFORCE clients whose users prefer a graphical user interface. It sends the users’s requests to the p4d server program for processing, and communicates with p4d via TCP/IP.
- P4 has the same core functionality as P4WIN, but is run from the command line, and runs on all PERFORCE platforms. It does everything that P4WIN does, and handles additional tasks, such as administrative jobs, as well.

Each p4win program needs to know the address and port of the p4d server that it communicates with. Setting this address is the topic of the next section

Verifying the Connection to the p4d Server

A p4 client needs to know two things in order to talk to the p4d server:

- The name of the host that p4d is running on
- The port that p4d is listening on.

Together, these make up the P4D server address. This address is set through the **Perforce>Options...** dialog box, which can also be displayed by clicking the  button. The P4D server address is originally set when P4WIN is installed; if it’s not set correctly, you’ll need to change it.



*Example:
Startup error:
the p4d server
connection is
incorrectly specified*

P4WIN looks for the P4D server every time P4WIN starts. If the P4D address is set incorrectly, you'll see a variant of this message:

```
Connect to server failed, check $P4PORT
TCP Connect to host:port failed.
host: host unknown
```

This message means that the server address has not been set correctly in the **Options** dialog.

Telling P4 Where P4D is

Before continuing, you'll need to ask your system administrator the name of the host that P4D is located on, and the number of the TCP/IP port it's listening on. Once you've obtained this information, set the value of the P4 Port text box in the **Options** dialog to *host:port#*, where *host* is the name of the host that p4d is running on, and *port#* is the port that P4D is listening on. For example:

If the p4d host is named...	and the p4d port is named...	set P4 Port to:
dogs	3435	dogs:3435
x.com	1818	x.com:1818

The definition of P4PORT can be shortened if p4win is running on the same host as p4d. In this case, only the p4d port number need be provided to p4. And if p4d is running on a host named or aliased *perforce*, listening on port 1666, the definition of P4PORT for the p4 client can be dispensed with altogether. For example:

If the p4d host is named...	and the p4d port is...	set P4 Port to...
<same host as the p4 client>	9783	9783
perforce	1666	<no value needed>

If the setting of P4 Port is still incorrect, you'll receive the same error message when closing the Options dialog. If the error doesn't appear, then PERFORCE is ready to use.


Depots, Clients, Files and Changelists: Quick Start

This chapter teaches basic P4WIN usage. You'll learn how to move files to and from the common file repository, how to back out of these operations, and some basic P4WIN reporting tools.

These concepts and commands are painted with very broad strokes in this chapter; the details are provided in the next.

Underlying Concepts



All of the tasks discussed in this chapter utilize pending changelist pane at the right side of the main P4WIN window. To display this pane, press the  button in the toolbar.

The basic ideas behind PERFORCE are quite simple: files are created, edited, and deleted in the user's own directories, which are called *client workspaces*. PERFORCE commands are used to move files to and from a shared file repository known as the *depot*. PERFORCE users can retrieve files from the depot into their own client workspaces, where they can be read, edited, and resubmitted to the depot for other users to access. When a new revision of a file is stored in the depot, the old revisions are kept, and are still accessible.

PERFORCE was written to be as unobtrusive as possible; very few changes to your normal work habits are required. Files are still created in your own directories with a standard text editor; P4WIN actions supplement your normal work actions instead of replacing them.

File Configurations Used in the Examples

This manual makes extensive use of examples based on the Elm source code set. The Elm examples used in this manual are set up as follows:

A single depot is used to store the elm files, and perhaps other projects as well. The elm files will be shared by storing them under an `elm` subdirectory within the depot.

Each user will store his or her client workspace Elm files in a different subdirectory. The two users we'll be following most closely, Ed and Lisa, will work with their Elm files in the following locations:

User	Username	Client Workspace Name	Top of own Elm File Tree
Ed	edk	eds_elm	C:\Projects\elm
Lisa	lisag	lisas_ws	C:\Docs\elm



The use of the Elm source code set is described in the About This Manual chapter (page 7).

Setting Up a Client Workspace

To move files between a client workspace and the depot, the PERFORCE server requires two pieces of information, collectively called the *client specification*:


- A name that uniquely identifies the client specification, and
- The top-level directory of the client workspace.

Naming the Client Workspace

To name your client workspace, or to use a different workspace, set the value of the P4 Client text box in the **Options** dialog to the name of the client workspace.



*Example:
Naming the
client workspace*

*Ed is working on the code for Elm. He wants to refer to the collection of files he's working on by the name eds_elm. He presses the **Options** button  and types eds_elm in the P4 Client text box.*

Describing the Client Workspace to the PERFORCE Server

Once the client workspace has been named, it must be identified and described to the PERFORCE server with the **Client>Create/Edit my Client** command. Running this command brings up the client specification dialog box; once the dialog is filled in and closed, the PERFORCE server will be able to move files between the depot and the client workspace.

The **Client** dialog has a number of fields; the two most important are the **Root** and the **View**. The meanings of these fields are as follows:

Field	Meaning
Root:	Identifies the top subdirectory of the client workspace on the local NT or Windows 95 machine. This should be the lowest-level directory that includes all the files and directories that you'll be working with in this workspace.
View:	Describes which files and directories in the depot are available to the client workspace, and where the files in the depot will be located within the client workspace.



Example:
Setting the client
root and the client
view

Ed is working with his elm files in a setting as described above. He's set his client workspace to eds_elm; now he chooses Client>Create/Edit my Client, and sees the following dialog:

Perforce Client Specification		
Client:	eds_elm	Update Client
Owner:	edk	Cancel
Date:	1997/11/17 16:41:56	
Description:	Created by edk.	
Root:	C:\	
Options:	nomodtime noclobber	
View:	//depot/. ... //eds_elm/...	

If he were to leave the form as is, all of the files under Ed's C:\ directory would be mapped to the depot, and they would map to the entire depot, instead of to just the elm project. He changes the values in the Root: and View: fields as follows:

Perforce Client Specification		
Client:	eds_elm	Update Client
Owner:	edk	Cancel
Date:	1997/11/17 16:41:56	
Description:	Created by edk.	
Root:	c:\Projects\elm	
Options:	nomodtime noclobber	
View:	//depot/elm/... //eds_elm/...	

This specifies that C:\Projects\elm is the top level directory of Ed's client workspace, and that the files under this workspace directory are to be mapped to the depot's elm subtree.



*To use PERFORCE properly, it is **crucial** to understand how views work. Views are explained in more detail at the start of the next chapter.*

When Ed's done, he clicks the Update Client button, and the client specification is updated.


The read-only Client: field contains the current client name as defined in the Options dialog. Description: can be filled with anything at all (up to 128 characters); this provides an arbitrary textual description of what's contained in this client workspace. The View: field describes the relationship between files in the depot and files in the client workspace.

Creating a client specification has no immediate visible effect; no files are created when a client specification is created or edited. The client specification simply indicates where files will be located when P4WIN is used to move files between the depot and the client workspace.

Editing an Existing Client Specification

Client>Create/Edit my Client can be used at any time to change the client specification. Just as when a client specification is created, changing a specification has no immediate affect on the locations of any files; the location of files in the depot and workspace is affected only when the client specification is used in subsequent commands. But there is an important distinction between changing the client's root and changing the client's view: if you change the root, PERFORCE assumes that you will manually relocate the files as well. If you change the view and then bring files into the client from the depot, PERFORCE will delete and add files as necessary to make the client workspace reflect the view.

Deleting an Existing Client Specification

An existing client specification can be deleted by displaying the client pane () , selecting a client workspace within that pane, and choosing Client>Delete. Clients can only be deleted if there are no files open for addition, edit, or delete within the client.

Deleting a client specification has no effect on any files in the client workspace or depot; it simply removes the P4D server's record of the mapping between the depot and the files within the client workspace. To delete existing files from a client workspace, select the files to be removed within the depot pane and choose File>Sync/Remove>Remove from Client or throw the files into the Windows Recycle Bin *after* deleting the client specification.



If you're working in an already-established PERFORCE environment, and want to start by retrieving already-existing files, you can skip to page 24 and come back to this section later.

Copying Files from the Workspace to the Depot

Any file in a client workspace can be added to, updated in, or deleted from the depot. This is accomplished in two steps:

1. PERFORCE is told the new state of client workspace files with the commands **File>Add To Source Control** (to add files), **File>Check Out for Edit** (to open files for edit), or **File>Check Out for Delete** (to open files for deletion). When these commands are given, the corresponding files are listed in a PERFORCE *changelist*, which is a list of files and operations on those files to be performed in the depot.
2. The operations are performed on the files in the changelist when the change is selected and the **Changelist>Submit...** command is given.



If a submit of the default changelist fails, that changelist will be assigned a number.

“Changelists” on page 34 discusses the creation and use of numbered changelists.

The commands listed in step one above do not immediately add, edit, or delete files in the depot. Instead, the affected file and the corresponding operation are listed in a *changelist*, and the files in the depot are affected only when this changelist is submitted to the depot with **Changelist>Submit...** This allows a set of files to be updated in the depot all at once: when the changelist is submitted, either all of the files in the changelist are affected, or none of them are.

When a file has been added to a changelist, or checked out for editing or deletion, but the corresponding changelist has not yet been submitted in the depot, the file is said to be *open* in the client workspace.

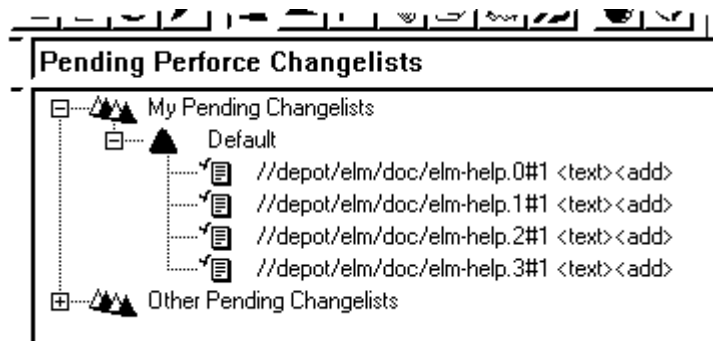
Adding Files to the Depot

To add a file or files to the depot, use **File>Add To Source Control....**, or drag the file(s) or the enclosing folder from the Explorer window to a changelist in the pending changelist pane. These commands opens the file(s) for edit and lists them in a changelist, but they won't be added to the depot until the files in the changelist are submitted to the depot with **Changelist>Submit...**



*Example:
Adding files to a changelist*

Ed is writing a help manual for Elm. The files are named elm-help.0 through elm-help.3, and they're sitting in the doc subdirectory of his client workspace root. He wants to add these files to the depot, so he drags them from the Windows Explorer window to the default changelist in the pending changelist pane. After answering a dialog confirming his choice of files, he sees



At this point, the files that Ed wants to add to the depot have been added to his default changelist. However, the files are not actually added to the depot until the changelist is submitted to the depot.



Example:
Submitting a
changelist to the
depot

Ed is ready to submit his added files to the depot. He selects the default changelist in the rightmost pane and chooses *Changelist>Submit...* The following dialog appears:

Perforce Change Specification		
Change:	<input type="text" value="new"/>	<input type="button" value="Submit Changelist"/>
Client:	<input type="text" value="eds_elm"/>	<input type="button" value="Update Changelist"/>
User:	<input type="text" value="rlo"/>	<input type="button" value="Cancel"/>
Status:	<input type="text" value="new"/>	
Description:	<input type="text" value="<enter description here>"/>	
Jobs:	<input type="checkbox"/> job000006 <input type="checkbox"/> job000005 <input type="checkbox"/> job000004	
Files:	<input checked="" type="checkbox"/> //depot/elm/doc/elm-help.3 <input checked="" type="checkbox"/> //depot/elm/doc/elm-help.2	



Jobs are discussed
in chapter 9.

Ed changes the contents of the *Description:* field to describe what these file updates do. When he's done, he quits from the editor; the new files are added to the depot.

The *Description:* field contents *must* be changed, or the depot update won't be accepted. Files can be unchecked in the *Files:* field; any files deleted from this list will carry over to the next default changelist, and will appear again the next time the default changelist is submitted.

File Permissions

The operating system's *read-only* attribute on submitted files is turned on in the client workspace when the file is submitted to the depot. This helps ensure that file editing is done with PERFORCE's knowledge. The *read-only* attribute is turned off when the file is opened for edit.

You may have noticed that the filenames are always displayed as *filename#n* in the pending changelist pane. PERFORCE always displays filenames within changelists with a *#n* suffix; the *#n* indicates that this is the *n*-th revision of this file. Revision numbers are always assigned sequentially.



Files opened for add are, of course, already in the client workspace; other files must be retrieved into the client workspace before they can be edited. Discussion of this starts on page 24.



If a file is checked out for edit or deletion, and another user already has the file open, a file conflict may occur when the file is submitted. Conflict resolution is discussed in chapter 6.



*Example:
Deleting a file from the depot.*

Editing Depot Files




To open file(s) for edit, select the file(s) in the depot pane and choose **File>Check Out For Edit**. This has two effects:

- The file(s) read-only permissions are turned off on the client workspace's machine, and
- The file(s) to be edited are added to a pending changelist.

Since the files must have their read-only permission turned back on before they can be edited, the edit command must be given before the files are actually edited.

To save the new file revision to the depot, use **Changelist>Submit...**, as above.

*Example: Ed wants to make changes to his elm-help.3 file. He selects the file in the depot pane at the left and chooses **File>Check Out For Edit**. The file appears in the pending changelist pane in his default changelist.*

Files may be opened for edit in a number of ways: files can be dragged from the depot pane to the desired changelist in the pending changelist pane, or they can be selected in the depot pane and then the  button in the toolbar can be clicked. Once a file has been checked out for edit, a red checkmark will appear at the left of the file icon ; if another user already has the file checked out for edit, the file icon will have a blue checkmark at its right ; but the file can still be checked out.



Deleting Files From the Depot


Files are opened for deletion from the depot similarly to the way they are opened for edit: the file(s) are selected in the depot pane, **File>Check Out For Delete** is used to open the files for deletion in the default changelist and to delete the file from the client workspace, and then **Changelist>Submit** is used to delete the file from the depot. In essence, **File>Check Out For Delete** replaces the MS/DOS `del` command for files within a client workspace by allowing the file to be deleted both locally and on the server.

*Ed's file doc/elm-help.3 is no longer needed. He deletes it from the client workspace by selecting the file and choosing **File>Check Out for Edit**. The file is added to his default changelist for deletion; once he submits the changelist with **Changelist>Submit**, the file will be deleted from the depot.*

Once the changelist is submitted, it will appear as if the file has been deleted from the depot; however, old file revisions are never actually removed. This makes it possible to read older revisions of 'deleted' files back into the client workspace.

Retrieving Files from the Depot into a Workspace

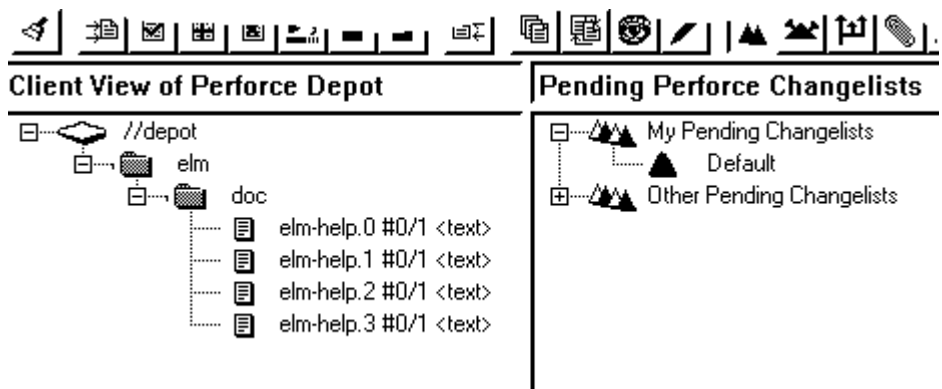
Files from a depot that are not yet in the local client workspace can be recognized by the empty file icon . Once a file has been read from the depot to the client workspace, the file icon will contain the green "synced" dot, and will appear as . Files in the client workspace can be synced with files in the depot by selecting the files in the depot pane and then choosing **File>Sync/Remove/Sync to Head Revision** (alternatively, select the file(s)

and then press ). The *head revision* of a file is the newest version of a file within the depot; as we'll see in chapter 5, it is also possible to retrieve older revisions of a file into a client workspace.

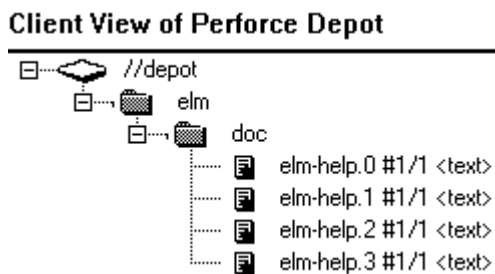
Lisa has been assigned to fix bugs in Ed's code. She creates a directory called `lisas_ws` within her own directory, and sets up a client workspace; now she wants to copy all the existing `elm` files from the depot into her workspace. She sees



*Example:
Retrieving files from
the depot
into the
client workspace.*



None of the four files in the `//depot/elm/doc` directory contain a green dot in their icons; this means that these files were created by another user and have not yet been read into her client workspace. She selects the entire `doc` folder and chooses `File>Sync/Remove>Sync to Head Revision`, and these files are read into her client workspace. Lisa now sees the following in the leftmost pane:




The green dot in the middle of each icon indicates that Lisa now has those files within her client workspace

The **Sync to Head Revision** command maps depot files through the client view, compares the result against the current client contents, and then adds, updates, or deletes files in the client workspace as needed to bring the client contents in sync with the depot.

The job of *sync* is to match the state of the client workspace to that of the depot; thus, if a file has been deleted from the depot, **Sync to Head Revision** will delete it from the client workspace.

We've already mentioned that the green dot in the file icon indicates that the current client workspace has had those files synced; another indicator are the numbers to the right of the file name. The revision specifier `#m/n` means that the depot has *n* revisions of the file, and that the client workspace contains the *m*-th revision of that file.

Reverting Files to their Unopened States

Any file can be removed from a changelist and reverted to its unopened state by selecting the file within the pending changelist pane and then choosing **File>Revert to Saved**, or by selecting the file and then pressing the  button.



Example:
Reverting a file back to the last version gotten.

*Ed wants to edit a set of files in his src directory: leavembox.c, limit.c, and signals.c. He opens the files for edit by selecting them in the depot pane and then choosing **File>Checkout For Edit**, but then realizes that signals.c is not one of the files he will be working on, and that he didn't mean to open it. He can revert signals.c to its unopened state by selecting it in the pending changelist pane and choosing **File>Revert**.*

If a file that had been checked out for deletion is reverted, it will appear back in the client workspace immediately. If the file was originally opened for add, **Revert** will remove it from the changelist but leave the client workspace file intact. If the reverted file was originally checked out for edit, the last synced version will be written back to the client workspace, overwriting the newly-edited version of the file. In this case, you may want to save a copy of the file before reverting it.

Basic Reporting

Reporting commands are those commands that supply information about objects without altering anything within the client workspace or the depot. Two reporting commands are used quite frequently: the first supplies information about the current P4D version and the client workspace, and the second allows viewing of depot file metadata.

To view basic P4D version and client workspace metadata information, choose **Perforce>Info...**, or click the button. A dialog box like the following will appear:

```

User Name=      edk
Client Name=    eds_elm
Client Root=    c:\Projects\elm
Client Address= 206.14.52.212:3249
Client Version= 97.3.0.4723 11/16/97 (beta)
Server Address= computer.perforce.com:1667
Server Version= P4D/FREEBSD/97.3/4586 (11/03/97)
Server License= Perforce Software 100 clients 2000 users
Server Root=    /usr/depot/p4test

Client OS=      Windows NT, ver. 4.0
                Build= 1381, Service Pack 3

```

OK

To view file metadata, select a file in the depot pane and choose **File>Properties...** The following dialog will be displayed:

Depot Path:	//depot/elm/doc/elm-help.0		
Client Path:	c:\Projects\elm\doc\elm-help.0		
File Type:	text		
Head Revision:	1	Have Revision:	1
Head Action:	add	Head Change:	36
Last ModTime:	1997/11/18 21:39:27		
Opened By:	edk@eds_elm		
Locked By:			

OK

The meaning of each field is as follows:

Field	Meaning
Depot Path	The full path of the file within the depot, in relation to the server root.
Client Path	The full path of the file within the client workspace, as mapped through the client view.
File Type	The type of the file (see page “File Types” on page 37 for a full discussion)
Head Revision	The highest-numbered revision of this file within the depot
Head Action	The action associated with the head revision of the file: add, edit, delete, branch, or integrate.
Have Revision	The revision of the file last synced to the client workspace.
Head Change	The number of the changelist that the current head revision of the file was submitted in.
Last Mod Time	The date and time the current head revision of the file was submitted.
Opened By	The usernames and client names of all users who have the file opened for edit, delete, branch, or integrate.
Locked By	The usernames and client names of all users who have manually locked the file.

Depots, Clients, and Changelists: More Details

The *Quick Start* chapter explained the basics of using changelists to transfer files between the client workspace and the depot, but discussion of the practical details were deferred. This chapter, which supplements the *Quick Start*, provides additional information and covers the dry PERFORCE rules. The topics discussed include a detailed description of the client workspace, how to set up views to map the depot to the client workspace, how to access older file revisions, creation and use of numbered changelists, the different PERFORCE-supported file types, and options for displaying the depot pane.

It is assumed that the material in the *Quick Start* chapter has been read and properly digested.

Description of the Client Workspace

A client workspace is a collection of source files managed by PERFORCE on a host. Each such collection is given a name which identifies the client workspace to the PERFORCE server. The name is, by default, simply the host's name, but this can be overridden by changing the value of the P4 Client text box in the **Perforce>Options...** dialog box. There can be more than one PERFORCE client workspace on a client host.

All files within a PERFORCE client workspace share a common root directory, called the *client root*. The client root can be the C:\ directory, but in practice the client root is the lowest level directory under which the managed source files will sit.

PERFORCE manages the files in a client workspace in a few direct ways. It creates, updates, or deletes files when the user requests PERFORCE to synchronize the client workspace with the depot; it turns on write permission when the user requests to edit a file; and turns off write permission and submits updated versions back to the depot when the user is finished editing the file.

The entire PERFORCE client workspace state is tracked by the PERFORCE server. The server knows what files a client workspace has, where they are, and which files have write permission turned on.

PERFORCE's management of a client workspace requires a certain amount of cooperation from the user. Since client files are just plain files with write permission turned off, willful users can circumvent the system by turning on write permission, directly deleting or renaming files, or otherwise modifying the file tree supposedly under PERFORCE's control.

PERFORCE counters this with two measures: first, PERFORCE has explicit commands to verify that the client workspace state is in accord with the server's recording of that state; second, PERFORCE tries to make using PERFORCE at least as easy as circumventing it. For example: to make a temporary modification to a file, it is easier to use PERFORCE than it is to copy and restore the file manually.

Files not managed by PERFORCE may also be under a client's root, and they are largely ignored by PERFORCE. For example, PERFORCE may manage the source files in a client workspace, while the workspace also holds compiled objects, libraries, executables, as well as a developer's temporary files.

In addition to accessing the client files, the p4 client program sometimes creates temporary files on the client host. Otherwise, PERFORCE neither creates nor uses any files on the client host.



*The directory that temporary files are stored in can be set in the Temp Files tab of the **Perforce>Options...** dialog box.*

Mapping the Depot to the Client Workspace

Just as a client name is nothing more than an alias for a particular directory on the client machine, a depot name is an alias for a directory on the PERFORCE server. The relationship between files in the depot and files in the client workspace is described in the *client view*; this is set with **Client>Create/Edit my Client** command. When this command is run, the following dialog appears:

Perforce Client Specification		
Client:	<input type="text" value="eds_elm"/>	<input type="button" value="Update Client"/>
Owner:	<input type="text" value="edk"/>	<input type="button" value="Cancel"/>
Date:	<input type="text" value="1997/11/17 16:41:56"/>	
Description:	<input type="text" value="Created by edk."/>	
Root:	<input type="text" value="c:\Projects\elm"/>	
Options:	<input type="text" value="nomodtime noclobber"/>	
View:	<input type="text" value="//depot/elm/... //eds_elm/..."/>	

The contents of the `view:` field determine where client workspace files get stored in the depot, and where depot files are copied to within the client workspace.

Using Views

Views consist of multiple lines, or *mappings*, and each mapping has two parts. The left-hand side specifies one or more files within the depot, and has the form

```
//depotname/file_specification
```

The right-hand side of each mapping describes one or more files within the client workspace, and has the form

```
//clientname/file_specification
```

The left-hand side of a client view mapping is called the *depot side*; the right-hand side is the *client side*.

The default view in the example above is quite simple: it maps the entire depot to the entire client workspace. But views can contain multiple mappings, and can be much more complex. Any client view, no matter how elaborate, performs the same two functions:

- *The client view determines which files in the depot can be used in a client workspace.* This is determined by the sum of the depot sides of the mappings within a view. A view might allow the client workspace to retrieve every file in the depot, or only those files within two directories, or only a single file.
- *It constructs a one-to-one mapping between files in the depot and files in the client workspace.* Each mapping within a view describes a subset of the complete mapping. The one-to-one mapping might be straightforward; for example, the client workspace file tree might be identical to a portion of the depot's file tree. Or it can be oblique; for example, a file might have one name in the depot and another in the client workspace, or be moved to an entirely different directory in the client workspace. No matter how the files are named, there is always a one-to-one mapping.

To determine the exact location of any client file on the host machine, substitute the value of the clients dialog box's `Root:` field for the client name on the client side of the mapping. For example, if the client dialog box's `Root:` field for the client `eds_elm` is set to `C:\projects\edk\elm`, then the file `//eds_elm/doc/elm-help.1` will be found on the local machine in `C:\projects\edk\elm\doc\elm-help.1`.

Single Workspace, Multiple Drives

An NT client workspace can be spread across multiple drives by setting the client root to `null` and including the drive letter in the client view. For example:

```
Client:      eds_elm
Root:       null
View:
//depot/elm/docs/... //eds_elm/c:/Projects/elm/docs/...
//depot/elm/src/...  //eds_elm/e:/Code/Elm/src/...
```

P4

The P4 command-line interface allows the first two of these wildcards in any command that take file arguments.

Wildcards in Views

PERFORCE uses three wildcards for pattern matching in view specifications; these wildcards can be used in any view specification, such as the dialog that's displayed by **Client>Create/Edit my Client**. Any number and combination of these wildcards can be used in a single string.

Wildcard	Meaning
*	Matches anything except slashes, matches only within a single directory.
...	Matches anything including slashes; matches across multiple directories
%d	Used for parametric substitution; see the subsection "Changing the Order of Filename Substrings" on page 33 for a full explanation.

Any wildcard used on the depot side of a mapping must be matched with an identical wildcard in the mapping's client side. Any string matched by the wildcard will be identical on both sides.

In the client view

```
//depot/elm_proj/... //eds_elm/...
```

the single mapping contains PERFORCE's "..." wildcard, which matches everything, including slashes. The result is that any file in the eds_elm client workspace will be mapped to the same location within the depot's elm_proj file tree. For example, the depot file //depot/elm_proj/nls/gencat/README will be mapped to the client workspace file //eds_elm/nls/gencat/README, which is located on the client host at C:\Projects\elm\nls\gencat\README.

Types of Mappings

By changing the value of the view field, it's possible to map only part of a depot to a client workspace. It's even possible to map files within the same depot directory to different client workspace directories, or to have files named differently in the depot and the client workspace. This section discusses PERFORCE's mapping methods.

Direct Client-to-Depot Views

The default view in the client dialog maps the entire client workspace tree into an identical directory tree in the depot. For example, the default view

```
//depot/... //eds_elm/...
```

indicates that any file in the directory tree under the client workspace eds_elm will be stored in the identical subdirectory in the depot. This view is usually considered to be overkill; most users only need to see a subset of the files in the depot.

Mapping the Full Client to only Part of the Depot

Usually only a portion of the depot is of interest to a particular client. The left-hand side of the view field can be changed to point to only the portion of the depot that's relevant.

Bettie is rewriting the documentation for Elm, which is found in the depot within its elm_proj/doc subdirectory. Her client is named elm_docs, and her client root is C:\usr/bes/docs; she selects Client>Create/Edit my Client and sets the view: field of the dialog as follows:



*Example:
Mapping
part of the depot
to the client
workspace.*

```
//depot/elm_proj/doc/... //elm_docs/...
```

Mapping Files in the Depot to a Different Part of the Client

Views can consist of multiple mappings, which are used to map portions of the depot file tree to different parts of the client file tree. If there is a conflict in the mappings, later mappings have precedence over the earlier ones.



Example:
Multiple mappings in a single client view.

The `elm_proj` subdirectory of the depot contains a directory called `doc`, which has all the Elm documents. Included in this directory are four files named `elm-help.0` through `elm-help.3`. Mike wants to separate these four files from the other documentation files in his client workspace, which is called `mike_elm`.

To do this, he creates a new directory in his client workspace called `help`; it's located at the same level as his `doc` directory. The four `elm-help` files will go here; he fills in the `view` field of the client specification dialog as follows:

```
//depot/... //mike_elm/...
//depot/elm_proj/doc/elm-help.* //mike_elm/help/elm-help.*
```

Any file whose name starts with `elm-help` within the depot's `doc` subdirectory will be caught by the later mapping and appear in Mike's workspace's `help` directory; all other files are caught by the first mapping and will appear in their normal location. Conversely, any files beginning with `elm-help` within Mike's client workspace `help` subdirectory will be mapped to the `doc` subdirectory of the depot.

Excluding Files and Directories from the View

Exclusionary mappings allow files and directories to be excluded from a client workspace; this is accomplished by prefacing the mapping with a minus sign (-). Whitespace is not allowed between the minus sign and the mapping.



Example:
Using views to exclude files from a client workspace

Bill, whose client is named `billm`, wants to view only source code; he's not interested in the documentation files. His client view would look like this:

```
//depot/elm_proj/... //billm/...
-//depot/elm_proj/doc/... //billm/doc/...
```

Since later mappings have precedence over earlier ones, no files from the depot's `doc` subdirectory will ever be copied to Bill's client workspace. Conversely, if Bill does have a `doc` subdirectory in his client, no files from that subdirectory will ever be copied to the depot.

Allowing Filenames in the Client to be Different than Depot Filenames

Mappings can be used to make the names of files different in the client workspace than they are in the depot.



Example:
Files with different names in the depot and client workspace

Mike wants to store the files as above, but he wants to take the `elm-help.X` files in the depot and call them `helpfile.X` in his client workspace. He uses the following mappings:

```
//depot/elm_proj... //mike_elm/...
//depot/elm_proj/doc/elm-help.* //mike_elm/help/helpfile.*
```


Each wildcard on the depot side of a mapping must have a corresponding wildcard on the client side of the same mapping. The wildcards are replaced in the copied-to direction by the substring that the wildcard represents in the copied-from direction.

There can be multiple wildcards; the n -th wildcard in the depot specification corresponds to the n -th wildcard in the client description.

Changing the Order of Filename Substrings

The %d wildcard can be used to rearrange the order of the matched substrings.



Example:
Changing string order in client workspace names

Mike wants to change the names of any files with a dot in them within his doc subdirectory in such a way that the file's suffixes and prefixes are reversed in his client workspace. For example, he'd like to rename the Elm.cover file in the depot cover.Elm in his client workspace. (Mike can be a bit difficult to work with). He uses the following mappings:

```
//depot/elm_proj/...           //mike_elm/...
//depot/elm_proj/doc/%1.%2     //mike_elm/doc/%2.%1
```

Two Mappings Can Conflict and Fail

It is possible for multiple mappings in a single view to lead to a situation in which the name does not map the same way in both directions. When a file doesn't map the same way in both directions, the file is ignored.



Example:
Mappings that fail.

Joe has constructed a view as follows:

```
//depot/elm_proj/...           //joe/elm/...
//depot/nowhere/*             //joe/elm/doc/*
```

The depot file //depot/elm_proj/doc/help would map to //joe/elm/doc/help, but the same file in the client workspace would map back to the depot via the higher-precedence second line to //depot/nowhere/help. Because the file would be written back to a different location in the depot than where it was read from, PERFORCE doesn't map this name at all.

In older versions of PERFORCE, this was often used as a trick to exclude particular files from the client workspace. Because PERFORCE now has exclusionary mappings, this type of mapping is no longer useful, and should be avoided.

P4

In the P4 command-line interface, files provided as arguments to any P4 commands can be specified in one of three syntaxes: local OS syntax, PERFORCE depot syntax, or PERFORCE client syntax. The P4 manual describes all of these.

PERFORCE Syntax for File Names Within Views

File and directory names provided in view specifications are always referred to PERFORCE syntax, which remains the same across operating systems. Filenames specified in this way begin with two slashes and the client or depot name, followed by the path name of the file relative to the client or depot root directory. The components of the path are separated by forward slashes.

Examples of PERFORCE Syntax

```
//depot/...
//eds_elm/docs/help.1
```



Multiple depots can be provided within a single PERFORCE server; these must be set up with P4.

PERFORCE syntax is sometimes called *depot syntax* or *client syntax*, depending on whether the file specifier refers to a file in the depot or on the client. But the syntax is the same in either case.

The specifier `// . . .` is occasionally used; it means ‘all files in all depots’.

Name and String Limitations

File Names

Because of PERFORCE’s naming conventions, certain characters cannot be used in file names. These include unprintable characters, the above wildcards, and the PERFORCE revision characters `@` and `#`. Full file names, which include the entire directory specification, must be 128 characters or less.

Descriptions

Label, branch, user, and client workspace specifications have a silent limit of 128 bytes on descriptions. The description field of a changelist can be any length.

Depot and Client Names

Client names and depot names in a single PERFORCE server share the same namespace, so PERFORCE will never confuse a client name with a depot name. Client workspace names and depot names can never be the same.

Changelists

A PERFORCE *changelist* is a list of files, their revision numbers, and operations to be performed on these files. Commands such as **File>Add to Source Control** and **File>Check Out for Edit** include the affected files in a changelist; the depot is not actually altered until the changelist is submitted with **Changelist>Submit**.

When a changelist is submitted to the depot, the depot is updated *atomically*: either all of the files in the changelist are updated in the depot, or none of them are. This grouping of files as a single unit guarantees that code alterations spanning multiple files will update in the depot simultaneously. To reflect the atomic nature of changelist submissions, submission of a changelist is sometimes called an *atomic change transaction*.

When the command **Changelist>Submit** is given, a dialog is displayed that contains the files in the default changelist. Any file can be removed from this list by unchecking it; when a file is deleted, it is moved to the default changelist. A changelist must contain a user-entered description, which should describe the nature of the changes being made.

When the user quits from the changelist dialog, the changelist is submitted to the server and the server attempts to update the files in the depot. If there are no problems, the changelist is assigned a sequential number, and its status changes from *new* or *pending* to *submitted*. Once a changelist has been submitted, it becomes a permanent part of the depot’s metadata, and is unchangeable except by PERFORCE superusers.

Creating Numbered Changelists

A user can create a changelist in advance of submission with **Changelist>New...** This command brings up the same form seen when a changelist is submitted. All files in the default changelist are moved to this new changelist; when the user quits from the form, the changelist is assigned the next changelist number in sequence, and this changelist must be subsequently referred to by this change number. Files can be deleted from the changelist by editing the form; files deleted from this changelist are moved to the next default changelist. The status for a changelist created by this method is `pending` until the form is submitted.

Multiple changelists are created in order to keep fixes that span multiple files in a single, logical location. For example, a fix to one bug may involve changes to three files, and a new feature may mean changing four other files. The bug fix might be handled in one changelist, and the new feature might be added via another changelist. Each changelist would include only the files that affect that particular change to the system.

Any client file may be included in only one pending changelist.

Moving Files between Changelists

Files may be moved between pending changelists by dragging them from one changelist to the other.

Automatic Creation of Numbered Changelists

Submits of changelists will occasionally fail. This can happen for a number of reasons:

- A file in the changelist has been locked by another user with **File>Lock**;
- The client workspace no longer contains a file included in the changelist;
- There is a server error, such as not enough disk space; or
- The user was not editing the head revision of a particular file. The following sequence shows an example of how this can occur:

Ed checks out file foo for edit and puts it in his default changelist;
 Bettie checks out the same file for edit and puts it in her default changelist;
 Bettie submits his default changelist;
 Ed submits his default changelist.

Ed's submit is rejected, since the file revision of `f00` that he edited is no longer the head revision of that file.

If any file in a changelist is rejected for any reason, the entire changelist is backed out, and none of the files in the changelist are updated in the depot. If the submitted changelist was the default changelist, PERFORCE assigns the changelist the next change number in sequence, an error message will be written to the status pane, and the new numbered changelist will appear in the pending changelist pane.

If the submit failed because the client-owned revision of the file is not the head revision, a *merge* must be performed before the changelist will be accepted.



*Chapter 6
discusses the
merge/resolve
process.*

Changelists May Be Renumbered upon Submission

The change numbers of submitted changelists always reflect the order in which the changelists were submitted. Thus, when a changelist is submitted, it may be renumbered.



*Example:
Automatic
renumbering of
changelists*


Ed has finished fixing the filtering bug that he's been using changelist 29 for. Since he created that changelist, he's since submitted another changelist (change 30), and two other users have submitted changelists. Ed submits change 29, and the following message appears in the status pane:

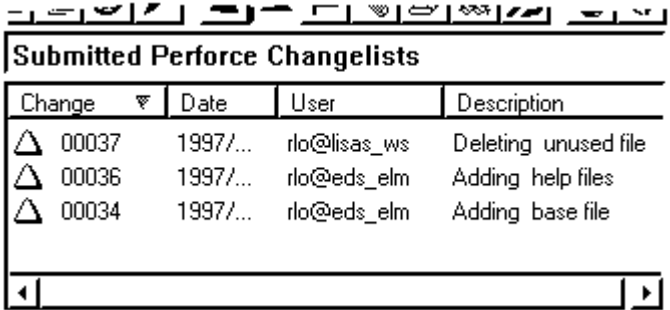
Change 33 submitted

Deleting Changelists

To remove a pending changelist that has no files or jobs associated with it, use **Changelist>Delete**. Pending changelists that contain open files or jobs must have the files and jobs removed from them before they can be deleted: simply drag the files or jobs to another changelist, use **File>Revert** to remove files from the changelist and revert them back to their synced revision, and/or use **Changelist>Remove Job Fix** to remove jobs from the changelist.

Viewing Submitted Changelists

A list of changelists that have been submitted to the depot can be viewed by clicking the submitted changelist pane button  in the toolbar. Four columns are displayed:



Change	Date	User	Description
00037	1997/...	rlo@lissas_ws	Deleting unused file
00036	1997/...	rlo@eds_elm	Adding help files
00034	1997/...	rlo@eds_elm	Adding base file

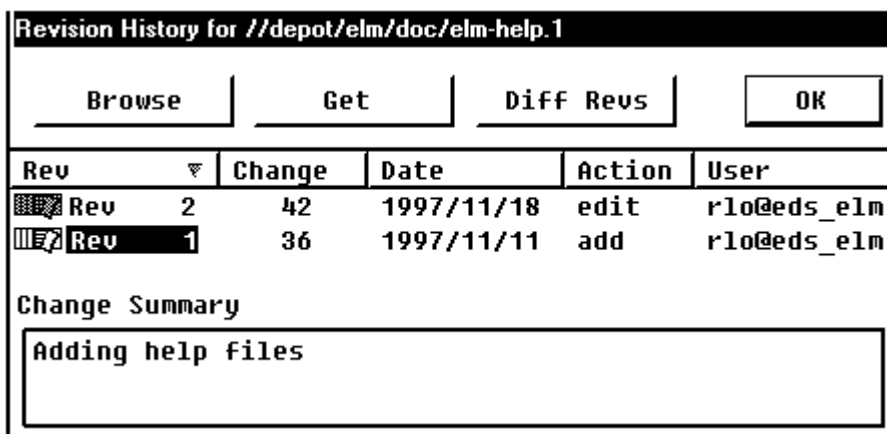
Any of the columns can be used to sort the changelists; click on the column that you want to sort by. The number of submitted changelists retrieved from the server can be set with the **Perforce>Options...** Connection Settings tab.

A list of files and jobs included in any changelist can be viewed by clicking on the changelist and choosing **Changelist>Describe...**

Accessing Older File Revisions

Thus far, we've seen how to sync only the most recent revision (the *head revision*) of a depot file to the client workspace, but PERFORCE allows older file revisions to be brought into the client workspace, and allows other operations on those files as well. Older file

revisions are accessed by selecting a single file within the depot window, and then choosing **File>Revision History...** When this command is run, the following dialog will appear:



For each revision of the file, the revision number is shown, along with the number of the changelist that the revision was submitted in, the date the revision was submitted, the action (add, edit, delete, integrate, or branch) that the file was submitted with, and the username and client workspace name from which the revision was submitted. Any revision can be selected; the description of the changelist that the revision was submitted in will be displayed in the `Change Summary` at the bottom.

Any file revision can be read into the client workspace by selecting the revision and clicking the `Sync` button. A revision can be viewed in an external editor by selecting the revision and pressing the `Browse` button; two revisions can be compared by selecting them both (using the `Control` key) and pressing the `Diff Revs` button.

File Types

PERFORCE supports normal text files as well as binary, “large text” files, keyword text files, Macintosh resource forks, and symbolic links; these file types are described on the next page. PERFORCE attempts to determine the type of the file automatically: when a file is opened for add, PERFORCE first determines if the file is a regular file or a symbolic link, and then examines the first part of the file to determine whether it is text or binary. If any non-text characters are found, the file is assumed to be binary; otherwise, the file is assumed to be text.

The detected file type can be overridden by selecting a file in the pending client pane, choosing **File>Change Type To**, and then selecting the desired file type. The supported file types are described on the next page.

Unless a file’s type is explicitly changed, it will remain the same from revision to revision.

PERFORCE must sometimes store the complete version of every file in the depot, but most often it stores only the changes in the file since the previous revision. This is called *delta storage*, and PERFORCE uses RCS format to store its deltas. The file’s type determines whether *full file* or *delta* storage is used. When delta storage is used, file merges and file



RCS format and delta storage are described in detail at the start of the next chapter.

compares can be performed. Files that are stored in their full form can't be merged or compared.

The PERFORCE file types are:

P4	File Type	Description	Comments	Storage Type	Full support for type in P4WIN?
<i>Files of type resource and symlink have limited support in P4WIN: although files of these types can be synced into the client workspace, and the edited versions of these synced files can be submitted, P4WIN doesn't allow files to change their type to symlink or resource.</i>	text	Text file	Treated as text on the client	delta	yes
	xtext	Executable text file	Like a text file, but execute permission is set on the client file when on UNIX hosts	delta	yes
	binary	Non-text file	Accessed as binary files on the client	full file	yes
	xbinary	Executable binary file	Like a binary file, but execute permission is set on the client file when on UNIX hosts	full file	yes
	ltext	Long text file	This type should be used for generated text files, such as PostScript files.	full file	yes
	symlink	Symbolic link	UNIX clients access these as symbolic links; on non-UNIX clients, these are text files containing symlink target file-names.	delta	no <i>(see note at left)</i>
	ktext	Text file with keyword expansion.	Any inclusion of the literal string <code>\$Id\$</code> within the file will be expanded to reflect the depot file name and revision number.	delta	yes
	kxtext	Executable text file with keyword expansion	Like a ktext file, but execute permission is set on the client file when on UNIX hosts	delta	yes
resource	Macintosh resource fork	Please see the Macintosh client release notes at http://www.perforce.com/perforce/doc/macnotes.txt	full file	no <i>(see note at left)</i>	

The types of existing files are displayed to the right of the file names in the depot pane.

Depot Pane Options

By default, the depot pane displays only those files in the depot that are mapped through the current client view. The entire contents of the depot can be displayed instead; this is controlled by deselecting the **Client View Only** menu item in the **Window** menu.

By default, the server is contacted to update the contents of the depot pane every 30 minutes. The time period between automatic updates can be specified in the `PERFORCE>OPTIONS...` dialog; automatic updating can even be turned off altogether.

Users with large depots should be careful in their use of these two options, since updates of the depot pane of a very large depot may be painfully slow.

PERFORCE Basics: Resolving File Conflicts

File conflicts can occur when two users edit and submit two versions of the same file. Conflicts can occur in a number of ways, but the situation is usually a variant of the following:

- Ed opens file `f00` for edit;
- Lisa opens the same file in her client for edit;
- Ed and Lisa both edit their client workspace versions of `f00`;
- Ed submits a changelist containing `f00`, and the submit succeeds;
- Lisa submits a changelist with her version of `f00`; her submit fails.

If PERFORCE were to accept Lisa's version into the depot, the head revision would contain none of Ed's changes. Instead, the changelist is rejected and a *resolve* must be performed. The resolve process allows a choice to be made: Lisa's version can be submitted in place of Ed's, Lisa's version can be dumped in favor of Ed's, a PERFORCE-generated merged version of both revisions can be submitted, or the PERFORCE-generated merged file can be edited and then submitted.

Resolving a file conflict is a two-step process: first the resolve is *scheduled*, then the resolve is *performed*. A resolve is automatically scheduled when a submit of a changelist fails because of a file conflict; the same resolve can be scheduled manually, without submitting, by syncing the head revision of a file over an opened revision within the client workspace.

PERFORCE also provides facilities for locking files when they are edited. This can eliminate file conflicts entirely.

RCS Format: How PERFORCE Stores File Revisions

PERFORCE uses RCS format to store its text file revisions; binary file revisions are always saved in full. If you already understand what this means, you can skip to the next section of this chapter; the remainder of this section explains how RCS format works.

Only the Differences Between Revisions are Stored

A single file might have hundreds, even thousands, of revisions. Every revision of a particular file must be retrievable, and if each revision was stored in full, disk space problems could occur: one thousand 10KB files, each with a hundred revisions, would use a gigabyte of disk space. The scheme used by most SCM systems, including PERFORCE, is to save only the latest revision of each file, and then store the differences between each file revision and the one previous.

As an example, suppose that a PERFORCE depot has three revisions of file `foo`. The head revision (`foo#3`) looks like this:

```
foo#3:
This is a test
of the
emergency
broadcast system
```

Revision two might be stored as a symbolic version of the following:

```
foo#2:
line 3 was "urgent"
```

And revision 1 would be a representation of this:

```
foo#1:
line 4 was "system"
```

From these partial file descriptions, any file revision can be reconstructed. The reconstructed `foo#1` would read

```
This is a test
of the
urgent
system
```


The *RCS* (Revision Control System) algorithm, developed by Walter Tichy, uses a notation for implementing this system that requires very little storage space and is quite fast. In *RCS* terminology, it is said that the full text of the head revisions are stored, along with the reverse deltas of each previous revision.

It is interesting to note that the full text of the *first* revision could be stored, with the deltas leading forward through the revision history of the file, but *RCS* has chosen the other path: the full text of the head revision of each file is stored, with the deltas leading backwards to the first revision. This is because the head revision is accessed much more frequently than previous file revisions; if the head revision of a file had to be calculated from the deltas each time it was accessed, any SCM utilizing *RCS* format would run much more slowly.

Use of ‘diff’ to Determine File Revision Differences

RCS utilizes the ‘GNU `diff`’ program to determine the differences between two versions of the same file; P4D contains its own *diff* routine which is used by PERFORCE servers to determine file differences when storing deltas. Because PERFORCE’s *diff* always determines file deltas by comparing chunks of text between newline characters, it is by default only used with text files. If a file is binary, each revision is usually stored in full, but binary files can be checked in as text files, insuring that only the deltas are stored.

Scheduling Resolves of Conflicting Files

Whenever a file revision is to be submitted that is not an edit of the file’s current head revision, there will be a file conflict, and this conflict must be resolved. A file that is in conflict will appear in the depot with a yellow explanation point on the icon: .

In slightly more technical terms: we’ll call the file revision that was read into a client workspace the *base file revision*. If the base file revision for a particular file in a client workspace is not the same as the head revision of the same file in the depot, a *resolve* must be performed before the new file revision can be accepted into the depot.

Before resolves can be performed, they must be scheduled. There are two ways of scheduling resolves:

1. The easiest way to schedule a resolve is to submit a changelist that contains the newly conflicting files; if a resolve is necessary, the submit will fail, and the resolve will be scheduled automatically.

If the changelist whose submission failed was the default changelist, it will be assigned a number, and the files from the default changelist will appear in the newly numbered changelist.

2. Resolves of conflicting files can be scheduled by selecting the conflicting files within the depot pane, and chose **File>Sync/Remove>Sync to Head Revision**. Remember that syncing’s job is to project the state of the depot onto the client. Thus, when a sync is performed on a particular file:

- If the file does not exist in the client, or it is found in the client but is unopened, it is copied from the depot to the client.
- If the file has been deleted from the depot, it is deleted from the client.
- If the file has been checked out for edit, the PERFORCE server can’t simply copy the file onto the client: any changes that had been made to the current revision of the file in the client would be overwritten. Instead, a *resolve* is scheduled between the file revision in the depot, the file on the client, and the base file revision (the revision that was last read into the client).



*Example:
Automatic and
manual scheduling
of resolves of
conflicting files.*

*Ed is making a series of changes to the *.guide files in the elm doc subdirectory. He has retrieved the //depot/elm/doc/*.guide files into his client and has checked out the files for edit. He edits the files, but before he has a chance to submit them, Lisa submits new versions of some of the same files to the depot. The versions Ed has been editing are no longer the head revisions; resolves must be scheduled and performed for each of the conflicting files before Ed’s edits can be accepted. Ed submits the changelist containing these files; the submit fails and error messages appear in the status pane. The resolves of the conflicting files are scheduled as part of the submission failure.*

Alternatively, Ed could have selected the `//depot/elm_proj/doc/*.guide_files` in the depot pane and chosen **File>Submit/Remove>Sync to Head Revision**. Since these files are already open in the client, PERFORCE doesn't replace the client files; instead, PERFORCE schedules resolves between the client files and the head revisions in the depot.

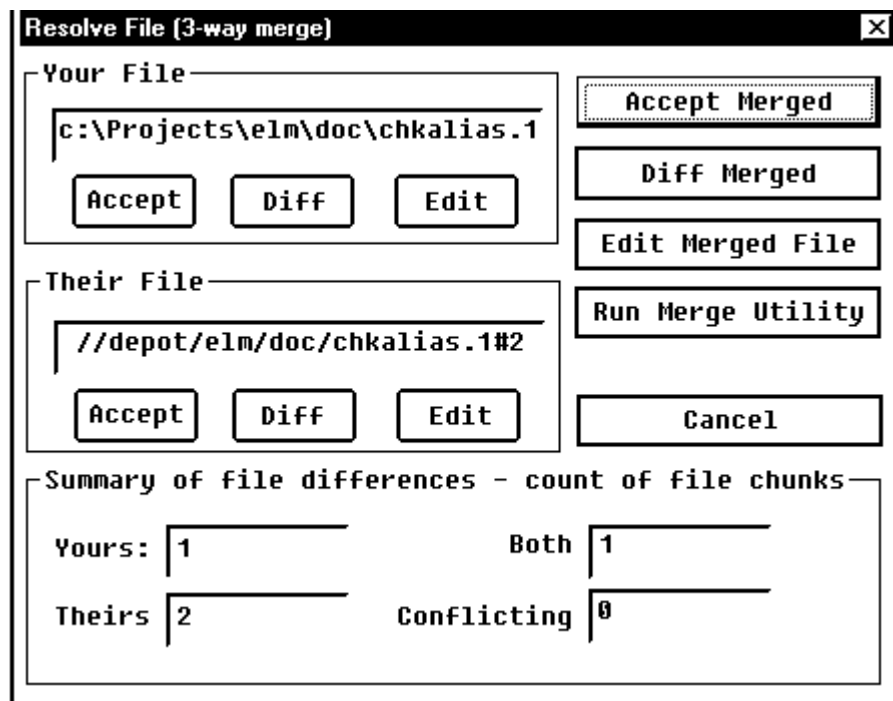
Resolving Conflicting Files

File conflicts can be fixed interactively with **File>Resolve...**, or automatically with **File>Auto-Resolve**. The latter is easier to use than the former, but provides fewer options.

Interactive File Resolution

Any number of conflicting files can be selected in either the depot pane or the pending changelist pane, each file is processed separately when **File>Resolve...** is chosen. The resolve process begins with three revisions of the same file and generates a fourth version; the user can accept any of these revisions to replace the current client workspace file, and can edit the generated version before accepting it. Of course, the new revision is not stored in the depot until it has been submitted in a changelist.

File>Resolve... brings up the file resolution dialog:



The remainder of this section explains what this means, and how to use this dialog.

File Revisions Used and Generated when Resolving

The resolve process begins with three revisions of the same file, generates a new version that merges elements of all three revisions, allows the user to edit the new file, and writes the new file (or any of the original three revisions) to the client. The file revisions used in the resolve process are these:

<i>yours</i>	The newly-edited revision of the file in the client workspace. This file is overwritten by <i>result</i> once the resolve process is complete.
<i>theirs</i>	The revision in the depot that the client revision conflicts with. Usually, this is the head revision, but resolves can be scheduled with any revision between the head revision and <i>base</i> .
<i>base</i>	The file revision in the depot that <i>yours</i> was edited from. Note that <i>base</i> and <i>theirs</i> are different revisions; if they were the same, there would be no reason to perform a resolve.
<i>merged</i>	File variation generated by PERFORCE from <i>theirs</i> , <i>yours</i> , and <i>base</i> .
<i>result</i>	The file resulting from the resolve process. <i>result</i> is written to the client workspace, overwriting <i>yours</i> , and must subsequently be submitted by the user. The instructions given by the user during the resolve process determine exactly what is contained in this file. The user can simply accept <i>theirs</i> , <i>yours</i> , or <i>merge</i> as the result, or can edit <i>theirs</i> , <i>yours</i> , and <i>merge</i> , generating a more reliable result.

The remainder of this chapter will use the terms *theirs*, *yours*, *base*, *merged*, and *result* to refer to the corresponding file revisions. The definitions given above are somewhat different when resolve is used to integrate branched files.



Discussion of resolving branched files begins on page 50.

Types of Conflicts Between File Revisions

The *diff* program that underlies the PERFORCE resolve mechanism determines differences between file revisions on a line-by-line basis. Once these differences are found, they are grouped into *chunks*: for example, three new lines that are adjacent to each other are grouped into a single chunk. *Yours* and *theirs* are both generated by a series of edits to *base*; for each set of lines in *yours*, *theirs*, and *base*, the resolve routine asks the following questions:

- Is this line set the same in *yours*, *theirs*, and *base*?
- Is this line set the same in *theirs* and *base*, but different in *yours*?
- Is this line set the same in *yours* and *base*, but different in *theirs*?
- Is this line set the same in *yours* and *theirs*, but different in *base*?
- Is this line set different in all three files?

Any line sets that are the same in all three files don't need to be resolved. The number of line sets that answer the other four questions are reported at the bottom of the resolve dialog:

Summary of file differences - count of file chunks			
Yours:	1	Both	1
Theirs	2	Conflicting	0

In this case, one line set is identical in *theirs* and *base* but is different in *yours*; two line sets are identical in *yours* and *base* but are different in *theirs*; one line set was changed identically in *yours* and *theirs*; and no line sets are different in *yours*, *theirs*, and *base*.

How the Merge File is Generated

The resolve process generates a preliminary version of the *merged* file, which can be accepted as is, edited and then accepted, or rejected. A simple algorithm is followed to generate this file: any changes found in *yours*, *theirs*, or both *yours* and *theirs* are applied to the *base* file and written to the *merged* file; and any conflicting changes will appear in the merge file in the following format:

```
>>>> ORIGINAL VERSION
(text from the original version)
==== THEIR VERSION
(text from their file)
==== YOUR VERSION
(text from your file)
<<<<
```

Thus, editing the PERFORCE-generated merge file is often as simple as opening the merge file, searching for the difference marker '>>>>', and editing that portion of the text. However, this is not always the case; it's often useful (and necessary) to examine the changes made to *theirs* to make sure they're compatible with other changes that you made.

The Resolve Dialog Options

The **File>Resolve...** command contains the following buttons:

Button Name	What it Does
Accept (under Your File)	Accept <i>yours</i> into the client workspace as the resolved revision, ignoring changes that may have been made in <i>theirs</i> .
Diff (under Your File)	Diff line sets from <i>yours</i> that conflict with <i>base</i>
Edit (under Your File)	Edit the revision of the file currently in the client
Accept (under Their File)	Accept <i>theirs</i> into the client workspace as the resolved revision. The revision that was in the client workspace is trashed.

P4

The P4 command-line version of this command, `p4 resolve`, takes a flag to generate difference markers even when only yours and base differ, or when only theirs and base differ.

Diff (under Their File)	Diff line sets from <i>theirs</i> that conflict with <i>base</i>
View (under Their File)	Edit the revision in the depot that the client revision conflicts with (usually the head revision). This edit is read-only.
Accept Merged	Accept <i>merged</i> into the client workspace as the resolved revision. The version originally in the client workspace is trashed.
Diff Merged	Diff line sets from <i>merge</i> that conflict with <i>base</i> .
Edit Merged File	Edit the preliminary merge file generated by PERFORCE .
Run Merge Utility	Call a third-party merge tool to generate the merged file. This tool must be able to take four file arguments in the order <i>base theirs yours merge</i> To use this option, you must set the environment variable MERGE to the name of a third-party program that merges the first three files and writes the fourth as a result
Cancel	Don't perform the resolve right now. The file remains in conflict.

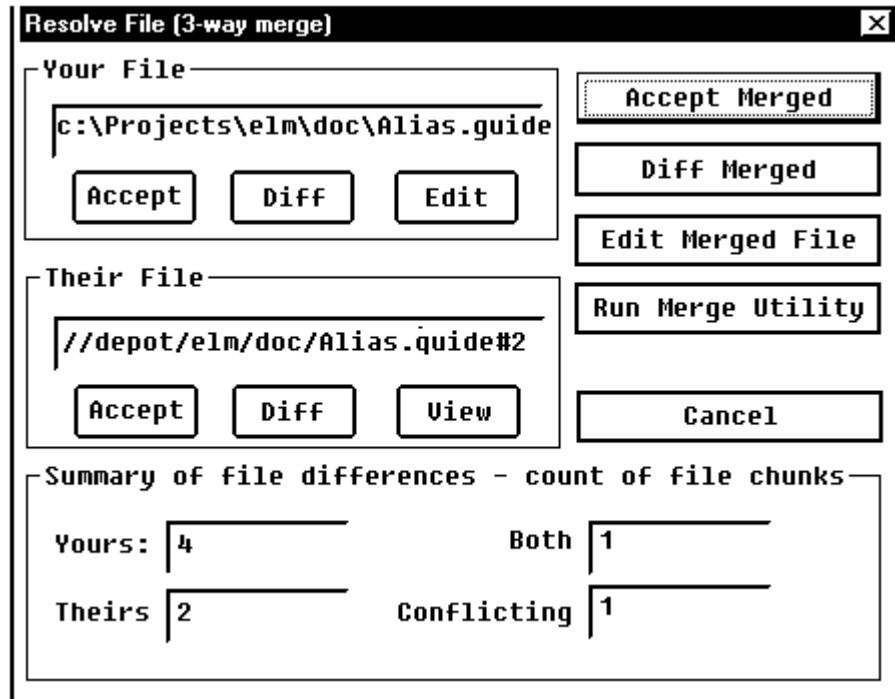
The *merge* file is generated by p4D's internal *diff* routine. But the differences displayed by all the *diff* options above are created by a *diff* routine internal to the P4 client program, and this *diff* can be overridden by specifying an external *diff* in the P4DIFF environment variable.



*Example:
Resolving
file Conflicts*

In the last example, Ed scheduled the doc/.guide files for resolve. This was necessary because both he and Lisa had been editing the same files; Lisa had already submitted versions, and Ed needs to reconcile his changes with Lisa's. To perform the resolves, he*

selects these files in the pending client pane, chooses **File>Resolve...** and sees the following:



This is the resolve dialog for doc/Alias.guide, the first of the files that Ed needs to resolve. Ed sees that he's made four changes to the base file that don't conflict with any of Lisa's changes; he also notes that Lisa has made two changes that he's unaware of. He clicks the Diff button in the Their File pane to view Lisa's diffs; he looks them over and sees that they're fine. Of most concern to him, of course, is the one conflicting change. He chooses Edit Merged File and searches for the difference marker '>>>>'. The following text is displayed:

```
Intuitive Systems
Mountain View, California
>>>> ORIGINAL VERSION
==== THEIR VERSION
98992
==== YOUR VERSION
98993
<<<<
```

He and Lisa have both tried to add a zip code to an address in the file; Ed had typed it wrong. He changes this portion of the file so it reads as follows:

```
Intuitive Systems
Mountain View, California
98992
```

The merge file is now acceptable to him: he's viewed Lisa's changes, seen that they're compatible with his own, and the only line conflict has been resolved. He quits from the editor and chooses Accept Merge; the edited merge file is written to the client, and the resolve dialog is displayed again for the next file that Ed needs to resolve.



File locking is described in “Locking Files to Minimize File Conflicts”, later in this chapter.

When a version of the file is accepted onto the client, the previous client file is overwritten, and the new client file must still be submitted to the depot. Note that it is possible for another user to have submitted yet another revision of the same file to the depot between the time a file is resolved and the time the file’s changelist is submitted; in this case, it would be necessary to perform another resolve. This can be prevented by locking the file before performing the resolve.

Automatic File Resolution

A file can be resolved automatically by PERFORCE by selecting the file and choosing **File>Auto-Resolve**. This command automatically accepts *yours*, *theirs*, or *merged* according to the following criteria:

- If there are no differences between *theirs* and *base*, *yours* is accepted;
- Otherwise, if there are no differences between *yours* and *base*, *theirs* is accepted;
- Otherwise, if there are differences between *yours* and *base*, and between *theirs* and *base*, but there are no conflicts between *yours* and *theirs*, *merged* is accepted;
- Otherwise, there are conflicts between changes made to *yours* and *theirs*, and the resolve is skipped.



Example:
Automatically accepting particular revisions of conflicting files

Ed has been editing the doc/.guide files, and knows that some of them will require resolving. He selects all the doc/*.guide files within his changelist and schedules them for resolve. He then chooses **File>Auto-Resolve...**; the merge files for all scheduled resolves are generated, and those merge files that contain no line set conflicts are written to his client workspace. He’ll still need to manually resolve all the other conflicting files, but the amount of work he needs to do is substantially reduced.*

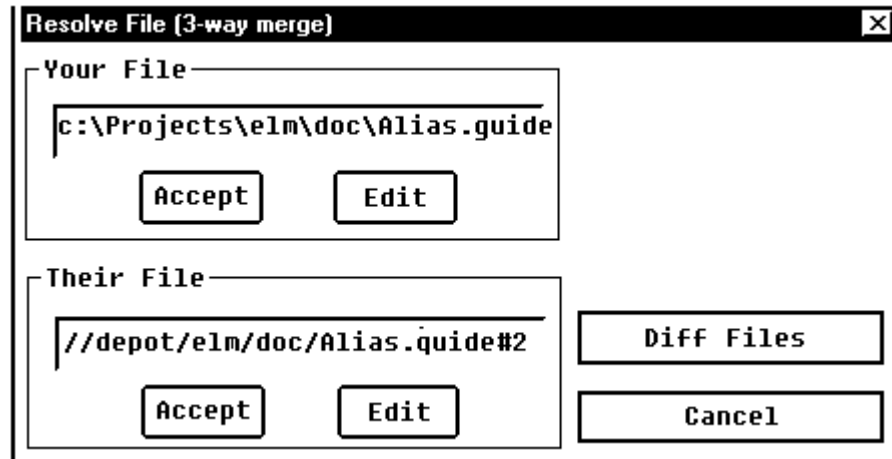
Previewing Automatic File Resolution

The results of automatic file resolution can be previewed with **File>Auto-Resolve (Preview Only)**. This command displays messages in the status pane that inform you what would happen if you chose **File>Auto-Resolve**.

Resolving Binary Files


If any of the three file revisions participating in the merge are binary instead of text, a three-way merge is not possible. Instead, `p4 resolve` performs a two-way merge: the

two conflicting file versions are presented, and you can edit and choose between them. The two-way merge dialog is a very limited version of the three-way merge dialog:



None of the commands that involve *base* or *merged* are available, since these revisions don't exist in a two-way merge.

Locking Files to Minimize File Conflicts

Once open, a file can be locked with **File>Lock** so that only the user who locked the file can submit the next revision of that file to the depot. Once the file is submitted, it is automatically unlocked. Locked files can also be unlocked manually by the locking user **File>Unlock**. A locked file will appear with a lock to its left  in both the depot pane and the pending changelist pane.

The clear benefit of locking a file is that once a file is locked, the user who locked it will experience no further conflicts on that file, and will not need to resolve the file. But this comes at a price: other users will not be able to submit the file until the file is unlocked, and will have to do their own resolves once they submit their revision. Under *most* circumstances, a user who locks a file is essentially saying to other users "I don't want to deal with any resolves; *you* do them." But there is an exception to this rule.

Preventing Multiple Resolves with File Locking

Without file locking, there is no guarantee that the resolve process will ever end. The following scenario demonstrates the problem:

- Ed opens file `f00` for edit;
- Lisa opens the same file in her client for edit;
- Ed and Lisa both edit their client workspace versions of `f00`;
- Ed submits a changelist containing that file, and his submit succeeds;
- Lisa submits a changelist with her version of the file; her submit fails because of file conflicts with the new depot's `f00`;
- Lisa starts a resolve;
- Ed edits and submits a new version of the same file;
- Lisa finishes the resolve and attempts to submit; the submit fails and must now

be merged with Ed's latest file.
<etc...>

File locking can be used in conjunction with resolves to avoid this sort of headache. The sequence would be implemented as follows: before scheduling a resolve, lock the file. Then sync the file, resolve the file, and submit the file. New versions can't be submitted by other users until the resolved file is either submitted or unlocked.

Resolves and Branching

Files in separate codelines can be integrated with **File>Resolve**; discussion of resolving branched files begins in the *Branching* chapter on page 61.

A PERFORCE *label* is simply a user-determined list of files and revisions. The label can later be used to reproduce the state of these files within a client workspace.

Labels provide a method of naming important combinations of file revisions for later reference. For example, the file revisions that comprise a particular release of your software might be given the label `release2.0.1`. At a later time, all the files in that label can be retrieved into a client workspace with a single command.

Create a label when:


- You want to keep track of all the file revisions contained in a particular release of the software;
- There exists a particular set of file revisions that you want to give to other users; or
- You have a set of file revisions that you want to branch from, but you don't want to perform the branch yet. In this case, you would create a label for the file revisions that will form the base of the branch.

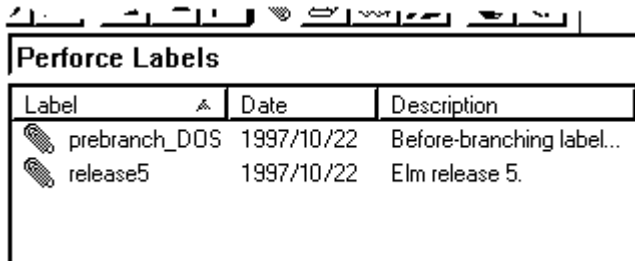
Why Not Just Use Change Numbers?

Labels share certain important characteristics with change numbers: both refer to particular file sets, and both act as handles to refer to all the files in the set. But labels have four important advantages over change numbers:

- the file revisions referenced by a particular label can come from different changelists;
- a change number refers to the state of all the files in the depot at the time the changelist was submitted; a label can refer to any arbitrary set of files and revisions;
- the files and revisions referenced by a label can be arbitrarily changed at any point in the label's existence; and
- changelists are always referred to by PERFORCE-assigned numbers; labels are named by the user.

Viewing Labels

Labels are created and edited within the labels pane. To display the labels pane, click the labels pane selection icon  in the toolbar. The labels pane will appear at the right of the window:



Label	Date	Description
prebranch_DOS	1997/10/22	Before-branching label...
release5	1997/10/22	Elm release 5.

This pane lists every label known to the current P4D server. Any of the three columns may be sorted on by clicking on the column title; more detailed information on any label is available by selecting the label and choosing **Label>Describe**.

Creating a Label

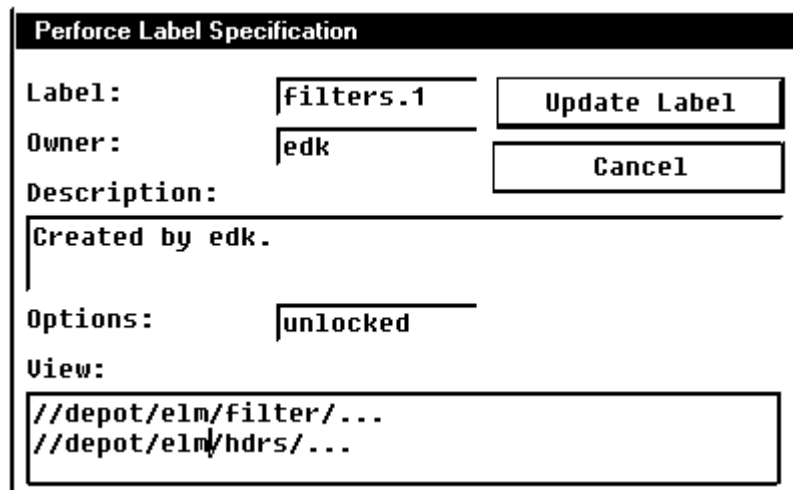
Labels are created with **Label>New...**; this command brings up a dialog similar to the client specification dialog. Like clients, labels have associated views; the label view limits which files can be referenced by the label. Once the label has been created, the **Label>Synchronize Label to Match Client** command is used to load the label with file references.

Label names share the same namespace as clients, branches, and depots; thus, a label name can't be the same as any existing client, branch, or depot name.



Example:
Creating a label

*Ed has finished the first version of filtering in elm; he wants to create a label that references only the head revisions of files in the filter and hdrs subdirectories. He wants to name the label filters.1; he types chooses **Label>New...** and fills in the label dialog as follows:*



Perform Label Specification

Label:

Owner:

Description:

Options:

View:

When he quits from the editor, the label is created.

Before following this example further, it's worth stopping for a moment to examine exactly what has and hasn't been accomplished. So far, a label called `filters.1` has been created. It can contain files only from the depot's `elm/filter` and `elm/hdrs` sub-directories. But the label `filters.1` is empty; it contains no file references. It will be loaded with its file references with **Label>Synchronize>Label to match Client**.

The `View:` field is used to limit the files that are included in the label. These files must be specified by their location in the depot; this view differs from other views in that only the depot side of the view is specified. The `locked/unlocked` option in the `Options:` field can prevent **Label>Synchronize>Label to Match Client** from overwriting previously synced labels (this is described further in "Preventing Accidental Overwrites of a Label's Contents" on page 54).

Adding and Changing Files Listed in a Label

Once a label has been created, references to files can be included in the label by selecting the label in the rightmost pane and choosing **Label>Synchronize>Label to Match Client**. The files that are added to the label will be those in the intersection of the label view and those that were last synced to the client workspace; the revisions in the label will be those last synced to the client workspace.



Example:
Storing
file references
in a label.

Ed has created a label called `filters.1` as specified above; now he wants to load the `filters.1` label with the proper file revisions. The client view of the depot in the leftmost pane shows which files and revisions are in his workspace:

The screenshot shows two panes. The left pane, titled "Client View of Perforce Depot", displays a tree view of the depot structure. The right pane, titled "Perforce Labels", displays a table of labels.

Label	Date	Description
<code>filters.1</code>	1997/11/22	Label to m
<code>prebranch_D...</code>	1997/11/22	Before-bra
<code>release5</code>	1997/11/22	Elm releas

The depot tree view shows the following structure:

- //depot
 - elm
 - doc
 - filter
 - actions.c #1/1 <text>
 - audit.c #7/9 <text>
 - filter.c #2/2 <text>
 - hdrs
 - src
 - utils
 - Changes #1/1 <text>
 - config.h.SH #1/1 <text>

*Ed clicks on the `filters.1` label and chooses **Label>Synchronize>Label to match Client**. The files included in the label are the intersection of those listed in the client view and the label view that the label was defined with; since the label view was defined to include only those files in the `filter` and `hdrs` subdirectories, only those files will be included in the label.*

P4

Although P4WIN requires the entire of contents of the client workspace to be stored in the label, the P4 command-line provides more precision, allowing individual files and revisions to be added to the files already listed in a label. Please see the Labels chapter of the Command-Line User's Guide for more information.

The revisions included in the label will be those last synced to the client workspace; by inspecting the client view of the depot above, we can see that this will include revision 7 of the file `audit.c` and revision 2 of `filter.c`.

Preventing Accidental Overwrites of a Label's Contents

Since **Label>Synchronize>Label to match Client** overwrites all the files that are listed in the label, it is possible to accidentally lose the information that a label is meant to contain. To prevent this, select the label in the labels pane, choose **Label>Edit Specification...** and set the value of the `Options:` field to `locked`. Syncing the label to the client will not be allowed unless the label is subsequently unlocked.

Retrieving File Revisions from a Label into a Client Workspace

Matching the Client Workspace to the Label

To retrieve all the files listed in a label into a client workspace, select the label in the label pane and choose **Label>Sync Client to Label...** This command will *match* the state of the client workspace to the state of the label, rather than simply adding the files to the client workspace. Thus, files in the client workspace that aren't in the label will be deleted from the client workspace.



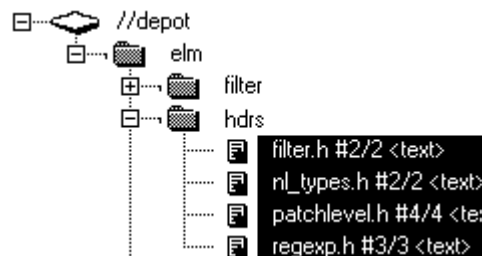
Example:
Retrieving files into a client workspace from a label

Lisa wants to make the state of her client workspace exactly match the files and revisions stored in Ed's `filters.1` label. She selects the `filters.1` label in the labels pane, and chooses **Label>Sync Client to Label...** Files are added to and deleted from her client workspace to make it exactly match the file revision listing in the label.

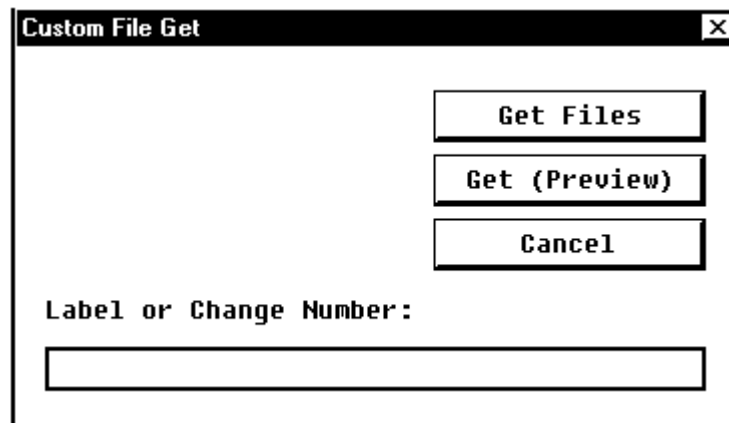
Retrieving a Subset of a Label's File Revisions Into the Client Workspace

To retrieve only a subset of the file revisions listed in a label into a client workspace, select the files in the depot pane and choose **File>Sync/Remove Files>Sync to Label or Change**.

Lisa wants to retrieve only those file revisions in the `//depot/elm/hdrs` subdirectory of the `filter.1` label into her client workspace; she wants to leave the rest of her client workspace intact. Within the depot pane, she shift-selects the files that she wants to sync from the label:



and chooses **File>Sync/Remove Files>Sync to Label or Change**. The following dialog is displayed:



Lisa types the label name `filters.1` and presses the **Get Files** button. Only those files she's selected are synced to the client workspace; the rest of her workspace is left intact.

If Lisa had selected the enclosing folder instead of the files in the folder, all the files in the folder would have been synced to the client workspace, as mapped through the label view. Thus, if the folder had contained files that weren't included in the label, they would have been deleted from Lisa's client view.

Previewing Sync's Results

A sync to the contents of a label can be previewed by selecting files within the depot pane, choosing **File>Sync/Remove Files>Sync to Label or Change** as above, typing in the label name, and pressing **Get (Preview)**. The status pane will display the operations that would occur were the sync to actually be performed.

Deleting Labels

A label can be deleted from the system by selecting it in the label pane and choosing **Label>Delete**.

P4

*It is often useful to view a list of files contained in a label. P4WIN does not currently allow this, but it can be accomplished by running the **PERFORCE** command-line command `p4 files @labelname`.*

Branching

PERFORCE's *Inter-File Branching*TM mechanism allows any set of files to be copied within the depot. By default, the new file set (or *codeline*) evolves separately from the original files, but changes in either codeline can be propagated to the other with P4WIN commands.

What is Branching?

Branching is a method of keeping in sync two or more sets of similar, but not identical, files. Most software configuration management systems have some form of branching; we believe that PERFORCE's mechanism is unique in that it mimics the style in which users create their own file copies when no branching mechanism is available.

Suppose that you're writing a program and are not using an SCM system. You're ready to release your program: what would you do with your code? Chances are that you'd copy all your files to a new location. One of your file sets would become your release codeline, and bug fixes to the release would be made to that file set; your other files would become your development file set, and new functionality to the code would be added to these files.

What would you do when you find a bug that's shared by both file sets? You'd fix it in one file set, and then copy the edits that you made into the other file set.

The only difference between this homegrown method of branching and PERFORCE's branching methodology is that PERFORCE *manages the file copying and edit propagation for you*. In PERFORCE's terminology, copying the files is called *making a branch*; each file set is known as a *codeline*, and copying an edit from one file set to the other is called *integration*. The entire process is called *branching*.

When to Create a Branch


Create a branch whenever two sets of code have different rules governing when code should be submitted, or whenever a set of files needs to evolve along different paths. For example:

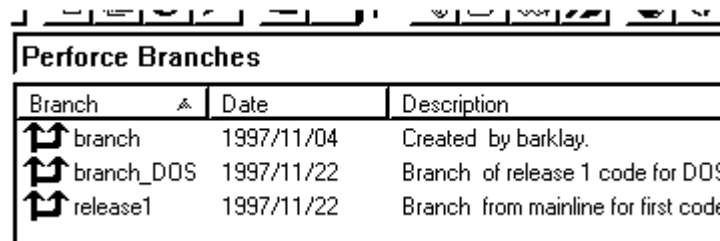
- The members of the development group want to submit code to the depot whenever their code changes, whether or not it compiles; but the release engineers don't want code to be submitted until it's been debugged, verified, and signed off on. They would branch the release codeline from the development codeline; when the development codeline is




ready, it would be integrated into the release codeline. Patches and bug fixes would be made in the release code; later, these changes could be integrated into the development code.

- A company is writing a driver for a new multi-platform printer. They've written a UNIX device driver; they're now going to begin work on a Macintosh driver, using the UNIX code as their starting point. They create a branch from the existing UNIX code; they now have two copies of the same code, and these codelines can evolve separately. If bugs are found in either codeline, bug fixes can be propagated from one codeline to the other with the integrate commands.
- At PERFORCE, we use branching to manage our releases. Development always proceeds in files located within `//depot/main/...`. When a new release is ready, it's branched into another codeline, for example, the code for this release was copied from `//depot/main/...` into `//depot/97.3/...`. Bug fixes that affect both codelines will be made within `//depot/main/...`, and later integrated into the other codeline. Development of release 98.1 will proceed in `//depot/main/...`, when the new release is ready, it will be branched into `//depot/98.1/...`, and the process will continue like this for all PERFORCE releases.

Viewing Branches

Branches are created and edited within the branches pane. To display the branches pane, click the branches pane selection icon  in the toolbar. The branches pane will appear at the right side of the P4WIN window:



Branch	Date	Description
 branch	1997/11/04	Created by barklay.
 branch_DOS	1997/11/22	Branch of release 1 code for DOS
 release1	1997/11/22	Branch from mainline for first code

This pane lists every branch known to the current P4D server. Any of the three columns may be sorted on by clicking on the column title; more detailed information on any branch is available by selecting the branch and choosing **Branch>Describe**.

Branching's First Action: Creating a Branch

As described above, two separate actions comprise branching: first, a branch is created (e.g., files are copied); second, edits are copied from one codeline to the other as needed. This section describes the first of these actions.

The steps to creating a branched codeline are:

1. Create the new branch view with **Branch>New....** Use the view in the dialog box to indicate which files are to be included in the branch, and where the branched codeline will be stored within the depot's file tree.

2. Make sure that the new files and directories are included in the client view of the client workspace that will hold the new files.
3. Use **Branch>Integrate** to open the new files for branching. The new files are listed in a changelist; the associated operation is branch.
4. Submit the changelist that contains the branched file to the PERFORCE server. This creates the new files in the depot.

The following example demonstrates each of these steps.

Step 1: Create the branch view

The first step is to create the branch view. Creating a branch view does four things:

1. Assigns the branched codeline a name;
2. Describes which files will be copied from;
3. For each original file, describes where the new copy will be stored within the depot;
4. Maintains a mapping between each original and branch file, so that changes to one can be easily propagated to the other.



Example:
Creating a branch

A version of Elm is ready for release, and a potential problem is foreseen: the developers will be submitting code to the depot for the next version of Elm, but the release engineers will be submitting fixes to the released version. The two policies are clearly incompatible; so a branched codeline, with duplicate Elm files, needs to be created. Kurt, one of the release engineers, is assigned to create the branch for the release engineers.

*The original code is stored in the depot under its elm subtree; Kurt decides to call the branch elm_rl, and will store the branched codeline in the depot under an elm_release1 subdirectory. He displays the branches in the rightmost pane, chooses **Branch>New...** The following dialog box appears:*

Perforce Branch Specification

Branch:

Owner:

Description:

View:

The default `View` above would map the entire depot to itself in a branch, which is useless. The `View` needs to map the original codeline's files on the left to branch files on the right; Kurt fills in a branch name and changes the `View` field as follows:

The screenshot shows a dialog box titled "Perforce Branch Specification". It has the following fields and controls:

- Branch:** A text input field containing "elm_r1".
- Owner:** A text input field containing "kurtv".
- Description:** A text area containing "Created by kurtv." with up and down arrow icons on the right.
- View:** A text area containing "//depot/elm/... //depot/elm_release1/..." with up and down arrow icons on the right.
- Buttons:** "Update Branch" and "Cancel" buttons are located in the top right corner.

This maps all the files in the depot's `elm` file tree to a new depot file tree called `elm_release1`. All files from the source subtree will be copied to the branch subtree at the end of this process; these files will be the contents of the branch.

Kurt quits the editor; the branch is created.

The new branch command does not copy files into the branch; it simply specifies which original file will correspond to which branched file.

Exclusionary mappings may be used within a branch view.

Step 2: Include the Branched Files in the Client View

In order to work with branched files, the branched files must be accessible through the client view.



*Example:
Including
branched files
in a client view*

*Kurt will be working with the branched files. His client is kurtv_cli; he chooses **Client>Create/Edit my Client**, and adds a line to his client view:*

Perforce Client Specification		
Client:	<input type="text" value="kurtv_cli"/>	<input type="button" value="Update Client"/>
Owner:	<input type="text" value="kurtv"/>	<input type="button" value="Cancel"/>
Date:	<input type="text" value="1997/11/17 16:41:56"/>	
Description:		
<input type="text" value="Created by kurtv"/>		
Root:	<input type="text" value="c:\Docs\Code\Elm"/>	
Options:	<input type="text" value="nomodtime noclobber"/>	
View:		
<input type="text" value="//depot/elm_release1/... //kurtv_cli/elm.r1/..."/>		

There might be other mappings within the client view; the only crucial factor is that the files in the depot's elm branch directory be mapped to some location in Kurt's client workspace. The mapping shown here accomplishes this.

Step 3: Use Integrate to Create the Target Files in the Client Workspace

To create the new branch files in the client workspace, select the branch in the branch pane and choose **Branch>Integrate>Source Line to Branch**. When the branch files don't yet exist in the depot, `integrate` creates the branched files in the client workspace and tells the server that the branch files are to be copied from the original files described in the branch mapping. The `integrate` command, like `add`, `edit`, and `delete`, does not actually affect the depot immediately; instead, it adds the affected files to a changelist which must be submitted.



*Example:
Using integrate
to create
branched files*

*Kurt has created the branch `elm_r1` as above, and he's ready to create the branched copies in the depot. He selects this branch in the branch pane and chooses **Branch>Integrate>Source Line to Branch**. The status pane tells him whether or not the files were copied successfully into the client workspace; all the files that are created in the client workspace are opened in the default changelist.*

Editing Newly Branched Files

By default, a file that has been newly created in a client workspace by the integration command cannot be edited before its first submission. To make a newly-branched file available for editing before submission, simply check out the file for edit.

Step 4: Submit the Changelist to Create the Files in the Depot

The previous step created the files within the client workspace and opened the files within the default changelist. The last step to create branched files is to submit the changelist. This keeps the branching operation atomic: either all the named files are affected at once, or none of them are.

Working With Branched Files

Once a branch has been created and the files have been copied into the branched codeline with the integrate command, the branched files are treated exactly like non-branched files, with the normal use of syncing, checking out for edit, checking out for delete, etc. Evolution of both codelines proceeds separately; additional PERFORCE commands are used only when changes to one codeline need to be propagated to the other.

Branching's Second Action: Propagating Changes from One Codeline to the Other

It is worth repeating that two separate actions comprise branching: first, one set of files is copied from one location in the depot to another location, and second, changes made to one codeline can be copied to the branched codeline *as needed*. The steps needed to accomplish the first action have been described above; now we'll discuss how to accomplish the second action.

Edits to a file in either codeline can be propagated to the corresponding file in the other codeline with the **File>Resolve...** command. Only one additional step needs to be performed: before resolving, the **Branch>Integrate** command is used to schedule the merge between the original files and the branched files.

*A bug has been fixed in the original Elm codeline. Kurt wants to propagate the same bug fix to the branched codeline he's been working on. He selects the `elm_r1` branch in the branch pane and chooses **Branch>Integrate>Source Line to Branch**; the files in the branch are scheduled for resolve. He switches to the changelist pane and selects the files he wants to resolve; the standard merge dialog appears on his screen.*

He resolves the conflicts with the resolution techniques described in chapter 6. When he's done, the result files overwrite the files in his branched client workspace, and they must still be submitted to the depot.

There is one fundamental difference between resolving conflicts in two revisions of the same file, and resolving conflicts between the same file in two different codelines. The difference is that PERFORCE will detect conflicts between two revisions of the same file and then schedule a resolve, but there are *always* differences between two versions of the same file in two different codelines, and these differences usually don't need to be resolved. You must tell PERFORCE that text in one file needs to be propagated to its branch



*Discussion of
file conflict
resolution
begins on page 43.*



*Example:
Propagating
original codeline
changes to the
branched codeline*

by using the integrate command. If the codelines evolve separately, and changes never need to be propagated, you'll never need to integrate or resolve the files in the two codelines.

P4

Access levels must be set through the P4 command line.

Please see the PERFORCE Command Line User's Guide for details.

The integrate command acts only on files that are the intersection of target files in the branch view and the client view. To run the integrate command, `write` access is needed on the target files, and `read` access is required on the donor files.

Propagating Changes from Branched Files to the Original Files

In PERFORCE terminology, changes are always propagated from *donor* files to *target* files. In the above example, the original codeline provided the donor files and the target files were located in the branched codeline, but changes can be propagated in the other direction by using **Branch>Integrate>Branch Back to Source**. When this *reverse integration* command is used to propagate changes from branched donors to original targets, the original source files must be visible through the client view.

Deleting Branches

To delete a branch, use **Branch>Delete**. Deleting a branch deletes only the branch view description, making the branch inaccessible from any subsequent integrate commands. If the files in the branched codeline are to be removed, they must be deleted with **File>Check Out for Delete**.

How Integrate Works

The preceding material in this chapter was written from a user's perspective. This section makes another pass at the same material, this time describing the mechanism behind the integration process.

Integrate's Definitions of yours, theirs, and base

The values of *yours*, *theirs*, and *base* in a three-way merge are quite different when propagating changes between two codelines:

yours The file that changes are being propagated to (also known as the `target` file). This file is in the client workspace, and it is overwritten by the result once the resolve process is complete.

In a forward integrate, this is a file in the branched codeline. In a reverse integration, this is a file in the original codeline.



yours, theirs, and base are first discussed in the File Conflicts chapter on page 44.

<i>theirs</i>	The file revision that changes are being read from (also known as the donor file). This file revision comes from the depot, and is unchanged by the resolve process. In a forward integrate, this is a file revision from the original codeline. In a reverse integration, this is a file in the branched codeline.
<i>base</i>	The last integrated revision of the donor file. When a new branch is created and integrate is used to create the branched copy of the file in the depot, the newly-branched copy is <i>base</i> .

The Integration Algorithm

The integration mechanism performs the following steps:

1. It applies the branch view to all target files to produce a list of donor/target file pairs. It notes individually each revision of each donor file that is to be integrated.
2. It discards any donor/target pairs for which the donor file revisions have been integrated in previous changes. Each revision of each file that has been integrated is remembered individually, in order to avoid making the user merge changes more than once.
3. It discards any donor/target pairs whose donor file revisions have integrations pending in files that are already opened in the client.
4. All remaining donor/target pairs will be integrated. The target file is opened on the client for the appropriate action (see below), and merging is scheduled.

Integrate's Actions

The integrate command will take one of three actions, depending on particular characteristics of the donor and target files:

Action	Meaning
branch	If the target file does not exist, it is opened for <i>branch</i> . The <i>branch</i> action is a variant of <i>add</i> , but PERFORCE keeps a record of which donor file the target file was branched from. This allows three-way merges to be performed between subsequent donor and target revisions with the original donor file revision as <i>base</i> .
integrate	If both the donor and target files exist, the target is opened for <i>integrate</i> , which is a variant of <i>edit</i> . Before a user can submit a file that has been opened for integration, the donor and target must be merged through PERFORCE's resolve process.
delete	When the target file exists but no corresponding donor file is mapped through the branch view, the target is marked for deletion. This is consistent with <i>integrate</i> 's semantics: it attempts to make the target tree reflect the donor tree.

When a forward integration is performed, the original codeline provides the donor files, and the branched codeline provides the targets. When a reverse integration is run, the branched codeline is the donor, and the original files are the targets.

Additional Command-Line Functionality

The P4 command line interface to PERFORCE provides additional branching functionality that is not available in P4WIN. P4 allows integration of a subset of files in a branch, integration of specific file revisions, the re-integration and re-resolving of already integrated code, and merging of two files that were previously not related. For more information, please see the *PERFORCE Command Line User's Manual*.

Job Tracking

A *job* is a written description of some modification to be made to a source code set. A job might be a bug description, like “the system crashes when I press `return`”, or it might be a system improvement request, like “please make the program run faster.”

Whereas a job represents work that is intended, a changelist represents work actually done. PERFORCE’s job tracking mechanism allows jobs to be linked to the changelists that implement the work requested by the job. A job can later be looked up to determine if and when it was fixed, which file revisions implemented the fix, and who fixed it. A job linked to a particular changelist is marked as completed when the changelist is submitted.


Jobs perform no functions internally to PERFORCE; rather, they are provided as a method of keeping track of what changes to the source are needed, which user is responsible for implementing the job, and which file revisions contain the implementation of the job. Since jobs do nothing more than provide this information to the user, the job reporting facilities are particularly important.

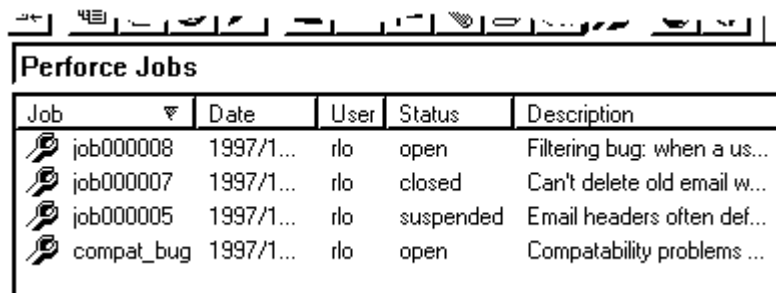
P4





Please see the “P4 User’s Manual” for information about daemons.

The job facilities in PERFORCE do not provide a full-scale job tracking system. They can be used as is, or integrated with another system via a daemon.

Viewing Jobs

Jobs are created and edited within the Jobs pane. To display the jobs pane, click the jobs pane selection icon  in the toolbar. The jobs pane will appear:



Perforce Jobs					
Job	Date	User	Status	Description	
 job000008	1997/1...	rl0	open	Filtering bug: when a us...	
 job000007	1997/1...	rl0	closed	Can't delete old email w...	
 job000005	1997/1...	rl0	suspended	Email headers often def...	
 compat_bug	1997/1...	rl0	open	Compatability problems ...	

This pane lists every job known to the current P4D server. Any of the five columns may be sorted on by clicking on the column title; more detailed information on any job is available by selecting the job and choosing **Job>Describe**.

Creating and Editing Jobs



Example:
Creating a Job

Jobs are created with the **Job>New...** command.

Sarah, who shares the same *PERFORCE* server as Ed, has found a bug in Elm's filtering code. Ed is fixing the code, so Sarah creates a new job and fills in the resulting dialog box as follows:

She has changed `User:` from her username to `edk`. Ed will see this job listed in the changelist dialog the next time he creates a new changelist.

The job dialog box's fields are:

Field Name	Description	Default
Job	The name of the job. Whitespace is not allowed in the name.	new
User	The user whom the job is assigned to, usually the username of the person assigned to fix this particular problem.	<i>PERFORCE</i> username of the person creating the job.
Status	open, closed, suspended, or new. An open job is one that has been created but has not yet been fixed. A closed job is one that has been completed. A suspended job is an open job that is not currently being worked on. New jobs exist only while the change creation form is open.	new; changes to open after job creation form is closed.
Description	Arbitrary text assigned by the user. Usually a written description of the problem that is meant to be fixed.	text that <i>must</i> be changed

The name that appears by default on the form is `new`, but this can be changed by the user to any desired string. If the `Job:` field is left as `new`, or is blank, *PERFORCE* will assign the job the name `jobN`, where `N` is a sequentially-assigned six-digit number.

Existing jobs can be edited with **Job>Edit Specification...** The owner and description can be changed arbitrarily, and the status can be changed to any of the three valid status values open, closed, or suspended.

Linking Jobs to Changelists, and Changing a Job's Status

Automatically Performed Functions

By default, all open jobs owned by a particular user will appear in all PERFORCE changelists subsequently created by that user. A job is automatically closed when one of its associated changelists is successfully submitted. Jobs can be disassociated from changelists by deselecting the job in the changelist's dialog box, and any job of any status may be added to a changelist.



*Example:
Including and
excluding jobs from
changelists*

*Ed is unaware of the job that Sarah has assigned to him. He is currently working on an unrelated problem; he chooses **Changelist>New...** and sees the following:*

Perforce Change Specification

Change: **Update Changelist**

Client: **Cancel**

User:

Status:

Description:

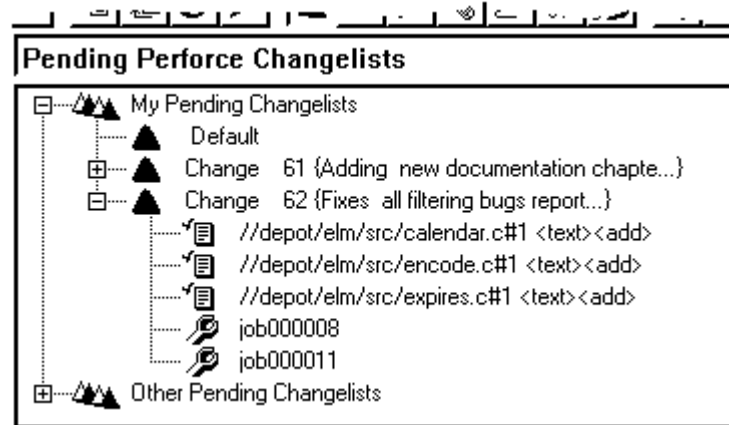
Jobs:
 job000011

Files:
 //depot/elm/src/find_alias.c
 //depot/elm/src/file.c

Since this job is unrelated to the work he's been doing, and since it hasn't been fixed, he leaves the job deselected and closes the dialog box. When the changelist is submitted; the job is not associated with it.

*Ed uses **Job>Describe** to read the job's details. He fixes this problem, and a number of other filtering bugs; when he creates his next changelist, the same job appears in the changelist dialog again and this time, since the job is fixed in this changelist, Ed selects the job. When he submits this changelist, the job is marked as closed, and will not appear in any subsequent changelists unless it is reopened.*

When a job has been linked to a pending changelist, the job will appear in that changelist when the changelist is expanded:



Controlling Which Jobs Appear in Changelists

The types of jobs that appear in new changelists created by a particular user can be controlled through **User>Create/Edit My User**. The dialog box brought up by this command has a `JobView:` field that allows one of three values:

Value of <code>JobView</code> field	Description
Mine	When a new changelist is created, automatically include all open jobs owned by the invoking user in the changelist dialog. This setting of <code>JobView</code> is the default.
None	Don't include any jobs in new changelist dialogs.
All	Include all open jobs owned by all users in all new changelists dialogs.

In all three cases, any unwanted job may be deselected from the form before leaving the editor, and additional jobs can be added.

Manually Associating Jobs with Changelists

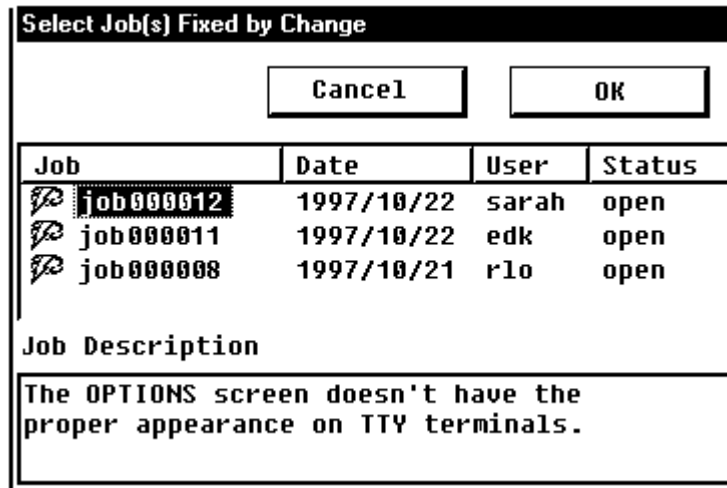
Any open job can be linked to any pending changelist other than the default changelist by selecting the changelist in the pending changelist pane and choosing **Changelist>Add Job Fix**.



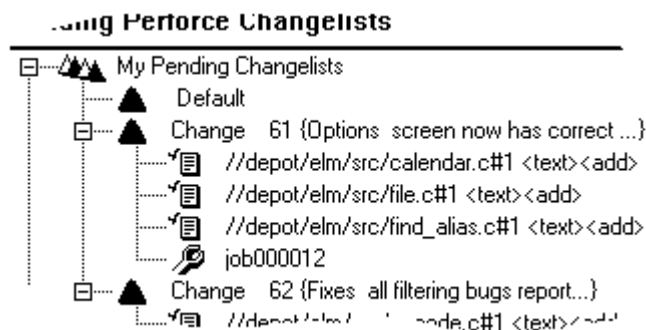
*Example:
Attaching a job
to a changelist*

Sarah has submitted a job called options-bug to Ed. Ed has already created a changelist that fixes this bug, but this changelist has not yet been submitted; Ed selects his

changelist in the pending changelist pane and links the job to his changelist with **Changelist>Add Job Fix**. He sees the following dialog:



The appearance of any job that Ed selects in the `Job` field will appear in the `Job Description` field at the bottom of the dialog. When Ed clicks `OK`, the job he's chosen will be added to the changelist he'd originally selected:



Arbitrarily Changing a Job's Status

We've already seen that a job is automatically closed when an associated changelist is submitted. The status of any job can also be changed by editing the job description with **Job>Edit Specification...** and then changing the status to one of the three allowed values. This is the *only* way of changing a job's status to suspended.

Deleting Jobs

A job that has been linked to a changelist can be unlinked from that changelist by selecting the job within the expanded changelist in the pending changelist pane, and choosing **Changelist>Remove Job Fix**. A job can be completely removed from the system by selecting the job in the jobs pane and choosing **Job>Delete**.