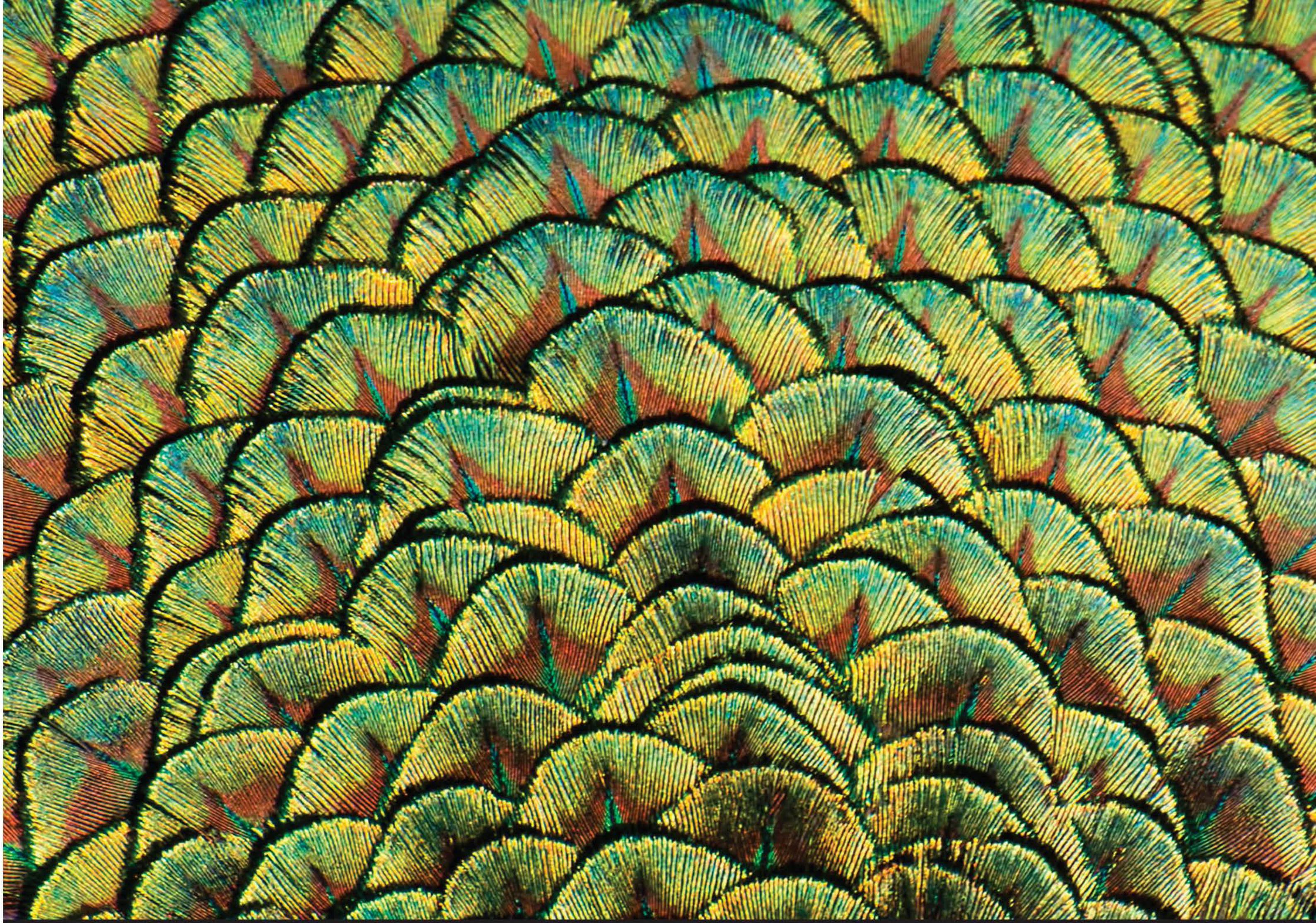


Professional Expertise Distilled

Learning Perforce SCM

A comprehensive guide to the world's leading enterprise configuration management system

Robert Cowham
Neal Ralph Firth

[PACKT] PUBLISHING enterprise
professional expertise distilled

Learning Perforce SCM

A comprehensive guide to the world's leading enterprise configuration management system

Robert Cowham

Neal Ralph Firth

[PACKT] PUBLISHING enterprise

professional expertise distilled

BIRMINGHAM - MUMBAI

Learning Perforce SCM

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: September 2013

Production Reference: 1170913

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-84968-764-5

www.packtpub.com

Cover Image by Jarosław Blaminsky (milak6@wp.pl)

Credits

Authors
Robert Cowham
Neal Ralph Firth

Reviewers
Roel Coucke
Mykhailo Moroz

Acquisition Editor
Edward Gordon

Lead Technical Editor
Madhuja Chaudhari

Technical Editors
Mrunmayee Patil
Sonali Vernekar

Project Coordinator
Apeksha Chitnis

Proofreader
Joel Johnson

Indexer
Rekha Nair

Graphics
Abhinash Sahu

Production Coordinator
Prachali Bhiwandkar

Cover Work
Prachali Bhiwandkar

About the Authors

Robert Cowham is from a software development background with experience in roles from programming to testing and project management. He came across Perforce as a user in the early days of the company when there were only four employees. He subsequently became one of the two pre-qualified Perforce Consulting Partners, and became a Perforce Certified Trainer as soon as that program was implemented. Since then he has consulted for many companies and organizations, and trained thousands of users around the globe, from California to Japan, including giving training courses in German, French, and Italian.

He has also written a number of integrations with Perforce. This includes P4OFC, which integrates with Microsoft Office, and is still provided as an officially-supported public download. APIs developed by him include P4COM, a COM based API for use on Windows, and P4Python which he subsequently handed over to Perforce to support. He has also written various full history migration tools for customers to get them into Perforce.

He has long had an interest in all aspects of configuration management and was for several years a coauthor with Brad Appleton and Steve Berczuk of the Agile SCM column in the online CM Journal. He was Chair of the specialist group for Change, Configuration and Release Management of the British Computer Society for seven years, organizing many events and conferences, and is still active on the committee. He regularly speaks at industry events.

As Services Director for Square Mile Systems, Robert now also works with configuration management of infrastructure and data centers, applying the same principles to the physical world.

Robert has practiced the Japanese martial art of Aikido for over 20 years, and runs a dojo near his home in London. He has even managed to combine his interests with a well-received presentation on "Black belt SCM techniques," including physical demonstrations of the principles involved!

I would like to thank my family for their love and support during the writing of this book. My wife Phoebe covered for the many hours spent closeted away, and my sons Benedict and Dominic also coped admirably with the competition for my attention.

I have worked together with Neal many times over the years. We have different but complementary strengths and writing styles, and it has been a pleasure to work with him on this book.

Neal Ralph Firth assumed both individual contributor and management roles during the design, test, and development of both hardware and software used in stand-alone, embedded system, and real-time environments in his early career. Since the late 90's he has focused on source control and the automation of test and build systems. He first encountered Perforce in its early days while investigating source control solutions for one of his first consulting customers. He subsequently became a Perforce Consulting Partner and a Perforce Certified Trainer as soon as those certifications were available to him. He has consulted for hundreds of companies and organizations, and trained thousands of Perforce users and administrators across the US and Canada.

His career has had a focus on automation and the tools that make people more productive. His early work with hardware microcode simulation was chronicled as part of Tracy Kidder's Pulitzer prize winning book "The Soul of a New Machine". He has presented papers and has spoken at conferences in the US, the United Kingdom, and Germany. He has published articles on hardware, software, and business topics. Hardware topics include peer reviewed IEEE articles on microcode. Software articles have dealt with the application of automated processes with a focus on legacy tool integrations and tool migrations.

As the principal provider of Perforce related services for VIZIM Worldwide, Neal's focus is on the migration of information between SCM products and integrations with legacy systems. He developed the framework for VIZIM's full history migration tool sets and authored the ClearCase and VSS-specific versions of those tools. He has created many Perforce-specific integrations for features such as the Perforce Defect Tracking gateway and IDEs such as JDeveloper.

I would like to thank my family for their support during the writing of this book. My wife Lynn for being a loving supportive partner, my son Eric for reminding me there is life beyond the keyboard, and my daughter Carolyn for reminding me to watch Dr. Who.

Robert and I have worked together many times over the years. Our different yet complementary skills have made for a pleasurable and dynamic interaction while working on this book.

About the Reviewers

Roel Coucke is a 3D generalist, specializing in developing hard surface assets, environments, advanced shaders, and tools for videogames. He graduated at Howest University College in Belgium with a Bachelor's Degree in Game Development, a parallel study in 3D art and programming.

He worked at Guerrilla Games, Amsterdam on Killzone 3, and is an instructor in the Creative Development Series on Digital-Tutors.

I would like to thank the authors of this book with whom I had the honor to collaborate.

Mykhailo Moroz is a software development engineer with seven years of professional experience. During his career he has been working for different projects from very small to very large companies architecting, building, and maintaining test automation frameworks.

His background includes deep knowledge in computer networks, programming, and test automation. Active and everyday usage of tools like Perforce was a prerequisite for successful completion of his projects.

www.PacktPub.com

Support files, eBooks, discount offers, and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read, and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Instant Updates on New Packt Books

Get notified! Find out when new books are published by following `@PacktEnterprise` on Twitter, or the *Packt Enterprise* Facebook page.

Table of Contents

Preface

This book is about using Perforce. There are many tools within the Perforce ecosystem ranging from clients, to plug-ins for **IDEs** (**Integrated Development Environments**), such as Eclipse or Microsoft's Visual Studio.

The focus of this book is the **P4V** client (**Perforce Visual Client**). P4V has many graphical and presentation features that make it an ideal client for new users, and can be used on a variety of different operating systems including Windows, Unix/Linux, and Macs.

What this book covers

Chapter 1, *Getting Started with Perforce*, is an introduction to the technology, terminology, and concepts at the core of Perforce operation. A basic knowledge of these core factors will make it easier to relate your actions to the results you're trying to achieve.

Chapter 2, *The P4V GUI*, introduces the P4V interface. The chapter focuses on the mechanisms for viewing status, performing actions, and viewing the results of those actions.

Chapter 3, *Basic Functions*, covers the techniques for adding, modifying, and deleting files within a workspace. It discusses how to do this as an individual and as a member of a team.

Chapter 4, *Changelists*, describes changelists which are the core record of change within Perforce. The chapter covers how to use changelists for organizing work, communicating with other users, and avoiding problems.

Chapter 5, *File Information*, is the key to understanding the history of a repository, how a code base has evolved, and what is happening to it now. This chapter covers how to make the most efficient use of the Perforce reporting commands to examine the information associated with a file.

Chapter 6, *Managing Workspaces*, describes the Perforce model where client programs work with copies of repository files in local storage areas called workspaces. This chapter covers how to establish and maintain the relationship between server copies of files and workspace copies of those same files.

Chapter 7, *Dealing with Conflicts*, covers how conflict in Perforce refers to the need for a human to resolve issues that may arise from independent modifications of the same repository file. Conflict scenarios are a natural consequence of the flow of development and this chapter addresses the origins of conflicts, and ways of predicting future conflicts, identifying current conflicts, resolving conflicts, and avoiding them.

Chapter 8, *Classic Branching and Merging*, introduces branching as the key version control technology for managing parallel development. This chapter covers the classic Perforce branching interface that provides you with complete control over the entire range of branching features supporting almost any branching pattern that you can envision.

Chapter 9, *Perforce Streams*, describes Perforce streams which use the mainline model to support parallel development. This chapter looks at the concept of branch stability and how streams use the merge-down, copy-up paradigm to support stability.

Chapter 10, *The P4V User Experience*, covers various ways that you can adapt P4V to make your work easier and more productive.

Appendix A, *A Demo Server*, provides a straightforward, step-by-step setup for running examples that align with the contents of this book. Only a limited level of experience is required to follow these steps.

Appendix B, *Command Line,* documents the underlying Perforce interfaces and the common set of commands used. This appendix relates P4V features to the underlying commands that implement them as an aid to understanding the automation associated with builds and other activities.

What you need for this book

This book does not require the P4V software. Many of the concepts and best practices have standalone descriptions.

However, it is best to have an operational copy of P4V available for follow-along exploration. You can download P4V and server software for free from the Perforce website. A license is not required for the examples in the book. *Appendix A*, *A Demo Server* provides straightforward, step-by-step instructions for establishing your own server, a test repository, and the client software.

Who this book is for

This book is intended for Perforce beginners although it includes concepts and information that advanced users will find educational. People in our training courses with years of basic user experience are often pleasantly surprised at the number of things they learn or perhaps realize that they hadn't properly understood previously.

Version control tools such as Perforce are traditionally thought of as the domain of software developers. However, anyone who deals with electronic files or "digital assets" can benefit from version control. Digital assets include source code, XML configuration files, documents, graphic files, web pages, IC test patterns, router configurations or almost anything. This means that Perforce and the concepts in this book apply to a wide range of users such as quality assurance engineers, hardware developers, analysts, documentation writers, artists, and designers.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "We can include other contexts through the use of the `include` directive."

Any command-line input or output is written as follows:

```
Perforce client error:
        Connect to server failed; check $P4PORT.
        TCP connect to 1248 failed.
        connect: 127.0.0.1:1248: WSAECONNREFUSED
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "clicking on the **Next** button moves you to the next screen".

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1 Getting Started with Perforce

This chapter introduces the technology, terminology, and concepts at the core of Perforce operation. A basic knowledge of these core factors will make it easier to relate interface activities to the results you're trying to achieve. These high-level introductions provide the basic framework for their specific application later in this book.

In this chapter we will cover:

- Version control and **Software Configuration Management** (**SCM**)
- The roles of Perforce servers and clients
- What repositories and depots are and how they relate to workspaces
- The core Perforce concept of a changelist
- Practical issues when using Perforce
- Tips for understanding the Perforce mindset

Version control

What is **version control** and what does a version control tool such as Perforce do for us? To many people version control is self-evident. They use it and talk about it without giving it much thought. Yet, like other self-evident activities, a little knowledge and experience can help you avoid problems and become a significantly more successful user.

When most people are asked to define version control they typically start by saying something along the lines of "tracking changes to files". This is followed by a long, disjointed list of features and buzzwords. Others attempt to abstract the details and talk about concepts or tasks. However, we have found that it is usually easier to describe what version control provides by imagining life without a version control tool such as Perforce.

The basic usage of any computer includes sets of files. These files might represent the source code of a program, or a website, or be the test data for an integrated circuit, and many other collections of files. Without version control, you have to manage the files using just the operating system tools. Version control makes the management of these collections both easier and less risky.

If you have a set of files you might want to change them. This might mean adding new files, modifying existing files, and deleting files that are no longer required. But you might also want to track why you made these changes in some form of a change log or even a document or spreadsheet. That introduces potential sources for ambiguity and error as you attempt to transfer information between the different tools. Don't forget the complexities associated with files and directories that are renamed or moved. Version control tracks all of these changes, without error, using a single tool.

At some point you realize that you might want to recover older versions of files in your collection. So you start to make copies of individual files or even the entire collection, for example, when you make a release. Another reason for copying files is when you might want to isolate changes to your files because you're not sure if what you want to do is possible or appropriate. Deciding what to copy, individual files or entire collections, isn't always particularly obvious, especially when you try to account for renames and moves. Version control manages storage space concerns and gives you access to older file versions quickly and easily. It also helps you integrate the changes you made in a copy with the primary versions of the files.

All of this was just considering the needs of a single user. What would happen if more than one user wanted to make changes to the same set of files? How would they avoid or resolve conflicts when working concurrently? How would they share work? Without a version control tool the answers to these questions get complicated very quickly.

So whatever your problem, version control is the solution, well perhaps that's a minor exaggeration, but it is incredibly useful!

Software configuration management

The terms **version control** and **software configuration management** have been in use for many years. Within the software development community they are often used interchangeably. Outside the realm of software development people often refer to SCM as a specifically software related activity. All of which can lead to reader confusion.

To reduce this, we avoid using the phrase SCM in this book. When necessary, the phrase version control will refer to Perforce features dealing with the versions of a single file; and the phrase **configuration management** will refer to Perforce features dealing with a collection of files.

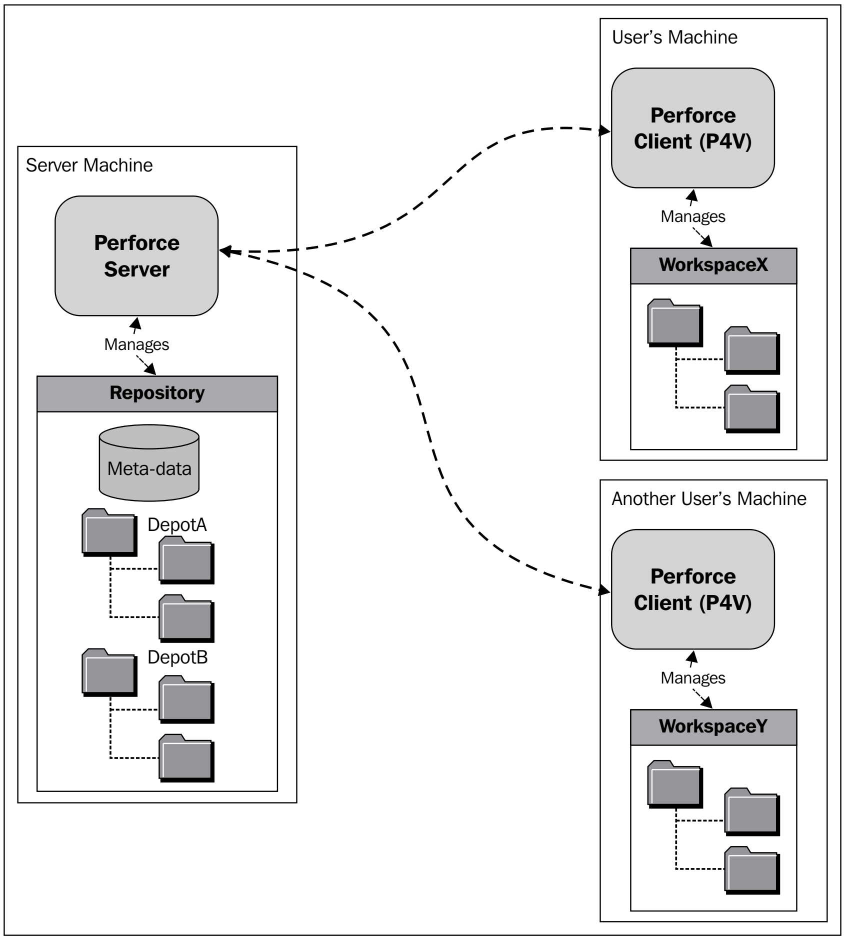
Centralized and distributed version control

Centralized version control and **distributed version control** are phrases commonly used to differentiate version control systems. Centralized systems (including Subversion, Perforce, and many other commercial tools) have a central, shared master repository of version information. Distributed systems (such as Git and Mercurial) provide each individual user with their own private copy of a repository.

Each technology has strengths and weaknesses relating to performance, communication, and security. Centralized systems often provide a way to operate even when they cannot access the central repository, and distributed systems still need a master repository if more than one user is doing work on the same code base. Perforce further blurs this distinction by being a centralized system with full support for distributed users. It now includes a bidirectional interface called Git Fusion in its recent versions. This allows developers to use Git if they wish, and then use Perforce as the master repository for sharing with other people and the rest of the organization.

Understanding Perforce clients and servers

The basic Perforce architecture is client and server. There is a single server program (**P4D**) managing a **repository** and typically many client programs (such as **P4V** or **P4**) which communicate with that server as shown in the following diagram:



The server is responsible for maintaining the history of the contents of files, and for all of the information that describes the events that created that content (the metadata). Each client program is responsible for managing files and their contents within a particular workspace, and for communicating status changes to the server. In order to manage files a client program must connect to a server. Connecting to a server is covered in *Chapter 2*, *The P4V GUI*.

Servers

Users and client programs have no direct access to the repository managed by the server, so it is best to view the server as a black box. It is still useful to understand some of the services that the server provides.

The obvious services are to track the historic content of all the files managed by the server. The most visible part of this tracking is the actual content of all the versions of every file. An equally important part of this tracking is the metadata that the server maintains about file versions. Metadata describes details such as user, workspace, time, date, and check in comments, these are all part of the context in which the historic content was created.

Another service provided by the server is to track the state of files within each workspace, which it does in coordination with the client program. The state includes things such as whether files are present in the workspace, which version is present, whether the user has checked out particular files or versions, and so on.

The most important service provided by the server is to control and coordinate change, so having the server store information, rather than the client program, makes it available to everyone. This supports good engineering practice in the modern dynamic development environment. For example, when you check out a file the server tells you if the same file is checked out in other workspaces. A file checked out in multiple workspaces at the same time may or may not be a problem. Only a human can make that decision.

Client programs

Client programs manage workspaces. At any one time, a client program manages a single workspace. It is a straightforward and fast operation for a client program to manage another workspace as necessary. It is common for users to have multiple workspaces on their local workstations such as their PC or laptop.

> **Terminology warning – the word "client" has three meanings!**
>
> The word client is used in three different ways when talking about Perforce: to mean a client program, a client machine, or a client workspace. The first two meanings are fairly standard, but you will see the "client workspace" meaning in some of the Perforce documentation and in warning or error messages within P4V. The context usually makes it clear which meaning is intended. Unless otherwise specified, we use client to mean a client program.

Clients and their workspaces are isolated and independent. Clients must communicate with the server to determine what is going on in other workspaces (whichever machine those workspaces might be on). This is true even if there is more than one client running on a single machine. If you tell the server about changes that you intend to make within a workspace such as an edit, add, delete, or rename, this information can be communicated to other clients and thus to other users.

Clients can operate even if they are not connected with the server. This can be useful at customer sites, in airplanes, and at times when communication with the server is either not available or unstable. Because of their specific roles, the client and the server can always determine how to bring the client context up-to-date when they next communicate.

Introducing the core Perforce concepts

This section introduces the core Perforce concepts and the contexts they apply to.

Depots

Depot is a word that you will frequently see within this book, Perforce documentation, and in the P4V interface. In general, it refers to a directory structure that the server uses to store the historic content of files.

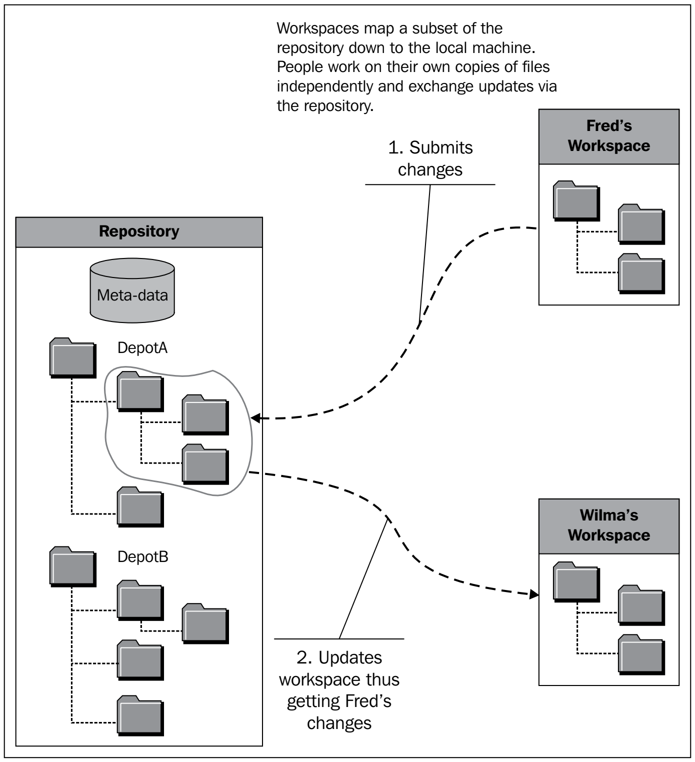
The repository that the server manages consists of one or more depots. Each depot has a unique name. A depot acts like a hierarchical file system containing folders and files. It is very common for a repository to have multiple depots. Creating depots is an administrator function.

> The word depot is sometimes used to mean the repository, and sometimes to mean a particular depot within the repository. We will refer to repository or depot specifically.

Workspaces

Workspace is a common word within the source control community. Unfortunately, there seem to be more definitions for workspace than there are tools that support them. For Perforce, the term workspace refers to a collection of files, usually on a user's local machine, that the user wants Perforce to manage. Implicit to the Perforce concept of workspace is the relationship between repository files and the files in that workspace.

Workspaces usually contain a copy of a small subset of all the files in the repository, for example, only the files relevant to a particular project. Users operate on the copies of the files in their workspace, and submit their changes back to the repository. If another user has their workspace configured to see the same project, they can update their workspace to receive the latest copies of any updated files whenever they are ready to do so. In the following diagram, two developers (**Fred** and **Wilma**) exchange changes between their workspaces via the repository:


Workspaces map a subset of the repository down to the local machine. People work on their own copies of files independently and exchange updates via the repository.
Fred's Workspace
1. Submits changes
Repository
Meta-data
DepotA
DepotB
Wilma's Workspace
2. Updates workspace thus getting Fred's changes

There are many aspects of workspaces that a user may want to manage. However, for basic operations a single workspace on your workstation may be all that you need. Managing workspaces is covered in *Chapter 6*, *Managing Workspaces*.

Changelists

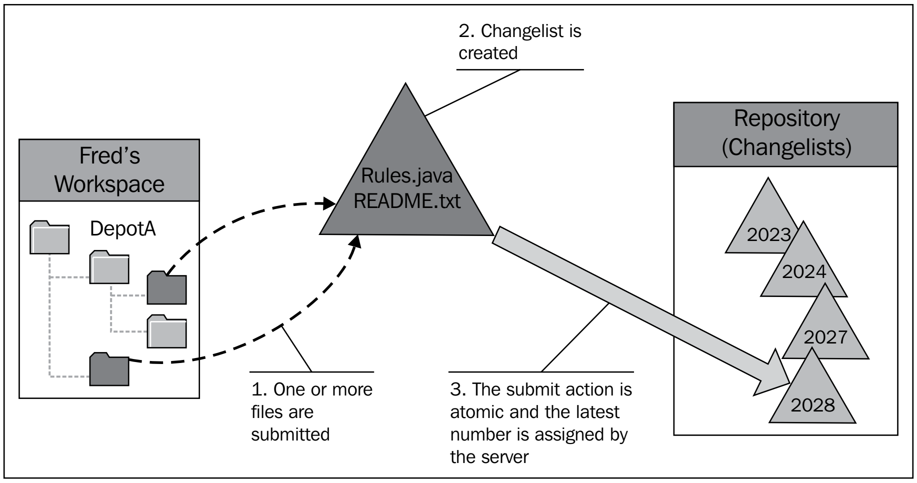
Changelist is a core Perforce concept. It is the fundamental unit of change to the repository, you don't check in files one-by-one, you check in changelists one-by-one. Each changelist contains one or many files. Changelists with thousands, tens of thousands, and even more files are not uncommon. In each workspace, it is possible to have multiple active changelists that are not yet checked in.

Changelist refer to a specific set of content and metadata changes made to the files managed by the server. Changelists have a unique numerical identifier and other attributes such as status, description, date and time of submission, and the user who submitted it. More than just a set of changes, changelists are also atomic. That is, changelists don't overlap with each other. Two changelists might be submitted at the same time, but the server will ensure that processing is completed for one changelist entirely before it starts processing the second changelist; changelists will never be intermingled by the server. Furthermore, once the server has successfully completed processing a changelist that changelist becomes immutable. That is, users can't add, remove, or modify any of the file changes represented by the changelist.

A reasonable analogy is to think of a changelist as a transaction. The Perforce equivalent is that if your changelist contains ten files, then either all ten will be submitted, or none of them. You will never get only a part of the total files submitted.

Many Perforce customers have servers handling thousands of users, and millions or more changelists. Changelists are ideal reference points for build systems, release processes, validation audits, and myriad other uses where a reproducible and verifiable set of managed sources is required.

In the following diagram, two files from the workspace are submitted as a single changelist to the repository. The changelist will automatically be assigned a unique number from an increasing sequence by the server. We cover this in more detail in *Chapter 4*, *Changelists*, including changelist states such as pending and submitted.


2. Changelist is created
Fred's Workspace
DepotA
Rules.java
README.txt
Repository (Changelists)
2023
2024
2027
2028
1. One or more files are submitted
3. The submit action is atomic and the latest number is assigned by the server

Dealing with installation

Perforce is a very straight forward system to install on various operating systems. It is not deeply embedded and thus it is relatively independent of different versions of operating systems.

Client and server version compatibility

Unlike many version control systems, Perforce does not require that clients and servers be at the exact same version. When a client connects with a server, they both determine the set of features they have in common and act accordingly. Some clients require a minimum server version. The 2013.2 version of P4V requires a server that is version 2008.1 or later.

The examples and images in this book are based on the 2013.2 version of P4V and the 2013.2 version of the Perforce server. If your P4V or server versions are not the same, you may see slightly different choices and options. The information you'll find in this book is still applicable.

Perforce platforms

Perforce servers and clients are available for a wide range of operating systems. The 2013.2 release of P4V is available for Windows, Linux, and Macintosh. You can download any of these from the Perforce website. However, some organizations have standardized on specific (older) releases. Check with your system administrator as appropriate.

Clients and servers can be running on different operating systems and still be able to work together. The servers take care of the differences between client and server host machines. The clients take care of issues that are specific to the host machine on which they operate such as text file line endings and directory separator characters.

As a user you may need to deal with character case sensitivity in directory paths and filenames. This typically comes up when a Windows-based client connects with a Perforce server that is not Windows-based. We discuss this in more detail in *Chapter 6*, *Managing Workspaces*, which focuses on workspaces. That chapter also addresses settings to handle different line endings between Windows and Linux/Unix operating systems.

Another issue that may come up is internationalization. The server is responsible for the character set used for the names of files or any part of their path. This allows you to store files with Unicode names, such as クライアントサーバモデル.doc which is the Japanese name for a file which would be called client-server model.doc in English. Consult with your Perforce administrator for further details. The client is responsible for dealing with the character set used within the contents of files.

Perforce interfaces

This book uses P4V as the client interface. P4V is a good choice as it has many presentation and graphical features. These features help to manage the large amount of information that typically describes files in a project.

P4 is the command-line client: p4.exe on Windows or just p4 on Linux/Unix. P4 is available for all of the operating systems supported by Perforce. The p4 client is ideal for automation and scripting. To assist with automation, there are language-specific interfaces providing optimized ways to execute p4 commands and process their results. These include interfaces for Perl, Ruby, Python, .NET, and Java. There is a logging option in P4V that shows the basic p4 commands being used to provide the P4V functionality. This can be very useful for individuals learning to how to automate actions or processes.

Appendix B, *Command Line* helps to relate the p4 commands to the P4V operations described in the various chapters.

Perforce integrations with IDEs such as Visual Studio and Eclipse are also common. All of the concepts and command features described within this book apply to the IDE integrations. It is not uncommon for users to use both P4V and an IDE integration. P4V provides access to the advanced information management features that don't always fit within the context of an IDE.

There are many more interfaces and integrations to Perforce. See the Perforce website (www.perforce.com) for the current list.

The Perforce mindset

Unlike some other version control tools, the Perforce clients and the server do not perform actions unless they are told explicitly to do them. So your workspace files will not be updated unless you select an action or run a command to do so. Your files will not be submitted to the server unless you, the user, select that action.

It is also expected that users inform the server about changes they intend to make to workspace files by running specific client commands such as add or edit. This allows the server to communicate those status changes to other users. It is a slightly different way of working to tools such as Git or Subversion, where users just start editing files locally and then resolve the status later at the time of commit.

Following along

You can get a lot of valuable information by just reading this book. You don't have to touch a keyboard, move a mouse, or click on a button. However, the authors have found that trying the described actions, experimenting with what ifs, and making a few mistakes, is a far better way to learn and will help you retain the information. Unfortunately, even if there weren't licensing and resource considerations, experimenting with source files on a production server is not usually encouraged. So it can be hard to get the hands-on experience that makes you a more effective tool user.

Fortunately, a Perforce server doesn't need extraordinary resources or licensing to operate. A server and client can easily be run on the same host machine including laptops. *Appendix A*, *A Demo Server* of this book describes how to obtain and install your own Perforce server and a test repository. All of the examples within this book were created using such a configuration.

Summary

This chapter has presented some of the key concepts that are important to understand when learning about Perforce such as version control, workspaces and changelists.

In the next chapter we will introduce the P4V interface.

2
The P4V GUI

P4V is a **Graphical User Interface** (**GUI**). Just as a picture is worth a thousand words, a GUI provides an excellent mechanism for displaying all the information associated with software configuration management tasks.

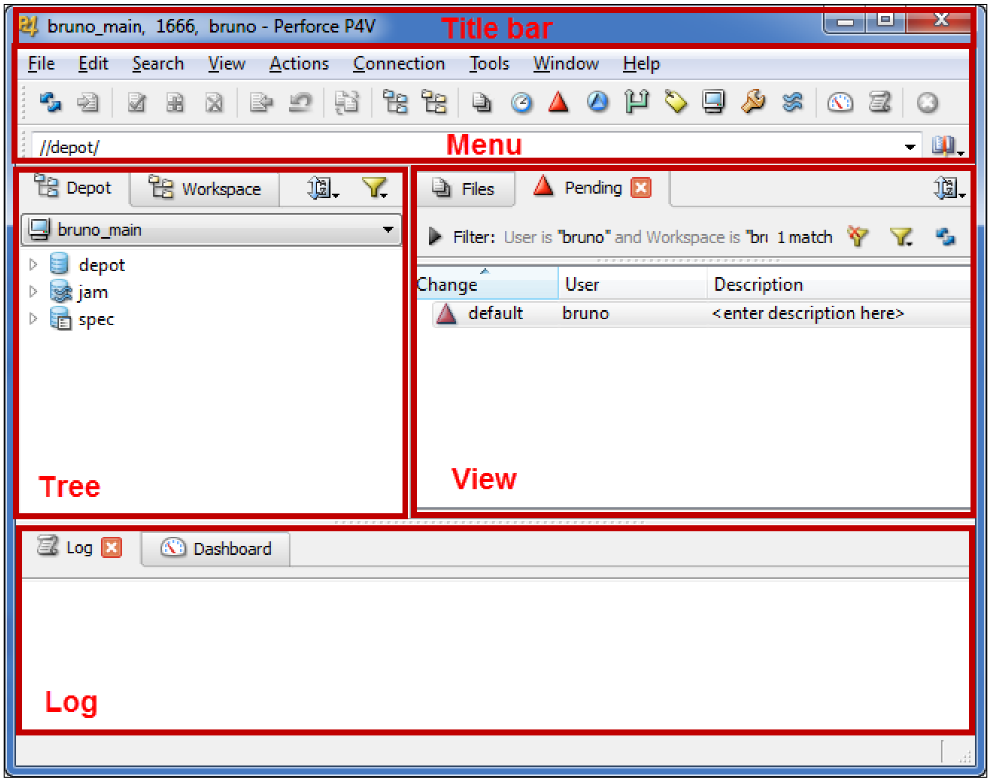
This chapter introduces the **P4V** interface. Like many GUIs, P4V uses system menus, context menus, tool tips, tabs, and other common features. We assume that the reader has some familiarity with these features, so we don't cover them in detail. Rather, this chapter covers the organization and presentation characteristics of the GUI and relates them to basic Perforce terminology. We provide interface usage details where we've found that our training students typically have questions about control functionality or location.

In this chapter we will cover:

- The layout of the P4V display
- Finding information within the display
- Accessing commands via menus, the toolbar, or shortcut keys
- An overview of the depot and workspace tabs on the tree pane
- View pane tabs and filters
- The log and dashboard tabs in the log pane

Understanding the P4V display

The P4V display is organized into four general areas plus the **title bar**. Each of these areas is identified in the following screenshot:



Filter: User is "bruno" and Workspace is "bri 1 match

Change	User	Description
default	bruno	<enter description here>

We say general areas because you as a user can adjust the boundaries of the areas and control when they appear. Each of the P4V display areas is dedicated to a specific activity or type of information. The title bar shows the current workspace, server we are connected to, and our Perforce username. As a quick overview of the others:

- **Menu**: Coordinates actions and provides control over the details presented
- **Tree**: References the files being managed by P4V
- **View**: Presents information views about managed files on various tabs
- **Log**: Provides information about actions that have happened (Log) or are recommended (Dashboard)

Each of these areas is detailed in the following sections.

Accessing P4V actions

Like most GUI applications, P4V allows actions via command menus, tools bars, and shortcut keys.

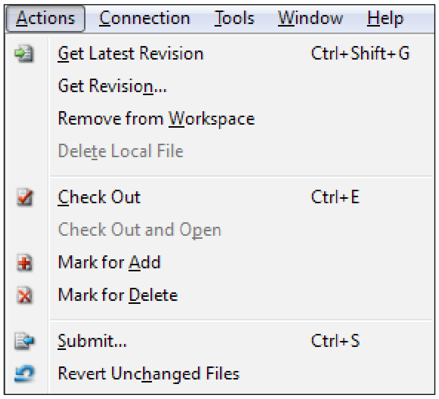
Command menus

All of the features provided by P4V can be accessed through command menus. However, many users find that most of the features they use are better accessed through more direct methods such as context menus. We'll cover those methods throughout this book. The organization of the menus is as follows:

- **File**: Used for creating new Perforce objects and accessing file content.
- **Edit**: Provides standard copy-paste editor features and access to preferences via **Edit | Preferences**. Preferences are covered in *Chapter 10*, *The P4V User Experience*.
- **Search**: Used for searching for files, text strings, and managing filters. More on managing filters is covered later in this chapter.
- **View**: Used for controlling information presented by the GUI such as tabs, filters, and sort orders.
- **Actions**: Provides source control activities relative to a selected item including **Get Latest**, **Add**, **Edit**, and **Submit**.
- **Connection**: Provides connections to the server including user and workspace management. We'll go into detail on connections and users in *Chapter 3*, *Basic Functions*. Workspaces are covered in *Chapter 6*, *Managing Workspaces*.
- **Tools**: Provides access to P4V custom tools. Also used for defining and managing bookmarks as discussed in *Chapter 10*, *The P4V User Experience*.
- **Window**: Mostly used for controlling information tabs.
- **Help**: Provides general help including the **Am I connected to the server** query by going to **Help | System Info**.

Shortcut key combinations

A selection of shortcut key combinations is available for users who are more comfortable or efficient with a keyboard interface. Common editor, search, action, and view functions are supported with shortcuts as shown in this menu:

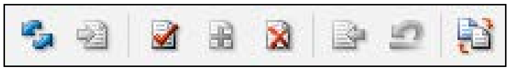


On the left-hand side of the menu we also see the equivalent toolbar icons where these exist for the command.

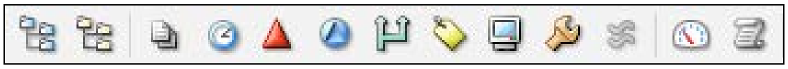
The Toolbar

The **toolbar** is a collection of buttons that provide one-click access to the most common actions and view tabs. When present, the toolbar resides just under the menus.

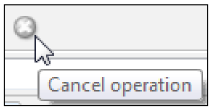
The buttons on the left-hand side of the toolbar provide access to common display management and source control actions:



The buttons on the right-hand side of the toolbar control the presentation of the tree, view tab, and log panes:



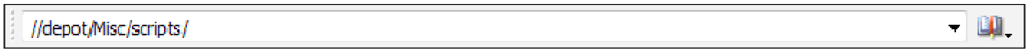
Finally, the button at the right end of the toolbar attempts to cancel the current operation (this also shows an example tool tip):



Cancel operation can be useful when you make a request that is running for an unusually long time or consuming more bandwidth than the network has available. We'll discuss reasons you might want to cancel an operation and the consequences of doing so at several points within this book.

The Address bar

The **address bar** optionally appears under the toolbar:



//depot/Misc/scripts/

The address bar identifies the currently selected object in the tree pane. This might be a directory or a file that other menu actions operate on. The book icon at the end of the address bar provides a bookmark navigation feature for frequently accessed tree pane structures. The bookmark feature is discussed in *Chapter 10, The P4V User Experience*.

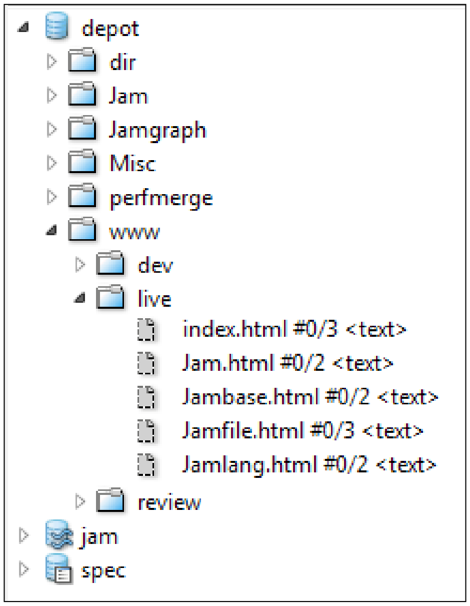
Exploring the tree pane

The depot (repository) tree and the workspace tree share the **tree pane**. However, only one of them is visible at a time. Many actions can be performed from either tree view. As we see, there are certain context specific actions that reference information specific to either a depot or a workspace (but not both).

> This is an example of Perforce using the word **depot** to mean the **repository** (which can contain multiple depots). We will call it a depot tree for consistency with P4V and Perforce documentation, even though it is better to think of it as a repository tree.

The depot tree tab

The depot tree tab presents information about files as the server understands them. In the following screenshot we see a depot called **depot** at the top, and then a tree of folders and files under that depot. The icon and version information **#0/4 <text>** associated with files provides a quick source of information about the status of files. The details will be explained in *Chapter 3*, *Basic Functions*.

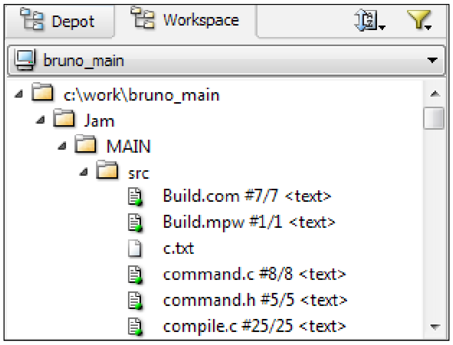


```
depot
  dir
  Jam
  Jamgraph
  Misc
  perfmerge
  www
    dev
    live
        index.html #0/3 <text>
        Jam.html #0/2 <text>
        Jambase.html #0/2 <text>
        Jamfile.html #0/3 <text>
        Jamlang.html #0/2 <text>
    review
jam
spec
```

The depot tree is not restricted by your current workspace and allows you to see anything you have access to in the repository. Sometimes it is useful to focus only on the items in your workspace, and sometimes you want to be able to view other areas of the repository, and even the contents of those files.

The workspace tree tab

The workspace tree tab presents information about files in your workspace. In the following screenshot we see a workspace tree. The tree starts at the top level directory of the workspace under which a structure of directories and files is presented. The icon and version information associated with the files in the workspace tree tab match those found in the depot tree tab.



bruno_main

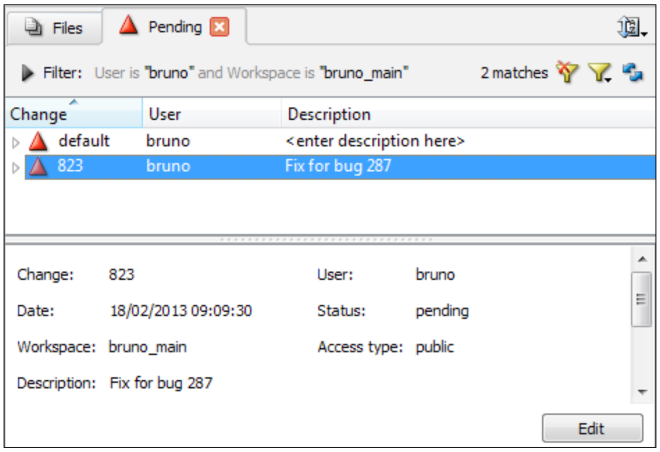
```
c:\work\bruno_main
  Jam
    MAIN
      src
        Build.com #7/7 <text>
        Build.mpw #1/1 <text>
        c.txt
        command.c #8/8 <text>
        command.h #5/5 <text>
        compile.c #25/25 <text>
```

The workspace tree tab only shows files as they currently exist in your workspace, which is typically on your workstation. As we can see in the preceding screenshot, the directory is displayed in the syntax of your operating system, in this case relative to the workspace root on a Windows machine of **c:\work\bruno_main**. Displaying information about all of the files in your workspace allows you to identify files that you may need to add to the depot such as **c.txt** in the preceding screenshot.

Exploring the view pane

The **view pane** presents information for the files managed by Perforce. In general, you select a file in the tree pane then select an action from either the menus or tool bar to make the information visible in the view pane. Each information view is a separate tab in the pane.

Although each of the individual tab views is unique, tabs have many common characteristics. Most allow you to select and customize the information presented. The following screenshot shows a view pane with the files and pending changelist tabs:



Filter: User is "bruno" and Workspace is "bruno_main"   2 matches

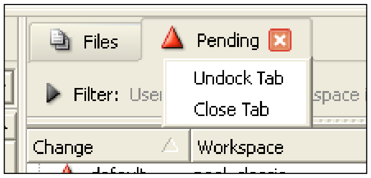
Change	User	Description
default	bruno	<enter description here>
823	bruno	Fix for bug 287

Change:	823	User:	bruno
Date:	18/02/2013 09:09:30	Status:	pending
Workspace:	bruno_main	Access type:	public
Description:	Fix for bug 287		

Closing, undocking, or docking information views

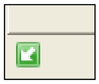
When you no longer need a view you can close it. Closing view tabs does not cause information to be lost or changed.

Undocking a view tab is controlled from the tab's context menu as shown in the following screenshot:



Undocking a tab creates a new window separate from P4V that contains only the information for that tab. You can move undocked tabs to other parts of your screen. Although undocked, the information is still connected to operations being performed by P4V.

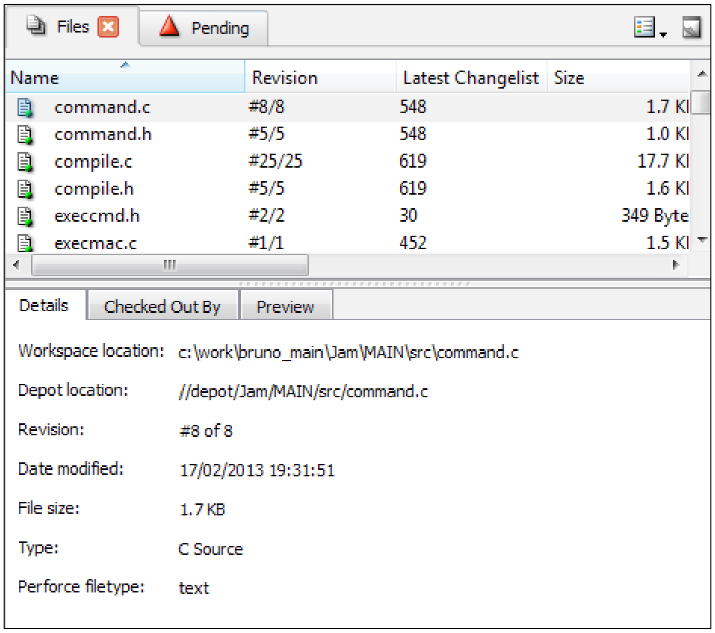
When you no longer want a tab to be in a separate window you dock the tab by selecting the docking arrow in the lower-left corner, as shown in the following screenshot:



Closing an undocked tab does not return it to the view pane the next time it is requested. P4V remembers that you wanted the tab undocked, so it saves you the positioning overhead and automatically brings it back undocked. To return a tab to the view pane you must dock it!

Viewing tabs with detail panes

Many tabs show multiple items. Additional information about a selected item in the tab is presented in an optional detail pane at the bottom of the tab. In the following screenshot, there is a detail pane for the file information tab:



Name	Revision	Latest Changelist	Size
command.c	#8/8	548	1.7 KI
command.h	#5/5	548	1.0 KI
compile.c	#25/25	619	17.7 KI
compile.h	#5/5	619	1.6 KI
execcmd.h	#2/2	30	349 Byte
execmac.c	#1/1	452	1.5 KI

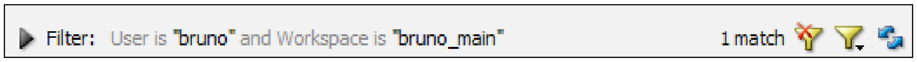
Workspace location:	c:\work\bruno_main\Jam\MAIN\src\command.c
Depot location:	//depot/Jam/MAIN/src/command.c
Revision:	#8 of 8
Date modified:	17/02/2013 19:31:51
File size:	1.7 KB
Type:	C Source
Perforce filetype:	text

When there is a lot of possible detail information, the detail panes may use sub-tabs within the tab. In the preceding example there are sub-tabs for **Details**, **Checked Out By,** and **Preview**. Unlike view tabs, the sub-tabs cannot be undocked or closed.

> Detail pane information is available through other interfaces. To recover the screen space used by the detail panes hide them using the **Show/Hide** option in the **View** menu.

Filtering view tabs

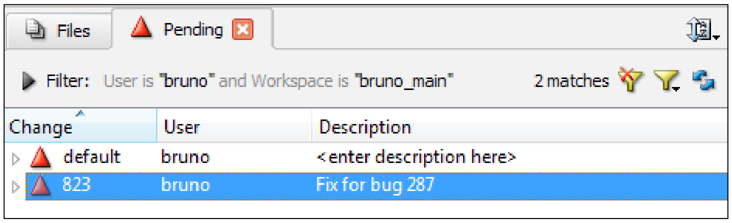
When appropriate, view tabs provide a filtering mechanism to limit the information presented in that view. Although each tab has a unique set of filter criteria, all filters use common interface mechanisms. We'll work through a single set of filters to demonstrate these mechanisms:



Filter: User is "bruno" and Workspace is "bruno_main"   1 match

The filter icons are shown on the right-hand side of the preceding screenshot.

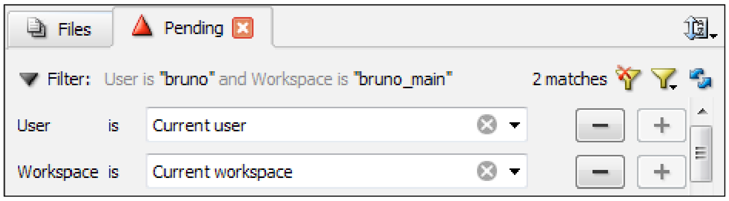
In the following example, we see that pending changelist views have a filter:



Filter: User is "bruno" and Workspace is "bruno_main"

Change	User	Description
default	bruno	<enter description here>
823	bruno	Fix for bug 287

To hide or access the filter select the right pointing triangle button in front of the word **Filter:**. An expanded filter specification is shown as seen in the following screenshot (the triangle points downwards):

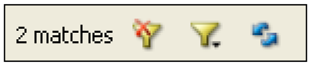


Filter: User is "bruno" and Workspace is "bruno_main"

User is Current user

Workspace is Current workspace

Each filter specification has a summary and a set of control buttons. The summary identifies the filter parameters and the total number of view items that match that filter. The control buttons provide ways to save and recover saved filter definitions, as well as forcing a refresh of the view display. The control buttons are shown in the following screenshot:

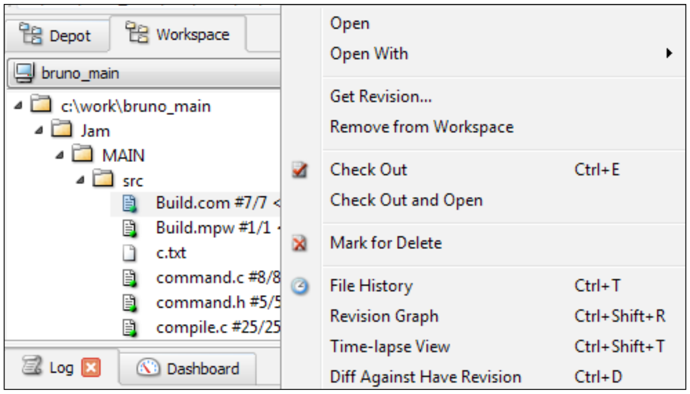


Like most parts of the P4V interface the filter control buttons have tool tips. The display of unused filter parameters is controlled with the add (**+**) and remove (**-**) buttons at the end of the parameters. Removing a filter parameter is the same as that parameter not having any effect.

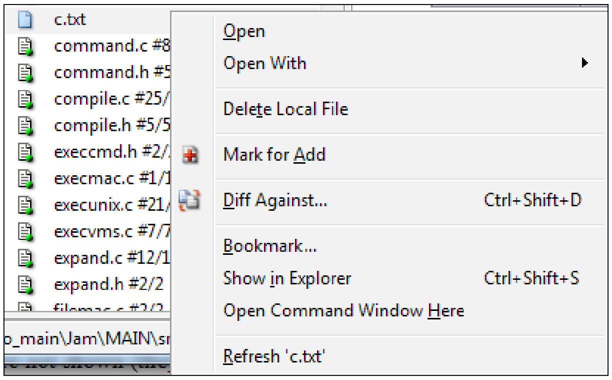
Right (context) click menus

These menus are available by context-clicking on a directory or file: in Windows this is a mouse right-click. Rather than using a toolbar or action menu, you get an item-specific menu which only shows the actions available for that item in its current state.

These are available in both the tree pane and view pane. The following screenshot shows a context menu in the workspace tree pane:



Any actions which are not available are not shown (they are not just greyed out or disabled). Different items may have different options available depending on their states. We will address specific examples throughout the book.



In the preceding example, we have right-clicked on a file which is not in Perforce. Therefore, we have the option to **Mark for Add**, this would not be available in the context menu for a file already in Perforce. Note the differences to the context menu in the previous screenshot.

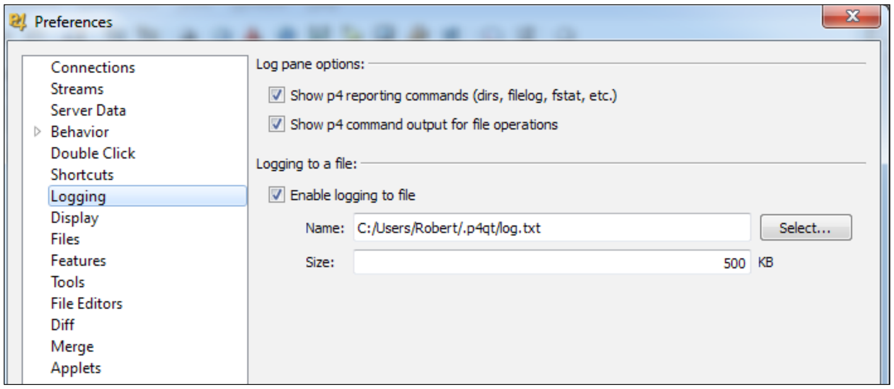
> If you expect to see a specific menu option and it is not available, this is an indication that the status of the item is not what you think!

Reviewing activities

The log pane presents information related to your activities. There are two possible tabs within the log pane: Log and Dashboard, each of which is optional.

The Log tab

The Log tab shows Perforce commands that have been used by P4V in response to the actions you have carried out. The contents of the Log tab are controlled by going to the **Edit** | **Preferences** | **Logging** menu option as shown in the following screenshot:



Log pane options:

☑ Show p4 reporting commands (dirs, filelog, fstat, etc.)

☑ Show p4 command output for file operations

Logging to a file:

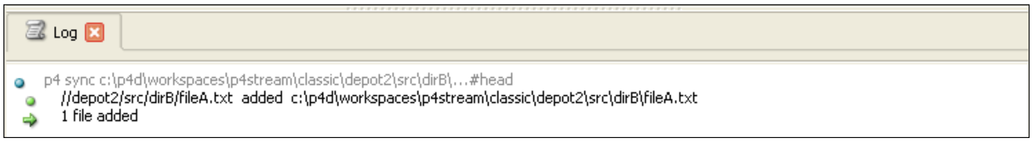
☑ Enable logging to file

Name: C:/Users/Robert/.p4qt/log.txt

Size: 500 KB

An action such as pressing *F5* (or going to **View** | **Refresh All**) refreshes the current view, this will result in various p4 commands being executed, and their results can be seen in the log tab.

In the following log example we see the command **p4 sync** and the fact that it added 1 file to the workspace:



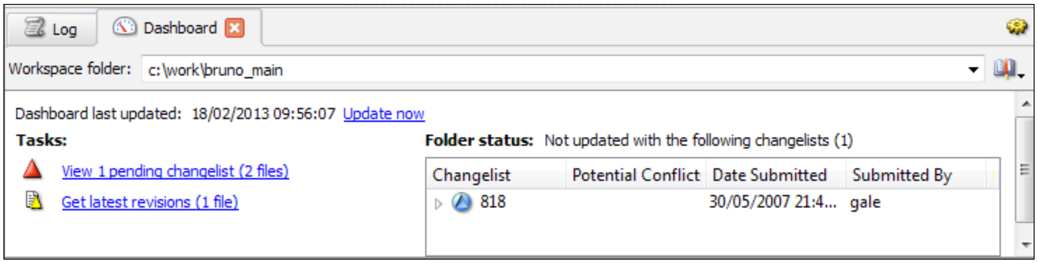
```
p4 sync c:\p4d\workspaces\p4stream\classic\depot2\src\dirB\...#head
    //depot2/src/dirB/fileA.txt  added  c:\p4d\workspaces\p4stream\classic\depot2\src\dirB\fileA.txt
    1 file added
```

Users often review log entries for details that are not otherwise found in other places. Logs are also useful for reviewing error and warning details. You can save log results to a file for analysis by your support team.

Another popular use of log entries is to identify Perforce commands that can be used in automation. *Appendix B* contains a summary of the Perforce commands used by P4V, related by book chapter.

The Dashboard tab

The Dashboard tab has two modes: display and definition. An example of the dashboard in display mode is shown in the following screenshot:



Workspace folder: c:\work\bruno_main

Dashboard last updated: 18/02/2013 09:56:07 Update now

Tasks:

- View 1 pending changelist (2 files)
- Get latest revisions (1 file)

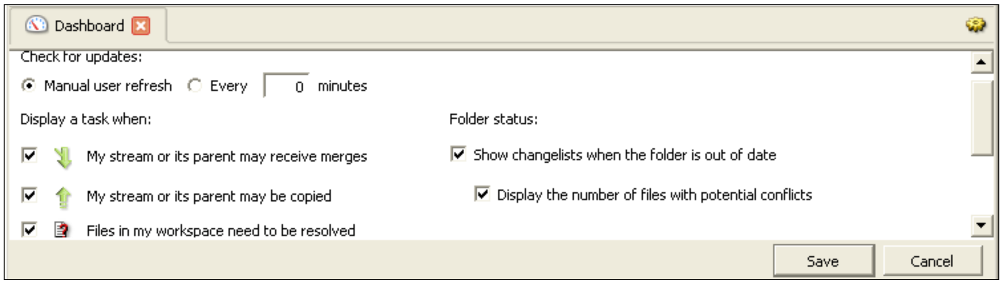
Folder status: Not updated with the following changelists (1)

Changelist	Potential Conflict	Date Submitted	Submitted By
818		30/05/2007 21:4...	gale

In the preceding example we see that one file in the current workspace is not at the latest revision known to the server, and that there is one changelist that impacts workspace files and has not been applied.

The gear button at the top-right of the dashboard toggles between the dashboard display mode we saw and the dashboard definition mode.

In definition mode, the dashboard looks like the following:



Check for updates:

● Manual user refresh ○ Every 0 minutes

Display a task when:

☑ My stream or its parent may receive merges

☑ My stream or its parent may be copied

☑ Files in my workspace need to be resolved

Folder status:

☑ Show changelists when the folder is out of date

☑ Display the number of files with potential conflicts

Many users specify manual refresh to cut down on the potential distraction associated with new display updates. When they are ready to work they request a refresh to guide their activities. Your system administrator will recommend appropriate settings for your site.

Summary

In this chapter we've introduced the functionality of the P4V interface and the terminology used to describe it. Some people may be concerned that the interface offers several ways to accomplish the same task. Not to worry. You will find that you quickly adapt to using the interface features that best fit your personal style.

In the next chapter we'll be looking at the basic Perforce functionality.

3

Basic Functions

The basic functionality of any SCM system is to track the files that are added, modified, and deleted within a workspace. This chapter demonstrates how to use the Perforce features as an individual and as the member of a team.

In this chapter we will cover:

- Accessing the server
- Populating a workspace
- Adding, editing, and deleting files
- Changing your mind
- Updating the server
- Understanding icons and file information
- Getting help

Getting something to work with

You need to connect to a Perforce server in order to have Perforce manage the files you work with. Connecting to a Perforce server involves several pieces of information, some of which are optional, so establishing your first connection can be a little intimidating. Not to worry, a failed connection attempt can't damage the repository or your user profile. The worst that happens is that you'll fail to connect and that you'll need to try again.

> You will get a lot more from this book if you follow along using P4V. The test repository detailed in *Appendix A* is ideal for follow along activities. If the test repository is not available, you may find that your organization has a standard test repository, or test area within the repository, for experimental activities. Ask your administrator. We strongly recommend against experimenting with activities that may make changes to files in a production repository.

Connecting (log in)

When you start P4V it needs information so that it can connect with a server. If you have previously connected, P4V will remember that information and provide it as the defaults for your new connection. If this is your first connection you will need to specify at least the server to connect to and your Perforce username.

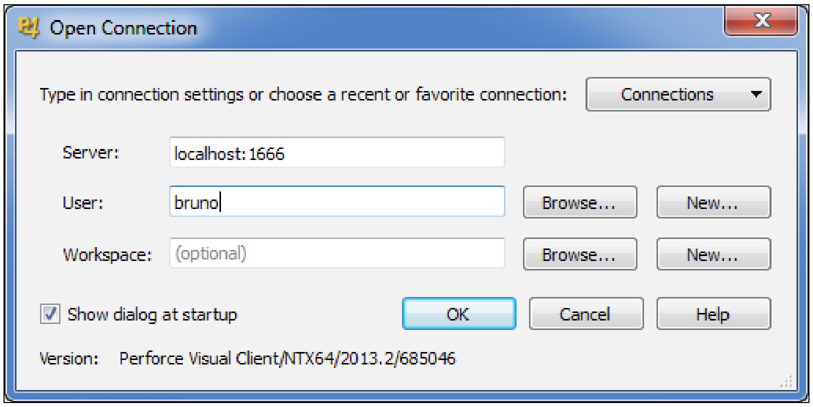
> The default installation of P4V usually causes it to launch a connection wizard when you first try to connect. In our experience users are just as happy if they skip the wizard and progress directly to the connection dialog.

The information you'll need to know in order to connect to a Perforce server is as follows:

- The network identifier for the server, for example, `perforce.abc_inc.com`.
- The port number the server listens on, for example, `1777`.
- Your Perforce user name. This may be different from your network login ID, although most administrators keep them the same, for example, `bruno`.
- An optional workspace specification. You don't need a workspace to log in and look around. An example is `bruno_main`.

The network identifier for the server and the port number are always specified as a pair separated with the : (colon) character. For example, `perforce.abc_inc.com:1777`. If you are not using the test repository your system administrator should provide you with the correct server, port, and Perforce user name.

An example connection dialog for connecting to the test repository detailed in *Appendix A, A Demo Server* is shown in the following screenshot:



Open Connection

Type in connection settings or choose a recent or favorite connection:

Server: localhost:1666

User: bruno

Workspace: (optional)

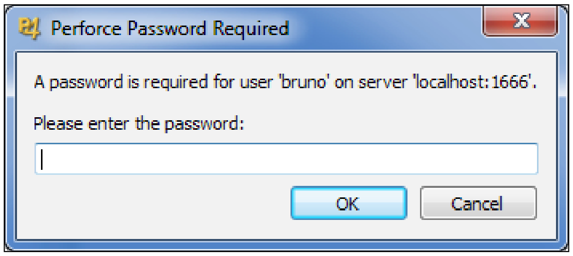
☑ Show dialog at startup

Version: Perforce Visual Client/NTX64/2013.2/685046

You can safely ignore the various buttons such as **Connections**, **Browse…**, and **New…**. They are intended for experienced users. Everything they provide is available through the standard P4V interface.

Passwords

If your Perforce user account has a password, you will be required to enter that password before a valid connection will be established between P4V and the server. Passwords are entered using the following password dialog, which is displayed automatically:


Perforce Password Required

A password is required for user 'bruno' on server 'localhost:1666'.

Please enter the password:

Enter your password and click on **OK**. If your password is correct then the P4V screen as shown in *Chapter 2*, *The P4V GUI*, will be displayed and you can perform activities.

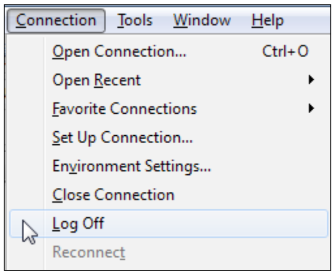
If you have installed the test repository from *Appendix A*, *A Demo Server*, then it is set up so that users do not have passwords. If you are connecting to a production server, your administrator can provide you with the password for your Perforce user account.

> If you leave P4V running for a long period of time your connection may expire. If this happens, you will be prompted to enter your password before P4V will allow you to continue.

You're done (log off)

Perforce does not require that you log off when you are done. You can simply exit P4V and let the connection expire. By default, password login sessions expire after 12 hours, so you usually only need to log in once per day, unless it is a long day! Your administrator can adjust this default if required, depending on your company's security requirements.

Log off is one of the few actions that can only be accomplished one way. You log off from P4V via the **Connection** menu:



Log off does not close P4V. It simply invalidates the current connection. If you need to reconnect, go to the **Connection** | **Open Connection...** menu to bring up the connection dialog discussed previously.

> If you exit P4V without logging off, your password login session for the server is valid until it expires. If you launch P4V and it detects a valid password login session, it won't prompt you for your password. While this is a convenience for you, it may not meet the security standards of your organization.

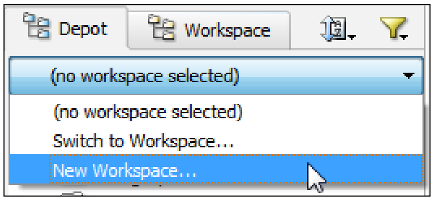
Specifying a workspace

To work with files managed by Perforce you use a workspace. A workspace is a directory structure with a root directory, usually somewhere on the file system of your workstation. You must specify the workspace you want to use. Once specified, the workspace becomes part of the context for your current connection with the server.

> You don't need a workspace to explore most of the repository information using P4V. Now, before you specify a workspace, it would be a good time to click on various items to explore the information that is available without a workspace context.

There are various ways to select a workspace in P4V. However, many users find that selecting workspaces from the workspace dropdown in the depot pane is the most convenient. The dropdown provides one-click selection of workspaces you have recently used, as well as one-click access to workspace creation and selection.

As we can see in the following screenshot, there is currently **(no workspace selected)** so the user has expanded the dropdown and selected **New Workspace...**:

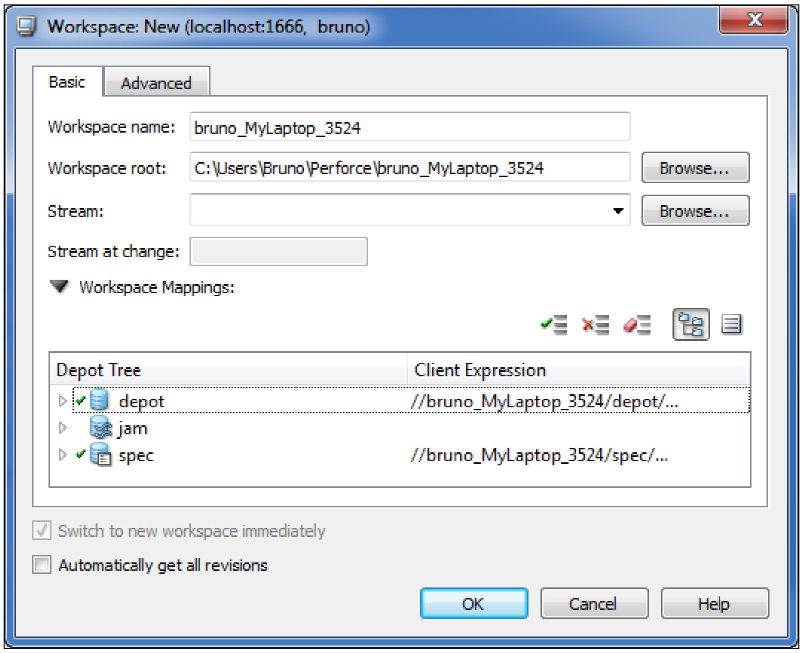


Creating an initial workspace

This section covers briefly the need to create a basic workspace in order to try some Perforce commands.

Trust us for now, if we keep to a basic level of detail, we will discuss all of the gory details about workspaces in *Chapter 6, Managing Workspaces*.

When you first create a workspace, it might look like the following:


Workspace: New (localhost:1666, bruno)

Workspace name: bruno_MyLaptop_3524

Workspace root: C:\Users\Bruno\Perforce\bruno_MyLaptop_3524

Stream:

Stream at change:

Depot Tree	Client Expression
depot	//bruno_MyLaptop_3524/depot/...
jam	
spec	//bruno_MyLaptop_3524/spec/...

☑ Switch to new workspace immediately

□ Automatically get all revisions

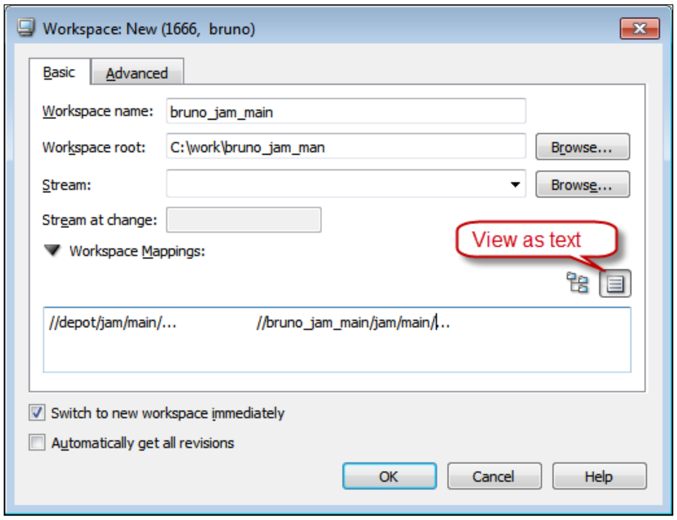
Perforce defaults to the name of the workspace as:

- <Username> such as `bruno`
- <Host machine name> such as `MyLaptop`
- <Unique numeric suffix> such as `3524`

And as you can see in the preceding screenshot, the root directory is usually under your `user` directory (this is valid for Windows 7, other versions of Windows or other operating systems may vary).

Your administrator may have recommended default settings for the workspace name and the root directory. Our recommendation is to use simple values such as:

- `bruno_jam_main`
- so <Username>_<description>



Workspace: New (1666, bruno)

Workspace name: bruno_jam_main

Workspace root: C:\work\bruno_jam_man

Stream:

Stream at change:

▼ Workspace Mappings:

```
//depot/jam/main/...          //bruno_jam_main/jam/main/...
```

☑ Switch to new workspace immediately

□ Automatically get all revisions

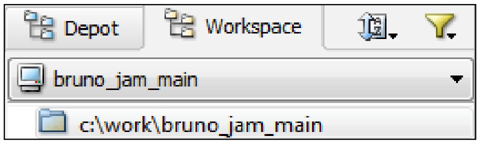
In this example, we have changed both the name (to `bruno_jam_main`) and also the root directory (to `c:\work\bruno_jam_main`).

You may also have noticed that the **Workspace Mappings:** has been changed to have a single line, after selecting the View workspace mapping as text icon:

```
//depot/Jam/MAIN/...     //bruno_jam_main/depot/Jam/MAIN/...
```

Managing workspace mappings is explained in *Chapter 6, Managing Workspaces*. We have also checked the **Switch to new workspace immediately** option.

When the workspace creation dialog closes we see that the only workspace tab information that has been updated is the workspace name and a directory path:



bruno_jam_main

c:\work\bruno_jam_main

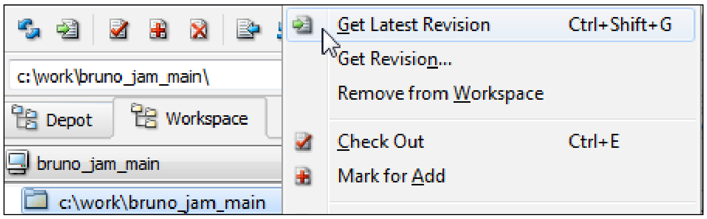
The path is the top level, or root, of the workspace area on your workstation. However, you don't see any workspace file information in the workspace tab of the tree pane. And if you try to explore `C:\work\bruno_jam_main` you probably won't even find the directory structure.

So where are the workspace files? You want to work on them! Well, you need to populate the workspace.

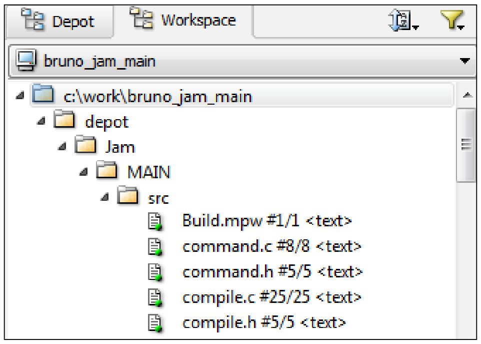
Updating a workspace – populating it with files

So far, you have only specified a workspace. You haven't requested an update of your local workstation with the server files referenced by that workspace.

If you right-click on the directory in the workspace tab, you will see a menu offering a range of activities. We're interested in the **Get Latest Revision** choice at the top of the menu:

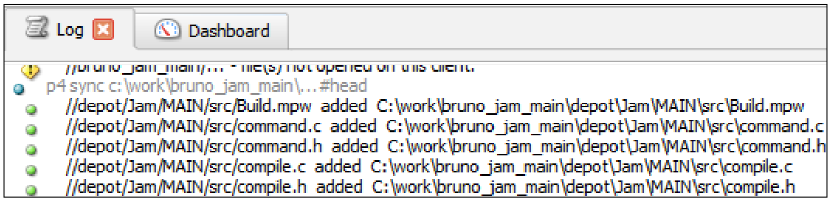


When you select **Get Latest Revision** your workstation is updated with the latest revisions of the workspace files. As you can see in the following screenshot, after the get latest has completed, file information is now visible in the workspace tab of the tree pane:



If you explore `C:\work\bruno_jam_main` on your workstation you'll also find that the expected files are now present.

If you have the log pane open when you request get latest you can see a detailed list of the transfer activities:



```
//bruno_jam_main/... - file(s) not opened on this client.
p4 sync c:\work\bruno_jam_main\...#head
    //depot/Jam/MAIN/src/Build.mpw  added  C:\work\bruno_jam_main\depot\Jam\MAIN\src\Build.mpw
    //depot/Jam/MAIN/src/command.c  added  C:\work\bruno_jam_main\depot\Jam\MAIN\src\command.c
    //depot/Jam/MAIN/src/command.h  added  C:\work\bruno_jam_main\depot\Jam\MAIN\src\command.h
    //depot/Jam/MAIN/src/compile.c  added  C:\work\bruno_jam_main\depot\Jam\MAIN\src\compile.c
    //depot/Jam/MAIN/src/compile.h  added  C:\work\bruno_jam_main\depot\Jam\MAIN\src\compile.h
```

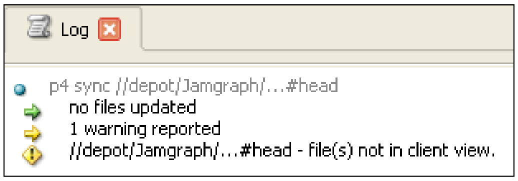
> The time required to update the log for a large number of files can have a noticeable impact on performance. Consider going to **Edit** | **Preferences...** | **Logging** to hide the optional details in the log output.

Populating a workspace from the depot tab

Workspace population from the depot tab is very similar to population from the workspace tab. The right-click menu choices, the generated log information, icons, file information, and other factors are effectively the same.

The biggest difference is that the depot tab shows all of the files that the server knows about. In the depot tab, files don't have to be local to your workspace or your workstation to appear in the P4V display. This allows you to selectively populate a workspace with subsets of the files known to the server. There are many advanced usage scenarios where this comes in handy.

However, there is a downside to populating from the depot tree. If you select a part of the depot not mapped by the current workspace and request **Get Latest**, nothing happens. A lack of an icon and version information changes is the clue that something didn't happen as you expected. Perforce only considers this a warning, it does not consider this an error: your request was valid, it just didn't result in any changes. You need to look in the log to see warnings. If you did, you might see something like the following:



```
p4 sync //depot/Jamgraph/...#head
    no files updated
    1 warning reported
    //depot/Jamgraph/...#head - file(s) not in client view.
```

Don't worry that you don't see the words **get latest** in any of the log text. Get latest is actually a special case of the Perforce `sync` command.

> **The text file(s) not in client view** appears in many warning and error messages. This is your clue that you are referencing files that aren't mapped by your workspace. The word client is a legacy name for workspace.

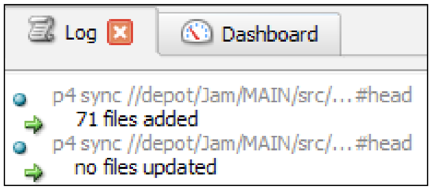
Workstation files

There are a few details about populating files on the workstation from the repository that we should note at this point.

First, the default action is to bring the files down from the server to your workstation, make them read-only, and make their last modified time be the workstation's current time and date. Making them read-only helps protect files from unintended changes or deletion.

> It is possible to have files in your workspace always writable by setting a workspace option (see *Chapter 6*, *Managing Workspaces*). This is the default behavior for tools such as Git and Subversion. The advantage of having files read-only is that it reminds users to check them out, which communicates to the Perforce server and thus other users the change of status of those files. Meanwhile, just because a file is read-only in the local workspace tree doesn't mean it is in Perforce. There are files which are created by tools in the local tree that do not belong in version control, but which happen to be read-only.

Secondly, the Perforce server is tracking the versions and status for the files in your workspace. As you will see throughout this book, server tracking is important to many of the features provided by Perforce. For example, we just did a **Get Latest Revision** to populate our workspace. The log entry in the example indicated 71 files were copied to the workspace. So what would you expect to happen if you immediately tried a **Get Latest Revision** of the same files? Well, make sure the log view is open and try it. You should see something like the following in the log:



```
p4 sync //depot/Jam/MAIN/src/...#head
  71 files added
p4 sync //depot/Jam/MAIN/src/...#head
  no files updated
```

Nothing happened (**no files updated**). This is because the server knew that the files in your workspace were up-to-date since it tracks all changes to the repository made by you and other users. The server didn't need to spend time verifying content or just blindly updating files, as it knew there was nothing to transfer. Server knowledge of workspace contents is one of the reasons Perforce can operate efficiently, even when accessed across a relatively slow network connection.

Basic SCM actions

Now that you have populated your workspace with files from the repository you'll want to work with them. As with other SCM tools, the basic Perforce actions for files include:

- Modifying a file (check out)
- Adding a file
- Deleting a file
- Reverting actions
- Identifying local changes
- Committing actions to the repository (submit)

Remember that Perforce does not allow you to directly modify the repository files (on the server). Instead, you modify your copies locally in your workspace. Then, when you're ready, you commit your changes to the repository. It's this commit that updates the repository.

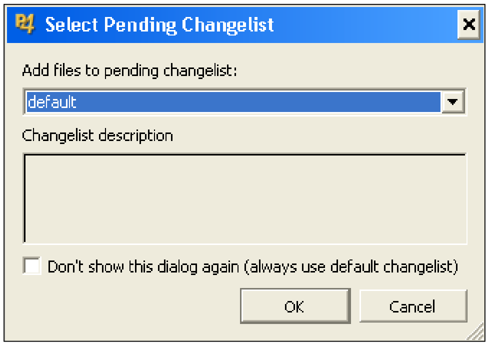
At this point, it occurs to some people that you could avoid a lot of overhead by simply letting Perforce detect what has changed in your workspace. Why not work along until you are ready, and then let Perforce determine what has changed? After all, we've told you that the server knows what should be in your workspace.

There are technical reasons this approach doesn't work well, if you think about features such as renaming and moving files. But more importantly, this approach doesn't work well because it means you are working in isolation. By coordinating your intent to change files with the server, you are communicating with other users. Nothing demonstrates the value of communication like working on a task in isolation for a week only to find that changes made by others invalidate your work. And when we get to more advanced usage, you'll find that you can use these same communication mechanisms as a reminder and as part of your own personal organizational techniques.

> Having said that, there is a relevant P4V action called **Reconcile Offline Work**, which we will cover in *Chapter 10, The P4V User Experience*.

Selecting a changelist

As you may remember from *Chapter 2*, *The P4V GUI*, Perforce tracks and organizes all changes made to the repository using changelists. So, to reference the actions, we'll talk about in this section they need to be associated with a changelist. By default, every action you perform will bring up the changelist selection dialog, as shown in the following screenshot:



Select Pending Changelist

Add files to pending changelist: default

Changelist description

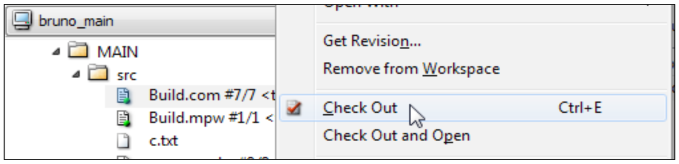
□ Don't show this dialog again (always use default changelist)

At this point, it is best to use the default changelist. Other choices are for more advanced usage as we cover in *Chapter 4*, *Changelists*. You can also select the checkbox next to **Don't show this dialog again (always use default changelist)**. When you need to use this dialog later you can always restore it by going to the **Edit** | **Preferences...** | **Behavior** menu option.

> The prefix phrases **open for** and **pending** can be found in Perforce documentation, error messages, and log entries. These prefix phrases indicate actions in progress, for example, **open for edit**. Don't worry about what is open. Just know that the action described is referenced by a changelist that is pending but not yet reflected in the repository.

Modifying existing files (check out)

You can check out an existing file in your workspace by selecting it in either the depot tab or workspace tab of the tree pane. The most common method is the **Check Out** option on the right-click menu, but notice the toolbar icon and the shortcut key: *Ctrl* + *E*.



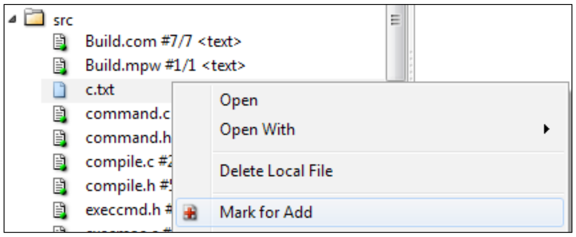
Some tools such as Subversion, use check out to mean what in Perforce is done by **Get Latest**. As you might notice in the log window, the P4V check out action results in a command called `p4 edit`, which is hinted at by the shortcut key being *Ctrl* + *E*, you are opening the file for editing, or declaring your intent to edit (modify) the file.

After a successful check out, the icon for the file will be updated with a red tick similar to the toolbar icon (also shown in the preceding menu screenshot). The file in your workspace will also be made writable and be added to a pending changelist. You can examine the contents of pending changelists using the **Pending Changelists** tab.

Although, the server now knows that you intend to modify a file, the server doesn't monitor the content of files in your workspace. As we will see, Perforce provides several techniques for saving important intermediate versions of work in progress. The best technique for your organization depends on a number of factors that we will cover throughout this book.

Adding files (mark for add)

This menu option is only available when you are in the workspace tab. Since the file is new to the repository there is nothing to display in the depot tab. The depot tab only displays information about repository files known to the server. Once the file to add is selected, the most common method is the **Mark for Add** option on the right-click menu as shown in the following screenshot:

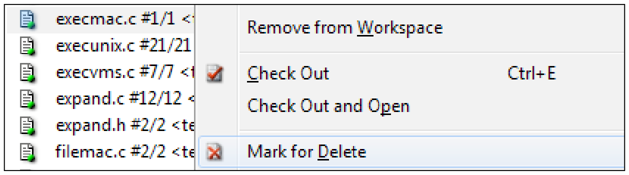


After a successful mark for adding the icon, the file will be updated with a red cross as per the toolbar icon. The file will also be added to a pending changelist. However, unlike some other actions, no changes are made by Perforce to the local file. You can continue to modify the contents of files being added up until the time you request a submit.

You may also have noticed that before **Mark for add**, the file in the display had only a name and a blank icon. After a P4V successful mark for add the file has a new icon and a Perforce file type indicator, such as **<text>** or **<binary>**. Don't worry about file types for now, Perforce does a good job of automatically determining the type. User intervention in type determination is only required for special or advanced situations and we will cover those in *Chapter 5*, *File Information*.

Deleting files (mark for delete)

You may delete a file using either the depot tab or the workspace tab in the tree view. Once the file to delete is selected, the most common method is the **Mark for Delete** option on the right-click menu as shown in the following screenshot:



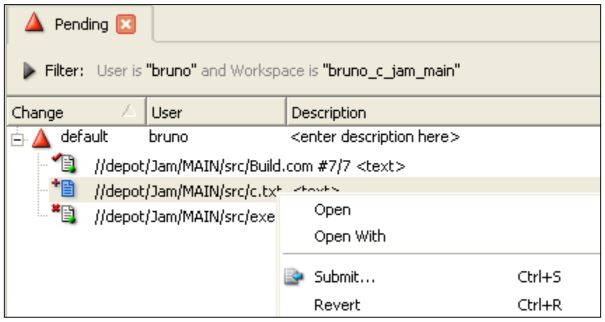
After a successful mark for delete the icon for the file, it will be updated with a red cross. The file will also be added to a pending changelist. And since you have specified that the file should be deleted, Perforce has supported engineering best practices by actually removing it from your workspace. This avoids the all too common mistake of saying that the file should be deleted then forgetting to actually delete it. This can lead to the age old problem of: "it works in my workspace"!

If you perform this action in the workspace tab, you will notice that the file immediately disappears from the tree view. However, the file does not disappear when you perform this action from the depot tab. It disappears from the workspace tab because the file is no longer in your workspace. It doesn't disappear from the depot tab because you have not yet submitted the change, so the server still knows the file still exists. This allows you to test what would happen if the file were deleted, and if you are happy, you can submit the delete action.

Reverting an action

So far we've shown you how to modify, add, and delete files from your workspace. However, what if you change your mind about the action you requested? Not to worry, you can revert any pending action.

You revert by selecting a file then using **Revert** from the right-click menu from either tab in the tree pane. However, if you're trying to revert multiple files, it can be problematic to find files in the tree pane with a large scroll region. For this reason, most users coordinate reverts from the **Pending** tab in the view pane. The **Pending** tab lists all of the open files in one convenient place. Like the tree pane, you select the file then use **Revert** from the right-click menu as shown in the following screenshot:



Filter: User is "bruno" and Workspace is "bruno_c_jam_main"

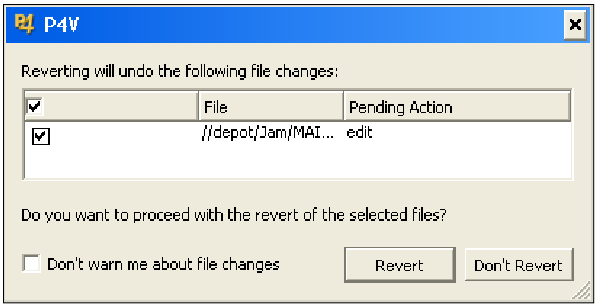
Change	User	Description
default	bruno	<enter description here>

//depot/Jam/MAIN/src/Build.com #7/7 <text>
//depot/Jam/MAIN/src/c.txt <text>
//depot/Jam/MAIN/src/exe

Note the shortcut key (*Ctrl* + *R*) which we tend to use frequently.

A revert removes the file from the pending changelist and updates view icons. What happens to the local file depends on the action being reverted:

- If you revert a file currently **Marked for Add**, there are no local changes to that file. It becomes one of the files in the workspace that is no longer under the control of Perforce.
- If you revert a file currently **Marked for Delete**, a read-only copy of the file contents that existed when you marked the file for delete are restored to your workspace.
- If you revert a check out for a file, you may be presented with a dialog to confirm your choice. This dialog only appears if you made local changes to the file and you haven't selected **Don't warn me about file changes** in a previous instance of the dialog:



Reverting will undo the following file changes:

☑	File	Pending Action
☑	//depot/Jam/MAI...	edit

Do you want to proceed with the revert of the selected files?

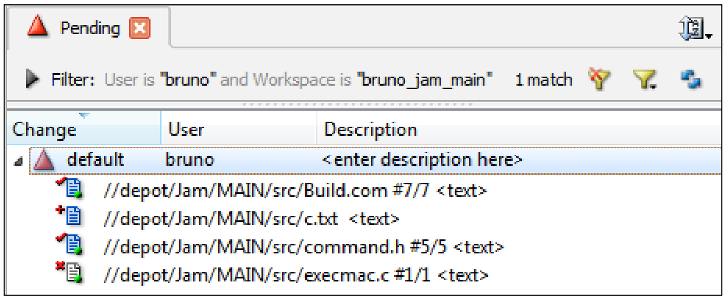
□ Don't warn me about file changes

If you choose **Don't Revert** there are no local file changes and the file stays in the pending changelist. If you choose **Revert**, or this dialog does not appear, any local changes are discarded and a read-only copy of the file contents that existed when you checked the file out is restored to your workspace.

> Now is a good time to thoroughly explore the actions we've talked about, their impact on local workspace files, and what happens when you revert an action. If you select **Don't warn me about file changes** in the check out confirmation dialog you can restore the dialog by going to **Edit** | **Preferences...** | **Behavior** menu.

Which files are you working on?

The answer to this question is always available in the Pending changelists tab in the view pane:



Pending

Filter: User is "bruno" and Workspace is "bruno_jam_main" 1 match

Change	User	Description
default	bruno	<enter description here>
//depot/Jam/MAIN/src/Build.com #7/7 <text>		
//depot/Jam/MAIN/src/c.txt <text>		
//depot/Jam/MAIN/src/command.h #5/5 <text>		
//depot/Jam/MAIN/src/execmac.c #1/1 <text>		

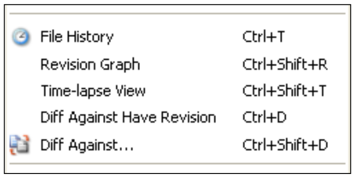
In the preceding screenshot we see the files that are checked out for edit, marked for add, and marked for delete. They are also in the default changelist for the workspace as suggested earlier.

> If you have been doing follow along, you'll see that these are the files we have been using in the preceding examples.

Identifying local changes (diff against have revision)

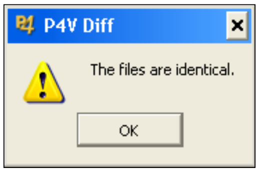
At some point, you are likely to want to know what changes you've made to a file that you've checked out. There are a number of offline techniques that you could use, but they tend to require a lot of work on your part. Fortunately, P4V can leverage server knowledge and save you a lot of effort.

From anywhere you can select a file that has been checked out and you can bring up a right-click menu. Although menus depend on the context you will see a sequence similar to the following within that menu:


File History Ctrl+T
Revision Graph Ctrl+Shift+R
Time-lapse View Ctrl+Shift+T
Diff Against Have Revision Ctrl+D
Diff Against... Ctrl+Shift+D

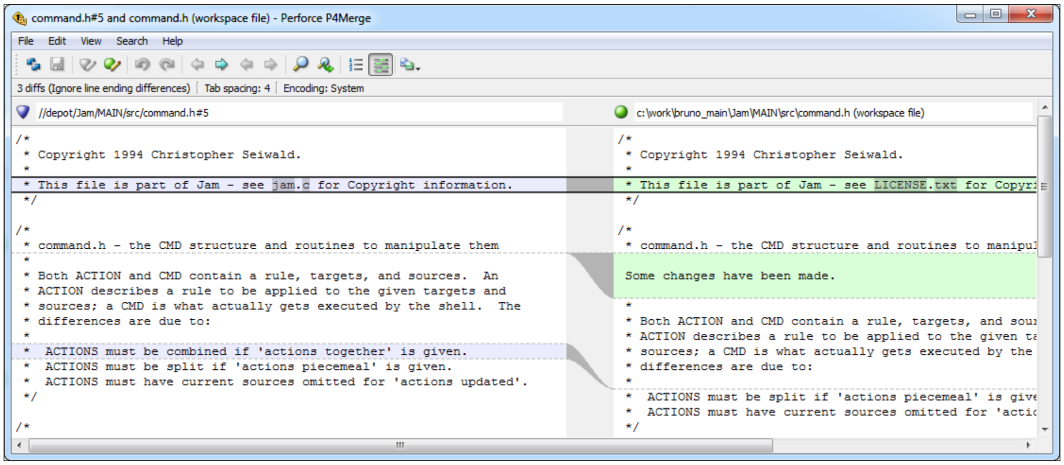
Select **Diff Against Have Revision** or use the shortcut (*Ctrl* + *D*). The other selections are for advanced features that we'll cover later.

What happens when you make this selection is that P4V makes a local temporary copy of the contents of the file as it existed before the check out and launches the difference tool to compare the local copy with that temporary copy. If there are no changes, or you made changes and then manually backed them out, you'll see the following dialog telling you about that fact:



The files are identical.

But, if there are differences, you'll see the Perforce difference tool showing you all the differences. The difference tool is part of the standard Perforce distribution and is installed along with P4V. The following is a typical difference tool display showing lines that have been modified, added, and deleted:


command.h#5 and command.h (workspace file) - Perforce P4Merge
File
Edit
View
Search
Help

3 diffs (Ignore line ending differences) | Tab spacing: 4 | Encoding: System

//depot/Jam/MAIN/src/command.h#5

```
/*
 * Copyright 1994 Christopher Seiwald.
 *
 * This file is part of Jam - see jam.c for Copyright information.
 */

/*
 * command.h - the CMD structure and routines to manipulate them
 *
 * Both ACTION and CMD contain a rule, targets, and sources.  An
 * ACTION describes a rule to be applied to the given targets and
 * sources; a CMD is what actually gets executed by the shell.  The
 * differences are due to:
 *
 *  ACTIONS must be combined if 'actions together' is given.
 *  ACTIONS must be split if 'actions piecemeal' is given.
 *  ACTIONS must have current sources omitted for 'actions updated'.
 */

/*
```

c:\work\bruno_main\Jam\MAIN\src\command.h (workspace file)

```
/*
 * Copyright 1994 Christopher Seiwald.
 *
 * This file is part of Jam - see LICENSE.txt for Copyri
 */

/*
 * command.h - the CMD structure and routines to manipul
Some changes have been made.
 *
 * Both ACTION and CMD contain a rule, targets, and sour
 * ACTION describes a rule to be applied to the given ta
 * sources; a CMD is what actually gets executed by the
 * differences are due to:
 *
 *  ACTIONS must be split if 'actions piecemeal' is give
 *  ACTIONS must have current sources omitted for 'actic
 */
```

The difference tool has many useful features, but we don't want to take too much space in this book describing them exhaustively. It is an important tool and we have found that the best way to learn about it is to use it. Check out a file, make some changes to it then launch the difference tool to see the results. Then close the tool make more changes and see the new results. See what changes make the various buttons active. Click on the buttons to see what happens. Don't worry about mistakes, you can always revert the file when you're done.

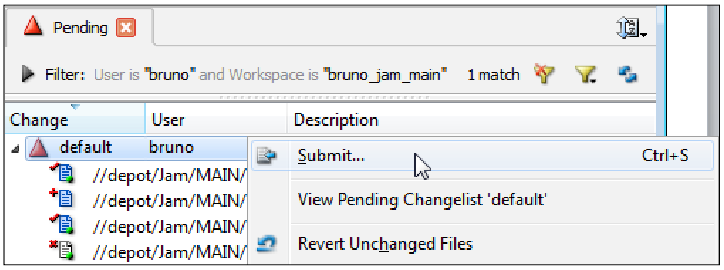
Submit – updating the server

You've completed your work. Now, you want to update the server so that there is a permanent record of your changes that others can access. So where's the check in button?

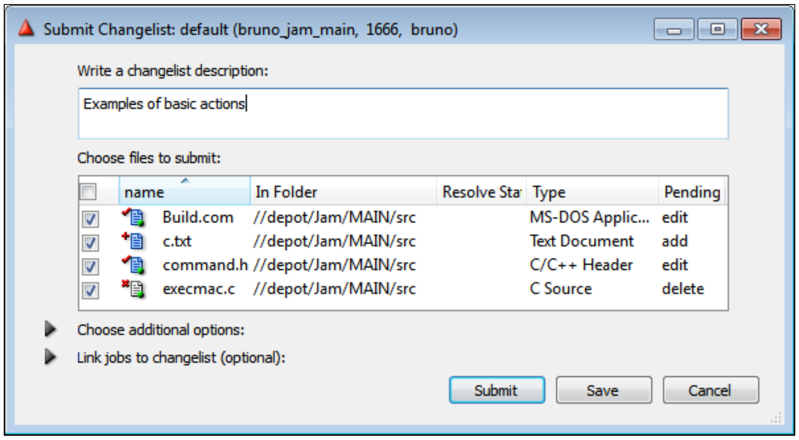
Don't worry, you didn't miss it. Perforce uses the term **submit** rather than check in. There are several reasons Perforce uses submit. But, they basically boil down to the fact that people associate check out with edits. Likewise, they associate check in with check out. Yet Perforce accounts for all differences: edits, adds, deletes, and other actions that we'll see later. So Perforce uses the **submit** term to remind you that all changes are being submitted to the server.

> Most people use the words check in and submit interchangeably.

P4V has been tracking your changes in a default pending changelist. So, the best place to coordinate submit is from the pending tab in the view panel. Select a pending changelist (in this case default) then right-click and select the top item **Submit**:



After selecting **Submit**, the submit dialog opens:



Submit Changelist: default (bruno_jam_main, 1666, bruno)

Write a changelist description:

Examples of basic actions

Choose files to submit:

□	name	In Folder	Resolve Sta	Type	Pending
☑	Build.com	//depot/Jam/MAIN/src		MS-DOS Applic...	edit
☑	c.txt	//depot/Jam/MAIN/src		Text Document	add
☑	command.h	//depot/Jam/MAIN/src		C/C++ Header	edit
☑	execmac.c	//depot/Jam/MAIN/src		C Source	delete

Choose additional options:
Link jobs to changelist (optional):
Submit
Save
Cancel

You'll need to enter a description and confirm the files to submit. The additional options are for special case scenarios that we'll explore later. There may be a **jobs** option. Jobs are an interface to external tools, such as defect tracking or issue management and relate to productions environments. Your project manager will tell you if you need to concern yourself with jobs.

You must enter a description. Make the description significant within the first 32 characters. Many P4V displays present the first 32 characters of these descriptions as summaries. Workspace name, time, date, file lists, and user identification are already available through other P4V interfaces. As you can see in the preceding example, we've entered the description: `Examples of basic actions`. It's short and clearly states the submit purpose.

Fast access file information

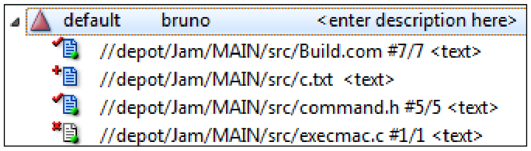
As we'll see in *Chapter 5*, *File Information*, P4V provides access to a broad range of detailed file information. However, for general operations you don't usually require that level of detail. Nor do you want to go through the effort of accessing detailed file information to figure out the general status of files in your workspace.

Fortunately, there are a number of interface features that provide fast access to general file information. Some of these you've already seen. If you've been following along, you've probably encountered others. Now that we've covered basic actions it's time to review the information sources in context.

Icons

Icons are used to convey information about files, directory elements, depots, and changelists. There is an extensive set of these icons. We'll cover the icons here that apply to basic operations. You will encounter more icons as we deal with more advanced operational topics.

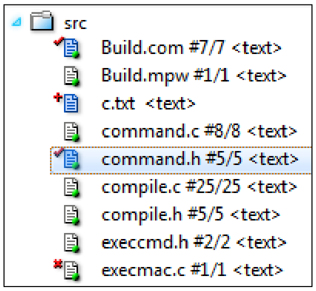
As an example, let's look at the default pending changelist presentation we saw in the preceding submit topic:



default bruno <enter description here>
//depot/Jam/MAIN/src/Build.com #7/7 <text>
//depot/Jam/MAIN/src/c.txt <text>
//depot/Jam/MAIN/src/command.h #5/5 <text>
//depot/Jam/MAIN/src/execmac.c #1/1 <text>

Note that there are unique decorations on the left-hand side of the icons indicating edit, add, and delete. The file icons themselves indicate modified and current content. The red triangle associated with the default changelist is also significant and indicates a changelist ready to submit.

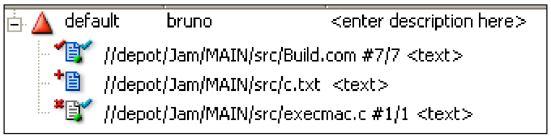
Icons are used consistently everywhere a reference to an element is displayed. For example, if we look at the depot pane in the tree view for the default preceding changelist, we'll see the same icons conveying the same information:



src
Build.com #7/7 <text>
Build.mpw #1/1 <text>
c.txt <text>
command.c #8/8 <text>
command.h #5/5 <text>
compile.c #25/25 <text>
compile.h #5/5 <text>
execcmd.h #2/2 <text>
execmac.c #1/1 <text>

If you look in P4V you would see a blue folder (check this in P4V itself) next to the src directory name: it indicates a folder in the Perforce depot.

And there's more. Let's look at the default changelist after some of the same files have been opened in another workspace:



default bruno <enter description here>
//depot/Jam/MAIN/src/Build.com #7/7 <text>
//depot/Jam/MAIN/src/c.txt <text>
//depot/Jam/MAIN/src/execmac.c #1/1 <text>

Note that the icons we saw now have blue check mark decorations on the right-hand side of the icon. Decorations on the right-hand side of an icon indicate a pending action to that file in another workspace, usually made by another person.

Versions and type

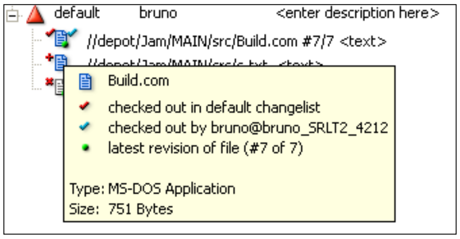
The default pending changelist screenshot also shows other information.

Look at the **#7/7 <text>** after **Build.com**. Don't worry, advanced math is not required! The **#7/7** indicates that version 7 of the 7 versions of the file **Build.com** was in your workspace when you did the check out. The **<text>** is the Perforce file type for the file.

Note that there is a type **<text>** after **c.txt**, but there is no **#1/0** or other numbers. There is no versioning information for the file because the file does not currently exist within the repository.

File status tool tips

So far we've covered information presentations that are always visible. While useful, sometimes you need more details, such as who else has the file checked out. Additional information is available through file tips. These appear when you hover over the icon for the file. For example, let's look at the tool tip that would appear if you were to hover over the icon for **Build.com** in the preceding scenario where **Build.com** is checked out in another workspace:



default bruno <enter description here>

//depot/Jam/MAIN/src/Build.com #7/7 <text>

//depot/Jam/MAIN/src/c.txt <text>

Build.com

✓ checked out in default changelist

✓ checked out by bruno@bruno_SRLT2_4212

● latest revision of file (#7 of 7)

Type: MS-DOS Application

Size: 751 Bytes

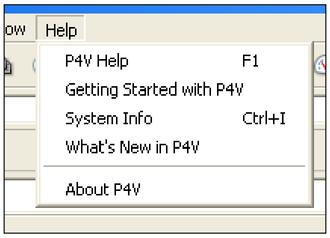
In addition to the always visible icon, version, and type information, the file tip contains additional information such as the size of the file, the meaning of the icon decorations, and information about the other workspace where the file is checked out.

Getting help

Details that you aren't familiar with, new features, and the features you haven't tried before, all need the support of a good help system. The help system built into P4V is dedicated to P4V and is often context sensitive.

The help menu

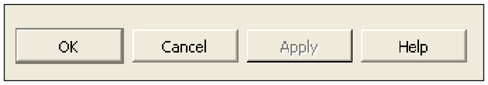
The **Help** menu can be used to access a full range of help features. Most of the time you will be using the **P4V Help** selection. This brings up the **P4V help viewer**. The P4V help viewer provides search, bookmarks, and a host of other features:



The **System Info** selection provides operational information that may be required by support or your administrator.

Help buttons

Many dialogs have a help button. Selecting the help button will bring up the P4V help viewer focused on the topic that relates to that dialog:



The Perforce website

From documentation to user forums the Perforce website (`www.perforce.com`) contains a wealth of information for Perforce users. The search interface is a powerful resource. Results are organized by origin to help isolate the style of information you'll find.

Although the Perforce website contains a wealth of information, the amount of information available can be intimidating to new users. Moreover, the range of versions covered is broad. The topics, information, and screenshots you find may not be appropriate to the version of P4V that you're using.

> Take the opportunity to investigate a topic using the built-in P4V help, then search for the same topic on the Perforce website. Icon is a good topic choice. This will help you to appreciate the differences between the two resources.

Summary

Because add, edit, and delete are self-evident, people have rarely taken the time to actually identify all the steps required to accurately track and reproduce such changes. Moreover, it's even less likely that they would have considered interacting with other users or providing a seamless baseline for more advanced activities.

Thus, many first time users make the observation that there seems to be far too much overhead for tracking simple file activities. They don't appreciate that Perforce is providing a rich set of features in a very efficient interface.

In the next chapter we will cover details about changelists and related concepts.

4 Changelists

Changelists are a record of change. Before it is submitted, that record represents pending actions that are not yet permanent. After submit, changelists become an immutable record of what changed.

In this chapter we will look beyond using changelists as a simple record of change. We'll see how to use them for organizing work, communicating with other users, and avoiding problems.

In this chapter we will cover:

- Creating useful changelists
- Using changelists for organization
- File management at submit
- Changelist numbering

Creating useful changelists

As we will see throughout this book, changelists are the primary mechanism for change tracking within Perforce. We submit our changes to the repository changelist by changelist, not file by file.

In and of themselves, the myriad change details automatically tracked by Perforce are just details. It is up to you to give purpose to these details. You are the one that makes the difference between changelists that are useful and changelists that are just collections of details. If you think changelists are overhead that make you less efficient, that might be a sign that your changelists aren't as useful as they could be!

The good news is that useful changelists are easy to create. You just need to follow some straightforward best practices.

Associating files with a changelist

The user has total control over the choice of file(s) to submit with a changelist. Following the simple best practice of creating changelists of related files will reap you and your team large rewards. Not doing so can create significant extra work in the future!

For example, let's say you fix 10 bugs, each of which requires changes to several files. While it is possible to submit a single changelist with all of the changed files in it, that is not usually the best practice. A single changelist implies that all of the file modifications are interrelated. Or, said in another way: these aren't 10 separate bugs, they are actually just one large bug. This is unlikely to be true. Moreover, a single changelist makes it hard to isolate the files required to fix a specific bug in the future. So, the best practice is to submit the modified files for each bug fix in its own separate changelist, one by one. This uses changelists to isolate the files required to resolve a specific bug and makes the history easier to understand.

In a similar way, if you need to modify 10 files to fix a bug, you could submit these modifications using 2, 3, or even 10 separate changelists. But, determining that those separate changelists are related in the future may require manual overhead and it introduces a higher likelihood of error. The most useful changelist would contain all 10 file modifications. This useful changelist not only reduces future overhead, but also creates a record of relationships that might otherwise be lost.

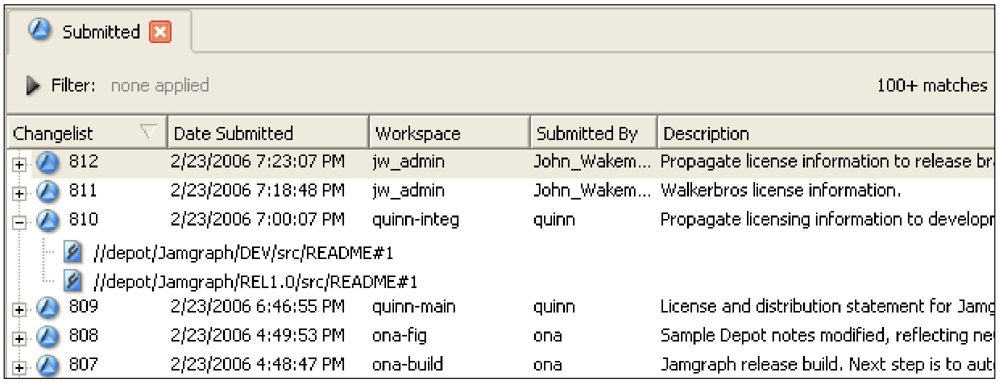
These are ideal scenarios. They are not always achievable in the real world. For example, you think you have fixed a bug so you submit the changelist. You then realize you missed an aspect of the bug fix and need to make further modifications. These new modifications will have to go into a separate changelist, you can't add them to the previously submitted changelist.

> When in doubt, think about trying to re-create your current work in the future. Make sure those are the files in your changelist.

Effective descriptions

In the previous chapter we introduced the need for effective changelist descriptions. We stated reasons without providing specific details. Now it's time to take a closer look at how those reasons apply.

Perhaps the best way to see what we're talking about is to look at the submitted changelist tab view. We see an example as follows:



Submitted

Filter: none applied

100+ matches

Changelist	Date Submitted	Workspace	Submitted By	Description
812	2/23/2006 7:23:07 PM	jw_admin	John_Wakem...	Propagate license information to release br
811	2/23/2006 7:18:48 PM	jw_admin	John_Wakem...	Walkerbros license information.
810	2/23/2006 7:00:07 PM	quinn-integ	quinn	Propagate licensing information to developr
//depot/Jamgraph/DEV/src/README#1				
//depot/Jamgraph/REL1.0/src/README#1				
809	2/23/2006 6:46:55 PM	quinn-main	quinn	License and distribution statement for Jamg
808	2/23/2006 4:49:53 PM	ona-fig	ona	Sample Depot notes modified, reflecting ne
807	2/23/2006 4:48:47 PM	ona-build	ona	Jamgraph release build. Next step is to aut

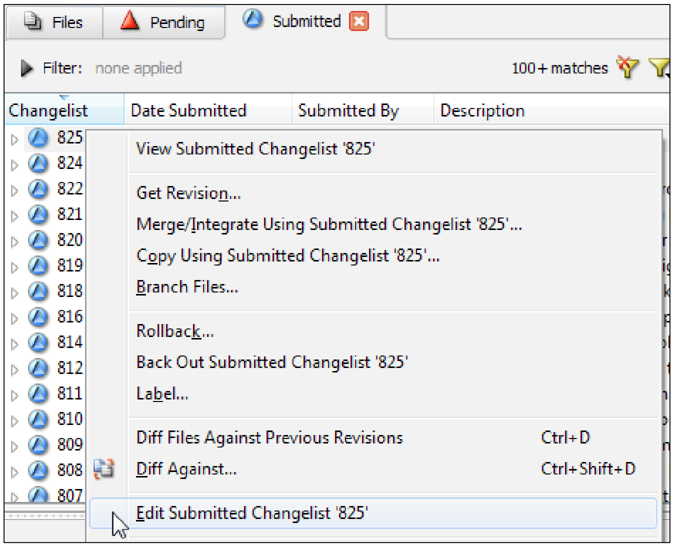
As you can see, Perforce tracks date, workspace, and user information. We've expanded changelist **810** to show that information about the files involved is also readily available. There is no need to duplicate this information in a description.

Now look at the **Description** column. The text comes from the first few lines of the description entered by the user. If you look at these descriptions, you can imagine looking back months from now to quickly identify changelists of interest. You can do that because the description starts with the submit purpose. Additional details are fine, but useful descriptions start with a summary of this purpose.

Many organizations have standards for changelist descriptions. For compatibility with legacy procedures and tools these standards sometimes conflict with the recommendations in this book. Always use the standards of your organization.

Updating descriptions for submitted changelists

While useful descriptions are the objective, we're all human. Typographical errors, missing details, incorrect details, and other problems are inevitable. Not to worry. You can modify the description of a submitted changelist. This is an option on the right-click menu in the submitted **Changelists** tab view as shown in the following screenshot:


Files
Pending
Submitted
Filter: none applied
100+ matches
Changelist
Date Submitted
Submitted By
Description
825
824
822
821
820
819
818
816
814
812
811
810
809
808
807
View Submitted Changelist '825'
Get Revision...
Merge/Integrate Using Submitted Changelist '825'...
Copy Using Submitted Changelist '825'...
Branch Files...
Rollback...
Back Out Submitted Changelist '825'
Label...
Diff Files Against Previous Revisions Ctrl+D
Diff Against... Ctrl+Shift+D
Edit Submitted Changelist '825'

This option will only be available to you if you have selected a changelist that you have submitted. You are not allowed to update another person's changelists unless you are an administrator.

Using changelists for an organization

So far we've only talked about the default changelist associated with a workspace. However, workspaces can have one or more pending changelists in addition to their default changelist. Among other uses, these additional pending changelists provide a mechanism for organizing your work in progress.

For example, separate pending changelists might be used to organize the files involved in separate bug fixes. A less obvious organizational technique is to use pending changelists as file action notepads. For example, you might need to modify files to insert temporary trace features. Collecting all of the check out actions for these files in a separate changelist makes them easy to identify and revert.

Pending changelists also have a communication role. Other users can identify the files you have open for some action. They can also see the descriptions associated with your pending changelists. This allows users to identify overlaps or potential conflicts in work effort, and even previously unknown relationships between different work activities.

> The list of files associated with a pending changelist can be changed, whereas the list of files associated with a submitted changelist is immutable and cannot be changed.

The default changelist

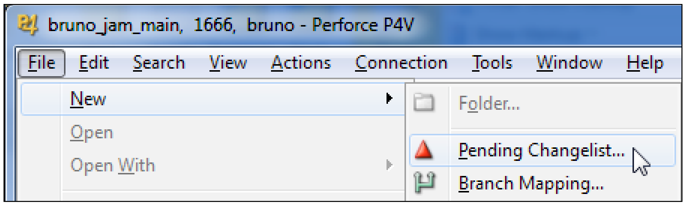
Every workspace has a pending changelist called **default**. You don't have to do anything to create it, it is always there. You can't change its name, nor can you delete it. Always having a default changelist assures that pending actions can always be associated with a changelist.

The downside to the default changelist is that you can't pre-establish its description. This removes a potentially vital piece of communication detail. For example, another user can see you have a file checked out in your workspace's default changelist. However, there is no description for the default changelist so they can't tell if you're working on a bug, adding a feature, or doing some other type of work.

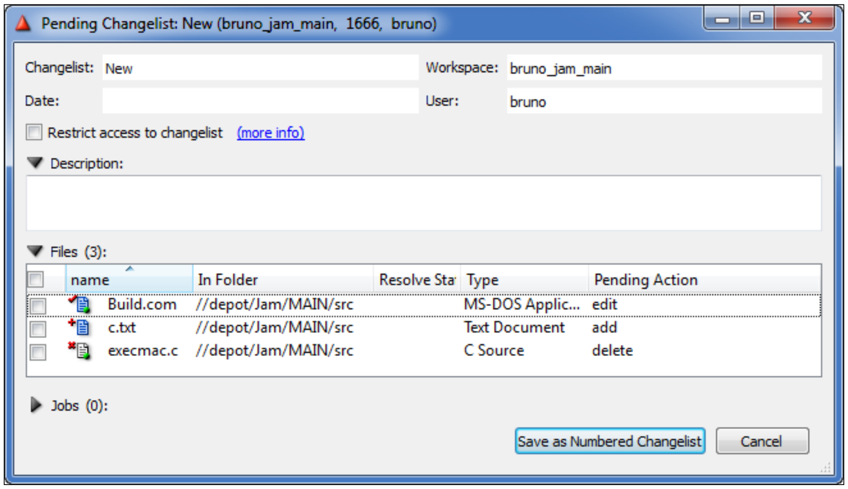
Other pending changelists

While a workspace has only one default changelist, it can have any number of pending changelists. Pending changelists, other than the default, are assigned a unique number by the server. You can't choose that number and you can't change it after it is assigned.

There are many interfaces that allow you to create new pending changelists. You can navigate to the **File** | **New** | **Pending Changelist...** menu shown in the following screenshot, or use the *Ctrl* + *N* shortcut, and the **New Pending Changelist...** option choice from the right-click menu that appears when you select a changelist in the **Pending** tab view:



Regardless of the interface used you will be presented with the pending changelist creation dialog:



Pending Changelist: New (bruno_jam_main, 1666, bruno)

Changelist: New

Workspace: bruno_jam_main

Date:

User: bruno

□ Restrict access to changelist (more info)

Description:

Files (3):

□	name	In Folder	Resolve Sta	Type	Pending Action
□	Build.com	//depot/Jam/MAIN/src		MS-DOS Applic...	edit
□	c.txt	//depot/Jam/MAIN/src		Text Document	add
□	execmac.c	//depot/Jam/MAIN/src		C Source	delete

Jobs (0):

You had a reason for creating this pending changelist. Be sure to record that reason in the **Description:** area. If you don't provide a description P4V will provide a default. However, keep in mind that a default description has limited communication value.

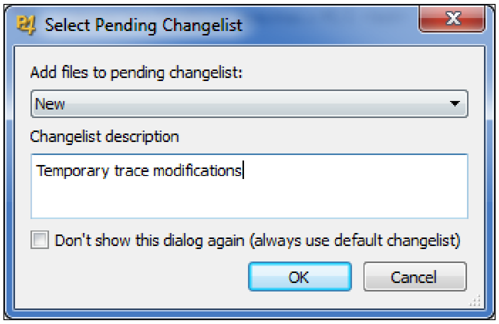
So where did the files in the **Files** list come from? They came from the default changelist. Files in the default changelist are used because people are typically trying to organize files that are already open. However, you aren't forced to put these files into the new pending changelist. In fact, you must explicitly select files to be in the new changelist by selecting the checkbox next to the file name. If there are no files in the default changelist or you don't select any of the files in the **Files** list, then an empty pending changelist is created. This is actually quite handy.

You should not select **Restrict access to changelist**. This is an advanced feature supporting scenarios that require restricted information communication.

The select pending changelist dialog

As we saw in the previous chapter, you can elect to make action requests such as **Check Out**, **Mark for Add**, and **Mark for Delete** prompt you to select a pending changelist to associate with the action.

If you want to allocate a new pending changelist select **New** from the **Add files to pending changelist:** dropdown and enter a description, as shown in the following screenshot:



Select Pending Changelist

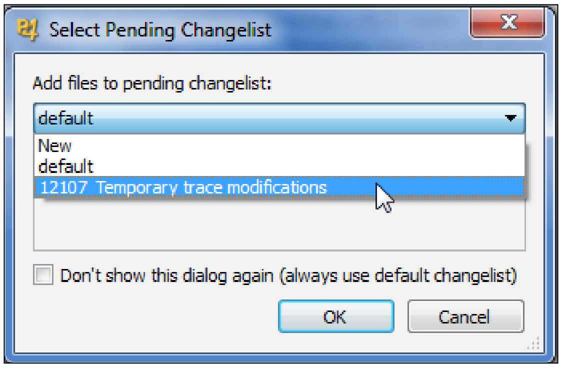
Add files to pending changelist: New

Changelist description

Temporary trace modifications

□ Don't show this dialog again (always use default changelist)

Or, if an appropriate changelist already exists in the **Add files to pending changelist:** dropdown you can select it as shown in the following screenshot:



Select Pending Changelist

Add files to pending changelist: default

New
default
12107 Temporary trace modifications

□ Don't show this dialog again (always use default changelist)

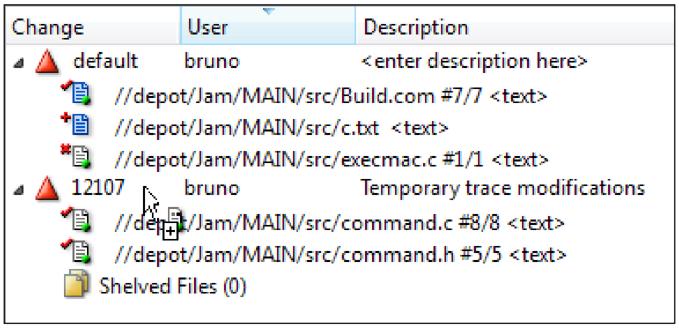
Unlike the general case of creating a new pending changelist, these dialogs do not involve the files already open in the default changelist. They only add the file you are checking out, marking for add, or marking for delete to the changelist selected.

If you previously selected **Don't show this dialog again**, you can always restore this dialog by going to the **Edit** | **Preferences...** | **Behavior** menu option and selecting **Prompt for changelist when checking out, adding, or deleting files**.

Moving files between pending changelists

Perhaps you've changed your mind about the changelist you want to associate an action with. Or maybe you want to consolidate the actions from several separate changelists into a single changelist. Not to worry, there are ways to move files between changelists.

P4V is a GUI. So, you can simply select the files you're interested in moving and then drag them to the target changelist row, with the (red) triangle icon, and drop them:

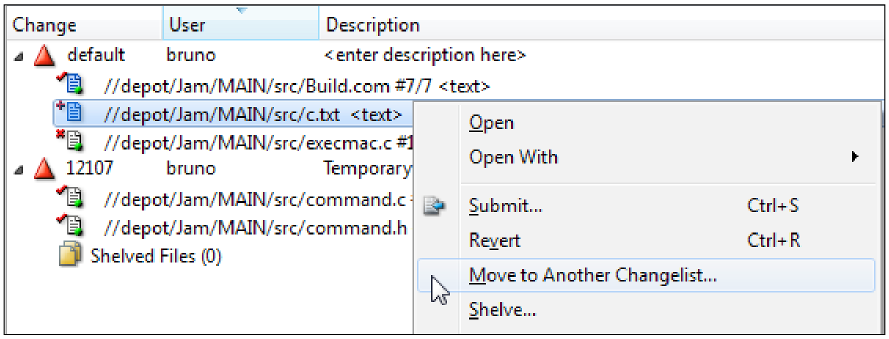


Change	User	Description
default	bruno	<enter description here>
//depot/Jam/MAIN/src/Build.com #7/7 <text>		
//depot/Jam/MAIN/src/c.txt <text>		
//depot/Jam/MAIN/src/execmac.c #1/1 <text>		
12107	bruno	Temporary trace modifications
//depot/Jam/MAIN/src/command.c #8/8 <text>		
//depot/Jam/MAIN/src/command.h #5/5 <text>		
Shelved Files (0)		

While drag-and-drop is convenient, it doesn't always provide you with the level of control that you desire. Equally, drag-and-drop may not fit your work style.

> Be careful where you drop files you're trying to move between changelists. If you drop them on top of other files you may launch the difference tool, which is the default action for file-to-file drag-and-drop.

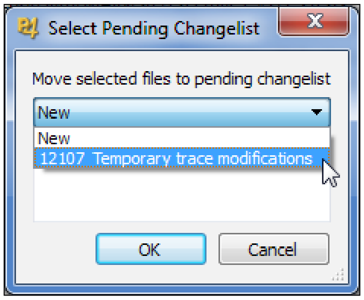
You can always select one or more files in the **Pending** changelists tab view and then select **Move to Another Changelist...** from the right-click menu, as shown in the following screenshot:



Change	User	Description
default	bruno	<enter description here>
//depot/Jam/MAIN/src/Build.com #7/7 <text>		
//depot/Jam/MAIN/src/c.txt <text>		
//depot/Jam/MAIN/src/execmac.c #1		
12107	bruno	Temporary
//depot/Jam/MAIN/src/command.c		
//depot/Jam/MAIN/src/command.h		
Shelved Files (0)		

Move to Another Changelist... is also found in tree pane right-click menus. One advantage of the right-click menus is that you can select multiple files from different changelists for consolidation activities.

After you've identified the files to move, you will be prompted with the **Select Pending Changelist** dialog as shown in the following screenshot:


Select Pending Changelist
Move selected files to pending changelist
New
New
12107 Temporary trace modifications
OK
Cancel

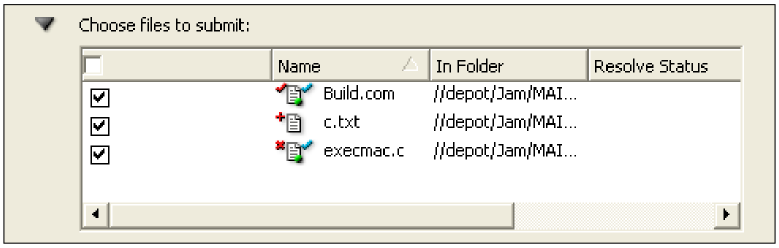
This is the same **Select Pending Changelist** dialog we've seen before. It just doesn't have a **Don't show this dialog again** check box.

> If you haven't done so already, now is a good time to follow along and experiment with the various ways to create new pending changelists and move files between them.

Dynamic organization at submit

There are times when you need to submit a subset of the files in a pending changelist. As we saw earlier in this chapter, creating a new pending changelist to select some files for submitting would usually be the best practice. However, there may be restraints that make new pending changelists impractical. For example, some third-party tools or IDE integrations that can interact with Perforce do not support multiple workspace changelists.

Not to worry, the submit dialog provides you with a mechanism for specifying selective submits. By default, all files in a changelist are selected when you request submit. However, you can click on the triangle next to the **Choose files to submit:** area to expose the selection list. Using the selection list you can explicitly identify the files that you want to submit. The exposed selection list for a submit dialog is shown in the following screenshot:


Choose files to submit:

□	Name	In Folder	Resolve Status
☑	Build.com	//depot/Jam/MAI...	
☑	c.txt	//depot/Jam/MAI...	
☑	execmac.c	//depot/Jam/MAI...	

To include a file in a submit you select the checkbox next to its name. To exclude a file from the submit you clear the checkbox next to its name. The checkbox at the top of the column is a check/uncheck all mechanism.

So what happens to files you don't select for submit? They are moved to the default changelist and remain open. If they were already in the default changelist they just remain open.

> Be careful about submitting a subset of the files in your pending changelist as it may make it hard for other users to re-create your work. This is the "works in my workspace" dilemma. Use the **Description:** to reflect the selective nature of the submit.

Limits to multiple pending changelists

The ability to create multiple pending changelists is potentially very useful. These pending changelists allow you to use one workspace to work on different tasks in parallel. However, there are limits to what you can do in a single workspace.

The key limitation is that a workspace cannot have the same file in two different pending changelists at the same time. This makes sense if you think about it. There is only one copy of the file on disk in your workspace. Therefore, you can't update that single copy differently in two different ways at the same time!

One common approach to addressing this limitation is to combine the previously independent changelists into a single changelist. This resolves the interdependence issue. However, it creates potential future issues if the files and modifications you have combined do not really belong together.

If you realize after starting the two tasks that they depend on each other, then another common approach is to work on the two tasks one after the other. Revert the files for the conflicting task, continue to work on the current task, and submit that changelist. After the submit check out the files for the next task, including any files which were in common with the previous task.

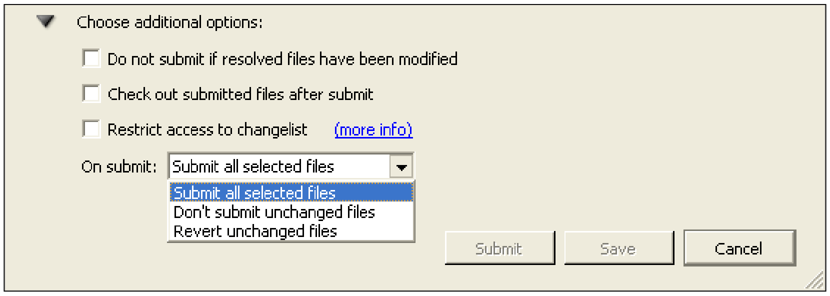
More advanced options are to: shelve a changelist (covered at the end of this chapter), or create a new workspace and do the other task in that (covered in *Chapter 6*, *Managing Workspaces*).

> Interdependence is a source control problem that has no magic silver bullet solution. Find the solution or set of solutions that best addresses the needs of your environment and stick with them. Always recognize that there will be exceptions that require extra effort to support.

File management at submit

Much of the time you're dealing with files effectively on an individual basis. However, submit is a time where you're working with files collectively.

Until now, we've ignored the **Choose additional options** section of the submit dialog. This section of the submit dialog deals with files collectively using the default changelist as a catch-all for files that don't conform to specific requirements. Now that we've covered multiple changelists it's time to look at these options, as outlined in the following screenshot:



Choose additional options:

□ Do not submit if resolved files have been modified

□ Check out submitted files after submit

□ Restrict access to changelist (more info)

On submit: Submit all selected files

The various scenarios are covered in the following subsections.

> Additional options only apply to an actual submit. If you **Save** from this dialog, a new numbered pending changelist is created, but the options won't persist to when you submit that pending changelist.

Handling unmodified files

So what is an unmodified file and how did it get that way? An unmodified file is a file that you have checked out but never changed. There are many ways that this might happen. Maybe a tool required that the file be checked out just in case. Or perhaps you checked it out thinking you would need to modify it. Ultimately the reasons aren't important. Unmodified files can and do exist in the course of normal development activities.

The **On submit:** dropdown provides you with the ability to handle unmodified files at the time of submit. You can achieve the same results with individual P4V actions prior to submit. However, dealing with them at submit time can be more efficient.

> The default **On submit:** action can be set in your workspace options as covered in *Chapter 6, Managing Workspaces*.

Submit all selected files as a choice submits all files whether or not they've been modified. If you submit using this option, the version number of unmodified files changes, even though there is no content change between versions. Occasionally you might have a set of files that you always want to submit together, even if some of the files have not changed. However, most of the time this option just creates unnecessary history for the unmodified files.

Don't submit unchanged files as a choice only applies to files that are checked out. If they have not changed then they remain checked out in the default changelist after submit. This is probably the most common choice for users, and the option we use ourselves.

Revert unchanged files as a choice reverts files that are checked out, but have not changed. After submit, the previously unmodified files will no longer be checked out.

Choosing between the latter two options is usually a matter of personal preference, find a style of working and stick to it!

Making multiple changes to the same set of files

It is quite common to make multiple changes to basically the same set of files. You are adding features one by one. It can be annoying to check out the files, make modifications, submit them, and then have to remember which files to check out again ready for the next change. This is particularly true if the files are in various different directories within the repository structure.

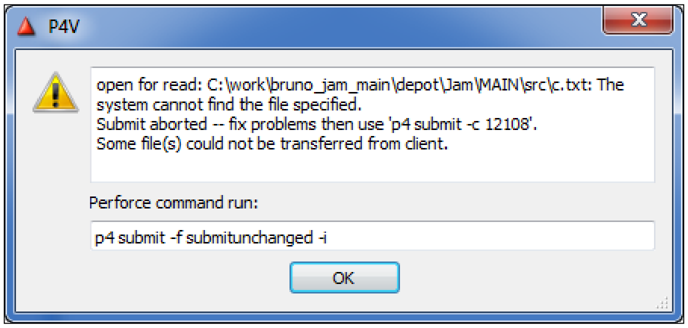
Using the **Check out submitted files after submit** selection solves this problem. After the successful submit, the files are immediately checked out again and will be in the default pending changelist.

Any files in the changelist that you marked for add will be checked out (for edit). Files you have marked for delete are gone and won't be checked out.

Failed submits

There are various reasons that the server will fail a submit request. Typically, these reasons relate to context inconsistencies that you may not be aware of. An example of inconsistency would be a file marked for add that was deleted from the workspace before submit. The server was expecting a file which was not available. That's a consistency problem that will cause a submit to fail.

If the submit fails, you will be presented with a warning dialog detailing the reason for the failure. A dialog for being unable to add a file would look like the following:



```
open for read: C:\work\bruno_jam_main\depot\Jam\MAIN\src\c.txt: The
system cannot find the file specified.
Submit aborted -- fix problems then use 'p4 submit -c 12108'.
Some file(s) could not be transferred from client.
```

Perforce command run:

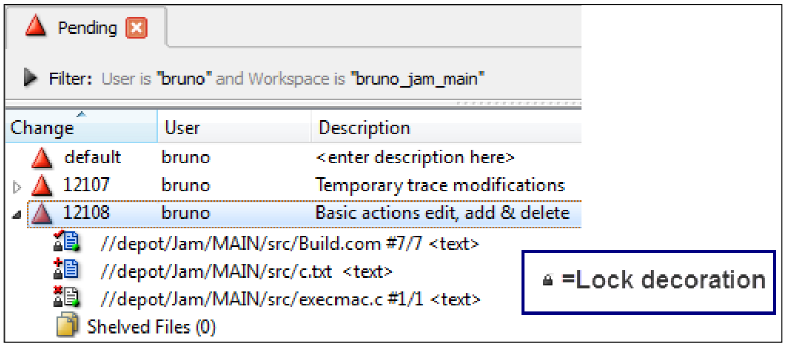
```
p4 submit -f submitunchanged -i
```

As you can see, there are a lot of details in the dialog. You can ignore most of them as they are intended for support or advanced users. However, even a general user should be able to identify that the cause of the failure was a missing file.

Remember when we mentioned in *Chapter 1*, *Getting Started with Perforce* that the submission of changelists is an atomic action? This is an example of where we see this happening. There was a problem with submitting one file in the changelist, so the whole changelist submission fails.

So what happens now? If we were submitting the default changelist then it will have been assigned a number. If the pending changelist already had a number, then that number will not have changed. What we should do is fix the problem and resubmit the changelist.

Let's look at the Pending tab view:



Filter: User is "bruno" and Workspace is "bruno_jam_main"

Change	User	Description
default	bruno	<enter description here>
12107	bruno	Temporary trace modifications
12108	bruno	Basic actions edit, add & delete

//depot/Jam/MAIN/src/Build.com #7/7 <text>
//depot/Jam/MAIN/src/c.txt <text>
//depot/Jam/MAIN/src/execmac.c #1/1 <text>
Shelved Files (0)

Notice that a lock decoration (identified in the callout box) has been added to bottom-left of the icons for all of the files in the changelist. This assumes you can resolve the problem quickly and re-submit the changelist. If you can't do this, then we suggest you unlock the files (from the right-click menu), to allow other people to submit changes. This may create conflicts, but those are normally easily resolved as we will cover in *Chapter 7, Dealing with Conflicts*.

Locking files does not prevent other people from checking those files out, but it does prevent them from submitting any changes. This can be disruptive to good teamwork.

When you have fixed the problems, don't forget to submit the changelist!

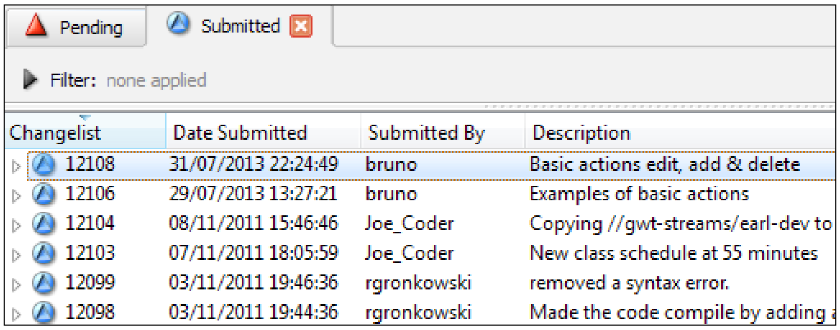
Changelist numbering

So far, we've avoided some of the details associated with changelist numbering. If you've been doing follow along, you're likely to have encountered some aspects of this. In this section we go a little deeper into the subject.

How submitted changelists are numbered

A key characteristic of changelist numbers is that changelists with a higher number have always been submitted after a changelist with a lower number. This creates a direct relationship between increasing numbers and the date/time of submission. This is very useful when it comes to reporting as we will see in *Chapter 5, File Information*.

If we look at the **Submitted** tab view with no filter, we can see a list of all submitted changelists:



Filter: none applied

Changelist	Date Submitted	Submitted By	Description
12108	31/07/2013 22:24:49	bruno	Basic actions edit, add & delete
12106	29/07/2013 13:27:21	bruno	Examples of basic actions
12104	08/11/2011 15:46:46	Joe_Coder	Copying //gwt-streams/earl-dev to
12103	07/11/2011 18:05:59	Joe_Coder	New class schedule at 55 minutes
12099	03/11/2011 19:46:36	rgronkowski	removed a syntax error.
12098	03/11/2011 19:44:36	rgronkowski	Made the code compile by adding

The relationship between changelist numbers and submit date is clear.

When a changelist is successfully submitted, the server ensures the number is the next changelist number in sequence. Exactly what this number will be is not always easy to predict, especially if you have hundreds or thousands of colleagues all submitting changes to the same Perforce server.

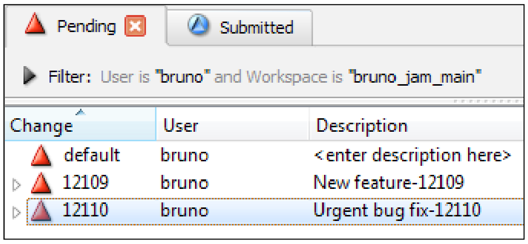
Implications for changelist numbering

The sharp-eyed reader will have noted gaps in the number sequence (for example, **12107** is missing) in the preceding **Submitted** tab view example. This is not a mistake.

Let's walk through a scenario to demonstrate how changelist numbering works in practice. You can follow along with the following sequence of actions:

1. Create two pending changelists with numbers
2. Submit them in reverse order, so the one with a higher number is submitted first
3. Take a guess as to what the numbers will be before you submit them (and then verify your guess!)

Let's have a look step-by-step:

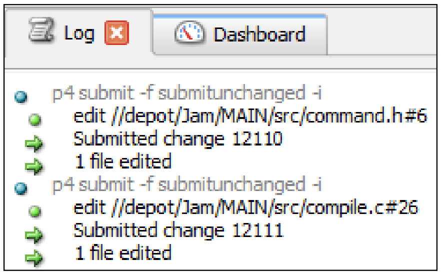


Filter: User is "bruno" and Workspace is "bruno_jam_main"

Change	User	Description
default	bruno	<enter description here>
12109	bruno	New feature-12109
12110	bruno	Urgent bug fix-12110

In the preceding diagram we are going to submit first changelist **12110** and then changelist **12109**. What do you think the numbers will be? What might change in this scenario if other users are also submitting changes to the server at this time?

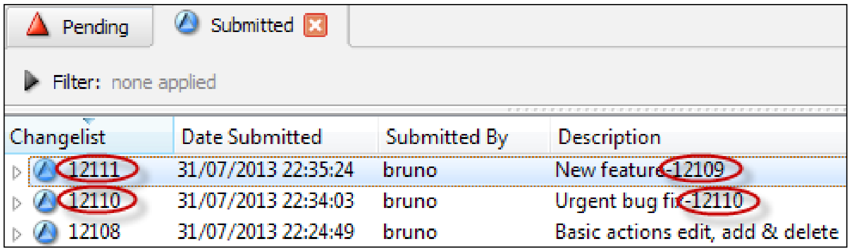
If we try this, then the results in the log window might be:



```
p4 submit -f submitunchanged -i
  edit //depot/Jam/MAIN/src/command.h#6
  Submitted change 12110
  1 file edited
p4 submit -f submitunchanged -i
  edit //depot/Jam/MAIN/src/compile.c#26
  Submitted change 12111
  1 file edited
```

If you don't see exactly this output, then go to **Edit** | **Preferences** | **Logging** settings. We have checked **Show p4 command output for file operations**.

So we see here that changelist **12110** was submitted with the same number, and yet changelist **12109** became **12111** when it was submitted. If you don't believe us, then you try it! The order the pending changelists was submitted in is shown in the submitted changelists tab:



Changelist	Date Submitted	Submitted By	Description
12111	31/07/2013 22:35:24	bruno	New feature-12109
12110	31/07/2013 22:34:03	bruno	Urgent bug fix-12110
12108	31/07/2013 22:24:49	bruno	Basic actions edit, add & delete

In our training classes, people are often a little surprised when we show them this scenario. If we ask for a show of hands before we do it, opinion can be evenly divided as to what changelist number will be assigned!

But what about the numbers assigned to the pending changelists you've been creating? Since these are pending changelists, there is no need for the numbers to meet the relationship criteria outlined earlier. When you submit a pending changelist, the server, if needed, will assign it a new reference number before submit completion.

Note that the precise number allocated by the server for a submitted changelist will vary depending on who else is submitting changelists at the same time. In the preceding scenario, pending changelist **12110** kept the same number when it was submitted. If someone else had submitted a changelist in the meantime, then it would have been renumbered as well.

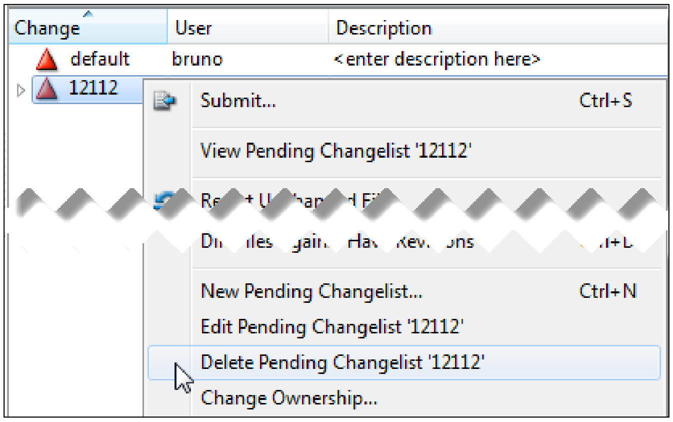
You may also have noticed that there is now a gap in the numbering sequence, changelist **12109** no longer exists (since it was pending but was renumbered on submission). This makes no difference to reporting, but you need to be aware of this behavior.

Don't rely on the number of pending changelist being the same after submit, it is likely to change. When you need a specific changelist number, P4V provides you with many ways to find that number.

Deleting pending changelists

Having created pending changelists, you may have moved files out of them and into others. There is no harm in having empty changelists. However, they do tend to create visual clutter within the P4V display, so you can delete empty changelists.

Select the changelist in the **Pending** tab view, right-click and select **Delete Changelist** as shown in the following screenshot:


Change
User
Description
default
bruno
<enter description here>
12112
Submit...
Ctrl+S
View Pending Changelist '12112'
New Pending Changelist...
Ctrl+N
Edit Pending Changelist '12112'
Delete Pending Changelist '12112'
Change Ownership...

This option is only available on the right-click menu if the pending changelist is empty (has no files in it).

> To delete a pending changelist that isn't empty, either move all its files to another pending changelist, or revert them.

Shelving

So far, we've seen that changes to files in your workspace are made available to other workspaces through the submit process. However, there are times when you want file changes to be available without having to submit those changes. One method for achieving this would be to manually recreate the desired changes. This of course would be time consuming and error prone. The best practice is to use Perforce's shelving feature.

Shelving associates your file changes with a changelist and creates a copy of the contents of the files on the server. With a changelist for reference and the server as a common storage area, any user can access the shelved changes and recreate them in another workspace. In effect, you are using the server as a shelf to exchange file changes.

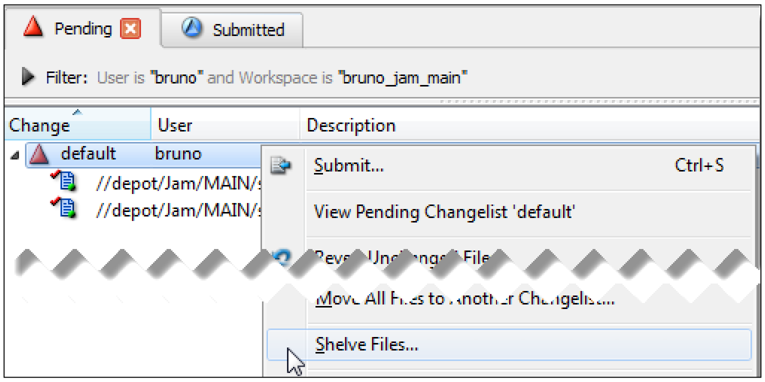
Common uses of shelving are as follows:

- Saving a copy of work in progress so that you can revert your current workspace to perform an urgent task. Then restoring the work in progress after the urgent task has been completed.
- Reviewing changes prior to a submit.
- Automated build validation prior to a submit.
- Transferring work in progress between workspaces, such as between office and home, or office and test station.

> Submit is an action that creates a permanent record that can be examined and traced. Shelving activities are temporary and create no permanent records.

Shelving files in a changelist

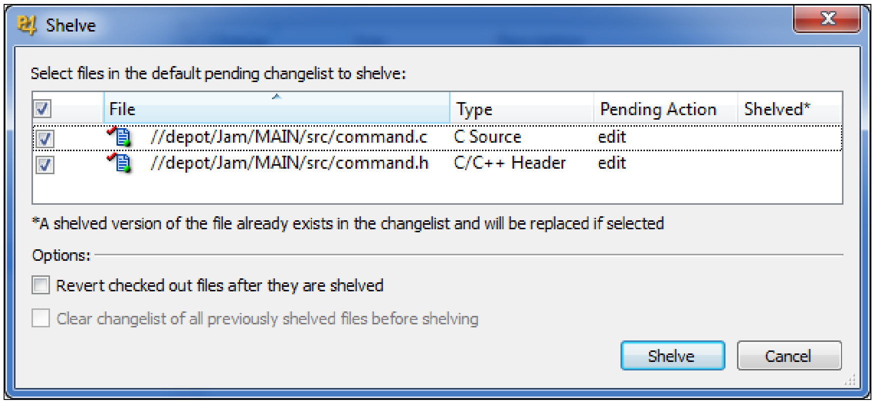
In order to shelve files, their associated activity (add, edit, delete, and so on) must already be associated with a pending changelist. To shelve a file, you select it in a pending changelist, then right-click and select **Shelve Files...** as we can see in the following screenshot:


Pending
Submitted
Submit...
Ctrl+S
View Pending Changelist 'default'
Revert Unchanged Files
Move All Files to Another Changelist...
Shelve Files...

Filter: User is "bruno" and Workspace is "bruno_jam_main"

Change	User	Description
default	bruno	
//depot/Jam/MAIN/		
//depot/Jam/MAIN/		

Making the shelve request will prompt you with a dialog that allows you to verify the files involved and to specify optional actions as we see:



Select files in the default pending changelist to shelve:

☑	File	Type	Pending Action	Shelved*
☑	//depot/Jam/MAIN/src/command.c	C Source	edit	
☑	//depot/Jam/MAIN/src/command.h	C/C++ Header	edit	

*A shelved version of the file already exists in the changelist and will be replaced if selected

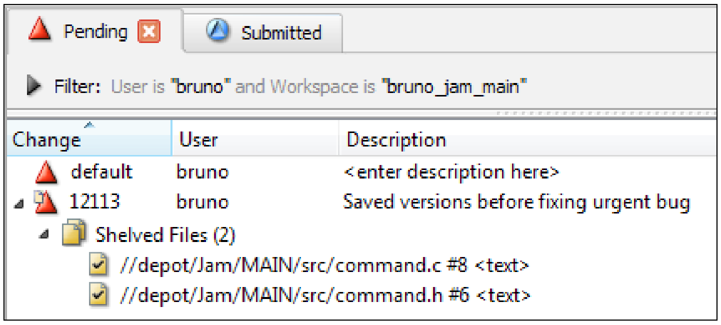
Options:

☐ Revert checked out files after they are shelved

☐ Clear changelist of all previously shelved files before shelving

The **Revert checked out files after they are shelved** option provides a one-step mechanism for saving work in progress and restoring a workspace. By reverting, you make your workspace clean so that changes can be made independently of work in progress that has now been shelved. The work in progress is restored when you unshelve the files.

The results of the shelve action are shown in the **Pending** tab, and might be as follows:



Filter: User is "bruno" and Workspace is "bruno_jam_main"

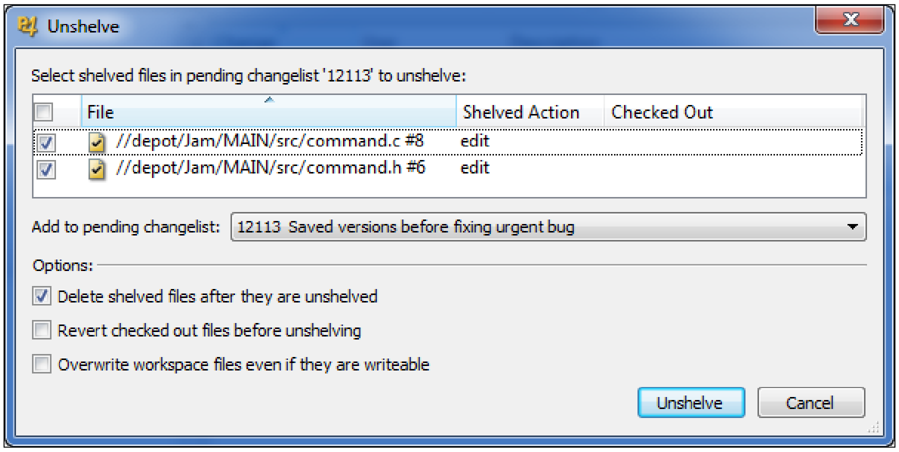
Change	User	Description
default	bruno	<enter description here>
12113	bruno	Saved versions before fixing urgent bug
Shelved Files (2)		
//depot/Jam/MAIN/src/command.c #8 <text>		
//depot/Jam/MAIN/src/command.h #6 <text>		

In the preceding example, the shelved files were initially associated with the default changelist. However, for reference reasons, shelved files require a numbered changelist. The shelving dialog sequence would have provided you with an opportunity to specify or create a numbered changelist to shelve the files.

[If you're following along, check out a file, modify it, then shelve it with and without the revert option. Then look at the right-click menu options for shelved files, including **Diff Against Source Revision** and **Diff Against Workspace File**. Make sure the contents are what you were expecting!]

Unshelving files

When you want to restore the contents of a workspace file to the shelved contents, you unshelve the file. The request to **Unshelve...** is available on the right-click menus for both changelists and individual files. This results in the following dialog:



Select shelved files in pending changelist '12113' to unshelve:

□	File	Shelved Action	Checked Out
☑	//depot/Jam/MAIN/src/command.c #8	edit	
☑	//depot/Jam/MAIN/src/command.h #6	edit	

Add to pending changelist: 12113 Saved versions before fixing urgent bug

Options:

☑ Delete shelved files after they are unshelved

□ Revert checked out files before unshelving

□ Overwrite workspace files even if they are writeable

The result will be that the action associated with the file when it was shelved (as shown in the **Shelved Action** column) will be recreated in the current workspace. The **Overwrite workspace files even if they are writeable** option allows you to override the normal protections provided by P4V. Use it with care as files overwritten can't be recovered. The other options provide for one-step support of delete and update activities that we'll discuss next.

Deleting shelved files

When the shelved file action no longer needs to be retained, the owner of the changelist must delete the shelved files from the changelist. A changelist with shelved files when right-clicked has a **Delete Shelved Files...** option. This allows you to delete them without having to unshelve the files first. There is also the **Delete shelved files after they are unshelved** option in the unshelve dialog.

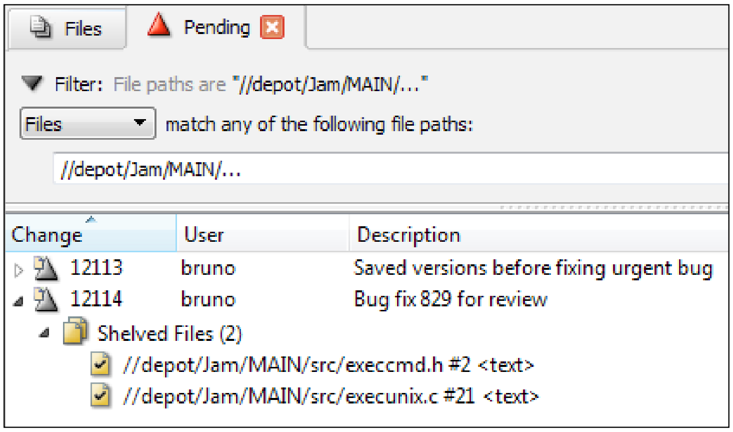
Keep in mind that shelved file content is transient. Deletion of a shelved file action without an unshelve discards any shelved file content. This may be what you want. However, it is rarely the desired result for the common use scenarios described previously.

> If you are the owner of the shelved changelist, but you aren't presented with delete capabilities, check your current workspace.

Finding shelved files

We initially said that a common use of shelved files was to pass them over to colleagues for code reviews. This implies that other people can unshelve files into a different workspace. However, first they need to be able to find the shelved changelist, this is done in the **Pending** changelists tab.

The default filter for this tab is to show pending changelists for the current user and workspace. To view another person's shelved changelists we need to modify the filter options as shown in the following screenshot:



Filter: File paths are "//depot/Jam/MAIN/..."

Files match any of the following file paths:

//depot/Jam/MAIN/...

Change	User	Description
12113	bruno	Saved versions before fixing urgent bug
12114	bruno	Bug fix 829 for review

Shelved Files (2)
- //depot/Jam/MAIN/src/execcmd.h #2 <text>
- //depot/Jam/MAIN/src/execunix.c #21 <text>

In this instance, Gale is looking to review Bruno's changelist **12114**. By changing the filter to be only on a depot file path, she can see the shelved changelist, and the files within it.

She can then use the changelist by right-clicking and then selecting the **Unshelve Files...** option, or she can open or diff the files directly from the right-click menu for the individual files.

Note that you may need to change your filter options in order to see shelved files in other workspaces, even other workspaces you own.

Modifying shelved files

A file can be both checked out for edit and shelved. So you might wonder what happens if you modify that file in your workspace. The answer is that you have a modified file in your workspace. The shelved contents of the file known to the server are unchanged until you shelve the file again. This also means that you need to explicitly unshelve files in different workspaces in order to get more recent versions of those file contents. There is no automatic update of shelved or unshelved file content.

Likewise, you may wonder what happens to a file that you unshelve and then modify in a different workspace. Let's call the original workspace X and the other one Y. If you are in workspace Y, the changelist from which you unshelve belongs to workspace X, so you are not allowed to shelve a file into it. You have two ways to transfer these changes to workspace X. The first option is to shelve the file in a new changelist in workspace Y, then unshelve files from that new changelist in workspace X. The other option is to submit the changes from workspace Y then, in workspace X, you perform a get latest to receive Y's changes (to keep things tidy you should also delete the shelved files from the original changelist).

> If you are doing follow along, now is a good time to experiment with the various characteristics of shelve and unshelve. Although they appear simple, shelving can lead to some very advanced usage scenarios. You should plan to revisit shelving after we cover workspaces in *Chapter 6, Managing Workspaces* and conflicts in *Chapter 7, Dealing with Conflicts*.

Managing shelved files

You can't submit a changelist that has a shelved file. If you want to submit the unshelved files of a changelist with shelved files, you can delete the shelved files or delete them as you unshelve them. You could also move the files that aren't shelved to a new or existing changelist and submit that changelist. This technique completes the submit and retains the shelved files in a changelist that others may be familiar with.

Shelved files consume server resources. Treat shelving as a transient activity. Be sure to delete when the shelved file activity is no longer of interest.

A workspace can have only one instance of a file open in any of the pending changelists associated with it. However, the same file can be shelved in more than one changelist associated with a workspace. Even with good changelist descriptions this can lead to confusion if you want to submit a version of the file. Best practice is to avoid this scenario.

Summary

Changelists are a fundamental tracking and organizational mechanism. Used haphazardly, they will be a source of aggravation and errors. However, following the basic best practice changelist techniques outlined in this chapter will make development, test, production, and support more efficient and less error prone.

In the next chapter we'll look at using the file information tracked by the server to answer questions common to development and support activities.

5
File Information

File information is the key to understanding the history of a repository, how a code base or set of files has evolved, and what is happening to it now.

File information is exposed in many parts of the P4V interface. In this chapter, we will cover how to make the most efficient use of the Perforce reporting commands to examine the information associated with a file. More importantly, we'll explain how to interpret this information to maximize the value it provides you.

In this chapter we will cover:

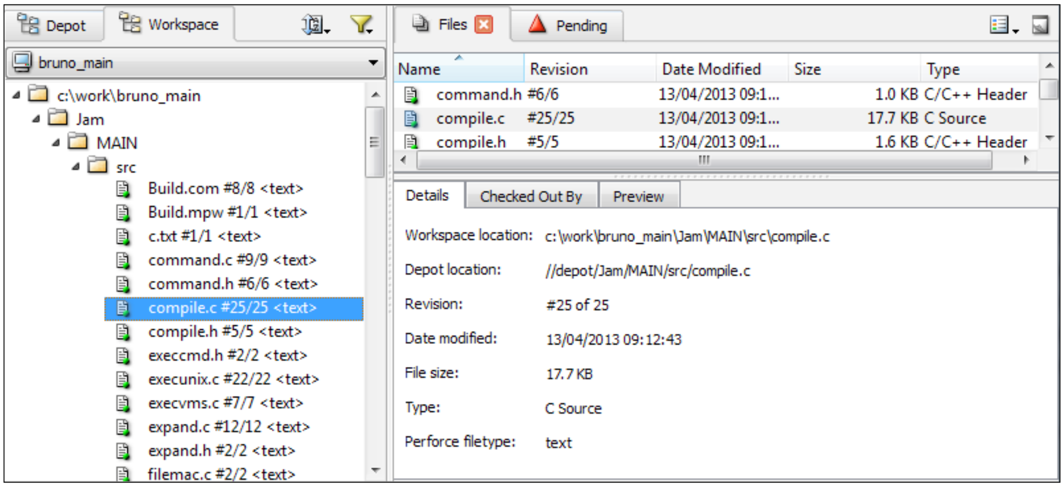
- Properties of files
- File versions and their relationship to changelists
- Finding files in the repository
- The many ways of referencing file versions
- Comparing different versions of files and folders
- Examining how file content has evolved over time
- Perforce file types and how they impact usage and workspaces

File properties

Every file has properties. Some of these properties relate to how files are represented in your workspace. Some relate to the history of the file. Still others will impact how Perforce can manage that file.

While various file properties are exposed in different parts of the P4V interface, the **Files** tab in the view pane provides you with the maximum amount of detail with the minimum effort.

The following screenshot shows a file selected in the tree panel along with the information that would be presented in the **Files** tab:



```
c:\work\bruno_main
  Jam
    MAIN
      src
        Build.com #8/8 <text>
        Build.mpw #1/1 <text>
        c.txt #1/1 <text>
        command.c #9/9 <text>
        command.h #6/6 <text>
        compile.c #25/25 <text>
        compile.h #5/5 <text>
        execcmd.h #2/2 <text>
        execunix.c #22/22 <text>
        execvms.c #7/7 <text>
        expand.c #12/12 <text>
        expand.h #2/2 <text>
        filemac.c #2/2 <text>
```

Name	Revision	Date Modified	Size	Type
command.h	#6/6	13/04/2013 09:1...	1.0 KB	C/C++ Header
compile.c	#25/25	13/04/2013 09:1...	17.7 KB	C Source
compile.h	#5/5	13/04/2013 09:1...	1.6 KB	C/C++ Header

Workspace location:	c:\work\bruno_main\Jam\MAIN\src\compile.c
Depot location:	//depot/Jam/MAIN/src/compile.c
Revision:	#25 of 25
Date modified:	13/04/2013 09:12:43
File size:	17.7 KB
Type:	C Source
Perforce filetype:	text

The **Files** tab presentation has two primary sections. The upper section presents information for the files in a folder using a tabular format. This provides a useful, easy to use, and comparative presentation. The lower section provides information specific to a single file selected in the upper tabular section. This provides you with the ability to focus on additional details once you've identified files of interest using the tabular upper section.

The tabular section presents a selected set of the information available for a file. As you'll see shortly, the information selected is under your control.

The **Details** tab provides all of the available information, and shows in addition the explicit mapping between the **Workspace** and **Depot** locations of the file. These locations can be seen in various parts of the P4V interface.

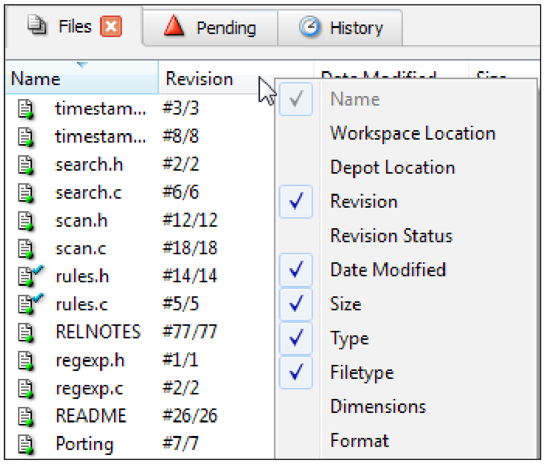
The **Checked Out By** tab provides details about all of the workspaces that have a file checked out. These are the full details. You will see summaries of this information in the tree panel hover tips.

The **Preview** tab shows the contents of text files. This is particularly useful when browsing repository files that are not currently present in your workspace.

Customizing the tabular display

While the tabular section of the **Files** tab is useful, without order it is still just a collection of data. P4V allows you to sort on any of the displayed columns. It also allows you to select which columns are displayed to help you focus on important factors.

You can reorder the columns by clicking on a column header, dragging it, and dropping it in the position you desire. You can also select the column displayed in the table by right-clicking on the header bar and then checking the columns you want displayed, as shown in the following screenshot:



Name	Revision
timestam...	#3/3
timestam...	#8/8
search.h	#2/2
search.c	#6/6
scan.h	#12/12
scan.c	#18/18
rules.h	#14/14
rules.c	#5/5
RELNOTES	#77/77
regexp.h	#1/1
regexp.c	#2/2
README	#26/26
Porting	#7/7

Note the triangle indicator in the **Name** column header. This specifies that the table entries are sorted based on the value in the **Name** column. A triangle pointing upwards sorts in ascending order, a triangle pointing downwards sorts in descending order.

Sorting on either the **Latest Changelist** or **Date Last Submitted** columns allows you to quickly identify more recently or least recently changed files.

Explaining the # characters

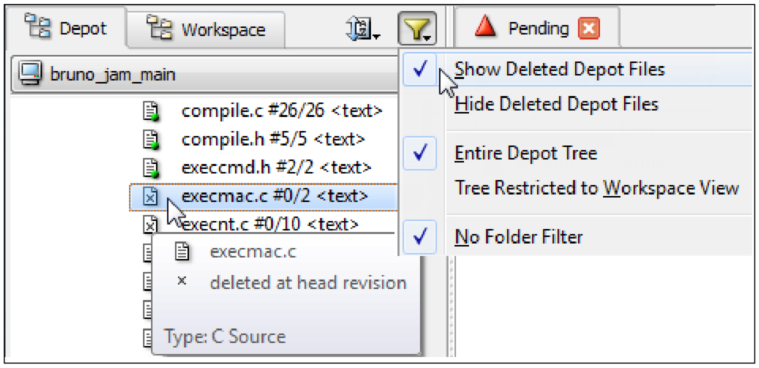
The # character is the Perforce notation indicating the file revision. So `foo.txt#3` indicates revision (or version) 3 of the file `foo.txt`. Using this notation saves a lot of screen space and reduces visual clutter. It's also the versioning notation you'll see in the log panel or which you would use on the command line.

But what about things like `#5/5`, `#3/5`, `#0/5` or even `#25 of 25`? Not to worry, long division and set theory are not involved. This is simply a notation that indicates the version of the file in your workspace and the maximum number of file versions the server knows about. So `#5/5` would indicate that you have version 5 of the file in your workspace and that is the most up-to-date version the server knows about. Likewise, `#3/5` would indicate that you have version `3` of the file in your workspace, yet the server knows about two more recent versions: `4` and `5`. While `#0/5` indicates that you have no version of the file in your workspace. Finally, the `#25 of 25` notation has replaced the "/" with "of" to increase readability.

> 0 is the workspace version of deleted files and files marked for add, but not yet submitted.

Showing deleted files

The deleted files are visible in the depot tree only if you turn on the filter to show them, as we see in the following screenshot:



The filter **Show Deleted Depot Files** has been checked. As a result, we can see a file **execmac.c** where the latest revision is **2**, and the tooltip (as well as the icon) shows us that it is **deleted at head revision**. As we will discuss later in this chapter, the head revision is the latest revision in the repository.

This filter is not available if you have selected the **Workspace** tab since deleted files do not exist locally.

Type and filetype

P4V understands two separate type concepts relating to files. Knowing the difference and how they apply will come in handy.

One concept is known as **type** and refers to an association based on the file name extension. This is managed by the operating system on the local workstation. For example, Windows might associate the `.docx` extension with the Microsoft Word application. The server maintains no information about type. By default, this relationship is used by P4V to select the tool to use when it opens a file for you. For example, if you ask P4V to open a file called `foo.html` and you have a web development tool installed, it may launch that tool. If you don't have such a tool, then it might launch a browser. Not to worry, you can override this behavior. We'll cover this and other productivity topics in *Chapter 10, The P4V User Experience*.

The other concept is known as **filetype**, and refers to the information Perforce uses to define workspace population characteristic, control P4V processing, and define server storage parameters. The server maintains filetype information on a per-revision basis. When a file is first added it is assigned a filetype. That filetype persists until it is explicitly changed. Most of the time, you will not need to worry about specifying a particular filetype. P4V uses several techniques to assure that the appropriate filetype is established when a file is first added (including defaults configured by your administrator). Of course you can always explicitly specify a filetype, but that is for advanced users.

The two basic filetypes are text and binary. Perforce knows that text files have line endings. When it populates your workspace with a text file it automatically adapts the line endings within the file to the encoding appropriate for your operating system. It is also understood that text files can be compared in ways that humans can understand. On the other hand, binary files are treated as a collection of bytes. They receive no special processing when populating your workspace, and it is understood that they can't usually be compared in ways that humans would understand.

> Filetype can also specify other attributes of files, such as making files always writable in workspaces. They can also set the executable bit on Linux/Unix for shell scripts or executable binaries. Full details on filetypes are documented at the end of the Command Reference Guide (See *Appendix B*, *Command Line* for how to access this).

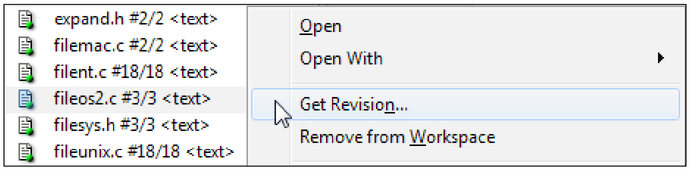
Understanding file versions and history

In *Chapter 4, Changelists*, we saw how to submit a modification to a file as a new revision, and how submitted changelists were numbered. In this chapter, we are going to first look at how to get older revisions of files into our workspace and how to understand the history of a file or a set of files. That includes understanding how file revisions relate to changelists, and the state of the repository as of a particular date and time.

We often need to have a look at older versions of files. When looking at a project, you may need to reproduce an older release, or understand a particular baseline which is a set of files or a complete folder or tree.

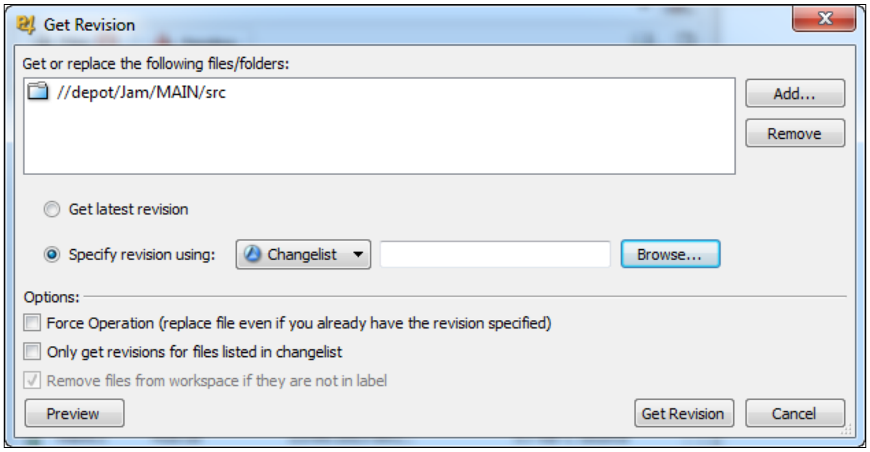
Getting different revisions of files

On the right-click menu for a file or folder there is a **Get Revision...** option, as shown in the following screenshot:



```
expand.h #2/2 <text>
filemac.c #2/2 <text>
filent.c #18/18 <text>
fileos2.c #3/3 <text>
filesys.h #3/3 <text>
fileunix.c #18/18 <text>
```

Clicking on the preceding option brings up the following dialog:



Get Revision

Get or replace the following files/folders:

//depot/Jam/MAIN/src

○ Get latest revision

● Specify revision using: Changelist

Options:

□ Force Operation (replace file even if you already have the revision specified)

□ Only get revisions for files listed in changelist

☑ Remove files from workspace if they are not in label

Various actions are possible with this dialog. For now, we will focus on choosing between **Get latest revision** and **Specify revision using:**.

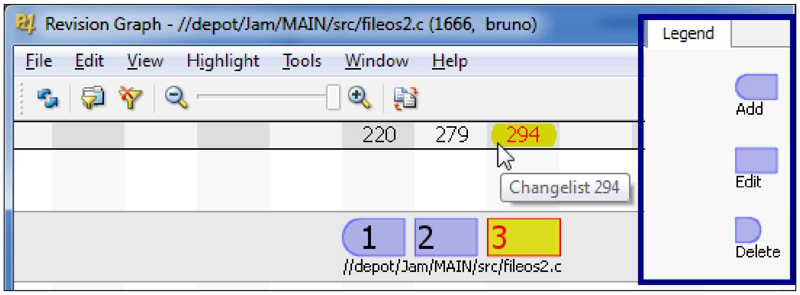
Get latest revision updates the workspace files with the highest revision of the files known to the server. This is also known as getting or syncing to the **head** revision. The head version is a special version with its own reference `#head`. You will see `#head` in the log pane and in various error and status reports.

Specify revision using: updates the workspace files with either explicit revisions or revisions that are implied from a context such as date/time, label, or changelist.

> Don't worry that you don't know what all of the **Specify revision using:** choices are or what they do. Take a moment to scan through the choices. Note the **Browse…** results. They are designed to reduce typing errors to get you the specific revision you're interested in.

How file revisions relate to changelists

The following screenshot shows the history of `fileos2.c` using the **Revision Graph** tool, which is available on the right-click menu for the currently selected file within the tree pane (right-click, then select **Revision Graph**). Note that for clarity we have turned off various panes within the tool to show the minimum necessary, and included the **Legend** option to explain the shapes:


Revision Graph - //depot/Jam/MAIN/src/fileos2.c (1666, bruno)
File Edit View Highlight Tools Window Help
220
279
294
Changelist 294
1
2
3
//depot/Jam/MAIN/src/fileos2.c
Legend
Add
Edit
Delete

This shows us the association between revisions of a file and the changelists in which those revisions were submitted to the repository. In this example, revision `3` was part of changelist `294`, and revision `2` was part of changelist `279`. The file was first created and thus added to the repository as revision `1` in changelist `220`.

It is fairly clear that a **Get latest** would give us revision `3` of the file in our workspace. Similarly, if we **Specify revision using:** changelist `279` we would get revision `2` of the file.

So what would you expect if we chose changelist `280`? Doing a get which specifies a changelist gives us the state of the repository at the point of that changelist. In this case, it will give us revision `2`. Revision `2` is the latest revision up to or including the specified changelist. In fact for this file, getting any changelist between `279` and `293` inclusive, will give us revision `2`.

What would we expect if you do a get which specifies changelist `100`? The preceding history shows us that this file did not exist in the repository as of that changelist. Therefore, a get of that changelist would remove the file from our workspace because it didn't exist as of that changelist. Using Perforce terminology, we would have revision 0, or the revision which does not exist, in our workspace.

> Removing version 0 files from a workspace is designed to avoid the inevitable user errors that would occur if you had to delete the file yourself.

Potentially surprising get revision results!

Following on from the preceding examples, what would you expect if you tried to get changelist `9999999` or some equally large number, for which a changelist doesn't yet exist? Hopefully, it is not a big surprise that we just get the latest revision of the file. In the preceding case, revision `3`. As we saw previously, even though the changelist doesn't directly contain this file, the file still associates revisions with every changelist, even those that don't yet exist.

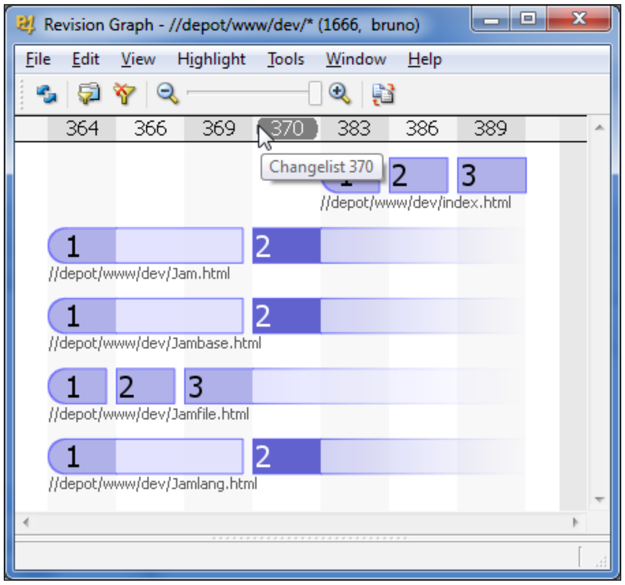
> This large changelist technique is actually just an obscure version of get latest. Obscure is not usually a good engineering practice.

But watch out! What will happen if we try to get revision `5` of this file? For this file, revision `5` does not exist. Therefore the get request treats this as if you requested revision `0` so it will remove the file from the workspace.

> Be very careful getting more than one file as of a specific version number. When more than one file is involved, this gets that version of every file that has that many versions or more and removes (#0) the rest. This is usually not what you want.

Changelists and folders

People don't always realize that the **Revision Graph...** right-click option can also apply to folders. The folder view shows all of the files at one time, as seen in the following screenshot:



Revision Graph - //depot/www/dev/* (1666, bruno)
File Edit View Highlight Tools Window Help
364 366 369 370 383 386 389
Changelist 370
2 3
//depot/www/dev/index.html
1 2
//depot/www/dev/Jam.html
1 2
//depot/www/dev/Jambase.html
1 2 3
//depot/www/dev/Jamfile.html
1 2
//depot/www/dev/Jamlang.html

In the preceding screenshot, the cursor is hovering over changelist `370`, and the 3 files in the folder that had new versions created by that changelist are highlighted.

What would we get if we synced the entire folder to changelist `370`? By default, we would get the files in that particular changelist and the version for every other file in the folder as of that changelist. In the preceding screenshot that would be version `3` of `Jamfile.html`, which was submitted in changelist `369`. But, `index.html` at version 0 would be removed from the workspace because it didn't exist until changelist `383`.

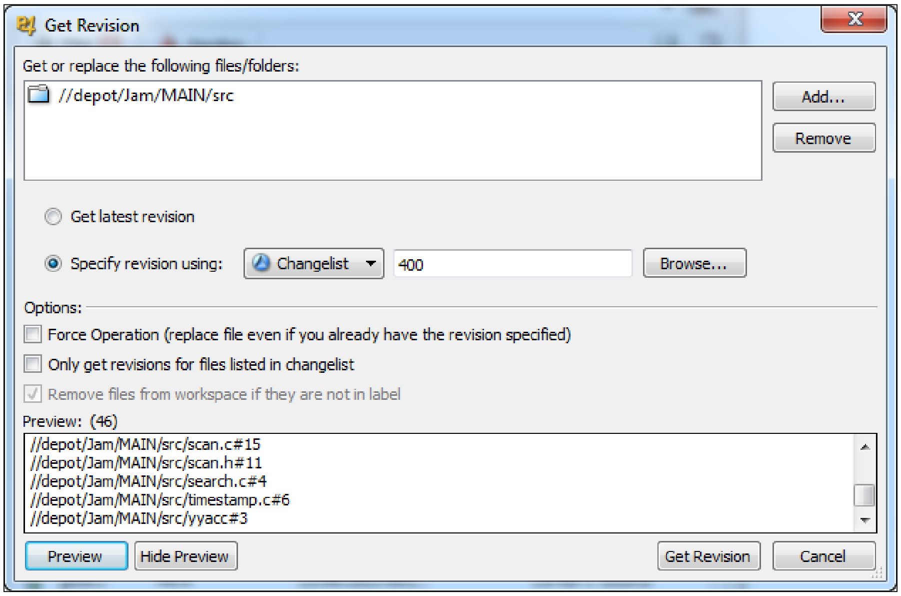
Therefore, we can see that a changelist can be used in two ways:

- To refer to the set of files submitted in that changelist
- To refer to the state of the entire repository (or any part of it such as a folder), up to and including that changelist

When waxing philosophical, we sometimes refer to this as the particle nature versus the wave nature of changelists!

Get revision options

We haven't yet covered the options in the **Get Revision...** dialog shown in the following screenshot. They provide functionality that can be difficult to achieve using other techniques.



Get or replace the following files/folders:

//depot/Jam/MAIN/src

○ Get latest revision

● Specify revision using: Changelist 400

Options:

□ Force Operation (replace file even if you already have the revision specified)

□ Only get revisions for files listed in changelist

☑ Remove files from workspace if they are not in label

Preview: (46)

```
//depot/Jam/MAIN/src/scan.c#15
//depot/Jam/MAIN/src/scan.h#11
//depot/Jam/MAIN/src/search.c#4
//depot/Jam/MAIN/src/timestamp.c#6
//depot/Jam/MAIN/src/yyacc#3
```

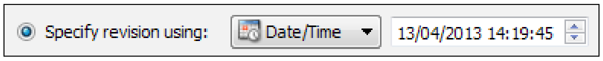
The preceding screenshot shows the output on clicking **Preview**. We see what the results would be, without actually affecting anything in our workspace. This can help avoid surprises and perhaps prevent unnecessary mistakes.

The **Force Operation (replace file even if you already have the revision specified)** checkbox, if checked, will update the files in our workspace to the specified revisions, even if P4V thinks we already have those revisions. You need to force operations when files have been removed from the workspace without coordinating that removal with Perforce.

Following on from the discussion in the previous section about a **Get Revision**, always getting files up to or including the changelist, we can modify this behavior by checking **Only get revisions for files listed in changelist**. With this option you update your workspace with only the file versions created by that changelist. If the following changelist is discovered to cause a problem, then it can be very useful to get a workspace into the state immediately before that changelist.

Referencing a specific date and/or time

The **Get Revision** dialog allows us to specify a date and time, as shown in the following screenshot:



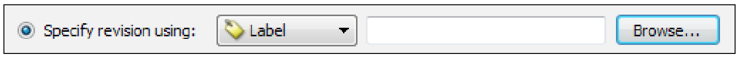
Specify revision using: Date/Time 13/04/2013 14:19:45

Every submitted changelist has an associated date and time of submission. The value specified in this dialog will give the same results as if you were to do a get specifying the changelist that was submitted precisely at or most closely before the specified date and time.

> Most people prefer to reference using changelist numbers rather than date and time. Changelists are unique and unambiguous. More than one changelist may be submitted within the one second granularity of date and time tracking. And communicating date and time across time zones is error prone at best.

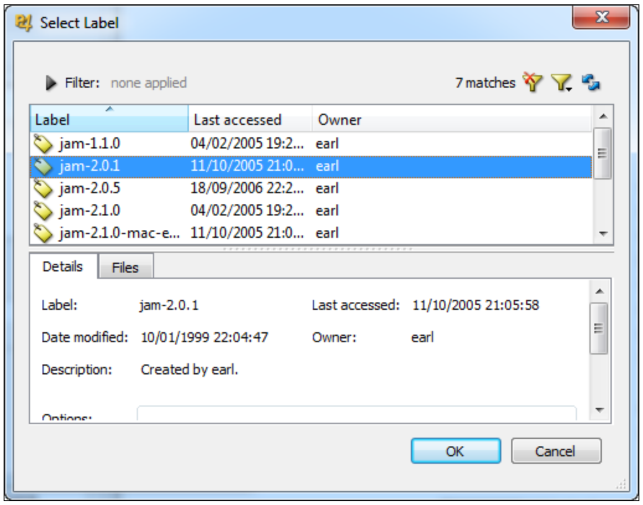
Referencing a label

As stated in the preceding section, we can also do a get relative to a particular label:



Specify revision using: Label

You can type the name of the label in directly. However, most people prefer to select an appropriate label from the choices presented when you click on the **Browse...** button.



Select Label

Filter: none applied

7 matches

Label	Last accessed	Owner
jam-1.1.0	04/02/2005 19:2...	earl
jam-2.0.1	11/10/2005 21:0...	earl
jam-2.0.5	18/09/2006 22:2...	earl
jam-2.1.0	04/02/2005 19:2...	earl
jam-2.1.0-mac-e...	11/10/2005 21:0...	earl

Label: jam-2.0.1

Last accessed: 11/10/2005 21:05:58

Date modified: 10/01/1999 22:04:47

Owner: earl

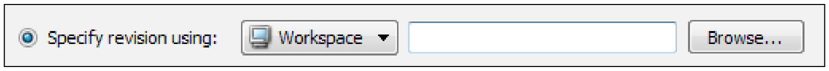
Description: Created by earl.

Options:

Most Perforce users don't use labels. They use changelists, the latest versions of branches or streams as we will cover in *Chapter 9, Perforce Streams* and *Chapter 10, The P4V User Experience*.

Files in another workspace

You can also reference file versions in another workspace:

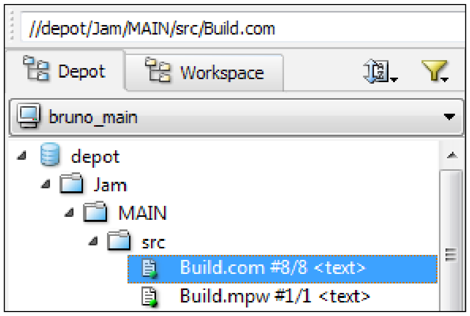


As previously stated, we can type the workspace name in the field or browse to select a workspace.

This feature is most useful when you're trying to resolve scenarios where it works/fails in my workspace. Most people use it with **Preview** so that they don't impact their current workspace. However, it does require that both workspaces reference the same depot files. This is only likely to be true for continuous build and code review environments.

Depot paths

The path of currently selected files or folders is shown in the address bar at the top of the screen, as per the following screenshot:


//depot/Jam/MAIN/src/Build.com

You can turn off the address bar by right-clicking on it and unchecking the option.

There is also a shortcut key (*Ctrl* + *C*) which copies the full path of the currently selected file or folder to the clipboard. This can be useful for documentation purposes. Depending on whether you have the depot or workspace tab showing, you might get:

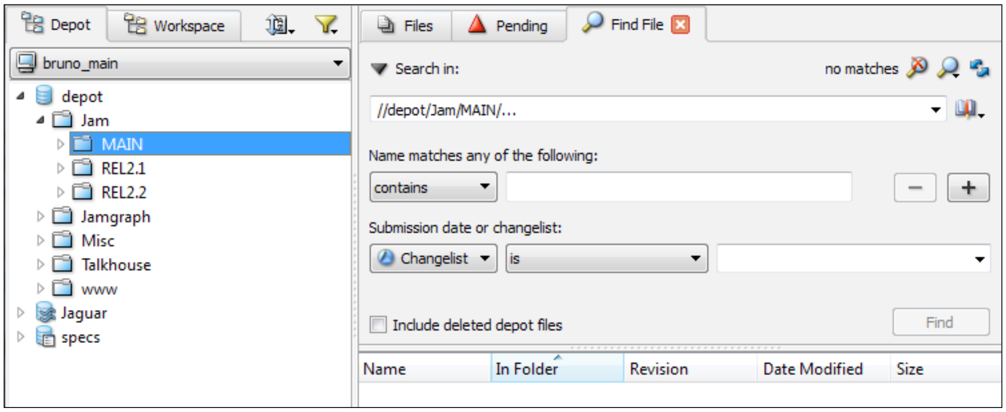
```
//depot/Jam/MAIN/src/Build.com
```

Or, depending on your workspace definition:

```
c:\work\bruno_main\Jam\MAIN\src\Build.com
```

Finding files – an introduction to wildcards

It is easy to find files with certain characters in their names by going to the **Search | Find File...** menu option. This will give you the **Find File** tab, as shown in the following screenshot:



Search in: //depot/Jam/MAIN/...

Name matches any of the following: contains

Submission date or changelist: Changelist is

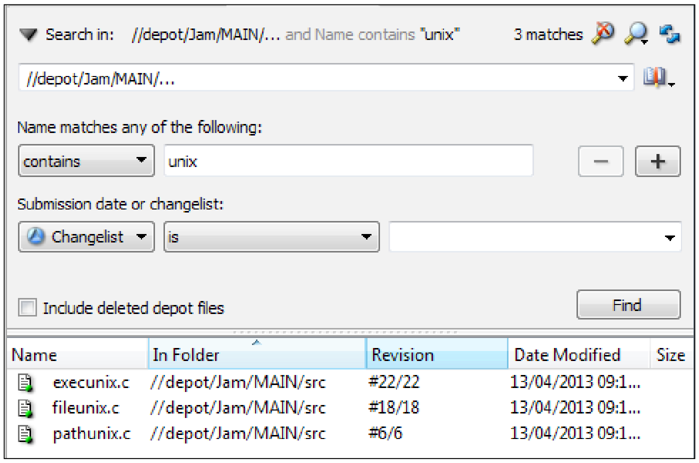
□ Include deleted depot files

In the preceding screenshot, we had a particular folder selected, and that is copied to the **Search in:** field. Sharp eyed readers will have noticed that the full path contains some extra characters: `//depot/Jam/MAIN/...`. In this instance, P4V has added the `...` wildcard to the directory path (this wildcard is also known as an ellipsis).

This wildcard means that all subfolders should be included in the search. We will look at more wildcard options in the next section of this chapter.

> You can also drag-and-drop directory paths from the tree pane on the left-hand side to the **Search in** field and the contents will be appropriately updated.

If we enter, for example, `unix` into the **Names matches any of the following** field and click on **Find**, we might get results as shown in the following screenshot:



Search in: //depot/Jam/MAIN/... and Name contains "unix" 3 matches

//depot/Jam/MAIN/...

Name matches any of the following:

contains: unix

Submission date or changelist:

Changelist: is

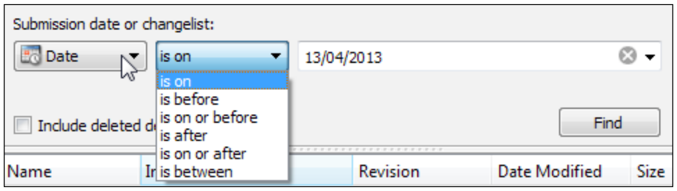
☐ Include deleted depot files

Name	In Folder	Revision	Date Modified	Size
execunix.c	//depot/Jam/MAIN/src	#22/22	13/04/2013 09:1...	
fileunix.c	//depot/Jam/MAIN/src	#18/18	13/04/2013 09:1...	
pathunix.c	//depot/Jam/MAIN/src	#6/6	13/04/2013 09:1...	

In this particular example, we can see that there are three files in that folder (or in any subfolders) containing `unix` in their filename.

By default, the results do not include any files where the latest revision is marked as deleted. You can change this by checking **Include deleted depot files**.

Note that it is possible to filter the results further using the **Submission date or changelist option:**, for example:

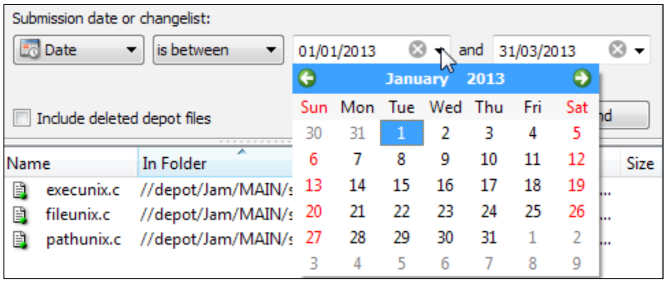


Submission date or changelist:

Date: is on: 13/04/2013

☐ Include deleted d

When working with dates you can search for particular periods of time, as seen here:



Submission date or changelist:

Date: is between: 01/01/2013 and 31/03/2013

☐ Include deleted depot files

Name	In Folder
execunix.c	//depot/Jam/MAIN/s
fileunix.c	//depot/Jam/MAIN/s
pathunix.c	//depot/Jam/MAIN/s

We invite you to explore this powerful feature on your own!

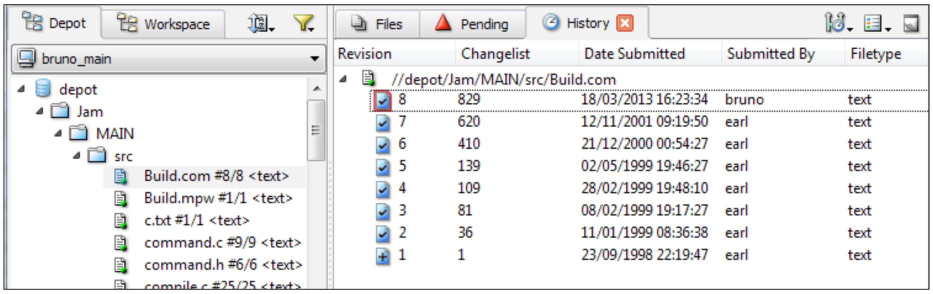
Showing history

The history of both files and folders can be seen via the **History** tab. You can go to the **View** | **History** menu option or click on the appropriate toolbar icon.

This tab can show the history for the currently selected file or folder in the tree pane. You can right-click on a file in the tree pane and select **File History**. If you have a folder selected the menu option is **Folder History**.

File history

The history for a file might be:



- depot
  - Jam
    - MAIN
      - src
        - Build.com #8/8 <text>
        - Build.mpw #1/1 <text>
        - c.txt #1/1 <text>
        - command.c #9/9 <text>
        - command.h #6/6 <text>
        - compile.c #25/25 <text>

//depot/Jam/MAIN/src/Build.com

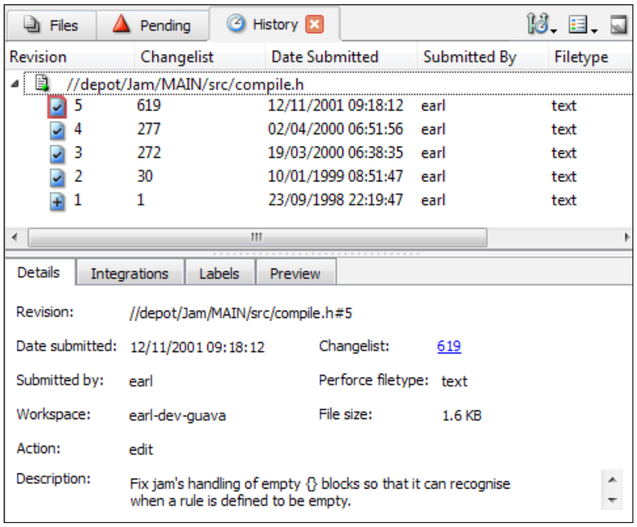
Revision	Changelist	Date Submitted	Submitted By	Filetype
8	829	18/03/2013 16:23:34	bruno	text
7	620	12/11/2001 09:19:50	earl	text
6	410	21/12/2000 00:54:27	earl	text
5	139	02/05/1999 19:46:27	earl	text
4	109	28/02/1999 19:48:10	earl	text
3	81	08/02/1999 19:17:27	earl	text
2	36	11/01/1999 08:36:38	earl	text
1	1	23/09/1998 22:19:47	earl	text

In this example, we can see the basic history information for the selected file. For each revision we see information such as **Changelist** and **Date Submitted**. As previously seen with the **File Properties** tab, by right-clicking the column headers you can select other columns to be displayed.

When this tab is displayed, clicking to select another file in the tree pane will cause the tab to be updated dynamically.

The icon for revision `8` in the preceding screenshot has a highlight around it, this shows that this revision is the one currently present in the workspace.

The **History** tab also has a details tab which can be shown if you wish, by dragging the splitter bar upwards with the mouse, as shown here:



Revision	Changelist	Date Submitted	Submitted By	Filetype
//depot/Jam/MAIN/src/compile.h				
5	619	12/11/2001 09:18:12	earl	text
4	277	02/04/2000 06:51:56	earl	text
3	272	19/03/2000 06:38:35	earl	text
2	30	10/01/1999 08:51:47	earl	text
1	1	23/09/1998 22:19:47	earl	text

Revision: //depot/Jam/MAIN/src/compile.h#5

Date submitted: 12/11/2001 09:18:12 Changelist: 619

Submitted by: earl Perforce filetype: text

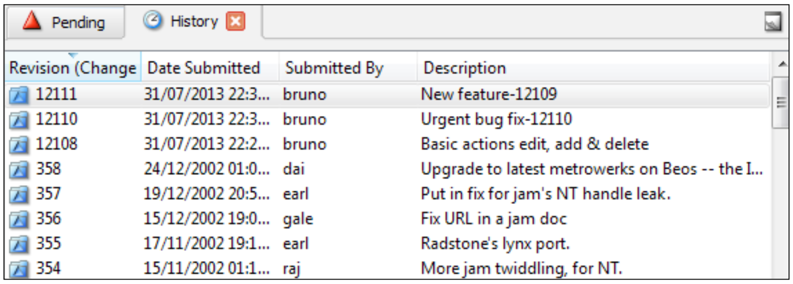
Workspace: earl-dev-guava File size: 1.6 KB

Action: edit

Description: Fix jam's handling of empty {} blocks so that it can recognise when a rule is defined to be empty.

Folder history

For a folder, the **History** tab shows all changelists that have affected files in that folder or any of its subfolders. This acts in the same way for a depot (which is like a top level folder, as discussed in *Chapter 1*, *Getting Started with Perforce*). An example is shown here:



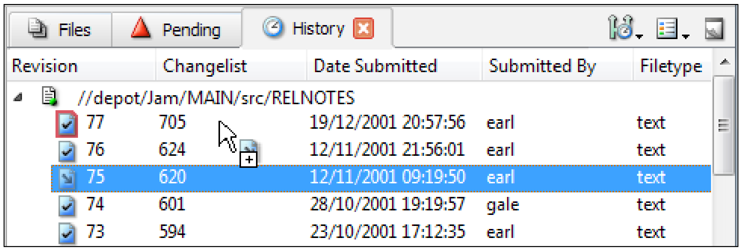
Revision (Change	Date Submitted	Submitted By	Description
12111	31/07/2013 22:3...	bruno	New feature-12109
12110	31/07/2013 22:3...	bruno	Urgent bug fix-12110
12108	31/07/2013 22:2...	bruno	Basic actions edit, add & delete
358	24/12/2002 01:0...	dai	Upgrade to latest metrowerks on Beos -- the I...
357	19/12/2002 20:5...	earl	Put in fix for jam's NT handle leak.
356	15/12/2002 19:0...	gale	Fix URL in a jam doc
355	17/11/2002 19:1...	earl	Radstone's lynx port.
354	15/11/2002 01:1...	raj	More jam twiddling, for NT.

Notice that there is no column for revisions as they only apply to files, not to folders.

If you drag-and-drop a column header onto another one (left or right), you can change the order they are displayed in.

Showing differences between file revisions

The easiest way to compare two revisions of a file is to drag-and-drop one revision onto another from the **History** tab for a file, as shown here:



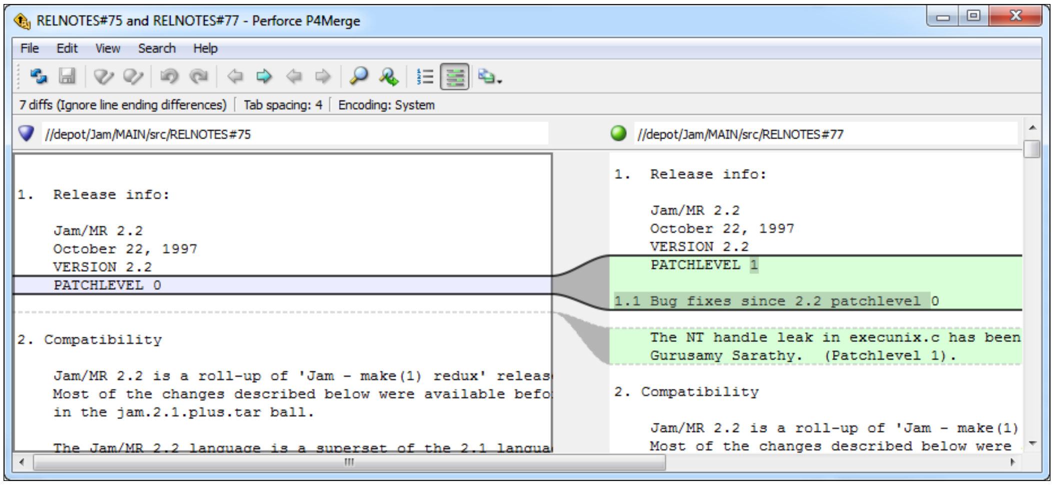
Revision	Changelist	Date Submitted	Submitted By	Filetype
//depot/Jam/MAIN/src/RELNOTES				
77	705	19/12/2001 20:57:56	earl	text
76	624	12/11/2001 21:56:01	earl	text
75	620	12/11/2001 09:19:50	earl	text
74	601	28/10/2001 19:19:57	gale	text
73	594	23/10/2001 17:12:35	earl	text

In the preceding example, I have clicked on revision 75 of the file RELNOTES and am about to drop it on to revision 77.

If I complete the drop, then P4V runs the diff tool **P4Merge** on those 2 revisions.

The P4V diff tool – P4Merge

This is the built-in diff tool which is installed with P4V. Following on from the previous section we can see an example diff:



7 diffs (Ignore line ending differences) | Tab spacing: 4 | Encoding: System

//depot/Jam/MAIN/src/RELNOTES#75

```
1.  Release info:

    Jam/MR 2.2
    October 22, 1997
    VERSION 2.2
    PATCHLEVEL 0

2. Compatibility

    Jam/MR 2.2 is a roll-up of 'Jam - make(1) redux' releas
    Most of the changes described below were available befo
    in the jam.2.1.plus.tar ball.

    The Jam/MR 2.2 language is a superset of the 2.1 langua
```

//depot/Jam/MAIN/src/RELNOTES#77

```
1.  Release info:

    Jam/MR 2.2
    October 22, 1997
    VERSION 2.2
    PATCHLEVEL 1

1.1 Bug fixes since 2.2 patchlevel 0

    The NT handle leak in execunix.c has been
    Gurusamy Sarathy.  (Patchlevel 1).

2. Compatibility

    Jam/MR 2.2 is a roll-up of 'Jam - make(1)
    Most of the changes described below were
```

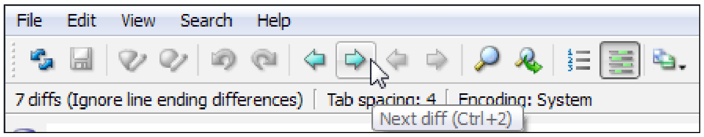
The title bar shows which revisions are being compared (#75 and #77 in this case).

In this example, we can see lines and blocks of text which have been changed, including particular parts of those lines which are highlighted in grey.

The summary is shown in the status bar at the top: **7 diffs**.

Navigating between diffs

You can scroll through the two files, or click on the arrow icons in the toolbar to go to the next or previous diff, as shown:



7 diffs (Ignore line ending differences) | Tab spacing: 4 | Encoding: System

You can also find the **Next Diff** and **Previous Diff** options in the **Search** menu

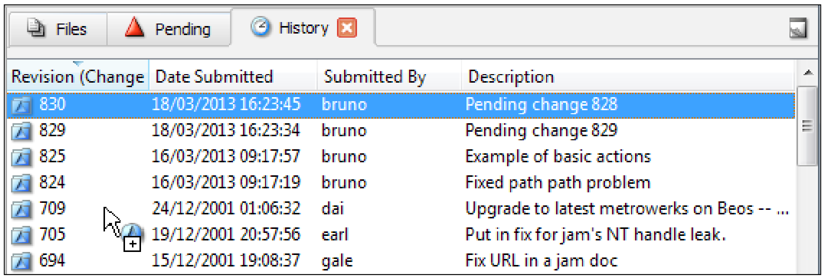
More P4Merge options

Navigate to the **Edit** | **Preferences** menu option and consult the help information for more details.

We recommend you explore the tool further on your own. In our experience it is powerful and capable. However, personal preference may lead you to want to use a different tool, this can be easily configured by going to P4V's **Edit** | **Preferences** | **Diff** menu option.

Showing folder/directory differences

In a similar way as for files, drag-and-drop of one changelist on top of another in the **History** tab will show us a folder diff, as shown:



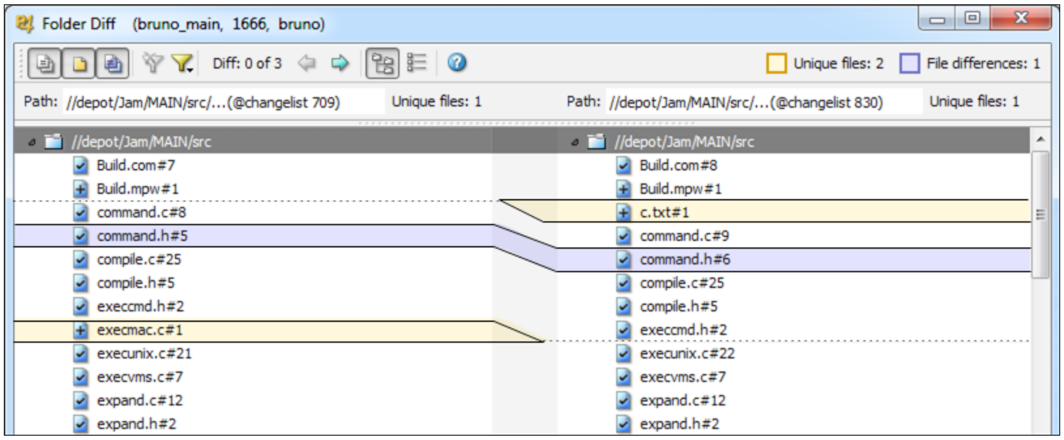
Revision (Change	Date Submitted	Submitted By	Description
830	18/03/2013 16:23:45	bruno	Pending change 828
829	18/03/2013 16:23:34	bruno	Pending change 829
825	16/03/2013 09:17:57	bruno	Example of basic actions
824	16/03/2013 09:17:19	bruno	Fixed path path problem
709	24/12/2001 01:06:32	dai	Upgrade to latest metrowerks on Beos -- ...
705	19/12/2001 20:57:56	earl	Put in fix for jam's NT handle leak.
694	15/12/2001 19:08:37	gale	Fix URL in a jam doc

In the preceding example, I clicked on changelist `830` and am about to drop it onto changelist `709`. Completing the drop action causes P4V to run the Folder Diff tool, which performs a recursive diff on all files in the folder and any sub-folders.

The folder diff tool – recursive folder differences

This is a very powerful tool which works well and performs quickly even for folders with tens of thousands of files in them.

Following on from the previous section we can see an example diff:



Path: //depot/Jam/MAIN/src/...(@changelist 709) Unique files: 1

Path: //depot/Jam/MAIN/src/...(@changelist 830) Unique files: 1

//depot/Jam/MAIN/src	//depot/Jam/MAIN/src
Build.com#7	Build.com#8
Build.mpw#1	Build.mpw#1
	c.txt#1
command.c#8	command.c#9
command.h#5	command.h#6
compile.c#25	compile.c#25
compile.h#5	compile.h#5
execcmd.h#2	execcmd.h#2
execmac.c#1	
execunix.c#21	execunix.c#22
execvms.c#7	execvms.c#7
expand.c#12	expand.c#12
expand.h#2	expand.h#2

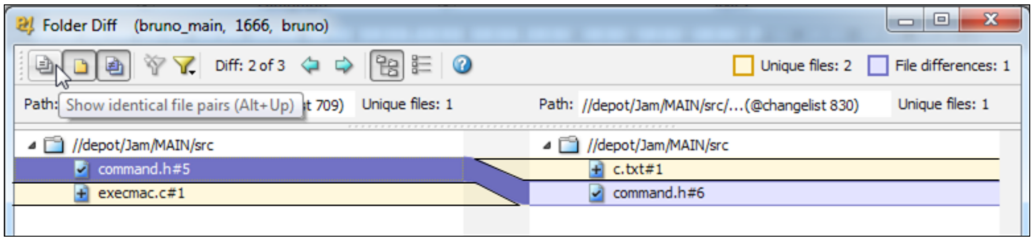
The two **Path:** options show which folders are being compared. In this case, the same folder is being compared against itself, but at two different points in time. As this statement implies, it is also possible to compare two different folders (we will cover this later in the chapter).

The summary information at the top-right gives us an overall idea of how many files are different. In this example, only one file has different contents.

However, there are two unique files which means they have either been added or deleted. On the right-hand side `c.txt#1` (highlighted) has been added between changelist `709` and `830`. The file `execmac.c#1` on the left-hand side has been deleted between the changelists.

Filtering the folder diff view

We can filter the view to only show the files that are actually different. This can be very useful when there are lots of files which are the same and we want to focus only on the differences.


Folder Diff (bruno_main, 1666, bruno)
Show identical file pairs (Alt+Up)
Diff: 2 of 3
Unique files: 2
File differences: 1

Path: ...t 709) Unique files: 1
Path: //depot/Jam/MAIN/src/...(@changelist 830) Unique files: 1

//depot/Jam/MAIN/src
command.h#5
execmac.c#1

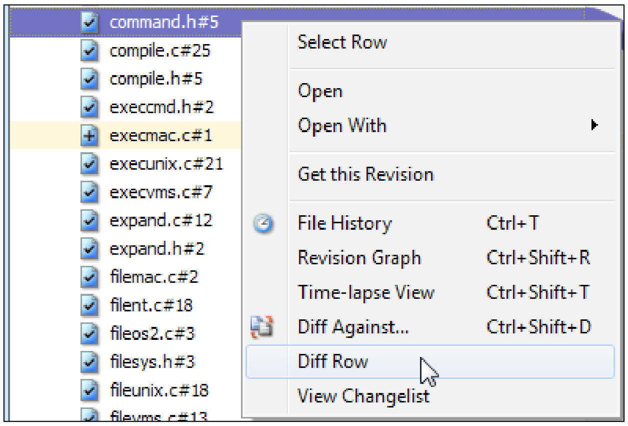
//depot/Jam/MAIN/src
c.txt#1
command.h#6

As shown in the screenshot, the toolbar icon in the top left condenses the view and removes identical file pairs.

Explore the other toolbar icons yourself or consult the help.

Showing individual file diffs

If a row is highlighted (in blue) then it means the file contents are different. In this case, there will be a **Diff Row** option on the right-click menu:


command.h#5
compile.c#25
compile.h#5
execcmd.h#2
execmac.c#1
execunix.c#21
execvms.c#7
expand.c#12
expand.h#2
filemac.c#2
filent.c#18
fileos2.c#3
filesys.h#3
fileunix.c#18
filevms.c#13

Select Row
Open
Open With
Get this Revision
File History Ctrl+T
Revision Graph Ctrl+Shift+R
Time-lapse View Ctrl+Shift+T
Diff Against... Ctrl+Shift+D
Diff Row
View Changelist

Clicking that option will run **P4Merge** on the two file versions.

Double-clicking the row is a shortcut action to perform the same diff.

Other options for comparing files or folders

We have seen the basic use of drag-and-drop for comparing files or folders, primarily from the **History** tab. It is worth noting that you can change drag-and-drop behavior by going to **Edit** | **Preferences** | **Behavior** | **Enable diff comparisons on file-to-file drag and drop**.

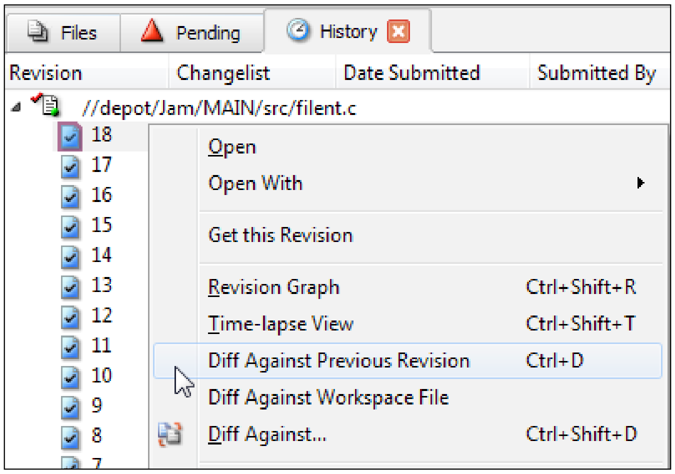
There are a number of other useful options for comparisons which we will discuss in this section.

Showing local changes for edited files

In *Chapter 3*, *Basic Functions*, we saw the **Diff Against Have Revision** option (with shortcut *Ctrl* + *D*). This option makes more sense now that we know that the **have** revisions (or `#have`) is the revision currently synced to our workspace. So we are comparing any local changes we have made with the revision we last synced to. Note that this is not necessarily the same as the latest (or `#head` revision), since someone might have checked in a new revision since we checked our revision out.

Ctrl + D as a useful shortcut for diffing

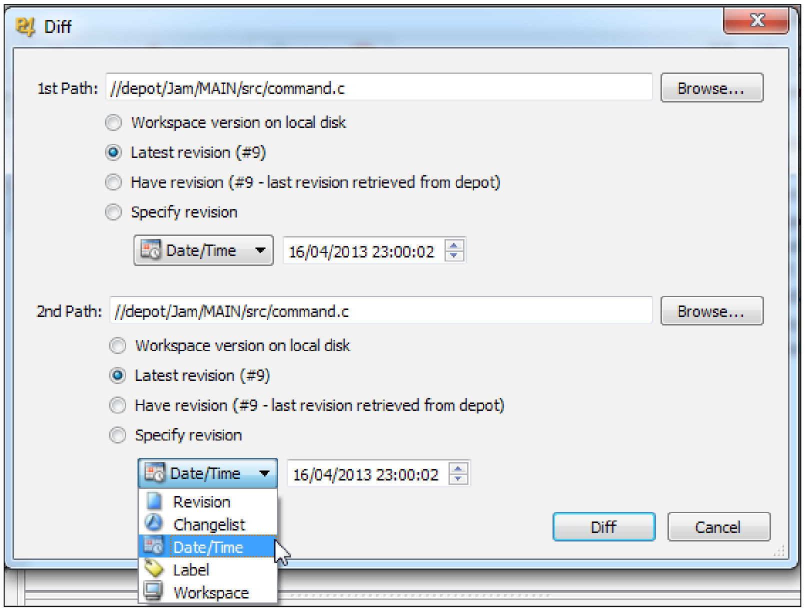
The *Ctrl* + *D* shortcut is useful in other situations than just for diffing edited files. As we can see in the following screenshot, on the right-click menu for the **History** tab, **Ctrl + D** is a shortcut for **Diff Against Previous Revision**:


Files
Pending
History
Revision
Changelist
Date Submitted
Submitted By
//depot/Jam/MAIN/src/filent.c
18
17
16
15
14
13
12
11
10
9
8
7
Open
Open With
Get this Revision
Revision Graph Ctrl+Shift+R
Time-lapse View Ctrl+Shift+T
Diff Against Previous Revision Ctrl+D
Diff Against Workspace File
Diff Against... Ctrl+Shift+D

The same shortcut key also works if you click on a revision in **Revision Graph**.

Diff against for files

On several right-click menus, including the screenshot in the previous section, you can see **Diff Against...** as an option. This will show you the following dialog:



Diff

1st Path: //depot/Jam/MAIN/src/command.c

○ Workspace version on local disk
● Latest revision (#9)
○ Have revision (#9 - last revision retrieved from depot)
○ Specify revision

Date/Time 16/04/2013 23:00:02

2nd Path: //depot/Jam/MAIN/src/command.c

○ Workspace version on local disk
● Latest revision (#9)
○ Have revision (#9 - last revision retrieved from depot)
○ Specify revision

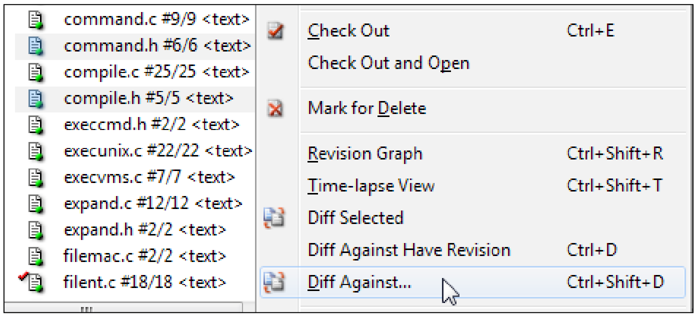
Date/Time 16/04/2013 23:00:02

As you can see in this screenshot, there are many powerful options for comparing the same file against itself. All of the various ways of designating a revision can be compared against other revisions. For example, you can compare the #head (latest) revision of a file against the revision valid at a particular date and time. You could also compare the file at one date against the version valid at another date, for example, the beginning and end of a month.

Behind the scenes, P4V translates each revision specifier such as date/time or label or workspace, into a valid revision and compares one revision against another. Obviously, if you end up comparing the same revision against itself, there will be no differences to see and you will see a warning message instead.

Using Diff against for different files

In the following example I used *Ctrl* + click to select two different files, and can then select **Diff Against…** on the right-click menu:



In this instance, the two files will be put in the **1st Path:** and **2nd Path:** fields for the dialog in the previous screenshot. All of the other options remain the same.

Diff against for folders

The same **Diff Against…** action is on the right-click menu for folders. In a similar way as for files, this allows you to compare the same folder against itself at a different point in time. Thus you could see what happened in the month of March.

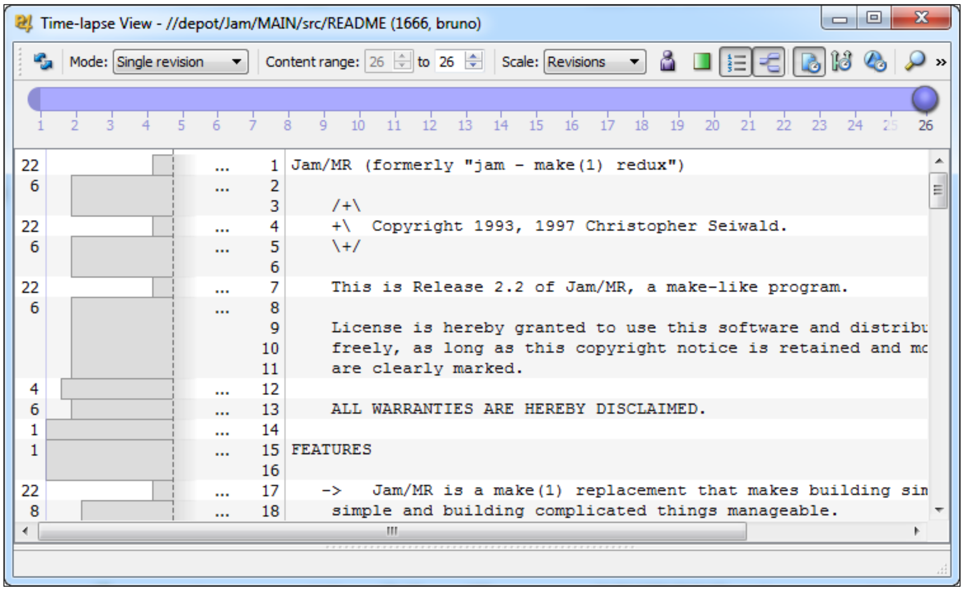
Equally, we can *Ctrl* + click on two folders and compare them from the right-click menu (or the standard shortcut key).

P4V time-lapse view

The **Time-lapse View** tool is available on the right-click menu for files.

It is a bit like a diff on steroids! It allows you to see how a file has evolved since it was first added to the repository.

We encourage you to play with this tool and work out how it will best help you to understand how a file has changed over time.



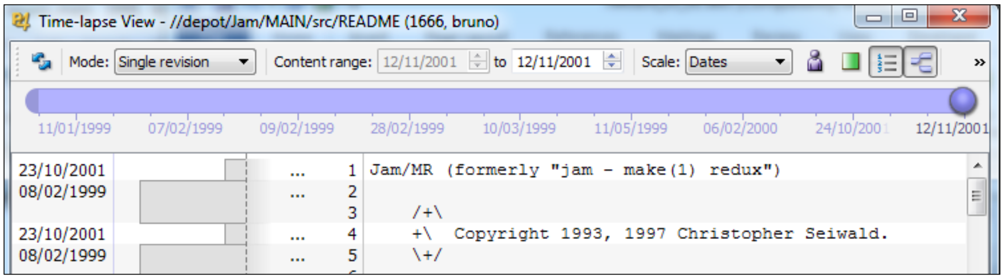
Time-lapse View - //depot/Jam/MAIN/src/README (1666, bruno)

Mode: Single revision   Content range: 26 to 26   Scale: Revisions

```
22   ...   1  Jam/MR (formerly "jam - make(1) redux")
 6   ...   2
           3      /+\
22   ...   4      +\   Copyright 1993, 1997 Christopher Seiwald.
 6   ...   5      \+/
           6
22   ...   7      This is Release 2.2 of Jam/MR, a make-like program.
 6   ...   8
           9      License is hereby granted to use this software and distribu
          10      freely, as long as this copyright notice is retained and mc
          11      are clearly marked.
 4   ...  12
 6   ...  13      ALL WARRANTIES ARE HEREBY DISCLAIMED.
 1   ...  14
 1   ...  15  FEATURES
          16
22   ...  17     ->   Jam/MR is a make(1) replacement that makes building sin
 8   ...  18      simple and building complicated things manageable.
```

Every line in the file is shown (line numbers are displayed here, they can be toggled off) and the boxes on the left-hand side show which revision was the last one to affect that line or block of lines. In this example, there are 26 revisions, but the first line was last modified as part of revision 22 (the figure on the left-hand side of the line). The block consisting of lines 2 and 3 was last changed in revision 6.

In the preceding screenshot, **Mode: Single revision** and **Scale: Revisions** show us everything in terms of revisions. We can change these options, for example:

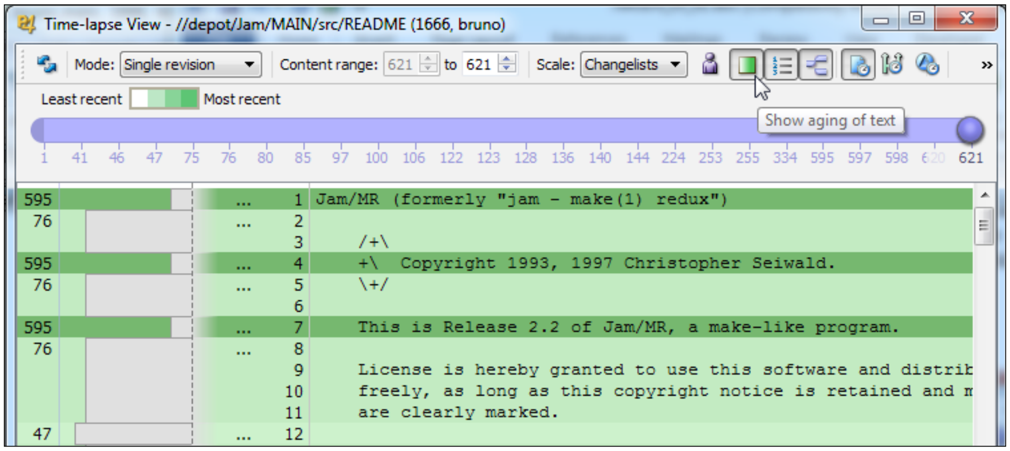


Time-lapse View - //depot/Jam/MAIN/src/README (1666, bruno)

Mode: Single revision   Content range: 12/11/2001 to 12/11/2001   Scale: Dates

```
23/10/2001   ...   1  Jam/MR (formerly "jam - make(1) redux")
08/02/1999   ...   2
                   3      /+\
23/10/2001   ...   4      +\   Copyright 1993, 1997 Christopher Seiwald.
08/02/1999   ...   5      \+/
```

We have now changed the scale to **Dates**, and can see in the toolbar how the file has evolved over time (rather than over revisions as per the previous screenshot). Alternatively, we can set the scale to **Changelists** and colour the text (in green):



Time-lapse View - //depot/Jam/MAIN/src/README (1666, bruno)

Mode: Single revision

Content range: 621 to 621

Scale: Changelists

```
595   ...  1 Jam/MR (formerly "jam - make(1) redux")
 76   ...  2
           3     /+\
595   ...  4     +\  Copyright 1993, 1997 Christopher Seiwald.
 76   ...  5     \+/
           6
595   ...  7     This is Release 2.2 of Jam/MR, a make-like program.
 76   ...  8
           9     License is hereby granted to use this software and distrib
          10     freely, as long as this copyright notice is retained and m
          11     are clearly marked.
 47   ... 12
```

The tool now has the toolbar icon toggled to **Show aging of text**. The more recent blocks of text are colored in a darker shade of green than the older ones, it can almost start to look psychedelic! How much you like this view depends on your own personal tastes.

Play around with some of the other modes and toolbar icons and find out which options you prefer.

> Some people swear by the **Time-lapse View**, and others seldom go near it, see how well it works for you. In our experience it can be incredibly useful at times, even if we tend to use ordinary diffs between two revisions most of the time.

Summary

This chapter provided in-depth coverage of some key concepts. Once you grasp these concepts, you can really start to appreciate the power of the reporting options available to you. The ability of Perforce to show comparisons between a folder tree that contains a thousand or more files can be hugely valuable. Precision file diffs and the time-lapse view make the analysis of changes a straightforward task that would be hard to accomplish using other methods.

In the next chapter, we'll look at modifying, creating, or otherwise managing your workspaces.

6
Managing Workspaces

In the Perforce model, you don't directly access or modify files in the server repository. Instead, client programs work with copies of repository files in local storage areas called workspaces. This chapter is about establishing and maintaining the relationship between server copies of files and workspace copies of those files.

Used properly, workspaces can save you hours of time and eliminate tedious and error prone workspace population tasks. Used improperly, workspaces will add to your workload and potentially impact overall server performance. Not to worry, workspace best practices are easy to apply once you know them.

In this chapter we will cover:

- The functions that workspaces enable
- Managing workspaces
- Working with workspace specifications
- Exchanging file content between clients and the server
- Workspace features for reducing your workload
- Workspace best practices

What does workspace mean?

The word **workspace** has several meanings in Perforce. The appropriate meaning is almost always implied by the usage context. Even then, most people are not very specific. They just say workspace to cover a range of meanings.

That won't do for this chapter, we need to be explicit about context and usage. There are some important points that are easy to miss if you aren't explicit. But don't worry, once you've seen the details, they'll quickly become second nature.

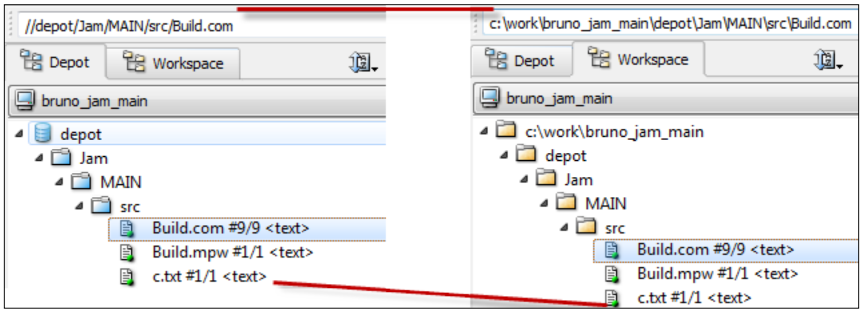
> The concepts implied by the term workspace are fairly consistent within the SCM communities of today. However, this has not always been the case. Perforce originally used the word **client** to differentiate its explicit workspace definitions from the implicit definitions of other tools. Because of legacy compatibility, you'll often see the word client instead of workspace in error messages and the command output in the log pane. We often use the term client workspace in our training sessions to reinforce the linkage between the terms.

Actions within a workspace context

Some actions are independent of any current workspace context, such as some of the reporting commands. However, various actions only have meaning within the context of a specific workspace: the current workspace. These actions include get latest, edit, mark for add or delete, and submit, indeed any action which potentially modifies a file.

When modifying files, we accessed and modified local copies within a workspace. When we were done with our activities and submitted our changes, the server updated its repository files with the contents of the files from our workspace.

Among other things, a workspace includes a mapping between files in the repository to files in our local file system as shown in the following screenshot:



```
//depot/Jam/MAIN/src/Build.com
bruno_jam_main
depot
  Jam
    MAIN
      src
        Build.com #9/9 <text>
        Build.mpw #1/1 <text>
        c.txt #1/1 <text>
```

```
c:\work\bruno_jam_main\depot\Jam\MAIN\src\Build.com
bruno_jam_main
c:\work\bruno_jam_main
  depot
    Jam
      MAIN
        src
          Build.com #9/9 <text>
          Build.mpw #1/1 <text>
          c.txt #1/1 <text>
```

The preceding screenshot shows the **Depot** and **Workspace** tabs in the tree panel for the same workspace view. We can see that files under `//depot/Jam/MAIN/` have been mapped to local directory `c:\work\bruno_jam_main\`. In this case, the folder and file structure is similar in both views.

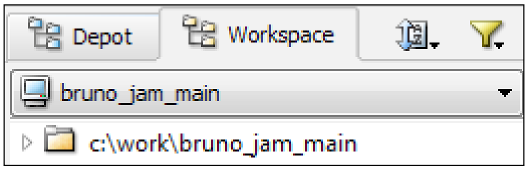
As we will see later in this chapter, it is very common to map only a small subset of the repository into our workspace. This is not too surprising, as a repository may contain hundreds or thousands of projects, and developers tend to work on only one project at a time.

A workspace – the specification

What we haven't talked about before this point are the specifics of how the client and server know what to do when transferring files between the repository and a workspace. We have implied that these specifications exist. And if you're doing follow-along, you selected a workspace based on available local storage. But beyond a relationship with the local file system (the mapping discussed in the previous section), we haven't provided any details about what a workspace name and its associated specification impacts.

The details of file transfer between server and workspace are defined in a workspace specification. Current versions of P4V have dropped the word specification. P4V simply refers to workspace names. You are expected to know from the context whether that name refers to the files in a workspace, a workspace specification, or both.

You've already seen examples of workspace names under the tabs in the tree pane. For example, `bruno_jam_main`, as we see here:



We will explore other ways of accessing and working with workspace specifications throughout this chapter.

Classic workspaces versus stream workspaces

In this chapter we deal with classic workspaces. When we cover the streams interface in *Chapter 9*, *Perforce Streams*, we'll see that stream workspace specifications look, on the surface, very different from classic workspace specifications. However, under the covers, most of the stream specification features are classic workspace specification features. Thus, most of the information in this chapter will apply when we get to streams.

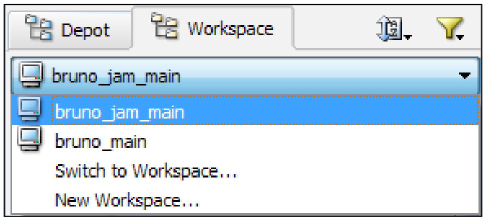
Managing workspaces

Every user has at least one workspace. Often users have more than one. Thus, a repository may have many hundreds, or even thousands, or tens of thousands of workspaces. While the various filters available within P4V can reduce the visual clutter, there are still the server and client resources dedicated to managing workspaces that we should consider. From the client perspective, workspaces occupy local storage (on the local file system). The server, on the other hand, uses resources to track workspace content even if there is no local storage involved. Thus, proper management of workspaces is a user activity that can impact both server and client resources.

> Because of the potential for a workspace to use server resources, and thus impact performance, some organizations have quite restrictive policies regarding creating and editing workspaces.

In this section we're going to talk about selecting, creating, deleting, and editing workspaces. In the next section we'll cover the details of the workspace specification itself. Since workspaces have a broad set of implied functionalities, the various workspace management interfaces focus on providing the most likely features for their context.

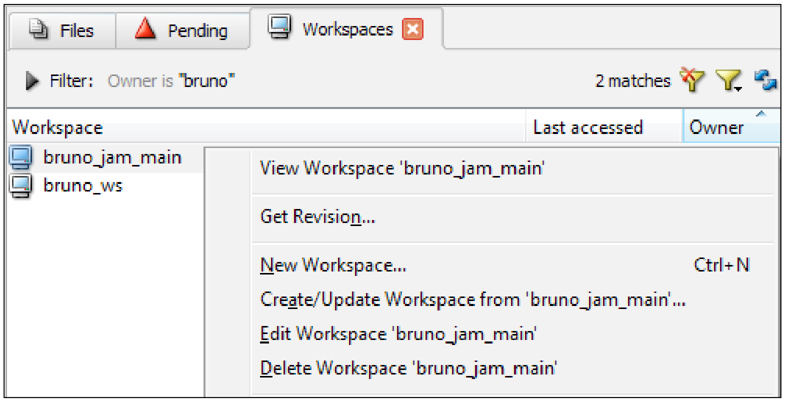
So far, you've been exposed to the workspace selection dropdown at the top of the tree pane as we see in the following screenshot:



The currently active workspace is shown in the dropdown, in the preceding case it is **bruno_jam_main**.

From this dropdown you can select previously active workspaces, switch to other workspaces, and even create a new workspace. These are all features consistent with presenting the relationship between repository storage and local storage.

The **Workspaces** view panel tab has a workspace related context menu:



Filter: Owner is "bruno"

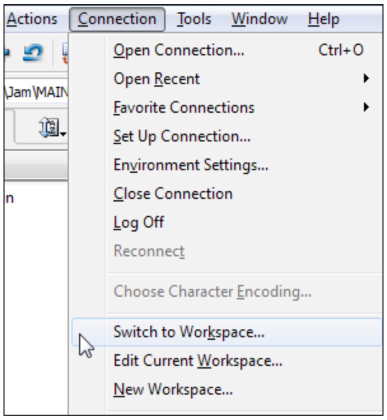
Files
Pending
Workspaces
2 matches
Workspace
Last accessed
Owner
bruno_jam_main
bruno_ws
View Workspace 'bruno_jam_main'
Get Revision...
New Workspace... Ctrl+N
Create/Update Workspace from 'bruno_jam_main'...
Edit Workspace 'bruno_jam_main'
Delete Workspace 'bruno_jam_main'

From this right-click menu you can select activities that apply to a specific workspace.

Note that there is a filter. As previously mentioned, without a filter, you must be prepared to sort through the hundreds or even thousands of workspace specifications that are present in typical Perforce server installations.

> If you're doing follow-along, now is a good time to experiment with the various workspace filter options. Consider why you would want to use the variations you come up with.

Finally, you will find workspace management related features under the **Connection** menu, such as **Switch to Workspace...**, as we see in the following screenshot:



These features apply to establishing a workspace context for the connection between P4V and the server.

We are often asked why workspace creation isn't found under the **File** | **New** menu. It's under the **Connection** menu to emphasize that workspace context is part of what defines a connection between P4V and the server.

Switching workspaces

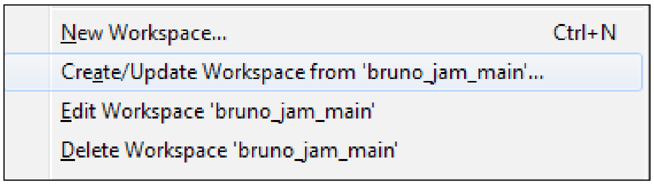
Switching to a particular workspace simply establishes a context for the relationship between P4V and the server. It does not change or update the locally stored files associated with that workspace, for example, by automatically doing a **Get Latest…**. You must explicitly specify an appropriate get action to update local files within the workspace.

Some people find it inconvenient that P4V doesn't automatically update local storage when you select a workspace. We, on the other hand, find that this behavior provides a valuable buffer in time. First, it avoids the transfer of data and the resulting need for local clean up if you switch to the wrong workspace. Second, it allows you to use the **Preview** feature of the **Get Revision...** dialog to assess what is going to happen.

Creating workspaces by copying

When a user creates a workspace, they will typically create a new workspace. A new workspace does not use local resources until you select and populate it. However, a new workspace always uses server resources. The server tracks what's in a workspace from the moment it is created.

Creating a new workspace can be a source of confusion for new users. When the relationship between repository and workspace files is complex, workspace creation can be a source of errors for even advanced users. For these reasons, Perforce provides the ability to use one workspace as a template, as we see in the context menu of the **Workspaces** tab in the view panel:


New Workspace... Ctrl+N
Create/Update Workspace from 'bruno_jam_main'...
Edit Workspace 'bruno_jam_main'
Delete Workspace 'bruno_jam_main'

You can use this template to create new workspaces: this copies the workspace definition and views as we cover in the following sections. You can also use it to verify that a workspace is consistent with a master template.

Editing workspaces

Exactly what editing involves is covered in the next section.

The important point here is to note that like selection, changes made by editing a workspace are not automatically applied. You need to do something, such as populate the workspace or submit a file, that references the modified workspace context, before anything will change.

Don't overlook the value of the **Preview** feature of the **Get Revision...** dialog if your edits have modified the relationship between server and local files. In addition to the normal value, errors reported by the **Preview** process will expose typos and other problems with your specification.

Deleting workspaces

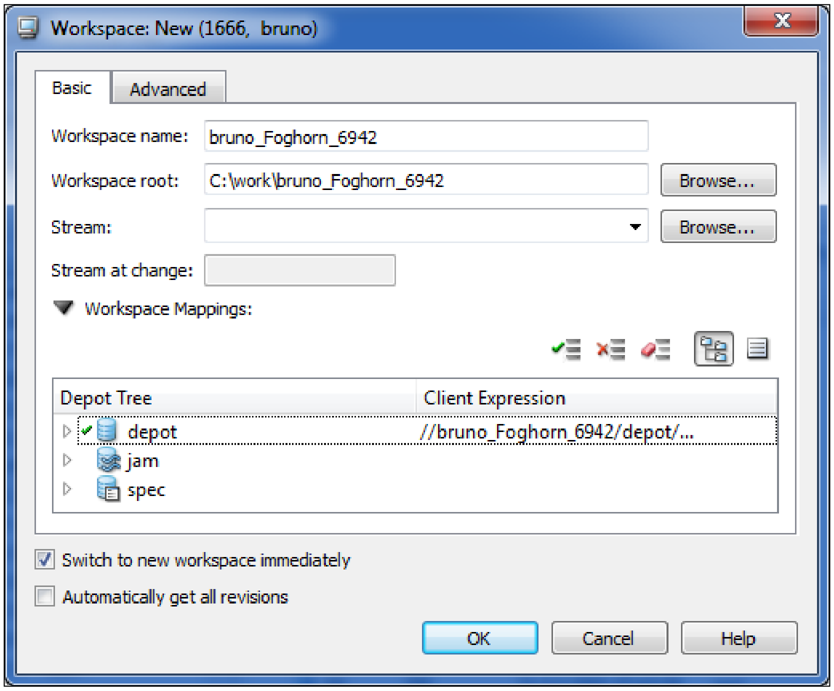
When a workspace is no longer needed, it should be deleted. This frees both server and client resources. Simply removing the files from local storage is not enough. As long as a workspace exists, the server is using resources to track the contents of that workspace.

Although you can remove locally stored files using standard file system features, such as deleting a directory, that isn't the best practice. Best practice is to use **Remove from Workspace** from the tree panel context menu to remove files from local storage. The benefit of using Perforce to remove the files is that anything left after the removal was not under source control. Stories of users deleting workspaces using operating system features only to discover that a critical file or test harness component had been forgotten and never added to source control are legend. Learn from the mistakes of others!

Specifying a workspace

Creation and modification of workspace specifications are coordinated through a two-tab dialog. The features accessed through the **Basic** tab are concerned with the relationship between server repository files and workspace files. The **Advanced** tab features provide control over the characteristics of population and the relationship between workspaces and the submit process.

Here, we see the **Basic** tab panel for a new workspace:



Workspace name: bruno_Foghorn_6942

Workspace root: C:\work\bruno_Foghorn_6942

Stream:

Stream at change:

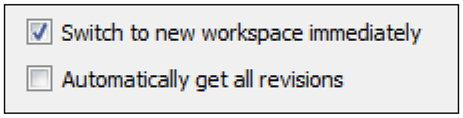
Depot Tree	Client Expression
✓ depot	//bruno_Foghorn_6942/depot/...
jam	
spec	

☑ Switch to new workspace immediately

□ Automatically get all revisions

The **Workspace name:**, **Workspace root:**, and **Workspace Mappings:** are the most common values that users need to specify when creating or editing workspaces.

Note the options at the bottom of the panel. These concern selection and population as we discussed in the workspace management section.

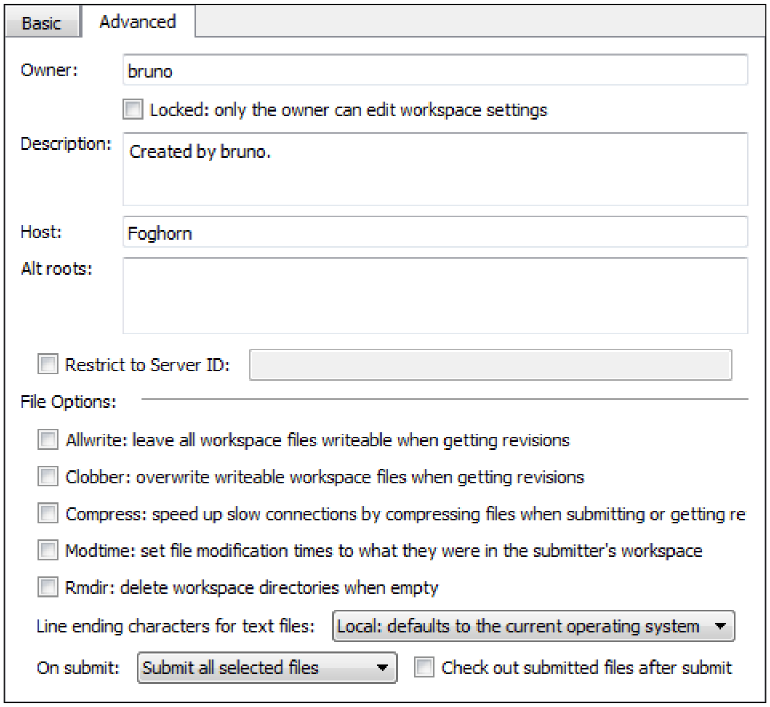


☑ Switch to new workspace immediately

□ Automatically get all revisions

By default, you must explicitly request the population of your workspace. This behavior reflects best practice workspace management techniques.

The **Advanced** tab mixes standard Perforce object features, such as **Owner:** and **Description:**, with workspace population and submit characteristics:



Owner: bruno

☐ Locked: only the owner can edit workspace settings

Description: Created by bruno.

Host: Foghorn

Alt roots:

☐ Restrict to Server ID:

File Options:

☐ Allwrite: leave all workspace files writeable when getting revisions

☐ Clobber: overwrite writeable workspace files when getting revisions

☐ Compress: speed up slow connections by compressing files when submitting or getting re

☐ Modtime: set file modification times to what they were in the submitter's workspace

☐ Rmdir: delete workspace directories when empty

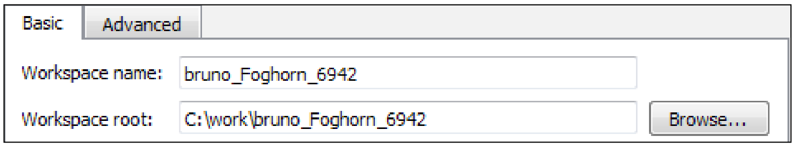
Line ending characters for text files: Local: defaults to the current operating system

On submit: Submit all selected files ☐ Check out submitted files after submit

Be careful when modifying these values. As we will see, there are consequences to these values that may not be immediately obvious.

Workspace names

When you create a new workspace, Perforce will assign a default name:



Workspace name: bruno_Foghorn_6942

Workspace root: C:\work\bruno_Foghorn_6942

This default is intended to avoid workspace name conflicts on larger systems. While the default name is almost certainly unique, most people don't find it particularly meaningful. In this case, `bruno` is our username and `Foghorn` is the name of the PC on which we are running P4V. We suggest using workspace names that mean something to you. These typically start with your Perforce username, for uniqueness, followed by project and branch names. If more uniqueness is required, append that after a common root name.

Most organizations have a workspace naming convention. If your organization is small and you don't have a workspace naming convention, establish one now while there are fewer non-conforming names. A commonly seen example is <username>-<project name>-<branch or task name>, so for example, bruno-projectX-main.

Workspace location

The top level directory of the local storage structure that contains workspace files is specified as the **Workspace root**.

In the preceding example, this was `c:\work\bruno_Foghorn_6942`.

The default workspace root is rarely what you want. You can use the **Browse...** button to find an existing structure or you can enter a path by hand. Don't worry if some or all of the directories in the path don't exist. P4V will create these path elements when it populates the workspace.

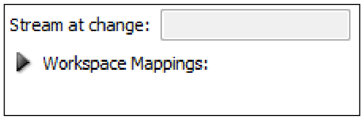
You may have noticed that you can't expand or contract the path at the top of the workspace tab in the tree panel. That's because it's the workspace root. A workspace is not concerned with files that aren't under the root.

If you are operating on a Windows platform be careful with the length of the root path. The root is pre-pended to every workspace file path. If you are using an application such as SQL that tends to create verbose path and file names, or a language such as Java that tends to create deep directory structures, the combination of working path, filename, and root may exceed the path length limit imposed by Windows (which is usually 260 characters in total). Likewise, be aware that if you are using such a system, the need for extremely short workspace roots may make it difficult for others to create workspaces that can access your work.

Many people use a common top-level directory for workspaces. Be sure that your workspace root path is unique enough so that you can add additional workspaces later without running into naming conflicts.

Relating repository files to workspace files

Specifying the relationship between repository files and workspace files is known as **workspace mapping**. You will find the mapping specification at the bottom of the **Basic** tab panel. If, as shown in the following screenshot, you see **Workspace Mappings:** but no mapping interface, simply click on the arrow head before **Workspace Mappings:** to display the mapping interface:

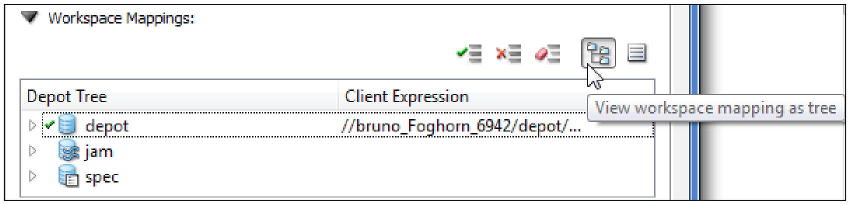


Stream at change:

Most of the time you don't need to map every repository file to your workspace. However, the default is mapping everything you currently have read access to, because the server doesn't know what you want. Even if your current repository is small, don't accept the default of mapping everything. Get in the habit of only mapping what you want to use now and it won't be hard to adjust in the future.

> If your administrator is on the ball, he or she will have created a trigger to set the default mapping to something more sensible than everything in the repository!

P4V provides two modes for specifying mappings: tree mode and text mode. The tree mode interface is appropriate for users who need to create basic repository to workspace mappings. It presents depot structures in a graphical format similar to the presentation used in the depot tree pane. Text mode is for advanced users. Select the mode using the buttons at the top-right of the **Workspace Mapping** panel:



Depot Tree	Client Expression
✓ depot	//bruno_Foghorn_6942/depot/...
jam	
spec	

Like other parts of the P4V interface, hover tips are available to remind you of available functionality.

> Before you create a workspace you should plan the mapping. This plan needs to identify the root directory and the repository files that you want in your workspace. The interface is deceptively easy to use. Having a plan avoids problems.

A workspace specification example

At this point we could present you with a long list of the features available through the mapping interface. However, without context, this would mostly be just words on a page. Instead, let's create a workspace for the user bruno, which will align with the follow-along workspace.

The first thing we need to do is create a plan. The plan identifies:

- The repository files we want in the workspace
- The workspace name
- The root directory for local workspace storage

In this case, we identify that we wanted to work with files in the `MAIN` branch of the `Jam` project. And we identify that the `Jam` project is found in the depot called `depot`. We'll cover depot structure and naming in *Chapter 8*, *Classic Branching and Merging*. For now, you only need to know that depot folder structures are determined by the user (and depots are created by administrators). In this case, users followed the common convention of using the project name (`Jam`) as the top level depot folder under which folders for each branch are created (in this case `MAIN`).

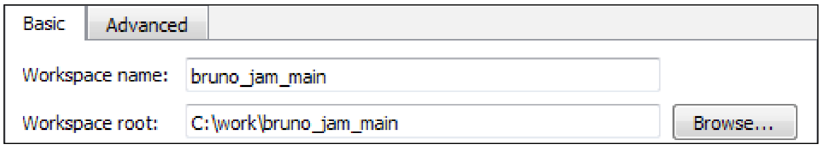
> Determine the repository files first. They tend to influence the workspace name and root directory.

The workspace name `bruno_jam_main` follows the common convention of starting with the Perforce user name (`bruno`) followed by a project name (`jam`) and a branch or task name (in this case a branch name `main`).

Likewise, the Workspace root (`c:\work\bruno_jam_main`) was determined by following the common convention of creating a project subdirectory (`jam`) under a local top level directory (`c:\work`), then creating a directory under the project directory based on a branch or task name (in this case the branch `main`). This path is likely to be unique relative to other root directories that we might specify at a later time.

> You can move workspace roots. However, it's messy and error prone. Even if it's a little extra work, start with a structure that ensures workspace root directories never overlap with other workspaces. For example, don't have one workspace root directory be a sub-directory of another workspace. The convention of using the local top-level directory mentioned in the previous paragraph ensures this.

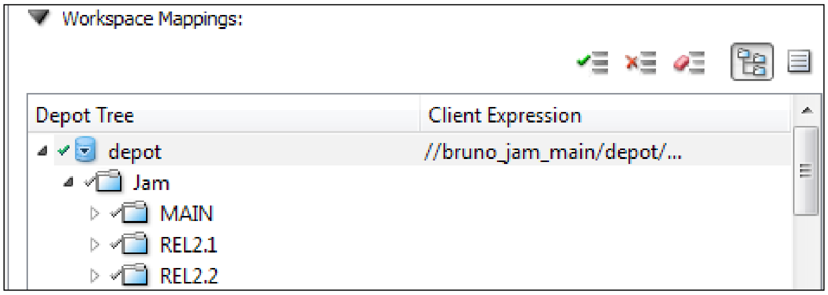
Now that we have a plan, we request a new workspace. When presented with the workspace specification dialog we selected the **Basic** tab and entered values for **Workspace name:** and **Workspace root:** as shown in the following screenshot:



Workspace name: bruno_jam_main

Workspace root: C:\work\bruno_jam_main

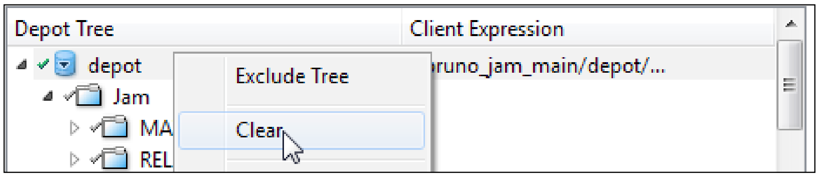
Now it's time to specify the mappings of repository files to workspace files. When we look at the **Workspace Mappings:** panel we see that the default selection of all depot files had been made for us:



Depot Tree	Client Expression
depot	//bruno_jam_main/depot/...
Jam	
MAIN	
REL2.1	
REL2.2	

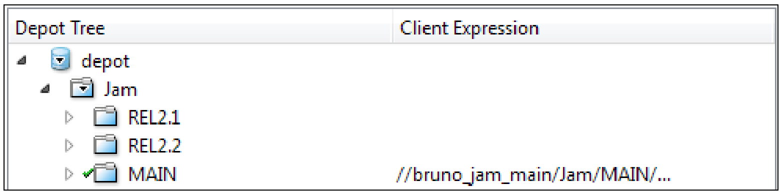
We know that everything has been included because there is a blue check mark next to each name. Further, we know that the mapping specification only explicitly references the depot because only the line for the **depot** node has a **Client Expression**. The other folders are included by implication because they are children of the depot node, this is why they have a gray check mark.

But we only want files from the main branch of the `Jam` project. When we examine the available action buttons, it's obvious that we want the include feature. So what is the best way to eliminate the other nodes? We find that there are two possible ways that might achieve this: exclude and clear. While exclude would work, it's not the best way. The best way is to clear the depot mapping and start fresh as we see in the following screenshot:



Depot Tree	Client Expression
depot	runo_jam_main/depot/...
Jam	
MA	
REL	

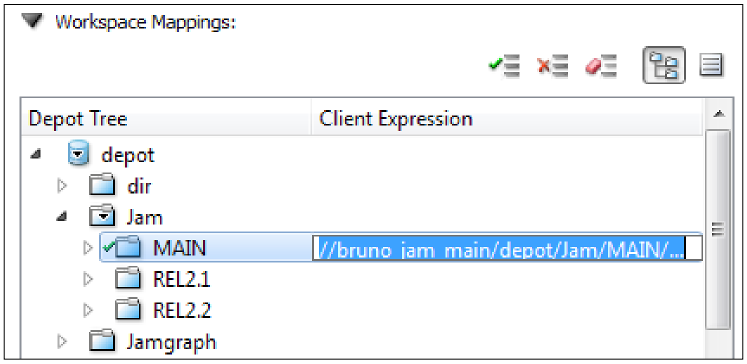
Now we simply select the `MAIN` node under the `Jam` project node and right-click and select **Include Tree**. This includes that folder and any files and sub-folders in our workspace:



Depot Tree	Client Expression
depot	
Jam	
REL2.1	
REL2.2	
MAIN	//bruno_jam_main/Jam/MAIN/...

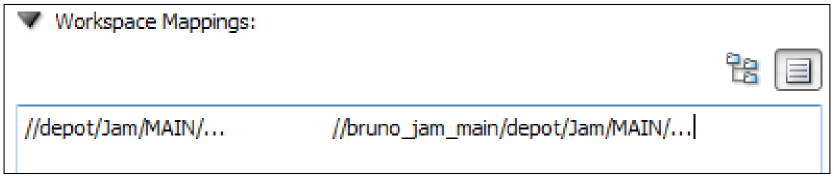
However, we're not quite done. Look at the **Client Expression** (`//bruno_jam_main/Jam/MAIN/...`) for the `MAIN` node. This is known as either **workspace syntax** or **client syntax**. We'll go with client syntax since client is the word most commonly used. When P4V interprets client syntax the workspace name (`//bruno_jam_main`) is substituted by the root directory of the workspace. In this case, that means that files are going to the workspace directory: `c:\work\bruno_main_jam\Jam\MAIN`. In this instance, we might want to change this to remove one or more elements of the directory name.

There are three ways to deal with this. One way is to modify the workspace root directory, but this is rarely the best option. So let's find another way. For example, it would be convenient if we could modify the **Client Expression** in the panel, and this is what we will do. To modify the **Client Expression** you first click on the node name then move your cursor over the **Client Expression** and click again. This will put the panel into edit mode and allow us to change it as we see in the following screenshot:



Depot Tree	Client Expression
depot	
dir	
Jam	
MAIN	//bruno_jam_main/depot/Jam/MAIN/...
REL2.1	
REL2.2	
Jamgraph	

We delete the extraneous `/depot/Jam/MAIN` and we're done. Of course we could have also changed to text mode by clicking the button on the right-hand side above the tree. This shows a panel that allows us to directly edit the mapping values in a text format:



//depot/Jam/MAIN/... //bruno_jam_main/depot/Jam/MAIN/...

However, values are presented in two columns. Which do we modify? Well, let's look at the columns. The left-hand side column has what is known as depot syntax path (`//depot/Jam/MAIN/...`) which specifies what we want to get from the repository. The right-hand side column has the client syntax path we saw in tree mode. This is where we specify where the files should be put locally. The workspace name component (`//bruno_jam_main`) is mandatory, as it represents the workspace root directory. So let's modify the client syntax path and eliminate the `/depot/Jam/MAIN`. This leaves us with `//bruno_jam_main/...`.

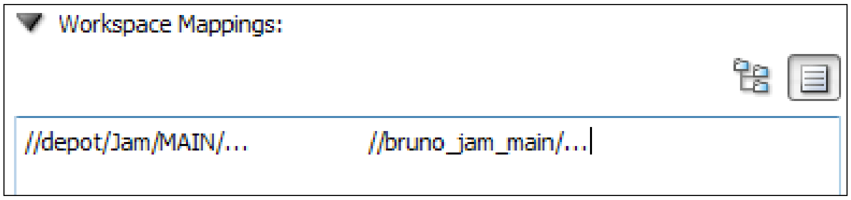
> Remember the tips about using preview to identify mapping specification issues? This is exactly why we identified those tips.

After you remove the `/depot/Jam/MAIN` and transition back to the tree mode you will see that the **Client Specification** has changed. This is because tree mode and text mode are simply different ways of looking at the mapping information.

What the ... (ellipsis) notation means

Remember from *Chapter 5*, *File Information*, that the ... (or ellipsis) notation is a Perforce wildcard that matches all characters including the directory separator character. It is the way to specify every file in this directory and all subdirectories.

Let's look at the resulting text mode for the example we just completed:



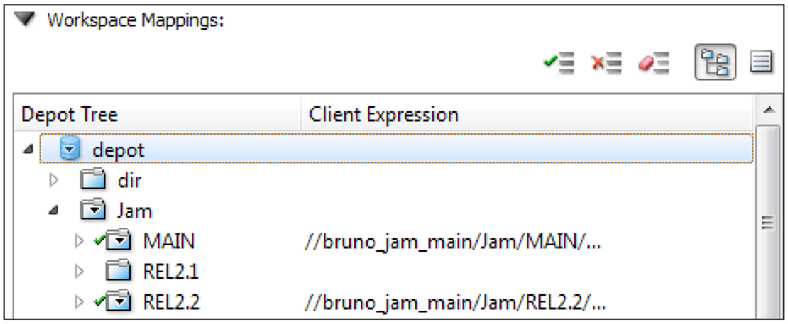
//depot/Jam/MAIN/... //bruno_jam_main/...

We can read this as: all files in the repository depot called depot (`//depot`) that exist in or under (`/...`) the directory `/Jam/MAIN` are mapped to the root directory for the workspace `bruno_jam_main`.

Note that both expressions end with `/...`. The rule of thumb for specifications is that wildcards used in the depot syntax (on the left-hand side) must match the wildcards in the client syntax (on the right-hand side).

How to exclude files

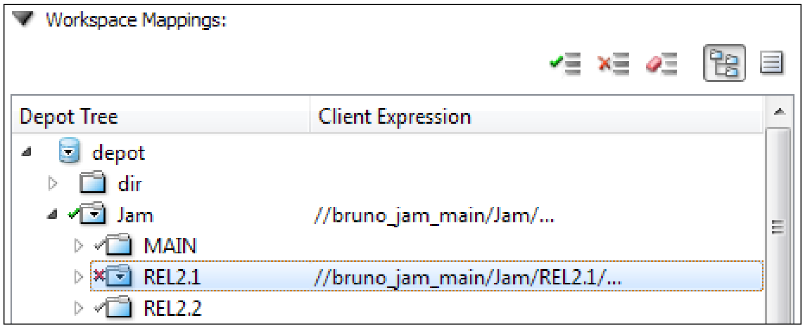
Using the current depot structure, consider how we would include both `Jam/MAIN` and `Jam/REL2.2` in the same workspace. One solution would be to create a workspace mapping that explicitly included both `Jam/MAIN` and `Jam/REL2.2`. You would wind up with a workspace mapping that looked like this:



Workspace Mappings:

Depot Tree	Client Expression
depot	
dir	
Jam	
✓ MAIN	//bruno_jam_main/Jam/MAIN/...
REL2.1	
✓ REL2.2	//bruno_jam_main/Jam/REL2.2/...

As an alternative, you could include `Jam` and explicitly exclude `Jam/REL2.1` which would result in a workspace mapping that looks like this:



Workspace Mappings:

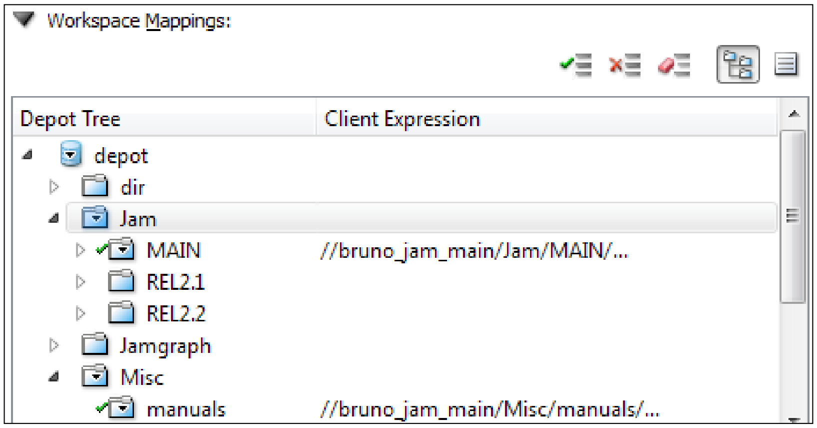
Depot Tree	Client Expression
depot	
dir	
✓ Jam	//bruno_jam_main/Jam/...
✓ MAIN	
✗ REL2.1	//bruno_jam_main/Jam/REL2.1/...
✓ REL2.2	

The different techniques yield workspaces with the same set of files. However, there is a significant difference between the workspaces that are created. In the first case your workspace is always limited to just the two branches `MAIN` and `REL2.2`. However, in the exclusion case new branches added under `Jam` would be implicitly added to your workspace because they would be children of `Jam` which is explicitly included together with its descendants. Keep the effects of future additions in mind when you're working with more complex workspace mappings.

> A particular folder might have ten subfolders, including one named `binaries` which you don't want to map to your workspace. It is possible to specify that the nine subfolders you want mapped one at a time. Alternatively, it is often easier to map the higher-level folder and then just exclude the `binaries` subfolder. The result in the workspace would be the same.

Can I reference more than one depot in a workspace?

Say you want to create a workspace that has the sources from the `MAIN` branch as well as the available project `manuals`. Since the project manuals are under the `misc` directory you might think you're in for manual activities. Not to worry, you simply include both `MAIN` and `manuals` as we see in the following screenshot:

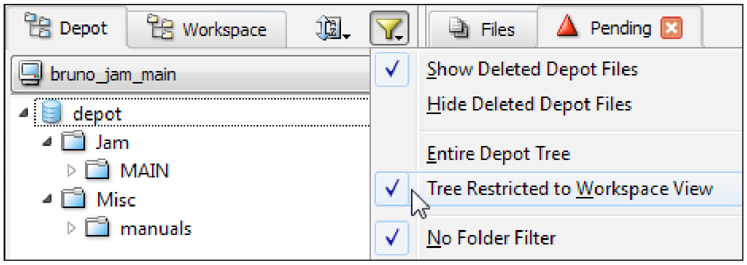


Depot Tree	Client Expression
depot	
dir	
Jam	
✓ MAIN	//bruno_jam_main/Jam/MAIN/...
REL2.1	
REL2.2	
Jamgraph	
Misc	
✓ manuals	//bruno_jam_main/Misc/manuals/...

You may want to modify the **Client Expressions** to create a more consistent workspace structure. Perhaps in your workspace you want the manuals as a directory under `MAIN`. No problem, just modify the **Client Expression** for the `manuals` node to be `//bruno_jam_main/MAIN/manuals/...`. P4V and the server use the mapping specification to relate files in the workspace to the appropriate files in the repository. You get the workspace structure you want and it's all under source control without an ongoing effort on your part.

Referencing directories or files from more than one depot is the same as referencing multiple directories. The only difference is likely to be including the name of the depot in the right-hand side mappings in order to distinguish between the multiple depots.

Isn't this going to make finding the depot version of a workspace file hard? It can if you look at the entire depot in the depot tab of the tree pane. However, the depot tab filter of the tree pane has a **Tree Restricted to Workspace View** option, as we see in the following screenshot:



This causes the depot display to reflect only those files that are found within the currently selected workspace.

The potential of workspace mappings

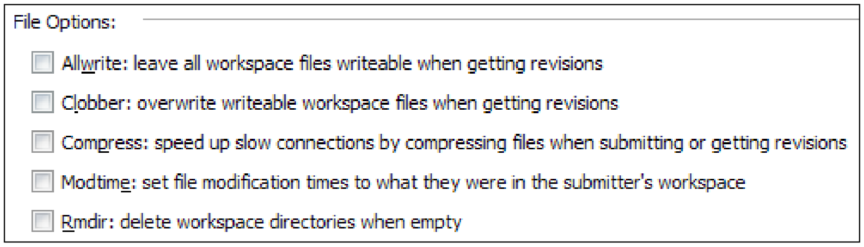
The preceding section covered what most users need to know. However, it has only scratched the surface of what is possible with workspace mappings. Among other things, advanced usage can restructure and rename down to the file-level.

Other wildcards can be used in mappings such as the asterisk (*). It is also possible to map individual files one at a time. These are more advanced usages and are discussed in the online Perforce Command Line Reference manual as mentioned in *Appendix B, Command Line*.

Now is a good time to try some examples. Create a workspace and work with include and exclude. If you're feeling adventuresome, you might try multiple structure references with edits of the **Client Expression**.

Population characteristics

Most of the time, the default **File Options:** are appropriate. However, there are times when you will need to change them. These options are displayed here:



File Options:

□ Allwrite: leave all workspace files writeable when getting revisions

□ Clobber: overwrite writeable workspace files when getting revisions

□ Compress: speed up slow connections by compressing files when submitting or getting revisions

□ Modtime: set file modification times to what they were in the submitter's workspace

□ Rmdir: delete workspace directories when empty

While the implications of selecting most of the **File Options:** are obvious from the descriptions, some have implications that most people overlook.

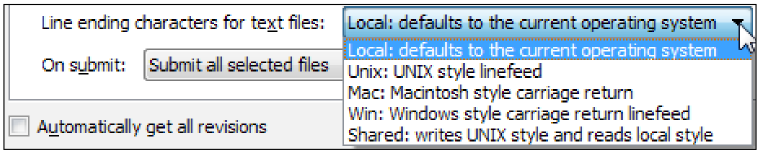
Normally when a workspace is populated with a file, that file is set to be read-only. For Perforce, read-only does not indicate under source control. A file is set to read-only to protect it against uncontrolled modification. However, some IDE tools assume that files that aren't read-only are either not under source control or are already checked out. If you select the **Allwrite** option and use such tools you might miss modifications.

Normally, directories are not removed once they are created even if they are empty. However, setting the **Rmdir** option can be useful. If you use P4V to **Remove from Workspace**, then the only files remaining in your workspace are files not under source control. If you select the **Rmdir** option it's easier to identify the files that are not under source control. It is also neater and tidier, and the one option that the authors invariably turn on!

Line endings

When a text file is synced down to a workspace on a Unix server or a Windows PC, the line endings will normally be different and appropriate to the operating system. Likewise, when a text file is submitted to the server, the line endings are normalized to the server format. This allows the server to use a common repository file to support multiple potential client environments. It's best to ignore server representation and concentrate on what is the appropriate text file line ending in your workspace.

This line ending transformation process is specified by **Line ending characters for text files:** within the **File Options:** section of the **Advanced** tab:



Line ending characters for text files: Local: defaults to the current operating system

On submit: Submit all selected files

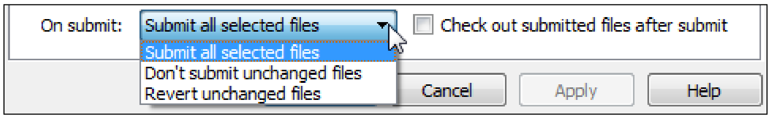
□ Automatically get all revisions

Local: defaults to the current operating system
Unix: UNIX style linefeed
Mac: Macintosh style carriage return
Win: Windows style carriage return linefeed
Shared: writes UNIX style and reads local style

Most of the time **Local: defaults to the current operating system** is appropriate. If you choose something else, be alert to editors and other tools that may create line endings that are inconsistent with the expected local text file line ending format. **Shared: writes UNIX style and reads local style** is a special choice for exactly this case. With shared, text files are synced into a workspace with Unix line endings. If there are any Windows style line endings that are created in a text file, they are converted to Unix style line endings before the file is passed to the server.

Submit options

The submit options set the default action for every submit within a workspace. Setting them sensibly can save you making a mistake each time you perform a submit. These values can always be overridden when submitting, so select values that are the best choice:



On submit: Submit all selected files

□ Check out submitted files after submit

The **On submit:** options are as follows:

- **Submit all selected files**: Submit checked out files even if they haven't changed. This is unlikely to be your best choice, in most cases there is little point in submitting a new version of a file if it hasn't changed.
- **Don't submit unchanged files**: Files that are checked out, but have not been changed, are left behind in the default changelist. The assumption is that you checked these files out for a reason so they should remain checked out until they are modified. This is probably our preferred option.
- **Revert unchanged files**: Files that were checked out, but have not been changed, are reverted and thus not one of the files submitted. This setting accounts for tools that are overly aggressive with auto-check out.
- **Check out submitted files after submit**: This is useful if you tend to always work on the same set of files and don't want to need to check them out individually after each submit. It can be useful when working with some projects or technologies.

> This is a good time to try some submit actions. Look for override options in the submit dialog and try the various options.

Perforce filetypes

There are some cases where individual files require special handling when you populate a workspace with them. Rather than adjusting the workspace characteristics to handle these special cases, consider using Perforce filetypes.

For example, certain tools require files to exist, but they always overwrite the file contents as part of their processing. It is hard to coordinate check out and revert for these files. The bad choice is to select the **Allwrite** option for the workspace. The better alternative is to use the Perforce filetype (see *Chapter 5*, *File Information*) to always populate the workspace with a writeable version of that file.

Common best practice questions

For the most part, workspace best practices boil down to variations on the following rules:

- Map only what you need
- Use straightforward mappings
- Avoid content changes you can't control
- Clean up when done

This is not to say that there aren't valid exceptions to these rules. However, such exceptions are rare in typical development scenarios.

Changelists and open files

Remember that a pending changelist belongs to only one workspace and that an open file belongs to only one changelist. If you find that you need the same file to be open in more than one changelist then consider multiple workspaces. Multiple workspaces provide a controlled environment for the management of potential conflicts. We'll talk more about conflicts in *Chapter 7*, *Dealing with Conflicts*.

More than one workspace

It is very common for users to have more than one workspace. So long as you follow the good workspace management practices outlined previously this shouldn't be a problem. However, one workspace per feature or bug fix is usually overkill. When you've completed a task, reuse the workspace for the next task.

[If you are in an environment where each feature or bug fix requires a unique workspace, then consider the streams feature.]

Sharing workspaces

Sharing a workspace is usually a recipe for mistakes. It is very likely that sharing a workspace will result in the loss of content modifications. There are two variations on this theme.

In the first variation, the same workspace is used on more than one client machine. The problem is that the server is tracking contents relative to a workspace, and it assumes that the same workspace always refers to precisely the same directory on disk, independently of which machine the workspace is being used on. Thus this option only makes sense if the workspace root is on a network file share, which is accessible from both machines! If that is not the case, then both you and the server are likely to get confused as to which files are synced on which machines. Even if you are referring to a network file share, it is not a recommended practice as there is a much greater risk of edits being lost and similar problems.

The one exception to the no workspace sharing rule is a build farm, where multiple hosts use a common workspace for read-only workspace population. This build scenario requires that you follow very specific use rules.

Summary

In this chapter we covered the basics of workspace management and its definition. We've also looked at the best practices associated with workspaces. In general, workspaces might look complex, but they are actually very straightforward. If you find significant complexity in your workspace mappings, or find that you constantly need to edit your workspaces, you are probably doing things the hard way. In such cases it is worth talking to your administrator and trying to work out how to permanently change your repository structure in order to keep the workspaces simpler.

In the next chapter we'll look at handling conflicts when working with files, including merging your changes with the work of others.

7
Dealing with Conflicts

To Perforce, conflict does not indicate a problem. Rather, it refers to the need for a human to resolve issues that may arise from independent modifications of the same repository file. If there is more than one user, branch, or workspace there are conflict scenarios.

As we will see, conflict scenarios are a natural consequence of the flow of development. In this chapter, we'll review the origins of Perforce conflicts, predicting future conflicts, identifying current conflicts, resolving conflicts, and ways to avoid them.

In this chapter we will cover:

- File conflicts
- Content conflicts
- Interactive resolution
- Automatic resolution
- Avoiding conflicts

The origin of a conflict

Conflicts arise from concurrent changes to the same file.

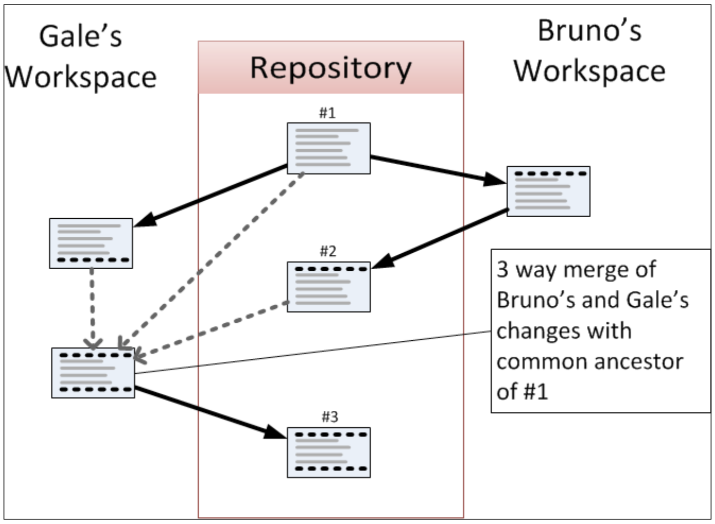
Let's consider what happens when two people concurrently modify the same file without tool support. The second person to save their changes risks overwriting the other person's changes. Most of us have experienced this at one time or another! This is the essence of a conflict.

P4V detects and manages conflicts so users can coordinate their changes effectively and avoid losing their work.

Other systems sometimes refer to conflicts and the actions to resolve them as merge. Since Perforce allows you to manage more than a simple merge of content they use the more accurate conflict and resolve terminology.

Working with conflicts

When using Perforce, users Bruno and Gale both make changes to the same repository file `//depot/dir/README.txt`, but independently, each using their own workspace copy as shown here:


Gale's Workspace
Repository
Bruno's Workspace
#1
#2
#3
3 way merge of Bruno's and Gale's changes with common ancestor of #1

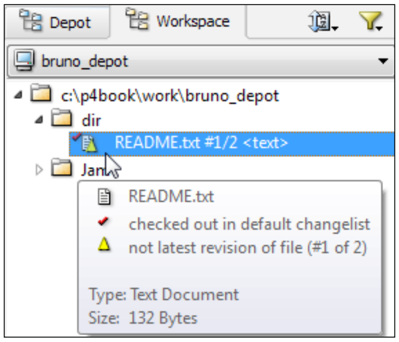
Both Bruno and Gale checked out and modified version `#1` of the file. When Gale submitted her changes she created revision `#2`, which in turn created a conflict for Bruno. Because of the conflict, any attempt by Bruno just to submit his changes will not be allowed by the server. Instead, Bruno needs to resolve the conflict by combining his changes with Gale's changes using a standard version control technique known as a 3-way merge. Only once the conflict is resolved does the server allow Bruno to submit his file which creates version `#3`.

How the server detects conflicts, and the details of a 3-way merge, are explained in detail in the following section.

Conflicts and submit

In the preceding example, both users started with version #1 of the file in their workspace which is the #head revision. Both users checked the file out and modified it. The checkout and modification did not create the conflict. The conflict was created when one user submitted their changes; in this example Gale. When the server creates version #2 of the file it knows that Bruno is working on an older version of the file (which is no longer the #head), and thus there is a conflict.

So, how does Perforce communicate the conflict situation, and what will happen if Bruno tries to submit his changes without resolving the conflict? When P4V performs a refresh of the display it shows that there is a newer version of the file Bruno is working on. Since Bruno's version is out of date, a (yellow) triangle decorates the icon to indicate the version status. The tooltip provides Bruno with additional information about the conflict, as seen here:

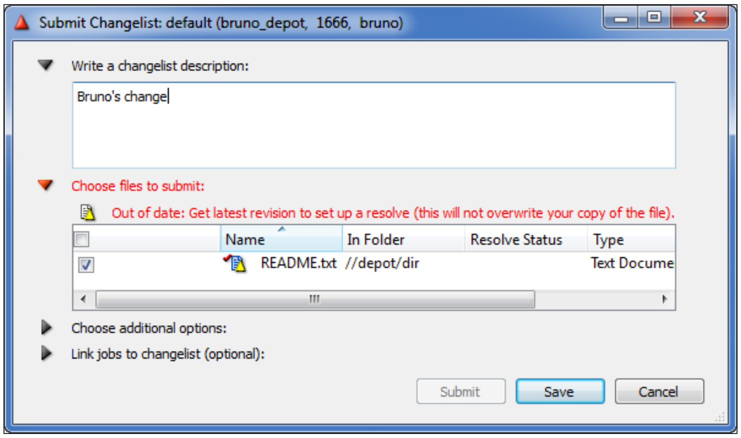


README.txt
checked out in default changelist
not latest revision of file (#1 of 2)

Type: Text Document
Size: 132 Bytes

The tooltip is explaining that he now has an "old" version of the file.

If Bruno attempts to submit his changes he will get this dialog:


Submit Changelist: default (bruno_depot, 1666, bruno)

Write a changelist description:

Bruno's change

Choose files to submit:

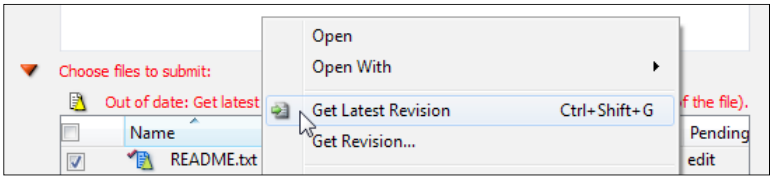
Out of date: Get latest revision to set up a resolve (this will not overwrite your copy of the file).

□	Name	In Folder	Resolve Status	Type
☑	README.txt	//depot/dir		Text Docume

Choose additional options:
Link jobs to changelist (optional):
Submit
Save
Cancel

He sees the warning that the file is **Out of date** and the **Submit** option is disabled. This will happen even if he has hundreds of files in his pending change list and only one of them has a conflict.

He has several options at this point. The first is to revert his changes, get the latest version from the repository, check it out, and redo his changes. While possible, this is potentially time consuming and seldom the best option. The second is to cancel the attempted submit and perform resolve actions on the files in the pending change list. However, when a small number of files are involved, it is also possible and indeed easy, to do everything without quitting the submit dialog:



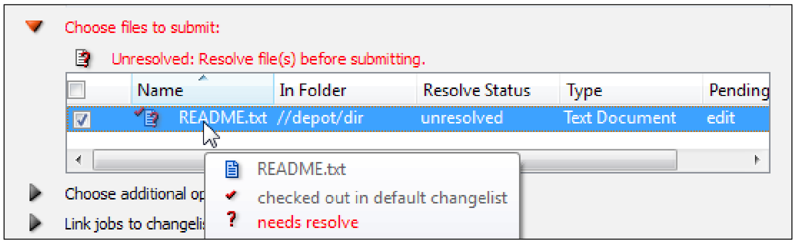
Choose files to submit:

Out of date: Get latest … f the file).

☐	Name	…	Pending
☑	README.txt	…	edit

Directly from the submit dialog, he can right-click and perform the **Get Latest Revision** action which P4V is suggesting.

As noted in the original submit dialog warning message, this "Get Latest" will not update the local copy (since that would mean his changes getting lost). Instead, the file is marked as being in the **conflict state**. Conflicts need to be resolved, hence, the **needs resolve** tooltip is then displayed:



Choose files to submit:

Unresolved: Resolve file(s) before submitting.

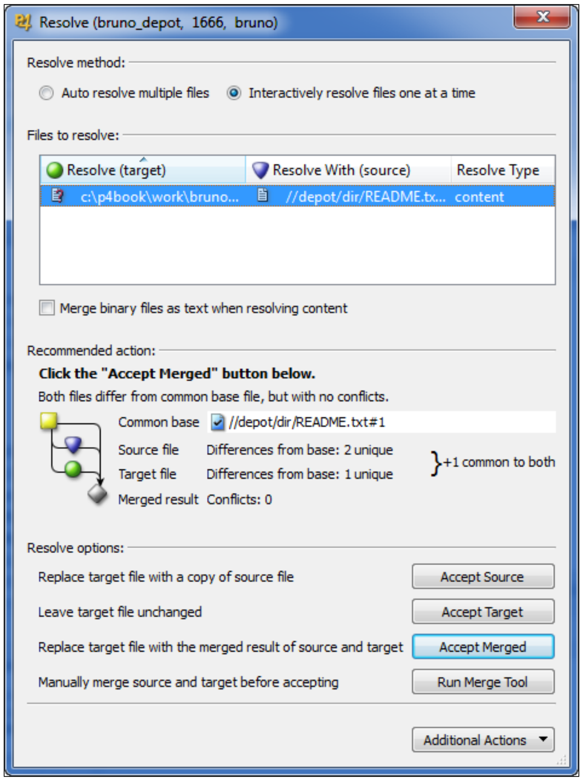
☐	Name	In Folder	Resolve Status	Type	Pending
☑	README.txt	//depot/dir	unresolved	Text Document	edit

README.txt
✔ checked out in default changelist
? needs resolve

Note that the icon decoration has changed from a (yellow) triangle to a (red) question mark. This indicates that a potential conflict has now become an acknowledged conflict that needs resolution. The server will not allow a submit if the pending change list is based on out of date files or contains a file with an acknowledged conflict.

An acknowledged conflict does not imply anything about the actual content of the file. It just means there were independent updates which need a human to resolve, or at least be consulted!

By right-clicking on the file and selecting **Resolve...** you will get the following dialog:



Resolve (bruno_depot, 1666, bruno)

Resolve method:

○ Auto resolve multiple files ● Interactively resolve files one at a time

Files to resolve:

Resolve (target)	Resolve With (source)	Resolve Type
c:\p4book\work\bruno...	//depot/dir/README.tx...	content

□ Merge binary files as text when resolving content

Recommended action:

Click the "Accept Merged" button below.

Both files differ from common base file, but with no conflicts.

Common base	//depot/dir/README.txt#1	
Source file	Differences from base: 2 unique	}+1 common to both
Target file	Differences from base: 1 unique	
Merged result	Conflicts: 0	

Resolve options:

Replace target file with a copy of source file

Leave target file unchanged

Replace target file with the merged result of source and target

Manually merge source and target before accepting

This dialog appears complex on the surface, but the basics are fairly straightforward:

- Perforce has detected a conflict state and a decision is required.
- The status information shown is guiding the user.
- The user needs to perform one of the **Accept** actions. This will tell Perforce that a decision has been made, and to change the state of the file so that it is no longer marked as "in conflict".
- The recommended **Accept** action is highlighted. Which one it is, depends on the specific file changes that have been made.
- Closing the dialog box will abort the resolve and leave all of the files unchanged.

In our preceding example, if Bruno clicks on **Accept Merged** and submits the file, he will get a clean merge result, and most likely all will be well. That is because his changes are able to be merged cleanly with Gale's changes (we cover other possibilities later in the *File content during merge* section of this chapter).

Base, Source, and Target: a 3-way merge

When merging the contents of a file, Perforce and many other source control tools use a technique known as 3-way merge. A 3-way merge provides users with difference information relative to a version of the file both users had in common. As we will see, this difference information makes it easier to identify and combine changes.

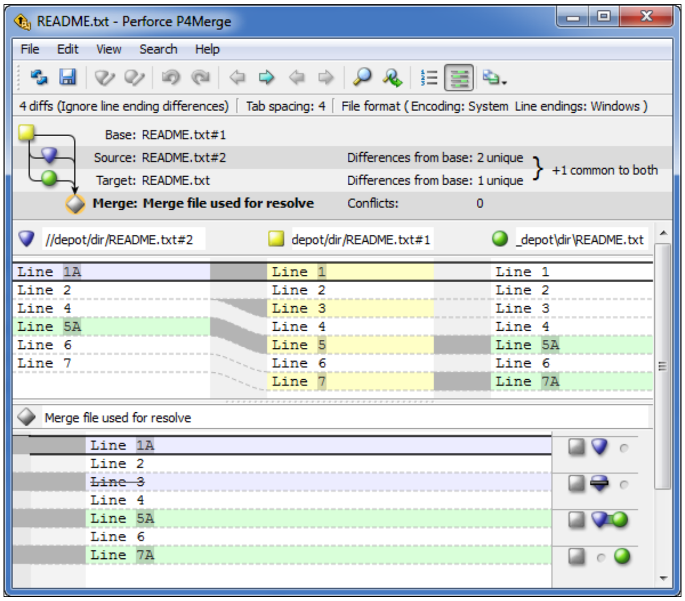
3-way merge uses three significant terms that relate to Perforce objects:

- **Target**: the workspace file that needs to be resolved.
- **Source**: the repository file version causing the conflict.
- **Base**: the repository file version that **Target** and **Source** have as a common ancestor.

> 3-way merge and 3-way merge tools pre-date P4V and include the command-line version of P4. These tools use the terminology **Theirs** instead of **Source**, and **Yours** instead of **Target**. For compatibility reasons you still see references to **Yours** and **Theirs** in error, log, and other messages within P4V.

An introduction to the P4Merge tool

It is also an option for Bruno to click on **Run Merge tool** as shown in the following example:



4 diffs (Ignore line ending differences) | Tab spacing: 4 | File format (Encoding: System Line endings: Windows)

Base: README.txt#1

Source: README.txt#2 Differences from base: 2 unique

Target: README.txt Differences from base: 1 unique } +1 common to both

Merge: Merge file used for resolve Conflicts: 0

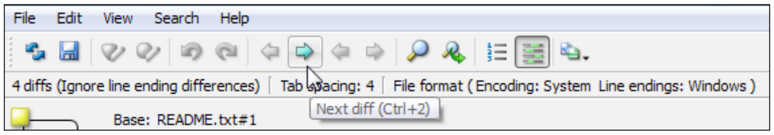
//depot/dir/README.txt#2	depot/dir/README.txt#1	_depot\dir\README.txt
Line 1A	Line 1	Line 1
Line 2	Line 2	Line 2
Line 4	Line 3	Line 3
Line 5A	Line 4	Line 4
Line 6	Line 5	Line 5A
Line 7	Line 6	Line 6
	Line 7	Line 7A

Merge file used for resolve

```
Line 1A
Line 2
~~Line 3~~
Line 4
Line 5A
Line 6
Line 7A
```

This shows the base (common ancestor) in the middle pane, and source and target file version changes to the left and right respectively. The bottom pane shows the default merge results. The default results are also the automatic results.

The status bar at the top contains a summary of the differences. The tool bar icons allow you to step through the file, one difference at a time (shown below). These options also appear on the **Search** menu.



4 diffs (Ignore line ending differences) | Tab spacing: 4 | File format (Encoding: System Line endings: Windows)

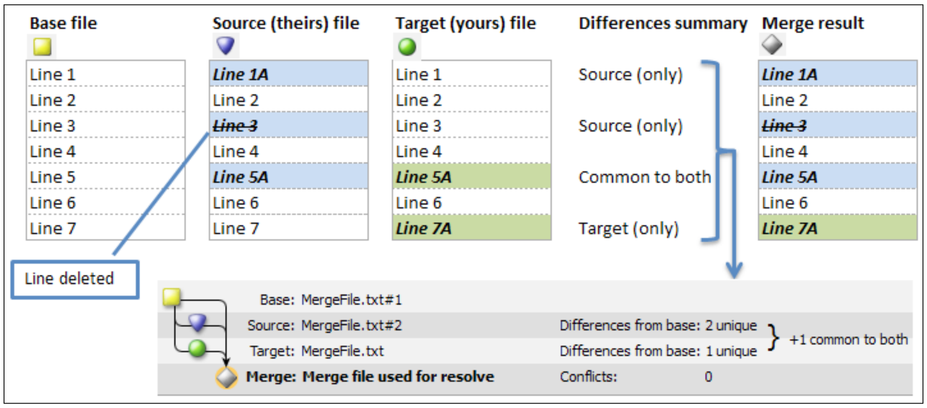
Base: README.txt#1

We will revisit this tool with an expanded example shortly. For now Bruno can just close the tool, and click on **OK** to the prompt **Do you want to save your changes to the merge file?** Alternatively he could use **File** | **Save** and **File** | **Exit** or click the appropriate toolbar icons. P4V saves the new merged file on top of his file in the workspace, and he can submit it without any further problems.

> Saving the merge results overwrites the workspace copy of the file. There is no automatic backup of the original contents.

Differences from base

Differences from base is what allows 3-way merges to determine a default set of merge actions. **Differences** are a sequence of one or more consecutive lines within a file. The following diagram shows a basic example. For simplicity in this example, all differences are a single line rather than the typical multi-line sequences normally encountered.



Base file	Source (theirs) file	Target (yours) file	Differences summary	Merge result
Line 1	***Line 1A***	Line 1	Source (only)	***Line 1A***
Line 2	Line 2	Line 2		Line 2
Line 3	***~~Line 3~~***	Line 3	Source (only)	***~~Line 3~~***
Line 4	Line 4	Line 4		Line 4
Line 5	***Line 5A***	***Line 5A***	Common to both	***Line 5A***
Line 6	Line 6	Line 6		Line 6
Line 7	Line 7	***Line 7A***	Target (only)	***Line 7A***

Base: MergeFile.txt#1
Source: MergeFile.txt#2 Differences from base: 2 unique
Target: MergeFile.txt Differences from base: 1 unique } +1 common to both
Merge: Merge file used for resolve Conflicts: 0

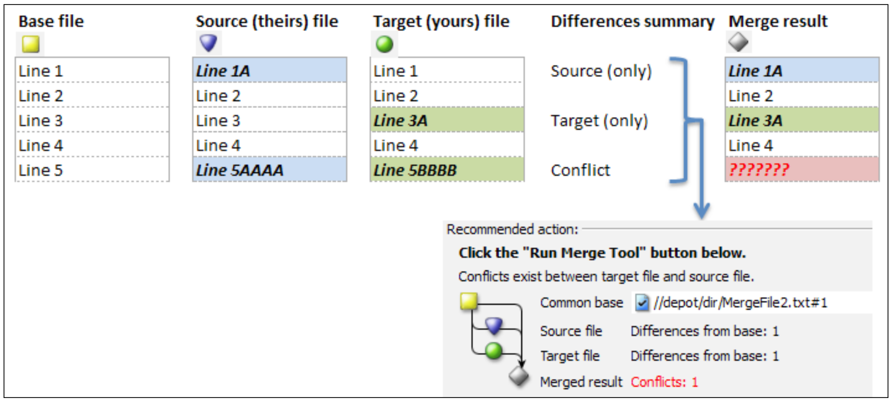
As we can see in the **Differences summary** column and the combined screenshot excerpt from P4Merge, there are two places where the source file differs from the base, and just one place where the target file differs from the base. Where a difference is common to both (**Line 5A** as shown in the preceding screenshot), it means that the same character-for-character change was made in both the source and target. Note that a deletion (**Line 3** as shown in the preceding screenshot) is just treated as a difference, it is not a special type of difference.

In the preceding example, P4V will merge the changes to give us the result shown. Because none of the changes overlap with each other the merge is easy to understand and in most (but not all) cases is going to be correct!

Dealing with content conflicts

So far we've dealt with merges where every difference is either common to both files or unique to one file. But what happens when different changes are made to the same section in both files? This is known as content conflict and requires human resolution.

An example of content conflict is shown as follows:



Base file	Source (theirs) file	Target (yours) file	Differences summary	Merge result
Line 1	*Line 1A*	Line 1	Source (only)	*Line 1A*
Line 2	Line 2	Line 2		Line 2
Line 3	Line 3	*Line 3A*	Target (only)	*Line 3A*
Line 4	Line 4	Line 4		Line 4
Line 5	*Line 5AAAA*	*Line 5BBBB*	Conflict	*???????*

Recommended action:

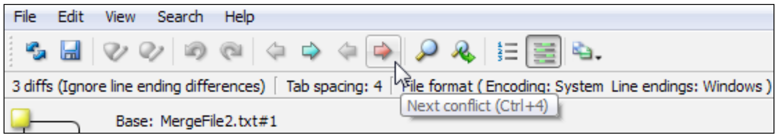
Click the "Run Merge Tool" button below.

Conflicts exist between target file and source file.

Common base	//depot/dir/MergeFile2.txt#1
Source file	Differences from base: 1
Target file	Differences from base: 1
Merged result	Conflicts: 1

As we can see, there are conflicting changes to line `5` and the summary shows **Conflicts: 1** (in red). Perforce will merge the non-conflicting changes (lines `1` and `3`) but the recommendation to **Run Merge Tool** means the user needs to decide what the appropriate action to take should be.

When there are conflicts present, the (pink) toolbar icon to go forward and backward to the next conflict is enabled, as shown in the following screenshot:

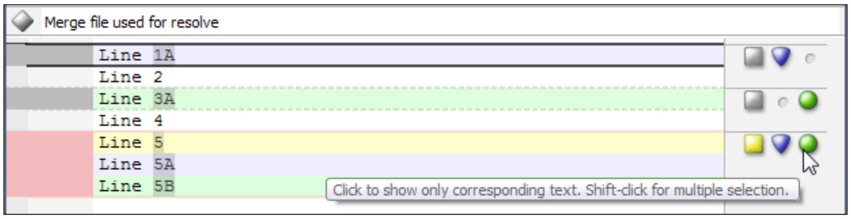


3 diffs (Ignore line ending differences) | Tab spacing: 4 | File format (Encoding: System Line endings: Windows)

Next conflict (Ctrl+4)

Base: MergeFile2.txt#1

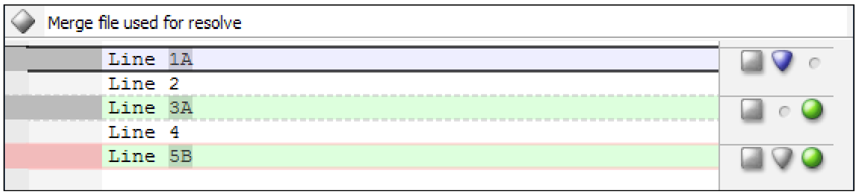
In this instance the merge pane shows the merged lines and has the conflict lines highlighted (in pink):


Merge file used for resolve

```
Line 1A
Line 2
Line 3A
Line 4
Line 5
Line 5A
Line 5B
```

Click to show only corresponding text. Shift-click for multiple selection.

By clicking on the appropriate icon: (yellow) square for base, (blue) triangle for source, or (green) circle for target, the user can select the content for the merge result:

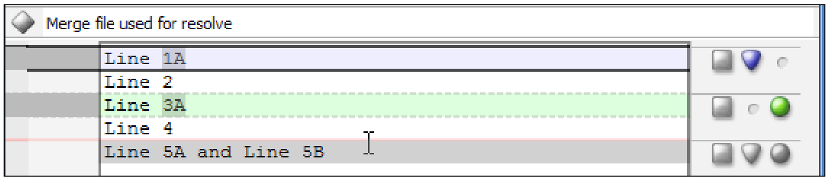

Merge file used for resolve

```
Line 1A
Line 2
Line 3A
Line 4
Line 5B
```

In this example he has chosen to include only the target version. Notice that for lines `1` and `3`, the respective icons are highlighted to show which version was selected.

Editing in the merge pane

Sometimes the best thing to do is to click in the merge pane and directly edit the file to get the results that you want:


Merge file used for resolve

```
Line 1A
Line 2
Line 3A
Line 4
Line 5A and Line 5B
```

In this instance the icons for `Line 5` are all greyed because none of them is directly being used. Instead, the resulting text is marked (also in grey).

> If you're doing follow-along, set up this conflict example and explore it. You can create the conflict using two users and two workspaces. Or, you can have one user use two workspaces to create the correct sequence of checkout and submit.

File content during merge

The interpretation of file content is a human activity. Perforce just looks for blocks of text, without considering any possible meaning or semantics, and merges them.

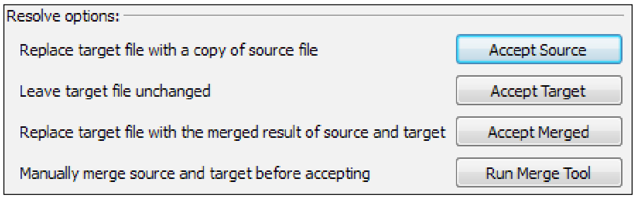
How often does the automatic merge do the right thing? It will all depend on the types of files being merged and their content. In our experience, the majority of merges give the correct semantic result. Of course, especially with source code, what looks like a correct result may not actually be correct.

> Any time you do a merge, it is always a good idea to test the result. If a merge breaks the build because the resulting source file won't compile, then that is easy to spot. It is more dangerous when a merge looks OK, compiles successfully, but has introduced a subtle bug. Good automated unit and functional tests can really pay dividends in such situations.

Some files are inherently difficult to merge in such a textual fashion (for example, `XML` files). With experience, you will understand what types of code or changes will merge easily and safely. When in doubt, step through the merge manually, difference by difference.

Overwriting or discarding changes on purpose

We discussed the resolve actions earlier in the chapter, but didn't cover what **Accept Source** or **Accept Target** did:



Resolve options:

Replace target file with a copy of source file

Leave target file unchanged

Replace target file with the merged result of source and target

Manually merge source and target before accepting

In the earlier scenario, Gale's changes are in the repository as revision `#2`. These changes are also the source of a 3-way merge. Bruno's changes are still in his workspace as the target of the merge.

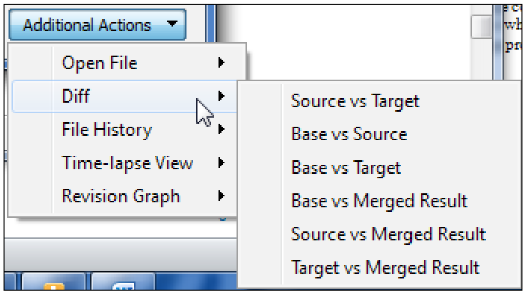
If Bruno selects **Accept Target**, what happens? P4V will leave his file unchanged, consider the conflict resolved and allow him to submit his file. The contents from Bruno's workspace will overwrite Gale's version in the repository. It's up to Bruno to determine if this is appropriate, and Gale may have an alternate opinion!

On the other hand, if he selects **Accept Source**, what happens? P4V will overwrite the file in Bruno's workspace, consider the conflict resolved and allow him to submit his file. It's up to Bruno to determine if that is appropriate (as we discussed in *Chapter 6*, *Managing Workspaces*, workspace defaults may result in no file being submitted if his file is not different to the latest version in the repository). Of course in this case he could also simply revert the file and there would be no conflict and nothing to submit.

> In some circumstances, a merge can appear too complex to merge using P4Merge. In such situations it may help to save a copy of your changes locally, do an **Accept Source**, and then reapply your changes on top. With experience you will know which option works well in which situation.

Other P4Merge options

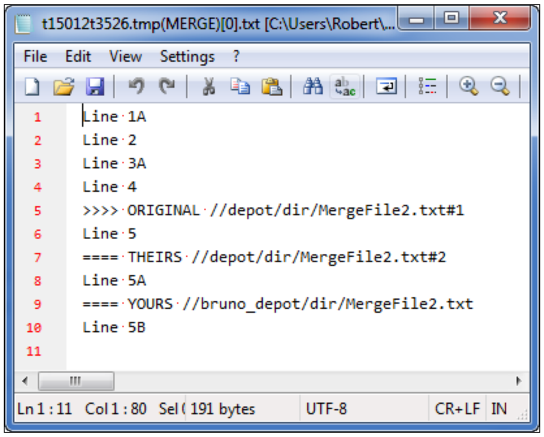
At the bottom of the resolve dialog is an **Additional Actions** button which offers a variety of potentially useful options:



If you are trying to work out what to do about a conflict, you can perform comparisons between various combinations of files. When it refers to **Merged Result**, this is a temporary file that P4V creates, which contains the results of an automatic 3-way merge.

This can be very helpful in understanding the history of the file and also what might be the most appropriate way to resolve the conflict.

You can go to **Open File | Merged Result** to see the contents of the temporary merge in your favorite editor:


t15012t3526.tmp(MERGE)[0].txt [C:\Users\Robert\...
File Edit View Settings ?

```
Line 1A
Line 2
Line 3A
Line 4
>>>> ORIGINAL //depot/dir/MergeFile2.txt#1
Line 5
==== THEIRS //depot/dir/MergeFile2.txt#2
Line 5A
==== YOURS //bruno_depot/dir/MergeFile2.txt
Line 5B

```

Ln 1 : 11 Col 1 : 80 Sel (191 bytes UTF-8 CR+LF IN

However, note that P4V is showing us special conflict markers around the blocks of text. These markers are used by legacy merge tools so don't expect them to change from **ORIGINAL** (base), **THEIRS** (source), and **YOURS** (target). Although this option can be useful, most users stick with the graphical merge tool.

The way to resolve conflicts when shown like this is to manually edit the file, removing the markers and leaving behind the desired file contents. When you save and exit, the file will be assumed to be the results of the merge that you want to keep.

What if you miss a conflict?

Despite due diligence, you are bound to sometimes miss conflicts that exist in a pending change list prior to submission. After all, P4V can only create icon decorations and tooltips based on what it knows. If the version information known to the server changes after the most recent P4V request for status information there will be conflicts you don't know about.

Not to worry. The worst case scenario is that the server rejects your submit. You will need to resolve the conflicts before the submit will be allowed, but there are no changes to the repository to deal with.

> If your submit fails, then the files in the pending change list may be locked. We recommend that to avoid issues with your colleagues, you either unlock the files or you resolve the conflicts as soon as you can. See the discussion on locking next.

Delaying resolution

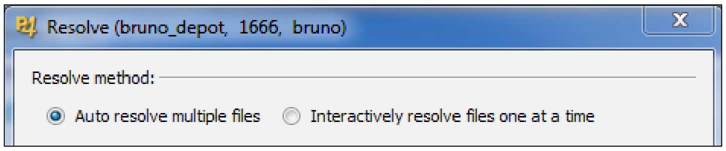
There is nothing that requires that a conflict be resolved as soon as it is detected. You can delay resolution until just before you submit. However, keep in mind that conflict resolution can change file content. Also remember content changes impact your workspace environment, which in turn has a potential impact on testing that happens with content from before the conflict resolution.

> Delays are fine, but conflicts must be resolved before you can submit a file. Best practice is to resolve conflicts as soon as possible.

Automatic resolution

To this point in the chapter, we have been discussing interactive resolves which work well when you are dealing with a small number of files.

There is an **Auto resolve** option which is selected at the top of the resolve dialog, shown below:



Resolve (bruno_depot, 1666, bruno)

Resolve method:

● Auto resolve multiple files ○ Interactively resolve files one at a time

This option allows you to perform the same resolve action on all of the files in the pending change list. This can be a major time saver when lots of files are involved.

However, we tend to find that this option is most useful, indeed essential, when dealing with merging between branches, so we will cover the details in *Chapter 8, Classic Branching and Merging*.

Avoiding conflicts

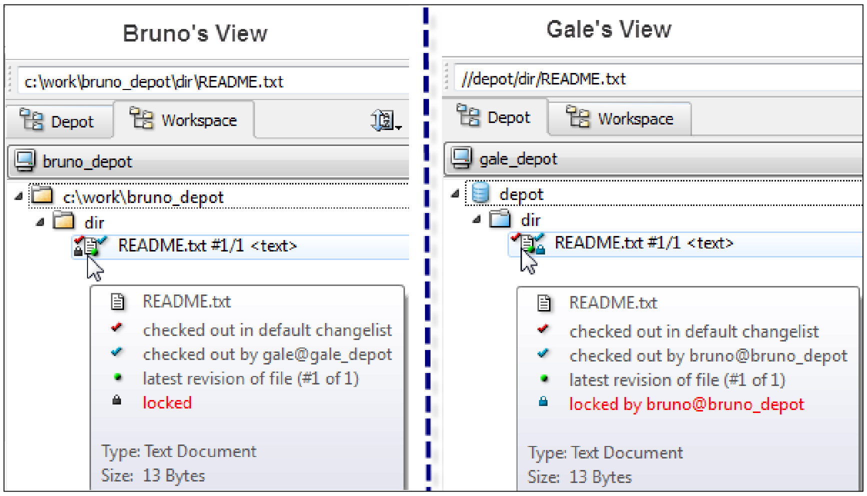
While we encourage you to allow Perforce to manage conflicts, there are times when it is best to avoid them. In particular, this can be true when your changes have priority over other peoples' or when a file cannot easily be merged. We'll cover both of these scenarios in this section.

Priority modifications – locking files

Sometimes you need to take priority when modifying a file. In Perforce, this equates to guaranteeing that you are the next person to submit a file. The mechanism is called locking a file. Note that needing priority is common after a failed submit, so P4V automatically locks all of the files in a failed submit.

Although it is not a best practice, you can always explicitly lock any file you have checked out. The most common way is to select **Lock** from the context menu in the tree panel or the pending change lists view panel.

Next, we see the result of Bruno locking a file for both himself and Gale:



README.txt
checked out in default changelist
checked out by gale@gale_depot
latest revision of file (#1 of 1)
locked

Type: Text Document
Size: 13 Bytes

README.txt
checked out in default changelist
checked out by bruno@bruno_depot
latest revision of file (#1 of 1)
locked by bruno@bruno_depot

Type: Text Document
Size: 13 Bytes

Notice the padlock icon decoration indicating a locked file (the decoration is on the left or right of the file icon depending on who is viewing it). The tooltip provides valuable information about the lock.

If a file is locked, then only the person who locked it is able to submit the file. Other people can check out the file, but will not be able to submit their changes until the lock is released.

Locks are not intended for general use. If everyone immediately locks files when they are checked out overall team productivity will suffer. The best practice is to lock a file for the shortest time possible. If you can't resolve your issues quickly, it is best to unlock the file and let Perforce manage conflicts in the usual way.

Locked files are automatically unlocked if the file is reverted or the file is part of a successful submit. Files can also be explicitly unlocked using the **Unlock** menu option in the context menus of the tree panel or the pending change lists view panel.

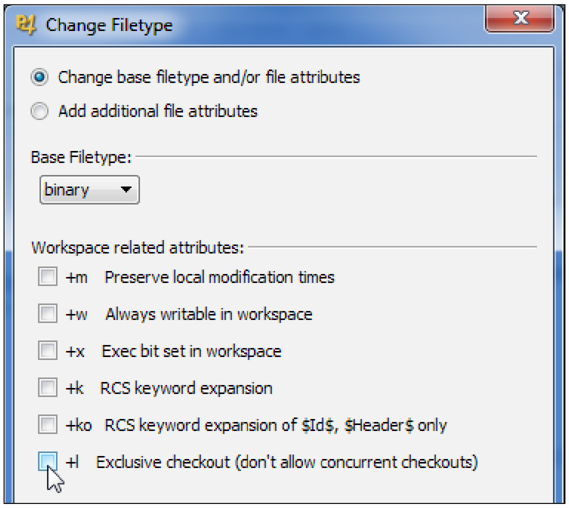
Unlike some other SCM tools with lock capabilities, Perforce allows users to checkout a file locked by another user. This behavior allows the Perforce server to track intended changes. Better to track a pending change than potentially lose track of it.

Files that can't be merged

Some files don't have effective merge tools. Spreadsheets, documents, graphics, and other binary formats are common examples of these files. Because they can't be merged, you can only sensibly resolve conflicts by accepting either the source or the target. This makes it important to avoid conflicts with these files. Otherwise, you might have a potentially significant amount of work manually merging content changes.

Closely monitoring the checking out of such files might avoid conflicts, but it's not a viable strategy. You could attempt to lock such files one-by-one as discussed earlier, but that only tends to delay the conflict discovery. A better solution is the exclusive checkout Perforce file type.

You change the file type by right-clicking on the file and then selecting **Change Filetype...** option (also on the **Actions** menu this option is only available if the file is open for some action) shown as follows:



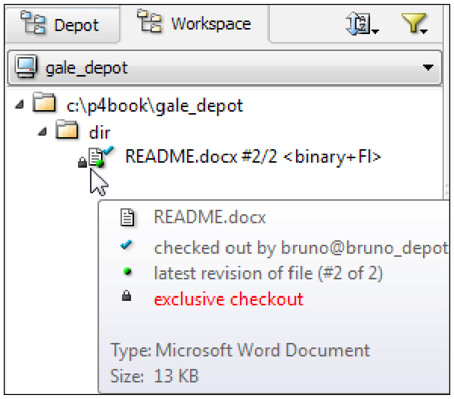
Change Filetype

● Change base filetype and/or file attributes
○ Add additional file attributes

Base Filetype:
binary

Workspace related attributes:
□ +m Preserve local modification times
□ +w Always writable in workspace
□ +x Exec bit set in workspace
□ +k RCS keyword expansion
□ +ko RCS keyword expansion of Id, $Header$ only
□ +l Exclusive checkout (don't allow concurrent checkouts)

Check the **+l Exclusive checkout** attribute, leaving the base file type unchanged. The base type is likely to be binary since 3-way binary merge tools are relatively rare. However, any file type can be exclusively checked out. Submit the file once you have changed its file type. Another user (such as Gale) might see this result:



gale_depot

c:\p4book\gale_depot

dir

README.docx #2/2 <binary+Fl>

README.docx

checked out by bruno@bruno_depot

latest revision of file (#2 of 2)

exclusive checkout

Type: Microsoft Word Document

Size: 13 KB

In the preceding example, Bruno has checked out a file which has the Perforce exclusive lock file type. The tooltip tells Gale that an **exclusive checkout** is in force, and she will get an error message if she attempts to check the file out.

Summary

Conflicts are a natural consequence of concurrent development. In this chapter, we've seen that dealing with conflicts only looks complex. In most cases, the identification and resolution of conflicts is actually rather straightforward. Most users only need to apply a small set of the available Perforce features for most of their conflict resolution needs. However, when more advanced techniques are required, those are available too. Perhaps most importantly, Perforce does not allow accidents. The server ensures that users acknowledge and resolve conflicts before it allows updates to the repository.

In the next chapter, we'll look at classic branch and merge operations, and we will see how merging between branches builds on the conflict resolution we went through in this chapter.

8
Classic Branching and Merging

Branching is the key version control technology for managing parallel development. Classic Perforce branching gives you complete control over the entire range of branching features. This supports almost any branching pattern that you can envision.

In this chapter, we'll cover the principles behind branching and merging. Then, we'll apply those principals to the branching patterns that you're most likely to encounter.

In this chapter we will cover:

- Branching principles
- Creating branches
- Maintaining branches
- Conflicts and branches
- Exploring branching history
- Branching patterns

Understanding branching

If you've been exposed to version control then you are probably familiar with the term branching. This is an area within the SCM community where both the terminology and the technology have evolved over time, which has generally been positive for users. An unfortunate side-effect is that, when you think about branching, it is very likely that you're thinking about the implementation characteristics of a particular tool. If this is your situation then clear your thoughts! Attempting to mentally equate classic Perforce branching to the implementation of another version control tool can be very confusing.

Branching is about managing the relationship between the files in two or more branches. This is a great definition except that it requires you know the definition of branch: a **branch** is a set of files that originated as an exact copy of another set of files and has evolved independently since the copy was made.

Before we finish our exploration of definitions, we need to address two additional terms that are often associated with branching: **codeline** and **mainline**. Technically, the term codeline refers to how a set of files is related. Most people use the terms codeline and branch interchangeably. A mainline is the equivalent of the trunk in some other tools such as Subversion. In classic Perforce branching, any branch can be the mainline for a project. It is just about being classified as, and used as, the mainline by the users. Common practice is to call it `MAIN` or a similar name. We will discuss this and other related branching patterns in more detail at the end of this chapter.

Why you should branch

Branching enables controlled parallel development. As such, it's the key to supporting important development activities such as:

- Multiple releases of software products
- Experimental feature development
- Isolating high risk development
- Concurrent feature development

There is overhead associated with managing branches. Without good tool support this overhead can impact productivity. Fortunately, Perforce was designed with class-leading branching capabilities from the start. With Perforce, the branching overhead is easily managed.

Using classic branching in Perforce

Classic Perforce branching gives you complete control over all aspects of branch management. The upside to this level of flexibility is that you can implement almost any branching pattern that you can envision. The downside is that there is nothing that keeps you from creating bad branching patterns! If you stick to the patterns we discuss in this chapter, you should be safe.

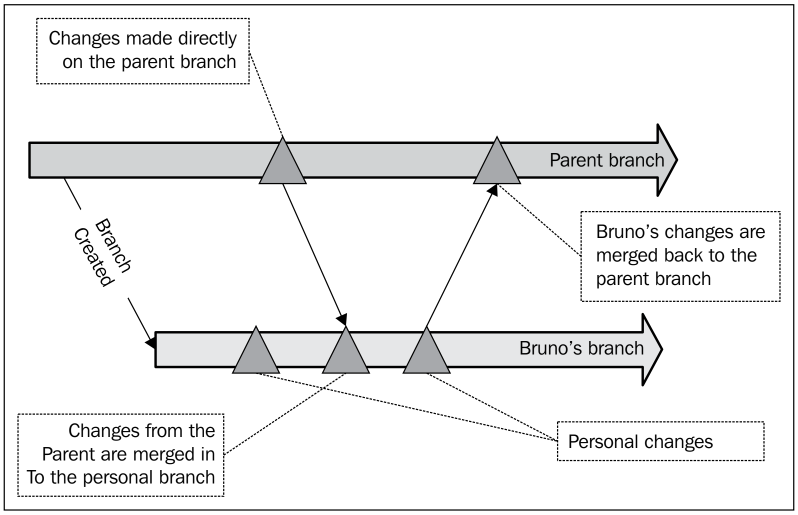
The basic branching pattern

While there is some commonality within the SCM community, the names associated with the various branching patterns tend to vary by tool and from organization to organization. Each branching pattern has a particular process associated with these activities:

- Creating the branch (from its parent)
- Making changes on the branch
- Updating a branch by propagating changes from its parent
- Propagating branch changes to its parent

The branching pattern that you are most likely to encounter is typically used to isolate changes until they are ready for a wider distribution. Such branches are often called project, feature, task, personal, or private branches. One or more developers may have workspaces that refer to such a branch. We will be talking about this pattern in most of this chapter, because it is the most common pattern used in most organizations. For example, release branches tend to be created and managed by only a small team within the organization. This makes release branches a less useful example.

Such a personal branching pattern might look like this:


Changes made directly on the parent branch
Parent branch
Branch Created
Bruno's changes are merged back to the parent branch
Bruno's branch
Changes from the Parent are merged in To the personal branch
Personal changes

We can see in this diagram that **Bruno's branch** is created from the **Parent branch**. Bruno makes changes to files in his branch. He also updates his branch from the parent to accommodate changes that others have made to the parent. Finally, when ready, Bruno propagates his changes back to the parent branch.

This pattern provides Bruno with several advantages:

- He is isolated from changes in the parent until he is ready to bring those changes into his branch.
- Changes to his branch are isolated from other developers who are using the parent or other branches.
- He decides when to expose his changes to other users of the parent branch (by propagating them to the parent).

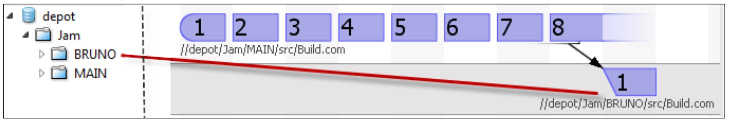
Of course this does not imply that Bruno is free from any overhead. As we will see, Bruno still needs to be involved with decisions about when to update or propagate branch content.

The major challenge for using this branching pattern is deciding the frequency with which you update your branch from the parent. Updating too frequently potentially wastes time. On the other hand, if you don't update often enough, your work can be destabilized when an update involves files that you have been working with in your branch. This being said, there is no hard and fast rule about update frequency. The appropriate update frequency may also change during the life of a project. We recommend doing it as often as you can, while taking into account the overhead required in doing so. It is fairly rare to find a situation where updates shouldn't be made at least once a week. Once a day, usually at the start or end of the day, is very common.

Viewing classic branches in Perforce

Perforce does not have a branch object per se. In fact, classic branching tracks individual file relationships. The fact that there are commands that aggregate a group of files and calls them a branch is a convenience for the user. It is not a requirement of the tool. You may see the words branch and merge in the P4V interface, but you will not see branch icons or other similar artifacts within the P4V GUI. We will discuss branch mappings, which have their own tab, later in the chapter (they are not directly branch objects).

So what do you see? Well, you see folders. It's only by convention that you know that a folder represents a branch. This is actually a very powerful system that allows an organization to adopt conventions and structures that meet their specific needs. Within a branch, each file will have a complete branching history, as shown in the following diagram:


depot
Jam
BRUNO
MAIN
1
2
3
4
5
6
7
8
//depot/Jam/MAIN/src/Build.com
1
//depot/Jam/BRUNO/src/Build.com

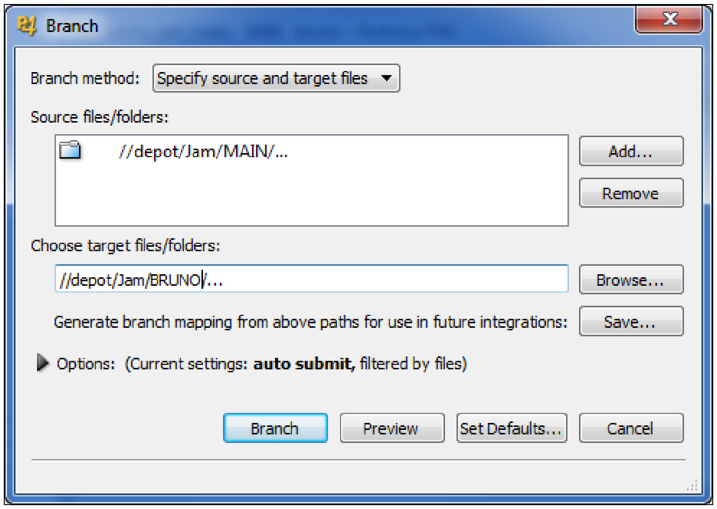
The preceding diagram combines a screenshot of the tree view and the Revision Graph for a file. We see two folders `//depot/Jam/`**`MAIN`** and `//depot/Jam/`**`BRUNO`**. We also see how one of the files in the code line, `//depot/Jam/`**`MAIN`**`/src/Build.com`, has been branched into the `//depot/Jam/`**`BRUNO`** code line as `//depot/Jam/`**`BRUNO`**`/src/Build.com`.

Creating our first branch

Let's start by creating a branch within Perforce.

> You will get a lot more from following along if you try the various actions as you read about them. Branching is one of those follow along activities where your mistakes will teach you more than your successes!

On the right-click menu for the folder we wish to branch, select the option **Branch Files...** to give the following dialog:



Branch

Branch method: Specify source and target files

Source files/folders:

//depot/Jam/MAIN/...

Choose target files/folders:

//depot/Jam/BRUNO/...

Generate branch mapping from above paths for use in future integrations:

Options: (Current settings: **auto submit**, filtered by files)

In this dialog we have chosen **Branch method** as **Specify source and target files**, and we have edited the target files/folders to be `//depot/Jam/BRUNO/`.... We want to use a simple repository structure convention where all code line branches for the Jam product are in the same folder in the repository (immediately under `//depot/Jam`). Such a simple convention works surprisingly well, but don't use it if you are going to have large numbers of branches (many tens or hundreds) at that level. Remember, the **...** (ellipsis) wildcard means all files and sub-folders below that path.

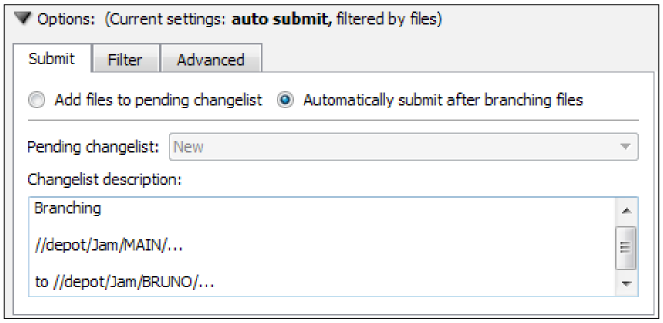
When we click on **Branch**, everything is done for us and is submitted to the repository. That is because the **Options** include **auto submit**.

We can check if the branch action has completed by looking in the depot tree, making sure we have set the filter to **Entire Depot Tree** (rather than **Tree Restricted to Workspace View**), and if necessary running **Revision Graph** on any of the files will confirm it has been branched.

> Versions of the Perforce server before the 2012.1 release required you to add the target files to your workspace before you are able to complete this action. Otherwise you would get the error message **No target file(s) in both client and branch view**.

An introduction to Options

The **Options** can be expanded by clicking on the triangle icon, as shown in the following screenshot:



▼ Options: (Current settings: **auto submit,** filtered by files)

○ Add files to pending changelist ● Automatically submit after branching files

Pending changelist: New

Changelist description:

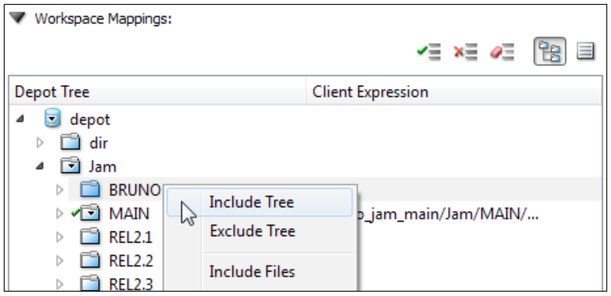
```
Branching

//depot/Jam/MAIN/...

to //depot/Jam/BRUNO/...
```

This shows the description of the changelist that will be automatically created by the `Branch` command. We will look at the other tabs later in this chapter.

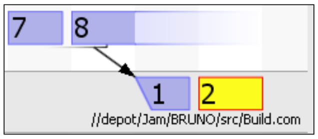
Propagating changes between branches

In order to be able to propagate changes, we need to make some changes on the branch first!

We can only modify the newly branched files by adding a view mapping to our current workspace view (**Connection | Edit Current Workspace...**) and then doing a **Get Latest** on them:

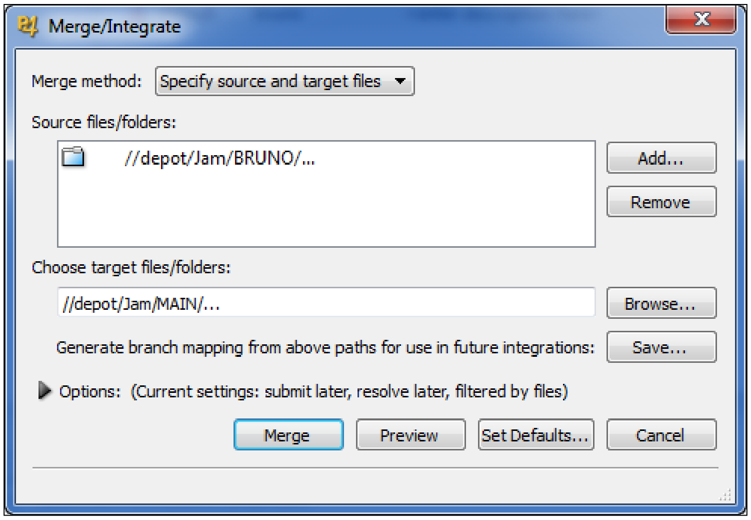


If we modify a file on the branch (checkout, edit it, and submit), the Revision Graph might be similar to:



We can see that revision 2 has been made and submitted. Don't forget that you can drag-and-drop one revision on to another to get a diff, or right click on the file and select **Diff Against Previous Revision**.

We start the process of propagating the changes by right-clicking and selecting **Merge/Integrate...** on the folder which is our source branch (`//depot/Jam/BRUNO`) to get this dialog:


Merge/Integrate

Merge method: Specify source and target files

Source files/folders:

//depot/Jam/BRUNO/...

Choose target files/folders:

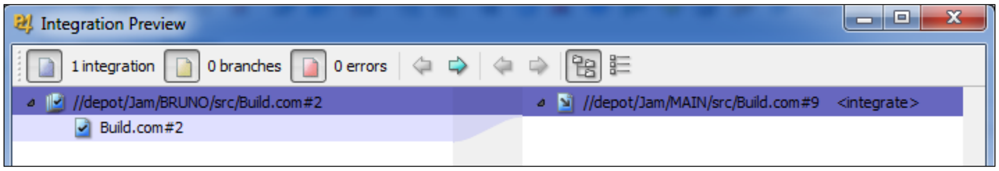
//depot/Jam/MAIN/...

Generate branch mapping from above paths for use in future integrations:

Options: (Current settings: submit later, resolve later, filtered by files)

Note that this time the target is `//depot/Jam/MAIN/...`.

We can click on the **Preview** button to get a list of all possible changes to be merged:

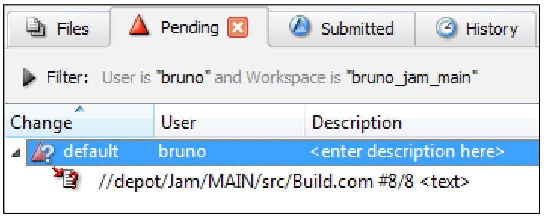


Integration Preview

//depot/Jam/BRUNO/src/Build.com#2	//depot/Jam/MAIN/src/Build.com#9 <integrate>
Build.com#2	

In this case, we see there is a single change to a file (`Build.com#2`) which will be propagated.

We can close the preview dialog, and by clicking on the **Merge** button we will get a changelist with the file marked as in the conflict state and thus needing to be resolved:

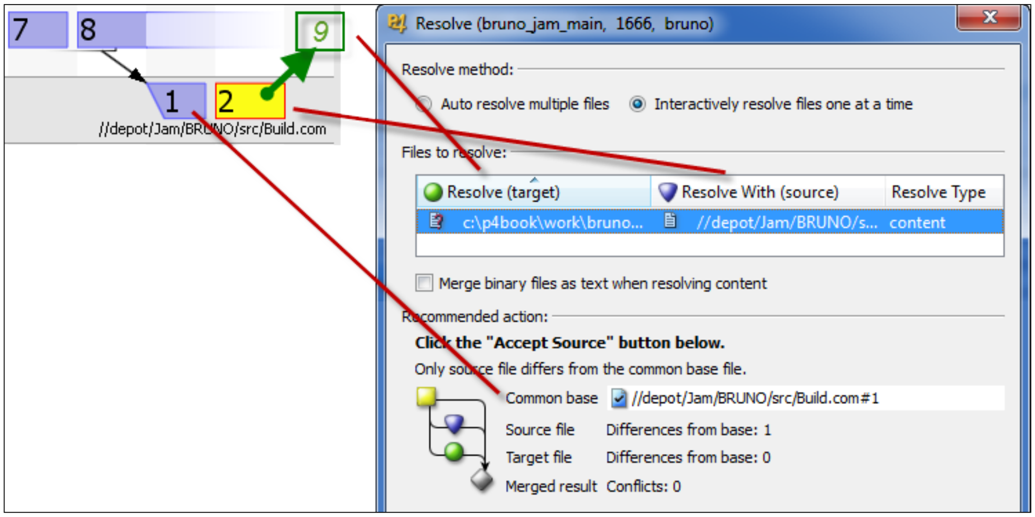


Filter: User is "bruno" and Workspace is "bruno_jam_main"

Change	User	Description
default	bruno	<enter description here>
//depot/Jam/MAIN/src/Build.com #8/8 <text>		

Resolving our merge conflicts

This is the same resolve that we saw in *Chapter 7, Dealing with Conflicts* on conflict resolution. Right-click on the file in the pending changelist and click on **Resolve…** to get:



Resolve (bruno_jam_main, 1666, bruno)

Resolve method:

○ Auto resolve multiple files ● Interactively resolve files one at a time

Files to resolve:

Resolve (target)	Resolve With (source)	Resolve Type
c:\p4book\work\bruno...	//depot/Jam/BRUNO/s...	content

□ Merge binary files as text when resolving content

Recommended action:

Click the "Accept Source" button below.

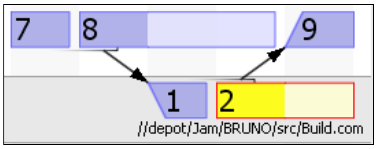
Only source file differs from the common base file.

Common base	//depot/Jam/BRUNO/src/Build.com#1
Source file	Differences from base: 1
Target file	Differences from base: 0
Merged result	Conflicts: 0

It's a standard 3-way merge, and as we can see in the diagram, the definitions are:

- **Common base**: which is `//depot/Jam/BRUNO/src/Build.com#1`
- **Source**: which is `//depot/Jam/BRUNO/src/Build.com#2`
- **Target**: the workspace file which will be checked in as `//depot/Jam/MAIN/src/Build.com#9`

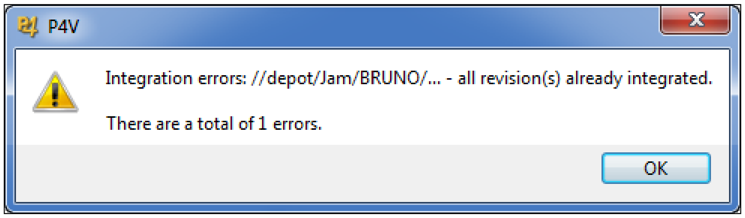
In this example, the recommended action is to click on the **Accept Source** button, because only the source file has changed. If we do that and submit our change, the Revision Graph will now look as follows:



Revision `9` has been created and the icon shows that it is a copy of revision `2`.

Perforce only propagates changes once

Normally, Perforce will only propagate a change or revision once. When a change has been propagated, successfully resolved, and submitted, Perforce will remember that it does not need to propagate that change in the future. If, after having propagated changes successfully between two code lines, we try to do it again, we will get this result:



Integration errors: //depot/Jam/BRUNO/... - all revision(s) already integrated.

There are a total of 1 errors.

This is very useful because we want Perforce to track what has been done on our behalf. Propagating changes would very quickly become impossible to manage at any scale without good tool support.

The meaning of integrate in merge/integrate

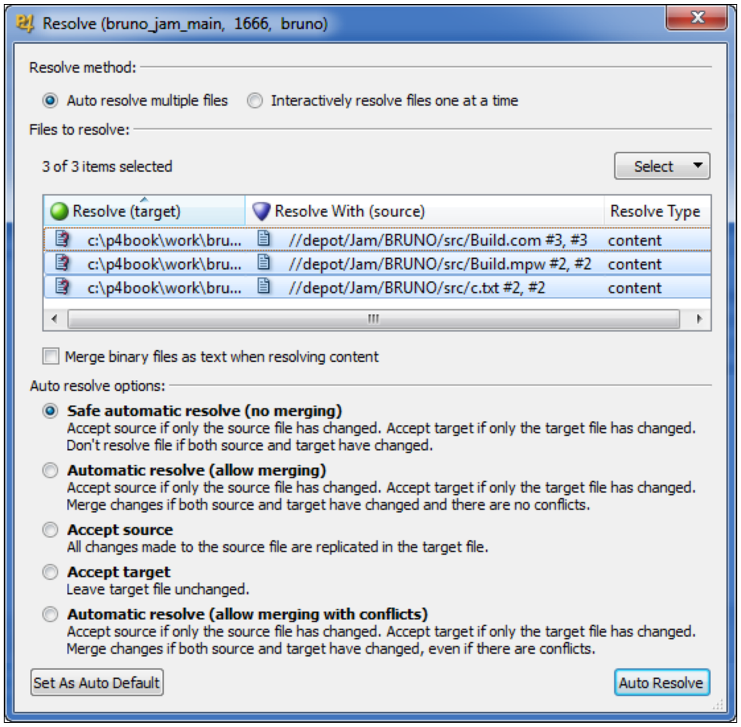
In today's SCM environment, merge and integrate are effectively synonyms for Perforce users. However, this hasn't always been the case. Perforce has supported classic branching and merging since its earliest days. Since the internal Perforce command that performs both branching and merging actions is called **integrate** that's the term that many Perforce users are familiar with. In those early days, merge was typically a reference to content. Yet the Perforce integrate command would also account for track added and deleted files. As the term merge has grown within the SCM community to encompass actions similar to what the integrate command provides, Perforce has adapted the wording within their user interfaces.

Automatic resolve

We deferred talking about the automatic resolve options in *Chapter 7, Dealing with Conflicts* because it really comes in to its own when propagating changes between branches.

It is not uncommon to change a large number of files on a branch. If we had to manually resolve conflicts, file-by-file, in order to propagate changes between branches, the overhead would quickly consume most of a developer's time. Yet we know that most of the time there will be no content conflicts that require human intervention. This is where the automatic resolve actions come into play. They provide the power of automation, and with the safety and control of human supervision. What does this look like?

On our pending changelist (not an individual file) we use the right-click option and click on **Resolve…** to get the dialog shown as follows:



Resolve (bruno_jam_main, 1666, bruno)

Resolve method:

● Auto resolve multiple files ○ Interactively resolve files one at a time

Files to resolve:

3 of 3 items selected

Resolve (target)	Resolve With (source)	Resolve Type
c:\p4book\work\bru...	//depot/Jam/BRUNO/src/Build.com #3, #3	content
c:\p4book\work\bru...	//depot/Jam/BRUNO/src/Build.mpw #2, #2	content
c:\p4book\work\bru...	//depot/Jam/BRUNO/src/c.txt #2, #2	content

□ Merge binary files as text when resolving content

Auto resolve options:

● **Safe automatic resolve (no merging)**
Accept source if only the source file has changed. Accept target if only the target file has changed.
Don't resolve file if both source and target have changed.

○ **Automatic resolve (allow merging)**
Accept source if only the source file has changed. Accept target if only the target file has changed.
Merge changes if both source and target have changed and there are no conflicts.

○ **Accept source**
All changes made to the source file are replicated in the target file.

○ **Accept target**
Leave target file unchanged.

○ **Automatic resolve (allow merging with conflicts)**
Accept source if only the source file has changed. Accept target if only the target file has changed.
Merge changes if both source and target have changed, even if there are conflicts.

This shows all of the files in that changelist that need resolving, three files in this case. Remember, we are propagating changes from source to target files, in this case from `//depot/Jam/BRUNO/`... to `//depot/Jam/MAIN/`....

The **Safe automatic resolve** option will copy any necessary changes from the source to the target files, but only if the corresponding target file has not been modified. This is considered safe because the file was only changed on one code line and not on both. If a file has been modified on both source and target code lines, then it will be necessary to merge changes, which is much less likely to be safe.

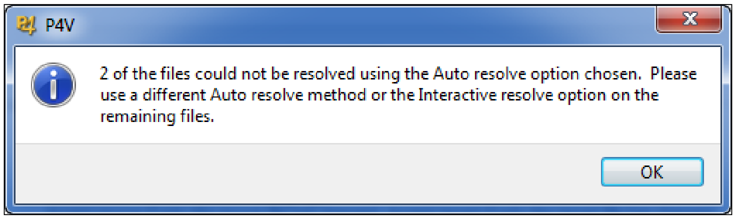
The **Automatic resolve** option will also perform any clean automatic merges that are necessary, so it does not process any files where there is a conflicting change. Remember that Perforce is only performing textual merges and doesn't understand the contents or meaning of any files, so you need to think about when this option is appropriate.

The **Accept source** option will result in any changed source files being copied to the target so that the target is character-for-character the same file. This will overwrite any changes in the target files, and you need to make sure you mean to do this! By contrast, the **Accept target** option is also known as the discard or ignore option. It leaves the target file unchanged, but marks the source file revision as having been propagated and will not consider them again in the future.

We use automatic merging all the time, and typically recommend a three step process:

- **Safe automatic resolve**: will typically process more than 50 percent of the files to be merged.
- **Automatic resolve**: we use this option when we have reasonable belief that the changes will merge cleanly and correctly. This depends on the code base we are merging.
- **Interactively resolve**: which we use for any remaining conflicts, one-by-one.

The results of a **Safe automatic resolve** might be as follows:

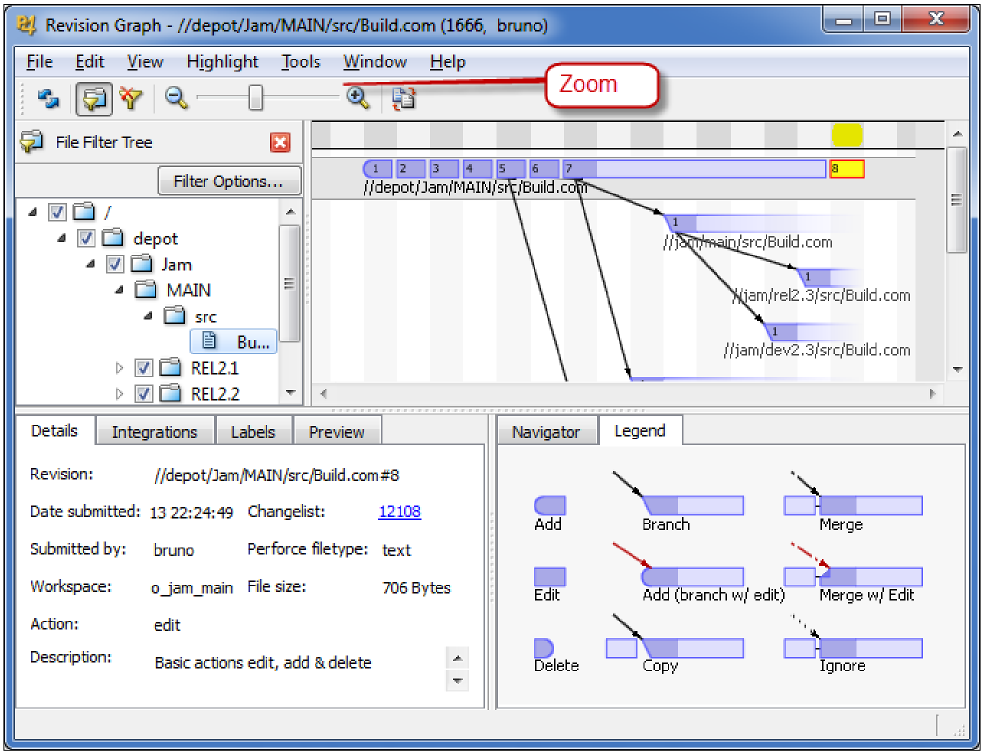


2 of the files could not be resolved using the Auto resolve option chosen. Please use a different Auto resolve method or the Interactive resolve option on the remaining files.

This then prompts the user to select the next level of resolve: either automatic resolve or interactive.

More on P4V Revision Graph

We have already seen some sample screenshots from Revision Graph earlier in the chapter. It is a powerful tool which conveys a lot of information.



Revision: //depot/Jam/MAIN/src/Build.com#8

Date submitted: 13 22:24:49 Changelist: 12108

Submitted by: bruno Perforce filetype: text

Workspace: o_jam_main File size: 706 Bytes

Action: edit

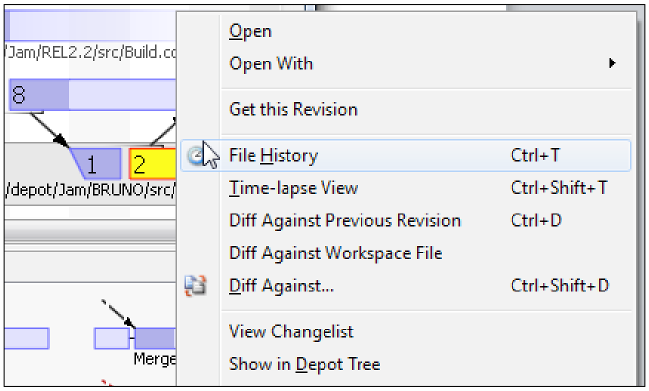
Description: Basic actions edit, add & delete

The **File Filter Tree** in the top-left is made visible via the relevant toolbar icon (also controlled via **View | Show/Hide | File Filter Tree**). It is very useful in deciding which code lines to make visible. It is not uncommon for there to be tens or hundreds of code lines for a single file and this allows you to focus on the ones that are interesting to your current activity.

The **Details** tab shows information for the currently selected file revision which is highlighted (in yellow) in the right-hand pane. It is very similar to the **Files** tab discussed in *Chapter 5, File Information*. The **Legend** tab describes each of the revision icons and their associated meanings. We will discuss these shortly. Note also the **Zoom** options to control the size of the right-hand pane.

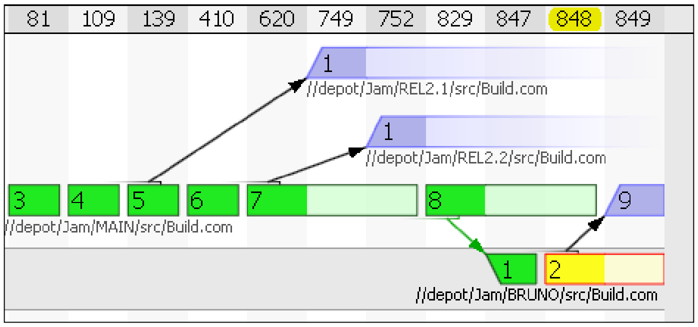
Diffing and other actions

There are lots of actions available on the right-click menu for a revision, as shown in the following screenshot:



The most commonly used ones are the various **Diff** options. However, most people just drag-and-drop one revision onto another to do a basic diff.

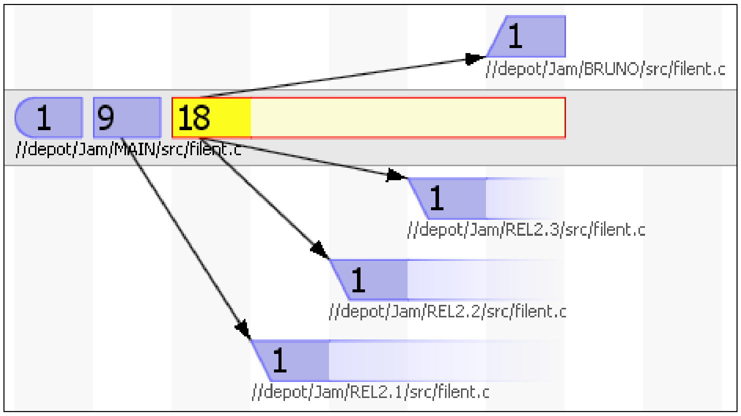
The **Highlight** | **Ancestors of selection** will color all ancestor revisions green, which starts to be very helpful when we look into the diagram to understand more complex graphs and branching history.


81
109
139
410
620
749
752
829
847
848
849
1
//depot/Jam/REL2.1/src/Build.com
1
//depot/Jam/REL2.2/src/Build.com
3
4
5
6
7
8
9
//depot/Jam/MAIN/src/Build.com
1
2
//depot/Jam/BRUNO/src/Build.com

It is not obvious from the printed screenshot, but revision 2 in the bottom right corner has been selected, and thus revision 1, and then from 8 down to 3 are all highlighted as ancestors.

Compressing the history

Another very useful feature for helping to understand complex histories is the **View | Compressed Integration History**. An example of this is shown as follows:


1
//depot/Jam/BRUNO/src/filent.c
1
9
18
//depot/Jam/MAIN/src/filent.c
1
//depot/Jam/REL2.3/src/filent.c
1
//depot/Jam/REL2.2/src/filent.c
1
//depot/Jam/REL2.1/src/filent.c

This shows that revisions between `1` and `9` and between `9` and `18` have been hidden. They are edits, which are considered less interesting (in this case) than revisions which have contributed to the branching history of the file.

> Whether or not you're doing follow along, now is a good time to explore the Revision Graph tool. Most people miss most of the power within this tool because they haven't taken the time to explore it.

Selectively propagating changes

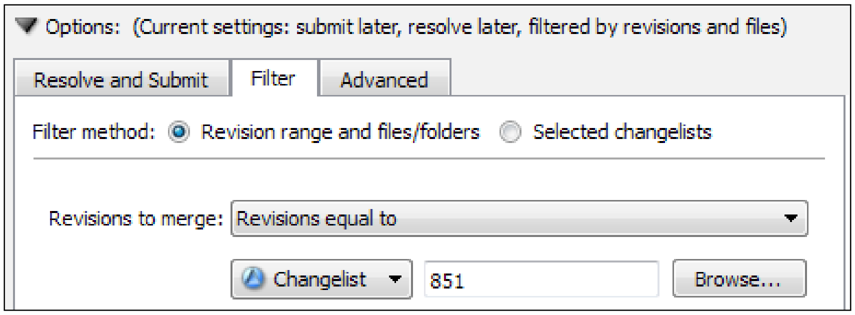
It is quite common for people to want to be able to select a single changelist to be propagated from one code line to another, instead of all the un-propagated changelists. Bruno might have fixed bug 3242 on his branch, which has modified several files. He is doing some other work on his branch as well, which he does not want to propagate back to `MAIN` yet. Meanwhile, Fred has been affected by the same bug and it would help him to have the bug fix.

The first solution that occurs to many people is to just email the particular file revisions with the bug fix to the requesting person (slightly more advanced, and more common on Unix/Linux, is to email patch files generated by the diff tool). If the changelist is a single file, then this appears superficially as a simple solution. But don't do it! It doesn't scale well to changelists with multiple files, and it also stores up trouble and makes future merges more complicated.

The Perforce way to handle this situation is to selectively merge that bug fix only, without any of the other changes to the other code line. Perforce will track this action and remember it when changes are propagated, which avoids a nasty type of duplicate merge problem.

> This is a very useful feature, and as we see in the following section, it is not difficult. However, it depends on how you commit changes, and will only work well if you are careful to keep each bug fix in its own changelist. If you mix things up, then this sort of action becomes much harder.

What are the steps to do this? We do a standard **Merge/Integrate**, selecting source and target files. We then use the **Filter** tab to be specific about the changelist to be merged. This filter option is:



Options: (Current settings: submit later, resolve later, filtered by revisions and files)

Filter method: ● Revision range and files/folders ○ Selected changelists

Revisions to merge: Revisions equal to

Changelist: 851

You can click on the **Browse...** button to interactively select a particular changelist, or you can just type it in as shown in the preceding screenshot.

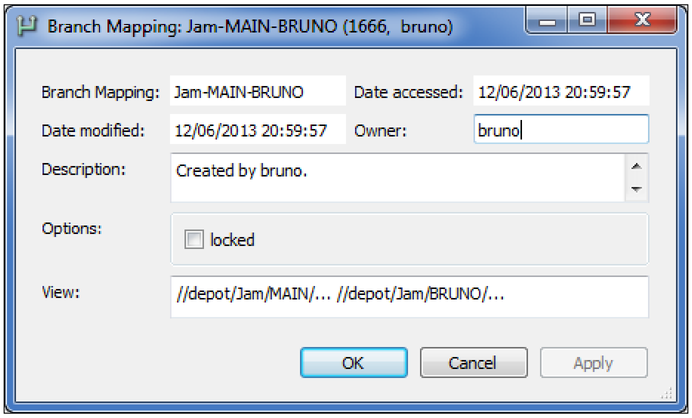
Then click on **Merge** and repeat the steps to resolve all changes before submitting your change.

> While selective propagation (also known as "cherry picking") can be very useful, and is often quite easy to do, it should not be something that you do all the time. If you find you are using it too frequently, then start thinking about your processes and branching patterns, and how you are organizing work between people.

Using branch mappings

In our examples so far in this chapter we have specified source and target files directly. It is quite common to use a **branch mapping** instead.

You create a mapping from the **Branch Mappings** tab or **File** | **New** | **Branch Mapping...**, seen here:



Branch Mapping: Jam-MAIN-BRUNO (1666, bruno)

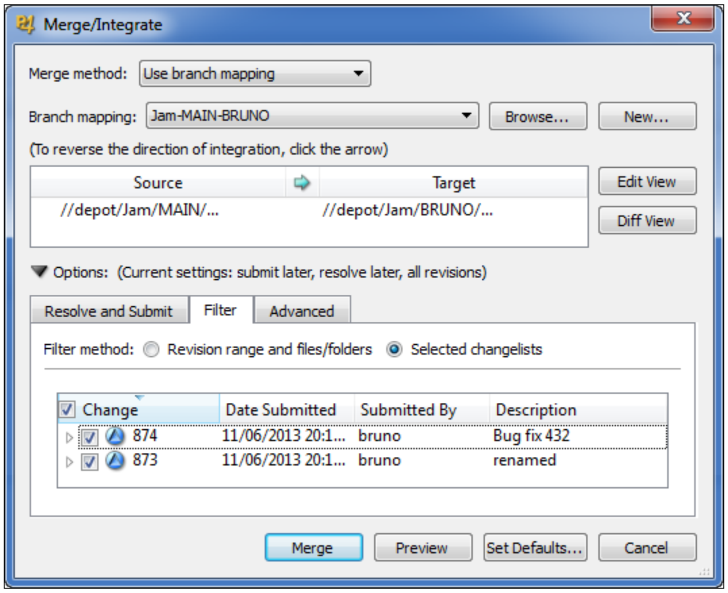
Branch Mapping: Jam-MAIN-BRUNO
Date accessed: 12/06/2013 20:59:57
Date modified: 12/06/2013 20:59:57
Owner: bruno
Description: Created by bruno.
Options: □ locked
View: //depot/Jam/MAIN/... //depot/Jam/BRUNO/...

As with other objects, such as workspaces, it is important to have a good naming convention for branch mappings. Consult your administrator for guidance. We have used a simple mapping called `Jam-MAIN-BRUNO` which reflects the name of the project and the two codelines involved.

The **View:** is particularly important and will need to be edited from the default. Like a workspace view, it is a mapping, but in this case it is a mapping between one area of the repository and another: `//depot/Jam/MAIN/...` to `//depot/Jam/BRUNO/...`.

> When having to type repository path names, we often find it convenient to copy the path from the address bar in P4V and paste it in to the appropriate dialog. If necessary, you can then edit it, and this is still likely to be less error prone than just typing the full value in.

Having saved our branch mapping, we can use it with right-click options to either **Branch Files...** or to **Merge/Integrate...**, as shown in the following screenshot:



Merge method: Use branch mapping

Branch mapping: Jam-MAIN-BRUNO

(To reverse the direction of integration, click the arrow)

Source	→	Target
//depot/Jam/MAIN/...		//depot/Jam/BRUNO/...

Options: (Current settings: submit later, resolve later, all revisions)

Filter method: ○ Revision range and files/folders ● Selected changelists

☑ Change	Date Submitted	Submitted By	Description
☑ 874	11/06/2013 20:1...	bruno	Bug fix 432
☑ 873	11/06/2013 20:1...	bruno	renamed

Use the **Browse...** button to select our branch mapping. Here we can see that it has been used to complete the **Source** and **Target** columns within the dialog.

This example also shows the **Filter** tab with **Selected changelists** checked, which would allow us to selectively propagate just one or several changelists if we want to.

The power of branch mappings

The underlying actions of **Branch** or **Merge/Integrate** actions will be exactly the same as if you specify source and target files directly. So what are the advantages of using branch mappings?

Probably the most important reason is that branch mappings may contain multiple view lines, including options such as excluding individual files or subfolders. Another example is bringing in libraries from other parts of the repository. This can be very useful, and we see it used a lot, particularly in larger organizations.

> In many organizations, the existence of a branch mapping indicates that a branch is in active use. Once the branch is no longer in active use the branch mapping is deleted, but it can always be recreated if necessary.

Merging – the gory details

It is useful to have some understanding of the algorithms Perforce uses for propagating and merging changes, and how it works out which changes to propagate/merge. This helps you to work with the tool instead of against it, and avoids surprises and getting yourself into strange situations. The full details of this are beyond the scope of this book, but this section gives an overview of the main principles.

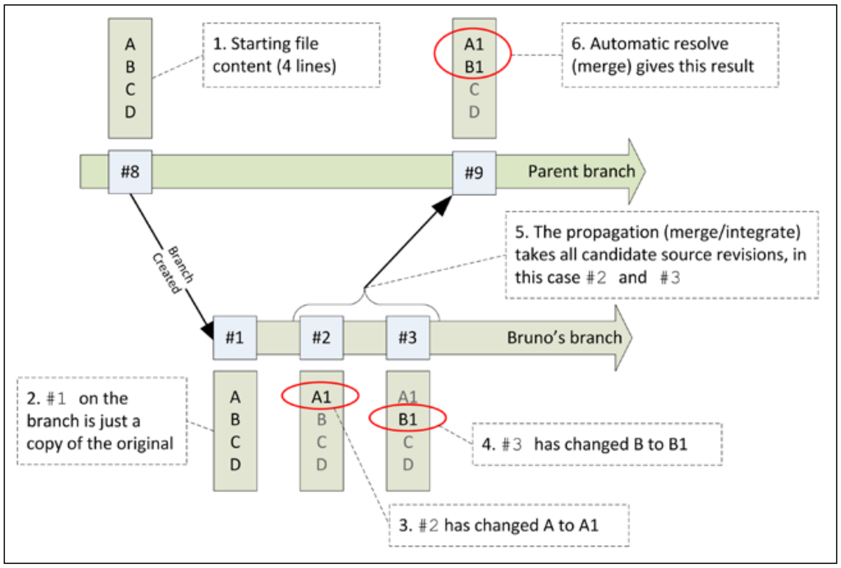
The main algorithm for merging changes between a set of source and target files is to prepare a list of individual source and target files (there may be tens, thousands, or more on two code lines). Then, for each pair of files:

1. Identify which source revisions are candidates to be merged, which includes ignoring any revisions which have already been merged in the past.
2. Decide how to merge those candidates according to any filter being applied by the user.

As we discussed earlier in this chapter, Perforce remembers all changes propagated and will not prompt us to propagate a change that has already been propagated.

Merges and file content

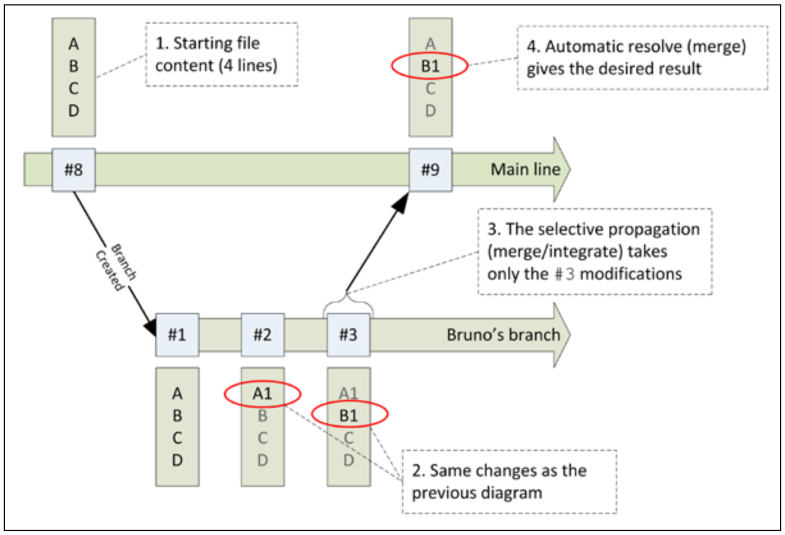
Perforce tracks merges by revision, but each merge is of file contents. This is a key distinction to make. Perforce looks at the revisions that need to be merged and then merges those changes, using the contents. In a simple example, as in the following diagram, the results seem fairly obvious:


A
B
C
D
1. Starting file content (4 lines)
A1
B1
C
D
6. Automatic resolve (merge) gives this result
#8
#9
Parent branch
Branch Created
5. The propagation (merge/integrate) takes all candidate source revisions, in this case #2 and #3
#1
#2
#3
Bruno's branch
2. #1 on the branch is just a copy of the original
A
B
C
D
A1
B
C
D
A1
B1
C
D
4. #3 has changed B to B1
3. #2 has changed A to A1

Note that the revision 1 (#1) of the branched file has the same contents as the original file. There are then two subsequent changes made, #2 and #3 with file contents as shown. The default merge will take all candidate revisions, which means both #2 and #3, and the automatic merge (resolve) gives the result shown. In this case, it looks like the desired result, and in most situations it will be just what we want.

How selective merges are done and tracked

If we consider the example of a selective merge of only a single revision, the algorithm is shown in the following diagram:


A
B
C
D
1. Starting file content (4 lines)
A
B1
C
D
4. Automatic resolve (merge) gives the desired result
#8
#9
Main line
Branch Created
3. The selective propagation (merge/integrate) takes only the #3 modifications
#1
#2
#3
Bruno's branch
A
B
C
D
A1
B
C
D
A1
B1
C
D
2. Same changes as the previous diagram

We only want the modification of B to B1 to be merged to the target. The key thing to understand is that this is the same as merging revision 3 only, and leaving behind the changes made by revision 2.

As we can see, there is a difference between the contents of the file at revision 3 (which contains A1 and B1) and the actual modification made by revision 3 (which contains B1 only).

> It is important to understand that in this example Perforce will track that #3 has been merged, but #2 has not. For future merges between these two code lines, #2 will be a candidate to be merged (unless explicitly excluded, for example, by another selective merge). This creates more complexity, and our advice is to avoid selective merges unless you really need them!

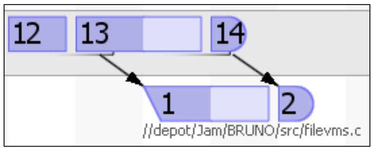
Understanding this concept is key to understanding Perforce's actions when merging or integrating changes.

Dealing with renames and deletes

These actions cause some complications in standard merging of changes between code lines. We will not address all possible situations in this section, but give you enough information to be a little wary and take extra care!

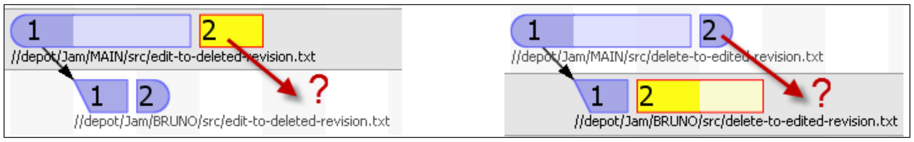
Dealing with deletes

In a simple example we have deleted revision `14` of a file and then merged that change to our branch, as shown in the following screenshot (remember the **Legend** tab to show the meaning of icons):



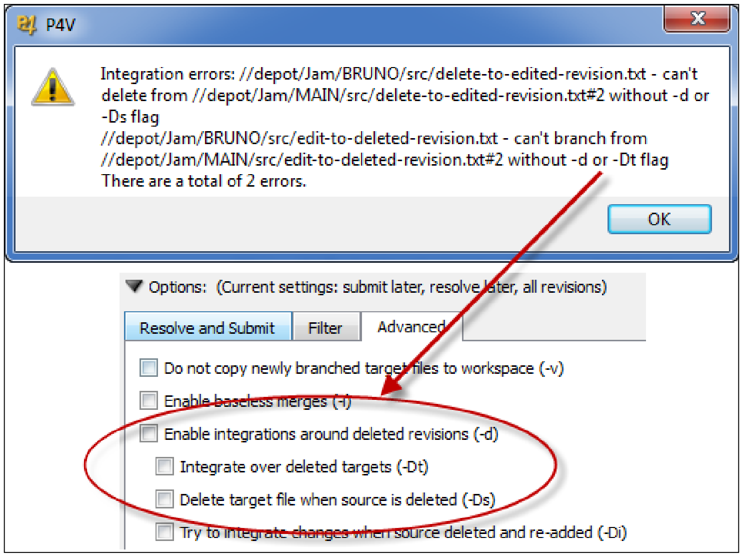
In this example, the merge action has just propagated the delete from source to target, and all is well.

However, life is not always so simple, and there are a couple of scenarios where the right answer is not obvious, and where you as the user need to decide what Perforce should do (by enabling flags).


1
2
//depot/Jam/MAIN/src/edit-to-deleted-revision.txt
1
2
?
//depot/Jam/BRUNO/src/edit-to-deleted-revision.txt
1
2
//depot/Jam/MAIN/src/delete-to-edited-revision.txt
1
2
?
//depot/Jam/BRUNO/src/delete-to-edited-revision.txt

In the first instance, we are attempting to propagate an edit on top of a deleted revision. In the second example, there is a delete to be propagated on top of an edited revision.

It is a fairly important decision as to whether a file should be deleted or recreated, or and edit be ignored or not, and P4V will not automatically make this decision for us. It will give us an error message on the **Merge** which we can override via advanced options:



Integration errors: //depot/Jam/BRUNO/src/delete-to-edited-revision.txt - can't delete from //depot/Jam/MAIN/src/delete-to-edited-revision.txt#2 without -d or -Ds flag
//depot/Jam/BRUNO/src/edit-to-deleted-revision.txt - can't branch from //depot/Jam/MAIN/src/edit-to-deleted-revision.txt#2 without -d or -Dt flag
There are a total of 2 errors.

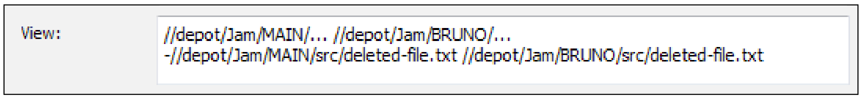
OK
Options: (Current settings: submit later, resolve later, all revisions)
Resolve and Submit
Filter
Advanced

- □ Do not copy newly branched target files to workspace (-v)
- □ Enable baseless merges (-i)
- □ Enable integrations around deleted revisions (-d)
  - □ Integrate over deleted targets (-Dt)
  - □ Delete target file when source is deleted (-Ds)
  - □ Try to integrate changes when source deleted and re-added (-Di)

The important point if you receive such an error is not to just immediately check the suggested option on the **Advanced** tab of the **Merge** dialog (shown in the preceding screenshot) and rerun the merge. Instead, first investigate and consider the future implications and think about what the correct result should be. If in doubt, consult with your administrator! It is better to delay than to rush in such situations.

Using branch mappings to ignore deletes

Another useful feature of branch mappings is that if you don't want a delete to be propagated between 2 code lines, then you can add a specific view line to the mapping removing the deleted file:



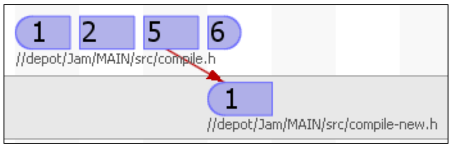
View:

```
//depot/Jam/MAIN/... //depot/Jam/BRUNO/...
-//depot/Jam/MAIN/src/deleted-file.txt //depot/Jam/BRUNO/src/deleted-file.txt
```

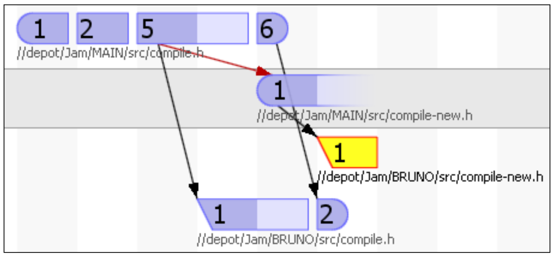
This example shows an exclude mapping (which begins with a minus or '-' character) for a specific file. With such a branch mapping, P4V will no longer attempt to propagate any changes, including delete actions to that file.

Dealing with renames

You rename a file by using the right-click option and selecting **Rename/Move…** command from the tree view. Behind the scenes, this basically just deletes the old file and adds a copy of it with the new name. In Revision Graph, this looks like:


1
2
5
6
//depot/Jam/MAIN/src/compile.h
1
//depot/Jam/MAIN/src/compile-new.h

Notice that the same changelist contains both the delete and the branch/copy. If we do a **Merge/Integrate** of this rename action, then we get:


1
2
5
6
//depot/Jam/MAIN/src/compile.h
1
//depot/Jam/MAIN/src/compile-new.h
1
//depot/Jam/BRUNO/src/compile-new.h
1
2
//depot/Jam/BRUNO/src/compile.h

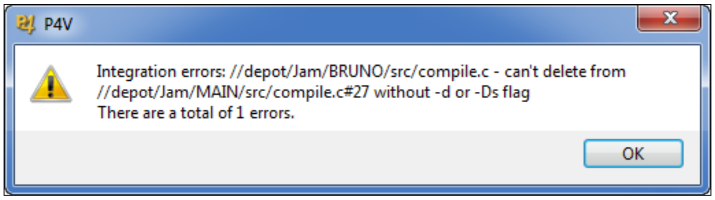
We can see that the delete of `compile.h` is propagated to the `BRUNO` code line and the new name `compile-new.h` is also added as a new file.

When renames cause complications

Life can become significantly more complicated when you have a file which is renamed on one code line, and yet on another related code line:

- You do not wish to rename the file, and yet you want any edits to be merged between the file with the old name and the file with the new name.
- Edits have occurred to a file on one branch which has been renamed on another branch.

Such situations give rise to **Merge/Integrate** errors such as:



Integration errors: //depot/Jam/BRUNO/src/compile.c - can't delete from //depot/Jam/MAIN/src/compile.c#27 without -d or -Ds flag
There are a total of 1 errors.

It is beyond the scope of this book to go in to all of the possible problem scenarios and what to do about them.

> Our main advice is to seek help first from an experienced Perforce user before doing anything such as setting option flags like d, Ds, or Dt. These flags can cause you more problems than they solve if you are not careful!

Other branching patterns

So far we've covered the most common branching pattern that you are likely to use in your day-to-day work. If you have understood what we've covered so far, then you have a fundamental understanding of all branching patterns as they all have similar characteristics. In this section, we'll help you understand those characteristics and then we'll use that information to cover some additional patterns that you're likely to encounter.

> The patterns discussed in this section are part of the mainline model. The mainline model is a time-tested method for supporting project development.

Characteristics of a branching pattern

When all is said and done, branching patterns can be characterized with answers to a relatively small set of questions. Review the following questions and consider how they apply to what you've already done:

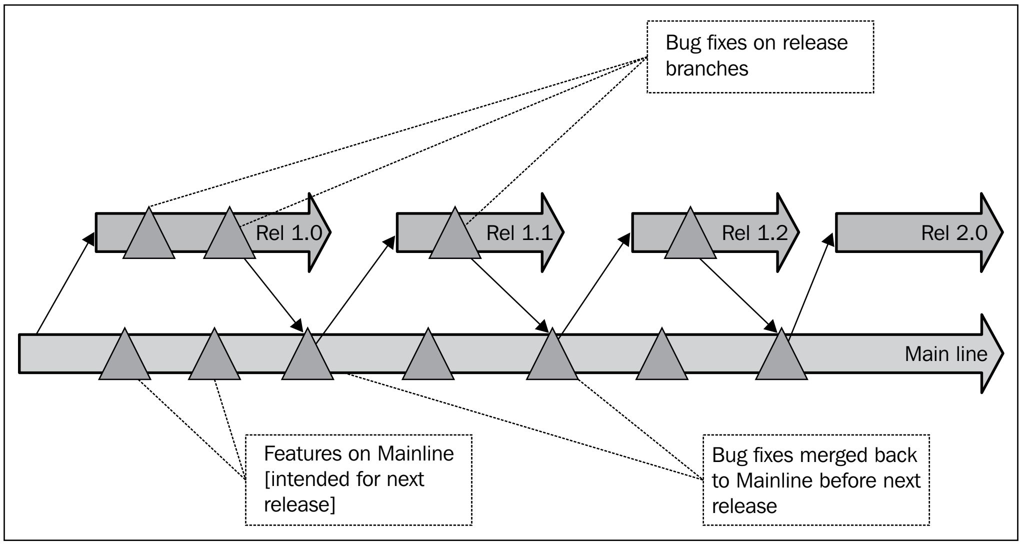
- What is the branch used for? The answer should be a short declarative sentence. Conditionals in the sentence indicate a need for multiple branches. Paragraph size answers indicate not enough uniformity for a stable branch.
- Who authorizes updates of the branch from other branches? Someone needs to control when things will change for any consumer of the branch contents. Benevolent dictators are much better than democracies!
- Who authorizes updates of other branches from this branch? True, anyone with read access to a branch can use it as a source. However, someone needs to be able to confirm and assure that the branch meets its conditions.

- What are the submit requirements? Test, build, task association, comment content, and others are all possible requirements. These requirements implicitly and explicitly define the stability and quality of branch contents. Automation to validate that a requirement is met can be useful. Automation to create information to satisfy a requirement is typically counterproductive.
- Who fixes problems after a submit? A question that doesn't always have an obvious answer. No one is probably a better choice than a group or individual that wasn't involved in the activities that created the content being submitted.

If your branching patterns don't have answers for these questions, or the questions have the wrong sort of answer, then you will have problems.

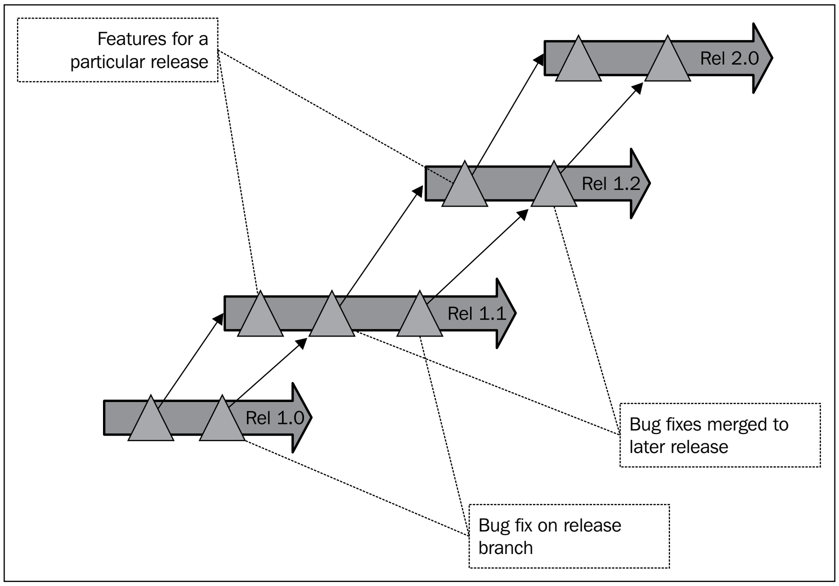
The mainline pattern – why it is better than alternatives

A mainline branch is usually a branch which contains all long-lived changes to a project. The quality of the branch is usually consistent with becoming a release branch, but it is recognised that bug fixes can be made on those release branches. Such bug fixes are usually propagated back to the mainline so that they are present when future release branches are made, as we see in the following diagram:


Bug fixes on release branches
Rel 1.0
Rel 1.1
Rel 1.2
Rel 2.0
Main line
Features on Mainline [intended for next release]
Bug fixes merged back to Mainline before next release

There are multiple release branches made from the same mainline.

An alternative, when there are multiple release lines which are prepared one after the other, is to have those release lines cascade from each other, as we see in the following screenshot:


Features for a particular release
Rel 2.0
Rel 1.2
Rel 1.1
Rel 1.0
Bug fixes merged to later release
Bug fix on release branch

Using this example, over time we will have many release branches, and all of the developers will need to keep switching their workspaces between the release branches as they become of release quality.

Industry best practice is the use of a single mainline as it is easier to understand and support, and less error prone.

We will discuss this in more detail in *Chapter 9*, *Perforce Streams* as it is one of the main branching patterns used in streams.

Release branches

The description of a release branch (from a mainline) is usually something like: release candidate. The submit requirements often include extensive testing requirements. Release branches have the mainline as their parent. If change is allowed on a release branch then only those changes required to meet release requirements, such as bug fixes, are allowed.

Perforce branching is efficient. Each release candidate is often a new and unique branch. As separate branches, the Perforce tools provide you with extensive and efficient methods for determining change.

The integration pattern

The description of an integration branch is usually something like: combine features before merge with primary development branch. The submit requirements need to adapt to the needs of the developers accessing the branch.

Integration branches often originate from developer branches, yet propagate into a development branch. They take full advantage of Perforce's ability to merge between branches that do not have a direct parent to child relationship.

OS copy is not a branching activity

Some people don't like to branch files between code lines that aren't siblings or direct descendants. So what do they do? They use the native workstation OS to copy the file into their current workspace. We encounter this often enough to know it's a common practice. However, it's a practice that sets the stage for many future problems.

The fundamental issue with an OS copy is that it breaks any connection between the files. Without that connection you won't be able to easily track the evolution of the files. In particular, if you find a bug, it will be hard to find all of the files that might contain that bug. Likewise, breaking that connection is a problem for edits. Without a connection, there can't be a 3-way merge.

The counter to this that we often hear is that they'll track the relationship in the submit comments. If you neglect the loss of tool features you might be able to make this work. However, think about how many bad submit comments you've seen. How often is information missing? Now ask yourself if you can really depend on those submit comments.

Summary

In this chapter, we've provided an extensive introduction to Perforce's classic branching and merging features. We've explored creating branches and propagating changes (merging) between those branches. P4V's revision graph tool, and its ability to provide you with extensive branching related data mining, was also introduced. Finally, we looked at some of the branching patterns that support the mainline model.

In the next chapter, we'll look at how streams build on the branching operations we have discussed in this chapter. Streams provide many behaviors that make implementation and support of the mainline model significantly more efficient.

9
Perforce Streams

In this chapter, we cover Perforce streams. Out of the box, streams support a set of predefined branching models. These are based on the metaphor of a stream where change flows in specific ways.

In this chapter, we'll cover the fundamentals of using streams. We'll see that under the covers, streams are a specific way of using the mainline model to support parallel development. Additionally, we'll look at the concept of branch stability and how streams use the merge-down, copy-up paradigm to support stability.

In this chapter we will cover:

- Basic stream patterns
- The mainline model
- Branch stability
- Creating streams
- The merge-down, copy-up paradigm
- More advanced options

Understanding streams

Unlike the wide-open, user-managed situation with classic Perforce branching, streams present the user with a predefined set of branching models. This may make streams seem limited, but there are many advantages. Streams instantiate the mainline model flow-of-change, which is an industry best practice. Moreover, streams can automate most of the workspace management and propagation tasks users would normally need to do for themselves. This makes streams much easier to manage, and helps an organization keep their branching policies and processes consistent.

> This chapter builds on concepts from other chapters, so there aren't many step-by-step examples. If you're doing follow along, feel free to start up P4V at any point and explore along with the topic being presented.

The primary stream models

There are three primary stream models:

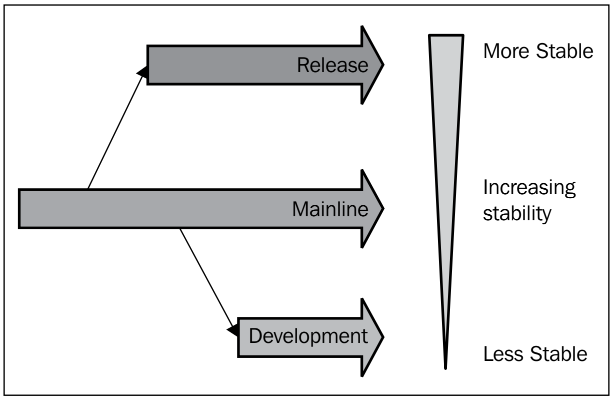
- Mainline
- Development
- Release

The flow of change between the various stream models is defined at the point each stream is created. Users must still validate the policy factors, such as the testing requirements, before or after propagating changes between streams.

Readers will recognize that the names of the stream models align with branching patterns discussed in *Chapter 8*, *Classic Branching and Merging*. More precisely, those names align with what is known as the **mainline** model.

Branch stability

The different types of branches within the mainline model are a response to different policies as to the desirability of change on those branches. This is also described as stability. The more stable the contents of that branch should be, the less desirable it is to make changes in a branch. We see that in this diagram:



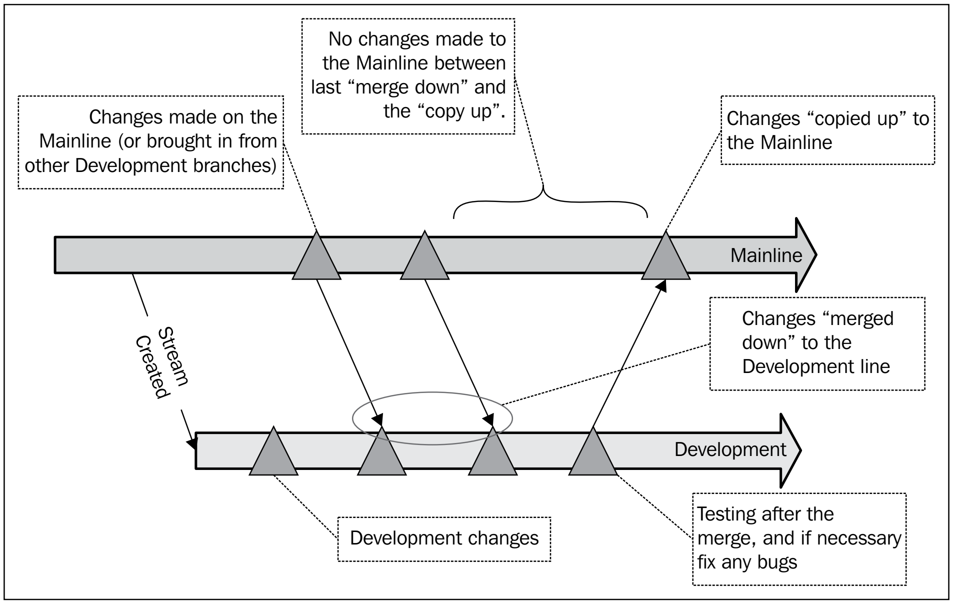
In an ideal world, release branches are very stable with almost no changes occurring. This is possible because changes are developed for a future release on the mainline or development branches, and then a new stable release branch is created. In practice, release branches usually have bug fixes made on them, so the quality of the release has an impact on its stability.

The merge-down, copy-up paradigm

The merge-down, copy-up paradigm is part of the way that streams maintain branch stability. The idea of the paradigm is that you merge-down from a more stable branch into a less stable branch and update a more stable branch by copying from the less stable branch.

> All of the diagrams we include in this book order streams on the page from top-to-bottom, with the most stable streams at the top. This simple convention helps convey the merge-down, copy-up paradigm (as opposed to merge-up, copy-down, or any other way of describing it). It is also how P4V shows streams on the **Streams** tab.

Let's relate this to the conflict and branching work we've seen in previous chapters. Merge implies that there may be conflicts that need to be resolved. On the other hand, copy implies that there are no conflicts and that the source files are copied character-for-character to replace the target files. This diagram shows the merge-down, copy-up paradigm in action:


No changes made to the Mainline between last "merge down" and the "copy up".
Changes made on the Mainline (or brought in from other Development branches)
Changes "copied up" to the Mainline
Mainline
Stream Created
Changes "merged down" to the Development line
Development
Development changes
Testing after the merge, and if necessary fix any bugs

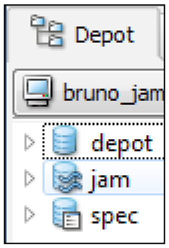
Changes are merged into the **Development** stream to isolate the impact of those changes. Because we have merged all of the mainline changes into our development stream, the resulting files can be copied back to the mainline. This is because after the merge, the development stream files contain both the development changes and the mainline changes.

The main reason for merge-down, copy-up is to perform the merges (which are risky) on the least stable branch. If there is a problem with the merge, it can be fixed if necessary via a subsequent change list on the less stable branch, before being copied up to the more stable branch.

This is not to say that merge-down, copy-up is effortless. You still need to coordinate and resolve merges. You must also ensure that your stream is up-to-date relative to a parent before you can copy its contents into that parent. P4V will warn you if this is not the case.

Creating a stream

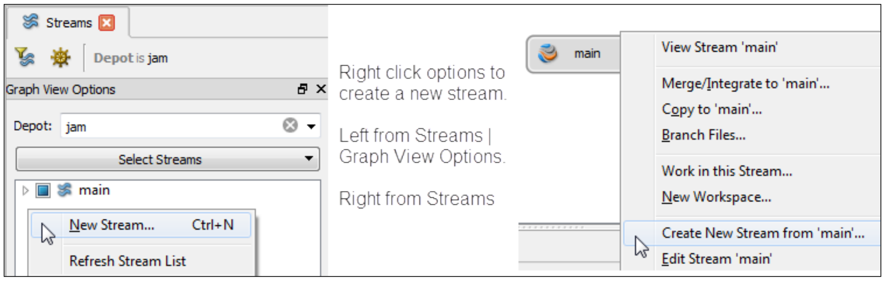
Streams require a stream depot. Stream depots can be identified in the tree panel depot view by their unique icon, as we see in the following screenshot:



In the preceding example, we see a stream depot called **jam**. Note that the icon is different from the classic depot called **depot**, and the spec depot called **spec**.

The first stream in a stream depot must be a **Mainline** stream. This is likely to have been created by your administrator, and populated with the appropriate files. These files are typically branched from elsewhere in the Perforce repository using the techniques discussed in *Chapter 8*, *Classic Branching and Merging*. We will also cover this later in the chapter in the section *Creating a Mainline from a classic branch*.

We create a development stream from the **Streams** tab. There are two options, as shown in the following screenshot:

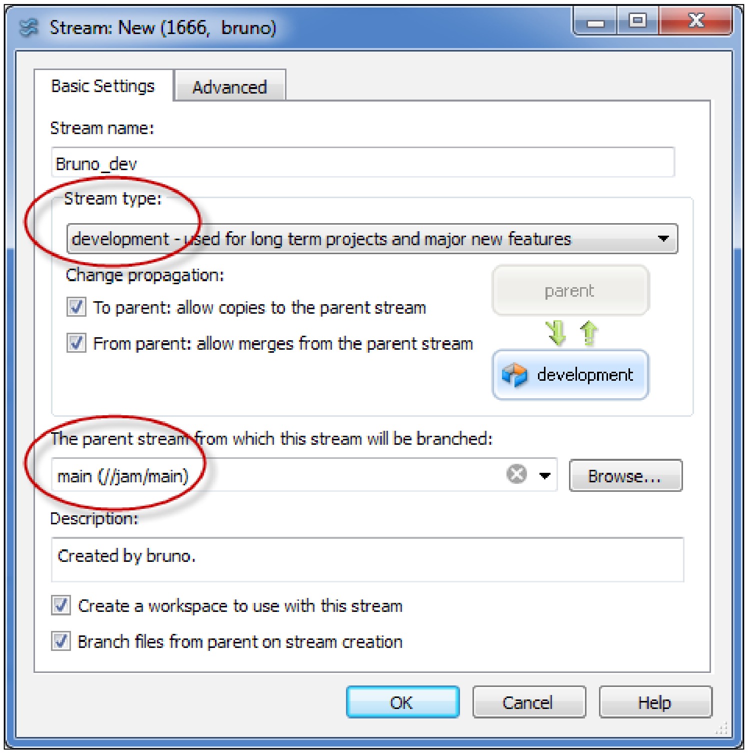


Right click options to create a new stream.

Left from Streams | Graph View Options.

Right from Streams

In the left-hand example, we see the **New Stream…** option which is generic within the streams depot. In the right-hand example, we are selecting **Create New Stream from 'main'…**. In both cases we will be presented with the following dialog:



Stream: New (1666, bruno)

Stream name: Bruno_dev

Stream type: development - used for long term projects and major new features

Change propagation:

☑ To parent: allow copies to the parent stream

☑ From parent: allow merges from the parent stream

The parent stream from which this stream will be branched: main (//jam/main)

Description: Created by bruno.

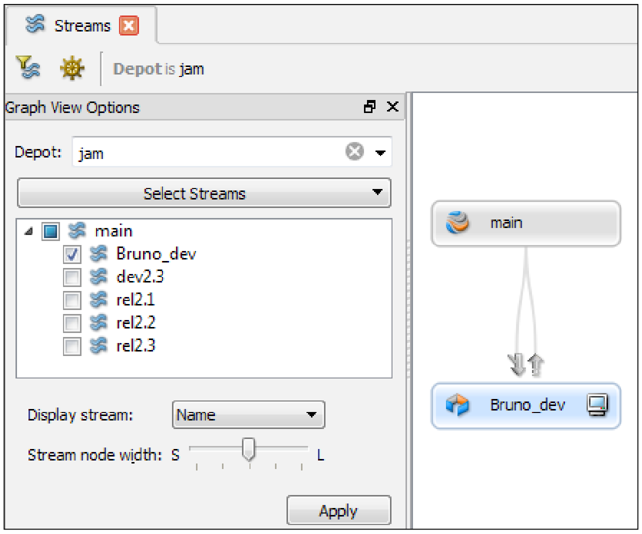
☑ Create a workspace to use with this stream

☑ Branch files from parent on stream creation

We are creating a **development** stream, which requires a **parent** stream. The **OK** button will be disabled until you enter the name of your development stream and have specified the parent. If you get to this dialog from the **New Stream…** option (not by right-clicking `main`), then you will need to select the **Browse…** option and click on `main`.

Particularly for a development stream, it is normal to check the option **Branch files from parent on stream creation**. It is possible to do this subsequently, as a separate step.

Clicking on **OK** creates our new stream and the results are shown graphically:



Depot is jam

Depot: jam

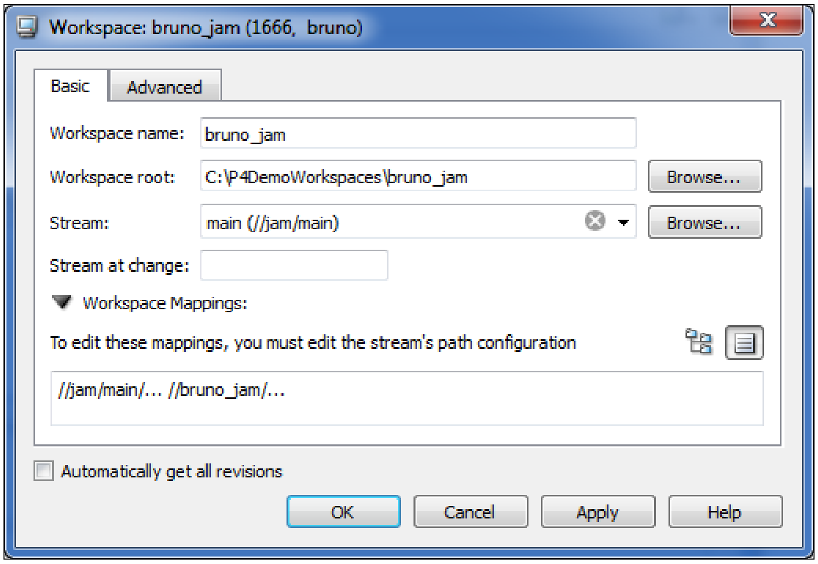
- ■ main
  - ☑ Bruno_dev
  - □ dev2.3
  - □ rel2.1
  - □ rel2.2
  - □ rel2.3

Display stream: Name

Notice the workspace icon next to the stream name in the preceding screenshot. Because we checked the box **Create a workspace to use with this stream**, P4V created this workspace based on the stream.

Stream workspaces

Creating a stream workspace with defaults will set its root and name automatically. Alternatively, select the stream by right-clicking on the file and selecting the **New Workspace…** option to allow you to customize these values, in a similar way as seen in *Chapter 6*, *Managing Workspaces*. This will give you the dialog box shown in the following screenshot:



Workspace: bruno_jam (1666, bruno)

Workspace name: bruno_jam

Workspace root: C:\P4DemoWorkspaces\bruno_jam

Stream: main (//jam/main)

Stream at change:

Workspace Mappings:

To edit these mappings, you must edit the stream's path configuration

//jam/main/... //bruno_jam/...

☐ Automatically get all revisions

This example shows a "follow along" workspace called **bruno_jam**.

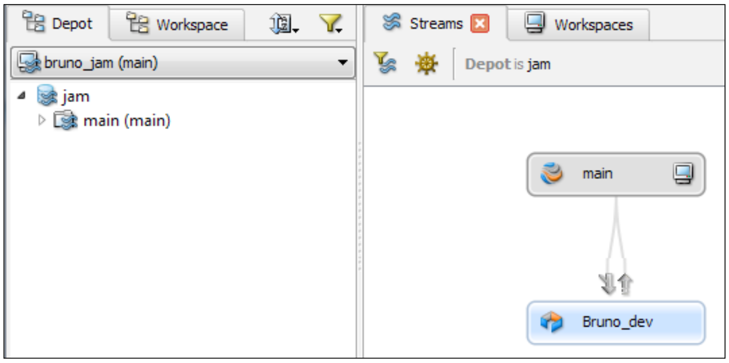
You can thus adjust the workspace name and also its root to the appropriate standards for your organization.

> We recommend creating a workspace name that does not include the stream name. As we will see shortly, workspaces are easy to swap between streams. A workspace name that contains a stream name may be confusing and lead to mistakes.

Moving your workspace between streams

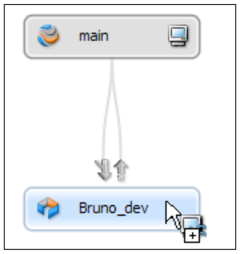
When we covered workspaces back in *Chapter 6, Managing Workspaces*, we recommended that each of those workspaces should have a separate directory structure in the local file system. As we've seen, each workspace that backs your streams can also have a unique local file system structure. However, because of the relationship between streams this unique file system structure recommendation can be relaxed. The ability of streams to share file system structures saves you set up overhead if you change context frequently.

The current stream your workspace is associated with is shown in a couple of ways.

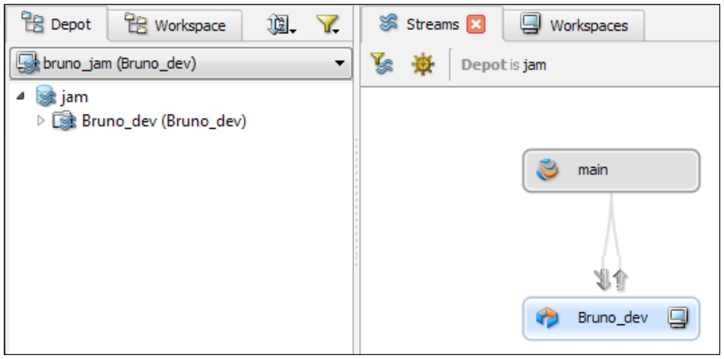


We can see the name of our workspace in the top-left of the tree pane, and the stream view in that pane. P4V also shows us the workspace icon on the **Bruno_dev** stream in the right-hand view.

Changing the workspace is relatively easy to do via drag-and-drop of the workspace icon:



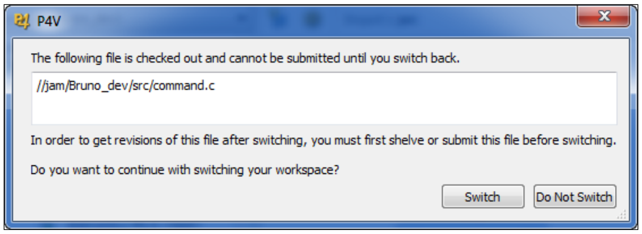
This shows the workspace icon being dropped on to the **Bruno_dev** stream, which will change the definition of the workspace:



Note the **bruno_jam (Bruno_dev)** in the top-left of the preceding screenshot.

Behind the scenes, P4V will change the definition of the workspace and also perform a Get Latest on the files. This is a fast and efficient operation, and works slickly.

You need to be a little careful about switching your workspace to a different stream when you have files checked out in a pending changelist. You don't need to worry too much, as in such a scenario you will receive a warning:



The following file is checked out and cannot be submitted until you switch back.

//jam/Bruno_dev/src/command.c

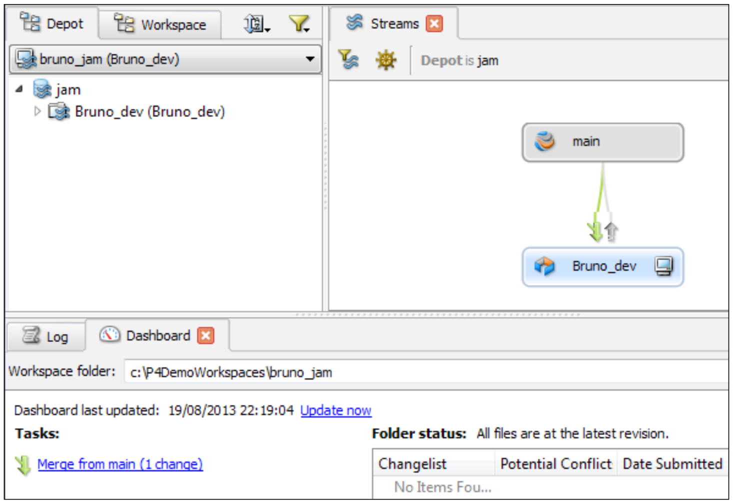
In order to get revisions of this file after switching, you must first shelve or submit this file before switching.

Do you want to continue with switching your workspace?

We recommend not switching in this case, and instead completing the work before switching. It is always possible to create a new workspace instead of switching if you really want to.

Communicating the status of changes to be propagated

One of the main advantages with streams is that P4V can easily communicate the changes that need to be propagated between the streams. The **Dashboard** tab is particularly valuable for this (**View** | **Dashboard** menu option makes it visible):



Workspace folder: c:\P4DemoWorkspaces\bruno_jam

Dashboard last updated: 19/08/2013 22:19:04 Update now

Tasks:

Merge from main (1 change)

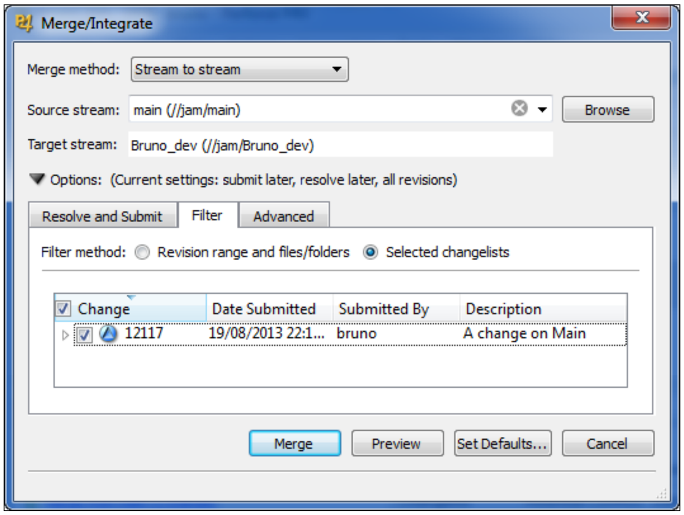
Folder status: All files are at the latest revision.

Changelist	Potential Conflict	Date Submitted
No Items Fou...		

In this example, it is explaining to us that there is a single change, which is a candidate to be merged from the Mainline under **Tasks**.

Merging changes from the Mainline

The easiest way to do this is to click on the **Merge from main (1 change)** link, to get:



Merge/Integrate

Merge method: Stream to stream

Source stream: main (//jam/main)

Target stream: Bruno_dev (//jam/Bruno_dev)

Options: (Current settings: submit later, resolve later, all revisions)

Filter method: ○ Revision range and files/folders ● Selected changelists

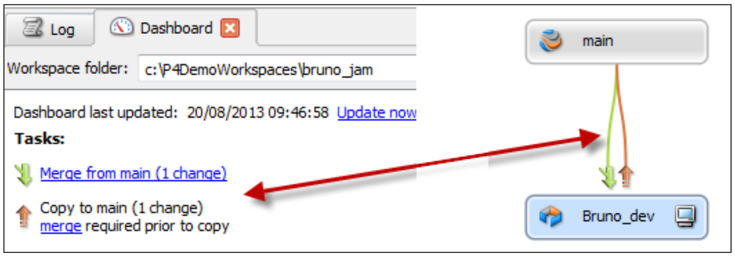
☑ Change	Date Submitted	Submitted By	Description
☑ 12117	19/08/2013 22:1...	bruno	A change on Main

In this example, we have chosen to set the **Filter** tab to view the change lists needing to be merged. It is often a useful confirmation of the changes involved.

The process of merging, including the dialog above, resolving conflicts, and submitting of the change is exactly the same as for classic branches, as discussed in *Chapter 8, Classic Branching and Merging*.

Copying our changes to the Mainline

If there are changes to be copied in both directions, then the icons (colored arrows) and **Dashboard Tasks** will be shown as:



Workspace folder: c:\P4DemoWorkspaces\bruno_jam

Dashboard last updated: 20/08/2013 09:46:58 Update now

Tasks:

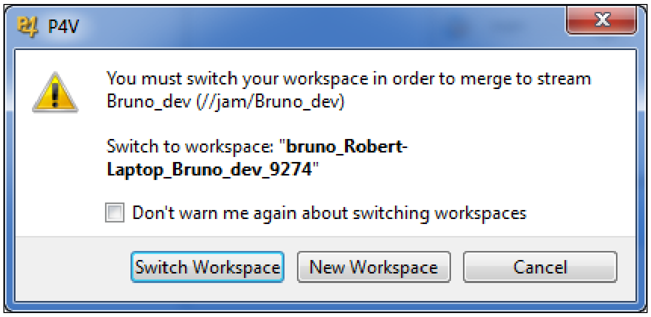
Merge from main (1 change)

Copy to main (1 change)
merge required prior to copy

The key thing here is to note that **merge is required prior to copy**: this is P4V enforcing the appropriate flow of change between the streams.

Performing the merge is the same as in the previous section.

You may receive a warning like this:



You must switch your workspace in order to merge to stream Bruno_dev (//jam/Bruno_dev)

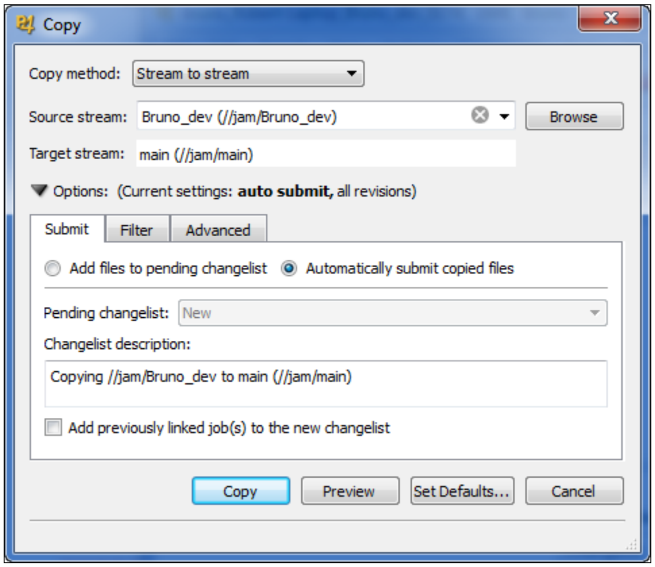
Switch to workspace: "**bruno_Robert-Laptop_Bruno_dev_9274**"

□ Don't warn me again about switching workspaces

The options here are either to switch to the other workspace, or to move your workspace to the new stream. We tend to move the workspace.

> You can check the option **Edit | Preferences | Streams | Use the same workspace and switch it between streams** so that you are not prompted to do this as the switch happens automatically.

The defaults for the **Copy** action are:



Copy method: Stream to stream

Source stream: Bruno_dev (//jam/Bruno_dev)

Target stream: main (//jam/main)

Options: (Current settings: **auto submit,** all revisions)

○ Add files to pending changelist ● Automatically submit copied files

Pending changelist: New

Changelist description:

Copying //jam/Bruno_dev to main (//jam/main)

□ Add previously linked job(s) to the new changelist

Note in this case the **auto submit** option is enabled, and the default change list description will be used.

Because this is a copy action, there is no need for resolving or merging. As we saw in the previous section, P4V will not allow a copy if there is merge required to the development stream.

Migrating from classic branches to streams

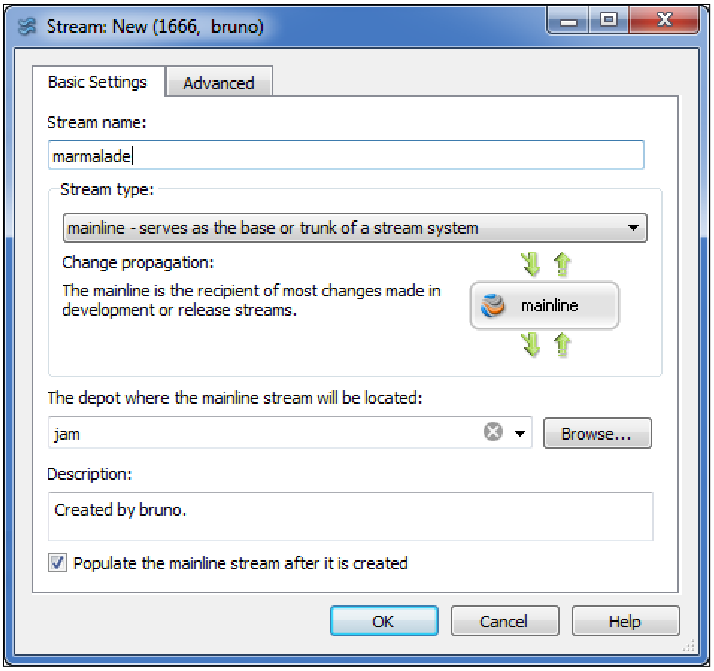
You may wish to migrate an existing classic branch or code line which already exists in your repository into a stream. In principle this is easy to do. The steps are:

1. Create a new Mainline stream by branching files from existing code lines.
2. Create new development and release streams from the new Mainline.

> Migrating a set of existing branches with pre-existing relationships is possible, but more complex and beyond the scope of this book. It requires careful planning and typically is the result of being conducted as a specific project. If you can, we recommend migrating project-by-project at an appropriate time in their lifecycle where you can start just by migrating a single Mainline.

Creating a new Mainline stream

First, we create our new mainline, via right-clicking on the file and selecting **New Stream...** to get:



Stream: New (1666, bruno)

Basic Settings | Advanced

Stream name: marmalade

Stream type: mainline - serves as the base or trunk of a stream system

Change propagation:

The mainline is the recipient of most changes made in development or release streams.

The depot where the mainline stream will be located: jam

Description: Created by bruno.

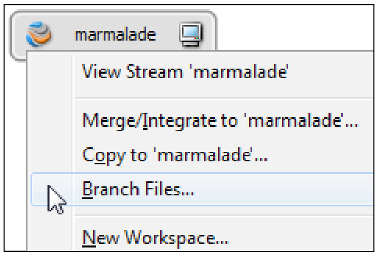
☑ Populate the mainline stream after it is created

In this case, we defined the **Stream type** as **mainline**, given the new stream name as **marmalade** and have checked **Populate the mainline stream after it is created**.

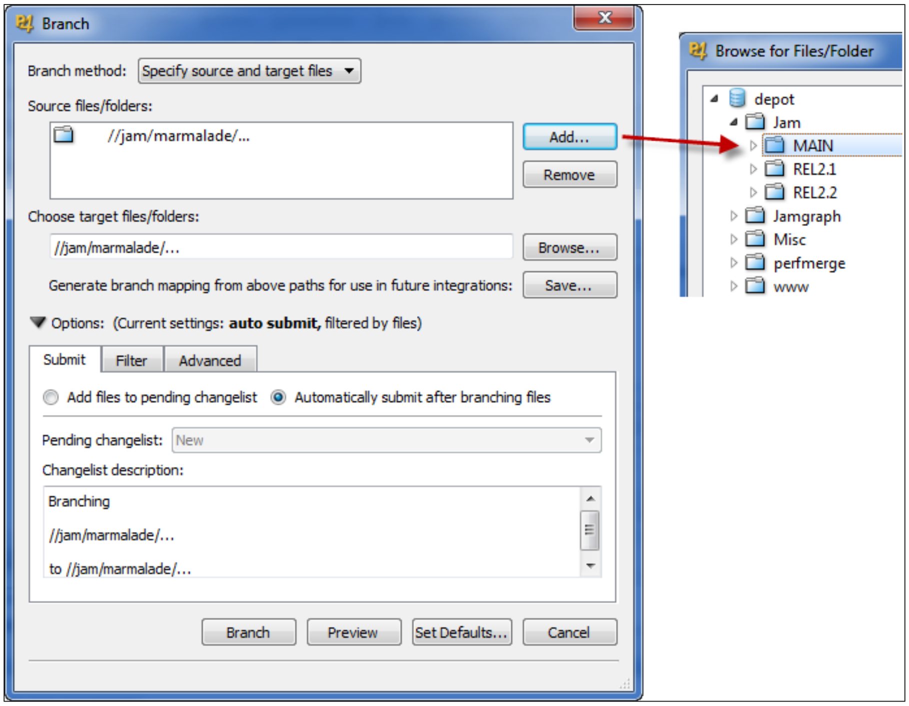
> You need to be sure that it makes sense to create a second Mainline in the same stream depot (**jam**). Consult with your administrator as to whether you should be doing this in a new depot or not. For follow-along activities it is fine to use the **jam** depot.

Populating our Mainline stream

Having created the marmalade stream, we can populate it using the **Branch Files...** right-click option, as shown in the following screenshot:



This will give us the following dialog (with possible subdialog):



Branch

Branch method: Specify source and target files

Source files/folders: //jam/marmalade/...

Choose target files/folders: //jam/marmalade/...

Generate branch mapping from above paths for use in future integrations:

Options: (Current settings: **auto submit,** filtered by files)

○ Add files to pending changelist ● Automatically submit after branching files

Pending changelist: New

Changelist description:

Branching

//jam/marmalade/...

to //jam/marmalade/...

We need to **Remove** the default **Source files/folders** and then click on **Add...** to select an appropriate new source, as shown in the preceding screenshot.

By clicking on **Branch** we will have created our new mainline stream. Note that the **Filter** and **Advanced** tabs can be used as discussed in *Chapter 8, Classic Branching and Merging*. This allows us to do things like branch from a label or other non-head revision of files, if we want to.

Other standard types of stream

So far we've looked at mainline and development streams. These are the streams that you're most likely to encounter during day-to-day activities. The streams interface has these additional standard types:

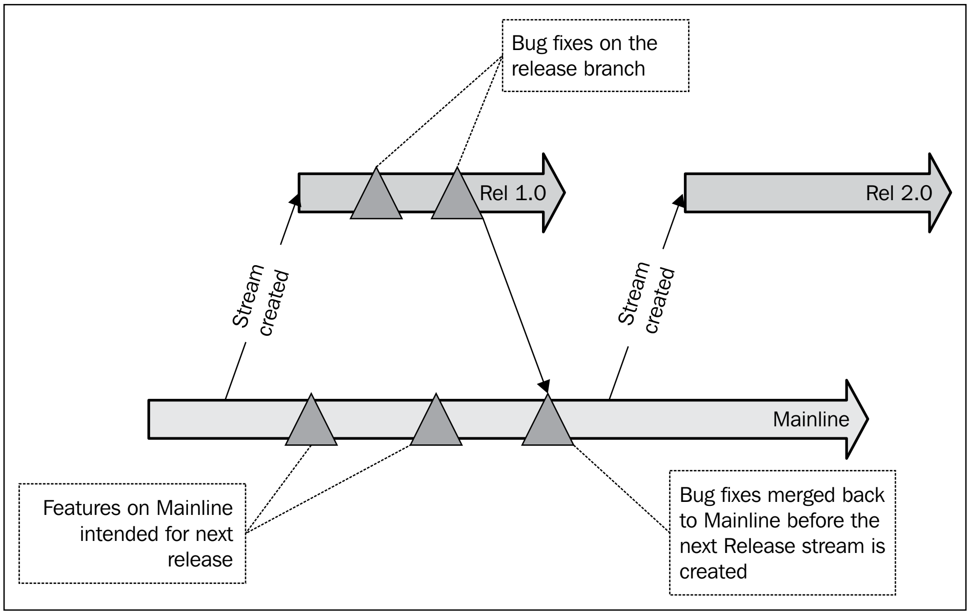
- Release streams: for managing releases
- Virtual streams: these partition larger streams for more efficient access
- Task streams: useful when only a relatively small number of files are likely to be changed

Release streams

Release streams are intended to support releases. Being closer to use by a consumer, release streams are designed to be more stable than mainline streams.

Typically, changes made on a release stream are intended to stabilize or improve quality. These fixes are usually merged back to the mainline so that when the next release stream is created from the mainline, it will already incorporate the bug fixes.

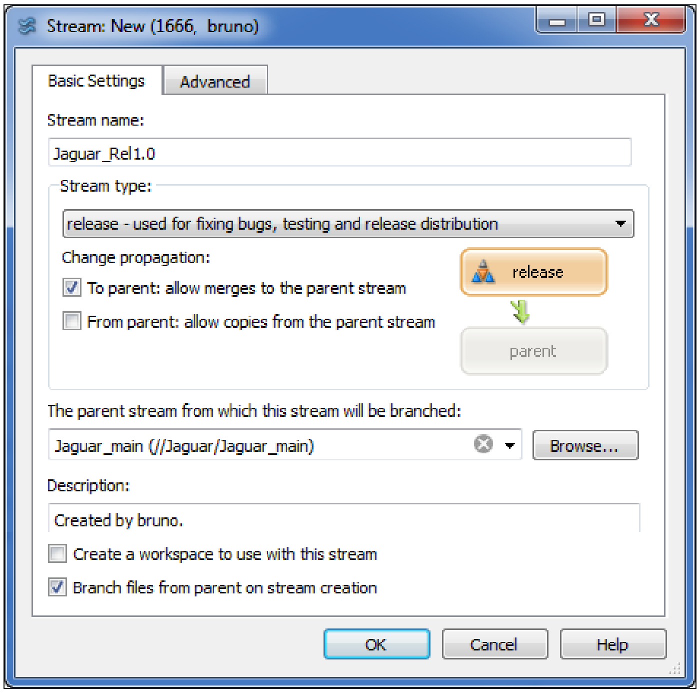
This diagram shows the typical relationship between mainline and release streams.


Bug fixes on the release branch
Rel 1.0
Rel 2.0
Stream created
Stream created
Mainline
Features on Mainline intended for next release
Bug fixes merged back to Mainline before the next Release stream is created

The features for `Rel 2.0` are made on the mainline ready for the release stream to be created. The reason for not making bug fixes on the mainline and propagating them to the `Rel 1.0`, is that such propagation risks taking with it the new features. Although this can be done via selective propagation, it is more risky and likely to destabilize the release. Therefore, it is best to do it as shown in the preceding diagram.

Change propagation for Release streams

To support the model we have just described, when you create a release stream the **Change propagation** option is shown as follows:



Stream name:
Jaguar_Rel1.0

Stream type:
release - used for fixing bugs, testing and release distribution

Change propagation:
☑ To parent: allow merges to the parent stream
☐ From parent: allow copies from the parent stream

The parent stream from which this stream will be branched:
Jaguar_main (//Jaguar/Jaguar_main)

Description:
Created by bruno.

☐ Create a workspace to use with this stream
☑ Branch files from parent on stream creation

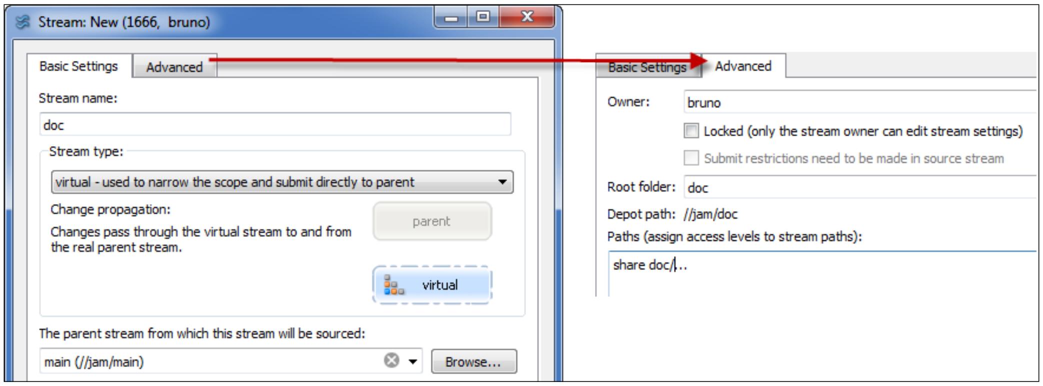
Thus we see that it supports propagating changes to the parent but not from the parent (the mainline in this instance).

Virtual streams

A virtual stream isn't actually a stream at all. Instead, it is simply a filtered view of its parent stream. This means that a virtual stream typically includes only a subset of what is in its parent. If you propagate changes to the virtual stream, you are in fact propagating them directly to the parent.

The power of virtual streams is that they make it easier for different teams or people to work only on a subset of all the files in the parent stream. It also means that you have fewer files to merge down before you copy your changes up to the mainline,which can be a very useful time saver. Virtual streams can also be used for re-parenting lots of child streams in one action. See the section *Re-parenting streams* later in this chapter.

In the following example, we are creating a virtual stream for only documents within its parent stream of main:



Stream: New (1666, bruno)

Basic Settings | Advanced

Stream name: doc

Stream type: virtual - used to narrow the scope and submit directly to parent

Change propagation: Changes pass through the virtual stream to and from the real parent stream.

The parent stream from which this stream will be sourced: main (//jam/main)

Basic Settings | Advanced

Owner: bruno

☐ Locked (only the stream owner can edit stream settings)

☐ Submit restrictions need to be made in source stream

Root folder: doc

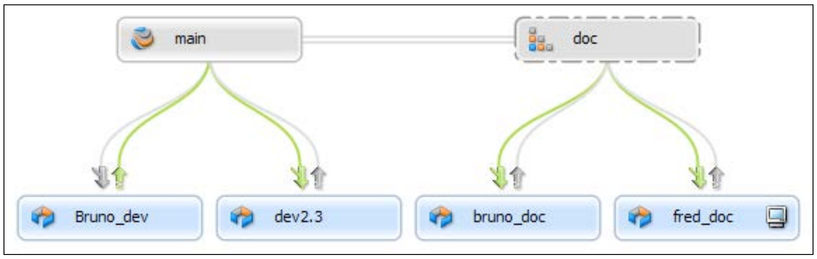
Depot path: //jam/doc

Paths (assign access levels to stream paths):

share doc/...

Note that on the **Advanced** tab we have specified in **Paths** to **share doc/...** rather than **share ...**. This means we'll only include the doc directory and its contents and not the rest of the stream.

Once created, the virtual stream appears on the streams graph, and can be used as the parent of other streams:



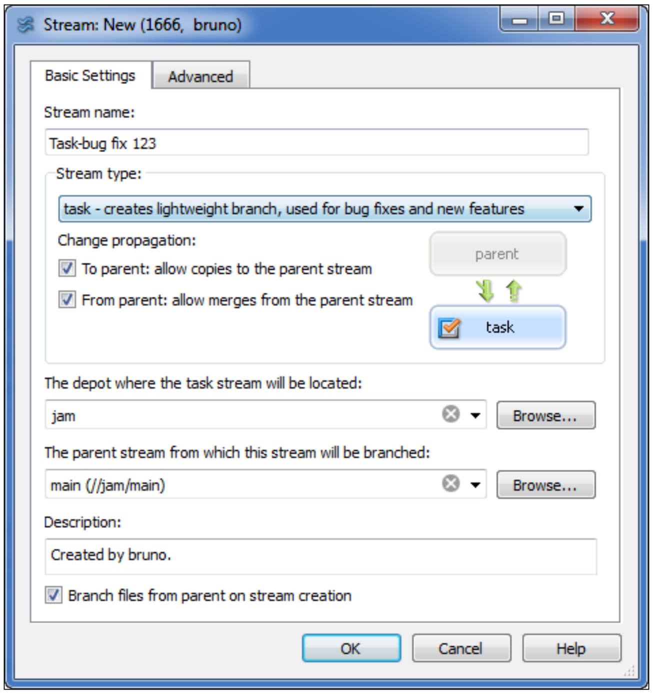
Here we see that bruno_doc and fred_doc are using the doc stream as a parent. So using a virtual stream is the same as using a real stream. However, if they propagate changes to doc, then they are actually propagating those changes to main.

Task streams

Task streams are a specialized version of development streams introduced with Perforce server release 2013.1. Task streams are intended to support a focused development activity that typically requires only a short time to complete. Supporting a focused development activity means that task streams cannot be the parent of another stream. However, you can convert a task stream into a standard development stream if you discover that the task has become a larger project.

A not so user-visible characteristic of task streams is that the server can process them more efficiently than other types of streams. This results in better performance for large installations. However, this processing efficiency advantage is lost when the number of files being modified in the task stream becomes large (typically thousands or greater).

Creating a task stream is just like creating other streams: right-click on the file and select **New Stream…**, or right-click on the file and select **Create New Stream from …**. You then need to specify that the stream is of type task, as seen here:



Stream: New (1666, bruno)

Stream name: Task-bug fix 123

Stream type: task - creates lightweight branch, used for bug fixes and new features

Change propagation:

☑ To parent: allow copies to the parent stream

☑ From parent: allow merges from the parent stream

The depot where the task stream will be located: jam

The parent stream from which this stream will be branched: main (//jam/main)

Description: Created by bruno.

☑ Branch files from parent on stream creation

Working with a task stream is just like working with a development stream. However, task streams are intended to be short-lived, and once you have used your task stream via merge-down, copy-up, it is good to tidy things up and delete or unload the stream.

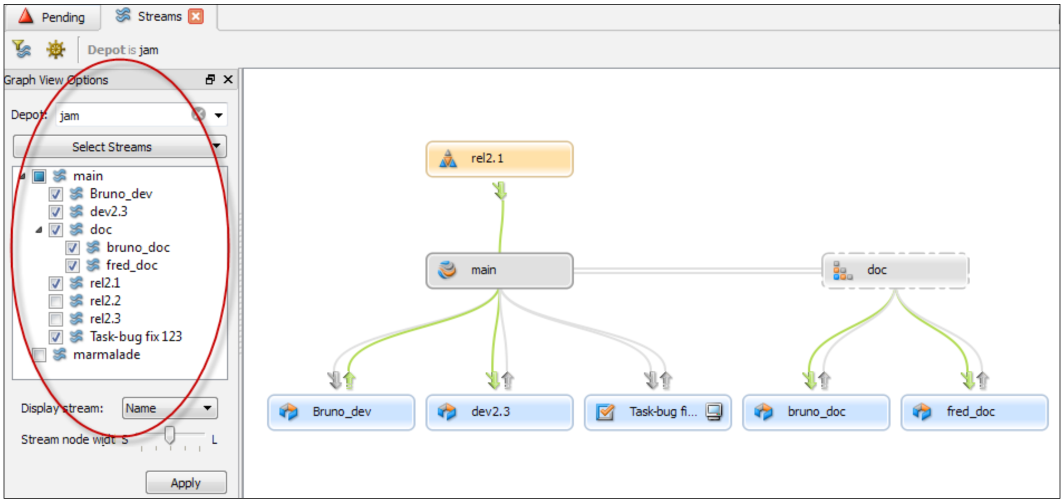
> The unload option is typically used by administrators of larger Perforce repositories to avoid visual clutter in P4V.

Managing streams

In this section, we'll look at various techniques for managing streams. In general, these techniques are similar to other parts of the P4V interface. However, the presentation is unique to the needs of the streams interface.

Applying the stream filter

It is not uncommon to find a hundred or more streams in a depot. Even if there aren't a large number of streams in a depot it is often useful to focus on just the streams that are relevant to your work. Creating that focus is easy with the filter option show as follows:



Depot is jam

Graph View Options

Depot: jam

Select Streams

- ■ main
  - ☑ Bruno_dev
  - ☑ dev2.3
  - ☑ doc
    - ☑ bruno_doc
    - ☑ fred_doc
  - ☑ rel2.1
  - □ rel2.2
  - □ rel2.3
  - ☑ Task-bug fix 123
- □ marmalade

Display stream: Name

Stream node widt S L

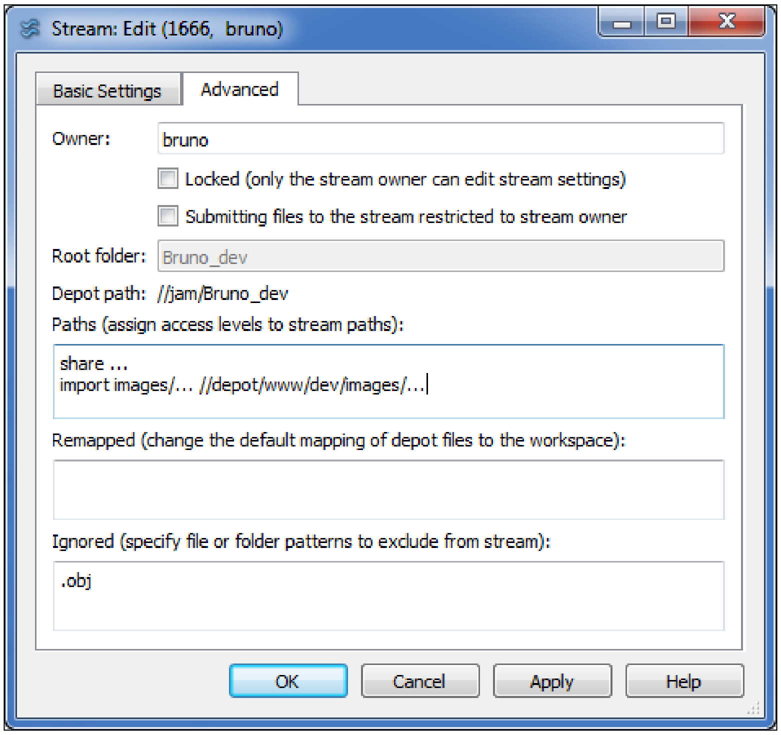
Apply

Check the various streams to see the ones you wish to on the left-hand side. Explore the **Select Streams** dropdown options to check different groups of streams.

Mapping stream files

The **Advanced** tab for a stream provides options that control how files are mapped in a stream. This is similar to the view specification in a classic workspace. However, the files in a stream must relate to their parent. Thus, the specifications have an implicit reference to the files of the parent stream. Even though the format may be different, the concepts are similar to those described in *Specifying a workspace* in *Chapter 6*, *Managing Workspaces*.

Like the view specification of the classic workspace, a stream's advanced specifications also define the structure of any workspaces associated with the stream as we can see in the following screenshot:



Owner: bruno

□ Locked (only the stream owner can edit stream settings)

□ Submitting files to the stream restricted to stream owner

Root folder: Bruno_dev

Depot path: //jam/Bruno_dev

Paths (assign access levels to stream paths):

```
share ...
import images/... //depot/www/dev/images/...
```

Remapped (change the default mapping of depot files to the workspace):

Ignored (specify file or folder patterns to exclude from stream):

```
.obj
```

This section is not intended to provide an exhaustive coverage of stream workspace mappings. Rather, it is intended to provide users with the knowledge required to read and understand such a mapping.

In the preceding example, the **Paths** section is importing a specific directory (`//depot/www/dev/images/...`) into the local directory `images/...` (which is directly under the stream root). All files in this directory and below will be treated as read-only and can not be modified in the stream. The various dialog options are further explained in the following sections in this chapter.

> The **Help** is always available to you, and in particular you can click on the **stream view** link to get an explanation with various examples.

Consult your local Perforce administrator for guidance specific to your organization.

Paths/access levels

Paths define how you may access stream files in your workspace and how they propagate to and from this stream. The following path types are available:

- **share**: Changes to files in shared paths will flow to and from other streams. For example, if you modify files in this stream you can merge them down or copy them up to other streams as appropriate. This is the default.
- **isolate**: Changes to files in isolated paths do not propagate to other streams. Isolated paths are useful for storing build and other generated files. So you can store built binaries on a release branch, but isolate them from the mainline parent which handles only source code.
- **import**: Imported files populate a workspace but can not be modified and do not propagate to other streams. Imported paths are useful for components such as third-party libraries or other shared resources that are not to be modified by a project or stream.
- **exclude**: Prevents files that would be inherited from the parent files from becoming part of the stream or populating the stream workspace. This allows the creation of more focused child streams.

> It is important to remember that streams inherit the mappings of their parent. If you exclude files they are not available to children and cannot be restored (for example, by grandchildren).

Remapping of files

Files can be remapped, which enables you to have them synced to a different workspace location than their corresponding relative location in the repository. For example, the following view specifies that the parent's `doc` files will be mapped beneath the `doctools` path in workspaces which use this stream:

```
doc/... doctools/doc/...
```

Ignoring files

You can specify file types to be ignored, which is useful for ensuring that artifacts such as object files or other interim build files are never checked in or propagated to other streams. These types are excluded by the workspace view of workspaces associated with the stream. For example:

```
.obj
/tmp/...
```

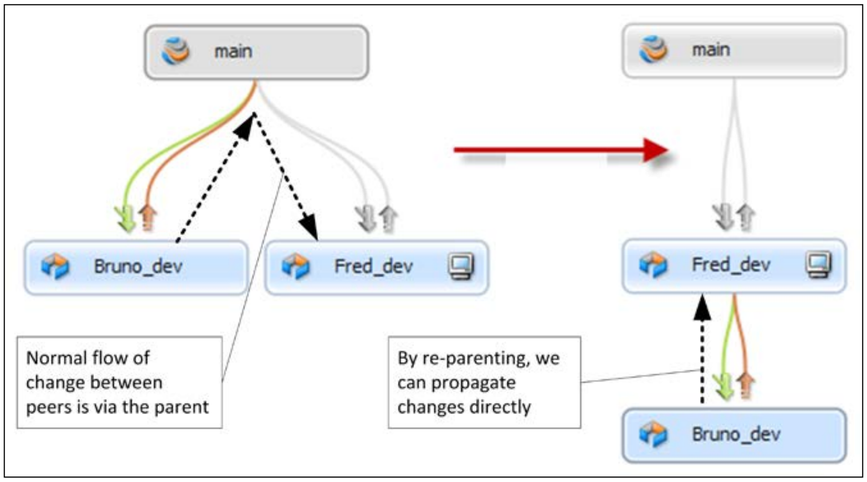
Re-parenting streams

While this is relatively easy to do, we need to consider the situations when it is a good idea. The two main scenarios are:

- Propagating changes across the stream hierarchy
- Moving development streams from one parent to another in more complex stream hierarchies

Propagating changes across the stream hierarchy

Propagating changes across the hierarchy allows you to perform selective propagation of changes. Normally, changes only flow between parent-child streams. If you have two child streams of the same parent then they normally exchange changes via the parent:

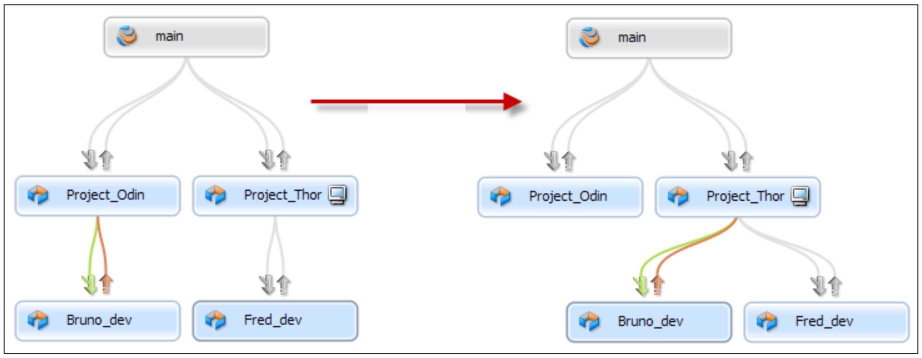

main
Bruno_dev
Fred_dev
Normal flow of change between peers is via the parent
main
Fred_dev
Bruno_dev
By re-parenting, we can propagate changes directly

You might not want to exchange changes via the parent as it is potentially making development changes prematurely visible on the mainline. Therefore, you can re-parent one child to the other and then perform the selective migration directly between them.

> While this is not particularly difficult to do, it should be the exception rather than the norm in your development processes. Selective propagations are more risky than straight forward propagations.

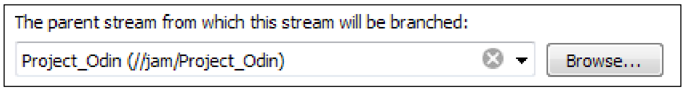
Moving groups of development streams

It is quite common to have multiple development streams grouped under projects. Given that project priorities sometimes change, you may want to move a development stream from one project to another:



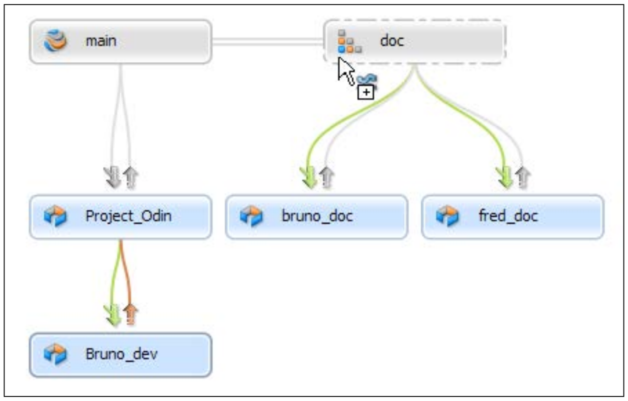
Directly editing versus drag-and-drop

There are two main ways of re-parenting a stream. You can edit the stream (**right-click | Edit Stream 'Bruno_dev'**) and change its parent easily, typically using the, **Browse...** button:


The parent stream from which this stream will be branched:

Project_Odin (//jam/Project_Odin)

In a similar way to using drag-and-drop to move your workspace to a different stream, you can re-parent a stream by dragging and dropping it on a new parent, as shown in the following screenshot:

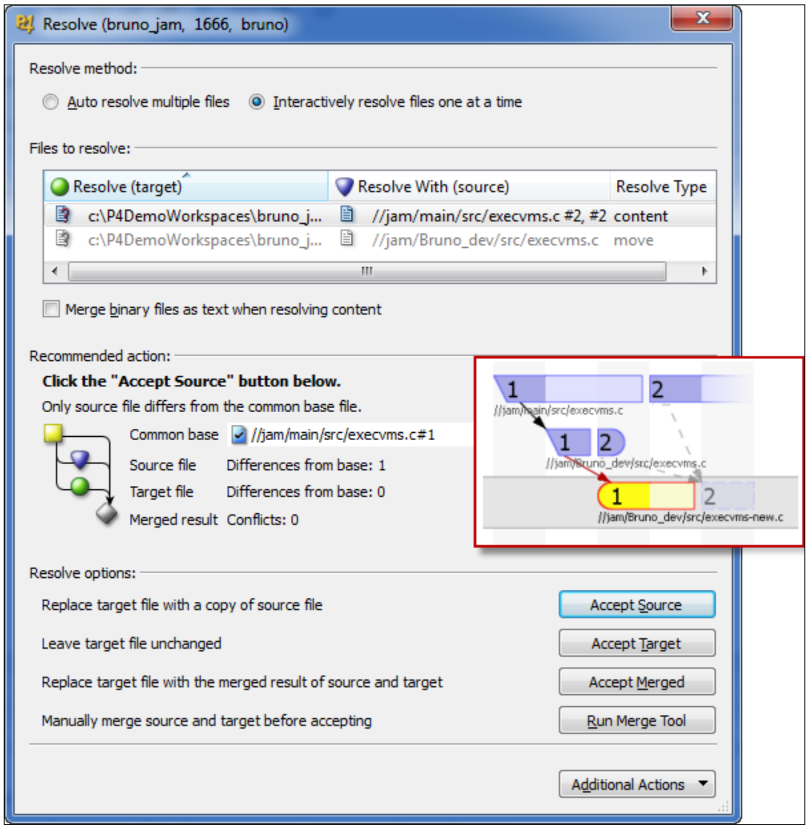


This shows Bruno_dev about to be dropped on to the doc virtual stream. One advantage of virtual streams is that they make it easy for all of their child streams to be re-parented just by re-parenting the virtual stream itself.

> You need to be careful with drag-and-drop and make sure you drop on the correct parent. It is all too easy to drop on the wrong target! People also accidentally re-parent streams when they just intended to move their workspace between streams.

How streams handle renames

We saw in *Chapter 8*, *Classic Branching and Merging* that handling renames of files across different branches can be problematic. With streams, these problem situations are detected and handled straight forwardly. For example, renaming a file on one stream and then finding that the equivalent file has been edited on the other stream will show up in your resolve like this:



Resolve (bruno_jam, 1666, bruno)

Resolve method:

○ Auto resolve multiple files ● Interactively resolve files one at a time

Files to resolve:

Resolve (target)	Resolve With (source)	Resolve Type
c:\P4DemoWorkspaces\bruno_j...	//jam/main/src/execvms.c #2, #2	content
c:\P4DemoWorkspaces\bruno_j...	//jam/Bruno_dev/src/execvms.c	move

□ Merge binary files as text when resolving content

Recommended action:

Click the "Accept Source" button below.

Only source file differs from the common base file.

Common base	//jam/main/src/execvms.c#1
Source file	Differences from base: 1
Target file	Differences from base: 0
Merged result	Conflicts: 0

Resolve options:

Replace target file with a copy of source file

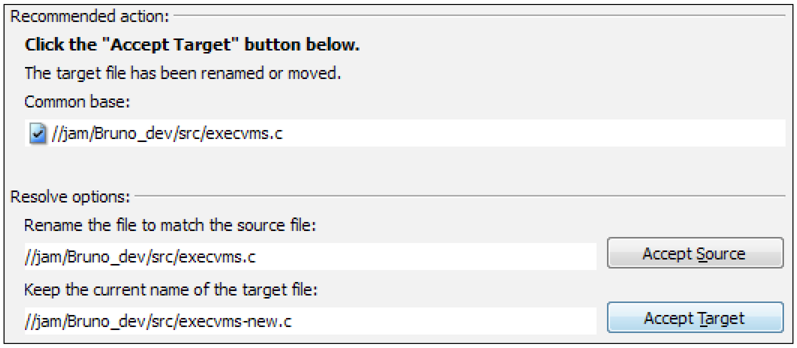
Leave target file unchanged

Replace target file with the merged result of source and target

Manually merge source and target before accepting

In the preceding screenshot, we see the combination of a resolve dialog and the revision graph for the files involved. The file `execvms.c` was edited on `main`, and has been renamed on the `Bruno_dev` stream to `execvms-new.c`.

The next screenshot clearly shows the options as to how to handle the renamed file:



Recommended action:

Click the "Accept Target" button below.

The target file has been renamed or moved.

Common base:

☑ //jam/Bruno_dev/src/execvms.c

Resolve options:

Rename the file to match the source file:

//jam/Bruno_dev/src/execvms.c

Keep the current name of the target file:

//jam/Bruno_dev/src/execvms-new.c

The recommended action will merge the source changes into the file with the new name, which is typically what we would want. We do have the choice not to do this if it is appropriate.

> The revision graph screenshot was obtained via the **Additional Actions** button in the bottom right of the resolve dialog. It is a very useful option at times to help understand the history of a file and decide what the appropriate action should be.

Summary

This chapter has covered the basics of using streams. We've seen how they provide an implementation of the mainline model. They also automate most of the activities required to manage branches and keep them up to date.

In the next chapter, we'll look at configuring P4V so that it better supports your work style.

10

The P4V User Experience

For many people the P4V user experience is fine out of the box. Also, there are many ways that you can adapt P4V to make your work easier and more productive.

In this chapter we will cover:

- Navigating large trees easily
- Integrating access to other tools
- Controlling P4V operations
- Reconciling offline work

> The actions described in this chapter are unique to your specific instance of P4V. Most have no server impact so the follow-along suggestions can be used even if you only have a connection to a production server.

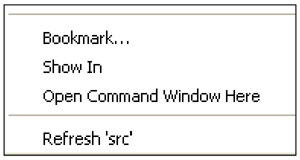
Navigating large trees of folders

Navigating a large project tree can be tedious. P4V has a bookmarks feature to help you deal with this situation.

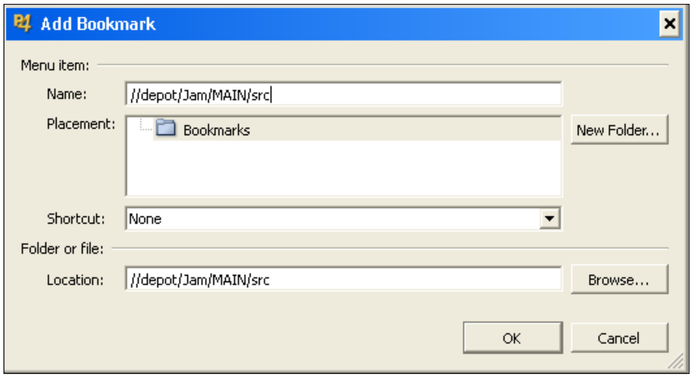
Bookmarks

Viewing a folder which is six levels deep in a tree requires at least six clicks, with each click resulting in P4V asking the P4D server for information for items at that level. This can be time consuming as well as tedious. P4V provides a bookmark interface that can quickly reposition either the depot tree or the workspace tree to locations that you define. Bookmarks are easy to declare, manage, and remove.

You can explicitly declare a bookmark using the **Tools** | **Bookmarks** menu interface. However, most people prefer to select a directory or file in the tree panel, right-click on the page, and then click on **Bookmark...** near the bottom of the menu, shown below:



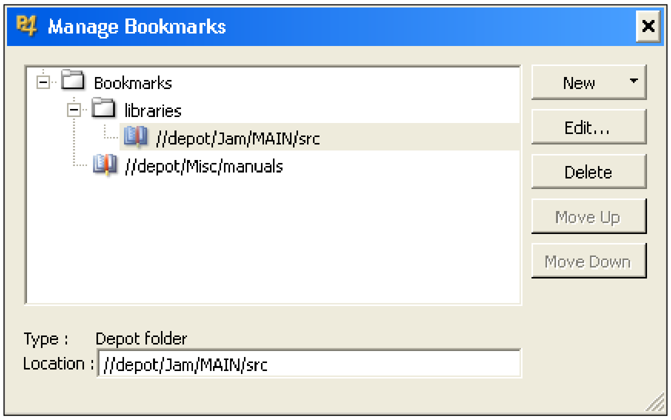
Either method brings you to the add bookmark dialog shown as follows where you can define the attributes of the bookmark.



Menu item:
Name: //depot/Jam/MAIN/src
Placement: Bookmarks
Shortcut: None
Folder or file:
Location: //depot/Jam/MAIN/src

To access a bookmark, you can use the optional shortcut key defined when you add the bookmark. You can also use the **Tools** | **Bookmarks** menu.

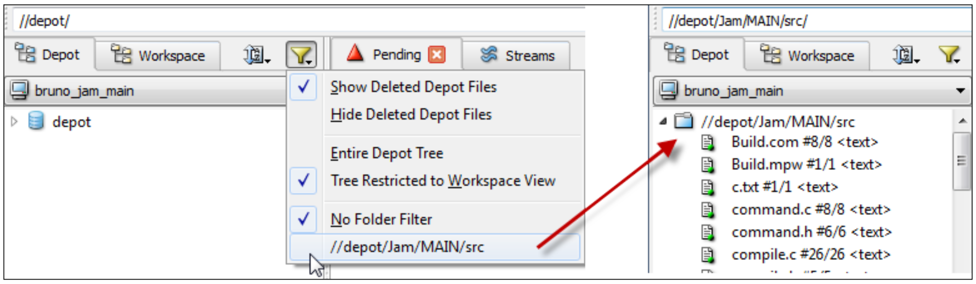
You use the **Tools** | **Bookmarks** | **Manage bookmarks...** menu to launch the bookmark management dialog, shown here:



Type : Depot folder
Location : //depot/Jam/MAIN/src

This allows you to create new bookmarks, edit them, or delete existing bookmarks.

Bookmarks can be used either to return to a folder or file, or to filter the current view to only a bookmarked folder, as shown in the following screenshot:



In the preceding screenshot we can see that the folder filter shows the bookmarks currently available. Selecting one of them will filter the current folder appropriately, which can be very convenient.

> To follow-along, set bookmarks in both the depot tree view and the workspace tree view. Collapse or expand related sections of the display. Select the various bookmarks to see how the display repositions and changes.

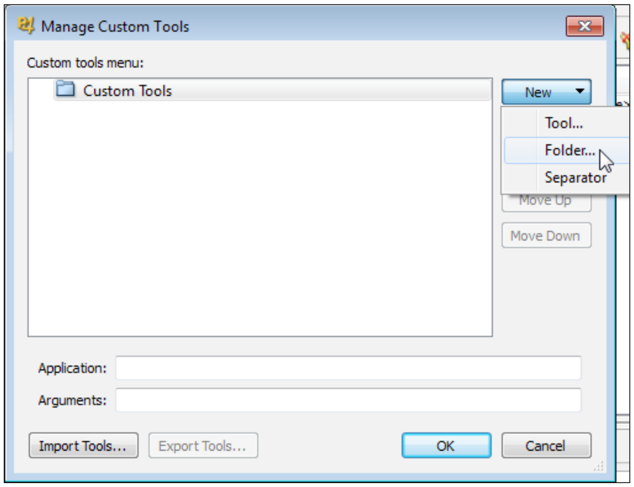
Custom P4V tools

P4V has a custom tool interface that allows you to integrate tools into the P4V menu system. Build, code review, formatting, and other process customization tools are commonly integrated into the P4V menu context using this technique.

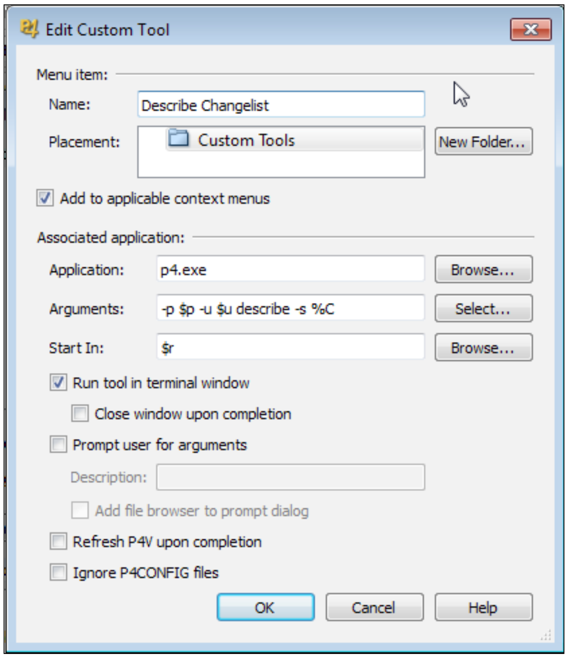
Custom tools have been present for many years in P4V, and in some cases, the reason for the use of a custom tool has been made redundant by new P4V functionality.

Custom tools can be as simple as a straight forward p4 command-line output (useful for copy and paste of the text), or they can execute a script which performs multiple process steps.

To add or manage custom P4V tools select the **Tools** | **Manage Custom Tools...** menu to bring up this dialog:



From this dialog you have complete control of the custom tool structure. You can even coordinate the import and export of tools between different hosts. To manage the individual tools, you select either the **New** or **Edit...** button as appropriate. This brings up the following dialog where you can specify the characteristics of the tool:



Menu item:

Name: Describe Changelist

Placement: Custom Tools

☑ Add to applicable context menus

Associated application:

Application: p4.exe

Arguments: -p $p -u $u describe -s %C

Start In: $r

☑ Run tool in terminal window

☐ Close window upon completion

☐ Prompt user for arguments

Description:

☐ Add file browser to prompt dialog

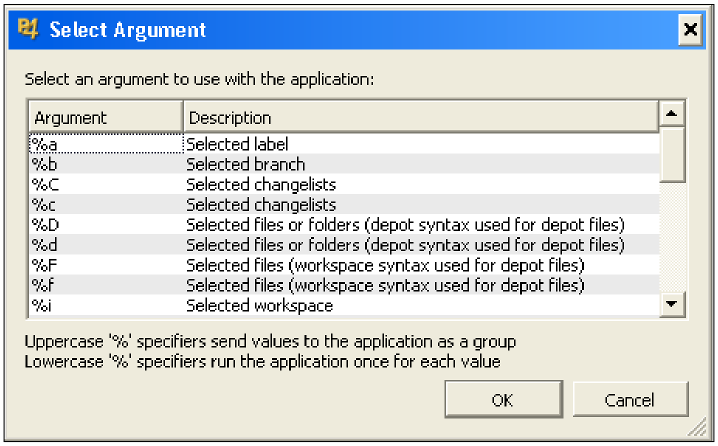
☐ Refresh P4V upon completion

☐ Ignore P4CONFIG files

In this example we are running a simple `p4 describe` (a Perforce command-line command as mentioned in *Appendix B, Command Line*) on a particular changelist. The inclusion of the following arguments makes sure that the command is executed for the correct server and as the correct user `-p $p -u $u`.

Note the check box labeled **Add to applicable context menus**. If selected, the tool will appear at the bottom of context menus where it applies.

One of the powers of the tool specification is the ability to specify arguments by type. The **Select...** button at the right side of the **Arguments:** prompt brings up a dialog that allows you to specify arguments. We see a sample of the argument choices here:



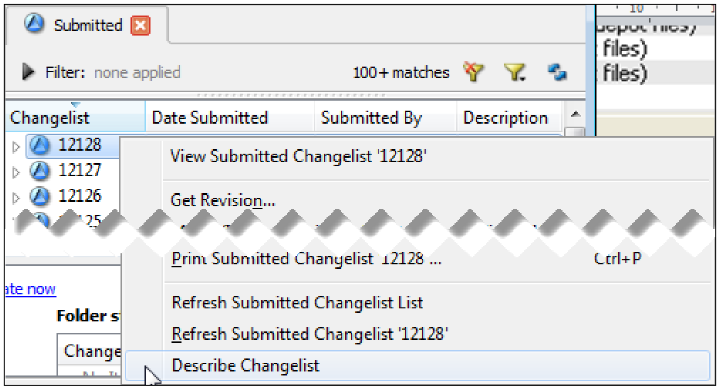
Select an argument to use with the application:

Argument	Description
%a	Selected label
%b	Selected branch
%C	Selected changelists
%c	Selected changelists
%D	Selected files or folders (depot syntax used for depot files)
%d	Selected files or folders (depot syntax used for depot files)
%F	Selected files (workspace syntax used for depot files)
%f	Selected files (workspace syntax used for depot files)
%i	Selected workspace

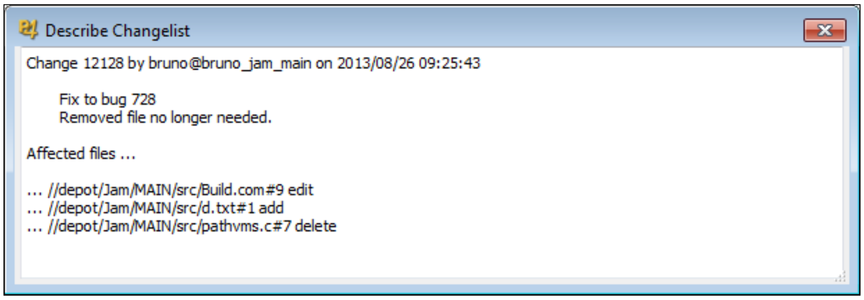
Uppercase '%' specifiers send values to the application as a group
Lowercase '%' specifiers run the application once for each value

Note that the character case of the argument specification can control how the tool is invoked.

For the previous example, when you right click on a changelist the context menu would be:



Thus we see that the description **Describe Changelist** appears on the menu and can be clicked. The result might be:



```
Change 12128 by bruno@bruno_jam_main on 2013/08/26 09:25:43

	Fix to bug 728
	Removed file no longer needed.

Affected files ...

... //depot/Jam/MAIN/src/Build.com#9 edit
... //depot/Jam/MAIN/src/d.txt#1 add
... //depot/Jam/MAIN/src/pathvms.c#7 delete
```

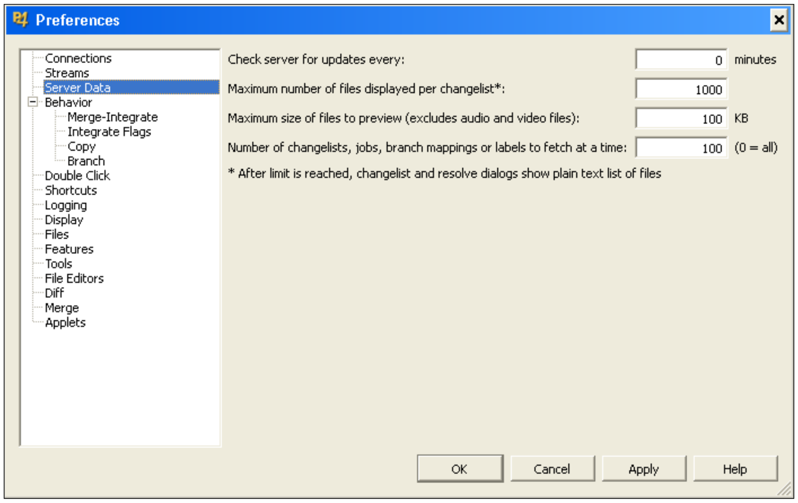
In this instance you can copy and paste the text (for example, into a defect tracking tool).

> Follow-along by experimenting with different specifications. The tools don't need to exist. However, you should assure benign results in case you select them.

P4V preferences

P4V provides users with access to preferences that control most of its default and general operational characteristics. You can control the characteristics of communications with the server to adjust the responsiveness of P4V. You can specify alternative edit, difference, and merge tools by file type. You can even restore dialogs that you have chosen not to be shown again.

Management of preferences is available through the **Edit** | **Preferences...** menu. Selecting this option brings up the **Preferences** dialog, as we see in the following screenshot:



Check server for updates every: 0 minutes

Maximum number of files displayed per changelist*: 1000

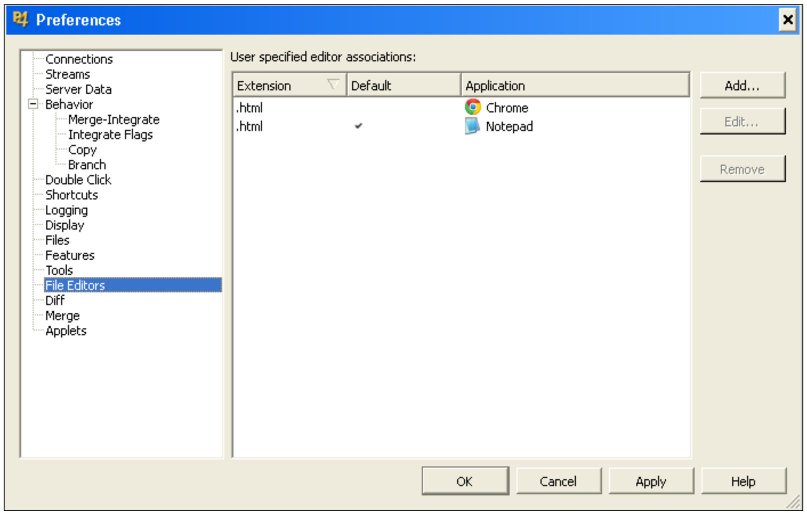
Maximum size of files to preview (excludes audio and video files): 100 KB

Number of changelists, jobs, branch mappings or labels to fetch at a time: 100 (0 = all)

* After limit is reached, changelist and resolve dialogs show plain text list of files

Selecting topics in the left column changes the preference values shown in the right side of the dialog. As we see in the preceding screenshot, **Server Data** displays preference choices that control the granularity of data exchanged with the server. Changes to these values provide you with a level of control of the responsiveness of P4V when it needs data from the server. As the numbers of users grows, the importance of these values becomes more important.

As you can see, there are a large number of preferences; too many to cover at any level of detail in this book. Therefore, we will be providing an overview of the more common preferences that users adjust. As a warning, be careful about modifying any of the preferences associated with the selections under the **Behavior** topic.



User specified editor associations:

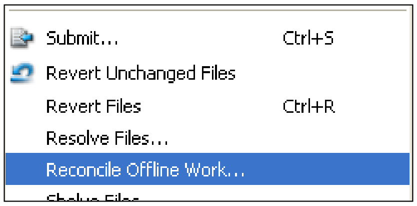
Extension	Default	Application
.html		Chrome
.html	✓	Notepad

The **File Editors**, **Diff**, and **Merge** preferences allow you to specify alternative tools for these functions based on the extension of the filename. For example, a browser is often the default for the `.html` files. Developers on the other hand often want to launch an editor rather than a browser, but there are still times when they would prefer to launch a browser. P4V allows you to specify both with the editor as a default.

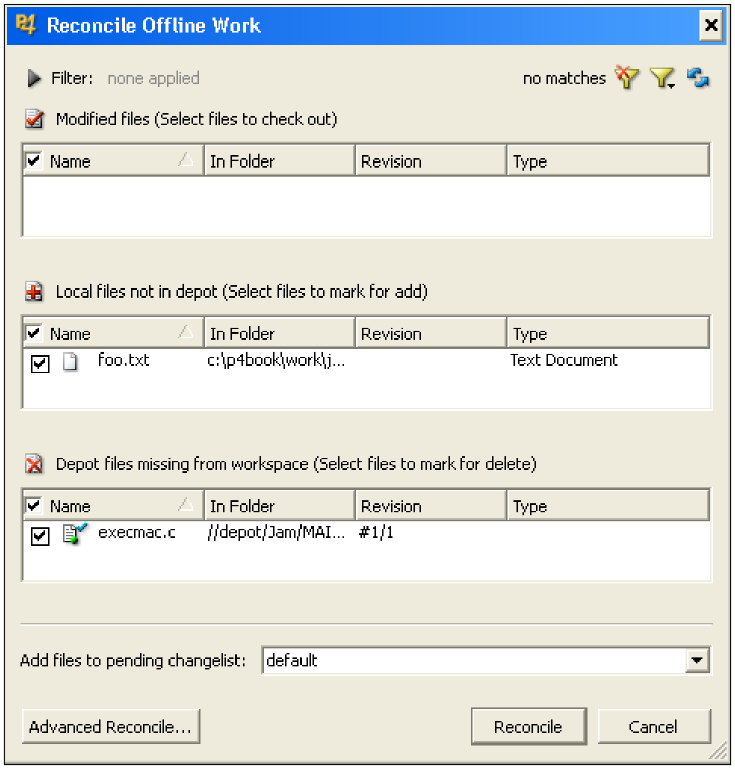
> An interesting follow-along is to specify multiple alternate applications for the `.html` extension (similar to the specification shown previously). Now see how this impacts the **open** and **open with** context menu choices.

Reconciling offline work

There are going to be times when you need to make changes to the files in your workspace but you don't have connectivity with the primary server. If you are offline you can't perform a check out, mark for add, mark for delete or other similar Perforce action. However, you have the files locally and you can manually make them writable and modify them, or delete them. When you reconnect to the primary server you can request to reconcile your offline work using the **Actions | Reconcile Offline Work...** menu, or you can select **Reconcile Offline Work...** from a context menu in the tree pane, as shown in the following screenshot:



Either selection will scan your workspace for files that are different from what the server expects. This saves you having to remember what files were modified, added or deleted while you were offline. These files are organized based on how they differ and are presented in the following dialog:



Modified files (Select files to check out)

☑ Name	In Folder	Revision	Type

Local files not in depot (Select files to mark for add)

☑ Name	In Folder	Revision	Type
☑ foo.txt	c:\p4book\work\j...		Text Document

Depot files missing from workspace (Select files to mark for delete)

☑ Name	In Folder	Revision	Type
☑ execmac.c	//depot/Jam/MAI...	#1/1	

Add files to pending changelist: default

The dialog is organized into files that are added, deleted, and modified. You select files to include in a reconcile changelist. The **Advanced Reconcile...** button at the bottom-left of the dialog brings up a comparison dialog that organizes the files by location. Selecting files from the dialog provides details as to why the file was selected for inclusion as part of the reconcile activities.

> To follow-along, make changes to files in your workspace using the operating system. Then request to reconcile offline work. Be careful not to submit in a production environment.

Summary

At the start of the chapter we said that for most people P4V operates just fine out of the box. This chapter has covered some useful methods to let you adjust P4V to better suit how you work.

A
A Demo Server

In this appendix we provide details and links for a straightforward, step-by-step setup for running examples that accompany the contents of this book.

Examples in the book

The examples in this book are taken from a slightly customized version of the sample repository which Perforce has made available on its website.

This customized test repository is available (together with detailed installation instructions) as detailed in the next section *Sample repository*. This sample repository can be easily installed and used without a Perforce license under their 20/20 licensing model: free for 20 users and 20 workspaces (or unlimited users and 1,000 files). This is without support, although you may already have access to support if your organization has Perforce licenses.

Sample repository

The page to download these files from is:

`http://www.packtpub.com/learning-perforce-scm/book#support`

Please download one of these files:

- `sampledepot.zip` (for Windows users)
- `sampledepot.tar.gz` (for non-Windows users)

The files are roughly 2Mb in size and unzip to about 7Mb.

Platform specific executables

You will need to install the appropriate executables for your platform.

These are located at:

`http://www.perforce.com/downloads/Perforce-Software-Version-Management/complete_list/20-20`

You will need the following files (different versions depending on your operating system):

- `p4d.exe` (just `p4d` on Mac/Unix/Linux)
- `p4.exe` (just `p4` on other operating systems)
- P4V: the full package

Install it

The basic principles for platform-specific instructions are:

1. Download the appropriate server for your operating system (as mentioned in the previous section)
2. Download the zipped test repository (see link mentioned in the previous section)
3. Unzip the repository in a local directory
4. Initialize the Perforce repository in that directory
5. Run the Perforce server (`p4d` or `p4d.exe`) specifying a particular port and the correct directory

The contents of the zip file

When the file is uncompressed, it creates a directory named `PerforceSample`. The `PerforceSample` directory contains the following files:

- `checkpoint` (a file containing Perforce metadata)
- `checkpoint.md5` (checkpoint checksum)
- `readme.html` (a description including installation instructions)
- `server.bat` (a Windows batch file to run the server)

It also contains the following directories containing Perforce archive files:

- `depot`
- `jam`
- `spec`

Detailed installation instructions for Windows

Please note that a copy of these instructions is also available at the web page linked in the previous *Platform specific executables* section. Use the online instructions if they are different to what is contained here:

1. Uncompress the `sampledepot.zip` file using a standard compression utility such as WinZip or 7-Zip. On Windows 7 or above, Windows Explorer can open `.zip` files as compressed folders: right-click the folder, click on **Extract All**, and then follow the instructions.
2. The `PerforceSample` directory where you unzipped the files, for example `c:\PerforceSample`, is known as the `P4ROOT` directory.
3. Copy the `p4d.exe` executable to the `P4ROOT` directory using Windows Explorer.
4. Run a command prompt window (click on **Start**, type `Command Prompt` and click on **Run**).
5. Change to the `P4ROOT` directory:

   ```
   cd c:\PerforceSample
   ```

6. To create your sample Perforce database, restore the checkpoint file by issuing the following command (from the `P4ROOT` directory):

   ```
   p4d -r . -jr checkpoint
   ```

7. To ensure that the database format is consistent with the Perforce server version that you are using, issue the following command:

   ```
   p4d -r . -xu
   ```

8. If the command is successful, you will see the message: **...upgrades done**.
9. Run the file `server.bat` (see the following sections for details).

Detailed installation instructions for non-Windows operating systems

Please note that a copy of these instructions is also available at the web page linked in the previous *Platform specific executables* section. Use the online instructions if they are different to what is contained here.

1. Uncompress the `sampledepot.tar.gz` file using the following commands, we will assume in your `/tmp` directory:

   ```
   cp sampledepot.tar.gz /tmp
   cd /tmp
   gunzip sampledepot.tar.gz
   tar xf sampledepot.tar
   ```

   The `PerforceSample` directory where you unzipped the files, for example `/tmp/PerforceSample`, is known as the `P4ROOT` directory.
2. Copy the `p4d` executable to the `P4ROOT` directory:

   ```
   cp p4d /tmp/PerforceSample
   ```
3. Change to the `P4ROOT` directory:

   ```
   cd /tmp/PerforceSample
   ```
4. To create your sample Perforce database, restore the checkpoint file by issuing the following command (from the `P4ROOT` directory):

   ```
   p4d -r . -jr checkpoint
   ```
5. To ensure that the database format is consistent with the Perforce Server version that you are using, issue the following command:

   ```
   p4d -r . -xu
   ```

   If the command is successful, you will see the message: **...upgrades done**.
6. Run the server with the following command:

   ```
   p4d -r . -p 1666
   ```

Windows privileges and details

The preceding installation instructions will allow you to run the test repository as an ordinary user, without requiring administrator privileges.

However, you may still have problems on locked down corporate installations of Windows. The privileges required are to:

- Download the executables and the zipped repository
- Being able to unzip them to a directory on your local PC
- Being able to run a perforce server executable (`p4d.exe`) which uses a (configurable) port, you may need to be given permission for this

The authors have created a simple batch file (`server.bat`) as part of the zipped sample repository which shows you the command format and provides some simple guidance. You can just run this file from a cmd window, or by double-clicking on it from Windows Explorer:

```
C:\PerforceSample>server.bat

Please note that the following command will not exit and will just
say:
   Perforce Server starting...

This is perfectly normal!  Just minimise this window and leave it
running.
If you close this Window, or press Ctrl+C the server will stop.

C:\PerforceSample>p4d -p 1666 -r c:\PerforceSample\ -vserver=3 -L p4d.
log
Perforce Server starting...
```

The server.bat file is located within the sample repository directory. The contents of the file are:

```
@setlocal
@echo off
REM Start of possible customization
REM You can change the following line to set the value
REM to a different port on the local machine if you wish
REM -----
set P4PORT=1666
REM ------
REM End of customization

REM This sets the environment variable to the path of this file
set CURR_DIR=%~dp0
set P4ROOT=%CURR_DIR%
set P4JOURNAL=%CURR_DIR%journal

REM The "title" command just sets the title of the CMD window
title Perforce server - don't stop!

@echo.
@echo Please note that the following command will not exit and will
just say:
@echo    Perforce Server starting...
@echo.
@echo This is perfectly normal!  Just minimise this window and leave
it running.
@echo If you close this Window, or press Ctrl+C the server will stop.

REM We assume the p4d.exe is present in this directory
@echo on
%P4ROOT%p4d.exe -p %P4PORT% -r %P4ROOT% -vserver=3 -L p4d.log

@endlocal
```

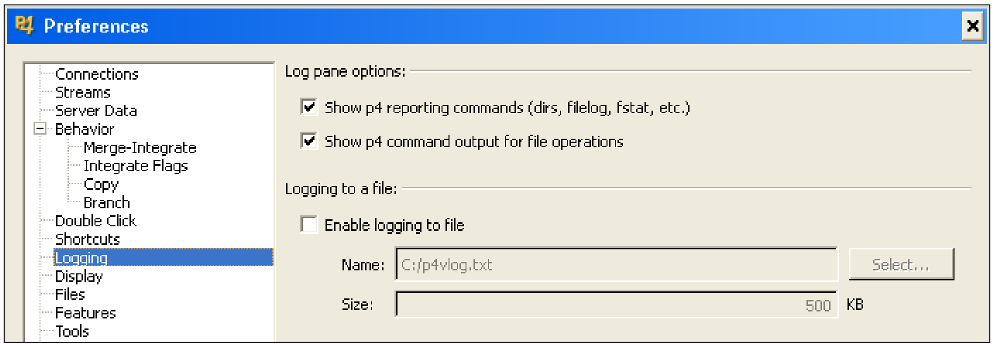
B
Command Line

In this appendix we relate P4V features to the underlying commands that implement them. All Perforce interfaces use a common set of commands. Being able to interpret these commands can help you create and understand the automation associated with builds and other activities.

This appendix is not intended to be an exhaustive description of all commands, which is well provided by the Perforce help and online manuals. We include some examples of basic commands to illustrate the principles and show different types of command.

What is P4V using?

Learning by example is a tried and true method. Fortunately, P4V provides you with full access to the commands that it is using. You control the command details displayed using **Edit | Preferences** shown as follows:



Log pane options:

☑ Show p4 reporting commands (dirs, filelog, fstat, etc.)

☑ Show p4 command output for file operations

Logging to a file:

□ Enable logging to file

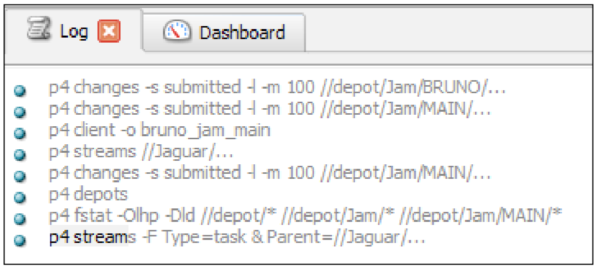
Name: C:/p4vlog.txt

Size: 500 KB

The log panel always contains a record of the file operation commands that it uses. These are the commands that support actions such as check out, add, delete, and submit. By default, P4V trims the detailed information that these commands generate. You can see it with the **Show p4 command output** selection.

P4V needs to update its display information frequently. Most of the time, these commands can be considered as visual clutter. However, this also means that you won't see the commands associated with activities such as populating status panels with information about file histories or changelists. You can see these commands with the **Show p4 reporting commands** checkbox.

The results might look something like the following screenshot:



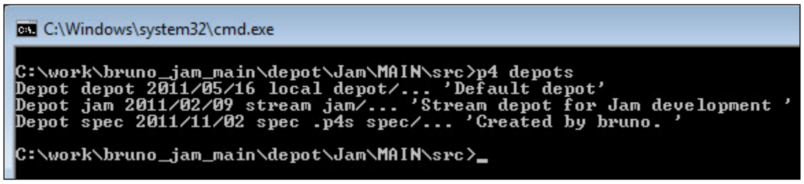
```
p4 changes -s submitted -l -m 100 //depot/Jam/BRUNO/...
p4 changes -s submitted -l -m 100 //depot/Jam/MAIN/...
p4 client -o bruno_jam_main
p4 streams //Jaguar/...
p4 changes -s submitted -l -m 100 //depot/Jam/MAIN/...
p4 depots
p4 fstat -Olhp -Dld //depot/* //depot/Jam/* //depot/Jam/MAIN/*
p4 streams -F Type=task & Parent=//Jaguar/...
```

We can see here various Perforce commands being executed. In most cases, the results are not shown as they are processed by P4V.

The Perforce API

These commands are available via the **P4API** (**Perforce application programming interface**). The Perforce command line client **p4** is a very thin wrapper around the underlying P4API. P4V does much more processing of the results in order to provide a graphical user interface.

One of the very useful features of P4V is that you can copy commands from the log window, paste them into a command line terminal (such as `cmd.exe` on Windows), and run them. They will produce sensible results such as the following:



```
C:\work\bruno_jam_main\depot\Jam\MAIN\src>p4 depots
Depot depot 2011/05/16 local depot/... 'Default depot'
Depot jam 2011/02/09 stream jam/... 'Stream depot for Jam development '
Depot spec 2011/11/02 spec .p4s spec/... 'Created by bruno. '

C:\work\bruno_jam_main\depot\Jam\MAIN\src>_
```

This shows output of the **p4 depots** command on a Windows machine using the `cmd.exe` command prompt. Other p4 commands will produce output in a similar format. All commands take parameters which are discussed in the following section.

Command prompts require appropriate environment settings, this is discussed in the next section.

The easiest way to create such a command prompt with an appropriate environment is to use **Open Command Window Here** from near the bottom of the depot or workspace trees right-click menu.

Command-line help is always available

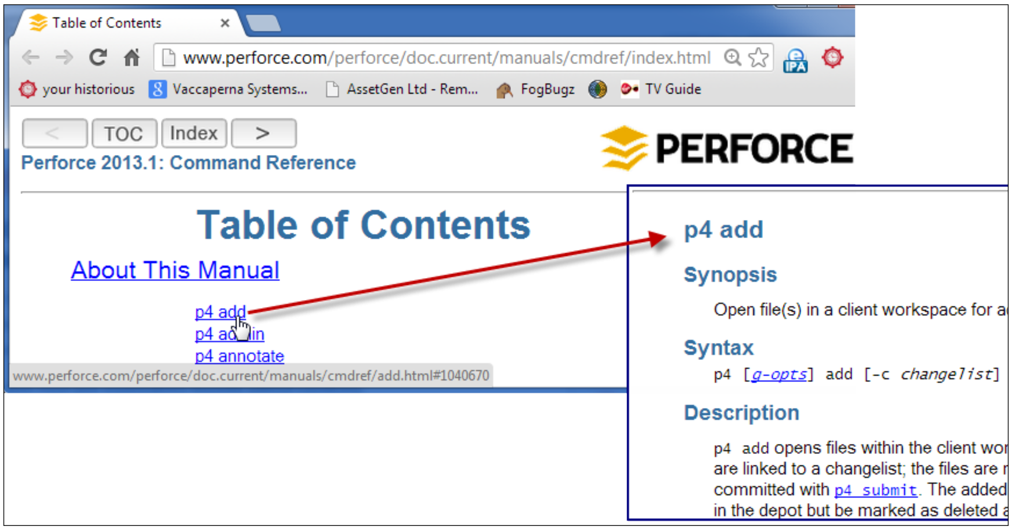
There are two options for getting help on the Perforce command line.

P4 Command Reference Guide

This is available from the Perforce website in two formats:

- HTML:
  `http://www.perforce.com/perforce/doc.current/manuals/cmdref/index.html`
- Downloadable PDF document:
  `http://www.perforce.com/perforce/doc.current/manuals/cmdref/cmdref.pdf`

Notice that the URLs are not version specific, so you will get the latest version of the particular files.



Open file(s) in a client workspace for a

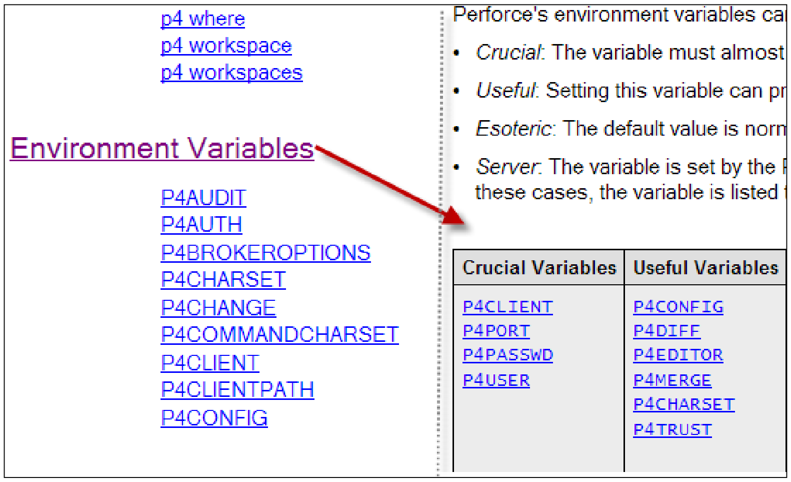
```
p4 [g-opts] add [-c changelist]
```

p4 add opens files within the client wor
are linked to a changelist; the files are r
committed with p4 submit. The added
in the depot but be marked as deleted a

As shown previously, the HTML version of the help includes a complete list of all Perforce commands. The specific page for each command explains command line flags and cross links to other related commands.

Setting environment variables

This is fully discussed in the command reference manual, towards the bottom of the table of contents:



p4 where
p4 workspace
p4 workspaces

Environment Variables

P4AUDIT
P4AUTH
P4BROKEROPTIONS
P4CHARSET
P4CHANGE
P4COMMANDCHARSET
P4CLIENT
P4CLIENTPATH
P4CONFIG

Perforce's environment variables ca

- *Crucial*: The variable must almost
- *Useful*: Setting this variable can pr
- *Esoteric*: The default value is norm
- *Server*: The variable is set by the these cases, the variable is listed

Crucial Variables	Useful Variables
P4CLIENT	P4CONFIG
P4PORT	P4DIFF
P4PASSWD	P4EDITOR
P4USER	P4MERGE
	P4CHARSET
	P4TRUST

Note the crucial variables which define the Perforce server you are communicating with (**P4PORT**), your username (**P4USER**), and your current workspace (**P4CLIENT**).

These will be set automatically if you use the **Open Command Window Here** option from P4V's right-click menu as discussed previously. Note that P4V itself does not take account of environment variables. Instead, it uses the settings defined by the **Connection** menu and **Edit | Preferences** options.

Run the `p4 set` command:

```
C:\work\bruno_jam_main\depot\Jam\MAIN\src>p4 set
P4CLIENT=bruno_jam_main
P4CONFIG=p4config.txt (set) (config 'noconfig')
P4EDITOR=C:\Windows\notepad.exe (set)
P4PORT=1666
P4USER=bruno
```

The output shows us the key environment variables already set up in the command window: `P4PORT` (`1666` is a port on the local machine), `P4USER`, and `P4CLIENT`.

The info command also shows us basic server information:

```
C:\work\bruno_jam_main\depot\Jam\MAIN\src>p4 info
User name: bruno
Client name: bruno_jam_main
Client host: Robert-Laptop
Client root: C:\work\bruno_jam_main
Current directory: c:\work\bruno_jam_main\depot\Jam\MAIN\src
Peer address: 127.0.0.1:52808
Client address: 127.0.0.1
Server address: Robert-Laptop:1666
Server root: C:\perforce\p4book\
Server date: 2013/08/26 09:42:58 +0100 GMT Daylight Time
Server uptime: 155:31:02
Server version: P4D/NTX64/2013.2.BETA/663432 (2013/06/25)
Server license: none
Case Handling: insensitive
```

A commonly seen error message for the info command is:

```
Perforce client error:
        Connect to server failed; check $P4PORT.
        TCP connect to 1248 failed.
        connect: 127.0.0.1:1248: WSAECONNREFUSED
```

This suggests that either we have configured the wrong value for P4PORT, or perhaps that the server is not running, or there is a network issue. Check with your administrator to resolve this.

> If you do much work at the command line then we recommend reading up on the P4CONFIG environment variable. It makes life much more convenient!

P4 help

Help is always available when running at the command line:

```
C:\work\bruno_jam_main>p4 help

    Perforce -- the Fast Software Configuration Management System.
    p4 is Perforce's client tool for the command line.  Try:

        p4 help simple          list most common commands
        p4 help commands        list all standard commands
        p4 help command         help on a specific command

        p4 help administration  help on specialized administration topics
        p4 help charset         help on character set translation
        p4 help configurables   list server configuration variables
        p4 help environment     list environment and registry variables
        p4 help filetypes       list supported file types
        p4 help jobview         help on jobview syntax
        p4 help networkaddress  help on network address syntax
        p4 help revisions       help on specifying file revisions
        p4 help streamintro     introduction to streams
        p4 help usage           generic command line arguments
        p4 help views           help on view syntax

        p4 help legal           legal and license information

    The full user manual is available at http://www.perforce.com/manual.
```

It is often useful to refer to help on specific commands, a list of which is available:

```
C:\work\bruno_jam_main>p4 help commands

    Perforce client commands:

        add          Open a new file to add it to the depot
        annotate     Print file lines along with their revisions
        attribute    Set per-revision attributes on revisions
        branch       Create or edit a branch specification
        branches     Display list of branches
        change       Create or edit a changelist description
        changes      Display list of pending and submitted changelists
        changelist   Create or edit a changelist description
```

```
changelists  Display list of pending and submitted changelists
client       Create or edit a client specification and its view
clients      Display list of known clients
:
```

Basic notes on using commands

Many commands have a singular version (for example, workspace) that provides for create, modify, and delete of an individual element. The plural version of that same command (for example, workspaces) lists all those elements. This makes descriptions of what you're trying to achieve using a command grammatically correct, except of course for the few commands such as `sync`, `fstat`, and `filelog` which aren't actual English words.

Command options

Many commands provide a preview capability. Unlike most systems, Perforce preview output is the same as the output generated for full functionality. This can be very useful for development and debugging. The `-n` option provides preview for the commands that support it.

Filters are provided by command options and arguments. Using filters can significantly reduce the overhead associated with processing a command.

If you look closely you'll find that P4V uses command options that provide functionality that is unique to the needs of GUIs such as P4V. You don't need to duplicate every command option that you see P4V use.

> If you are not sure which command is the right one, you will get helpful responses (including perhaps from the authors!) if you post a question on the Perforce User Forums (`http://forums.perforce.com/`). Posts are also mirrored to a mailing list which you can subscribe to (see `http://www.perforce.com/community` for more options).

Command input and output

There are several different types of command. They all take parameters and options. They can be loosely classified as:

- Action commands: such as `edit`, `add`, and `delete` which typically act on one or more files, and report the results on standard output. They usually only work on files within the current workspace.

- Reporting commands such as `files`, `filelog`, and `opened` which do not affect files, but reports the results of the command on standard output. These commands do not necessarily depend on the current workspace.
- Editor form commands such as `submit` or `client` (`workspace`) which require the user to edit a formatted temporary file with fields to specify the input, and then report the results on standard output.
- Interactive commands such as `login` or `resolve` which prompt the user for input and then act on the results. Note that `resolve` could also be classified as an action command.

Logging in – an interactive command

For normal production servers you need to log in and you will be prompted to type in your password:

```
C:\work >p4 login

Enter password:

User bruno logged in.
```

Therefore, the password is specified on standard input, and the results are shown on standard output.

Action commands

Examples of these commands include the `sync` command which updates the current client workspace, for example:

```
C:\work\bruno_jam_main\depot\Jam\MAIN\src>p4 sync

//depot/Jam/MAIN/src/Build.com#8 - added as C:\work\bruno_jam_main\depot\
Jam\MAIN\src\Build.com

//depot/Jam/MAIN/src/Build.mpw#1 - added as C:\work\bruno_jam_main\depot\
Jam\MAIN\src\Build.mpw

//depot/Jam/MAIN/src/c.txt#1 - added as C:\work\bruno_jam_main\depot\Jam\
MAIN\src\c.txt

//depot/Jam/MAIN/src/command.c#8 - added as C:\work\bruno_jam_main\depot\
Jam\MAIN\src\command.c
```

The `edit` command performs the equivalent of checkout in P4V:

```
C:\work\bruno_jam_main\depot\Jam\MAIN\src>p4 edit Build.com

//depot/Jam/MAIN/src/Build.com#8 - opened for edit
```

Reporting commands

Examples of these include the `opened` command to show files in a pending changelist within the current workspace:

```
C:\work\bruno_jam_main\depot\Jam\MAIN\src>p4 opened
//depot/Jam/MAIN/src/Build.com#8 - edit default change (text)
//depot/Jam/MAIN/src/d.txt#1 - add default change (text)
//depot/Jam/MAIN/src/pathvms.c#6 - delete default change (text)
```

The `files` command shows the names of files matching the parameter:

```
C:\work\bruno_jam_main\depot\Jam\MAIN\src>p4 files jam*.c
//depot/Jam/MAIN/src/jam.c#35 - edit change 352 (text)
//depot/Jam/MAIN/src/jambase.c#33 - edit change 358 (text)
//depot/Jam/MAIN/src/jamgram.c#21 - edit change 351 (text)
```

The `filelog` command shows history for one or more files.

```
C:\work\bruno_jam_main\depot\Jam\MAIN\src>p4 filelog Build.com
//depot/Jam/MAIN/src/Build.com
... #9 change 12128 edit on 2013/08/26 by bruno@bruno_jam_main (text)
'Fix to bug 728 Removed file no '
... #8 change 12108 edit on 2013/07/31 by bruno@bruno_jam_main (text)
'Basic actions edit, add & delet'
... #7 change 346 edit on 2002/11/12 by earl@earl-dev-guava (text) 'Mucho
jam reorganization in ant'
... ... branch into //depot/Jam/REL2.2/src/Build.com#1
... ... branch into //jam/main/src/Build.com#1
... #6 change 289 edit on 2001/12/21 by earl@earl-dev-yew (text) 'Changes
from sybase. '
... #5 change 139 edit on 2000/05/02 by earl@earl-dev-guava (text) 'VMS
DECC changes. '
... ... branch into //depot/Jam/REL2.1/src/Build.com#1
... #4 change 109 edit on 2000/02/28 by earl@earl-dev-guava (text) 'No !.
'
... #3 change 81 edit on 2000/02/08 by earl@earl-dev-guava (text) 'VMS
mei-larch. '
... #2 change 36 edit on 2000/01/11 by earl@earl-dev-guava (text) 'VMS
ready. '
... #1 change 1 add on 1999/09/23 by earl@earl-dev-guava (text) 'Initial
revision '
```

An editor form command – submitting a changelist

The difference with the `submit` command (and other similar commands) is that it requires an editor form. Because the amount of information for the command is large, when you run the command you are put into an editor for a temporary file. You modify the file, save it, and exit. P4 parses the file, and if valid, the command is executed. On Windows the default editor is `Notepad.exe`. This is configurable via the `P4EDITOR` environment variable on your operating system.

An example of a submit temporary file is:

```
# A Perforce Change Specification.
#
#  Change:      The change number. 'new' on a new changelist.
#  Date:        The date this specification was last modified.
#  Client:      The client on which the changelist was created.  Read-
only.
#  User:        The user who created the changelist.
#  Status:      Either 'pending' or 'submitted'. Read-only.
#  Type:        Either 'public' or 'restricted'. Default is 'public'.
#  Description: Comments about the changelist.  Required.
#  Jobs:        What opened jobs are to be closed by this changelist.
#               You may delete jobs from this list.  (New changelists
only.)
#  Files:       What opened files from the default changelist are to be
added
#               to this changelist.  You may delete files from this list.
#               (New changelists only.)

Change:    new

Client:    bruno_jam_main

User:    bruno

Status:    new

Description:
    <enter description here>

Files:
    //depot/Jam/MAIN/src/Build.com    # edit
    //depot/Jam/MAIN/src/d.txt    # add
    //depot/Jam/MAIN/src/pathvms.c    # delete
```

Lines beginning with `#` are comment lines for information purposes only. Notice that some fields are commented to be `read-only`, for example, `Status`. If you modify the values of such fields then p4 will give an error.

We need to modify `Description`, and we can optionally modify some of the other fields, including deleting lines from the `Files` section. This would not submit them as part of this changelist, but leave them opened in the workspace.

In this instance we can modify `Description` to be:

```
Description:
    Fix to bug 728
    Removed file no longer needed.
```

Lines for fields such as `Description` must start with a space or a tab, and may be multi-line as shown previously.

When we save the file, and exit the editor, P4 shows us the results of the submit:

```
C:\work\bruno_jam_main\depot\Jam\MAIN\src>p4 submit
Change 12128 created with 3 open file(s).
Submitting change 12128.
Locking 3 files ...
edit //depot/Jam/MAIN/src/Build.com#9
add //depot/Jam/MAIN/src/d.txt#1
delete //depot/Jam/MAIN/src/pathvms.c#7
Change 12128 submitted.
```

This shows a successful submit.

> Commands such as submit which take an editor form, also take parameters such as `-o` and `–i` which write the form to standard output and read it from standard input. This makes it easier for scripting. Consult the help for more information.

Command summary

Some commands are used frequently. This section organizes the commands by the chapter where they are first used. Use the previous help options and the command classification examples to find out more details about the commands mentioned.

Tree reporting commands

These commands are used to build trees. Where they are used to provide chapter specific information we'll repeat them.

- `clients`: list all clients (workspaces)
- `client`: show information for a particular client (workspace)
- `depots`: list depots
- `dirs`: list directories
- `fstat`: list files within a directory and establish file status information

File information – chapter 2

These commands are specific to file information operations:

- `changes`: lists changelists
- `fstat`: file attribute information
- `files`: file names (more appropriate for command line users)
- `sizes`: file size information

Basic operations – chapter 3

These commands are specific to the basic operations:

- `login`: login to the server
- `sync`: populate a workspace (synchronize workspace with server)
- `edit`: edit a file (check out)
- `add`: add a file
- `delete`: delete a file
- `diff`: establish file differences
- `revert`: revert a change
- `submit`: submit a change

Changelists – chapter 4

These commands are specific to changelist operations:

- `changes`: list changelists
- `change`: create, edit, delete a changelist

Detailed file information – chapter 5

These are commands specific to detailed file information operations:

- `filelog`: file history
- `fstat`: detailed file information
- `labels`: list labels
- `annotate`: generates time-lapse details

Workspaces – chapter 6

These are commands specific to workspace operations:

- `workspaces`: list workspaces (or clients)
- `workspace`: define, edit, delete a workspace (or client)

Dealing with conflicts – chapter 7

These are commands specific to conflict operations:

- `resolve`: resolve a conflict
- `lock`: lock a file
- `unlock`: unlock a file

Branching – chapter 8

These are commands specific to branching operations:

- `integrate`: create, merge, propagate branches (also known as integ)
- `propagate`: a specific variation of integrate for initial population
- `resolve`: resolve conflicts
- `fstat` and `filelog`: used to generate revision graphs
- `branch`: create, edit, delete branch mappings

Streams – chapter 9

These are commands specific to stream operations:

- `stream`: create, modify, delete a stream
- `streams`: list streams

- `propagate`: for initial creation of streams
- `integrate`: merge-down between streams
- `copy`: for use on copy-up between streams

Some basic best practices

P4V caches command responses. Many P4V commands respond with a wealth of information. Caching those responses and re-scanning, as opposed to making duplicate requests, limits the overhead associated with automated processing.

Scripting Perforce

Because of the power of the underlying API, Perforce is very easy to script and automate. Details of this are outside the scope of this book, but you can find a lot of information on their website or on the forums.

> We have done lots of scripting over the years. While it is possible to run P4 commands as described previously and parse the results, we always prefer to use languages such as Python, Ruby, Perl, .Net, or Java. Each of these languages has its own fully supported API for Perforce.

If you are going to parse the results, then it might be easier to do if you use the `-ztag` option. See the different format of the output for two variants of the same command as follows:

```
C:\work\bruno_jam_main\depot\Jam\MAIN\src>p4 files jam*.c
//depot/Jam/MAIN/src/jam.c#35 - edit change 352 (text)
//depot/Jam/MAIN/src/jambase.c#33 - edit change 358 (text)
//depot/Jam/MAIN/src/jamgram.c#21 - edit change 351 (text)
C:\work\bruno_jam_main\depot\Jam\MAIN\src>p4 -ztag files jam*.c
... depotFile //depot/Jam/MAIN/src/jam.c
... rev 35
... change 352
... action edit
... type text
... time 1037232364
... depotFile //depot/Jam/MAIN/src/jambase.c
... rev 33
... change 358
... action edit
... type text
... time 1040691992

... depotFile //depot/Jam/MAIN/src/jamgram.c
... rev 21
... change 351
... action edit
... type text
... time 1037140869
```

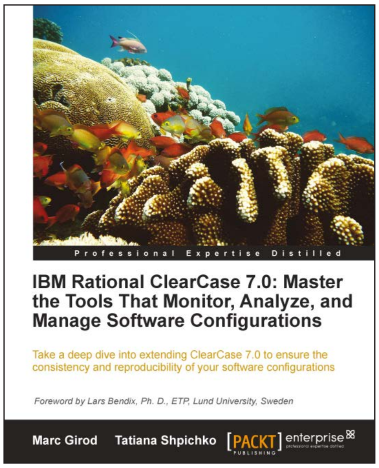
Summary

The Perforce command line is powerful, and all of the capabilities of P4V are available by running one or more commands. Our aim is to give you a taster for this and encourage you to explore on your own. We have not aimed to be exhaustive as that is outside the scope of this book.

In our experience, people who are used to working at the command line, such as Unix/Linux users, will quickly pick up the necessary Perforce commands required for their everyday work.

Index

[PACKT] enterprise
professional expertise distilled
PUBLISHING


Professional Expertise Distilled
IBM Rational ClearCase 7.0: Master the Tools That Monitor, Analyze, and Manage Software Configurations
Take a deep dive into extending ClearCase 7.0 to ensure the consistency and reproducibility of your software configurations
Foreword by Lars Bendix, Ph. D., ETP, Lund University, Sweden
Marc Girod
Tatiana Shpichko
[PACKT] enterprise

IBM Rational ClearCase 7.0: Master the Tools That Monitor, Analyze, and Manage Software Configurations

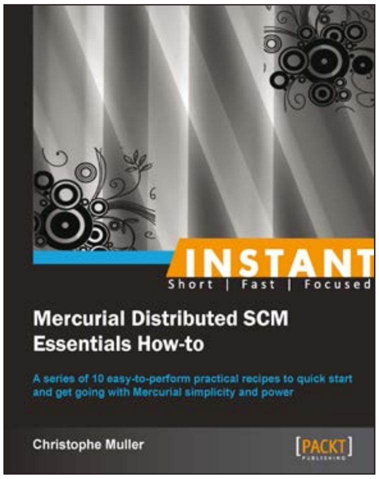
ISBN: 978-1-84968-012-7 Paperback: 360 pages

Take a deep dive into extending ClearCase 7.0 to ensure the consistency and reproducibility of your software configurations

1. Master ClearCase from the inside out: Go technical for consistent, well-structured, and robust software

2. Cut out chaos! Introduce order through reproduction with clearmake

3. Use and extend ClearCase in manageable collaborations


INSTANT
Short | Fast | Focused
Mercurial Distributed SCM Essentials How-to
A series of 10 easy-to-perform practical recipes to quick start and get going with Mercurial simplicity and power
Christophe Muller
[PACKT]

Instant Mercurial Distributed SCM Essentials How-to

ISBN: 978-1-78216-991-8 Paperback: 64 pages

A series of 10 easy-to-perform, practical recipes to make the most of Mercurial's simplicity and power

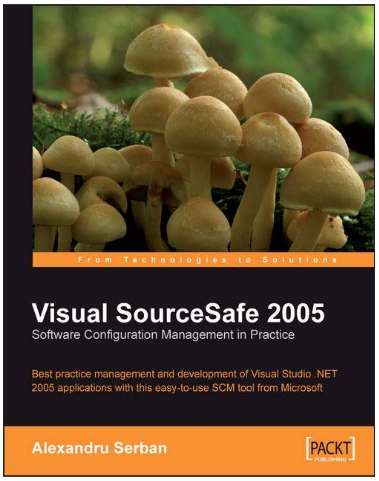
1. Learn something new in an Instant! A short, fast, focused guide delivering immediate results.

2. Learn version control quickly using Mercurial basics and advanced features

3. Set up and work with the Mercurial server for collaborative software development

Please check **www.PacktPub.com** for information on our titles

[PACKT] enterprise
PUBLISHING
professional expertise distilled



From Technologies to Solutions

Visual SourceSafe 2005
Software Configuration Management in Practice

Best practice management and development of Visual Studio .NET 2005 applications with this easy-to-use SCM tool from Microsoft

Alexandru Serban

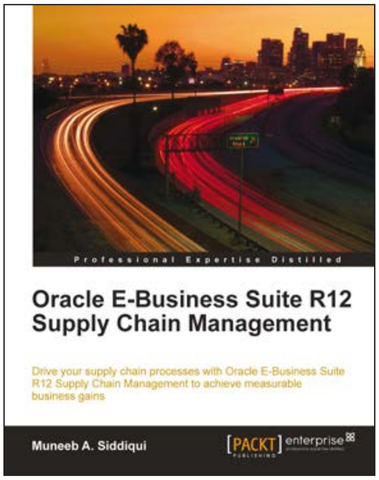
[PACKT]

Visual SourceSafe 2005 Software Configuration Management in Practice

ISBN: 978-1-90481-169-5 Paperback: 404 pages

Best practice management and development of Visual Studio .Net 2005 applications with this easy-to-use SCM tool from Microsoft

1. SCM fundamentals and strategies clearly explained
2. Real-world SOA example: a hotel reservation system
3. SourceSafe best practices across the complete lifecycle
4. Multiple versions, service packs and product updates



Professional Expertise Distilled

Oracle E-Business Suite R12
Supply Chain Management

Drive your supply chain processes with Oracle E-Business Suite R12 Supply Chain Management to achieve measurable business gains

Muneeb A. Siddiqui

[PACKT] enterprise

Oracle E-Business Suite R12 Supply Chain Management

ISBN: 978-1-84968-064-6 Paperback: 292 pages

Drive your supply chain processes with Oracle E-Business R12 Supply Chain Management to achieve measurable business gains

1. Put supply chain management principles to practice with Oracle EBS SCM
2. Develop insight into the process and business flow of supply chain management
3. Set up all of the Oracle EBS SCM modules to automate your supply chain processes
4. Case study to learn how Oracle EBS implementation takes place

Please check **www.PacktPub.com** for information on our titles