Extracted from:

Pragmatic Version Control

with CVS

This PDF file contains pages extracts from Pragmatic Version Control, one of the Pragmatic Starter Kit series of books for project teams. For more information, visit http://www.pragmaticprogrammer.com/starter_kit.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printer versions; the content is otherwise identical.

Copyright © 2004 The Pragmatic Programmers, LLC. All rights reserved.
No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

Chapter 2

What Is Version Control?

A version control system is a place to store all the various revisions of the stuff you write while developing an application. They're basically very simple systems. Unfortunately, over the years, various people have started using different terms for the various components of version control. And this can lead to confusion. So let's start off by defining some of the terms that *we'll* be using.

2.1 The Repository

You may have noticed that we wimped out; we said that, "a version control system is a place to store... the stuff you write," but we never said exactly where all this stuff is stored. In fact, it all goes in the *repository.*

repository

In almost all version control systems, the repository is a central place that holds the master copy of all versions of your project's files. Some version control systems use a database as the repository, some use regular files, and some use a combination of the two. Either way, the repository is clearly a pivotal component of your version control strategy. You need it sitting on a safe, secure, and reliable machine. And it should go without saying that it needs to get backed up regularly.

In the old days, the repository and all its users had to share a machine (or at least share a filesystem). This turns out to be fairly limiting; it was hard to have developers working at

Different Flavors of Networked Access

The writers of version control systems sometimes have different definitions of what "networked" means. For some, it means accessing the files in a repository over shared network drives (such as Windows shares or NFS mounts). For others it means having a client-server architecture, where clients interact with server repositories over a network. Both can work (although the former is hard to design correctly if the underlying file-sharing mechanism doesn't support locking reliably). However, you may find that deployment and security issues dictate which systems you can use.

If a version control system needs access to shared drives, and you need to access it from outside your internal network, then you'll need to make sure that your organization allows you to access the data this way. Virtual Private Network (VPN) packages allow this kind of secure access, but not all companies run VPNs.

CVS uses the client-server model for remote access.

different sites, or working on different kinds of machines or operating systems. As a result, most version control systems today support networked operation; as a developer you can access the repository over a network, with the repository acting as a server and the version control tools acting as clients. This is tremendously enabling. It doesn't matter where the developers are; as long as they can connect over a network to the repository, they can access all the project's code and its history. And they can do it securely; you can even use the Internet to access your repository without sharing your precious source code with a nosy competitor. Andy and I regularly access our source code over the Internet when we're on the road.

This does lead to an interesting question, though. What happens if you need to do development, but you don't have a network connection to your repository? The simple answer is, "it depends." Some version control systems are designed solely

Prepared exclusively for xxyyxxzzxx

for use while connected to the repository; it is assumed that you'll always be online, and that you won't be able to change source code without first contacting the central repository. Other systems are more lenient. The CVS system, which we use for our examples in this book, is one of the latter. We can edit away on our laptops at 35,000 feet, and then resynchronize the changes when we get to our hotel rooms. This online/offline issue is a crucial one when choosing a version control system; make sure that whatever product you choose supports your style of working.

2.2 What Should We Store?

All the things in your project are stored in the repository. But what exactly are the *things* we're talking about?

Well, you obviously need program source files to build your project: the Java, or C#, or VB, or whatever language you're using to write your application. In fact, some folks think that this source code is such an important component of version control that they use the term "Source Code Control Systems."

The source code is certainly important, but many people make the mistake of forgetting all the other things that need to be stored under version control. For example, if you're a Java programmer, you may use the Ant tool to compile your source. Ant uses a script, normally called `build.xml`, to control what it does. This script is part of the build process; without it you can't build the application, so it should be stored in the version control system.

Similarly, many projects use metadata to drive their configuration. This metadata should be in the repository too. So should any scripts you use to create a release CD, test data used by QA, and so on.

In fact, there's an easy test when it comes to deciding what goes in and what stays out. Simply ask yourself "if we didn't have an up to date version of *x*, could we build and deliver our application?" If the answer is "no," then *x* should be in the repository.

Prepared exclusively for xxyyxxzzxx

Joe Asks...

What About Generated Artifacts?

If we store all the things needed to build the project, does that mean that we should also be storing all the generated files? For example, we might run JavaDoc to generate the API documentation for our source tree. Should that documentation be stored in the version control system's repository?

The simple answer is "no." If a generated file can be reconstituted from other files, then storing it is simply duplication. Why is this duplication bad? It isn't because we're worried about wasting disk space. It's because we don't want things to get out of step. If we store the source and the documentation, and then change the source, the documentation is now outdated. If we forget to update it and check it back in, we've now got misleading documentation in our repository. So in this case, we'd want to keep a single source of the information, the source code. The same rules apply to most generated artifacts.

Pragmatically, some artifacts are difficult to regenerate. For example, you may have only a single license for a tool that generates a file needed by all the developers, or a particular artifact may take hours to create. In these cases, it makes sense to store the generated artifacts in the repository. The developer with the tool's license can create the file, or a fast machine somewhere can create the expensive artifact. These can be checked in and all other developers can then work from these generated files.

As well as all the files that go toward creating the released software, you should also store all your non-code project artifacts under version control (anything that you'll need to make sense of things later on), including the project's documentation (both internal and external). It might also include the text of significant e-mails, minutes of meetings, information you find on the web—anything that contributes to the project.

Prepared exclusively for xxyyxxzzxx

2.3 Workspaces and Manipulating Files

The repository stores all the files in our project, but that doesn't help us much if we need to add some magic new feature into our application; we need the files where *we* can get to them. This place is called our local *workspace*. The workspace is a local copy of all of the things that we need from the repository to work on our part of the project. For small to medium-sized projects, the workspace will probably simply be a copy of all the code and other artifacts in the project. For larger projects, you may arrange things so that developers can work with just a subset of the project's code, saving them time when building, and helping to isolate subsystems of the system. You might also hear the workspace called the *working directory* or the *working copy* of the code.

workspace

In order to populate our workspace initially, we need to get things out of the repository. Different version control systems have different names for this process, but the most common (and the one used by CVS) is *checking out*. When you check out from the repository, you extract local copies of files into your workspace.[1] The check out process ensures that you get up-to-date copies of the files you request, and that these files are copied into a directory structure that mirrors that of the repository.

check out

As you work on a project, you'll make changes to the project's code in your local workspace. Every now and then you'll reach a point where you'll want to save your changes back to the repository. This process is called *committing*; you're committing your changes back into the repository.

commit

Of course, all the time that you're making changes, so are other members of your team. They'll also be committing their changes to the repository. However, these changes do not affect your local workspace; it doesn't suddenly change just because someone else saved changes back into the repository. Instead, you have to instruct the version control system to *update* your local workspace. During the update, you'll receive

update

[1]Even if you do your work on the same computer that stores the repository, you'll still need to check files out before using them; the repository should be treated as a black box.

Prepared exclusively for xxyyxxzzxx

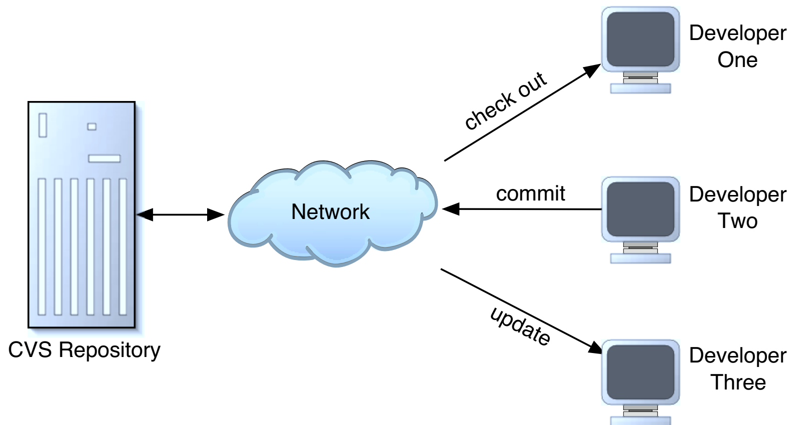

Figure 2.1: CLIENTS AND A REPOSITORY

the latest set of files from the repository. And when your colleagues do an update, they'll receive your latest changes too. (Just to confuse things, however, some folks also use the term "check out" to refer to updating, as they are checking out the latest changes. Because this is a common idiom, we'll also use this at times in this book.) These various interactions are shown in Figure 2.1.

Of course there's a potential problem here: what happens if you and a colleague both want to make changes to the same source file at the same time? It depends on the version control system you're using, but all have ways of dealing with the situation. We talk about this more in the section on page 19 on *locking options*.

2.4 Projects, Modules, and Files

So far we've talked about storing *things*, but we haven't talked about how those things are organized.

At the lowest level, most version control systems deal with individual files.[2] Each file in your project is stored by name

[2]There are some IDE-like environments that perform versioning at the method level, but they're fairly uncommon.

Prepared exclusively for xxyyxxzzxx

in the repository; if you add a file called `Panel.java` to the repository, then other members of your team can check out `Panel.java` into their own workspaces.

However, that's pretty low-level. A typical project might have hundreds or thousands of files, and a typical company might have dozens of projects. Fortunately, almost all version control systems allow you to structure the repository. At the top level, they typically divide your work into projects. With each project, they then let you work in terms of modules (and often submodules). For example, perhaps you are working on *Orinoco*, a large web-based book ordering application. All the files needed to build the application might be stored in the repository under the Orinoco project name. If you wanted to, you could check it all out onto your local disk.

The Orinoco project itself might be broken down into a number of largely independent modules. For example, there might be a team working on credit card processing and another working on order fulfillment. With any luck, the folks in the credit card subproject won't need to have all the project's source to do their job; their code should be nicely partitioned. So when they check out, they really only want to see the parts of the project that they're working on.

CVS allows the repository administrator to divide a project into *modules*. A module is a group of files (normally contained in one or more file system directory trees) that can be checked out by name. Modules can be hierarchical, but they don't have to be; the same file or set of files can appear in many different modules. Modules even let you share code between projects (simply put the files to be shared into a module and let the other team reference it by name).

module

Modules give you many different views into your repository, allowing people in your teams to deal only with the things they need. We talk about modules in Chapter 9 on page 107.

2.5 Where Do Versions Come In?

This book is all about version control systems, but so far all we've talked about is storing and retrieving files in a repository. Where do versions come in?

Prepared exclusively for xxyyxxzzxx

Behind the scenes, a version control system's repository is a fairly clever beast. It doesn't just store the current copy of each of the files in its care. Instead it stores *every version* that has ever been checked in. If you check out a file, edit it, then check it back in, the repository will hold both the original version and the version that contains your changes.[3] Most systems use a simple numbering system for the versions of a file. In CVS, the first version of a file is assigned the revision number 1.1. If a changed version is checked in, that change is given the number 1.2. The next change gets 1.3, and so on. (We'll be talking about more complex numbering soon). Associated with each of these revision numbers is the date and time that the file was checked in, along with an optional comment from the developer describing the change.

version

This system of storing revisions is remarkably powerful. Using it, the version control system can do things such as:

- Retrieve a specific revision of a file.
- Check out all of the source code of a system as it appeared two months ago.
- Tell you what changed in a particular file between versions 1.3 and 1.5.

You can also use the revision system to undo mistakes. If you get to the end of the week and discover you've been going down a blind alley, you can back out all the changes you've made, reverting back to the code as it was on Monday morning.

There's a small wrinkle to the way revisions are numbered. Some version control systems assign a single revision number to all the files affected by a particular check in, while others give each file a unique sequence of revision numbers. CVS falls in to the latter camp. For example, we might check three files out of a repository and get the following version numbers:

```
File1.java      1.10
File2.java      1.7
File3.java      1.9
```

[3] In reality, most version control systems store the differences between versions of a file, rather than complete copies of each revision.

Prepared exclusively for xxyyxxzzxx

We edit `File1.java` and `File3.java`, but leave `File2.java` untouched. If we commit these changes back to the repository, it will increment the revision numbers on those files we changed:

```
File1.java     1.11
File2.java     1.7
File3.java     1.10
```

This means you can't use the individual file version numbers to keep track of things such as project releases (Version 1.3a of the Orinoco project, for example). Because this one point often causes grief in teams just starting to use CVS, let's repeat it. *The individual revision numbers that CVS assigns to files should not be used as external version numbers.* Instead, version control systems provide you with *tags* (or their equivalent).

2.6 Tags

All these revision numbers are great, but as people we seem to be better at remembering names such as "PreRelease2" rather than numbers like 1.47. We also have a problem when the different files that make up a particular release of our software have different revision numbers. In the previous example, we might be ready to ship the software built with `File1.java`, `File2.java`, and `File3.java`, but each file has its own revision number. So how do you tie all these different numbers together?

Tags to the rescue. Version control systems let you assign names to a group of files (or modules, or an entire project) at a particular point in time. If you assigned the tag "PreRelease2" to this group of three files, you could subsequently check them out using that same tag. You'd get revision 1.11 of `File1.java`, 1.7 of `File2.java`, and 1.10 of `File3.java`.

tag

Tags are a great way of keeping track of significant events in the history of your project's code. We'll be using tags extensively later in this document. In fact, tags and branches (the topic of the next section) have their own chapter, starting on page 87.

Prepared exclusively for xxyyxxzzxx

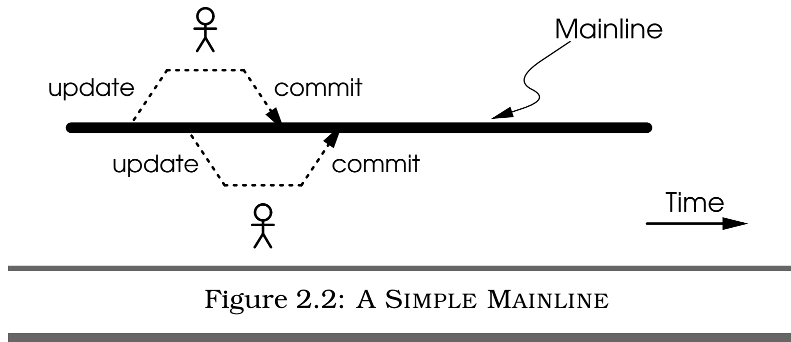

Figure 2.2: A SIMPLE MAINLINE

2.7 Branches

In the normal course of development, most folks are working on a common code base (although they'll likely be working on different parts of it). They'll be checking stuff out, making revisions, and checking the changes back in, and everyone will share this work. This river of code is often called a *mainline*. We show this in Figure 2.2. In this figure (and in the ones that follow) time flows from left to right. The thicker horizontal line represents the progression of code through time; it is the mainline of the development. Individual developers check in and check out code from this mainline into their individual workspaces.

mainline

But consider the time when a new release is about to be shipped. One small subteam of developers may be preparing the software for that release, fixing last minute bugs, working with the release engineers, and helping the QA team. During this vital period, they need stability; it would set back their efforts if other developers were also editing the code, adding features intended for the next release.

One option is to freeze new development while the release is being generated, but this means that the rest of the team is effectively sitting idle.

Another option would be to copy the source software out onto a spare machine and then have the release team just use this machine. But if we do that, what happens to the changes that they make after the copy? How do we keep track of them? If they find bugs in the release code that are also in the mainline, how can we efficiently and reliably merge these

Prepared exclusively for xxyyxxzzxx

fixes back in? And once they've released the software, how do we fix bugs that customers report; how can we guarantee to find the source code in the same state as when we shipped the release?

A far better option is to use the branching capabilities built into version control systems.

branch

Branching is a bit like the hackneyed device in science fiction stories where some event causes time to split. From that point forward there are two parallel futures. Some other event occurs, and one of these futures splits too. Soon you're dealing with a whole bunch of alternative universes (a great device for resolving the story when you run out of plot ideas).

Branching in a version control system also allows you to create multiple parallel futures, but rather than being populated by aliens and space cowboys, they contain source code and version information.

Take the case of the team about to release a new version of the product. So far, all the team has been working in the *mainline*, the common thread of code shown in Figure 2.2 on the page before. But the release subteam wants to isolate themselves from this mainline. To do this, they create a branch in the repository. From now until their work is done, the release subteam will check out from and check in to this branch. Even after the application is released, this branch will stay active; if customers report bugs, the team will fix them in this release branch. This situation is shown in Figure 2.3 on the following page.

A branch is almost like having a totally separate repository: people using that branch see the source code it contains and operate independently of people working on other branches or the mainline. Each branch has its own history and tracks revisions people make independently (although obviously if you look back past the point where the branch was made you'll see that the branch and the mainline become one).

This is exactly what you want when you're creating releases. The team working on the release will have a stable code base to polish up and ship. In the meantime, the main group of developers can continue making changes to the main line of

Prepared exclusively for xxyyxxzzxx

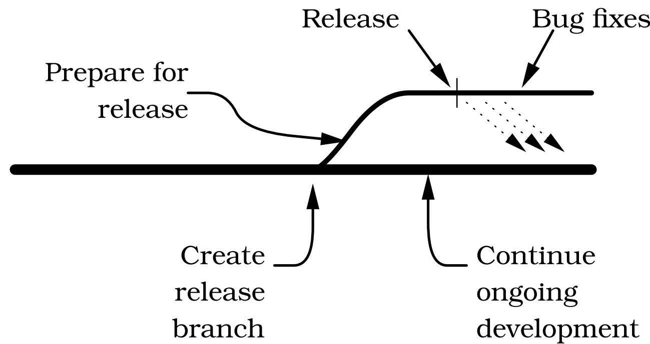

Figure 2.3: MAINLINE WITH A RELEASE BRANCH

code; there's no need for a code freeze while the release takes place. And when customers report problems in the release, the team will have access to the code in the release branch so they can fix the bugs and ship updated releases without including any of the newly developed code in the mainline.

Branches are identified by tags, and file revision numbers within a branch have extra levels in their numbers. So if `File1.java` is at revision 1.14 and you create a branch, you'll find that in the branch it may have a revision number of 1.14.2.1, while in the mainline it's still 1.14. Edit it in the mainline and you'll get revision 1.15; edit in the branch and the revision number will be 1.14.2.2.

You can create branches off of other branches, but typically you won't want to; we've come across many developers who have been put off branching for life because of some bad experiences with overly complicated branching in a project. In this book we'll describe a simple scheme that does everything you'll need but that avoids unnecessary complexity.

2.8 Merging

Back to the science fiction story with the multiple alternate futures. In order to spice up the plot, writers often allow their characters to travel between these different universes using wormholes, polyphase deconfabulating oscillotrons, or just a good strong cup of piping hot tea.

Prepared exclusively for xxyyxxzzxx

You can also travel between alternate futures in a version control system (the cup of tea is optional). Although each checked out version comes from a particular branch, and gets checked back in to that branch, it's easy to have multiple branches checked out on a single developer's machine (in different directories or folders on the hard drive, of course). That way a developer can be working on both the mainline and on (say) bug fixes in a release branch at the same time.

Even better, version control systems support merging. Say you fix a bug in the release branch and realize that the same bug will be present in the mainline code. You can tell the version control system to work out the changes you made to the source while you fixed the bug, and then to apply those changes to the code in the mainline. This largely eliminates the need to cut and paste changes back and forth between different versions of a system. We'll have a lot to say about merging later on.

merge

2.9 Locking Options

Imagine two developers, Fred and Wilma, working on the same project. Each has checked out the project's files onto their respective local hard drives, and each wants to edit their local copy of `File1.java`. What happens when they come to check that file back in?

A bad scenario would be for the version control system to accept Fred's changes, and then accept Wilma's version of the same file. As Wilma's copy won't have Fred's changes in it, storing Wilma's copy in the repository will effectively forget all Fred's hard work.

To stop this happening, version control systems implement some form of conflict resolution system (probably a good thing in the case of Fred and Wilma). There are two common versions of conflict resolution.

The first is called *strict locking*. In a strict locking version control system, all files that are checked out are initially flagged as being "read only." You can look at them, and you can use them to build your application, but you can't edit or change

strict locking

Prepared exclusively for xxyyxxzzxx

them. To do that, you have to ask the repository's permission: "please can I edit File1.java?" If no one else is editing that same file, then the repository gives you permission and changes the permissions of your local copy of the file to be "read/write." You can then edit. If anyone else asks to edit that same file while you have it flagged, they'll be refused. After you've finished your changes and checked the file back in, your local copy reverts back to being read only, and it becomes available for other folks to edit.

The second form of conflict resolution is often called *optimistic locking,* although it really is no locking at all. Here, every developer gets to edit any checked out file: the files are checked out in a read/write state. However, the repository will not allow you to check in a file that has been updated in the repository since you last checked it out. Instead, it asks you to update your local copy of the file to include the latest repository changes before checking in. This is where the cleverness lies. Instead of simply overwriting all your hard work with the latest repository version of the file, the version control system attempts to merge the repository changes with your changes. For example, let's look at `File1.java`:

optimistic locking

```
Line 1  public class File1 {
     -    public String getName() {
     -      return "Wibble";
     -    }
     5    public int getSize() {
     -      return 42;
     -    }
     -  }
```

Wilma and Fred both check this file out. Fred changes line 3:

```
return "WIBBLE";
```

He then checks the file back in. This means that Wilma's copy of the file is out of date. Not knowing this, Wilma changes line 6, so it returns 99 instead of 42. When she goes to check the file in, she's told that her copy is out of date; she needs to merge in the repository changes. This corresponds to the star marked **CONFLICT** in Figure 2.4 on the next page.

When Wilma merges the changes into her file, the version control system is clever enough to spot that Fred's changes do not overlap hers, so it simply updates her local copy with a new

Prepared exclusively for xxyyxxzzxx

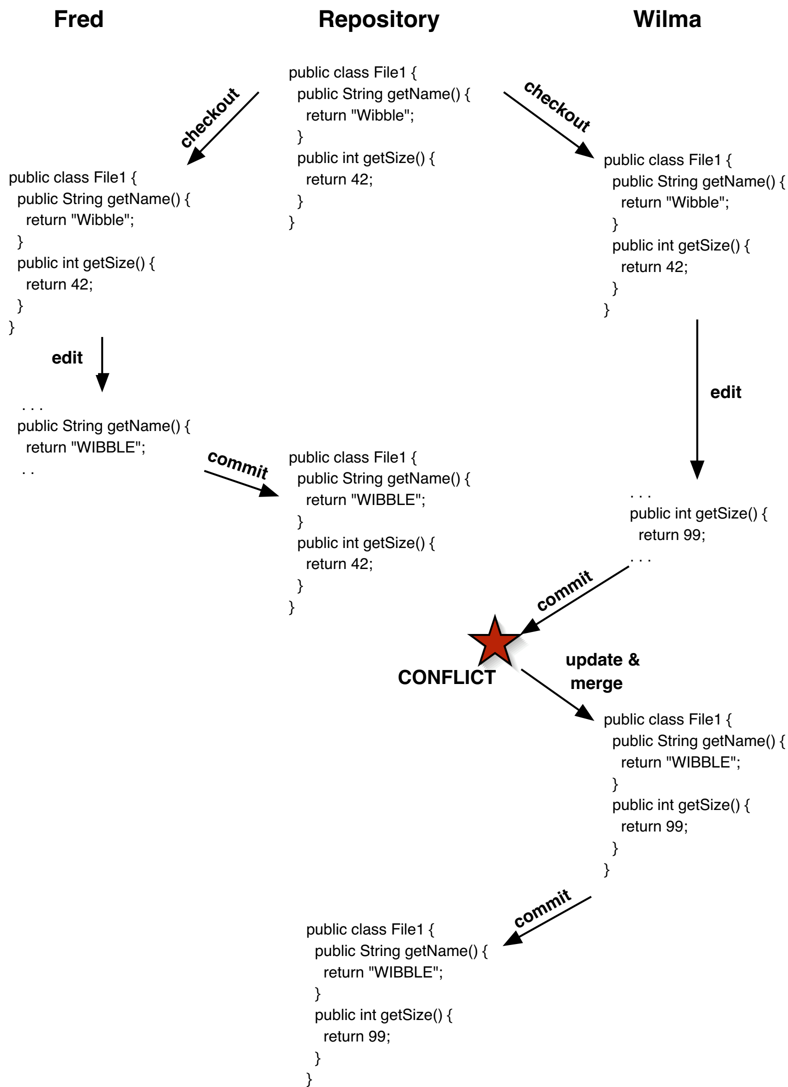

Figure 2.4: FRED AND WILMA MAKE CHANGES TO THE SAME FILE, BUT THE CONFLICT IS HANDLED BY A MERGE.

Prepared exclusively for xxyyxxzzxx

line 3, leaving her changes still in her file. When she checks in, she'll be storing back her changes and leaving Fred's intact.

What happens if Fred and Wilma both updated line 3, but made different changes to it? Assuming Fred checks in first, his changes will be accepted. When Wilma goes to check in, she'll again be told that her copy is out of date. This time, though, when she goes to merge in the repository version the system will notice that she's made a change to a line that has also been changed in the repository. There's a conflict. In this case, Wilma will see some warning messages, and the conflict will be marked up in her copy of the source file. She'll have to resolve it manually (probably by talking with Fred to find out why they were both working on the same line of code).

Given this description you might think that optimistic locking is a somewhat reckless way of developing systems; multiple people editing the same files at the same time. Often people who haven't tried it reason that it can't work, and insist on working only with version control systems that implement strict locking.

In reality, though, strict locking turns out to be a lot of extra hassle with no particular payback. If you try an optimistic locking system (such as CVS) you'll be surprised at just how rarely conflicts arise. It turns out that in practice the normal ways of dividing up work on a team mean that people work on different areas of the code; they don't bump in to each other that often. And when they *do* need to edit the same file, they're often working on different parts of it. In a strict locking system, one would have to wait for the other to finish and check in before proceeding. In an optimistic locking system, both can proceed. We've tried both kinds of locking over the years, and our strong recommendation is that the vast majority of teams should use a version control system with optimistic locking.

Prepared exclusively for xxyyxxzzxx

2.10 Configuration Management (CM)

Sometimes you'll hear folks talking about Configuration Management or Software Configuration Management systems (often abbreviated as CM or SCM). At first sight they seem to be talking about version control. And that's largely true; the practices of CM rely very heavily on having good version control in place. But version control is just one tool used by configuration management.

CM is a set of project management practices that enables you to accurately and reproducibly deliver software. It uses version control to achieve its technical goals, but also uses a lot of human controls and cross checks to make sure that things are not forgotten. You can think of configuration management as a way of identifying the things that get delivered, and version control as a means of recording that identification. CM is a large (and to some extent ill-defined) topic, and we won't be covering it more in this book.

For now, though, let's concentrate on how to use version control systems to get our jobs done. The next chapter is a gentle introduction to a particular version control system, CVS.

Prepared exclusively for xxyyxxzzxx