Extracted from:

Pragmatic Version Control

Using Git

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Pragmatic Version Control

Using Git

The Pragmatic Starter Kit—Volume 1

Travis Swicegood

Edited by Susannah Davidson Pfalzer

Pragmatic Version Control

Using Git

Travis Swicegood

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

A *version control system* (VCS) is a methodology or tool that helps you keep track of changes you make to the files in your project. In its simplest, manual form, a VCS is you creating a copy of the file you're working with and adding the date and time to the end of it.

Being pragmatic, we want something that will help automate that process. This is where VCS tools come in. They track all the changes for us, keeping a copy of every change made to the code in our projects.

Distributed version control systems (DVCSs) are no different in that respect. Their main goal is still to help us track changes we make to the projects we're working on. The difference between VCSs and DVCSs is how developers communicate their changes to each other.

In this chapter, we'll explore what a VCS is and how a DVCS—Git in particular—is different from the traditional, centralized model. You'll learn the following:

- What a repository is
- How to determine what to store
- What working trees are
- How files are manipulated and how to stay in sync
- How to track projects, directories, and their files
- How to mark milestones with tags
- How to track an alternate history with a branch
- What merging is
- How Git handles locking

All of these ideas revolve around the repository, so let's start there.

1.1 The Repository

The *repository* is the place where the version control system keeps track of all the changes you make. Most VCSs store the current state of the code, along with when each change was made, who made it, and a text log message that explains why they made the change.

You can think of a repository like a bank vault and its history like the ledger. Each time a deposit—what is called a *commit* in VCS lingo—is made, your VCS tool adds an entry to the ledger and stores the changes for safekeeping.

Originally, these repositories were accessible only if you were logged directly into the machines they were stored on. That doesn't scale, so tools such as CVS, and later Subversion, were created. They allowed developers to work

remotely from the repository and send their changes back using a network connection.

These systems follow a *centralized repository* model. That means there is one central repository that everyone sends their changes to. Each developer keeps a copy of the latest version of the repository, and whenever they make a change to it, they send that change back to the main repository.

The centralized repository is an improvement over having to directly access the machine where the repository lives, but it still has limitations. First, you have only the latest version of the code. To look at the history of changes, you have to ask the repository for that information.

That brings up the second problem. You have to be able to access the remote repository—normally over a network.

In this age of always-on, broadband Internet connections, we forget that sometimes we don't have access to a network. As I've worked on this book, I've written parts at my home office, in coffee shops, on cross-country plane flights, and on the road (as a passenger) while traveling across country. I even did some of the final editing at a rustic cabin in Lake of the Ozarks, Missouri.

That highlights one of the biggest advantages of a DVCS, which is the model that Git follows. Instead of having one central repository that you and everyone else on your team sends changes to, you each have your own repository that has the entire history of the project. Making a commit doesn't involve connecting to a remote repository; the change is recorded in your local repository.

Let's go back to our bank vault analogy for a minute. A centralized system is like having one bank that every developer on your team uses. A distributed system is like each developer having their own personal bank.

You might be wondering how you can keep in sync with everyone else's changes and make sure they have yours. Each developer still sends their changes back to the main project repository. They can have access to send the changes directly to the main repository (an action called *pushing* in Git), or they might have to submit patches, which are small sets of changes, to the project's maintainer and have them update the main repository.

1.2 What Should You Store?

The short answer: everything.

The slightly less short answer: everything that you need to work on your project. Your repository needs a copy of everything in your project that's essential for you to modify, enhance, and build new versions of it.

The first and most obvious thing you should store in the repository is your project's source code. Without that, you can't fix bugs or implement new features.

Most projects have some sort of build files. A couple of common ones are Makefiles, Rakefiles, or Ant's `build.xml`. These need to be stored so you can compile your source code into something usable.

Other common items to store in your repository are sample configuration files, documentation, images that are used in the application, and of course unit tests.

Determining what to include is easy. Ask yourself, "If I didn't have X, could I do my work on this project?" If the answer is no, you couldn't, then it should be included.

Like all good rules, there is an exception. The rule doesn't apply to tools that you should use. You should include the Ant `build.xml` file but not the entire Ant program.

It's not a hard exception, though. Sometimes storing a copy of Ant or JUnit or some other program in your repository can make sure the entire team is using the same version of the tools you use. These should be stored separately from your project, however.