THE EXPERT'S VOICE® IN SOFTWARE DEVELOPMENT

Pro Git

Everything you need to know about the Git distributed source control tool

Scott Chacon

Foreword by Junio C Hamano, Git project leader

Apress®

Getting Started

This chapter will be about getting started with Git. We will begin at the beginning by explaining som
background on version control tools, then move on to how to get Git running on your system a
finally how to get it setup to start working with. At the end of this chapter you should understand wh
Git is around, why you should use it and you should be all setup to do so.

About Version Control

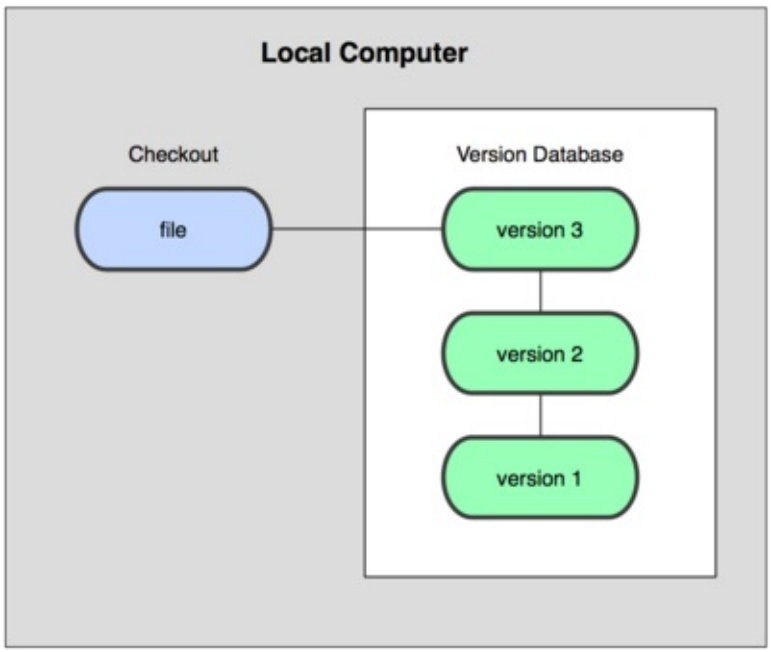
What is version control, and why should you care? Version control is a system that records changes t a file or set of files over time so that you can recall specific versions later. For the examples in th book you will use software source code as the files being version controlled, though in reality you ca do this with nearly any type of file on a computer.

If you are a graphic or web designer and want to keep every version of an image or layout (which yo would most certainly want to), a Version Control System (VCS) is a very wise thing to use. It allow you to revert files back to a previous state, revert the entire project back to a previous state, compa changes over time, see who last modified something that might be causing a problem, who introduce an issue and when, and more. Using a VCS also generally means that if you screw things up or lo files, you can easily recover. In addition, you get all this for very little overhead.

Local Version Control Systems

Many people's version-control method of choice is to copy files into another directory (perhaps time-stamped directory, if they're clever). This approach is very common because it is so simple, b it is also incredibly error prone. It is easy to forget which directory you're in and accidentally write the wrong file or copy over files you don't mean to.

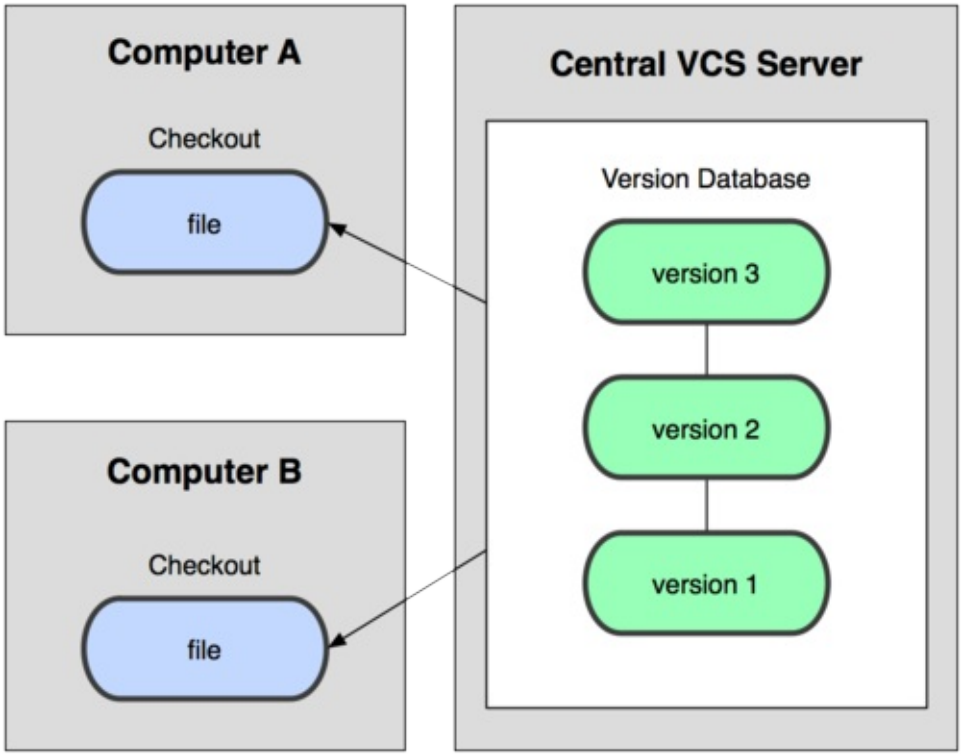
To deal with this issue, programmers long ago developed local VCSs that had a simple database th kept all the changes to files under revision control (see Figure 1-1).



One of the more popular VCS tools was a system called rcs, which is still distributed with mar computers today. Even the popular Mac OS X operating system includes the rcs command when yo install the Developer Tools. This tool basically works by keeping patch sets (that is, the differenc between files) from one change to another in a special format on disk; it can then re-create what an file looked like at any point in time by adding up all the patches.

Centralized Version Control Systems

The next major issue that people encounter is that they need to collaborate with developers on oth
systems. To deal with this problem, Centralized Version Control Systems (CVCSs) were develope
These systems, such as CVS, Subversion, and Perforce, have a single server that contains all t
versioned files, and a number of clients that check out files from that central place. For many year
this has been the standard for version control (see Figure 1-2).

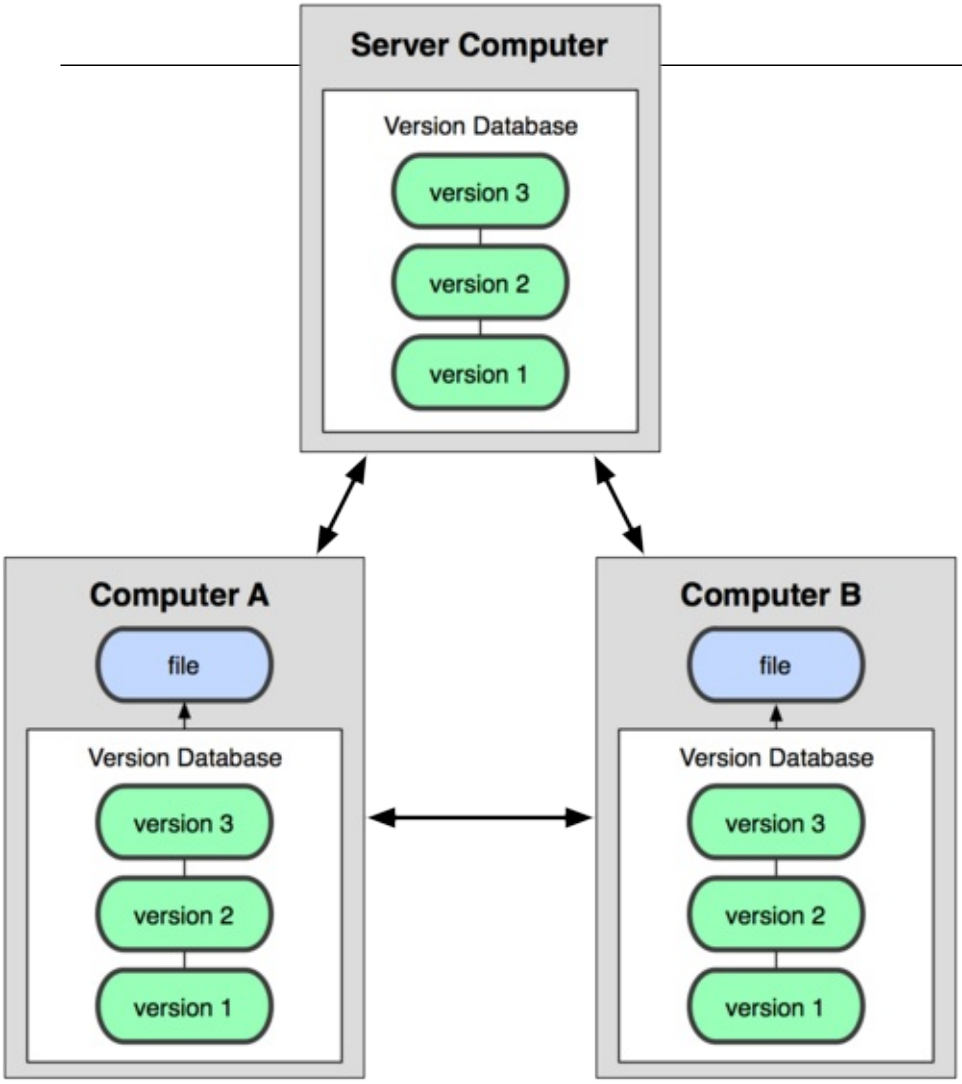


This setup offers many advantages, especially over local VCSs. For example, everyone knows to
certain degree what everyone else on the project is doing. Administrators have fine-grained contr
over who can do what; and it's far easier to administer a CVCS than it is to deal with local databas
on every client.

However, this setup also has some serious downsides. The most obvious is the single point of failu
that the centralized server represents. If that server goes down for an hour, then during that ho
nobody can collaborate at all or save versioned changes to anything they're working on. If the ha
disk the central database is on becomes corrupted, and proper backups haven't been kept, you lo
absolutely everything-the entire history of the project except whatever single snapshots people happ
to have on their local machines. Local VCS systems suffer from this same problem-whenever yo
have the entire history of the project in a single place, you risk losing everything.

Distributed Version Control Systems

This is where Distributed Version Control Systems (DVCSs) step in. In a DVCS (such as Gi
Mercurial, Bazaar or Darcs), clients don't just check out the latest snapshot of the files: they ful
mirror the repository. Thus if any server dies, and these systems were collaborating via it, any of th
client repositories can be copied back up to the server to restore it. Every checkout is really a fu
backup of all the data (see Figure 1-3).



Furthermore, many of these systems deal pretty well with having several remote repositories they ca
work with, so you can collaborate with different groups of people in different ways simultaneous
within the same project. This allows you to set up several types of workflows that aren't possible
centralized systems, such as hierarchical models.

A Short History of Git

As with many great things in life, Git began with a bit of creative destruction and fiery controvers
The Linux kernel is an open source software project of fairly large scope. For most of the lifetime
the Linux kernel maintenance (1991-2002), changes to the software were passed around as patches a
archived files. In 2002, the Linux kernel project began using a proprietary DVCS system call
BitKeeper.

In 2005, the relationship between the community that developed the Linux kernel and the commerci
company that developed BitKeeper broke down, and the tool's free-of-charge status was revoked. Th
prompted the Linux development community (and in particular Linus Torvalds, the creator of Linu
to develop their own tool based on some of the lessons they learned while using BitKeeper. Some
the goals of the new system were as follows:

- Speed
- Simple design
- Strong support for non-linear development (thousands of parallel branches)
- Fully distributed
- Able to handle large projects like the Linux kernel efficiently (speed and data size)

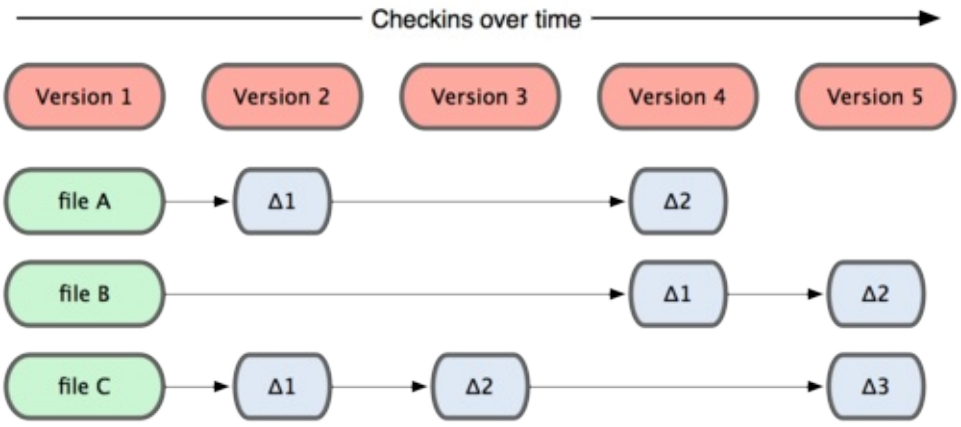
Since its birth in 2005, Git has evolved and matured to be easy to use and yet retain these initi
qualities. It's incredibly fast, it's very efficient with large projects, and it has an incredible branchi
system for non-linear development (See Chapter 3).

Git Basics

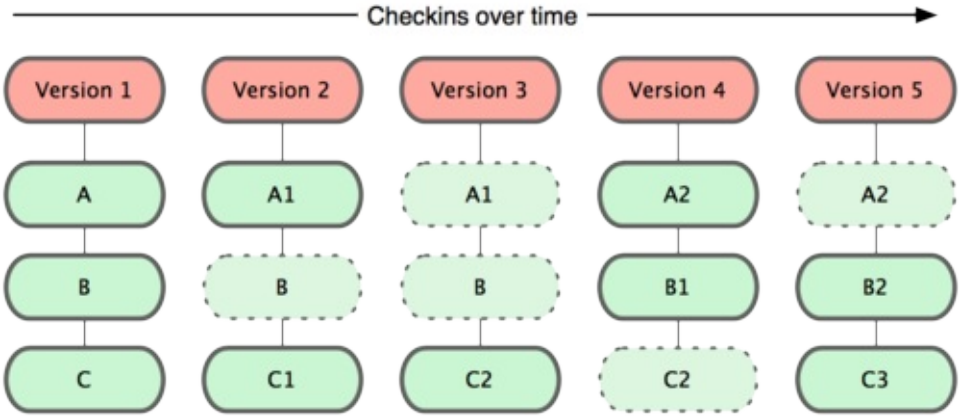
So, what is Git in a nutshell? This is an important section to absorb, because if you understand wh Git is and the fundamentals of how it works, then using Git effectively will probably be much easi for you. As you learn Git, try to clear your mind of the things you may know about other VCSs, suc as Subversion and Perforce; doing so will help you avoid subtle confusion when using the tool. G stores and thinks about information much differently than these other systems, even though the us interface is fairly similar; understanding those differences will help prevent you from becomi confused while using it.

Snapshots, Not Differences

The major difference between Git and any other VCS (Subversion and friends included) is the way G thinks about its data. Conceptually, most other systems store information as a list of file-base changes. These systems (CVS, Subversion, Perforce, Bazaar, and so on) think of the information the keep as a set of files and the changes made to each file over time, as illustrated in Figure 1-4.



Git doesn't think of or store its data this way. Instead, Git thinks of its data more like a set snapshots of a mini filesystem. Every time you commit, or save the state of your project in Git, basically takes a picture of what all your files look like at that moment and stores a reference to th snapshot. To be efficient, if files have not changed, Git doesn't store the file again-just a link to th previous identical file it has already stored. Git thinks about its data more like Figure 1-5.



This is an important distinction between Git and nearly all other VCSs. It makes Git reconsider almo every aspect of version control that most other systems copied from the previous generation. Th

makes Git more like a mini filesystem with some incredibly powerful tools built on top of it, rath
than simply a VCS. We'll explore some of the benefits you gain by thinking of your data this wa
when we cover Git branching in Chapter 3.

Nearly Every Operation Is Local

Most operations in Git only need local files and resources to operate - generally no information
needed from another computer on your network. If you're used to a CVCS where most operations hav
that network latency overhead, this aspect of Git will make you think that the gods of speed hav
blessed Git with unworldly powers. Because you have the entire history of the project right there o
your local disk, most operations seem almost instantaneous.

For example, to browse the history of the project, Git doesn't need to go out to the server to get th
history and display it for you-it simply reads it directly from your local database. This means you se
the project history almost instantly. If you want to see the changes introduced between the curre
version of a file and the file a month ago, Git can look up the file a month ago and do a loc
difference calculation, instead of having to either ask a remote server to do it or pull an older versio
of the file from the remote server to do it locally.

This also means that there is very little you can't do if you're offline or off VPN. If you get on a
airplane or a train and want to do a little work, you can commit happily until you get to a netwo
connection to upload. If you go home and can't get your VPN client working properly, you can sti
work. In many other systems, doing so is either impossible or painful. In Perforce, for example, yo
can't do much when you aren't connected to the server; and in Subversion and CVS, you can edit file
but you can't commit changes to your database (because your database is offline). This may not see
like a huge deal, but you may be surprised what a big difference it can make.

Git Has Integrity

Everything in Git is check-summed before it is stored and is then referred to by that checksum. Th
means it's impossible to change the contents of any file or directory without Git knowing about
This functionality is built into Git at the lowest levels and is integral to its philosophy. You can't los
information in transit or get file corruption without Git being able to detect it.

The mechanism that Git uses for this checksumming is called a SHA-1 hash. This is a 40-charact
string composed of hexadecimal characters (0-9 and a-f) and calculated based on the contents of a fi
or directory structure in Git. A SHA-1 hash looks something like this:

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

You will see these hash values all over the place in Git because it uses them so much. In fact, G
stores everything not by file name but in the Git database addressable by the hash value of i
contents.

Git Generally Only Adds Data

When you do actions in Git, nearly all of them only add data to the Git database. It is very difficult

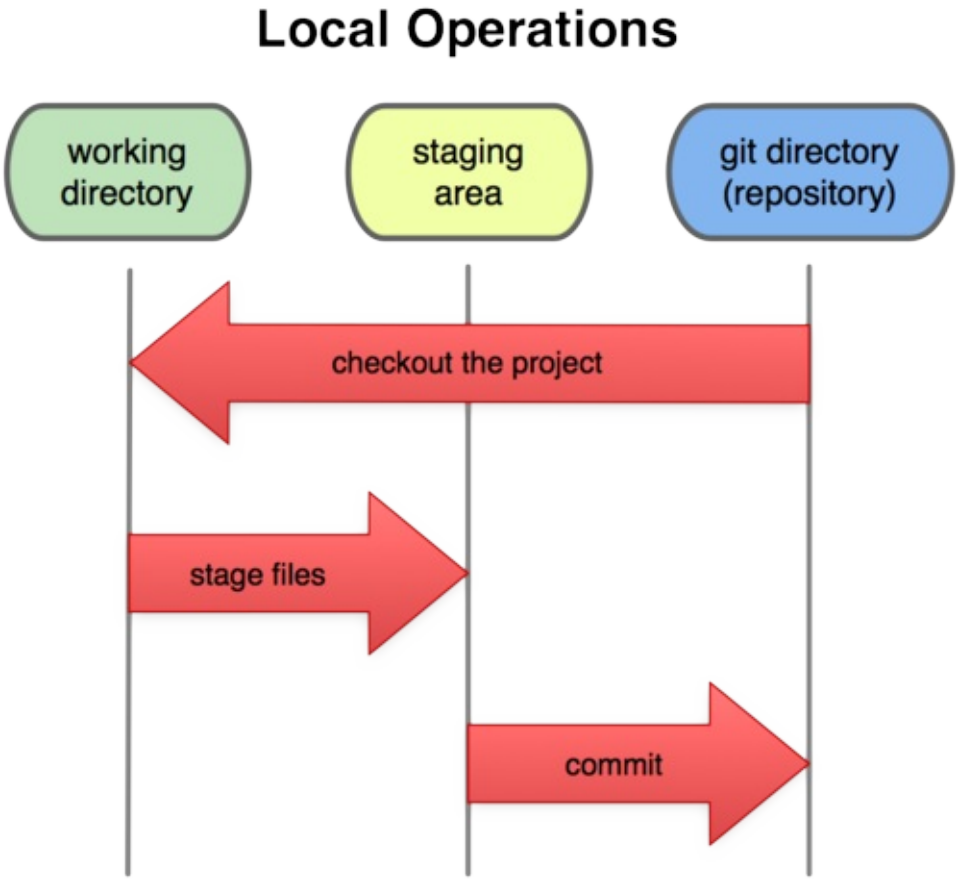
get the system to do anything that is not undoable or to make it erase data in any way. As in any VC
you can lose or mess up changes you haven't committed yet; but after you commit a snapshot into G
it is very difficult to lose, especially if you regularly push your database to another repository.

This makes using Git a joy because we know we can experiment without the danger of severe
screwing things up. For a more in-depth look at how Git stores its data and how you can recover da
that seems lost, see "Under the Covers" in Chapter 9.

The Three States

Now, pay attention. This is the main thing to remember about Git if you want the rest of your learnin
process to go smoothly. Git has three main states that your files can reside in: committed, modifie
and staged. Committed means that the data is safely stored in your local database. Modified mea
that you have changed the file but have not committed it to your database yet. Staged means that yo
have marked a modified file in its current version to go into your next commit snapshot.

This leads us to the three main sections of a Git project: the Git directory, the working directory, ar
the staging area.



The Git directory is where Git stores the metadata and object database for your project. This is th
most important part of Git, and it is what is copied when you clone a repository from anoth
computer.

The working directory is a single checkout of one version of the project. These files are pulled out
the compressed database in the Git directory and placed on disk for you to use or modify.

The staging area is a simple file, generally contained in your Git directory, that stores informatio

about what will go into your next commit. It's sometimes referred to as the index, but it's becomi
standard to refer to it as the staging area.

The basic Git workflow goes something like this:

1. You modify files in your working directory.
2. You stage the files, adding snapshots of them to your staging area.
3. You do a commit, which takes the files as they are in the staging area and stores that snapsh
   permanently to your Git directory.

If a particular version of a file is in the git directory, it's considered committed. If it's modified b
has been added to the staging area, it is staged. And if it was changed since it was checked out but h
not been staged, it is modified. In Chapter 2, you'll learn more about these states and how you c
either take advantage of them or skip the staged part entirely.

Installing Git

Let's get into using some Git. First things first-you have to install it. You can get it a number of way
the two major ones are to install it from source or to install an existing package for your platform.

Installing from Source

If you can, it's generally useful to install Git from source, because you'll get the most recent versio
Each version of Git tends to include useful UI enhancements, so getting the latest version is often t
best route if you feel comfortable compiling software from source. It is also the case that many Linu
distributions contain very old packages; so unless you're on a very up-to-date distro or are usi
backports, installing from source may be the best bet.

To install Git, you need to have the following libraries that Git depends on: curl, zlib, openssl, expa
and libiconv. For example, if you're on a system that has yum (such as Fedora) or apt-get (such as
Debian based system), you can use one of these commands to install all of the dependencies:

```
$ yum install curl-devel expat-devel gettext-devel \
  openssl-devel zlib-devel

$ apt-get install libcurl4-gnutls-dev libexpat1-dev gettext \
  libz-dev
```

When you have all the necessary dependencies, you can go ahead and grab the latest snapshot from t
Git web site:

```
http://git-scm.com/download
```

Then, compile and install:

```
$ tar -zxf git-1.6.0.5.tar.gz
$ cd git-1.6.0.5
$ make prefix=/usr/local all
$ sudo make prefix=/usr/local install
```

After this is done, you can also get Git via Git itself for updates:

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
```

Installing on Linux

If you want to install Git on Linux via a binary installer, you can generally do so through the bas
package-management tool that comes with your distribution. If you're on Fedora, you can use yum:

```
$ yum install git-core
```

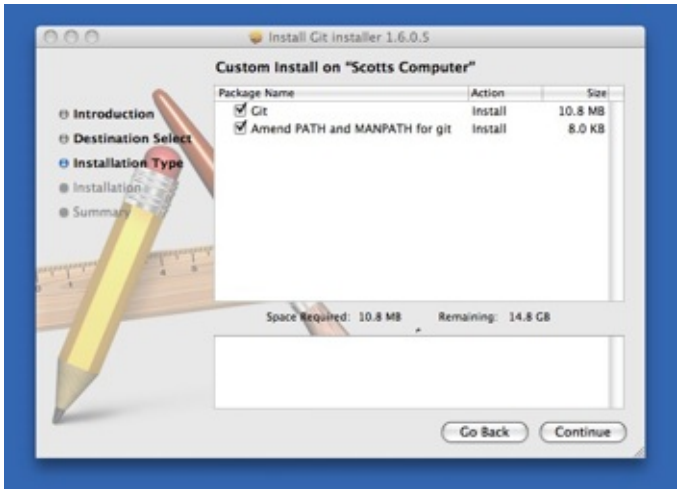
Or if you're on a Debian-based distribution like Ubuntu, try apt-get:

```
$ apt-get install git-core
```

Installing on Mac

There are two easy ways to install Git on a Mac. The easiest is to use the graphical Git installer, whic you can download from the Google Code page (see Figure 1-7):

```
http://code.google.com/p/git-osx-installer
```



The other major way is to install Git via MacPorts (`http://www.macports.org`). If you hav MacPorts installed, install Git via

```
$ sudo port install git-core +svn +doc +bash_completion +gitweb
```

You don't have to add all the extras, but you'll probably want to include +svn in case you ever have t use Git with Subversion repositories (see Chapter 8).

Installing on Windows

Installing Git on Windows is very easy. The msysGit project has one of the easier installatio procedures. Simply download the installer exe file from the Google Code page, and run it:

```
http://code.google.com/p/msysgit
```

After it's installed, you have both a command-line version (including an SSH client that will come handy later) and the standard GUI.

First-Time Git Setup

Now that you have Git on your system, you'll want to do a few things to customize your G environment. You should have to do these things only once; they'll stick around between upgrade You can also change them at any time by running through the commands again.

Git comes with a tool called git config that lets you get and set configuration variables that control a aspects of how Git looks and operates. These variables can be stored in three different places:

- `/etc/gitconfig` file: Contains values for every user on the system and all their repositories. you pass the option `--system` to `git config`, it reads and writes from this file specifically.
- `~/.gitconfig` file: Specific to your user. You can make Git read and write to this fil specifically by passing the `--global` option.
- config file in the git directory (that is, `.git/config`) of whatever repository you're current using: Specific to that single repository. Each level overrides values in the previous level, s values in `.git/config` trump those in `/etc/gitconfig`.

On Windows systems, Git looks for the `.gitconfig` file in the `$HOME` directory (`C:\Documents an Settings\$USER` for most people). It also still looks for /etc/gitconfig, although it's relative to th MSys root, which is wherever you decide to install Git on your Windows system when you run th installer.

Your Identity

The first thing you should do when you install Git is to set your user name and e-mail address. This important because every Git commit uses this information, and it's immutably baked into the commi you pass around:

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

Again, you need to do this only once if you pass the `--global` option, because then Git will alway use that information for anything you do on that system. If you want to override this with a differe name or e-mail address for specific projects, you can run the command without the `--global` optio when you're in that project.

Your Editor

Now that your identity is set up, you can configure the default text editor that will be used when G needs you to type in a message. By default, Git uses your system's default editor, which is general Vi or Vim. If you want to use a different text editor, such as Emacs, you can do the following:

```
$ git config --global core.editor emacs
```

Your Diff Tool

Another useful option you may want to configure is the default diff tool to use to resolve merg

conflicts. Say you want to use vimdiff:

```
$ git config --global merge.tool vimdiff
```

Git accepts kdiff3, tkdiff, meld, xxdiff, emerge, vimdiff, gvimdiff, ecmerge, and opendiff as val merge tools. You can also set up a custom tool; see Chapter 7 for more information about doing that.

Checking Your Settings

If you want to check your settings, you can use the `git config --list` command to list all th settings Git can find at that point:

```
$ git config --list
user.name=Scott Chacon
user.email=schacon@gmail.com
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
...
```

You may see keys more than once, because Git reads the same key from different file (`/etc/gitconfig` and `~/.gitconfig`, for example). In this case, Git uses the last value for eac unique key it sees.

You can also check what Git thinks a specific key's value is by typing `git config {key}`:

```
$ git config user.name
Scott Chacon
```

Getting Help

If you ever need help while using Git, there are three ways to get the manual page (manpage) help f
any of the Git commands:

```
$ git help <verb>
$ git <verb> --help
$ man git-<verb>
```

For example, you can get the manpage help for the config command by running

```
$ git help config
```

These commands are nice because you can access them anywhere, even offline. If the manpages ar
this book aren't enough and you need in-person help, you can try the `#git` or `#github` channel on t
Freenode IRC server (irc.freenode.net). These channels are regularly filled with hundreds of peop
who are all very knowledgeable about Git and are often willing to help.

Summary

You should have a basic understanding of what Git is and how it's different from the CVCS you ma
have been using. You should also now have a working version of Git on your system that's set up wit
your personal identity. It's now time to learn some Git basics.

Git Basics

If you can read only one chapter to get going with Git, this is it. This chapter covers every bas command you need to do the vast majority of the things you'll eventually spend your time doing wi Git. By the end of the chapter, you should be able to configure and initialize a repository, begin ar stop tracking files, and stage and commit changes. We'll also show you how to set up Git to igno certain files and file patterns, how to undo mistakes quickly and easily, how to browse the history your project and view changes between commits, and how to push and pull from remote repositories.

Getting a Git Repository

You can get a Git project using two main approaches. The first takes an existing project or directo and imports it into Git. The second clones an existing Git repository from another server.

Initializing a Repository in an Existing Directory

If you're starting to track an existing project in Git, you need to go to the project's directory and type

```
$ git init
```

This creates a new subdirectory named .git that contains all of your necessary repository files - a G repository skeleton. At this point, nothing in your project is tracked yet. (See Chapter 9 for mo information about exactly what files are contained in the `.git` directory you just created.)

If you want to start version-controlling existing files (as opposed to an empty directory), you shou probably begin tracking those files and do an initial commit. You can accomplish that with a few g add commands that specify the files you want to track, followed by a commit:

```
$ git add *.c
$ git add README
$ git commit -m 'initial project version'
```

We'll go over what these commands do in just a minute. At this point, you have a Git repository wit tracked files and an initial commit.

Cloning an Existing Repository

If you want to get a copy of an existing Git repository - for example, a project you'd like to contribu to - the command you need is git clone. If you're familiar with other VCS systems such as Subversio you'll notice that the command is clone and not checkout. This is an important distinction - G receives a copy of nearly all data that the server has. Every version of every file for the history of th project is pulled down when you run `git clone`. In fact, if your server disk gets corrupted, you ca use any of the clones on any client to set the server back to the state it was in when it was cloned (yo may lose some server-side hooks and such, but all the versioned data would be there-see Chapter 4 f more details).

You clone a repository with `git clone [url]`. For example, if you want to clone the Ruby Git libra called Grit, you can do so like this:

```
$ git clone git://github.com/schacon/grit.git
```

That creates a directory named "grit", initializes a `.git` directory inside it, pulls down all the data f that repository, and checks out a working copy of the latest version. If you go into the new `gri` directory, you'll see the project files in there, ready to be worked on or used. If you want to clone th repository into a directory named something other than grit, you can specify that as the ne command-line option:

```
$ git clone git://github.com/schacon/grit.git mygrit
```

That command does the same thing as the previous one, but the target directory is called mygrit.

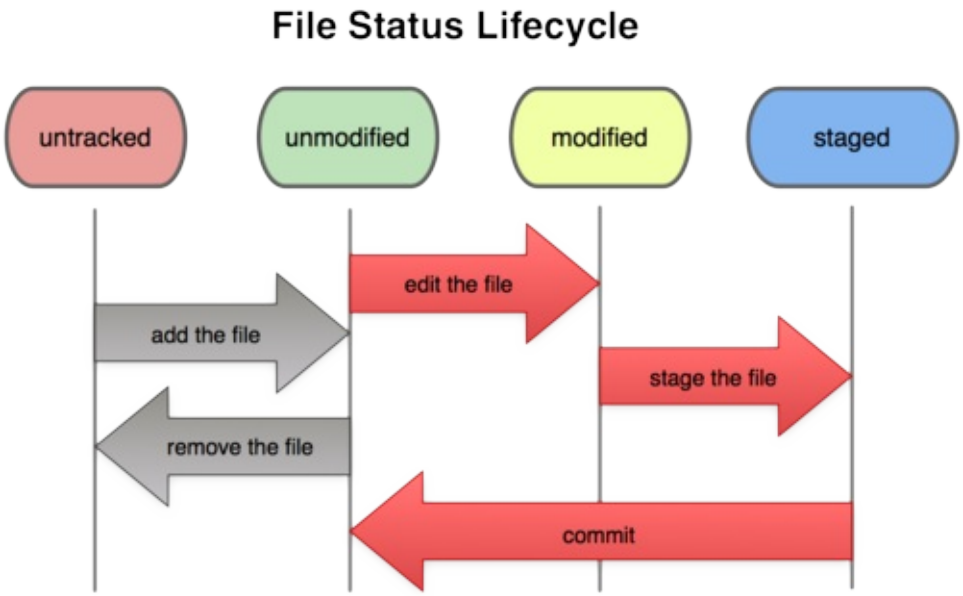
Git has a number of different transfer protocols you can use. The previous example uses the `git:/` protocol, but you may also see `http(s)://` or `user@server:/path.git`, which uses the SSH transf protocol. Chapter 4 will introduce all of the available options the server can set up to access your G repository and the pros and cons of each.

Recording Changes to the Repository

You have a bona fide Git repository and a checkout or working copy of the files for that project. Yo need to make some changes and commit snapshots of those changes into your repository each time th project reaches a state you want to record.

Remember that each file in your working directory can be in one of two states: tracked or untracke Tracked files are files that were in the last snapshot; they can be unmodified, modified, or stage Untracked files are everything else - any files in your working directory that were not in your la snapshot and are not in your staging area. When you first clone a repository, all of your files will b tracked and unmodified because you just checked them out and haven't edited anything.

As you edit files, Git sees them as modified, because you've changed them since your last comm You stage these modified files and then commit all your staged changes, and the cycle repeats. Th lifecycle is illustrated in Figure 2-1.



Checking the Status of Your Files

The main tool you use to determine which files are in which state is the git status command. If you ru this command directly after a clone, you should see something like this:

```
$ git status
# On branch master
nothing to commit (working directory clean)
```

This means you have a clean working directory-in other words, there are no tracked and modifie files. Git also doesn't see any untracked files, or they would be listed here. Finally, the command tel you which branch you're on. For now, that is always master, which is the default; you won't wor about it here. The next chapter will go over branches and references in detail.

Let's say you add a new file to your project, a simple README file. If the file didn't exist before, a you run `git status`, you see your untracked file like so:

```
$ vim README
$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#   README
nothing added to commit but untracked files present (use "git add" to track)
```

You can see that your new README file is untracked, because it's under the "Untracked files heading in your status output. Untracked basically means that Git sees a file you didn't have in t previous snapshot (commit); Git won't start including it in your commit snapshots until you explicit tell it to do so. It does this so you don't accidentally begin including generated binary files or oth files that you did not mean to include. You do want to start including README, so let's start trackin the file.

Tracking New Files

In order to begin tracking a new file, you use the command `git add`. To begin tracking the READM file, you can run this:

```
$ git add README
```

If you run your status command again, you can see that your README file is now tracked and staged

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   new file:   README
#
```

You can tell that it's staged because it's under the "Changes to be committed" heading. If you comm at this point, the version of the file at the time you ran git add is what will be in the historic snapshot. You may recall that when you ran git init earlier, you then ran git add (files) - that was t begin tracking files in your directory. The git add command takes a path name for either a file or directory; if it's a directory, the command adds all the files in that directory recursively.

Staging Modified Files

Let's change a file that was already tracked. If you change a previously tracked file calle `benchmarks.rb` and then run your `status` command again, you get something that looks like this:

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   new file:   README
#
# Changed but not updated:
```

```
#   (use "git add <file>..." to update what will be committed)
#   ------------------------------------------------------------------------
#   modified:   benchmarks.rb
#
```

The benchmarks.rb file appears under a section named "Changed but not updated" - which means th
a file that is tracked has been modified in the working directory but not yet staged. To stage it, you ru
the `git add` command (it's a multipurpose command - you use it to begin tracking new files, to stag
files, and to do other things like marking merge-conflicted files as resolved). Let's run `git add` no
to stage the benchmarks.rb file, and then run `git status` again:

```
$ git add benchmarks.rb
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   new file:   README
#   modified:   benchmarks.rb
#
```

Both files are staged and will go into your next commit. At this point, suppose you remember on
little change that you want to make in benchmarks.rb before you commit it. You open it again an
make that change, and you're ready to commit. However, let's run `git status` one more time:

```
$ vim benchmarks.rb
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   new file:   README
#   modified:   benchmarks.rb
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#   modified:   benchmarks.rb
#
```

What the heck? Now benchmarks.rb is listed as both staged and unstaged. How is that possible?
turns out that Git stages a file exactly as it is when you run the git add command. If you commit no
the version of benchmarks.rb as it was when you last ran the git add command is how it will go in
the commit, not the version of the file as it looks in your working directory when you run git comm
If you modify a file after you run `git add`, you have to run `git add` again to stage the latest versio
of the file:

```
$ git add benchmarks.rb
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   new file:   README
```

```
#   modified:   benchmarks.rb
#  ____________________________________________________________________
```

Ignoring Files

Often, you'll have a class of files that you don't want Git to automatically add or even show you being untracked. These are generally automatically generated files such as log files or files produce by your build system. In such cases, you can create a file listing patterns to match them name .gitignore. Here is an example .gitignore file:

```
$ cat .gitignore
*.[oa]
*~
```

The first line tells Git to ignore any files ending in .o or .a - object and archive files that may be th product of building your code. The second line tells Git to ignore all files that end with a tilde (~ which is used by many text editors such as Emacs to mark temporary files. You may also include log, tmp, or pid directory; automatically generated documentation; and so on. Setting up a .gitigno file before you get going is generally a good idea so you don't accidentally commit files that yo really don't want in your Git repository.

The rules for the patterns you can put in the .gitignore file are as follows:

- Blank lines or lines starting with # are ignored.
- Standard glob patterns work.
- You can end patterns with a forward slash (`/`) to specify a directory.
- You can negate a pattern by starting it with an exclamation point (`!`).

Glob patterns are like simplified regular expressions that shells use. An asterisk (`*`) matches zero more characters; `[abc]` matches any character inside the brackets (in this case a, b, or c); a questio mark (`?`) matches a single character; and brackets enclosing characters separated by a hyphen(`[0-9` matches any character between them (in this case 0 through 9) .

Here is another example .gitignore file:

```
# a comment - this is ignored
*.a       # no .a files
!lib.a    # but do track lib.a, even though you're ignoring .a files above
/TODO     # only ignore the root TODO file, not subdir/TODO
build/    # ignore all files in the build/ directory
doc/*.txt # ignore doc/notes.txt, but not doc/server/arch.txt
```

Viewing Your Staged and Unstaged Changes

If the `git status` command is too vague for you - you want to know exactly what you changed, n just which files were changed - you can use the `git diff` command. We'll cover `git diff` in mo detail later; but you'll probably use it most often to answer these two questions: What have yo changed but not yet staged? And what have you staged that you are about to commit? Although `gi status` answers those questions very generally, `git diff` shows you the exact lines added an

removed - the patch, as it were.

Let's say you edit and stage the README file again and then edit the benchmarks.rb file witho staging it. If you run your `status` command, you once again see something like this:

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   new file:   README
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#   modified:   benchmarks.rb
#
```

To see what you've changed but not yet staged, type `git diff` with no other arguments:

```
$ git diff
diff --git a/benchmarks.rb b/benchmarks.rb
index 3cb747f..da65585 100644
--- a/benchmarks.rb
+++ b/benchmarks.rb
@@ -36,6 +36,10 @@ def main
           @commit.parents[0].parents[0].parents[0]
         end

+        run_code(x, 'commits 1') do
+          git.commits.size
+        end
+
         run_code(x, 'commits 2') do
           log = git.commits('master', 15)
           log.size
```

That command compares what is in your working directory with what is in your staging area. Th result tells you the changes you've made that you haven't yet staged.

If you want to see what you've staged that will go into your next commit, you can use `git diff --cached`. (In Git versions 1.6.1 and later, you can also use `git diff --staged`, which may be easier remember.) This command compares your staged changes to your last commit:

```
$ git diff --cached
diff --git a/README b/README
new file mode 100644
index 0000000..03902a1
--- /dev/null
+++ b/README2
@@ -0,0 +1,5 @@
+grit
+ by Tom Preston-Werner, Chris Wanstrath
+ http://github.com/mojombo/grit
+
```