Revision Control

Keeping Track of Changes

- As you work on a project changes accumulate; as you fix bugs, add new features and optimize, your code base will diverge dramatically from where you began
- Sometimes you add a feature or fix a bug and think everything is working perfectly, but then such wistful thinking meets reality when a user (perhaps even yourself) employs your application in a way you did not foresee
- Or you hit what was thought to be an impossibly rare race condition when you port to a new platform with faster hardware and multiple CPUs
- At such a point you may have progressed considerably past the point where the problem was introduced, and you will have to locate the point in your development where things went wrong, fix the problem, and bring all the other changes forward to the current state

Manual Revision Control Methods

- Every developer has been sloppy about this at some point, particularly when they are the only one working on the project and code base
- While there is no substitute for a clean and intelligent design that has a lot of modular components, a design which is easy to break down and reassemble once pieces have been fixed, there are tools which can be used to greatly facilitate efficient cycles of development
- The simplest method is to simply keep regular backup snapshots of the work, and revert to them as problems arise - we have all done this
- But it is a manual process, and as the size of the project grows it becomes more and more unwieldy

Tightening and Tracking Control Better

- You will wind up differencing your current code base with earlier ones; why not just store the history of changes, and the differences instead of complete snapshots? (if nothing else, you will save disk space)
- More importantly you can track the changes directly instead of inferring them from the endpoint and starting point
- As other developers are added to the project the manual method also becomes difficult to manage:
    - One developer has to remain the master who handles all actual changes
    - Other developers cannot commit changes directly unless you want to confuse the result
    - Difficult to manage simultaneous contributions

Distributed Development

- Massively distributed projects (such as the Linux kernel) can involve thousands of contributors, and even the notion of a central master repository of all wisdom can become a hindrance
- In order to organize updates and facilitate cooperation, many different schemes are available for source control
- Standard features of such programs include the ability to keep an accurate history, or log of changes, be able to back up to earlier releases, coordinate possibly conflicting updates from more than one developer, etc.

Graphical Interfaces

- There are a number of graphical interfaces which can be used to view or maintain **git** repositories, some of the most important include:
  - **git-gui**: to use this you merely have to launch it from within the main repository directory, either by typing **`git gui`** or **`git-gui`**; it enables you to browse histories, make changes, commits, work with remote repositories, compare branches, etc.
  - **gitk**: this repository browser can actually be invoked from within git-gui and is a somewhat older interface more dedicated to looking at project history
  - **cgit**: can be installed in concert with a web server to enable very efficient repository browsing
  - **gitweb**: an older interface than cgit that works in a similar fashion

Note - git.kernel.org is a good site to see the Linux Kernel revision control and other things of importance. Do look into.

Some Basic Help while using Git

Any git installation brings with it a lot of documentation on your local machine.

Just typing:

```
1 $ git help
```

gives you the basic commands. Detailed help about any one of them is obtained with:

```
1 $ git help command
```

You can get help on particular commands in either of the two ways:

```
$ git help commit
$ man git-commit
```

the result being a conventional man page.

Two great sources of documentation are:

- The official Git User's Manual. There are many other useful documents, either residing at the same website, or linked to from there. This document is terse but authoritative, and extremely useful.
- Leoliger, J. (2009). ***Version Control with Git***. Sebastopol, CA: O'Reilly Media, Inc. and its second edition Leoliger, J. and McCullough, M. (2012). ***Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development***. Sebastopol, CA: O'Reilly Media, Inc. Out of several books using git, they are the most up-to-date, and the most thorough. There are many topics in this course that receive only cursory treatment; these books go into great detail, with many examples, bringing to the fore real life problems and experiences.

The best overall source of documentation can be found at the git home page, which contains a wealth of documentation and examples.

In addition, you can download the *Git Started with Community Software* presentation, presented at the Chicago Embedded Systems Conference in 2010, by Jerry Cooperstein or watch a a free on-line webinar by James Bottomley, *Introduction to Git*.

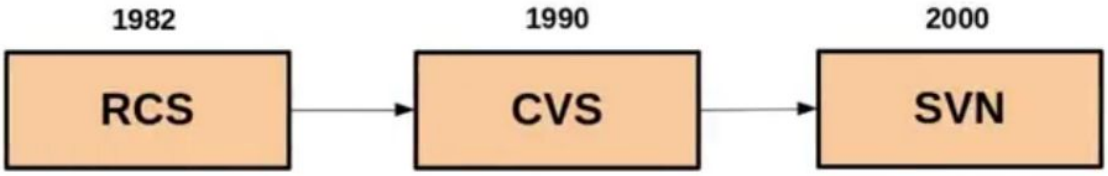
Note - As per the concerned linux distribution installed - check whether git is installed or not. Install appropriately using instructions for that particular distro. (Do some google.)
Try not installing from Source if you are a beginner as there are chances of missing out some steps - prefer Binary Installation.

Revision Control Systems

Converting Between Different Systems

- It is possible to bring projects which use other revision control systems into the git universe
- While it is always possible to simply taking the working copy of the project files, ignore the revision control information, and create a new repository, this throws away the previous history of changes, and the possibility of reverting to an earlier state
- There are tools for importing projects into git; indeed we will do exercises showing how to do this for CVS and Subversion
- It is also possible to go in the opposite direction, to export a git project to another system
- You might do this because you want to switch your revision system to one of the other choices; more likely you would do this because you need to mirror your git repository so those who have access only to one of the older systems can synchronize with it

Major Revision Control System History



Revision Control System (RCS)

- An old and widely used source code system is provided by **RCS**
- It is easy to convert projects using the earlier SCCS system to using RCS
- **make** has built-in rules for RCS which makes it easy to use in development projects
- **emacs** can be used tightly with RCS using the vc mode and related functions, e.g. vc-ediff
- Conventionally a sub-directory named RCS (or rcs) is situated under the directory containing the files under control; this is the repository
- RCS forces developers to work in a serial fashion:
  - When a file is unlocked no one can commit changes to a file, but the lock is available to anyone with access to the RCS repository
  - When a file is locked, only the holder of the lock can commit changes to the file
- As we shall see, more modern systems permit better cooperative development

Concurrent Version System (CVS)

- CVS is more powerful than RCS and does a more thorough job of letting multiple developers work on the same source concurrently
- However, it has a steeper learning curve, as it has many more commands and possible environment customizations
- All files are stored in a **centralized repository**
- One never works directly with the files in the repository; instead you get your own copies into your working directory, and when you are finished with changes you check (or commit) them back into the repository
- Repositories can be on the local machine or on a remote platform anywhere in the world

- The repository location may be specified either by setting the **CVSROOT** environment variable, or with the **-d** option; i.e. the two following lines are equivalent:

```
$ cvs -d /home/coop/mycvs init
$ export CVSROOT=/home/coop/mycvs ; cvs init
```

- A number of users can make changes to a given module at once
- There are a number of commands to help with resolving differences and merging work
- As for RCS, emacs can be used tightly with CVS using the vc mode and related functions

Subversion

- Subversion is an open source project designed to be the successor to CVS, and which has gradually replaced it in many projects
- In order to keep the transition smooth, the Subversion interface resembled that of its predecessor, and made it easy to migrate CVS repositories to Subversion
- Directories, copies and renames are versioned, not just files and their contents; metadata associated with files (such as permissions) can also be versioned
- Revision numbering is in a per-commit basis, not per-file, and atomicity of commits is complete; unless the entire commit is completed no part of it goes through

- A standalone Subversion server can be set up using a custom protocol, which runs as an inetd service or as a daemon, offering authentication and authorization; one can also use Apache to set up a http-based server
- There are many enhancements that improve efficiency and reduce storage size requirements; in particular, costs are proportional to the size of the change set, not that of the data set
- Binary files can be handled, and there are built in tools for the mirroring of repositories

git

- The Linux kernel development system has special needs in that it is widely distributed throughout the world, with hundreds (even thousands) of developers; it is all done very publicly, under the GPL
- For a long time there was no real source revision control system
- Then major kernel developers went over to the use of **BitKeeper**, which while a commercial project granted a restricted use license for Linux kernel development
- However, in a very public dispute over licensing restrictions in the spring of 2005, the free use of BitKeeper became unavailable for Linux kernel development
- The response was the development of **git**, whose original author was **Linus Torvalds**

- Technically, git is not a source control management system in the usual sense, and the basic units it works with are not files
- It has two important data structures: an **object database** and a **directory cache**
- The object database contains objects of three varieties:
  - **Blobs**: chunks of binary data containing file contents
  - **Trees**: sets of blobs including file names and attributes, giving the directory structure
  - **Commits**: changesets describing tree snapshots
- The directory cache captures the state of the directory tree
- By moving away from a file by file based system, one is better able to handle changesets which involve many files

git and Distributed Development

- Many projects have developers working separately, united by a common version control system; this is possible even when there is a central authoritative repository, such as with CVS
- What makes git different is that on a technical level, there is no such thing as central authoritative repository; the underlying framework is actually peer-to-peer in nature
- The importance and central role played by one particular location is political and sociological, not technical
- Distributed development is not defined by various parts of a project being worked on separately, with a partition between different host repositories
- In git, distributed development means every repository is authoritative, and contains not just one part of the project, but the entire code base

- git has no politics, and has no preferred model for organization
- The hierarchy of the development community can be very centralized and top down, or it can be very flat
- The exchange of information and changes can be pyramidal, with a number of development trees coalescing into a top level one, or it can be very egalitarian
- git is a tool, not a rigid method, and it can be used in any way that fits the needs and philosophy of a project

Converting CVS Repository to Git

Make sure you have the appropriate CVS software installed to do the following steps. In addition to the basic CVS package, you probably need **git-cvs** and **cvsps**.

The easiest way to get a copy, or clone, of a CVS project is to use **git cvsimport**.

To obtain a copy or part of the CVS repository itself (which one can always expect to be obtainable through CVS), you can do:

```
$ git cvsimport -v -C my_cvs_repo \
      -d:pserver:anonymous@cvs.savannah.nongnu.org:/sources/cvs diffutils
```

where we have chosen only the **diffutils** module of CVS in order to keep things small.

This creates a git repository under the **my_cvs_repo** directory, which you can examine. Notice all the git information goes in the **.git** subdirectory.

If you want to compare with the CVS repository, you can bring that down with:

```
$ cvs -d:pserver:anonymous@cvs.savannah.nongnu.org:/sources/cvs co diffutils
```

where the repository information will go under **diffutils/CVS**.

Comparing the results:

```
diff -r my_cvs_repo diffutils/
Only in diffutils/: CVS
Only in my_cvs_repo: .git
```

Converting Subversion Repository to Git

Make sure you have the appropriate subversion software installed to do the following steps. In addition to the basic subversion package, you need **subversion-perl**. There may be other useful packages which are obtainable from the EPEL repository on RHEL-based systems.

The easiest way to get a copy, or clone, of a Subversion project is to use **git svn**.

To obtain a copy or part of the Subversion repository itself, you can do:

```
$ git svn clone  \
      https://svn.apache.org/repos/asf/subversion/trunk/doc my_svn_repo
```

where we have chosen only the **doc** module of Subversion in order to keep things small.

However, this can take a long time as it gathers the entire history of the project and sometimes hangs. It is easier for learning purposes to just get the most recent version (which for git we would call a shallow clone). There is no easy way to just say give me the last version with Subversion; we need an actual release number which we can get with:

```
$ svn log  https://svn.apache.org/repos/asf/subversion/trunk/doc | head
------------------------------------------------------------------------
r1663949 | brane | 2015-03-04 05:55:13 -0600 (Wed, 04 Mar 2015) | 1 line

* doc/svn-square.jpg: Copy the logo used by Doxygen from the site tree.
------------------------------------------------------------------------
r1663948 | brane | 2015-03-04 05:53:09 -0600 (Wed, 04 Mar 2015) | 1 line
```

We can then pick just the latest release for the shallow clone:

```
$ git svn clone -r1663949  https://svn.apache.org/repos/asf/subversion/trunk/doc
  my_svn_repo
Initialized empty Git repository in /tmp/SVN/my_svn_repo/.git/
        A       doxygen.conf
        A       svn-square.jpg
        A       programmer/gtest-guide.txt
        A       programmer/WritingChangeLogs.txt
        A       README
        A       user/svn-best-practices.html
        A       user/lj_article.txt
        A       user/cvs-crossover-guide.html
r1663949 = f2f1312fe65123c1be0935421d05cc862d0d008e (refs/remotes/git-svn)
Checked out HEAD:
  https://svn.apache.org/repos/asf/subversion/trunk/doc r1663949
```

This creates a git repository under the **my_svn_repo** directory, which you can examine. Notice all the git information goes in the **.git** subdirectory.

If you want to compare with the original Subversion repository, you can bring that down with:

```
$ svn checkout https://svn.apache.org/repos/asf/subversion/trunk/doc doc
....
```

where the repository information will go under the **.svn** directories.

Comparing:

```
$ diff -qr my_svn_repo/ doc/
Only in my_svn_repo/: .git
Only in doc/: .svn
```

Basic Commands

You can see the version of git you have installed with:

```
$ git --version

git version 1.5.5.6
```

Detailed help information in the form of a man page can be obtained about any subcommand by doing:

```
$ git help [subcommand]
```

For example, the two following statements produce the same result:

```
$ git help status
$ man git-status
```

which furthermore points out that there are two ways to invoke particular commands, e.g. these two commands are equivalent:

```
$ git status
$ git-status
```

The second hyphenated form is historical and is now disfavored, so use the **git subcommand** form instead.

You can get a basic list of git commands by just typing **git**, which will give you the list showed in the following screenshot:

There are only a few global options that apply, those prefixed with -- in the above listing. Many of the subcommands have their own options, which are included in **[ARGS]** in the above.

If you cannot resist seeing the more complete set of commands, do:

```
1 $ git help --all
```

An Example

Let's get a feel for how git works and how easy it easy to use. For now, we will just make our own local project.

First, we create a working directory and then initialize git to work with it:

```
1 $ mkdir git-test
2 $ cd git-test
3 $ git init
```

We can see the current status of our project with:

```
$ git status

# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file: somejunkfile
#
```

Notice it is telling us that our file is staged, but not yet committed.

Let's tell git who is responsible for this repository:

```
$ git config user.name "Another Genius"
$ git config user.email "b_genius@linux.com"
```

This must be done for each new project, unless you have it predefined in a global configuration file.

Now, let's modify the file, and then see the history of differences:

```
$ echo another line >> somejunkfile
$ git diff
diff --git a/somejunkfile b/somejunkfile
index 9638122..6023331 100644
--- a/somejunkfile
+++ b/somejunkfile
@@ -1 +1,2 @@
some junk
+another line
```

To actually commit the changes to the repository, we do:

```
$ git add somejunkfile
$  git commit -m "My initial commit"

Created initial commit eafad66: My initial commit
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 somejunkfile
```

If you do not specify an identifying message to accompany the commit with the **-m** option, you will jump into an editor to put some content in. You must do this or the commit will be rejected. The editor chosen will be what is set in your **EDITOR** environment variable, which can be superseded with setting **GIT_EDITOR**.

You can see your history with:

```
$ git log

commit eafad66304ebbcd6acfe69843d246de3d8f6b9cc
Author: A Genius <a_genius@linux.com>
Date:   Wed Dec 30 11:07:19 2009 -0600

    My initial commit
```

and you can see the information you got in there. You will note the long hexadecimal string, which is the commit number; it is a 160-bit, 40-digit unique identifier which we will discuss later. git cares about these beasts, not file names.

You are now free to modify the already exiting file and add new files with **git add**. But they are staged until you do another **git commit**.

Now, that was not so bad. But we have only scratched the surface.

Concepts and Design Features

Concepts

- Many basic git commands resemble those of other version control systems
- Operations like adding files, committing changes, differencing with earlier versions, and logging the history are common to any serious source control system
- However, underneath the hood git is quite different than many of its antecedents
- For example, in git a file is not an essential object
- Common operations which may be somewhat cumbersome in other systems, such as renaming a file, are remarkably simple when using git
- The long hexadecimal numbers associated with commits are also somewhat strange; they are more than identifiers and they incorporate checksums that are computed from the contents of the repositories and the changes that are made

Design Features

- git grew out of the Linux kernel development community and was designed to meet its particular needs
- As time goes on, it has been adopted by additional projects, but its basic structure and motivation shows its roots
- Here we list the design features, many of which could not be found in pre-existing version control systems, which inspired the development of git:
  - **Facilitate distributed development**: developers had to be able to work in parallel without constant re-synchronization and without needing to constantly contact a central repository
  - **Scale to handle large numbers of developers**: the Linux kernel community has literally thousands of developers; this had to be handled both efficiently and reliably
  - **Attain high speed and maximal efficiency**: it is important to avoid copying unnecessary information, use compression, not bottle up networks, and be able to handle varying network latencies
  - **Maintain strong integrity and trust**: security demands that no unauthorized alterations can be inserted, and that repositories are authentic, not impostors; cryptographic hash functions are used for this purpose
  - **Keep everyone accountable**: all changes must be documented and ascribed to whomever did them; there is always a trail left behind that can be displayed
  - **Keep immutable data in the repository**: information, such as the history of the project, can not be changed
  - **Make atomic transactions**: when changes are made they all must go through, or none do; this avoids leaving the repository in an uncertain or corrupted state
  - **Support branching and merging**: git supports parallel branches of development, and has very robust methods for merging branches when it is time
  - **Make each repository independent**: each repository has all the history in it; there is never a need to consult a central repository
  - **Operate under a free unencumbered license**: git is covered by the GPL version 2

Git Architecture

Repositories

- The **repository** is a database
- It contains every bit of information required to store a project, manage revisions, and display its history
- The git repository contains not just the complete working copy of all the files that comprise the project's contents; it also contains a copy of the repository itself
- Each repository also contains a set of configuration parameters, such as the author's name and email address
- If you clone a git repository (make another copy of it for someone else, or even yourself) this configuration information is not carried forth; such information is instead maintained on a per-site, per-user, per-repository basis
- There are two important data structures that are maintained within a repository:
  - The **object store** contains the guts of the project, a set of discrete binary objects
  - The **index** is a binary file that changes dynamically as the project morphs; it contains a picture of the overall project structure at any given time

Objects

- Git places four types of objects in its **object store**
- All together, they contain everything known about the project and how to construct both its current state and its previous history:
  - **Blobs** (**B**inary **L**arge **Ob**jects): this is an opaque construct that contains a version of a file's content; it does not contain the file's name or any other metadata, just the content
  - **Trees**: these record blob identifiers, pathnames, file metadata, etc., for files in a directory; they can also refer to subdirectories and objects within them
  - **Commits**: every time a change is made to the repository a commit object is created containing the metadata that fully describes the change; each commit points to a tree object that gives a complete snapshot of the project; commits have parents and children; except for the initial (or root) commit which is parent-less
  - **Tags**: these assign human friendly names to those horrendously long hexadecimal numbers used internally by git

Content vs. Pathnames

- Unlike most other version control systems, git tracks content, not files
- It does not store information based on the file or directory names, or directory structure and file layout
- This is associated information, not directly stored in the blobs
- For instance, suppose you have two files in two directories with different names
- git stores only one binary blob associated with the content
- The content is associated with a unique hexadecimal string
- If either file is changed, its identifier changes, and now git associates a new blob with the new content
- Comparisons are very fast because git need only compare the 160-bit identifiers, rather than actually compare the blobs, much less the actual files

Content vs. Pathnames (Cont.)

- As files change new, uniquely identified, binary blobs are created for each version
- Other version control systems often keep a copy of a file at some point in time together with revision histories and have to play back (or forward) to reconstruct a given version of the file
- git can do all this more quickly because it always has knowledge of the complete content of any version of a file; in other words, in git's strange way of doing things, a file's history can be computed by tracking changes in content, or blobs, not files
- Filenames are treated as metadata distinct from file contents, and stored in tree objects and indexes
- The contents of the **`.git`** subdirectory tree look nothing at all like the working directories they represent
- This is totally different than the way older revision systems like RCS or CVS are set up

Contents v/s Pathnames

Let's see what this means in practice.

git commits tend to be all the changes for logical grouping (a feature, a bug fix, etc) including changes in all necessary files. For example, with RCS (Revision Control System) you might do:

```
$ rcs co src_slct.c
$ rcs co src_slct.dlg
$ rcs co src_slct.hlp
$ rcs co product manual.doc
```

to check out the files to revise. Then, you would edit them to get the bug fixed or feature implemented. Finally, you check them back in with:

```
$ rcs ci srv_slct.c
$ rcs ci srv_slct.dlg
$ rcs ci srv_slct.hlp
$ rcs ci product manual.doc
```

Tags must be used in RCS or other similar version control systems to group these changes together. If someone is in the middle of the check-in process and a system build begins, the build may fail. Backing out changes require finding all the files with the same tag. If someone mis-tags one of the updates, backing out the changes is nearly impossible.

With git, by comparison, you edit the files to get the bug fixed or feature implemented and then, you do:

```
$ git add srv_slct.c
$ git add srv_slct.dlg
$ git add srv_slct.hlp
$ git add product_manual.doc
$ git commit
```

The git commit step contains the new versions of all four files. Backing out the changes if a problem was discovered only involves removing the commit.

Committing v/s Publishing

Committing vs. Publishing

- The steps of committing and publishing are quite distinct in using git, as compared with other revision control systems
- A **commit** is a local process; it merely means saving the current state of your working project files in your local repository, thus setting up a convenient marker in the development history
  - Whether you commit often with small changes, or infrequently with large change sets is your decision
  - The commit operation requires no network access; you are free to reorganize your commits in your working repository before you make them public; you can pick the points to publish with care, so they are logical and well-defined
- **Publishing** means sharing your changes by making them public
  - This can be done either by a making a push or letting others pull, or by use of patches; this effectively freezes the repository history

Upstream v/s Downstream

Upstream vs. Downstream

- The parent repository is often considered **upstream** and the repository you are working on, at one point cloned from the parent, is often considered **downstream**
- Once again there is nothing in git itself which makes this distinction, akin to a server/client relationship, since git has a fundamentally peer-to-peer architecture
- Conceptually, any repository to which you send changes is considered upstream; any repository which is based on yours is considered downstream
- There can be multiple levels involved if one has sub-trees, or feeders
- One repository can have both downstream and upstream relationships; for instance, it might be the focus of a particular subsystem and the entire project repository can be upstream, while individual sub-projects can be downstream

Forking

Forking

- A project **forks** when someone takes the entire project and goes off in another direction; this is sometimes called branching
- In git this term has a different semantic meaning as there can be multiple branches within a given repository
- Technically, every time one clones a repository with git, one creates a fork, but these are not meant to be permanent bifurcations
- There are a number of reasons a project may fork: disputes among developers about the project direction or licensing issues, or political and personal conflicts
- What makes git so useful is that its robust merging capabilities make healing a fork quite simple
- Every time a branch is merged into the main repository the fork is healed; even forks which are quite old can often be merged back in with grace and ease

File Categories

File Categories

As far as git is concerned, files in your project directories fall into three categories:

- Tracked files
- Ignored files
- Untracked files

Tracked Files

- **Tracked files** are those that are already in the repository in their current working state, or have been staged; i.e. changed but not yet committed

Ignored Files

- **Ignored files** are invisible to git
- Each directory may contain a file named **`.gitignore`** which lists either specific files or name patterns which are to be ignored
- For instance many temporary files could be ignored with a specification like **`*.o`**
- The **`.gitignore`** file in any directory applies recursively to all subdirectories below it, so if you specify a filename or pattern in the main directory, all files fitting the description will be invisible

- It is also possible to override the rules by prefixing with **`!`**; for example, if your **`.gitignore`** file includes:

```
*.ko
!my_driver.ko
```

- The second specification overrides the first more general one, and the file **`my_driver.ko`** will be tracked

Untracked Files

- **Untracked files** are anything that does not fit in the previous two classifications
- All files in your current working directories are examined and anything that is not tracked or ignored is considered untracked
- This may be files that have not been added yet, they may be temporary files, etc., but if you commit the entire directory, they will become tracked files

Basic File Commands

git add

Adds one or more files as blobs to the object store and adds references to the new blobs in the index, by **sha1** hash. (Remember, the index contains a complete picture of the state of a project at a given time.) Thus, we are staging one or more files and directories. It can be used either to add new files or stage changed ones. The files will not be committed until there is a **git commit** done.

There are a number of options (see **git help add**). For instance, you can use **-i** for interactive choosing of files to stage, or **-u** to update only files already known to git. You can also specify wildcard patterns or directory names (in which the entire subdirectory tree is added).

It is worth repeating that until you do a **git commit** the changes are only staged; the repository is not updated.

git rm

Removes a file from the working tree and the index. This can be pretty dangerous if you do not realize what you are doing. If you only want to remove the file from the working directory, and not the repository index, just use the normal **rm** command.

While you remove files from the repository, you do not remove them from the history, as that would be dishonest. If you want to remove a file that has been staged but not committed, you have to add the **-cache** option, as in:

```
$ git add myfile
$ git rm myfile --cached
```

git mv

Renames a file and stages the new filename in the repository. It is equivalent to renaming the working file, and then doing a **git rm** on the old file name and a **git add** on the new one; i.e. the following operations are equivalent:

```
$ git mv oldfile newfile
$ mv oldfile newfile ; git rm oldfile ; git add newfile
```

This is an easier operation in git than it is in older revision control systems, where renaming a file means actually deleting the old one and adding a new one. In this case, the binary blobs associated with the file remain the same, only the index is updated.

This is an easier operation in git than it is in older revision control systems, where renaming a file means actually deleting the old one and adding a new one. In this case, the binary blobs associated with the file remain the same, only the index is updated.

Here is a table that shows how all three of these stages work (changes induced by basic **git** file commands):

Command	Source Files	Index	Commit Chain	References
git add	Unchanged	Updated with new file	Unchanged	Unchanged
git rm	File removed	File removed	Unchanged	Unchanged
git mv	File moved/renamed	Updates file name/location	Unchanged	Unchanged

git ls-files

Shows information about files in the index and working tree. By default, this command shows only files in the repository. If you want to show the untracked files, you can do:

```
$ git ls-files --others --exclude-standard
```

where the **--others** option shows the untracked files, and the **--exclude-standard** option says to ignore standard exclusions such as **.gitignore** files.

Exercise

- First, initialize the repository, configuring it with name and email, etc. Then, add a couple of files to the project and commit them.
- Remove one of the files with **git rm** and with **git diff** see the difference with the repository.
- Rename the remaining file with **git mv** and with **git diff** see the difference with the repository, once again.
- Commit again and look at the history with **git log**. Then, do **git ls-files** without any arguments.
- Add two new files, make one of them ignored by git and modify the original remaining file. Do **git ls-files** again.
- Now, try some different options to **git ls-files**, such as **-t** and **-o**. Do **man git ls-files** to see the various options available and try some others.
- Now, add the new files that are not ignored with **git add**, commit once again, and do **git ls-files** with some options to see the results. You may want to do **git log** again as well.

Making a commitment

Commit Process

- The commit process is familiar from other version control systems
- However, the way git handles this process is very particular
- When a commit is made, a commit object is created from the files in the index, and the commit object is placed in the object store
- The files themselves are in the object store already when they are placed in the index
- Any new files since the last commit result in new blobs and any new directories result in new trees; any unchanged object is simply re-used
- Thus minimal new storage is required

- Furthermore this process is very fast because blobs do not have to be compared directly; all git has to do is compare their hexadecimal identifiers to see if they are identical
- In particular if the hash describing a directory has not changed, nothing in any of its subdirectories has changed either
- The commits are connected in an ancestral tree
- You should pick well-defined points, but whether you do many small commits or fewer larger ones is your choice
- Note however, that git does handle large numbers of small commits very efficiently, and when one uses the bisection tool, it can speed up finding points where bugs and regressions were introduced

Making the commitment can be done in a number of ways. Let's say you have modified a number of files (i.e. they are staged), but you only want to commit the changes to one file:

```
$ git commit file1
```

If you want to commit all changes,, any of these forms will do it:

```
$ git commit
$ git commit ./
$ git commit -a
```

Here is a table that shows how the commit step works (changes induced by **git commit**):

Command	Source Files	Index	Commit Chain	References
git commit	Unchanged	Unchanged	A new commit object is created from the index and added to the top of the commit chain	**HEAD** in the current branch points to new commit object

Note that the command

```
$ git diff
```

will show all differences between your staged working directories and what has been previously committed. After you do the commit, it will show nothing differing.

Identifiers and Tags

Every time you do a commit, git assigns a unique 160-bit 40-character hexadecimal hash value to it. While you can refer to those commits with these values, it is obviously unwieldy. For instance, taking an example from the Linux kernel source repository, we can see the history of commits with:

```
$ git log | grep "^commit" | head -10
commit 56b24d1bbcff213dc9e1625eea5b8e13bb50feb8
commit 5a45a5a881aeb82ce31dd1886afe04146965df23
commit ecade114250555720aa8a6232b09a2ce2e47ea99
commit 2c4ea6e28dbf15ab93632c5c189f3948366b8885
commit 106e4da60209b508894956b6adf4688f84c1766d
commit 4b050f22b5c68fab3f96641249a364ebfe354493
commit 84c37c168c0e49a412d7021cda3183a72adac0d0
commit 0acf611997d9d05dbfb559c3c6e379c861eb5957
commit 434fd6353b4c83938029ca6ea7dfa4fc82d602bd
commit 852980808617299fe713ed3e03114058883ce3d8a
```

If you want to refer to or revert to a certain commit, typing in such a long string is obviously a pain. You can instead create and use a tag. Thus, you could do:

```
$ git tag ver_10 08d869aa8683703c4a60fdc574dd0809f9b073cd
```

or, even better:

```
$ git tag ver_10 08d869
```

If you have used a long enough part of the 40-character string in the second example, so that the reference is unique, then the short version will suffice.

git tag creates a tag or annotated tag (a text string that references a commit object). The tag is placed in **.git/refs/tags** unless it is an annotated tag, in which case, the tag is created as an object in the object store (changes induced by **git tag**):

Command	Source Files	Index	Commit Chain	References
git tag	Unchanged	Unchanged	Unchanged	A new tag is created

We will talk about checking things out later, but all you would have to do to revert to the development point labeled by **ver_10** would be:

```
$ git checkout ver_10
```

Reviewing Commit History

It is easy to display the history of commits with git using the command **git log**. For example, consider the following script which sets up a repository and then adds some files, modifies them and introduces four commits along the way:

```
#!/bin/bash

rm -rf git-test
mkdir git-test

cd git-test
git init

git config user.name "A Smart Guy"
git config user.email "asmartguy@linux.com"

echo file1 > file1
git add file1
git commit file1 -m "This is the first commit"

echo file2 > file2
git add file2
git commit . -m "This is the second commit"

echo file3 > file3
echo another line for file3 >> file3
git add .
git commit . -m "This is the third commit"

echo another line for file2 >> file2
git add .
git commit -a  -m "This is the fourth commit"
```

If we then ask to see the log, we see:

```
$ git log

commit 4b4bf2c5aa95b6746f56f9dfce0e4ec6bddad407
Author: A Smart Guy <asmartguy@linux.com>
Date:   Thu Dec 31 13:50:15 2009 -0600

    This is the fourth commit

commit 55eceacc9ab2b4fc1c806b26e79eca4429d8b52a
Author: A Smart Guy <asmartguy@linux.com>
Date:   Thu Dec 31 13:50:15 2009 -0600

    This is the third commit

commit f60c0c21764676beca75b7edc2f5f5e51b5dd404
Author: A Smart Guy <asmartguy@linux.com>
Date:   Thu Dec 31 13:50:15 2009 -0600

    This is the second commit

commit 712cbafa7ee0aaef03861b049ddc7865220b4e2c
Author: A Smart Guy <asmartguy@linux.com>
Date:   Thu Dec 31 13:50:15 2009 -0600

    This is the first commit
```

The commits are shown in reverse order of introduction. Even shorter, you can do:

```
$ git log --pretty=oneline

4b4bf2c5aa95b6746f56f9dfce0e4ec6bddad407 This is the fourth commit
55eceacc9ab2b4fc1c806b26e79eca4429d8b52a This is the third commit
f60c0c21764676beca75b7edc2f5f5e51b5dd404 This is the second commit
712cbafa7ee0aaef03861b049ddc7865220b4e2c This is the first commit
```

You can also see the actual patches made with the **-p** option and can view only part of the history by specifying a particular commit, as in:

```
$ git log -p f60c

commit f60c0c21764676beca75b7edc2f5f5e51b5dd404
Author: A Smart Guy <asmartguy@linux.com>
Date:   Thu Dec 31 13:50:15 2009 -0600

    This is the second commit

diff --git a/file2 b/file2
new file mode 100644
index 0000000..6c493ff
--- /dev/null
+++ b/file2
@@ -0,0 +1 @@
+file2

commit 712cbafa7ee0aaef03861b049ddc7865220b4e2c
Author: A Smart Guy <asmartguy@linux.com>
Date:   Thu Dec 31 13:50:15 2009 -0600

    This is the first commit

diff --git a/file1 b/file1
new file mode 100644
index 0000000..e212970
--- /dev/null
+++ b/file1
@@ -0,0 +1 @@
+file1
```

Reverting and Resetting Commits

From time to time, you may realize that you have committed changes unwisely. Either you may have made changes you never should have, or you committed prematurely.

You can back out a particular commit with:

```
$ git revert commit_name
```

where **commit_name** can be specified in a number of ways and need not be the most recent.

Commits can be delineated with:

- **HEAD** : most recent commit
- **HEAD~** : previous commit (the parent of **HEAD**)
- HEAD~~ or
- HEAD~2 : the grandparent commit of **HEAD**
- **{hash number}** : specific commit by full or partial **sha1** hash number
- **{tag name}** : a name for a commit.

Note that **git revert** puts in a new commit set of changes, i.e. a reversed patch as an additional commit. This is the appropriate thing to do if someone else has downloaded a tree containing the changes that have been reversed. This will change your working copy of source files.

In short, **git revert** builds and adds a new commit object, sets **HEAD** to it and updates the working directory (changes induced by **git revert**):

Command	Source Files	Index	Commit Chain	References
git revert	Changed to reflect reversion	Uncommitted changes discarded	New commit created; no actual commits removed	**HEAD** of current branch points to new commit

In the case where you are the only one that has seen the updated repository, the **git reset** command is more appropriate. For example, if you want to pull out the last two commits, you can do:

```
1  $ git reset HEAD~2
```

This will ***not*** change your working copy of source files; it just reverses the commits and makes the index match the specified commit (changes induced by **git reset**):

Command	Source Files	Index	Commit Chain	References
git reset	Unchanged	Discard uncommitted changes	Unchanged	Unchanged (unless form as used below; then **HEAD** of current branch moves to a prior commit)

If using any one of the options **--soft**, **--mixed**, **--hard**, **--merge** or **--keep**, the behavior is different. In this case, the **HEAD** reference in the current branch is also set to the specified commit, optionally modifying the index and the source files to match:

- **--soft**: just moves the current branch to prior commit object (index unchanged in this case)
- **--mixed** (default): also updates the index to match new head (un-stages everything)
- **--hard**: same as **--mixed**, but also updates working directory to match new head (un-edits your files).

Suppose you have been furiously coding and realize that your last three commits are still a work in progress, but everything before that should be made available to others. Then, you should create a new working branch while resetting the master branch back three commits, as in:

```
1  $ git branch work
2  $ git reset --hard HEAD~3
3  $ git checkout work
```

which restores the master branch to its previous state, while leaving the speculative work in work where you will continue to play. We will discuss branches in detail later.

Tidying Repositories

As your project grows through a series of commits, your repository may grow large in size. You can optimize and compact your repository by issuing **git gc**, as in:

```
$ du -shc .git
47M     .
47M     total

$ git gc
Counting objects: 8, done.
Compressing objects: 100% (8/8), done.
Writing objects: 100% (8/8), done.
Total 8 (delta 2), reused 0 (delta 0)

$ du -shc .git
29M     .git
29M     total
```

where **gc** stands for garbage collection.

You can also check your repository for certain kinds of errors with the command **git fsck**. The most likely and harmless errors it will find will be dangling objects; while these are sometimes useful for recovering corrupted repositories, they can generally be safely removed with **git prune** as in:

```
$ git prune -n
$ git prune
```

where the first command just checks to see what would be done, and if you are happy with it, you can issue the second pruning command.

How to blame

It is possible to assign blame for whom is responsible for a given set of lines in a file. For example, using **file2** from the previous example:

```
$ git blame file2

f60c0c21 (A Smart Guy 2009-12-31 13:50:15 -0600 1) file2
4b4bf2c5 (A Smart Guy 2009-12-31 13:50:15 -0600 2) another line for file2
```

shows the responsible commit and author. It is possible to specify a range of lines and other parameters for the search. For example, for a more complicated file in the Linux kernel source (each line had to be broken because of width limitations here):

```
c7:/usr/src/linux>git blame -L 3107,3121 kernel/sched/core.c
e220d2dc kernel/sched.c      (Peter Zijlstra      2009-05-23 18:28:55 +0200 3107
  )
e418e1c2 kernel/sched.c      (Christoph Lameter   2006-12-10 02:20:23 -0800 3108
  ) #ifdef CONFIG_SMP
6eb57e0d kernel/sched.c      (Suresh Siddha       2011-10-03 15:09:01 -0700 3109
  )       rq->idle_balance = idle_cpu(cpu);
7caff66f kernel/sched/core.c (Daniel Lezcano      2014-01-06 12:34:38 +0100 3110
  )       trigger_load_balance(rq);
e418e1c2 kernel/sched.c      (Christoph Lameter   2006-12-10 02:20:23 -0800 3111
  ) #endif
265f22a9 kernel/sched/core.c (Frederic Weisbecker 2013-05-03 03:39:05 +0200 3112
  )       rq_last_tick_reset(rq);
^1da177e kernel/sched.c      (Linus Torvalds      2005-04-16 15:20:36 -0700 3113
  ) }
^1da177e kernel/sched.c      (Linus Torvalds      2005-04-16 15:20:36 -0700 3114
  )
265f22a9 kernel/sched/core.c (Frederic Weisbecker 2013-05-03 03:39:05 +0200 3115
  ) #ifdef CONFIG_NO_HZ_FULL
265f22a9 kernel/sched/core.c (Frederic Weisbecker 2013-05-03 03:39:05 +0200 3116
  ) /**
265f22a9 kernel/sched/core.c (Frederic Weisbecker 2013-05-03 03:39:05 +0200 3117
  )  * scheduler_tick_max_deferment
265f22a9 kernel/sched/core.c (Frederic Weisbecker 2013-05-03 03:39:05 +0200 3118
  )  *
265f22a9 kernel/sched/core.c (Frederic Weisbecker 2013-05-03 03:39:05 +0200 3119
  )  * Keep at least one tick per second when
265f22a9 kernel/sched/core.c (Frederic Weisbecker 2013-05-03 03:39:05 +0200 3120
  )  * active task is running because the sch
265f22a9 kernel/sched/core.c (Frederic Weisbecker 2013-05-03 03:39:05 +0200 3121
  )  * yet completely support full dynticks
```

Bisecting (good topic)

Suppose you had a version of your code that worked, and now you are many versions (and commits) later and you have found out that it is no longer working.

git has the ability to bisect in order to rapidly find the change set that screwed things up. The number of steps is no more than the logarithm to the base 2 of the number of commits, which is much faster than a brute force approach. In other words, if a bad change has been done somewhere in the last 1024 commits, you can find it in no more than 10 bisection steps.

You do this by first typing:

```
$ git bisect start
$ git bisect bad
$ git bisect good V_10
```

where it is assumed that the current commit is bad and version V_10 is known to be good. git will then leave you at a commit halfway in between. You then test the code to see if the bug is still there. If it is, you type:

```
$ git bisect bad
```

If the code does not have the bug yet, you type:

```
$ git bisect good
```

You continue this iteratively until you find the bug. Then you type:

```
$ git bisect reset
```

to get back to your current working state.

For a working example, lets try the Linux kernel repository:

```
$ git bisect start
$ git bisect bad
$ git bisect good v2.6.30
Bisecting: 16539 revisions left to test after this
[b4f3fda5d475931d596d5cf599a193f42b857594] Staging: hv: coding style cleanup of
                                                    include/HvVpApi.h
$ git bisect bad
Bisecting: 8270 revisions left to test after this
[0de4adfb8c9674fa1572b0ff1371acc94b0be901] Blackfin: fix accidental reset in
                                                     some boot modes
$ git bisect good
Bisecting: 4136 revisions left to test after this
[b9caaabb995c6ff103e2457b9a36930b9699de7c] Merge branch 'master' of
         git://git.kernel.org/pub/scm/linux/kernel/git/holtmann/bluetooth-next-2
           .6
.........

$ git bisect good
Bisecting: 60 revisions left to test after this
[5d48a1c20268871395299672dce5c1989c9c94e4] Staging: hv: check return value
                                                    of device_register()

.........
/usr/src/GIT/work>git bisect bad
Bisecting: 3 revisions left to test after this
[b57a68dcd9030515763133f79c5a4a7c572e45d6] Staging: hv: blkvsc: fix up
  driver_data usage

$ git bisect good
Bisecting: 1 revisions left to test after this
[511bda8fe1607ab2e4b2f3b008b7cfbffc2720b1] Staging: hv: add the Hyper-V virtual
                                                      network driver
$ git bisect good
Bisecting: 0 revisions left to test after this
[621d7fb7597e8cc2e24e6b0ca67118b452675d90] Staging: hv: netvsc: fix up
  driver_data usage

$ git bisect reset
```

If it is possible to construct a script that can test the current version to see if the bug is present, the process becomes even easier.

Suppose you have written a script, **my_script.sh**, that returns 0 if the current version is good, and any value between 1 and 127 if it is bad. Then, after initializing the bisection with a good and bad version, you can simply do:

```
1 $ git bisect run ./myscript.sh
```

and the process will terminate when it locates the bug.

You can replay the bisection history with **git bisect log** or **git bisect visualize**.

If you do small incremental changesets, bugs can be found very quickly with bisection. Commits which have many changes will mean quite a few places may have to be examined even after you identify the last working version and the first faulty one.

Exercise

- First, initialize a repository, configuring it with name and email address, etc.
- Then, make a significant number of commits, say 64. Each commit should add a file. In one of the commits, have the file include the string BAD. We will interpret this as the bug introduction.
- Now, start a **git bisect** procedure. Designating the last commit with:

```
1 $ git bisect bad
```

and the first one with:

```
1 $ git bisect good
```

See how many bisections it takes to find the one that introduced the bug, the file with BAD in it.

- You can do this manually, or you can use the **git bisect run** ... procedure with a script to make it automated.
- When done, check the history of your bisection with **git bisect log**.

Branching

What Is a Branch?

- At some point it may become necessary to move off the main line of development to create an independent **branch**
- For example, when a major release of a product comes out one may want to establish a maintenance release branch and a development branch:
  - The maintenance release branch may be restricted to fixing bugs and serious performance bottlenecks, and plugging security holes
  - Besides including these changes, the development branch would also carry forth incorporation of new features and optimizations, not intended for public release until the next major product version is ready
- A separate branch isolates a particular area of development or work on a very serious bug, without being burdened by the noise introduced by other simultaneous changes
- What makes git so useful is that both branching and inverse merging processes are easy, and the whole infrastructure has been structured with this in mind

- This was inspired by the way the Linux kernel development community evolved
- Many parallel branches exist for things like network drivers, networking and USB
- In the git way of doing things, while it is the collection point for changes and is in some sense the most important branch, in a technical sense it is no different than any other branch (its importance is social, political, and tactical, not structural)
- One branch can track another while proceeding independently; it can be brought up to date with changes in the other branch without having to start over; the differences between the parallel branches can be kept as small as possible as they both advance
- While you can have many branches in a project as part of the same repository, you can only have one active (i.e. current) at a time
- The files in your working directories are those from that branch; if you switch branches, the files will change

Branch Names vs. Tags

- Sometimes people get confused about the difference between **branch names** and **tags**
- In fact it is possible to use the same string for both as they operate in different namespaces, but it does require some care and is not recommended for the non-expert
- A branch name represents a line of development:
  - Usually the main line of development is known as the master branch, as it is called by default
  - As time goes on other branches will be born, given names, and develop in parallel
  - They might have names like **`devel`**, **`debug`**, or **`stable`**
  - While the contents of the branch will change as time goes on and new commits are made, the name of the branch will not normally change, except in the case where another branch is embarked upon

- Tags on the other hand, represent a stage of a particular branch at one point in its history (unless you are asking for a lot of pain you would never change the name of a tag in the future; they should be **immutable**)
- If two branches share a common ancestor they will share tags that predate the separation
- If two parallel branches adopt the same name for a tag after they separate, the merging process will have to deal with this
- But remember the tag can just be thought of as a nickname for one of those long hexadecimal strings which are the fundamental commit identifiers

Branch Creation

The basic command for creating a new branch is:

```
$ git branch [branch_name] [starting_point]
```

If you do not give any arguments, you get a list of branches with the active one starred. A very detailed history of the branches can be obtained with:

```
$ git show-branch
```

If you are creating a new branch, you must give it a name. There are some rules, like no blank spaces in the name, no special or control characters, no slashes at the end, etc. Keep it simple.

A branch is like a tag, but you can add commits to it (changes induced by git branch):

Command	Source Files	Index	Commit Chain	References
git branch	Unchanged	Unchanged	Unchanged	A new branch is created in .git/refs/heads **HEAD** for the new branch points to **HEAD** of the current branch; the current branch is set to the new branch

The starting point is any commit. If there is a tag that describes it, you can use that instead of the long string. If you do not give the argument, you create a copy of the active branch as of its last commit. So, you might do:

```
$ git branch devel
```

to create a new development branch off the mainline.

You can delete the **devel** branch with:

```
$ git branch -d devel
```

which cannot be your current working branch. Recovering an accidentally deleted branch is rather difficult, although not always impossible, so use care.

Branch Checkout

The checkout process lets you switch branches. If you do:

```
$ git checkout devel
```

you have now switched to the development branch in the preceding example, and any files that have been changed will have their contents changed to reflect it. **HEAD** is set to the top commit of the branch.

Note that you have not lost the old branch; all the information to go back to it is still in the repository. All you would have to do is:

```
$ git checkout master
```

and the active branch will be reset and the file contents will revert. It is all blindingly fast too (changes induced by **git checkout**):

Command	Source Files	Index	Commit Chain	References
git checkout	Modified to match commit tree specified by branch or commit ID; un-tracked files not deleted	Unchanged	Unchanged	Current branch reset to that checked out; **HEAD** (in **.git/HEAD**) now refers to last commit in branch

If you have made changes to your working directory that have not yet been committed, switching branches would be a bad move. So, git will refuse to do it and will spit out an error message.

Suppose you do:

```
$ git branch devel
$ echo hello > hello
$ git add hello
$ git commit -a
$ git checkout devel
```

you will see the file **hello** does not exist in the devel branch.

It is also possible to combine the operations of creating a new branch and checking it out, by use of the **-b** option to the **checkout** operation. Doing:

```
$ git checkout -b newbranch startpoint
```

is entirely equivalent to:

```
$ git branch newbranch startpoint
$ git checkout newbranch
```

Getting Earlier File Versions

Suppose you want to see an earlier version of a particular file. You can do this with **git show** if you specify a path name in addition to a tag, as in:

```
$ git show v2.4.1:src/myfile.c
```

Note the colon!

If you actually want to restore that version, you can do it with:

```
$ git checkout v2.4.1 src/myfile.c
```

where there is no colon.

Differencing files

The common UNIX **diff** command is part of the standard toolbox. It can show the difference between any two files, or applied recursively, two complete directory trees.

As a simple example, suppose **file1** contains:

```
This is the
contents
of a simple
file.
```

and **file2** contains:

```
This is the
contents of a slightly
different
file.
```

Simply comparing the files gives:

```
$ diff file1 file2

2,3c2,3
< contents
< of a simple
---
> contents of a slightly
> different
```

However, this is not the most useful form of output. One usually applies the **-u** option, to give what is termed the unified output which is used in **patch** commands:

```
$ diff -u file1 file2

--- file1       2010-01-03 15:48:15.933974603 -0600
+++ file2       2010-01-03 15:48:11.621507573 -0600
@@ -1,4 +1,4 @@
 This is the
-contents
-of a simple
+contents of a slightly
+different
 file.
```

Note the following:

- The **---** notes the first file and **+++** the second file.
- The **@@** line gives the line number context for both files.
- Lines that have been removed in going from **file1** to **file2** are denoted by **-** and lines that have been added are denoted by **+**.
- The output also shows the context of the differences by showing the unmodified lines before and after the patch.

When comparing two directory trees the form used is often:

```
$ diff -Nur directory1 directory2
```

where the **-r** option forces a recursive descent into the trees, and the **-N** option forces files which have been added or deleted to appear in the differencing, instead of just generating a warning that a file is in only one directory tree.

Diffing in Git

Differencing can be done with git in a number of ways. The simple command:

```
$ git diff
```

shows the differences between the current working version of your project and the last commit.

The next form:

```
$ git diff earlier_commit
```

shows the differences between your current working version and the earlier commit specified by **earlier_commit**. This may often be specified as a branch name.

Another form:

```
$ git diff --cached earlier_commit
```

shows the differences between the staged changes in the index and the commit. If you do not specify a commit, it defaults to **HEAD** for the current situation, and the output shows you how the next commit will differ from the current one. In git versions from 1.6.1 on you can say **--staged** instead of **--cached** which might be more intuitive.

The command:

```
$ git diff one_commit another_commit
```

shows the differences between two commits.

There are many other options that control either the nature of the differencing being done, or the form of the output. For example, you can use **--ignore-all-space** to ignore white space differences, or **--stat** or **--numstat** to generate brief statistics.

It is also possible to limit the scope of the **diff** to a particular part of the directory tree. For example, if you are in the Linux kernel git repository, the command:

```
$ git diff v4.2.1 v4.2.2 Documentation/vm
```

will show only the changes in the **Documentation/vm** subdirectory. Similarly, the command:

```
$ git diff --stat v4.2.1 v4.2.2 arch/x86_64
```

will show only brief statistics, rather than detailed differences, for changes to the **arch/x86_64** subdirectory tree.

You can work with one of your repositories created earlier, but it will be richer to work with a full repository, such as the one for the git project itself.

Working in the main project directory, first do:

```
$ git tag
```

to get a list of references. Then, to get a full difference between two versions, you can do something like:

```
$ git diff v1.7.0 v1.7.0-rc2
```

This is likely to be a long output; try with the **--stat** flag to see a short summary of changes.

Now, look at the changes to a particular directory, such as in:

```
$ git diff v1.7.0 v1.7.0-rc2 Documentation
```

or, you can pick to look at one or more particular files.

Merging

Merging

- All but the most primitive of revision control systems have to deal with the problem of merging
- This happens when more than one developer have been working on the project simultaneously, and the need arises to bring their changes back into a unified code base
- If the changes do not conflict with each other directly; i.e. they work on different files or on different parts of the same files, merging the work is not essentially difficult
- For example, if there are two change sets one could simply apply one and then the other; the order should not matter
- The only technical refinement is that after one set of changes there may be a shift of lines where the second set of changes has to work, due to insertions or deletions made by the first one

- Even in this simple case of non-overlapping change sets, more subtle problems can arise of the type that only humans are likely to notice
- Complications can easily be introduced, for example, if two distinct change sets both try to fix the same bug, but do it in different places with different methods
- In such cases using a git methodology of having many small patch sets and commits, combined with the use of tools like bisection, can help to rapid resolution of such subtle problems
- In the case of conflicting change sets, git has been designed from the outset to have strong tools for conflict resolution
- Merging can be seen as the inverse process to branching
- Without an efficient automated process for rejoining, we would have a much weaker revision control system

Merge Commands

Merging the **devel** branch into the current master branch is as simple as doing:

```
$ git checkout master
$ git merge devel
```

where the first step is only necessary if you are not already working on the master branch.

While not absolutely mandatory, the best practice is to tidy up before merging. This means committing any changes that been staged, getting rid of junk, etc. This can help minimize later confusion. You can check your current state in this regard with the command:

```
$ git status
```

If you had no conflicts, the preceding **merge** command will have given a happy report, and your master branch now includes everything that was in the **devel** branch; it has been synced up with the ongoing development. However, let's see what happens if the merge senses a conflict.

To examine this, we create **master** and **devel** branches that are identical except that in the **master** branch we create **file1** which has one line, saying **file901**. In the devel branch this file has the line **file701**. Following the above merge procedure, we get the result:

```
$ git merge devel

Auto-merged file1
CONFLICT (content): Merge conflict in file1
Automatic merge failed; fix conflicts and then commit the result.
```

Listing the files in the working branch with the command:

```
$ git ls-files

file1
file1
file1
file2
file3
```

shows the funny result of **file1** being listed three times! If we look at the contents of this file after the attempted merge, we see the curious result:

```
$ cat file1

<<<<<<< HEAD:file1
file901
=======
file701
>>>>>>> devel:file1
```

a three-way diff showing the problems.

Note that all other changes resulting from the merge have gone through successfully. The merge command will tell you specifically about each problem.

There are two basic approaches you can take to fixing the problems. The first is revert the merge with **git reset**, work on the conflicts in either of the two branches until no conflict is expected to result from a merge, and try again.

If you have made a mess of things with a premature or accidental merge, it is easy to revert back to where you were with:

```
$ git reset --hard master
```

The second approach is to work on the attempted merge file, the third copy that has all the funny markers in it. You can edit it to your heart's content to reflect what should be the product of the merge, and then simply commit, as in:

```
$ git add file1
$ git commit -m "A message for the merge"
$ git ls-files

file1
file2
file3
```

Note the odd appearance of three versions of **file1** is gone and the merge is now complete.

Which approach you take is a judgment call that depends on the size of the merge set, the number of conflicts that develop, etc., and it is totally up to you. The end result should be the same.

Rebasing

Suppose you establish a development branch at some point in time and the master branch that you began from continues to evolve at the same time you are making changes to your development branch.

Eventually, you intend to merge your work with the parallel development branches, but suppose your work is not quite ready for prime time, but you want to get the benefit of the other work going on in the parallel branch.

There are two basic methods that may seem similar, but are actually quite different; merging and rebasing. Merging simply means bringing in the other (probably mainline) evolving repository from time to time. Any conflicts will need to be resolved as usual.

Rebasing is quite different. When you follow this procedure, all your changes since your initial branch off the other line of development are reverted, the branch is brought up to date to its present state, and then your changes are reworked so they fit on the current state.

To give a concrete common example, suppose you are working on a new feature for the Linux kernel, and you began working when the kernel release status was version 4.16 . While you are happily coding away, the mainline kernel continues to evolve and version 4.17 is released. If you do a rebase, your changes, your patch set, are rewritten to apply to the 4.17 code rather than the 4.16 code, a hopefully simpler situation to deal with in the future. In fact, you may rebase further as time goes on.

The steps to a rebase are straightforward. Suppose at some point in the past you did:

```
$ git checkout -b devel origin
```

where **origin** is a remote tracking branch as we will discuss later (you could also use a local branch of course, but that would probably be less realistic). Then, you go about and make a series of changes to project files and make commits in your **devel** branch. Meanwhile, the **origin** branch continues to evolve.

The rebase operation begins with:

```
$ git checkout devel
$ git rebase master devel
```

When you do this, each commit done since the **original** branch point is removed, but is saved in **.git/rebase-apply**. The **devel** branch is then moved forward to the latest version of origin, and then each of the changes is applied to the updated branch.

Obviously, conflicts may emerge and if any are found you will be asked to resolve them as you did when you were merging. As you fix the conflicts, you will have to do **git add** to update the index, and when you are done fixing conflicts, instead of doing a commit, you do:

```
$ git rebase --continue
```

If things get disastrous somewhere along the way, you can revert back to where you were before the attempted rebase with:

```
$ git rebase --abort
```

There are potential problems associated with rebasing, some of which arise from its Orwellian nature. When you do a rebase, you change the history of commits, because the changes are temporarily removed and then all put back in. This leads to at least several problems:

- Presumably, you have been testing your work as you have been progressing it, and the code branch you were testing against came from the earlier branch point. There is no guarantee that just because things worked before when you tested they still will. Indeed, problems that arise may be very subtle.
- You may have done your work in a series of small commits, which is a good practice when trying to locate where problems may have been introduced, for instance when using bisection. But now you have collapsed matters into a larger commit.
- If anyone else has been using your work, has been pulling changes from your tree, you have just pulled the rug out from under their feet. In this case, many developers consider rebasing to be a terrible sin.

If you are doing merging rather than rebasing, other problems can arise, especially if you do it often or not at major stable development points. Once again, history can be confusing in your project. Whatever strategy you choose, think things through and only do merging or rebasing at good, well-defined stages of development to minimize problems.

Changes induced by **git rebase**:

Command	Source Files	Index	Commit Chain	References
git rebase	Unchanged	Unchanged	Parent branch commit moved to a different commit	Unchanged

Working with Distributed Repositories

Working with Distributed Repositories

- Many other revision control systems are built on the notion of a central, authoritative repository which is the hub that individual developers work with
- An essential difference in git's architecture is that no one repository retains such a central role, at least not structurally
- Maintaining separate repositories make sense in at least three situations:
  - A developer is working autonomously
  - Developers are separated across a large (even global) network; a local group of developers may share their own repository to collect changes of immediate import and interest
  - There are divergent projects or sub-areas that are worth developing deeply on their own, with a mind to eventual merging of changes that are found to be beneficial to the main project
- git has a strong infrastructure dedicated to branching and merging multiple repositories, it can handle relatively easily the complicated logistics of distributed development

Operations Involved in Handling Remote Repositories

- The essential operations involved in handling remote repositories are:
  - **Cloning**: establish an initial copy of a remote repository, and place in your own object database; the essential command is **`git clone`**
  - **Pulling**: bring home changes from the remote repository keeping your tracking branch up to date; the essential commands are **`git pull`** and **`git fetch`**
  - **Pushing**: submit your changes to the remote repository; the essential command is **`git push`**
  - **Publishing**: making your repository available for others to clone, and pull from, and perhaps to push to
- git uses tracking branches to handle content in remote repositories
- These are local branches that serve as proxies, or references, for specific branches in remote repositories

Bare Repository

- There is a special kind of repository called a **bare** repository (normally you work in a development repository)
- A bare repository has no working directories and is not used for development and has no checked out branches; it is used only as an authoritative place to clone and fetch from and push to
- It is created with the **`--bare`** option to the clone command

Cloning

Getting an initial clone of a remote repository is as simple as doing:

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

which brings down the entire git repository for git itself. It puts it in a directory called **git**, which contains the usual **.git** subdirectory with all the objects, indexes, etc. that are part of the repository.

This can be a large download, but it is a one time operation.

Note the use of the **git://** protocol in the remote specification. It is the preferred method, but not the only one that can be used. Other possibilities are:

```
file:///path/to/repo.git
ssh://user@remotesite.org[:port]/path/to/repo.git
user@remotesite.org:/path/to/repo.git
http://remotesite.org/path/to/repo.git
https://remotesite.org/path/to/repo.git
rsync://remotesite.org/path/to/repo.git
```

The **git://** method is the fastest and cleanest and should be used whenever it is supported.

If you do a clone of a local repository, git will use hard links where possible to save disk space. If you want or need to prevent this, use the **--no-hardlinks** option to git clone.

You can see the references within either a local or remote repository. (In this context, references are just a list of all branches and tags.) For example, in the local **git** project directory do:

```
$ git show-ref
bd757c18597789d4f01cbd2ffc7c1f55e90cfcd0 refs/heads/master
bd757c18597789d4f01cbd2ffc7c1f55e90cfcd0 refs/remotes/origin/HEAD
6d325dff7434895753dcad82809783644dec75f6 refs/remotes/origin/html
dc89689e86c991c3ebb4d0b6c0cce223ea8e6e47 refs/remotes/origin/maint
....
1250aafa65e7ec62cf776d863ca8c7e4f822928c refs/tags/v1.6.6-rc2
d205d24b8ae17232babad615572bb0265bc029f1 refs/tags/v1.6.6-rc3
09e5ddd756bca67552aad623bab374614ae5e60d refs/tags/v1.6.6-rc4
```

To see what is in the remote repository, do:

```
$ git ls-remote git://git.kernel.org/pub/scm/git/git.git

bd757c18597789d4f01cbd2ffc7c1f55e90cfcd0        HEAD
6d325dff7434895753dcad82809783644dec75f6        refs/heads/html
dc89689e86c991c3ebb4d0b6c0cce223ea8e6e47        refs/heads/maint
8407cc83c60b5e45869c5f64cdcaafee5e9f2f92        refs/heads/man
bd757c18597789d4f01cbd2ffc7c1f55e90cfcd0        refs/heads/master
....
d205d24b8ae17232babad615572bb0265bc029f1        refs/tags/v1.6.6-rc3
94058a90cf3e10122037cd80ea48d3d52be5efd9        refs/tags/v1.6.6-rc3^{}
09e5ddd756bca67552aad623bab374614ae5e60d        refs/tags/v1.6.6-rc4
ab0964d951e4ea88f9ea2cbb88388c1bcd4ae911        refs/tags/v1.6.6-rc4^{}
```

You will notice there are additional references in the remote repository which are not reflected in the clone. If you want to update your repository with changes made at the remote site, you can simply do:

```
$ git pull
```

to synchronize.

Publishing a Project

sSuppose you want to make your project available to others, either on your local machine or on the network, for cloning, pushing, pulling, etc. First, you should create a bare version of your project, as in:

```
$ git clone --bare git-test /tmp/git-test
```

where you see that **/tmp/git-test** now contains a copy of your project's repository, but none of the working files themselves.

A local user can easily make a new cloned copy with the same command without the **--bare** option, as in:

```
$ git clone /tmp/git-test my-git
```

But what about network users?

To make a copy available using the native git protocol, you will need to have the git daemon service installed. To make your bare repository accessible, you have to create an empty file in its main directory or in the **.git** subdirectory:

```
$ touch /tmp/git-test/git-daemon-export-ok
```

You can configure **git daemon** to run automatically by configuring either **xinetd** or **inetd** on your system, and in doing so, you can control behavior rather precisely, but to keep things simple you can just simply run the daemon in background:

```
$ git daemon &
```

Note that you do not have to run this as superuser. Then, a remote user can simply clone your repository with:

```
$ git clone 192.168.1.100:/tmp/git-test my-git
```

This gives them the ability only to clone and fetch, not to push changes back. To give everyone the ability to push changes back, you can do:

```
$ git daemon --enable=receive-pack
```

but this should only be done in a very friendly environment. Additionally, this will be done for all git repositories on the system. To enable write access for just one repository, you have to instead add to the **config** file in the repository the lines:

```
[daemon]
      receivepack = true
```

Note there is no **-** in **receivepack**.

To enable access through the http protocol, you will have to have a proper web server installed and running (probably apache) and know a little about how to configure it. You can place your project directory either under **/var/www/html**, or in the unprivileged place **/home/username/public_html** (in which case, your server has be configured to allow such access).

Before access is available, you have to go the project directory and run the command:

```
$ git --bare update-server-info
```

Accessing through the web server can then be done through these commands:

```
$ git clone https://192.168.1.100/git-test my-git
$ git clone https://192.168.1.100/~username/git-test
```

where you have to substitute the right IP address or domain, and the right username.

Suppose you want to make your project available to someone not using git, or you want to store archived material of your current working tree without including the **.git** repository information directories.

This is easy to do with **git archive** as in:

```
$ git archive --verbose HEAD | bzip2 > myproject.tar.bz2
```

If you want to create an archive corresponding to a particular point in time rather than the latest state, say to tag v1.7.1, you could do:

```
$ git archive --verbose v1.7.1 | bzip2 > myproject_v1.7.1.tar.bz2
```

Fetching, Pulling and Pushing

To bring your repository up to date with the original remote repository, you can merge in changes from the original's master branch with:

```
$ git fetch
$ git merge origin/master
```

It is possible to do this in one step with:

```
$ git pull origin master
```

and, if you have the master branch already checked out, you can simply do:

```
$ git pull
```

which merges from the **HEAD** branch of the origin repository. If you want to specify a particular branch, you can do either of:

```
$ git pull . branch
$ git merge branch
```

The inverse process to pulling is pushing; getting your changes into the remote repository. To publish your revisions, you should first make sure your repository is clean and committed up to date, then you can use any of the accepted protocols, such as:

```
$ git push git://remotesite.org/path/to/repo.git master
```

If you have write access, that will be all that is necessary; if you use ssh protocols, you will be prompted with passwords as you might expect, unless you have configured ssh to not require a password each time.

Note that when you push, it should be to a bare repository. Otherwise, the working tree of the remote repository will not be updated by the push. If you are pushing to the currently checked-out branch, the results will not be what you expect.

Why use Patches?

Integrating Patches with Git

- git is a peer-to-peer system, designed to have changes flow back and forth between developers and repositories using push and pull operations
- It works efficiently using its own **`git://`** protocol as well as using **`http://`** and **`ssh://`** protocols
- However, there are times when it is either necessary or desired to submit changes via a more conventional **patch** mechanism, generally through the vehicle of email

Why Use Patches?

- The first reason for doing this is to encourage review before changes are merged
  - For many projects, such as the Linux kernel, there are well established mailing lists and discussion groups
  - By enabling more eyeballs to view the patch, suggest or make changes and test before final submission, better development can be achieved
- A second reason is not all developers are using git
  - By reverting to the patch method these developers can also play with proposed change sets
- A third reason is that even if developers are using git, there may be interference, such as corporate firewalls, that interfere with using the git, ssh and even the http protocols
  - Email is likely to provide a method for bypassing these restrictions
- Fortunately, git has all the builtin infrastructure for handling patches and even for handling direct emailing

A brief review of the **patch** utility and its relation to **diff** is in order. Suppose you have a directory tree named **devel** which was based off of directory tree **stable** and has made some changes. A patch file is simply generated by doing:

```
$ diff -Nur stable_tree modified_tree > /path/to/my_patch
```

Remember the **-N** option means include files that have been added or removed in the patch, **-u** means a unified difference, and **-r** means recursive. If you want to just compare two individual files named **original** and **modified** you just do:

```
$ diff -u original_file modified_file > /path/to/my_patch
```

To apply the patch to the **stable** directory tree, another developer just has to do:

```
$ cd stable ; patch -p1 < /path/to/my_patch
```

where the **-p1** option indicates the patch was made while sitting one directory up.

Producing Patches

The basic command for producing patches is **git format-patch**. The arguments given control which and how many patches are produced.

For example, doing:

```
$ git format-patch -3
```

will produce a patch file for each of the last 3 commits, with names like:

```
0001-This-is-the-first-commit.patch
0002-This-is-the-second-commit.patch
```

where the names are generated from the commit messages, and they are sequentially ordered in historical sequence. Each patch will cover all files that have been changed in the commit in one single patch.

You can also do something like:

```
$ git format-patch master
```

to get all changes since the branch off the master branch, or use any kind of commit identifier or tag, and you can also specify a range of commits.

There are plenty of other options. A good one to use is **--signoff** or **-s** which adds a line in the form of:

```
Signed-off-by: A Smart Guy <asmartguy@linux.com>
```

according to the settings in your configuration file. This leads a clear trail as to who contributed what and is mandatory in some projects, such as the Linux kernel.

Emailing

`git send-email`

- Not surprisingly the command to directly email a patch is **`git send-email`**
- To send a message to the Linux kernel mailing list you would do:

  `$ git send-email -to linux-kernel@vger.kernel.org 001-first-commit.patch`
- When doing this, you will be prompted for some information, such as who the message is coming from
- Additional information such as your smtp mail agent configuration can also be specified, as well as additional recipients of the emailed patch
- Whether or not this will work without adjustment on your system's mail setup (such as a possible re-configuration of sendmail) and how to fix any problems is beyond our scope here, as it can be difficult depending on your system configuration, and it might be impossible to do without being a privileged user

Applying Patches

In a perfect world, applying the patches is as simply as doing:

```
$ git am 0002-This-is-the-second-commit.patch
```

assuming you are in a branch that has not yet had the changes incorporated in the patches. This command is very strong; it not only applies the patches to the working copies in your project directories, it also does a commit.

It is more than possible that one or more of the patches will fail due to conflicting lines of development between the patch sender and receiver. Such problems will have to be resolved one by one. For example, you may get a message like:

```
error: patch failed: file2:1
error: file2: patch does not apply
Patch failed at 0001.
When you have resolved this problem do: git am --resolved.
If you would prefer to skip this patch, instead do: git am --skip.
```

You can also back off the patches and restore the original branch with:

```
$ git am --abort
```

You may prefer to do things by hand more cautiously. You can apply individual patches directly to your working copy. For example, you can try:

```
$ patch --dry-run < 0002-This-is-the-second-commit.patch
```

If no problems are encountered, run again without the **--dry-run** option. Then, you would have to run **git add** on the affected files and eventually do a **git commit**.

There is another, lower-level command, **git apply**, which is the basis of **git am**. If you do:

```
$ git apply --check 0002-This-is-the-second-commit.patch
```

it functions like **patch --dry-run**; it will not actually do the patch. If there are no problems, you can run again without the **--check** option. This patches the working files in your directories and updates the Index. You will still have to invoke **git add** and **git commit** eventually.

Note that **git apply** does not modify the index; you can also use **--cached** to apply the changes only to the index.

Changes induced by **git** patch applying commands:

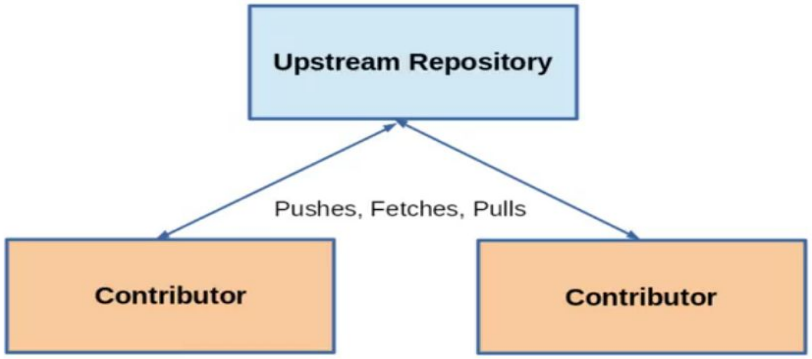
Command	Source Files	Index	Commit Chain	References
git am	Modified by patch	Updated to reflect patch	New commit object created and added to top of commit chain	**HEAD**; points to new commit object
git apply	Modified by patch (unless **--check** option specified)	Unchanged (unless **--index** option specified)	Unchanged	Unchanged

Gerrit

Models of Distributed Development

- Git has well-established methods of workflow
- It is pretty flexible and projects can choose quite different approaches, but it is always based on a cycle of:
  - Make changes in the code, probably in a development branch
  - Commit those changes to the branch; there may be one or more changes (patches) per commit
  - Publish those changes through a push or a pull request
  - The changes will be reviewed and merged if necessary or sent back for further work
- This method works well if you have a basic pyramid view of things where each subsystem has a maintainer managing their piece of the work, and that manager:
  - Has ultimate authority over the changes to that subsystem before passing them up the pyramid for ultimate review and merging

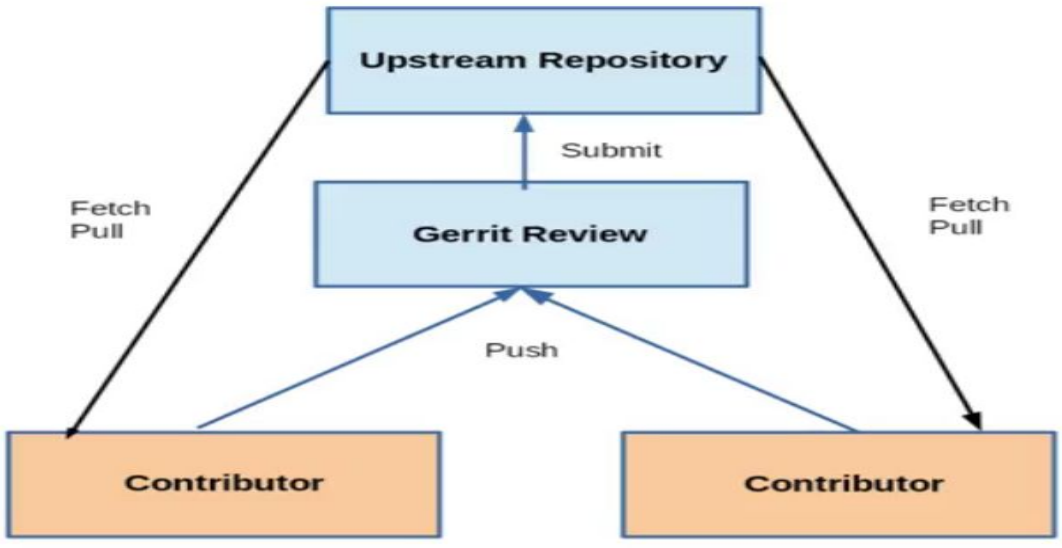
Simplified Git Workflow



Gerrit

- Gerrit comes in when you want to have more dispersed view
- While such a workflow is not exactly new (most projects have multiple reviewers with some structure for who makes the ultimate decisions) the Gerrit architecture is designed to formalize this procedure
- This works best when there is one change per commit, rather than a block of them, as it makes it easier to review and modify/reject/accept each one on its own merits

Git Workflow with Gerrit



Review Process

- From the preceding diagram, one can see that Gerrit introduces a reviewing layer that lies between the contributors and the upstream repository
  - Contributors submit their work (one change per submission is best) to the reviewing layer
  - Contributors pull the latest upstream changes from the upstream layer
  - Reviewers one the ones who submit work to the upstream layer
- The reviewers evaluate pending changes and discuss them
- According to project governing procedures they can grant approval and submit upstream, or they can reject or request modifications
- Gerrit also records comments about each pending request and preserve them in a record which can be consulted at any time to provide documentation about why and how modifications were made