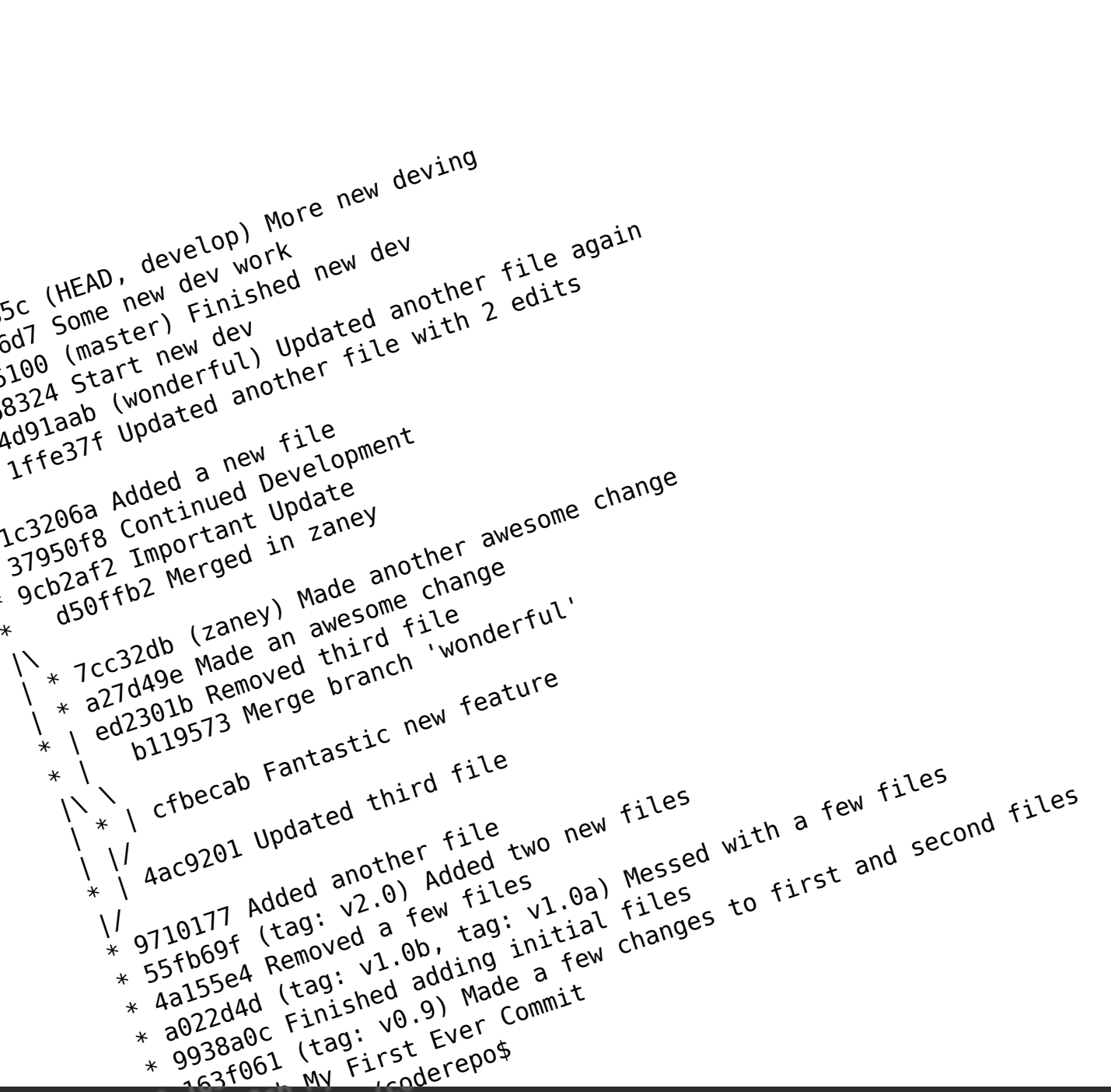


Git In The Trenches

Peter Savage

Git in the Trenches

Peter Savage

August 2011

Contents

Introduction

How this book works

Welcome to Git In The Trenches or GITT, a book designed to help you both apply and understand the subtleties of Git, perhaps the most powerful version control system in use today. This book is not supposed to be purely a technical reference. If you are looking for something more reference in nature, you should look at ProGit, by Scott Chacon, published by Apress, as it is a wonderfully detailed look at many of the Git commands. GITT is more of a scenario based book and by reading it, it is hoped that the experiences and scenarios that you encounter will help give ways to apply Git in practical situations. Git is a hugely powerful system and once harnessed you are most likely going to wonder how you managed without it.

GITT follows the lives of some developers at a fictional company called Tamagoyaki Inc. They are a small software outfit who write bespoke software for people. It may be that you work for a company that is very similar to Tamagoyaki Inc and you are looking to implement a version control system for your own company, or it could be that you have been using a version control for a long time and are looking for a helping hand in applying the Git system to your needs. Regardless of which box you fit into, GITT should provide you with some useful knowledge in a way that is designed to help you remember the scenarios and their associated solutions.

The book will follow the lead developer John, as he works to bring the company into line by implementing a version control system. It's not something he's ever really used in earnest and he feels a little out of his depth. It is hoped that your confidence and knowledge about both version control systems, and Git in particular, will grow whilst reading GITT.

The chapters are presented as weeks during the implementation of Tamagoyaki Inc's version control system project. Each chapter spells a new week in the project and you will follow the life of John and his colleagues as they solve problems and learn tricks of the Git trade. As well as presenting and solving common issues, the book will also be littered with breakout boxes, intended to tell you exactly what is happening inside

Git at each stage. This is intended to further your knowledge and understanding of this powerful piece of software. At the end of each chapter are "John's Notes" which should build into a quick reference guide.

After each week, there is also the opportunity for you to delve further into the guts of what has been presented during the week with the *After Hours* sections. These sections will take the knowledge that you have learned during the week and take it to a much deeper level, often showing diagrammatic ways of how commits take place, or even looking at the contents of files within the repository file structure itself. These chapters are not meant to scare you, but are presented in the hope that you can reinforce what you have learned through the week by reading through more complex material.

With a system as complex as Git, knowing the commands is often not the only piece of the puzzle. A good understanding of the underlying system, and how it reacts when you press that all important **Enter** button is essential if you want to be able to hold your cool in a crisis. It is rather difficult to break Git. There are many safeguards in place, but please be aware that you should try all of the items in the book in a test environment first, to ensure they perform, what you expect.

In the text of the book you will find numerous output listings like the one below. These not only show you the commands that are being run to interact with the Git system, but also the output that you should expect to see. You output will always differ to that which is in the book, if sometimes only slightly, but the output has been provided so that you can follow what is happening at each stage.

```
john@satsuki:~/coderepo$ ls -la
total 36
drwxr-xr-x  3 john john 4096 2011-07-07 19:12 .
drwxr-xr-x 13 john john 4096 2011-07-09 20:02 ..
-rw-r--r--  1 john john   35 2011-07-07 19:12 another_file
-rw-r--r--  1 john john   25 2011-07-07 19:12 cont_dev
drwxrwxr-x  8 john john 4096 2011-07-07 19:17 .git
-rw-r--r--  1 john john    8 2011-03-31 22:15 temp_file
john@satsuki:~/coderepo$
```

References in the text to commit IDs and branch names will usually be written in **bold** and words of general interest will be *emphasised*. Where commands of directory names are referred to in the text, they will be written in a `monospace` font for easy distinction. Throughout the book, you may also encounter *callout* boxes, like the one below.

Information

Callout boxes

These boxes are used to convey extra information, or to more accurately define terminology. They are there to give you that little bit of extra information.

With the introduction over, let's first go through a quick setup guide and then find out why Tamagoyaki Inc even needs a version control system in the first place.

Setting up

Making sure you have everything

Obtaining Git

Of course the most important tool we are going to need in this journey is Git itself and obtaining Git depends on the operating system you are using. Though this book uses the Linux operating system throughout its examples, almost all of the Git functionality described in this book can be performed no matter which operating system you choose. You may find a few Linux commands used to perform operations like listing directory contents or echoing strings into files. You operating system will likely have some functions that are similar, but these are not recorded here in the book. The version of Git used throughout this book is version `1.7.4.1`, but generally you should obtain the latest version as the functionality and output should not differ significantly.

Windows

For the Windows operating system, the easiest way to obtain Git is to use the *msysgit* package. This package contains several helpful options other than the main Git binaries. *msysgit* includes two context menus called *Git GUI Here* and *Git Bash Here*. These components attach themselves to the right-click menu in Windows so that if you right click on a folder that contains a Git repository, you can work on it either in a command line, or graphical way. This will be discussed in more detail later in the book. During the installation of the *msysgit* package, users are generally asked to make two choices, one is regarding their PATH setup and the other is to do with line endings. For both of these it is recommended, at least at this stage, two choose the default options. The Git implementation on Windows is currently considerably slower than implementations on Unix based systems. However, it is still operates very well on the Windows platform. The *msysgit* package is available from http://git-scm.com.

Linux

For the Linux operating system, most Linux distributions come with Git packaged in some way. In Ubuntu for example, one can install Git simply by running the command `sudo apt-get install git`. There are many extra items on Linux that you can install that are related to Git, however for the purpose of this book, just the core package and the gui are all that are required. If you cannot find a package for your distribution, you can always either compile it from source, or take a look at http://git-scm.com to see if there is one available for your system.

MacOS

For MacOS If you are using the MacOS platform, Git can be found as a download from http://git-scm.com.

Week 1

Day 1 - "Things need to change"

Meeting the Team

If you're already a seasoned version control user, you may want to skip this chapter. It's kind of like an introduction to why we even need version control systems in the first place. This chapter looks at Tamagoyaki Inc's requirements and why they choose the VCS that seemed right for them. Tamagoyaki Inc. creates software for turning a standard PC into a media center. Their product ships to the end user and they rely very heavily on having a good presence at trade shows, in order to bring in sales. The following conversation describes the events that led up to the defining "We need a VCS!" discussion.

> **In the trenches...** John sat at his desk and looked out of the window. The rain was drizzling down the pane, but he didn't care. It was a quiet Monday morning, the release had gone well on Friday and John was just thinking about implementing the new abstraction layer to the database he'd been asked for. Through the music playing in his headphones he hardly noticed his boss, the chief designer and the CEO approaching his desk.
>
> "John," shouted his boss, Markus, "get your team into the board room. Now!"
>
> Things didn't look good.
>
> * * *
>
> "So, what we'd like to know John, is just how a bug that was supposed to have been..." the CEO back-tracked, "that was demonstrated

as being fixed two weeks ago, made it into the final release of the software?"

"I'm sorry," John started, before being cut off.

"Sorry doesn't cut it John," said the CEO, Wayne Tobi, "This was almost a major embarrassment for Tamagoyaki Inc. and we need to ensure this doesn't happen again. The demonstration at the trade show was close to a complete failure. Luckily someone had the good sense to bring a backup machine." He turned to Markus. "I want a report on my desk by the end of the day that states what the problem was, how it slipped through our fingers and what safe guards we are going to put into place so that things like this never happen again."

"Of course sir," Markus replied. He was bright red with his own variety of embarrassment.

The room fell silent and a few minutes of silence passed before the meeting was drawn to a close and John and his team were allowed to leave.

* * *

"So, you're telling me that when Simon came back from holiday, he picked up an older copy of the library from the network share and pushed his latest code into that?" Markus was holding back the anger.

"It appears that way." Said John sullenly.

"Oh for crying out loud. How did this happen? Why wasn't he using the latest version? And why didn't QA pick up on it?" Markus looked across the meeting room at John. "John, you need to make sure this doesn't happen again. Find a solution!"

The trouble with storage

It's not like this situation is completely uncommon. At one point or another most people have managed to pull old code from somewhere and mistakenly use it in place of the latest, up to date, version. When storing code on network shares or on local disks, it's easy to lose sight of which version is which, no matter how good your naming

convention is. It's like trying to build one of the baked bean puzzles when you have three boxes of them and you tipped all the pieces into one box for simplicities sake. Not so simple any more is it?

People have a tendency to use folder names which mean something to them. However it doesn't necessarily follow that this name means something to another developer. "Version 2.3 – fixed bug a" only means something to you if you know what bug a is and something like "Version 2.3 – fixed bug a(2)" is even worse. Unfortunately allowing people to free form type their own descriptive file names will always lead to problems like this. When these files are stored on a network share, the problem is exacerbated ten fold because there is often no fixed reference point.

So what's the solution? Well, in a large number of cases version control can make sure that not only is there a defined place for data to sit, and with a defined structure, but also that you have a full history of the code. Accountability is very important in code development, especially when releasing software to customers. In some situations a customer will even mandate that the code being developed for them is stored in a version controlled environment. In this way, the customer can ask when a certain piece of code was edited, or when an addition first entered the code base.

Day 3 - "A possible solution"

Version Control Nuances

There are many offerings for version control out there, Git, Mercurial, Subversion, CVS, and Bazaar to name but a few of the open source ones. Perhaps a more pertinent question is just which version control system to use. Each of them has their relative advantages and disadvantages, but some will be suited to certain tasks more than others. Also, it's worth noting that if you are interacting with other pieces of software, or share some development with another set of developers, it is a good idea to enquire to see what they are using. Usually you'll find collaboration, forking and patching a lot easier if you're using the same version control system as your upstream or partners.

> **In the trenches...** "So really it seems like the only real solution to this problem bar Klaus' suggestions of reducing the workforce to only one developer, thank you Klaus," Klaus nodded in acknowledgement back to John, "is to implement a version control system."

> Markus chewed his lip. "I can see where you're coming from here John, but aren't version control systems really expensive?"
>
> "There are a number of open source offerings we could take a look at first," piped up a new voice in the discussion, "some of them are supposed to be pretty good."
>
> "Let's all go away, take a look at the various pros and cons and reconvene tomorrow to discuss the findings," said John. "Sound fair?"

So now we need to take a look at some various features of version control systems and see what the various advantages and disadvantages are of each. We are going to focus on Git here primarily, as this is what the rest of the book is all about. It is assumed that if you are reading this book, you have most likely already made the decision about which version control system you are going to use. So let's talk about the various features that are prevalent in most version control systems.

Distributed Version Control

Version control systems usually fit into one of two categories; centralised, or distributed. Git is a distributed version control system. It has been designed to run almost everything at a local level. This will become much more clear when we talk about other features of Git a little later on, but for now just understand that Git isn't tied to a centralised repository. This is super powerful. No Really!

Branching

Most version control systems offer branching as part of their default feature set. Branching allows developers to create in essence a clone of their repository and mess around with it, safe in the knowledge that they can switch back to the original whenever they need to. This allows developers the freedom to experiment with all manner of things without being afraid of affecting the original/clean code base.

Git implements branching in a special way. Most older version control systems implement branching in a way that almost creates a separate copy of the repository. This is slow and cumbersome. Git's branching method gives developers the ability to create multiple local branches to play with. Due to its distributed nature, when pushing code to a more central location for others to pull from, developers can choose which branches they want to push, allowing code to be experimented with privately.

The implementation of branching in Git is fast. Due to the fact that repositories are stored locally, the speed of creating a local branch is limited only by the speed of the disks on a local machine. Git's implementation is so lightweight that even this speed factor is negligible.

Staging

Git deals with committing changes into the repository differently to most other version control systems by the introduction of the staging area. The staging area allows developers to prepare their commits before they are written to the repository. Why is this useful or any different to any other version control system. In Git you can make a change to a file, add it to the staging area, and then continue to make changes to that file, even though you have not yet actually committed anything. It should be noted that it's not absolutely necessary to use the staging area, but it is there for developers wishing to utilise it.

Workflow

Due to the way that Git has been designed, it's possible to use it in practically any work flow you can think of. Three of the most common workflows are explained below, and Git can work in any of these, making it one of the more versatile systems out there.

Centralised Workflow

In a centralised workflow, a single shared repository is used. Multiple developers pull changes from here into local working copies, work on the local version, and then push it back up to the central location.

Git handles this workflow just like most other version control systems. A developer can not push his changes until he has pulled everything up to the latest from the central repository and resolved any conflicts that may arise.

Using the centralised model for the workflow, each developer has the same level of access to the repository and is considered as **important** as each other. For smaller teams, this method will work well, but as teams get larger, a centralised method may get tedious. As more and more people start to access the same files, conflicts and other issues often begin appearing more often.

Integration Manager Workflow

The integration manager workflow is similar to the centralised workflow because there is still a **blessed** repository which everyone uses as a reference. The difference here is that

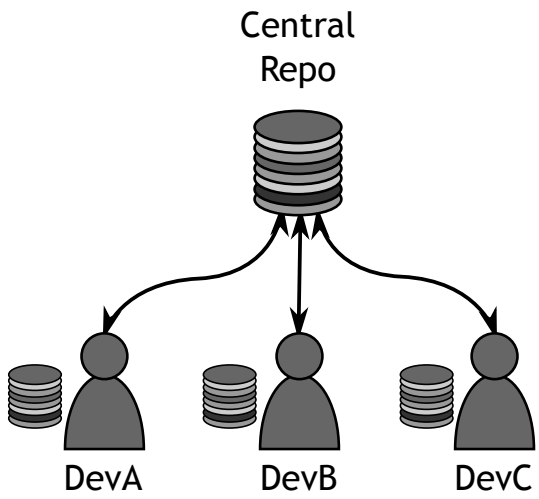


Figure 1: Centralised Workflow

there is only one person who has the rights to push changes to the **blessed** repository. This person is referred to as the Integration Manager.

This workflow is handled exceedingly well by Git. Developers will work on their repositories locally and then once they are happy, will push their changes to a location where the Integration Manager can see them. The Integration Manager will then review the changes that the developers have made and will merge them into his own local repository. Once they are happy that everything is working well, the Integration Manager will push their changes to the blessed repository so that all the other developers can access the changes.

Dictator and Lieutenant Workflow

The dictator and lieutenant workflow is practically an extension to the integration manager workflow. It is more suited to larger teams, where modules or sections of the code can be assigned to a **Lieutenant** who is responsible for blessing all of the changes to that particular section.

Once the Lieutenants are happy with their code, they make it available to the Dictator. The Dictator then takes on a role similar to the Integration Manager from the previous model. In the end, all of the changes are pushed to the blessed repository for the developers at the bottom of the tree to pull from.

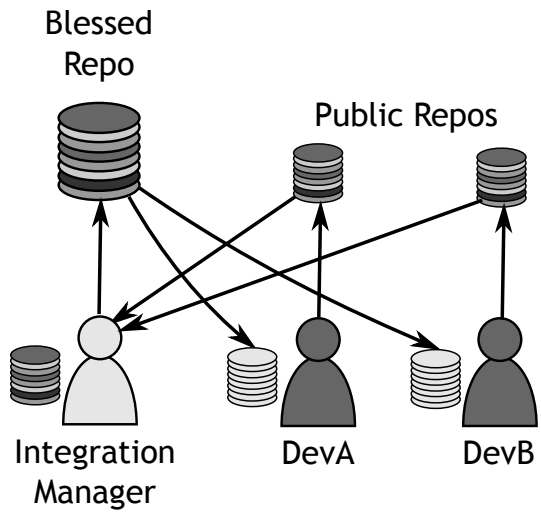


Figure 2: Integration Manager Workflow

The main thing to remember, is that Git can utilise any of these workflows. This makes it a very flexible system, allowing you to work in whichever way you decide.

Offline Committing

Perhaps one of the most useful and undervalued features of distributed version control systems is that of offline committing. It may be undervalued because not all version control systems have it. Offline committing is the ability to continue adding and committing files to the repository without being connected to a centralised repository.

When travelling or just simply when out of the office, developers and integrators alike are able to continue managing code, viewing histories, viewing diffs and committing changes to their repository. This is all due to the fact that Git does 99% of all of its operations locally. When a repository is cloned, Git actually sets up a copy of the entire repository locally, giving developers the flexibility to work anywhere, without requiring access to the company network.

When returning to the office, the developers simply push their changes to their "public" space, be it local or to a blessed location, and all of the commits that have been made whilst they are away are then made available to the rest of the team, including all history and snapshots.

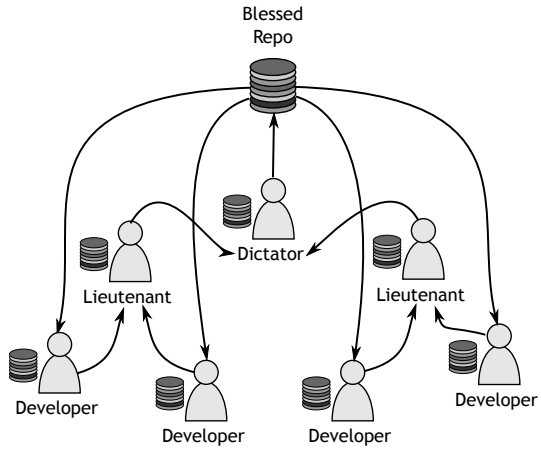


Figure 3: Dictator and Lieutenant Workflow

Terminology

Blessed

A blessed, or canonical, repository is one which has the approval of the managers of the project. The blessed repository is supposed to be the de facto standard where all other clones are made from. If there is one place where code should be correct, it is the blessed repository. If you are hosting the project in a public place, the blessed repository will usually be the one that is made available to people as a stable point for developing from.

Developer Interaction

One factor to consider when choosing a version control system, is that of developer interaction. By this we are referring to the way in which developers use and interact with the version control system itself. There are four main methods for VCS interaction

Graphical User Interface Client (GUI)

A graphical user interface allows the developer, or user, to physically manipulate the repository using a mouse pointer and a graphically rich environment. A GUI client will typically consist of separate application which is run when a user wants to make changes

to a repository such as adding files or committing changes.

Some developers prefer having a separate client with which to interact with their repository, whilst others prefer to have things integrated a little more.

Shell Extension Integration

Shell integration allows the user to interact with their repository using the graphical environment that they would usually use for manipulating files and performing routine directory maintenance. One of the most common Shell Extensions for Git is the msysgit interface which integrates itself into Windows Explorer, allowing a user to right click on an entity whilst inside a git working tree, and be presented with a context sensitive menu for entering a shell for VCS operations.

Command Line Interface (CLI)

The command line interface is favoured by many developers as they can script with it and can see exactly what is going on, often in much more detail that with a GUI. The CLI gives total control over the system and it is worth noting that almost all version control systems start life as command line driven interfaces. Why is this so? It can take a lot of time and effort to put all the options and nuances of a system into a GUI. The CLI will almost always be the most powerful of all tools, especially where version control systems are concerned.

Day 4 - "A decision is reached"

Analysing Your Requirements

The most important aspect of choosing a version control system is to define your requirements. These can be few, or they can be quite specific. Let's see what John and his team decide are the most important aspects for them and ultimately what VCS they decide upon.

> **In the trenches...** "Offline committing seems like it's a pretty useful thing to have." Mike said nodding. "Especially with people like John travelling all the time."
>
> "I have to admit, it would be nice to be on the plane, and be able to pull all the code together, knowing all of the history of each section," replied John. "The branching in Git seems to be quite powerful as well."

> "I must admit," chimed in Klaus, "I've used branching a bit in Subversion before and it was a lifesaver. It's supposed to be super fast in Git too."
>
> "And owing to the fact that Git seems to support several workflows, it means we can try them out and see how they work for us." Markus looked at the team. "Are we settled on Git then?"
>
> The team nodded and everyone walked out of the board room except John. Things were about to get interesting for him. Very interesting.

Since this book is all about Git, we won't delve too far into the workings or features of other version control systems. Hopefully, this chapter has given you enough information to go and check out some of the other systems if you feel the need to. The main thing to bear in mind is that Git is a Distributed Version Control System or DVCS. While this is so, it is equally important to remember that it can be used in the same workflow models as centralised version control systems.

John and his teams requirements are nothing special. They are a smallish team looking to reap the benefits of having their code in a well organised system. They are also looking to reorder their team functions and dynamics in order to fit around the version control system and really make it core to their development.

Version control is not a replacement for workflow. It is not intended to make everything better. If you have people going off and doing their own thing and being careless about the way they work, version control is not going to suddenly fix everything. A tool is just that, a tool and version control is no different. You can buy the messiest builder in the trade a nice shiny new tool box, but unless they have the mindset to want to change, you'll probably find that all the tools end up in the largest compartment at the bottom.

Day 5 - "Working like a team"

Team Organisation

Now that we have the basics dealt with, let's take a little look at how John arranges his team, and see whether version control is going to work for them. It is important that the team understands how the model should work, what they are expected to do and what level of access they have. Most of the time people will get more frustrated about

not knowing what they should or should not be doing, rather than that they do or don't have access to certain things.

In the trenches... It was 4:36pm on the Friday and the table in the board room was littered with empty coke cans, pizza boxes and one Japanese bento lunchbox, owned by a particularly stubborn member of the team who had vowed never to eat pizza again. It had been Marcus' idea to bring in the food reinforcement to help the discussions along. The team were trying to decide how to organise their model.

"There's nothing to say we can't use a combination of the models is there?" asked Mike.

"I suppose not," said John. "What did you have in mind." His glasses were slipping down his forehead now and he was getting pretty tired.

"Well, I figure, we basically have the software split into two parts. We have the library, which myself, Klaus and Jack work on. Then there's the UI elements which Simon, Martha and Rob handle. I know there are the tools which Eugene works on too." Everyone had started to listen to Mike as he continued. "John, you don't want to have to deal with the library component as this more Klaus' space. So why don't we have two dictators. Klaus and yourself have access to push up to the blessed repository. John can pull from his guys, Klaus from his and we end up with a good model for version control."

John raised his eyebrows, "Not bad Mike," he said, genuinely impressed. After spending a few hours going through the various models and who was in charge of what, it felt good to have finally reached a decision.

"So, we start on Monday then?" asked Markus, who had been listening from the other end of the table.

"Indeed." Announced Klaus, "Monday we all become Gits!"

After Hours Week 1

"History Lesson"

A Brief History of Version Control

The Very Early Days

Version control systems have been around for forty years (2011 at the time of writing). During this time they have undergone an intense amount of change and have evolved into some of the most incredibly powerful tools utilised in software development today. Chances are that in the early days you will have started off storing different versions of your source code and documents in separate files and folders. You may have even archived them off to compressed storage files, like zip or tar. Rest assured, you are not the first person to do this, and in 1972, someone called Marc J. Rochkind, decided to create system for storing revisions of documents and source code.

The system Marc created was called SCCS and stood for Source Code Control System, in essence probably the most apt description for what we mainly use a version control system for today. SCCS was originally written for an operating system called OS/360 MVT and was later ported to C, and was used as the most dominant version control system for UNIX, until ten years later, when RCS was introduced.

Time To Move On

In 1982, Walter F. Tichy released RCS, standing for Revision Control System. It was intended to be a free and offer more functionality than SCCS. RCS is still being maintained, as part of the GNU project, and at the time of writing is about to have its first new release, version 5.8, in over fifteen years.

However, RCS, like its predecessor SCCS, has no way of dealing with groups of files. Essentially, each file has its own repository which is stored near to the file under a different name. Whilst rather advanced, with primitive forms of branching, the interface, commands and version numbering have been described by some as rather cumbersome. Enter some successors.

CVS (Concurrent Versions System) was created in 1986, and began life as a set of shell scripts to operate on multiple files, using RCS to perform the actual repository management. As development continued, this way of working was dropped and CVS began operating on files itself, evolving into a version control system in its own right. The current iteration of CVS was released in 1989 and on November 1 1990, version 1.0 was released to the Free Software Foundation for distribution.

CVS did not version file renames or moves at all as at the time, re-factoring - a process of modifying code to improve some non-functional attributes of the software, was often avoided and so the feature was not required. CVS also did not support atomic commits. An atomic commit is used by more modern version control systems to safe guard the database. In essence atomic committing is the act of applying multiple changes in a single operation. If any of the changes do not apply correctly, all others are reverted and the commit is aborted. When designing CVS this was not seen as an obstacle, as it was thought by the developers that a server and network should have enough resilience that it would never crash whilst committing.

Whilst active development of CVS has apparently ceased, as of May 2008, it is worth taking note that CVS defined the model for branching that was included and refined in almost all version control systems since.

Offering Commercial Support

Now that version control was advanced enough and people had begun to rely on VCSs in general, commercial offerings began to spring up. Three prominent systems that were released within a short time of each other; ClearCase, VSS and Perforce. All three of these are proprietary systems which were developed and filled a gap in the market for commercially supported systems.

VSS, originally developed by One Tree Software for several platforms, was continually developed by Microsoft, who bought One Tree Software in 1994, with the one caveat that Microsoft ceased development of all VSS on all platforms other than Windows. VSS integrated into Visual Studio, Microsoft's Integrated Development Environment. VSS has now ceased development, but ClearCase, now developed by a division of IBM, and Perforce are still being actively developed and maintained.

The Millennium

The millennium brought with it a new breed of version control systems. Subversion, or SVN as it is colloquially known as, was developed primarily to be a replacement and mostly compatible successor to CVS. SVN was first released in 2000 and by 2001 was

able to sufficiently host its own source code due to its own advancement. In November 2009 Subversion was accepted into the Apache group and is currently developed and maintained by its community and by several commercial entities.

Subversion brought things to the table that previous version control systems did not. As it was released as free software, in the same vein as CVS, it was widely adopted by the open source community and later into commercial environments for its vastly improved feature set. For a start SVN offers true atomic commits. This gave it a definite advantage over CVS as it was seen as a truly robust alternative.

It also brought in features like the tracking of files through renames and moves, including their entire version history and the versioning of symbolic links. SVN moved with the times and introduced many other sought after features, such as HTTP serving, cheaper branching, efficient network operation and native support for binary files.

As with all version control systems, there are aspects that people dislike. In Subversion, people often find the implementation of tags – that is names that point to specific points in the history of a repository, an issue. In SVN, a tag is actually a branch. What makes this different to other systems such as Git and its predecessor CVS, which literally point to a specific commit in the tag, SVN actually creates a snapshot of the file system. Although it employs relatively cheap branching which is lightweight on the repository, the tagging model used can be incredibly heavyweight on the client.

Another issue with the tagging model in SVN is that it holds no history information. This makes it impossible, for example, to take two tags and try to find out all logged commits that occurred from one to the other. This is the difference between using a copy as a tag, and implementing a reference. Tags should also be read-only implicitly by their very nature, they should refer to a point in history. However as tags are implemented as branches in SVN this is not the case.

Introducing the Linus Factor

The Linux kernel was at one point maintained under a source control system called BitKeeper. The decision, in 2002, to use BitKeeper for the management of the Linux kernel source was rather controversial, the main opposition being that BitKeeper was a proprietary system that was offered by BitMover. At the time, BitMover offered certain open source projects the opportunity to use BitKeeper at no cost, so long as the community developers did not engage in the creation of a competing tool.

In April 2005, BitKeeper withdrew the free license that it had granted to the open source communities, after allegations of reverse engineering by some parties on an

unrelated project. Due to the way BitKeeper worked, and decisions made regarding licensing it became impossible for several key developers, and according to some reports including Linus himself, to actually own even a commercial version of BitKeeper.

It was due to these circumstances, that Linus Torvalds himself, decided to begin writing his own version control system that would enable him to have all of the features that he had had available to him with BitKeeper, the most important of these seemingly being a distributed environment. Linus decided on a set of criteria, which along with a distributed environment, also included a robust safeguard against corruption, be it accidental, or malicious, and very high performance.

Git development began on the 3rd of April 2005, and by April the 7th, the project had been announced and was already able to host itself. On June 16th, the release of the Linux kernel version 2.6.12 was managed by Git. Junio Hamano, who had been a major contributor to the project, took over maintenance of Git in July, and by December had released version 1.0 to the community.

Interestingly enough, another project was also created as a result of this chain of events. Mercurial or Hg as it is often known, is reported to have begun development on April 19th of the same year and was started by Matt Mackall with largely the same goals as Git. Though Git was chosen to be the version control system used by the Linux kernel, Mercurial is actively used by many other projects and shares a very similar design and concept to Git.

Design Changes

Git has some features which should be discussed here as many of them are different to every other version control system. Perhaps one of the most important of these is the very strong emphasis on non-linear development. Git provides many tools with which to work with many branches and merges, with a core principal being that changes will often be passed around more than they will be written.

Git is very fast. Though for certain operations it may be slower than some of its peers, Git has been consistently proven to be faster than most. This design implementation was essential as the Linux kernel is indeed a very large project.

When developing systems that offer any kind of security to an end user, it is essential to provide a way of auditing the history of the code, to ensure that no tampering has taken place. Due to the way that Git is implemented, each SHA-1 hash, used to identify a particular commit, depends on the entire history of the repository. What does this mean to someone viewing the repository? Once the repository is published, it

is impossible for someone to tamper with the history without someone noticing. This is called cryptographic authentication of history.

Of course the most important feature, and one which will be discussed in great detail later in the book, is that of distributed development. The emphasis on non-linear development and the implementation of very cheap and fast branching, makes Git one of the best version control systems on the market for distributed development.

As mentioned previously every version control system has advantages and disadvantages and it would not be fair to make out that Git was without flaws. Some that have been mentioned by people over the years are its particular steep learning curve for basic understanding. Whilst it is agreeable to some degree, it is largely due to the fact that Git is a distributed version control system and inherently these systems are more complex in their implementation that others. Another bug bear of some people, is the fact that Git will not track empty directories.

Wrapping Up

Though there are many other version control systems out there that are being actively developed, such as Bazaar, Plastic and Darcs, to name but a few, we are going to end our historical tale here and continue with learning more about the Git version control system. There is a plethora of information available on the Internet about version control, so if you want to find more information about any of the systems mentioned here, that would probably be the best place to start.

Week 2

Day 1 - "We are coders, we use Git!"

Setting Up the Environment

So now we are ready to begin delving into and actually using Git, right? Well, not exactly. First we have to decide upon how the workflow model we have envisaged is implemented in our version control system. With Git being so versatile, it's both a blessing and a danger. It is a good idea to define from early on, exactly how you would like the developers, lieutenants and dictators to behave, before you begin actually committing any code. Sometimes this isn't possible. It's quite feasible that you have never used a version control system like Git before and you begin by muddling your way through. This is normal, but if you are in charge of implementing this type of system for a professional environment, you should really consider first how this is going to work.

Conceptually, the model which was discussed previously is easy to imagine. We have two dictators, who both have access to the blessed repository and then several developers, who are going to have their changes reviewed and included, by the aforementioned dictators. The physical representation of the workflow model is summarised in the diagram below.

The physical structure is all well and good, but it doesn't determine exactly how the data is moved, just who is responsible for it at each stage in the process. What is required, is a detailed analysis of where the data flows from and to. A data flow diagram is useful, but not essential. However, we will create a slightly different form of diagram to show how the data will be moved from one person to another. Before we go ahead and look at the diagram, let's go back to the trenches to see how the guys are coping with their repository design.

> **In the trenches...** "John, why are we all sat in here at 9:45am on Monday morning." Klaus whined. "I haven't even ingested enough coffee to check emails yet, let alone meet with people."

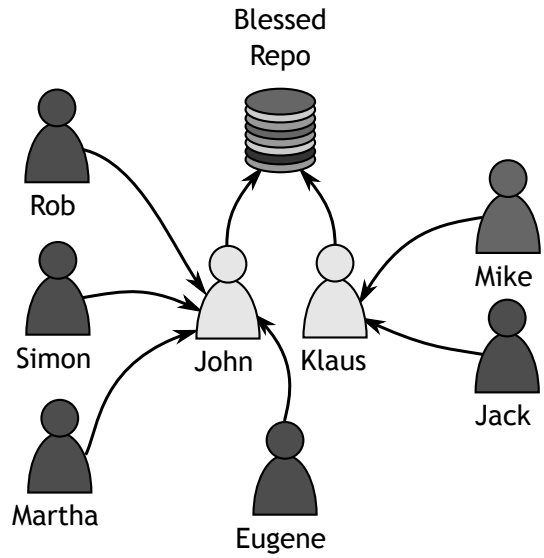


Figure 1: Tamagoyaki Inc's Physical Structure

John grinned, "I don't think any amount of coffee will help you there Klaus, it's your winning personality that will pull you through." The rest of the team laughed and then subsided as John started drawing furiously on the board. "So we have our physical model. We know which people are going to be in charge of things, but we don't know yet how to arrange our repositories."

"Good point," chimed Mike.

"So. Obviously we're going to have a blessed repository," said John, drawing a circle on the board. He stepped back, one hand on chin. "Then I would imagine Klaus and I will have clones of that repository on our local machines. We will then modify those and push our changes back up to the central copy."

"I thought Git didn't have a central copy?" asked Martha. There were other moans and grunts.

"Well," said John, "as far as I understand it, it doesn't. I mean Klaus

and I will have local copies of the repository too. We will work on those and then sync our changes back to the server. It's a sync, moreover a copy. I think it's actually called a clone." He nodded to himself, "And, since Klaus and I will hardly ever overlap on code, we shouldn't ever need to merge or deal with conflicts."

"But what about us monkeys?" asked Martha, "Where do we get our clones from?"

"From the central server of course," Rob stated smiling.

"Yes," John said, "but I think what Martha is trying to say, is how will you get your updates?" He started to walk around the room, and one or two of the developers followed him as he reached the windows and leant on the sill. "I guess you would merge your branch with the blessed one."

The room went silent and the only noise that could be heard was the rattling of the air conditioner in the ceiling above.

Simon spoke out, "Well, I was reading over the weekend about this thing called rebase and how in some cases a rebase is better than merging."

"What's rebase and how is it different to merging?" asked Mike.

"Well, rebasing is pretty darn clever. Think of it this way. You have an upstream branch, in this case, our blessed repository. You are happily making changes. When the upstream changes, you could merge the changes in from blessed. If you do this, you create a single commit which merges the changes in. It works, but..." he trailed off a little, "it can cause problems in certain instances. A better way to handle it is with rebasing. Rebasing can take all the changes you have made, squirrel them away, pull down all the changes to bring it up to date, and then whack your changes on top. It's not always the best choice, but we should consider it."

John breathed out, "It sounds pretty cool Simon, but one thing is abundantly clear, we need to learn more about the Git basics before we start delving into this merging and rebasing. Let's spend the rest of the day playing with some test repositories and reconvene tomorrow."

If you've never played with a version control system before it is a good idea to take some time to just play. Pretty soon you'll have learnt the basics and will be in a position where you will want to put your newly honed skills into practice. Though playing on test repositories is good, it is quite usual that you need to actually use the system in a real environment before real problems arise. The rest of this chapter is a very quick introduction to Git. It is presented as an introduction, because it is hoped and expected that the you will take some time out to get to know the system and how it works. Knowing something about the underlying mechanisms of Git will definitely help you as you progress and will save you an awful lot of frustration later on when operations do not seem to function as you expect. After Hours 2 focuses on the Git object model, something all Git users should have a basic understanding of.

Initialising A Repository

The first thing we need to do is to understand two very important things:

1. How to create a Git repository
2. What a Git repository actually is

The first of these is relatively easy to perform.

```
john@satsuki:~$ mkdir coderepo
john@satsuki:~$ cd coderepo/
john@satsuki:~/coderepo$ git init
Initialized empty Git repository in /home/john/coderepo/.git/
john@satsuki:~/coderepo$
```

If you are a Windows user, the above output may seem a little strange to you. In the Linux world, the shell often has a much more descriptive prompt than on Windows. In the case above, it takes the format `<user>@<host>:<current_directory>`. The `~` is a shortcut meaning *Home Directory*, so really `/coderepo` actually means `/home/john/coderepo`.

What we've done here is create a new directory called coderepo, moved into it, and then run the `git init` command. The result of this command is a new directory in the coderepo directory called `.git`. This directory will hold a local copy of our entire repository. This will allow us to create branches, merge changes, rebase things and ultimately push our changes to somewhere else.

Something that is crucial to the running of a repository, whether you are an administrator of Git, or a developer who is using it, is an understanding of how Git works. It is fine to jump in and play with the repository and test the water, but before committing to using Git in a production environment, you should understand what Git actually does in the background in some detail.

During the writing of this book several people have told me that Git is one of the only version control systems where a good understanding of how the underlying system works is not just highly recommended, but bordering on essential.

Let us take a few minutes to talk about how Git works internally and how the data is actually stored. Git doesn't store changes to files, but actual snapshots of files at specific points in time. In fact, each time a commit is made, Git actually makes a record of how the entire filesystem looked at that point, even if only one file is changed. It refers to files by calculating a SHA-1 hash of the file and using this as an ID, because the hash is unique to the contents of the file, it is easy to detect if a file has changed. If the SHA-1 hash of a file changes, then the file must have been modified.

When a commit is made to the repository, Git stores a few things. A commit object is created. This contains, among other things, information about who made the commit, the parent of the commit and a hash that points to a tree object. The tree object describes what the filesystem looked like at the time of the commit. In other words the tree object, tells Git what files were in there. Lastly, Git stores the files that were in the repository under their SHA-1 names in the objects directory. Of course Git is super clever here because if you have exactly the same file in multiple commits, the SHA-1 hash of that file doesn't change and therefore Git only stores one copy of the file to save space.

The commit object is also referred to by an SHA-1 hash. This is different to many other version control systems which use either a number that refers to the repository or a per file version number. Getting used to seeing 40 character SHA-1 hashes can take a little time. Saying "I need the commit referred to as bf81617d6417d9380e06785f8ed23b247bea8f6d," is certainly not as easy as saying you need revision 6. However, Git handles these hashes well, and you can reference a commit using a few of the characters from the beginning, as long as those characters uniquely refer to that commit, i.e., as long as your choice is not in any way ambiguous.

The above description may sound rather foreign to you. If this is the case, you should really spend some time reading through it again and possibly even jump to the After Hours section at the end of this chapter. Understanding the way Git stores objects is a rather important aspect of Git and though it may seem rather confusing at first, learning

this will help you later on in understanding more complex matters.

Day 2 - "Making commitments"

Let's work on our repository

The most simple way of committing a file into the repository is to create it, or copy it to the working copy of our repository and use the commands below. The working copy is the version of the repository we have currently checked out. This terminology will be explained more later on.

```
john@satsuki:~/coderepo$ touch my_first_committed_file
john@satsuki:~/coderepo$ git add my_first_committed_file
john@satsuki:~/coderepo$ git commit -m 'My First Ever Commit'
[master (root-commit) cfe23cb] My First Ever Commit
 0 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 my_first_committed_file
john@satsuki:~/coderepo$
```

What we have done here, is to create a new blank file, using the unix `touch` command and add it into the repository using the `git add` command. Windows users note, you will not have the touch command in your default command set, but if you are using Git Bash from the msysgit package, you should have it available to you. Then we have committed it into the repository using the `git commit` command. Let's make a few changes to our working copy and see what the result is. First we are going to add another two new files, then we are going to make changes to our original file and finally we are going to run git status to see what Git has to say about our changes.

```
john@satsuki:~/coderepo$ echo "Change1" > my_first_committed_file
john@satsuki:~/coderepo$ touch my_second_committed_file
john@satsuki:~/coderepo$ touch my_third_committed_file
john@satsuki:~/coderepo$ git status
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
# modified:   my_first_committed_file
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
```

```
#
# my_second_committed_file
# my_third_committed_file
no changes added to commit (use "git add" and/or "git commit -a")
john@satsuki:~/coderepo$
```

So we can see that `git status` is reporting that there are changes to our first committed file, and that our second and third files are **untracked**. Untracked files are ones which Git detects as being present in the working directory, but which haven't yet been added and there for upon running a commit, these files will not be added to the repository. Notice that if we tried to run a commit now, nothing would actually be committed to the repository. Even though there are changes to to my_first_committed_file, we have not asked Git to include these. So, let's go ahead and do that, and at the same time we'll make a few changes to my_second_committed_file, and add those too.

```
john@satsuki:~/coderepo$ git add my_first_committed_file
john@satsuki:~/coderepo$ echo "Change1" > my_second_committed_file
john@satsuki:~/coderepo$ git add my_second_committed_file
john@satsuki:~/coderepo$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# modified:   my_first_committed_file
# new file:   my_second_committed_file
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
# my_third_committed_file
john@satsuki:~/coderepo$
```

Now we can see that one of the sections has changed to "Changes to be committed". So this means that Git has recognised and remembered that we are expecting these files to be committed when we next run a `git commit`.

Committing the Uncommitted

In the trenches... "John, what is going on here?" shouted Klaus from across the hallway. The entire office had heard Klaus banging his

hands down on the desk for the last fifteen minutes. "John!" the shout turned into a scream.

"Calm down Klaus, I'm just coming." John walked over to Klaus and pulled up one of the folding plastic chairs. After a few minutes of fumbling he finally managed to take up his position next to an infuriated Klaus.

"John, Git is driving me crazy. I have added files to the repository and I keep running a commit, but the changes aren't getting put into the blasted repo." Klaus was clearly distressed and John resisted the urge make jokes.

John pointed at the screen. "Run a git status Klaus and I'll show you what the problem is."

To understand what Klaus was getting in a spin about, let's make a change to `my_second_committed_file` now and see how this affects things. Remember we have already added the file, but we haven't yet made a commit.

Note

A little Linux note

It should be noted that there is a subtle difference between > and ». The former will *redirect* the result of a command to a file, overwriting its contents. A double arrow appends the result of a command to the specified file. We use both of them in the examples as a way to show how we can simulate changing a file completely and appending extra lines to a file.

```
john@satsuki:~/coderepo$ echo "Change2" >> my_second_committed_file
john@satsuki:~/coderepo$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# modified:   my_first_committed_file
# new file:   my_second_committed_file
#
```

```
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
# modified:   my_second_committed_file
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
# my_third_committed_file
john@satsuki:~/coderepo$
```

How interesting! We now have three sections and one of our files appears twice under both Changes to be committed and Changed but not updated. What does this mean? If you remember back, we spoke about a staging area. This is one area in which Git differs to many version control systems. When you **add** a file into the repository, Git will actually make a copy of that file and move it into the staging area. If you then go ahead and change that file, you would need to run another git add in order for Git to copy your changed file into the staging area. The most important thing to remember is that Git will only ever commit what is in the staging area.

So, if we go ahead and run our commit now, we will only have the changes marked in Changes to be committed appearing in our repository.

```
john@satsuki:~/coderepo$ git commit -m 'Made a few changes to first and second files'
[master 163f061] Made a few changes to first and second files
 2 files changed, 2 insertions(+), 0 deletions(-)
 create mode 100644 my_second_committed_file
john@satsuki:~/coderepo$
```

In our examples, we have used the syntax `git commit -m 'Message'`. This is a slightly special way of committing, it allows us to specify our commit log message on the command line. If we wanted to, we could run the command git commit and this would open a text editor that we could use to input our commands.

Let us finish off our round of committing by using the `git commit -a` option. This commits all of the changes to files which are already tracked. Consequently we do not have to specify the files with `git add`, like we have had to previously. Any file which has been modified and has previously been added to the repository, will have it's changes committed upon running that command.

```
john@satsuki:~/coderepo$ git status
```

```
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
# modified:   my_second_committed_file
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
# my_third_committed_file
no changes added to commit (use "git add" and/or "git commit -a")
john@satsuki:~/coderepo$
```

```
john@satsuki:~/coderepo$ git commit -a -m 'Finished adding
 initial files'
[master 9938a0c] Finished adding initial files
 1 files changed, 1 insertions(+), 0 deletions(-)
john@satsuki:~/coderepo$
```

```
john@satsuki:~/coderepo$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
# my_third_committed_file
nothing added to commit but untracked files present (use "git add" to track)
john@satsuki:~/coderepo$
```

Day 4 - "Let's do this right, not fast"

Uh-Oh I Think I Made A Mistake

So now we are fairly well acquainted with adding files into the repository and performing commits. In a short while we will learn about how to view the changes we have made and perform diffs against various objects. Before we close out the week, we need to go back to the trenches one last time.

In the trenches... "Rob, ya got a second?" asked Mike.

> "Sure, what's up?" replied Rob from across the office. "Gimme two secs to make this commit." The office went silent again whilst Rob's fingers darted across the keyboard. "Ahh. Damn it!" shouted Rob.
>
> Mike rose from his chair and walked over to Rob. "What's up?"
>
> "I just added a file into the staging area, but I don't want it there." He shook his head, "Well not yet anyway."
>
> Mike chuckled, "Sorry for interrupting dude."
>
> "Nah, it's OK, I just need to know how to pull this file out of the index."
>
> "Git reset," shouted a voice. The stillness of the office was interrupted by a chair free wheeling across the floor. The occupant of the chair was Klaus. He seemed proud that he was finally getting to grips with things. "You can use git reset to reset a file that's in the index." He grabbed at the keyboard, "Here, lemme show you."

The `git reset` command is great at removing things from the index that you don't want to be there. Of course, it can do a great many other things, but for now, let us concern ourselves with the scenario presented above. We are working away, and have added a number of files into the index ready for committing, when we discover that we are actually not ready to commit them. In the following example, we are going to add the file my_third_committed_file and then remove it from the index.

```
john@satsuki:~/coderepo$ git add my_third_committed_file
john@satsuki:~/coderepo$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# new file:   my_third_committed_file
#
john@satsuki:~/coderepo$
```

Notice how my_third_committed_file is now ready to be committed to repository. The problem is we need to add something more to it before we do. Remember that when we run the git add command, we are copying the file from our working copy to the index. If we decide we no longer want that file in the repository, we can run the following.

```
john@satsuki:~/coderepo$ git reset my_third_committed_file
john@satsuki:~/coderepo$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
# my_third_committed_file
nothing added to commit but untracked files present (use "git add" to track)
john@satsuki:~/coderepo$
```

We have discarded the file which was residing in the index. This is very important to note. We are not moving the file from the index back into our working directory, we are literally just deleting the file from the index. Our working copy remains unaffected. We could run the `git reset` command without appending a file. If we did this, all the files in the index would have been discarded.

> **In the trenches...** "So, I think we are all agreed, I'll keep a version of the repository under Git version control. Until everyone else feels comfortable with some of the more advanced features." John looked around the room for any disagreements but there were none.
>
> "Agreed John," said Markus, "I'm pleased with how you guys are progressing, very pleased, but like John said, it's far better for us to take our time and to implement this correctly, than to rush it and to end up with something that we can't administrate and that we don't know how it works."
>
> "So next week, I want you all to start playing with diffing and logs and don't forget we have an important release due too." John pushed his glasses further up his nose. "The week after that we'll start looking at branching and by then we may be at the stage where we can implement our model."
>
> Everyone nodded in agreement.

Now we know how to add files into the repository. The question is, what do we do if we need to remove a file, or even rename it. Well, git has some commands to help with that. `git rm` and `git mv` delete and move files respectively. Usually when you want to remove files from the repository, or move them, this is how you will handle it, but what

Knowledge

How do we change the commit message editor?

We spoke earlier about the configuration file and how it stores information about our Git instance. Git can use any text editor you require, even a graphical one, though the need rarely arises. As mentioned earlier, Git has a preference lever when talking about configuration. First and foremost it will look in the repositories own 'config' file in the .git folder. Then, it will look in the users ~/.gitconfig file. Finally, Git will look in your distributions own global folder. If we wanted to change the editor that Git would use to modify commit messages, we can either modify the files directly, or run a command similar to the following;

```
git config core.editor "nano"
```

If we want the changes to apply globally, meaning it would affect all repositories we administrate as this user, unless overridden by a repository setting, we would run the following;

```
git config --global core.editor "nano"
```

It is worth noting that you can also use the $EDITOR environment variable to accomplish the same thing. Many people use this in preference to the modifying the Git configuration simply because many other programs honour this setting.

if you have already deleted a tracked file manually? Well, you have two options. You could run `git commit -a`, but remember this will commit all changes to tracked files. You could also run a `git rm <filename>` with the name of the file you have just deleted. Git will then push that change into the staging area ready for commit. The same applies to moving a file

However, it is worth noting something in the way that Git handles renames. Git does not track renames explicitly. This means that by running the `git mv <source> <dest>` command, you are essentially running a Linux `mv` command, followed by the `git rm` on the source file and `git add` on the destination file. Running the `git mv` command is a shorthand way of doing just that. It is worth playing with this to ensure that you understand what is happening. As an exercise, inspect the repository after each

command so that you understand at what point Git recognises your actions as a rename.

We have run through a few basic commands in Git. If you are familiar with version control systems, then possibly the only real difference you will have noticed is that of the staging area. It really is powerful, and allows you to organise and prepare your commits, so that they are both meaningful and coherent.

For Tamagoyaki Inc, their plan to implement version control was far too aggressive. Most of the members of the team had never even used a version control system. When deciding to implement version control, it is essential to ensure that you are doing it for the right reasons. Version control is a tool to help you to keep things in order, but remember tools are nothing without process. It is process that is key to the order.

Summary - John's Notes

Commands

- `git add` - Add files into the index or staging area
- `git commit` - Commit files into the repository, using text editor for commit message
- `git commit -m '<Message>'` - Commit files into the repository, using the command line to supply commit message
- `git commit -a` - Commit all tracked files into the repository that have changed, using text editor for commit message
- `git reset <path>` - Remove file from index or staging area
- `git status` - Show the status of tracked, changed, untracked files

Terminology

- **Branch** - A way of working on the same set of code in parallel without modifications overlapping
- **Commit** - A group of objects and a tree in a Git repository

After Hours Week 2

"A Little Of Git's Internals"

A Look At Plumbing

This After Hours section is going to get a little deep. For some of you it may be more information than you bargained for. However, sometimes, when the worst happens, it is comforting to know that you at least have a basic understanding of what is happening under the hood. These After Hours sections are designed to give you that knowledge. They are for the people who are not just satisfied with knowing that things work, but they want to know *why* things work.

To start with, we are going to use the Git repository that we have been playing with in Week 2 and take a deeper look at what is actually inside a Git repository. Let us view the directory structure, to see what has been created in the `.git` folder.

```
john@satsuki:~/coderepo/.git$ ls -la
total 52
drwxr-xr-x  8 john john 4096 2011-03-31 20:35 .
drwxr-xr-x  3 john john 4096 2011-03-31 20:28 ..
drwxr-xr-x  2 john john 4096 2011-03-31 20:22 branches
-rw-r--r--  1 john john   30 2011-03-31 20:34 COMMIT_EDITMSG
-rw-r--r--  1 john john   92 2011-03-31 20:22 config
-rw-r--r--  1 john john   73 2011-03-31 20:22 description
-rw-r--r--  1 john john   23 2011-03-31 20:22 HEAD
drwxr-xr-x  2 john john 4096 2011-03-31 20:22 hooks
-rw-r--r--  1 john john  208 2011-03-31 20:35 index
drwxr-xr-x  2 john john 4096 2011-03-31 20:22 info
drwxr-xr-x  3 john john 4096 2011-03-31 20:27 logs
drwxr-xr-x 13 john john 4096 2011-03-31 20:34 objects
drwxr-xr-x  4 john john 4096 2011-03-31 20:22 refs
john@satsuki:~/coderepo/.git$
```

branches - Though deprecated now, this folder stores shorthands for git pull, push and fetch commands, by creating a file, the name of which is passed to the command instead of the repository argument.

COMMIT_EDITMSG - This file holds the last commit message that was displayed in the editor.

config - This is the main configuration file for Git. It is the first place git looks for upon invocation. If this file is not present, Git will inspect `~/.gitconfig`. After this, Git will go to `/etc/gitconfig`. The file holds information about the remotes, tracking branches, push configurations and many more items.

description - This is a simple text file which gives a description to a repository when being view via gitweb or similar.

HEAD - This file is a pointer to the parent commit of your current branch.

hooks - Scripts can be placed in here to perform operations at certain points during the commit process.

info - The info folder contains some additional information about the repository

logs - The logs folder holds various logs regarding Git's operation

objects - The is the directory that holds all of the actual files that are stored in the repository. The files are named by their SHA-1 values. Inside the folder are a number of directories which make up the first 2 characters of the SHA-1 value. The remaining portion of the SHA-1 hash is used to name the file.

ORIG_HEAD - Hold the previous SHA-1 hash that HEAD pointed to. This allows certain operations to go back, in the case of failure. (Not present in our listing)

refs - This folder holds the files that contain information about local branches, remote branches and tags.

More files and folders will appear here during the running of the repository as you begin to start using different features in Git.

The most interesting of the folders here is the `objects` folder. This folder as previously described holds all of the objects that are stored in the repository. Now, what do we actually mean by objects. As yet, we have not really defined what an object is. We are going to look now at three types of object that Git places in this directory. These are the commit, tree and blob objects. We need a little more information as the names themselves do not fully describe what the item is.

- **commit** - A commit object is an object that describes a specific point in time. Whenever you perform a `git commit` from the command line, what you are actually doing is creating one of these objects in the repository. This object stores information about the committer, the date, a link to the previous commit object and most importantly a link to the tree object of the current commit.

- **tree** - A tree object defines which files were physically included in the commit when it was added to the database. The tree contains the name of the files that were present in the tree by recording their blob SHA-1 id and their filename. Sub-folders of a directory will be referenced by another tree object. In this way, a tree object will contain references to both tree objects and blob objects
- **blob** - A blob object actually contains the data that resides inside the file. There is one object generated per revision of a file. However, if exactly the same data resides in two separate files, then there will only be one object created and that object will be referenced by two trees. In this way, multiple copies of the same file, or files which have not changed through multiple revisions are not stored multiple times. Even though Git stores a complete snapshot of the file system at every revision.

So, when we commit into the repository we create a commit object. The commit objects houses links to a tree. The tree object contains links to either blobs or trees. Regardless, the structure of a basic repository may look something a little like Figure 1.

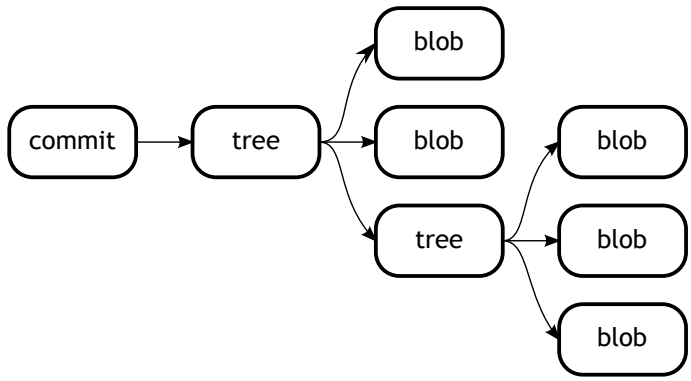


Figure 1: Overview of objects in a repository

So now we know what a basic repository should look like, let us go through each stage of our committing in **Week 2** and see how it is built up at each stage. Below is a consolidated list of all the objects in the repository.

```
./09/5b9cda52807c9c11781ec0a4aee927787b61f1
./16/3f06147a449e724d0cfd484c3334709e8e1fce
```

```
./34/a5dff148e70c12310cda0800d6bcaf82530bdc
./3a/d4cc3fe5a61c5563cb1b2ff3680d7e95be0fce
./3f/fa7ab6dafef2bc38a70a39c53604c333ed4d7a
./8d/664b74cce3a1f24d498d2d2bcc36e9915b5a65
./99/38a0c30940dccaeddce4bb2eb151fba3a21ae5
./cf/e23cbe0150fda69a004e301828097935ec4397
./e6/9de29bb2d1d6434b8b29ae775ad8c2e48c5391
```

In order to start rebuilding a picture of our repository, let us first find out what the first commit to our repository was. Looking back in *Week 2* we can see the following trimmed output.

```
[master (root-commit) cfe23cb] My First Ever Commit
```

So we need to start with the object that begins `cfe23cb`. Remember in the description above about the SHA-1 hashes, the first two bytes are the directory name. We are looking for a file that starts with the characters `e23cb` and it will be in the directory called `cf`. By George! Looking at the list above, we can see that there is one file which fits the bill. In fact it would be a little ambiguous if there were two. It *could* happen that we would have two SHA-1 hashes that started with the same seven characters, but it's not likely. The line we are interested in is listed below.

```
./cf/e23cbe0150fda69a004e301828097935ec4397}
```

It would be nice if we could find out a little more about this object, and confirm that it is what we expect it to be. To start with, let us run the Linux `file` command on it to see what it makes of the contents.

```
john@satsuki:~/coderepo/.git/objects/cf$ file e23cbe0150fda69a004e301828097935ec4397
e23cbe0150fda69a004e301828097935ec4397: VAX COFF executable - version 5185
john@satsuki:~/coderepo/.git/objects/cf$
```

The `file` command, tells us that the file is actually of `VAX COFF executable - version 5185` filetype. This is obviously not correct. Git stores its objects by Zlib compressing them, which is why it is difficult for the `file` command to make any sense out of them. Later in the book, you will find out exactly how to generate your own repository from scratch, but for now, let us just understand that Git first creates a header for the content, then adds the content to the header, creates the SHA-1 hash of such data and finally Zlib compresses it to store to disk.

Out comes the wrench

Wouldn't it be nice if we could view the data that was inside the file simply, without having to write our own tools. If you look through the Git man page, you will find one listed under the **Interrogation** section, called `git cat-file`. Anyone who has spent any time with Linux will know what the `cat` command does. `cat` is used to display the contents of files. In Git, this command is used to display the contents, type or size of a repository object, be it either of type commit, tree or blob.

Information

What's with all the wrenches?

You may have noticed the less than subtle references to plumbing. Git has two types of commands. Those that are readily available to the end user, and those that are not.
The commands that are readily available to the end user are called Porcelain commands because they have been refined and are simple to use, think Tap and Sink.
Commands which get a little down and dirty with the details are called Plumbing commands because they are dealing with the primitive objects that make things work, think Pipes and Joints.

If we run this plumbing command against our object, and supply the `-t` parameter, `git cat-file` will tell us the type of the object. If we run it with the `-s` parameter, it will tell us the size. Finally, running it with the `-p` parameter we can see what the object actually contains. Each of these has been demonstrated below.

```
john@satsuki:~/coderepo$ git cat-file -t cfe23c
commit
john@satsuki:~/coderepo$ git cat-file -s cfe23c
215
john@satsuki:~/coderepo$ git cat-file -p cfe23c
tree 34a5dff148e70c12310cda0800d6bcaf82530bdc
author John Haskins <john.haskins@tamagoyakiinc.koala> 1301599664 +0100
committer John Haskins <john.haskins@tamagoyakiinc.koala> 1301599664 +0100

My First Ever Commit
john@satsuki:~/coderepo$
```

Fantastic! We now have a way of confirming the objects are what we think they are. If we were to run the command against every file in the repository, we would end up with a list that would look something like the one below. Notice our commit object sat in the middle.

```
./09/5b9cda52807c9c11781ec0a4aee927787b61f1     blob
./16/3f06147a449e724d0cfd484c3334709e8e1fce     commit
./34/a5dff148e70c12310cda0800d6bcaf82530bdc     tree
./3a/d4cc3fe5a61c5563cb1b2ff3680d7e95be0fce     blob
./3f/fa7ab6dafef2bc38a70a39c53604c333ed4d7a     tree
./8d/664b74cce3a1f24d498d2d2bcc36e9915b5a65     tree
./99/38a0c30940dccaeddce4bb2eb151fba3a21ae5     commit
./cf/e23cbe0150fda69a004e301828097935ec4397     commit
./e6/9de29bb2d1d6434b8b29ae775ad8c2e48c5391     blob
```

Now things start to get interesting. Did you also notice the tree SHA-1 hash mentioned when we ran `git cat-file -p`? This is the tree object that stores the file structure. Why don't we run the command against that file too?

Note

Packing for a holiday?

It is worth noting that the objects noted above are not the only objects that you will find in the `objects` folder. Git will sometimes clean itself up and perform packing operations. This is out of scope for this chapter though.

```
john@satsuki:~/coderepo$ git cat-file -p 34a5dff
100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391
 my_first_committed_file
john@satsuki:~/coderepo$
```

We can now see a list of all files that were present in that snapshot of the working tree. In our case this just happens to be one file. Using our command one last time against the object name of the file, we can see the actual contents of the file.

```
john@satsuki:~/coderepo$ git cat-file -p e69de29
john@satsuki:~/coderepo$
```

At first glance one may consider it odd that there is nothing in there. However, if we take a trip back in time and go back to the first commit that we made to our repository, you will see that we never actually put any content in our file at all. So, the object contains nothing. Now we can draw up a diagram of our repository after the first commit.

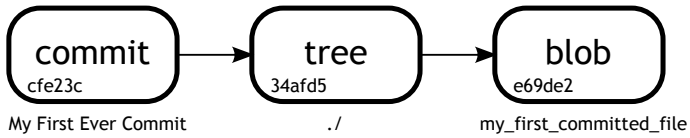


Figure 2: Overview of objects in repository after **first** commit

If we pick another commit object from the list and run our command on it we can see something new has appeared in the output.

```
john@satsuki:~/coderepo$ git cat-file -p 163f061
tree 3ffa7ab6dafef2bc38a70a39c53604c333ed4d7a
parent cfe23cbe0150fda69a004e301828097935ec4397
author John Haskins <john.haskins@tamagoyakiinc.koala> 1301599979 +0100
committer John Haskins <john.haskins@tamagoyakiinc.koala> 1301599979 +0100

Made a few changes to first and second files
john@satsuki:~/coderepo$
```

This commit object has an extra field. This field is the `parent` field. It tells Git which commit object is the parent to this one. We were lucky in our guess that the parent object **cfe23cb** was actually the object we had first encountered. By knowing this nugget of information, we can actually deduce that the object **9938a0c** was actually the next commit. Checking in Week 2 will confirm this. It seems we could now build a complete history of our repository so far. If we did so, we would end up with a diagram that looked a little like the one below.

Hopefully you can see from this diagram that Git will reuse objects as discussed above. This diagram describes the repository layout completely and if you correlate, we have correctly identified and accounted for each object in the repository. Now we know how to traverse a repository. In *Week 3* we find the Git commands to let us do that without having to break out the wrenches and resort to plumbing. After all, why use a wrench to turn on a tap?

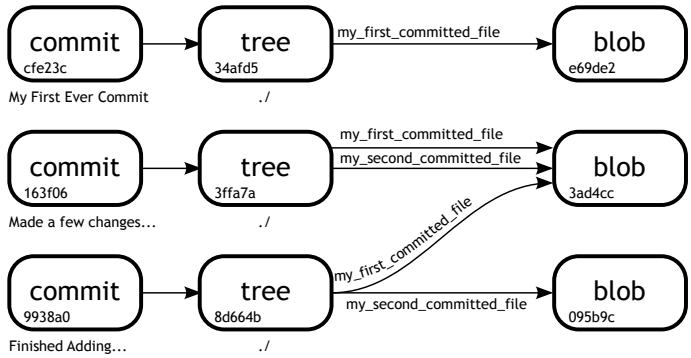


Figure 3: Overview of objects in repository after **three** commits

Week 3

Day 1 - "How do I see what's going on?"

Logging in Git

Perhaps the best feature of a version control system is the level of accountability that it offers if set up correctly. What do we mean by this? People often mistake the word **accountability** for the word **blame**. This is not true at all. Accountability is key in understanding the events that led up to a particular bug being introduced, or a situation occurring. How this is dealt with, is up to the management teams, but accountability should not be something that is revered, it should be something that is looked upon as a tool to help define the cause of a problem.

By its very nature, a version control system is also a logging system. Every time we committed something into the repository in the last chapter, we supplied a log message. In fact, if we don't supply a commit message, let us see what happens.

```
john@satsuki:~/coderepo$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
# my_third_committed_file
nothing added to commit but untracked files present (use "git add" to track)
john@satsuki:~/coderepo$ git add my_third_committed_file
john@satsuki:~/coderepo$ git commit -a -m ''
Aborting commit due to empty commit message.
john@satsuki:~/coderepo$ git reset my_third_committed_file
john@satsuki:~/coderepo$
```

Git will actually not allow you to commit with a blank message. This is actually fantastic news, as people are far less likely to write a useless message than they are a blank one. It is very important that when using a version control system you write in a useful commit message. If you fixed a bug, say so. If you added a new function, why not put that in too. When someone wants to find out what a certain commit was for, or

even when you come back to the project six months later and realise you've forgotten everything, log messages are crucial in piecing back together a history of development.

> **In the trenches...** "So John, I've been committing and all that," started Rob, "but how do I see the history of what I have done."
>
> "It's really pretty simple," replied John, "But it really depends on what you want to know." Rob placed his thumb and forefinger onto his chin. "Well, for now, I just want to see a list of all of my commits."
>
> "That one's the simplest of all."

At its simplest, `git log` will give an output of all of the commits that have been applied to the current branch. Depending on what type of machine you are using it on, the output from `git log` will be navigable, usually using the up and down arrows, with 'q' used to quit. Let's have a quick look at the output of our test repository and see what the log messages look like.

```
john@satsuki:~/coderepo$ git log
commit 9938a0c30940dccaeddce4bb2eb151fba3a21ae5
Author: John Haskins <john.haskins@tamagoyakiinc.koala>
Date:   Thu Mar 31 20:34:23 2011 +0100

    Finished adding initial files

commit 163f06147a449e724d0cfd484c3334709e8e1fce
Author: John Haskins <john.haskins@tamagoyakiinc.koala>
Date:   Thu Mar 31 20:32:59 2011 +0100

    Made a few changes to first and second files

commit cfe23cbe0150fda69a004e301828097935ec4397
Author: John Haskins <john.haskins@tamagoyakiinc.koala>
Date:   Thu Mar 31 20:27:44 2011 +0100

    My First Ever Commit
john@satsuki:~/coderepo$
```

The `git log` command shows us a chronological list of all of the commits to the repository and also gives us several more important pieces of information. In total there are four pieces of information displayed by default.

- **commit** - This is the SHA-1 hash of the commit object that is stored inside the repository. You can find more information about this in the *What's inside the Git repository?* section *Week 2*. This is how we refer to the commit. If someone asked you in what commit you *Made a few changes to first and second files*, you could reply that you did that in commit 163fo. As explained earlier, it is good to remember that you don't need to remember or type out the whole **163f06147a449e724docfd484c3334709e8e1fce**, only the first part is required. Generally, the first five characters will do.

- **Author** - This is the name and email address of the author of the commit. When we begin to look at merging, you will see that the author of a commit, is not necessarily the *committer* of the commit. If you want to find out more about how to set these options, see the breakout box in this Week, called *Changing your identity*.

- **Date** - The date is simply the date at which the commit was created. Again, note that when we start looking at merging, the date will be the date the commit was created, not the date it was merged into the repository.

- **Commit Message** - This is the log message that was added along with the commit when it was created. Hopefully you can now see how important it is to create useful and meaningful messages in here.

Note

Note on Commit IDs

Please note, that if you are following the commits and changes on your local computer, you may not and probably will not have the same commit IDs as are presented in this book. You are advised to use them here to follow what is happening, but to substitute them with your own values, when you start working with the rest of this chapter.

Note

Changing your identity

Particularly when working with other people, or when publishing your repository to a public location, it's a good idea to make sure people know who you are and how to get in contact with you. Every time you make a commit to a repository, Git gives the opportunity to take note of who posted the commit. When you first install Git, it probably won't have the correct information in there for you, so it's important that you take the time to set this up.
To set up your name and email address, we need to modify use `git config` again.

```
$ git config --global user.name "John Haskins"
$ git config --global user.email
"john.haskins@tamagoyakiinc.koala"
```

That's it. Now by default, Git will use this setting whenever you commit to a repository, unless you override it by locally modifying the repository's `.gitconfig`.

Day 2 - "But I need more information"

Digging a little deeper

In the trenches... "I know John, and next time I will make a note of it, but right now, I'd really like to know where this file got changed," Klaus pointed at the piece of paper containing a print out, "specifically when this function was introduced."

John smiled. His hands danced over the keyboard as he finished compiling an email. "And you've no idea when this was added at all?" he asked.

"No, sorry John, I don't." He pondered, "I guess I could write a script to untar all the versions we've created in the last week and search through them." He sighed, "Can't the wonderful Git help us out here?"

A head popped up over the cubicle wall. "You wanna find out when a function was introduced to a file?" It was Rob. "After John showed

> me the basics, I went and read up on it a little. Git has some really powerful searching within the log tool."
>
> "Well come on then," blurted Klaus, "Don't keep me hanging on."
>
> A chime of the popular 1966 hit sprang out in the office.
>
> Klaus pulled a hand down over his face, "Oh don't you all start!"

Git can actually do some rather powerful searching to assist a developer in their daily tasks. It would have been useful if the particular item that was being searched for had been included in the log, but sometimes, things either get missed, or there are just too many changes introduced in one commit to list them all.

In these instances, the `git log -S "<string>"` command comes to our aid. This command will search through the commits in a repository and will return a list of commits which introduced or removed a specific string into the repository. First of all, let's run this against our test repository.

```
john@satsuki:~/coderepo$ git log -S "Change1"
commit 163f06147a449e724d0cfd484c3334709e8e1fce
Author: John Haskins <john.haskins@tamagoyakiinc.koala>
Date:   Thu Mar 31 20:32:59 2011 +0100

    Made a few changes to first and second files
john@satsuki:~/coderepo$
```

You can see that `git log` has shown us the commit that instantiated the change. As you can imagine, when using a large code base, this tool can be invaluable. It allows us to pinpoint a specific moment when a certain string of text entered the repository. When running this against a very large repository, this could take a long time, and so the ability to shrink the search scope down will result in a much faster result. To do this we can append a path to our previous command.

```
john@satsuki:~/coderepo$ git log -S "Change2" my_first_committed_file
john@satsuki:~/coderepo$ git log -S "Change2" my_second_committed_file
commit 9938a0c30940dccaeddce4bb2eb151fba3a21ae5
Author: John Haskins <john.haskins@tamagoyakiinc.koala>
Date:   Thu Mar 31 20:34:23 2011 +0100

    Finished adding initial files
john@satsuki:~/coderepo$
```

If you remember from our committing back in Week 2, we added the string `Change2` to the second file but not the first. So the first time we run this command, it fails, as we are searching against `my_first_committed_file`. The second time we run it, we are searching against `my_second_committed_file` and this is where we see a result. Commit **9938a0c** contains the change we are looking for.

Day 3 - "What actually changed?"

Doing the diff dance

Knowing what the committer thinks they committed is brilliant. However, sometimes it's just not enough. The reason for this is stated fairly precisely in the first sentence of this paragraph, so let us add a little formatting to bring out the real meaning. Knowing what the committer *thinks* they committed is brilliant. By looking at the commit message we only know as much as the committer wants us to. If they are the helpful sort, this will probably be all that we need, most of the time. On the other hand there is always the situation where you'd like to know a little more about what was actually placed into the repository.

The `git diff` command can show us exactly that. For more information about diff in general, see the diff breakout box in this chapter. Think of a diff as an easy way of looking at the differences between two files, surrounded by a little context. This can often be enhanced by a visual diff viewer, but for now, let's stick with our simple text based `git diff`.

If we want to find out what the changes are between our current commit and one of the previous ones, we can write a command like the one below. Notice that below, **163f061** refers to the second commit that we made to the repository.

```
john@satsuki:~/coderepo$ git diff 163f061
diff --git a/my_second_committed_file b/my_second_committed_file
index 3ad4cc3..095b9cd 100644
--- a/my_second_committed_file
+++ b/my_second_committed_file
@@ -1 +1,2 @@
 Change1
+Change2
john@satsuki:~/coderepo$
```

What this is telling us, is that between **163f061** and our current commit **9938a0c**, we added the line *Change2* to the file `my_second_committed_file`. We can see this by

the preceding + on the line `Change2`. Let's make a few changes to our repository and see how the diffs look. We're actually going to make a few changes to the files using a text editor so that you can't see what we've done. Then, hopefully, when we run the `git diff` you'll be able to see clearly what has happened.

```
john@satsuki:~/coderepo$ git log HEAD~1..HEAD
commit a022d4d1edc69970b4e8b3fe1da3dccd943a55e4
Author: John Haskins <john.haskins@tamagoyakiinc.koala>
Date:   Thu Mar 31 22:05:55 2011 +0100

    Messed with a few files
john@satsuki:~/coderepo$
```

The command `git log HEAD~1..HEAD` tells Git to show us the git log for all commits between `HEAD 1` and `HEAD`. The notation used here is something new to us, but seeing as HEAD points to the most current commit, HEAD 1 points to the commit previous to HEAD. This is how we tell Git to show us only the most recently commit.

As it turns out, John Haskins didn't really create a very meaningful log message. *Messed with a few files* is pretty unhelpful in the grand scheme of things. So let's be thankful that this isn't Tamagoyaki Inc's core repository and take a look at what actually happened in the commit **a022d4d**.

```
john@satsuki:~/coderepo$ git diff HEAD~1..HEAD
diff --git a/my_second_committed_file b/my_second_committed_file
index 095b9cd..c9887f8 100644
--- a/my_second_committed_file
+++ b/my_second_committed_file
@@ -1,2 +1 @@
-Change1
-Change2
+Changed this file completely
diff --git a/my_third_committed_file b/my_third_committed_file
new file mode 100644
index 0000000..5d27866
--- /dev/null
+++ b/my_third_committed_file
@@ -0,0 +1 @@
+Addition to the line
john@satsuki:~/coderepo$
```

As you can see, we have several things going on here, so let's take each of them in isolation and see what is going on. We are going to dissect the diff to see what each section means.

```
diff --git a/my_second_committed_file b/my_second_committed_file
```

This first line tells us that we are dealing with `my_second_committed_file`. This is showing that we are comparing the first revision, or a, against the second revision, b.

```
index 095b9cd..c9887f8 100644
```

This second line actually tells us the beginning of the object IDs, as they are stored in the repository. Note that these IDs are not the commit IDs, but the actual blob IDs that Git uses to refer to the file. For more information on this, checkout the *Object's living in harmony* breakout box.

```
--- a/my_second_committed_file
+++ b/my_second_committed_file
```

The next few lines are telling us which is the original file, and which is the new file, so we can use this as a reference.

```
@@ -1,2 +1 @@
-Change1
-Change2
+Changed this file completely
```

Now we see a group of lines which are generally referred to as a hunk. The hunk has two important pieces of information. Section `-1,2` tells us that in the original file, we are looking at the original file (`-`), that the starting line where the change takes place is line 1 (1) and that the hunk applies to two lines (2). The next section tells us that in the new file, the change takes place as line 1, and because the comma and remaining number are omitted, we can infer that the hunk applies to only 1 line.

The three following lines show what change actually took place. Strings `Change1` and `Change2` were deleted from the file, whereas `Changed this file completely` was added to the file.

Looking at the next diff segment, we can see it applies to a different file. Essentially this hunk is no different to the last, the only interesting portion is shown below.

```
new file mode 100644
index 0000000..5d27866
--- /dev/null
+++ b/my_third_committed_file
```

This shows us that `my_third_committed_file` is actually a new file. Notice the `/dev/null` and the `0000000` object ID, indicating that there was no original file.

Diffing Over A Range

All the operations that we have performed so far have been on one commit. Whilst important and valuable, it may be that you want to see an entire range of changes.

> **In the trenches...** "I'm still not entirely convinced about this John," said Martha. "I've been playing around with Git, like you asked me, but it still just seems like we're replicating the work that we used to do with the readme changelogs and the tarball files."
>
> She sat down on a near-by chair and wheeled it over to John's desk. She surveyed the desk for an inch of vacant real estate before finally resting her elbow on the corner of his desk next to a copy of Pro Git.
>
> "Well, actually Martha, I can see exactly what you mean. Up until now, there is no difference between the old and the new process. I'm still in control of all the versions, so nothing has really changed." He thought long and hard, "Tell ya what. Why don't you give me an operation that you've always wanted to do against our code tree tarballs easily."
>
> "Easy," she snapped back, "I want to know what changes were made for the last two weeks whilst I had been away on holiday." She smiled an almost mischievous smile as she referenced 'The Incident', as it had become known throughout the office.
>
> "Easy," John quipped, mimicking her mannerisms. The two broke out in laughter. "We can use git log for that, and I think there are some date options too. Let me check the man page."

Looking at the man page for git log is a mind trip for the uninitiated. Weighing in at over 600 lines of text, it is abundantly clear that this tool does a whole lot more than viewing a simple history of commits to the repository. It is well worth taking the time

to read through the current available options by typing `man git log` on the command line. If you have the documentation installed, this will yield the *man* page for `git log`.

Listing all commits in our repository is useful, being able to filter this output is fantastic. This is one area in which the developers of Git have placed a great deal of time and effort. For example, we can use `git log` to not only show us the commit message, but also provide a diff output as well. This means that for each commit entry in the output, we will see a diff as well. Now, whilst we are further empowered by having the diff output in chronological order for each commit, we can take things further by filtering the commits.

Suppose we want to view all the commits that we made in the last week, typing the following into the command line in our test repository yields the following result.

```
john@satsuki:~/coderepo$ git log -p --since="last week"
commit a022d4d1edc69970b4e8b3fe1da3dccd943a55e4
Author: John Haskins <john.haskins@tamagoyakiinc.koala>
Date:   Thu Mar 31 22:05:55 2011 +0100

    Messed with a few files

diff --git a/my_second_committed_file b/my_second_committed_file
index 095b9cd..c9887f8 100644
--- a/my_second_committed_file
+++ b/my_second_committed_file
@@ -1,2 +1 @@
-Change1
-Change2
+Changed this file completely
diff --git a/my_third_committed_file b/my_third_committed_file
new file mode 100644
index 0000000..5d27866
--- /dev/null
+++ b/my_third_committed_file
@@ -0,0 +1 @@
+Addition to the line
...
...
```

Notice we get to see the diff that was presented before when we ran our `git diff HEAD~1..HEAD` command, but this time, as we have used the git log command instead, we get to see the diff output as well. This is what the `-p` flag is for. Take note of the way

we have specified the time period that we are interested in. The section `–since="last week"` tells Git to filter the output and show only the entries that were committed within the last week.

This type of filtering can be exceedingly useful to a developer. Often when problems arise, you do not have a defined point in time that you know when your code was last working. However most of the time, you can say with some certainty, "I know it was working two weeks ago". Using the methods described above, will give the user all of the changes, categorised by commit, that occurred in those two weeks, allowing them to narrow down the scope of exactly where to begin looking for the offending changes.

If the developer can further categorise the issue, such as, "I know which file the change must have occurred in", then the following example will demonstrate just how easy it is to filter the results even further. Even in the simplified example repository that we have been using, running this command filters the output to a single file.

```
john@satsuki:~/coderepo$ git log -p --since="last week" -- my_second_committed_file
commit a022d4d1edc69970b4e8b3fe1da3dccd943a55e4
Author: John Haskins <john.haskins@tamagoyakiinc.koala>
Date:   Thu Mar 31 22:05:55 2011 +0100

    Messed with a few files

diff --git a/my_second_committed_file b/my_second_committed_file
index 095b9cd..c9887f8 100644
--- a/my_second_committed_file
+++ b/my_second_committed_file
@@ -1,2 +1 @@
-Change1
-Change2
+Changed this file completely

commit 9938a0c30940dccaeddce4bb2eb151fba3a21ae5
Author: John Haskins <john.haskins@tamagoyakiinc.koala>
Date:   Thu Mar 31 20:34:23 2011 +0100

    Finished adding initial files

diff --git a/my_second_committed_file b/my_second_committed_file
index 3ad4cc3..095b9cd 100644
--- a/my_second_committed_file
+++ b/my_second_committed_file
```

```
@@ -1 +1,2 @@
 Change1
+Change2

commit 163f06147a449e724d0cfd484c3334709e8e1fce
Author: John Haskins <john.haskins@tamagoyakiinc.koala>
Date:   Thu Mar 31 20:32:59 2011 +0100

    Made a few changes to first and second files

diff --git a/my_second_committed_file b/my_second_committed_file
new file mode 100644
index 0000000..3ad4cc3
--- /dev/null
+++ b/my_second_committed_file
@@ -0,0 +1 @@
+Change1
john@satsuki:~/coderepo$
```

See how easy that is. Note, the – is necessary to tell Git the following string is a path. We no longer have the information for `my_third_committed_file` present in the output. We have filtered everything out but the information we are looking for. When you are up against deadlines, pouring through pages and pages of diffs and changes can be incredibly time consuming. Having the tools available to cut that output down to just the relevant material can be life saving.

Day 4 - "Finding a good reference point"

Tag you're it!

During software development, a project will generally get to a point where it is ready to be released to people outside of the development team. When this grand day occurs, it is crucial that both the developers and the users have a reference point with which to refer to the state of the code. Having a code name or a version number means that within a very short period of time, both parties can converse about a problem, safe in the knowledge that they are on the same page.

In most version control systems, the word tagging is used to describe a reference point in the code's history. A tag will usually refer to a single commit and labels that particular commit with a name that is easier to remember than a standard version

number or SHA-1 hash. The tag name used can be a codename, or a version number. Often people will follow a simple numbering scheme, like `v1.9`.

In this example, the `1` may refer to the major version number, and will denote a family of versions, often only changing a few times in a projects lifetime. The `9` is a minor version number and can refer to a much more frequent release schedule. You may also see textual items being appended to this version string, like `rc`, `b`, and `a`, denoting *Release Candidate*, *Beta* and *Alpha* respectively.

Git implements tags in a very elegant way. A tag is simply a label in Git that points to a single commit object. The tag name can then be used in place of the SHA-1 to refer to a point in the repositories history. Due to the simplicity of tags, it is also possible and very simple to tag a commit that occurred way into the past. Let us take a look at a simple tag example. We will make the current point in the repository `v1.0a`.

```
john@satsuki:~/coderepo$ git tag v1.0a
john@satsuki:~/coderepo$
```

On its own, this output from `git tag` doesn't really tell us much, but running the following, shows us a little more information

```
john@satsuki:~/coderepo$ git rev-parse v1.0a
a022d4d1edc69970b4e8b3fe1da3dccd943a55e4
john@satsuki:~/coderepo$
```

By running the `git rev-parse` with the tag name `v1.0a`, Git has returned us the SHA-1 hash of the commit we were referring to. If we look back up at the earlier output, we can see that the most recent commit into the repository was indeed **a022d4d1edc69970b4e8b3fe1da3dccd943a55e4**. To give us something to work with, let's tag the commit **163f061** with the tag name `v0.9`.

```
john@satsuki:~/coderepo$ git tag v0.9 163f061
john@satsuki:~/coderepo$
```

Now, we can do the following;

```
john@satsuki:~/coderepo$ git diff v0.9..v1.0a
diff --git a/my_second_committed_file b/my_second_committed_file
index 3ad4cc3..c9887f8 100644
--- a/my_second_committed_file
+++ b/my_second_committed_file
@@ -1 +1 @@
```

```
-Change1
+Changed this file completely
diff --git a/my_third_committed_file b/my_third_committed_file
new file mode 100644
index 0000000..5d27866
--- /dev/null
+++ b/my_third_committed_file
@@ -0,0 +1 @@
+Addition to the line
john@satsuki:~/coderepo$
```

Notice that instead of using the dynamic reference HEAD, we have now used the tag names `v0.9` and `v1.0a` to refer to our previous commits and have returned the combined diff output of all the changes which occurred between the two.

You can find out more about tags and how to specify more information in the *After Hours* section at the end of this Week.

Note

A little about tags

Tags are great and you should most definitely use them in your repositories to make good reference points, but there is one point that you should always remember. Though we have not yet delved into the realms of remote branches, it is important to note that once you have tagged a certain commit, and pushed that to the repository, you cannot then remove that tag if someone else clones, or pulls your tags from that remote branch. This will all become clearer during the next week, but fits in with the overall ethos of working with version control systems, if someone has seen the past, you should not EVER change it.

Day 5 - "Putting things back again"

Revert, I say. Revert!

Whilst working with your repository, something occurs quite often, is the need to go back in time, either temporarily or permanently, or even partially. Git allows you to do this in a multitude of ways. Let's see a real life situation where this need could arise.

In the trenches... "No, I don't have a copy of the file. I was stupid and after I submitted it to you I er... deleted it".

John gave Michael the raised eyebrow look. It wasn't the first time Michael had come to him with a similar problem. Usually John would have had Michael go rooting through the archives to find it. This time though, he wondered if Git might just come to the rescue.

"Tell ya what Michael," he grinned, "Since this isn't the first time you've come to me with this kind of predicament, why don't you go and find out how to use Git to get the file back." Michael sighed. "I have tagged the repo each time we created an archive, so tell me what I need to run to extract it."

* * *

"Man", started Michael running over to John's desk forty five minutes later. "I never knew there were so many ways to skin a cat"

Michael was a little out of shape and though he had only crossed a minor distance, he now stood there, leaned over John's desk ever so slightly out of breath.

"So, you learn much?" asked John.

"Where d'ya want me to start?"

Where exactly do we start? Well one of the neat things about Git is that there are many ways to produce the same result. While that may not seem like a benefit now, the trick is knowing just how to use each tool and what the benefit is of each method. Right now, we are ready to look at four methods for achieving the task of viewing old information in the repository. So how do we choose which method we wish to use? We need to answer a few more questions before we are ready to decide.

The table below shows the three methods that we have access to now. Note that this may not be a definitive list of methods, but that these can give us access to the data we need to view. The columns are requirements or criteria. We need to evaluate each command in order to determine which one is right for us. Once you have been using Git a while, these kind of evaluations will become second nature to you, but right now, we will take a look at all available options, just to see what is out there.

Method Name	Alters Repository	Changes History	Alters Working Copy	Reversible	Multiple Files
Reset	Possibly	Possibly	Possibly	Possibly	Yes
Checkout	No	No	Yes	No	Yes
Show	No	No	No	N/A	No

Let's take a look at each of these in turn. We are going to be covering two new commands and revisiting an old one. Let us start with `git reset`. We have already met this tool once. When we used it previously, its purpose was to pull files out of the index that we were not ready to commit. We were using `git reset` in its simplest state. Actually Git can perform several other kinds of reset. It should be noted here that using this can be quite dangerous as it can affect your index, your working copy, your branch and the pointer HEAD.

In order to use `git reset` in any sane way to achieve our goal, we would need to look at branching, which at the moment, we are not ready to do. In short `git reset` can drastically effect your working copy, affecting multiple files, and before we begin investigating, we really need to learn how to play in a safe environment.

The next method on our list to discuss is the `git checkout` command. This command can be used to bring back either a single file or multiple files and once again at this stage, is best employed in conjunction with branches. At this point, you may be wondering why we are placing such emphasis on the use of branches. As you will see next week, branches are incredibly powerful things, which allow you to experiment and play with your data, without the risk of losing anything. `git checkout` will pull files from a previous commit into our working copy. This is something remember. If we have any changes in our working copy, the checkout will fail.

The last method we can use to view data which was in a previous commit, is the `git show` command. This command literally pulls data from a previous version and dumps it to the standard output, a little like the `cat` command present in almost every single

*nix environment.

Now that we have taken a quick look at our three methods, we must decide which one is going to be the most useful to us. Looking at the scenario above, we can deduce that we really only need to pull out one file. If our intention was to do large amounts of work on an old branch and pull many files from it, `git reset` may have been a good choice. As we are looking for only a single file, we should consider looking at the checkout and show tools.

So now let us see how we can use `git checkout` to take one of our files back to the past.

```
john@satsuki:~/coderepo$ git status
# On branch master
nothing to commit (working directory clean)
john@satsuki:~/coderepo$ git checkout v0.9 -- my_second_committed_file
john@satsuki:~/coderepo$ cat my_second_committed_file
Change1
john@satsuki:~/coderepo$ git checkout HEAD -- my_second_committed_file
john@satsuki:~/coderepo$ cat my_second_committed_file
Changed this file completely
john@satsuki:~/coderepo$
```

```
john@satsuki:~/coderepo$ git status
# On branch master
nothing to commit (working directory clean)
john@satsuki:~/coderepo$ git checkout v0.9 -- my_second_committed_file
john@satsuki:~/coderepo$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# modified:   my_second_committed_file
#
john@satsuki:~/coderepo$ git checkout HEAD -- my_second_committed_file
john@satsuki:~/coderepo$ git status
# On branch master
nothing to commit (working directory clean)
john@satsuki:~/coderepo$
```

Notice how we first checked that we didn't have any local modifications by running the `git status` command. Then we are safe to run the `git checkout` command. We

used the `v0.9` tag from earlier to refer to an earlier commit state. The next part of the command is the double hyphen – that tells Git that what comes after it is the path. Finally we choose `my_second_committed_file` as the source file. After this, when we `cat` the file, we see that it has changed to what it used to be in `v0.9`.

We then switch the file back to the latest version by using the `HEAD` reference. Note that on the odd occasion, the `HEAD` reference doesn't always point to where you think it does, but this is an area we are yet to cover. Then we run the command one more time, but this time we intersperse it with a `git status` to see that there are changes made to out local working copy.

Show me the money

The `git show` command will have largely the same effect, except it grabs us the data without having to change existing files in our working copy. Let us view a quick example.

```
john@satsuki:~/coderepo$ git show v0.9:my_second_committed_file
Change1
john@satsuki:~/coderepo$ git show v0.9:my_second_committed_file > temp_file
john@satsuki:~/coderepo$ cat temp_file
Change1
john@satsuki:~/coderepo$
```

The format of the `git show` command is rather similar to the checkout command we used a few moments ago. The only difference is the presence of the colon, instead of the double hyphen. Notice how the effect of the first command is just to print out the contents of the requested file to the screen. With the Linux environment it is easy to pipe this output to a new file. In the example above, we pipe the output using the `>` character to the file called `temp_file`

Hopefully you can now see that there are often several ways to achieve the same result and it is important to ensure that you choose the right tool for the job. The reset command was too dangerous to use, the checkout command modified our working copy, but the show tool allowed us to create a new file, guaranteeing that our working copy remained untouched.

> **In the trenches...** "So, if I am currently have changes to the file you want, in my local repository," began John, "What command would you recommend I use?"

Michael paused, clearly considering each method in his head. The noise from the sandwich van's horn rang through the office and Michael immediately stood bolt upright and looked panicked. "The van John" he stuttered, "The van"

"You can go to the van when you tell me which command I should use." John smirked. Michael was one of the more junior members of the team and the managers often took the opportunity to haze him.

"I'm gonna go with git show," he said in a rush.

"Why?" asked John.

"So you don't harm the working tree." replied Michael smoothly, already walking out the door.

"You could have also branched," shouted Rob, who was a few steps ahead of him.

* * *

"So, what's the status then John?" asked Markus.

John pressed a button on his laptop and the slideshow on the screen advanced to show an organisational model. "Well, we've not had a whole lot of time this week as the release for project Manta, but we've managed to look at logging and diffing, which is something that we really needed to cover. Klaus also showed everyone how to tag things and went through our version numbering system again as several people had forgotten." Everyone in the room looked at Jack. "We also found out about how to pull older versions out of the repository in a variety of ways."

Markus looked pleased, "So, what's next?"

"Klaus?" asked John.

"Next, John put my team in charge of defining and teaching everyone about branching and merging. This is the really important stuff." Klaus took over control of the laptop and clicked onto the next slide, which

> detailed a list of features. "We really need to get a good handle on these topics to be successful. It is key to collaboration"
>
> "Well done team," ended Markus, "Wayne is going to be impressed with this."

We now have a good working knowledge of how to do many key things in Git. Logging and diffing is supremely important for inspecting what changes have occurred in the repository. Though the options here are not an exhaustive list, they should give you a basic understanding of how to use the tools. It is well worth looking at the man pages for these commands to get an idea of just how expansive they can be. For example, the diff tool can not only show you differences between your working copy and the index, but also between your index and the latest commit, using the `cached` option.

Next we move on to branching and merging. Branching can be a tricky subject, so it is important to understand what is happening at the repository level. It would be prudent to look over the *After Hours* section for Week 2 before continuing as some of the terminology may be a little confusing otherwise.

Summary - John's Notes

Commands

- `git log` - Return a navigable list of commits to a repository
- `git log -S "<string>"` - Show all commits that either introduced or removed a particular string from the repository
- `git log -S "<string>" <path>` - Show all commits that either introduced or removed a particular string from the repository, but restrict the search to a specific path
- `git log HEAD~1..HEAD` - Show all commits between HEAD 1 and HEAD, essentially the last commit
- `git diff HEAD~1..HEAD` - Show the actual differences between HEAD 1 and HEAD
- `git tag <name>` - Create a tag with the given name
- `git tag <name> <commit>` - Retrospectively tag a commit with a given name
- `git rev-parse <tag>` - Show the commit SHA-1 hash object referred to by the given name

Terminology

- **Branch** - A way of working on the same set of code in parallel without modifications overlapping
- **Diff** - Shows the actual differences between files
- **Hunk** - A section of a diff output

After Hours Week 3

"A Closer Look At Diffs and Tags"

The Diff Utility

We learnt in *Week 3* how to work with a diff, and what a diff actually represents. It is interesting to note how old the `diff` utility actually is and how it works. The diff algorithm was developed in the early 1970s and the research published in 1976, by Douglas McIlroy, who wrote the original diff utility and James Hunt. The algorithm we use today to perform diffs has become known as the Hunt-McIlroy after the research papers authors.

In essence the task of calculating a diff is that of finding the differences between two files on a line by line basis. Mathematically, this can be described as the LCS or Longest Common Subsequence problem, which is a classic computer science problem.

Though we are not going to go into the problem in great detail, it is useful to know what actually happens at this level. Essentially you have two sequences, for now we are going to simplify the problem and work on a string of letters.

Old string:

```
a b c d e j k l m p r s
```

New string:

```
a c d f g h i j n o p t u
```

The challenge is to find the longest sequence of items that is present in both of the strings above. This new sequence is found by deleting items from the first and second set until all that remains is a sequence of common items.

In our example case, this is as follows LCS string:

```
a c d j p
```

Comparing this to each of our strings above, it is easy to generate a diff. If the items are present in the *Old string*, but not in the LCS string then they must have been deleted.

Conversely if they are present in the *New string*, but not the LCS, then they must have been additions. Putting this into practice in our example and marking deletions with - and additions with +, we get the following:

Diff string:

```
b e fghi klm o rs tu
- - ++++ --- + -- ++
```

The actual algorithm for generating the LCS and the subsequent diff is too complex to describe here and is out of the scope of this book. This section was included to give you some idea of how Git performs some of its actions internally.

More about tags

Tags can actually do a little more than just hold a single identifier to a specific commit. A tag can also have a log message with it, similar to the commit objects we discussed earlier. In order to invoke this option, we need to use the `git tag -m 'message'` option. This will allow us to supply a message to be stored along with the tag. Let us see how this works in practice.

Firstly we can use the `git tag` command to show all tags that are currently in the repository.

```
john@satsuki:~/coderepo$ git tag
v0.9
v1.0a
john@satsuki:~/coderepo$
```

Now let us create a new tag and give it some extra information.

```
john@satsuki:~/coderepo$ git tag v1.0b -m 'This is an annotated tag'
john@satsuki:~/coderepo$
```

Unfortunately Git was not particularly forthcoming with information on the creation of the tag. On saying that, it is not difficult for us to use the `git show` command to see what we have done.

```
john@satsuki:~/coderepo$ git show v1.0b
tag v1.0b
Tagger: John Haskins <john.haskins@tamagoyakiinc.koala>
Date:   Thu Mar 31 23:55:50 2011 +0100
```

```
This is an annotated tag

commit a022d4d1edc69970b4e8b3fe1da3dccd943a55e4
Author: John Haskins <john.haskins@tamagoyakiinc.koala>
Date:   Thu Mar 31 22:05:55 2011 +0100

    Messed with a few files

diff --git a/my_second_committed_file b/my_second_committed_file
index 095b9cd..c9887f8 100644
--- a/my_second_committed_file
+++ b/my_second_committed_file
@@ -1,2 +1 @@
-Change1
-Change2
+Changed this file completely
diff --git a/my_third_committed_file b/my_third_committed_file
new file mode 100644
index 0000000..5d27866
--- /dev/null
+++ b/my_third_committed_file
@@ -0,0 +1 @@
+Addition to the line
john@satsuki:~/coderepo$
```

Notice that as well as showing us the tag itself, the `git show` command also gave us a diff of what exactly changed in this commit. Small details like this are what makes Git in particular a joy to use for developers.

If we found that we had actually created the tag incorrectly, we have two options, we could use the `-d` option to delete a tag, or we could use the `-F` option to forcibly overwrite a tag with the same name with different information. However please remember the warning about tags in *Week 2*. It is very dangerous to go changing tags, especially if you have already pushed them out somewhere where other people can grab them from.

Now that we have learnt a little about how to play with tags, we should probably take a look under the hood. This is an After Hours section after all.

The implementation of tags in Git is very simple indeed. We are going to jump into the `.git` directory and take a look at a simple output from some commands.

```
john@satsuki:~/coderepo$ cd .git/
john@satsuki:~/coderepo/.git$ ls
branches        config       HEAD   index  logs     refs
```

```
COMMIT_EDITMSG  description  hooks  info   objects
john@satsuki:~/coderepo/.git$ cd refs/tags/
john@satsuki:~/coderepo/.git/refs/tags$ ls
v0.9  v1.0a  v1.0b
john@satsuki:~/coderepo/.git/refs/tags$ cat v1.0a
a022d4d1edc69970b4e8b3fe1da3dccd943a55e4
john@satsuki:~/coderepo/.git/refs/tags$
```

Inside the `.git/refs/tags` folder, there is a file for every single tag in the system. This file contains a single string of characters. That string looks oddly like an SHA-1 commit to me. Using the tricks that we learnt in the last After Hours section, we can interrogate the Git repository, by throwing a few wrenches at it.

```
john@satsuki:~/coderepo$ git cat-file -p a022d4
tree 96551f45496232c0ec6b389731d55fa3d7e1c8fd
parent 9938a0c30940dccaeddce4bb2eb151fba3a21ae5
author John Haskins <john.haskins@tamagoyakiinc.koala> 1301605555 +0100
committer John Haskins <john.haskins@tamagoyakiinc.koala> 1301605555 +0100

Messed with a few files
john@satsuki:~/coderepo$ git cat-file -t a022d4
commit
john@satsuki:~/coderepo$
```

Excellent! Just as we expected. So Git stores the SHA-1 hash for the commit we are referring to, in a file which has the same name as the tag. Just for clarity, let us run the same set of commands against our newly created annotated tag.

```
john@satsuki:~/coderepo$ cd .git/refs/tags/
john@satsuki:~/coderepo/.git/refs/tags$ cat v1.0b
6cbcf47957589bf4b84cc934a26731636d021574
john@satsuki:~/coderepo/.git/refs/tags$
```

Hang on a minute! Shouldn't the output of the `v1.0a` and `v1.0b` files be the same. There were both created at the same point in the repositories history. They were both supposed to be pointing to the same commit. Let us use a little plumbing and see what is going on.

```
john@satsuki:~/coderepo$ git cat-file -t 6cbcf4
tag
john@satsuki:~/coderepo$
```

Interesting. So it seems as if there is another type of object that can be added to the list. The *tag* object. So what exactly is the difference here? To answer that we need to take a look at the difference between the two tags. `v1.0a` was a simple tag, whereas `v1.0b` is an annotated tag.

When the tag was merely a pointer to a commit object, the repository required nothing more that the object that it was referring to, to be stored in the file. Now we have more information, Git treats the information just as it would any other, by creating an object. We can investigate this further and use the `-p` parameter to the `git cat-file` command to see exactly what is stored within this file.

```
john@satsuki:~/coderepo$ git cat-file -p 6cbcf4
object a022d4d1edc69970b4e8b3fe1da3dccd943a55e4
type commit
tag v1.0b
tagger John Haskins <john.haskins@tamagoyakiinc.koala>
 Thu Mar 31 23:55:50 2011 +0100

This is an annotated tag
john@satsuki:~/coderepo$
```

So an annotated tag contains more information than just the commit ID we are referring to. This is pretty handy and later on in the book we will start talking about topics like signed tags, which are required if you wish to verify the identity of the person claiming to have created the tag.

A shorter After Hours section this time. Next time we will move on to taking a deeper look at what happens behind the scenes with branches and merging. Stay tuned.

Week 4

Day 1 - "We're getting somewhere"

Planting trees

Tamagoyaki Inc have now realised that they have to take these things slow and steady if they want to implement a stable and robust system. This week, they are going to start actually using branches and merging in changes, probably one of the largest topics to cover when doing any type of collaborative development.

> **In the trenches...** "Dude!" shouted Eugene from across the office. "Dude!!" he repeated.
>
> No one looked up and the hum of the computers seemed to drown out the murmurs of voices and clicking of keys.
>
> "Dud..." He was cut off by another voice. It was Klaus.
>
> "Maybe if you gave us some indication of who you were addressing Eugene," said Klaus in his usual matter-of-factly tone, "we may actually be able to help you."
>
> "John!" Shouted Eugene, as if ignoring Klaus entirely. The manager hadn't looked up from his monitor as he had advised the tools guy and he now sat there continuing to type with one hand, the other reaching over and yanking out the earbud that was playing a droning beat in John's ear.
>
> "Ouch!" said John, slightly startled.
>
> "Dork wants you," said Klaus, using his pet name for Eugene.
>
> John walked over to Eugene. "Sup?!"
>
> "I don't get branches," said the developer. "I mean I don't get what the heck they do, why I would ever want to use them, and how are they even different to tags anyway?"

First, we should probably start off by answering that very question and describing what branches are and what they can be used for. In Git, a branch is just a pointer to a commit in the repository. At first glance this might not seem any different to a tag. A tag points to a commit, so does a branch. So what distinguishes between the two? Let us start playing with branches a little and the answer will become obvious in a while. Branches allow you to try things out and even keep a history of the things you try without actually affecting your main branch. In essence you are able to take things in a completely different direction, safe in the knowledge that your core code base will be safe.

This is best illustrated by a little demonstration, so we are going to take our testing repository and branch off to try out new, wonderful and wacky things.

```
john@satsuki:~/coderepo$ ls
my_first_committed_file   my_third_committed_file
my_second_committed_file  temp_file
john@satsuki:~/coderepo$ git branch wacky
john@satsuki:~/coderepo$ git branch
* master
  wacky
john@satsuki:~/coderepo$
```

From looking at the output, it would appear that our `git branch wacky` command did not accomplish a whole lot. Running the `git branch` command will give us a list of all branches in the repository. You may have noticed the presence of the * in front of the word `master`. This is telling us that we are on the branch called **master**. Hang on though, did we not just create a new branch called **wacky**?

Well yes we did. However we have not yet switched to it. To do this, we use our good friend `git checkout`. This will change the working copy to reflect the most recent commit in that branch, and will reset our HEAD accordingly, so that it points to this latest commit

```
john@satsuki:~/coderepo$ git checkout wacky
Switched to branch 'wacky'
john@satsuki:~/coderepo$ git branch
  master
* wacky
john@satsuki:~/coderepo$
```

Now the * has moved to be in front of the word `wacky` and we have confirmation from the line above that we have in fact `Switched to branch 'wacky'`. Now we are

here in wonderland, what can we do? Well, potentially anything we could do in our previous branch, but with the added benefit that we are separated. Some of you maybe be thinking, but we never created an initial branch called **master**. Whilst this is true in one sense, the `git init` command actually created this branch for us when we initialised the repository.

We will start off by taking a deeper look at what is present in our branch. Running a `git log` shows us the history of our branch.

```
john@satsuki:~/coderepo$ git log
commit a022d4d1edc69970b4e8b3fe1da3dccd943a55e4
Author: John Haskins <john.haskins@tamagoyakiinc.koala>
Date:   Thu Mar 31 22:05:55 2011 +0100

    Messed with a few files

commit 9938a0c30940dccaeddce4bb2eb151fba3a21ae5
Author: John Haskins <john.haskins@tamagoyakiinc.koala>
Date:   Thu Mar 31 20:34:23 2011 +0100

    Finished adding initial files

commit 163f06147a449e724d0cfd484c3334709e8e1fce
Author: John Haskins <john.haskins@tamagoyakiinc.koala>
Date:   Thu Mar 31 20:32:59 2011 +0100

    Made a few changes to first and second files

commit cfe23cbe0150fda69a004e301828097935ec4397
Author: John Haskins <john.haskins@tamagoyakiinc.koala>
Date:   Thu Mar 31 20:27:44 2011 +0100

    My First Ever Commit
john@satsuki:~/coderepo$
```

You may notice here that the log messages being displayed are identical to that which we had before in our **master** branch. This is nothing to be worried about. You may be wondering how these can be present if we are in a totally separate environment. Well, though we have branched, the history that led us to this point is the same. As we have not made any changes yet, we do not notice any divergence. If we now make changes to the *repository* we can take a look and see how this will affect things. To start with,

let us remove a few files from the working tree, commit these actions, then add a few more, stage them and commit the new files.

```
john@satsuki:~/coderepo$ git rm my_first_committed_file
rm 'my_first_committed_file'
john@satsuki:~/coderepo$ git rm my_second_committed_file
rm 'my_second_committed_file'
john@satsuki:~/coderepo$ git commit -m 'Removed a few files'
[wacky 4a155e4] Removed a few files
 2 files changed, 0 insertions(+), 2 deletions(-)
 delete mode 100644 my_first_committed_file
 delete mode 100644 my_second_committed_file
john@satsuki:~/coderepo$ echo "A new file" > newfile1
john@satsuki:~/coderepo$ echo "Another new file" > newfile2
john@satsuki:~/coderepo$ git add newfile*
john@satsuki:~/coderepo$ git commit -m 'Added two new files'
[wacky 55fb69f] Added two new files
 2 files changed, 2 insertions(+), 0 deletions(-)
 create mode 100644 newfile1
 create mode 100644 newfile2
john@satsuki:~/coderepo$
```

So we have made two new commits to the repository under our new branch. If we run a Linux `ls` command to see the files which are in the working tree, we can see that our working copy has indeed altered. We will also use our `git log` tool to see what the latest commit is.

```
john@satsuki:~/coderepo$ ls
my_third_committed_file  newfile1  newfile2  temp_file
john@satsuki:~/coderepo$ git log -n1
commit 55fb69f4ad26fdb6b90ac6f43431be40779962dd
Author: John Haskins <john.haskins@tamagoyakiinc.koala>
Date:   Fri Apr 1 00:10:49 2011 +0100

    Added two new files
john@satsuki:~/coderepo$
```

Brilliant. As you can see, we have used `git log` in a slightly different way to limit the number of commits. This is what the `-n` parameter is used for. However, what happens if we go back to the **master** branch again? In theory we should have everything back the way we left it just before creating the branch. Let's move back into our **master** branch and examine the state of play.

```
john@satsuki:~/coderepo$ git checkout master
Switched to branch 'master'
john@satsuki:~/coderepo$ ls
my_first_committed_file   my_third_committed_file
my_second_committed_file  temp_file
john@satsuki:~/coderepo$ git log -n1
commit a022d4d1edc69970b4e8b3fe1da3dccd943a55e4
Author: John Haskins <john.haskins@tamagoyakiinc.koala>
Date:   Thu Mar 31 22:05:55 2011 +0100

    Messed with a few files
john@satsuki:~/coderepo$
```

Comparing that to our previous `ls` command, we can see that this is exactly what the working tree looked like at the beginning of the chapter. Let us take a look at a diagram of the commit history to see what has happened in our repository.

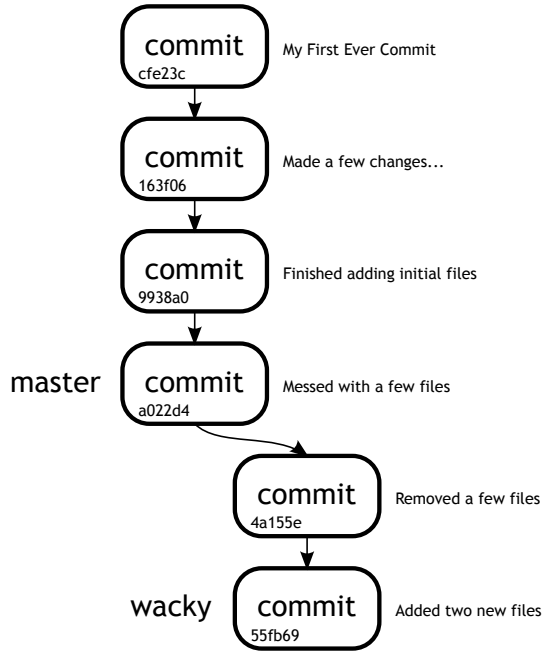


Figure 1: Our first branch

So there are really two pointers in our repository at the moment, from a branch point of view. One of them points to commit **a022d4d**... and is called **master**. The other is called **wacky** and points to **55fb69f**.... At this point you may be thinking that branches are pretty much the same thing as tags. Well, they are except for one important fact. We are going to use our `git branch` command, with a new parameter, to show us the difference. Take a look at the output of the following operations. We have pruned the output of the `git show` command for brevity.

```
john@satsuki:~/coderepo$ git checkout wacky
Switched to branch 'wacky'
john@satsuki:~/coderepo$ git branch -v
  master a022d4d Messed with a few files
* wacky  55fb69f Added two new files
john@satsuki:~/coderepo$ git tag v2.0
john@satsuki:~/coderepo$ git show v2.0
commit 55fb69f4ad26fdb6b90ac6f43431be40779962dd
...
...
```

So here we have added a tag in our current branch, **wacky**, and we can see that the commit ID for the tag **v2.0** points **55fb69f**. We can also see that the branch **wacky** is currently pointing to the same commit ID, **55fb69f**. Now let us add another file in, make a commit and see what happens after this.

```
john@satsuki:~/coderepo$ echo "New stuff" > another_file
john@satsuki:~/coderepo$ git add another_file
john@satsuki:~/coderepo$ git commit -m 'Added another file'
[wacky 9710177] Added another file
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 another_file
john@satsuki:~/coderepo$ git show v2.0
commit 55fb69f4ad26fdb6b90ac6f43431be40779962dd
...
...
john@satsuki:~/coderepo$ git branch -v
  master a022d4d Messed with a few files
* wacky  9710177 Added another file
john@satsuki:~/coderepo$
```

How interesting. The difference between tags and branches now becomes pretty clear. Whilst a tag always points to the same commit, a branch reference always points

to the tip of that branch. In essence the reference that a branch points to moves as subsequent commits are made. By doing this, the whole history of the branch can be retraced. Since we know the latest commit, we also know the parent of that commit and so on and so on.

Since a branch is just a pointer to a commit, performing operations like adding, modifying and deleting files in the repository can be done safely, without destroying any data in another branch. In short, it will allow us to completely redesign whatever is being stored in the current branch without worrying about how it will affect our baseline. For a developer, this is pretty crucial stuff. This gives people the chance to play with their data and experiment, which is often where the greatest ideas come from.

Day 2 - "Branches galore"

Working with branches

Now we know about branches in general, we should really learn about how to merge changes from one branch into another. Branches are fantastic for trying new things out and testing ideas, but if those ideas are successful, we need a way of pulling those changes into our **master** branch.

Of course we could do this the old fashioned way. We could switch into our **wacky** branch, do a little development, copy the files somewhere else, switch to our **master** branch, and paste the files over the top. Now, this is probably the simplest way of merging possible. In actual fact, this is not really merging at all. However one thing that would be lost is the history of how that branch has developed over time. Sometimes this can be crucial for knowing why certain things were changed during the development process.

Let us take a little look at the command output of a simple merge, explain it a little, and then look at a diagrammatic representation.

```
john@satsuki:~/coderepo$ git branch
  master
* wacky
john@satsuki:~/coderepo$
```

Firstly we check just to see what branch we are on. Next we checkout the branch we want our development branch to be merged into. In this case, we want to merge **wacky** into **master** and so we must first checkout the **master** branch. Then we can merge in the changes from our **wacky** branch.

```
john@satsuki:~/coderepo$ git checkout master
Switched to branch 'master'
john@satsuki:~/coderepo$
```

Now we can run the actual merge.

```
john@satsuki:~/coderepo$ git merge wacky
Updating a022d4d..9710177
Fast-forward
 another_file             |    1 +
 my_first_committed_file  |    1 -
 my_second_committed_file |    1 -
 newfile1                 |    1 +
 newfile2                 |    1 +
 5 files changed, 3 insertions(+), 2 deletions(-)
 create mode 100644 another_file
 delete mode 100644 my_first_committed_file
 delete mode 100644 my_second_committed_file
 create mode 100644 newfile1
 create mode 100644 newfile2
john@satsuki:~/coderepo$
```

We can see that the first line after our `git merge` command shows us which commit **master** is the latest common ancestor to both branches and then which commit is the last in our new branch. In this case we are merging from **a022d4d** to **9710177**. The line below this is even more important. This type of merge is called a *fast-forward* merge. We have not made any changes to our **master** branch since we began developing and subsequently, after finishing our development work, we literally only require fast-forwarding the **master** branch to the same point in time as our **wacky** one. Beneath this text, we see more information about just what is included in the merge.

We are going to perform a quick check, to see that we are in fact on the master branch and that the latest log message is the one from the point we last left the **wacky** branch.

```
john@satsuki:~/coderepo$ git log -n1
commit 9710177657ae00665ca8f8027b17314346a5b1c4
Author: John Haskins <john.haskins@tamagoyakiinc.koala>
Date:   Fri Apr 1 00:16:17 2011 +0100

    Added another file
john@satsuki:~/coderepo$ git branch
```

```
* master
  wacky
john@satsuki:~/coderepo$
```

Now let us take a quick look at a diagram to see how this change actually affected the commit flow. We have included tags in this diagram, so that you can see where they point to as well.

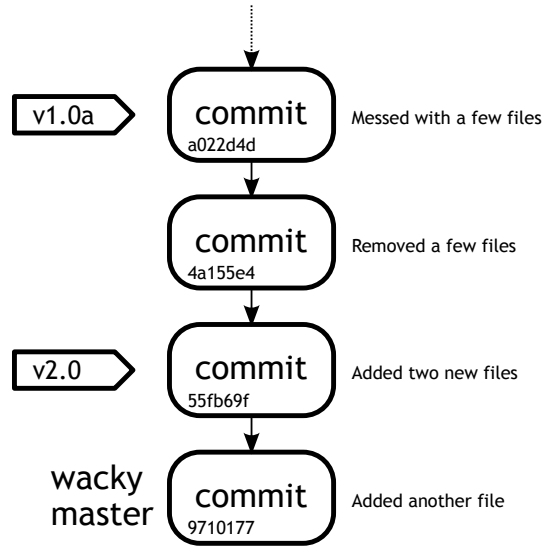


Figure 2: Our first merge

This picture should make it clear that the fundamental difference between tags and branches is that whilst the pointer to a branch moves with each commit to that branch, a tag points to a single commit only and never changes, unless forcibly so by the user.

There are a number of tricks that we can employ when using branches. They are possible because of the super flexible way in which branches are implemented in Git. As a branch is literally a pointer to a commit, certain operations are available to a user that other systems just can not implement. However, we should point something out at this point. Even though we have not yet made any of our repositories public or available to other people, something should always be in the back of your mind. Allow a few minutes to read the next few paragraphs below.

As someone once said, "With great power, comes great responsibility." Git is hugely powerful. However, with this power comes a certain level of responsibility. We are referring here to Git's ability to change history. If you have watched any science fiction films involving time travel, you should be aware of the difficulties and problems often associated with time travel. In Git, the same rule applies and the basics of the rule boil down to this: **If you have made a commit, or path of commits available, you should never ever change anything in the history of those commits or the commits themselves.**

If you are wondering why this is so important, consider this. If you are making a series of films, and you have already released the first two of a trilogy, you would not put elements in the third one that contradict the history of the others. You can not just act like the history of the first two did not happen. Occasionally this happens in the film industry and what is the reaction of the public? People get mad. Sometimes very mad. This is what will happen if you do the same with Git. People who are using your repository will end up with many problems and inconsistencies. **Do NOT do it, ever.** There are ways to revert certain behaviour and we will cover this at a later stage.

Having said this, you should not shy away from the awesome capabilities of Git. We are going to cover a few situations now which you may find yourself in. Some of them do alter history, some of them do not. This is why it is important to have an understanding of how Git works. It can be your best friend, but it can also cause you issues. If you take the time to tame the beast, it will be one of the most awesome tools in your developers tool bag and can save your life time and time again.

> **In the trenches...** "Oh man." The familiar cry of Simon needing something reverberated round the office. Rob could never understand why he didn't ask for help. Simon would sit there wallowing out loud until someone could take it no longer and would eventually go over and help him.
>
> "What to do... what to do."
>
> Rob could take it no more. Simon had been exclaiming now for about five minutes and Rob seemed to either be the only one who wasn't listening to music, or who was getting annoyed. He rolled his eyes, "What's up Si?"
>
> Simon grinned inanely to himself. "I just started modifying some files

> and well . . . I don't want to get rid of them . . . I'm not 100% sure what I've changed. I wish I had started this in a new branch."
>
> "Well, when did you last commit?"
>
> "Just before I started all this work," came the reply.
>
> Rob pointed at Simon's screen. "You can just create a new branch now and move all the changes into it in one go. You can do it in one command actually."

It is one hundred percent true. How many times have you been working on something and wished that you could move all the changes you had made to a new safe environment to protect your already good, working code. Well, the new safe environment we spoke of sounds suspiciously like a branch. In Git, any changes you have in your working copy can be taken into a new branch by issuing one command. It is important to note that these changes can also be taken into an existing branch, but you may run into problems if those changes conflict with items already in that branch.

For now let us see how we can take our working copy changes into a new branch to continue development of some wonderful new feature. We are going to start by making some changes to our newfiles.

```
john@satsuki:~/coderepo$ echo "and some more changes" >> newfile1
john@satsuki:~/coderepo$ echo "and a new feature" >> newfile2
john@satsuki:~/coderepo$
```

Now we are going to make a new branch and switch into it.

```
john@satsuki:~/coderepo$ git checkout -b wonderful
M newfile1
M newfile2
Switched to a new branch 'wonderful'
john@satsuki:~/coderepo$
```

Did you see that? We managed to create a new branch and move into it in one go. The `git checkout` command is usually what we would use to move into a branch, not create it. When we use the `-b` parameter, the `git checkout` command can be used to create a new branch and switch to it in one go. Notice we have chosen the name **wonderful** for this particular branch. There is also a status output below this that shows we have pulled two modifications into this branch, denoted by the letter `M` in front of the file name.

Now we can commit these changes.

```
john@satsuki:~/coderepo$ git commit -a -m 'Fantastic new feature'
[wonderful cfbecab] Fantastic new feature
 2 files changed, 2 insertions(+), 0 deletions(-)
john@satsuki:~/coderepo$ git diff master wonderful
diff --git a/newfile1 b/newfile1
index 24e7dfa..ef20984 100644
--- a/newfile1
+++ b/newfile1
@@ -1 +1,2 @@
 A new file
+and some more changes
diff --git a/newfile2 b/newfile2
index cba16cc..dac4357 100644
--- a/newfile2
+++ b/newfile2
@@ -1 +1,2 @@
 Another new file
+and a new feature
john@satsuki:~/coderepo$
```

After running the diff, we can see the differences between the **master** and the **wonderful** branches. Looking more closely at the hunks, the only differences between the two branches are those that we made before we created our **wonderful** branch. We have achieved what we set out to in bringing uncommitted changes from **master** into **wonderful** and subsequently committing them.

Day 3 - "Tricking the twigs"

More neat ways to work with branches

This is only the beginning of the fantastic feature set that Git offers. By knowing how Git handles your data, you can make it work for you. Let us take a look at another situation you may find yourself in.

> **In the trenches...** "Rob!" shouted Simon for the fourth time that morning. "Rob, I made a real boo boo this time."
>
> Rob walked over to Simon again, head lolling back on his shoulders and his eyes rolling. His arms weighed heavily by his side and his walk reminded Simon of that of a zombie.

> "What have you done THIS time?" asked the developer as he reached the desk.
>
> Simon drew in his breath deeply, "Well, I created a branch, started working away, committing like a good'un, but I only just realised I'm on the wrong branch. I forgot to switch." He hung his head in shame. "I'm screwed right?"
>
> "Not necessarily," replied his colleague.

This wouldn't work in all situations, but then there are other techniques described later in the book to deal with more special cases. In this scenario we have created a branch, forgotten to switch into it, and carried on committing as if we were in our new branch. At first glance it may seem as if we are stuck. Once we have committed, we can't undo those commits right? Well, that's not entirely accurate.

We actually have two ways of clearing up this particular situation. The first of these is to use `git revert` to undo the changes of each commit. Whilst this will work, we have two problems, a) we do not know how to use the `git revert` tool yet, and b) we can actually handle this situation much more cleanly. We are going to make a new branch, make a few commits and then look at a diagram of our recent work to see how we can work things out.

```
john@satsuki:~/coderepo$ git checkout master
Already on 'master'
john@satsuki:~/coderepo$ git branch zaney
john@satsuki:~/coderepo$ echo "and some awesome changes" >> newfile1
john@satsuki:~/coderepo$ git commit -a -m 'Made an awesome change'
[master a27d49e] Made an awesome change
 1 files changed, 1 insertions(+), 0 deletions(-)
john@satsuki:~/coderepo$ echo "and some more awesome changes" >>
 newfile2
john@satsuki:~/coderepo$ git commit -a -m 'Made another awesome change'
[master 7cc32db] Made another awesome change
 1 files changed, 1 insertions(+), 0 deletions(-)
john@satsuki:~/coderepo$
```

Figure 3 shows how our repository looks now, whereas Figure 4 shows how the repository should have looked if we had performed it properly. You should notice that the positions of `master` and `zaney` have switched places. How can we rectify this?

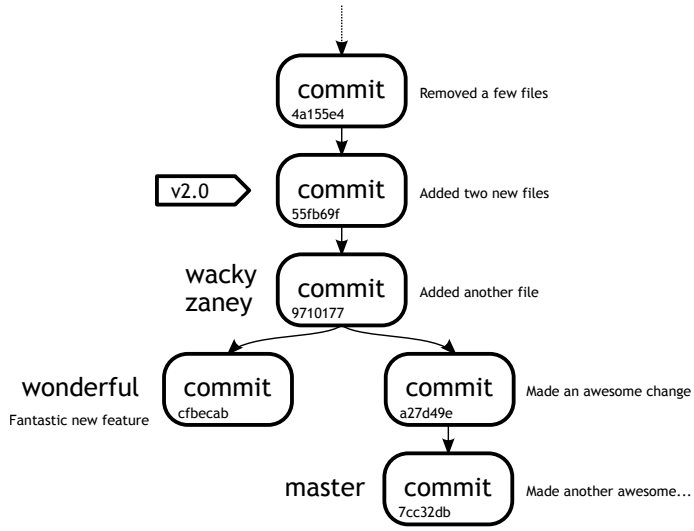


Figure 3: Repository including the mistake

We already discussed one method using `git revert`, which we are due to cover a little later. However, because of the way that the history has been written, we can do something very simple. We are going do the following the following steps.

1. Switch to our **zaney** branch
2. Fast-forward our zaney branch so that it points to the same commit using a merge
3. Switch back to our **master** branch
4. Reset our master branch back to the required point in time

So let us take a look at the command line output and see how we achieve this. Hopefully you should be familiar with most of the commands.

```
john@satsuki:~/coderepo$ git checkout zaney
Switched to branch 'zaney'
john@satsuki:~/coderepo$ git merge master
Updating 9710177..7cc32db
Fast-forward
```

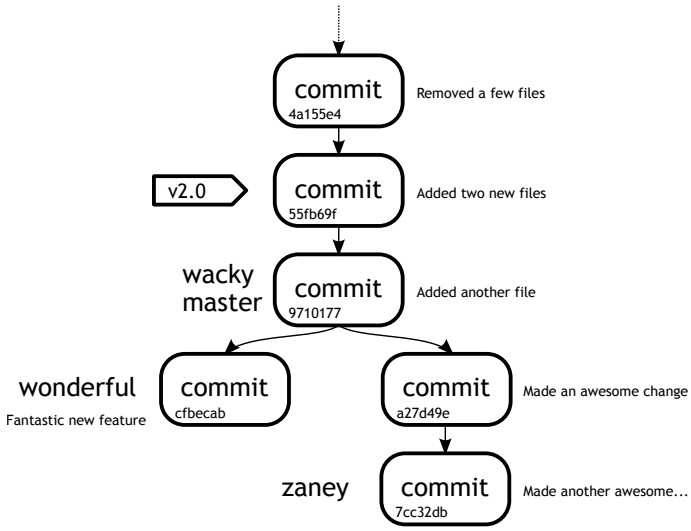


Figure 4: Repository showing how things *should* look

```
 newfile1 |    1 +
 newfile2 |    1 +
 2 files changed, 2 insertions(+), 0 deletions(-)
john@satsuki:~/coderepo$ git checkout master
Switched to branch 'master'
john@satsuki:~/coderepo$
```

Now we reach step four in our set of instructions. The one function we do not know how to perform yet is the resetting of our branch back to a previous point in time. The point we need to rewind back to is the point that we initially created the **zaney** branch at. We could have gotten this information by using `git log`. Instead, this time we can use the information presented in the merge output to show the common ancestor, which has to be the point that we created our branch at. In this case it is commit **9710177**.

We are now going to perform the last step using an old friend called `git reset`. You may be thinking that `git reset` is only used to reset files in the index, but in fact, `git reset` can actually perform many more tasks. We are going to use it with the `–hard` option. This option can be dangerous, as it will discard all modifications in the working tree, so use with caution. If we had uncommitted changes in our repository at this point, we could not have used this option. Let's use the command and see where we get.

```
john@satsuki:~/coderepo$ git reset --hard 9710177
HEAD is now at 9710177 Added another file
john@satsuki:~/coderepo$
```

As you can see, we are told that the HEAD of our master branch is now at commit **9710177**. We have successfully rewound our **master** branch to a previous state. The `–hard` parameter reset the index and the working tree to be at the state of the commit we tell it to. It disregards all working copy and staged modifications, so use it with care.

The `git reset` command does not only work for rewinding back in time. It can also be used to move a branch forward in time. As an example of this, we used a fast-forward merge to move our **zaney** branch forward to be in-line with master. We could just have easily used `git reset –hard 7cc32db` from within the zaney branch, to bring it to the same point as the master. In fact, though it looks scary, we could also have used `git reset –hard master` to reset the **zaney** branch to be at the same point as **master**. Saves typing out those horrid commits does it not?

Finally we are going to introduce one more use of the `git log` command to show us how our repository looks in a semi-graphical way.

```
john@satsuki:~/coderepo$ git log --graph --pretty=oneline --all --abbrev-commit
 ↪--decorate
* 7cc32db (zaney) Made another awesome change
* a27d49e Made an awesome change
| * cfbecab (wonderful) Fantastic new feature
|/
* 9710177 (HEAD, wacky, master) Added another file
* 55fb69f (v2.0) Added two new files
* 4a155e4 Removed a few files
* a022d4d (tag: v1.0b, v1.0a) Messed with a few files
* 9938a0c Finished adding initial files
* 163f061 (v0.9) Made a few changes to first and second files
* cfe23cb My First Ever Commit
john@satsuki:~/coderepo$
```

The `–graph` parameter, tells Git to draw a graph down the left hand column. The `–pretty=oneline` parameter reduces the commit details to one line, else we see the entire log message of the commit. `–all` shows all branches. The `abbrev-commit` in the command tells Git to abbreviate the commit IDs to a sensible length. Finally, `–decorate` shows us the tag and branch references. Hopefully if you compare this tree to the diagram earlier, you will see that the tree is actually completely in order.

Be aware that during this work we have changed the history of at least one of our branches. Had we pushed our changes to a public server, which is something that will be discussed next week, we would have to force these changes to be accepted at the server end. Git knows we are trying to change a past that may have been viewed by others and will warn us accordingly.

Day 4 - "I pressed delete..."

Handling the pressure

We have had some awesome fun working with branches, and hopefully you can see how utterly powerful Git is. Sometimes though we can get ourselves into trouble and it is here that Git can also come to our rescue. Let us learn how to delete a branch. We are going to delete our **wacky** branch now as we have already merged it and no longer require it.

```
john@satsuki:~/coderepo$ git branch -d zaney
error: The branch 'zaney' is not fully merged.
If you are sure you want to delete it, run 'git branch -D zaney'.
john@satsuki:~/coderepo$ git branch -D zaney
Deleted branch zaney (was 7cc32db).
john@satsuki:~/coderepo$ git branch -v
* master    9710177 Added another file
  wacky     9710177 Added another file
  wonderful cfbecab Fantastic new feature
john@satsuki:~/coderepo$
```

Ooops. Despite the firm warning from Git, we have inadvertently deleted the **zaney** branch by mistake. This does happen. When people are in the thick of it, they do make mistakes and it is comforting to know that Git is able to recover from this, but how? We have deleted the branch reference that pointed to the commit that was at the tip of the **zaney** branch by using the `-d` and `-D` parameters. So the question is, does that commit still exist any more? Maybe we did more damage than we thought. By deleting all references to the commit, how do we get it back?

All is most definitely not lost. Even though we have removed all references to the commits in question, they still exist in the repository. They will continue to do so unless we run a clean up on the repository, which will be covered later, or if the items are left to age for more than at least fourteen days. This means that we have up to two weeks

to try to recover the lost commits. Of course in practice one would hope that we would perform the recovery much earlier than that.

We already know the commit ID that the branch was pointing to. Git has been very kind and told us it, just before it carried out the delete. We are looking for commit **7cc32db**. If we run the command below, we will create and recover our branch in one go.

```
john@satsuki:~/coderepo$ git branch zaney 7cc32db
john@satsuki:~/coderepo$ git branch -v
* master    9710177 Added another file
  wacky     9710177 Added another file
  wonderful cfbecab Fantastic new feature
  zaney     7cc32db Made another awesome change
john@satsuki:~/coderepo$
```

Our branch has been restored and points to the same place as it did before we deleted it. As each commit points to its parent, we now have the complete history of **zaney** restored and the branch can be used as normal. To complete this action, let us delete the **wacky** branch as originally intended.

```
john@satsuki:~/coderepo$ git branch -d wacky
Deleted branch wacky (was 9710177).
john@satsuki:~/coderepo$ git branch -v
* master    9710177 Added another file
  wonderful cfbecab Fantastic new feature
  zaney     7cc32db Made another awesome change
john@satsuki:~/coderepo$
```

As you can see, when we deleted **wacky** we were not warned about unmerged changes. This is because the **wacky** branch is at the same point as the **master** branch.

Day 5 - "Conflicting information"

What to do when it all goes wrong

The team have been playing with branches for a few days now and are beginning to settle into the idea of branching often. As you can see, in Git, a branch is a really simple device for allowing experimentation and development. As the implementation of branches is so simple, it is also really fast to switch between branches. If you have been keeping up with the *After Hours* sections, you can hopefully see that when switching branches, Git only needs to alter what is different between one working copy and another.

Let us see what new issues the team runs into during Week 4.

> **In the trenches...** "Hmm," the puzzled noise reverberated round Klaus' corner of the office. Mumbling commenced. "So if I did that, then why is that not ...I mean I didn't think ...that's probably the problem Klaus ...sill 'luser'." He chuckled to himself, hardly noticing the looming figure of John.
>
> "You got a conflict," said John matter of factly.
>
> "I can see that," came Klaus' reply. "The problem is how do I solve it"
>
> "What were you doing?"
>
> "Well I had a few branches I had been working on and I was trying to merge them in."
>
> "Figures," said John.

Conflicts are bound to happen at some point. Even the best project processes in the world will sometimes end up with a conflict that has to be resolved. We should, at this point, clarify what a conflict actually is. A conflict occurs when an attempt is made to reconcile multiple changes to the same section of a file at the same time. This usually means that the same line of code is modified by two different parties at the same time. Now that we have several branches in our test repository, let us merge them back into master and see if we come up with a conflict.

First we are going to add a little change to one of our files.

```
john@satsuki:~/coderepo$ echo "Another update to this file" >> my_third_committed_file
john@satsuki:~/coderepo$ git commit -a -m 'Updated third file'
[master 4ac9201] Updated third file
 1 files changed, 1 insertions(+), 0 deletions(-)
john@satsuki:~/coderepo$
```

Now let us merge in our **wonderful** branch.

```
john@satsuki:~/coderepo$ git merge wonderful
Merge made by recursive.
 newfile1 |    1 +
 newfile2 |    1 +
 2 files changed, 2 insertions(+), 0 deletions(-)
john@satsuki:~/coderepo$
```

So far, our merge has gone pretty well. We have taken all of the changes that were made in the **wonderful** branch and pulled them into our **master** branch. If we wanted to, we could continue on making changes to the **wonderful** branch and we could keep pulling changes in from one to the other as time went by. Figure 5 shows what our repository looks like now.

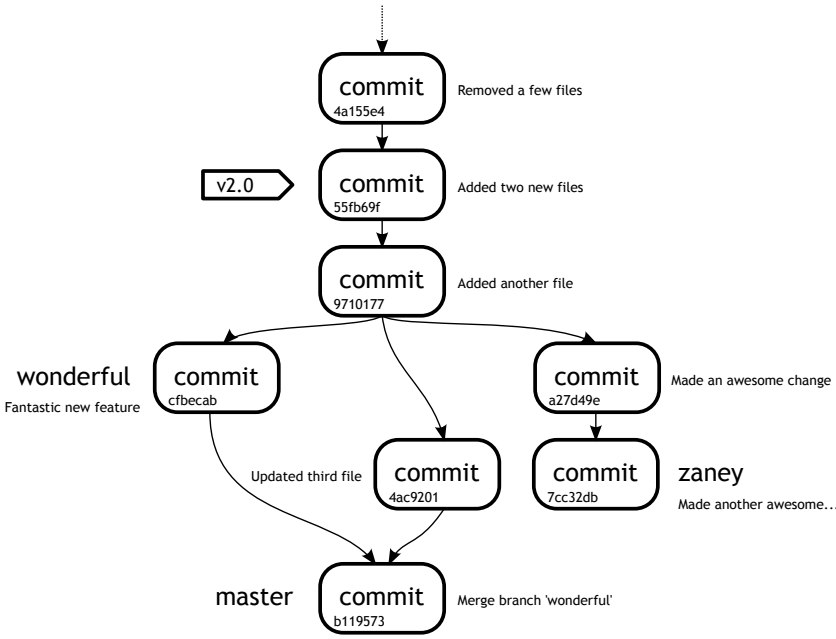


Figure 5: Repository state after **wonderful** merge

Our repository tree looks a little strange now. Notice that the most recent commit, **b119573**, has two parents. Can this be true? If we run our `git show` command against that commit ID, we see something new.

```
john@satsuki:~/coderepo$ git show b119573
commit b119573f4508514c55e1c4e3bebec0ab3667d071
Merge: 4ac9201 cfbecab
Author: John Haskins <john.haskins@tamagoyakiinc.koala>
Date:   Fri Apr 1 07:35:13 2011 +0100

    Merge branch 'wonderful'
```

```
john@satsuki:~/coderepo$
```

Notice the line `Merge: 4ac9201 cfbecab`, which simply tells us that this is a special type of commit, a merge commit. When we previously performed a merge, the branch we were merging back into had not changed since we branched off and did our development work. If you remember, we called this a *fast-forward* merge, simply because we moved our *destination* branch forwards to meet the tip of the development branch.

This time things are a little different. During out development time on **wonderful**, something changed on **master**. We did not make many alterations, but just the fact that we made some changed the way the merge was performed. In fact, we can see this if we look at the first line after the merge. Instead of indicating a *fast-forward* merge, it now states that the merge was *recursive*. If you are interested in the different types of merge algorithms, you should read the *After Hours* section for this chapter.

So now the history of this latest commit actually relies on two previous commits. The changes that have been introduced in these commits are merged together to form a combined tree of files. This is the snapshot that is represented in commit **baobfc**.

Let us now remove that third file before merging in the **zaney** branch.

```
john@satsuki:~/coderepo$ git rm my_third_committed_file
rm 'my_third_committed_file'
john@satsuki:~/coderepo$ git commit -a -m 'Removed third file'
[master ed2301b] Removed third file
 1 files changed, 0 insertions(+), 2 deletions(-)
 delete mode 100644 my_third_committed_file
john@satsuki:~/coderepo$
```

So now we are ready to merge in the **zaney** branch.

```
john@satsuki:~/coderepo$ git merge zaney
Auto-merging newfile1
CONFLICT (content): Merge conflict in newfile1
Auto-merging newfile2
CONFLICT (content): Merge conflict in newfile2
Automatic merge failed; fix conflicts and then commit the result.
john@satsuki:~/coderepo$
```

Oh dear! That probably did not go as well as we had hoped. We have two conflicts. The first is in `newfile1` and the second in `newfile2`. You may be wondering why. The

last commit we made did not make any changes to either of those files, so why do we suddenly have a conflict? When using a version control system like Git, conflicts are an every day part of life. A conflict does not mean you have done something wrong. It simply means that the automatic merging algorithm can not merge the changes.

In this case we have a conflict because the files have changed in both sides of the merge, since they diverged. Rephrasing this slightly: Since the point when we created the branch **zaney**, at commit **55fb69**, both the **master** branch and the **zaney** branch have both modified the same files in the same places. Right now, our merge is paused. For now, let us abort the merge so that we can take a closer look at what happened.

```
john@satsuki:~/coderepo$ git reset --merge
john@satsuki:~/coderepo$
```

So let us compare the changes between our latest common ancestor and our branch tips. We will first look at the difference between the ancestor and our **master** branch, but limit it to the files with conflicts.

```
john@satsuki:~/coderepo$ git diff 55fb69 master -- newfile*
diff --git a/newfile1 b/newfile1
index 24e7dfa..ef20984 100644
--- a/newfile1
+++ b/newfile1
@@ -1 +1,2 @@
 A new file
+and some more changes
diff --git a/newfile2 b/newfile2
index cba16cc..dac4357 100644
--- a/newfile2
+++ b/newfile2
@@ -1 +1,2 @@
 Another new file
+and a new feature
john@satsuki:~/coderepo$
```

Now we will do the same between the common ancestor and the tip of **zaney**.

```
john@satsuki:~/coderepo$ git diff 55fb69 zaney -- newfile*
diff --git a/newfile1 b/newfile1
index 24e7dfa..d94dacc 100644
--- a/newfile1
+++ b/newfile1
```

```
@@ -1 +1,2 @@
 A new file
+and some awesome changes
diff --git a/newfile2 b/newfile2
index cba16cc..45659d7 100644
--- a/newfile2
+++ b/newfile2
@@ -1 +1,2 @@
 Another new file
+and some more awesome changes
john@satsuki:~/coderepo$
```

So hopefully you can see that we have tried to change the second line in both files. This is where the conflict is arising from. Let us try to run the merge again and this time we will resolve the conflicts manually.

```
john@satsuki:~/coderepo$ git merge zaney
Auto-merging newfile1
CONFLICT (content): Merge conflict in newfile1
Auto-merging newfile2
CONFLICT (content): Merge conflict in newfile2
Automatic merge failed; fix conflicts and then commit the result.
john@satsuki:~/coderepo$ cat newfile1
A new file
<<<<<<< HEAD
and some more changes
=======
and some awesome changes
>>>>>>> zaney
john@satsuki:~/coderepo$ cat newfile2
Another new file
<<<<<<< HEAD
and a new feature
=======
and some more awesome changes
>>>>>>> zaney
john@satsuki:~/coderepo$
```

We have displayed the contents of `newfile1` and `newfile2` during this merge. At the moment, no commits have been made and we have a chance to tell Git exactly what these files should look like. You should be able to see that Git has modified the files to show us what both the branches think the file should look like. At the top, just after

the «««< HEAD is how the line was modified in the **master** branch. We then have a divider ======= and after that we have the line, as it was modified in the **zaney** branch, followed by »»»> zaney. We can now edit these files manually, remove the dividers, header and footer and commit the result.

We are going to edit newfile1 to look like this:

```
A new file
and some more awesome changes
```

We will also edit newfile2 to look like this:

```
Another new file
and a new awesome feature
```

Now we have resolved the conflicts and set the files straight, we can add the files and commit the result.

```
john@satsuki:~/coderepo$ git add newfile*
john@satsuki:~/coderepo$ git commit -a -m 'Merged in zaney'
[master d50ffb2] Merged in zaney
john@satsuki:~/coderepo$ git log -n2
commit d50ffb2fa536d869f2c4e89e8d6a48e0a29c5cc1
Merge: ed2301b 7cc32db
Author: John Haskins <john.haskins@tamagoyakiinc.koala>
Date:   Fri Apr 1 07:42:04 2011 +0100

    Merged in zaney

commit ed2301ba223a63a5a930b536a043444e019460a7
Author: John Haskins <john.haskins@tamagoyakiinc.koala>
Date:   Fri Apr 1 07:37:34 2011 +0100

    Removed third file
john@satsuki:~/coderepo$
```

Our merge has been completed. Let us now see what our repository looks like graphically, in Figure 6.

It is quite reasonable that you may want to continue working on branches after you have integrate those changes back into your master. Obviously you would have to merge your **master** branch back into **zaney** or any of the others, otherwise you could be developing on older versions of the code. This brings up an interesting point which will

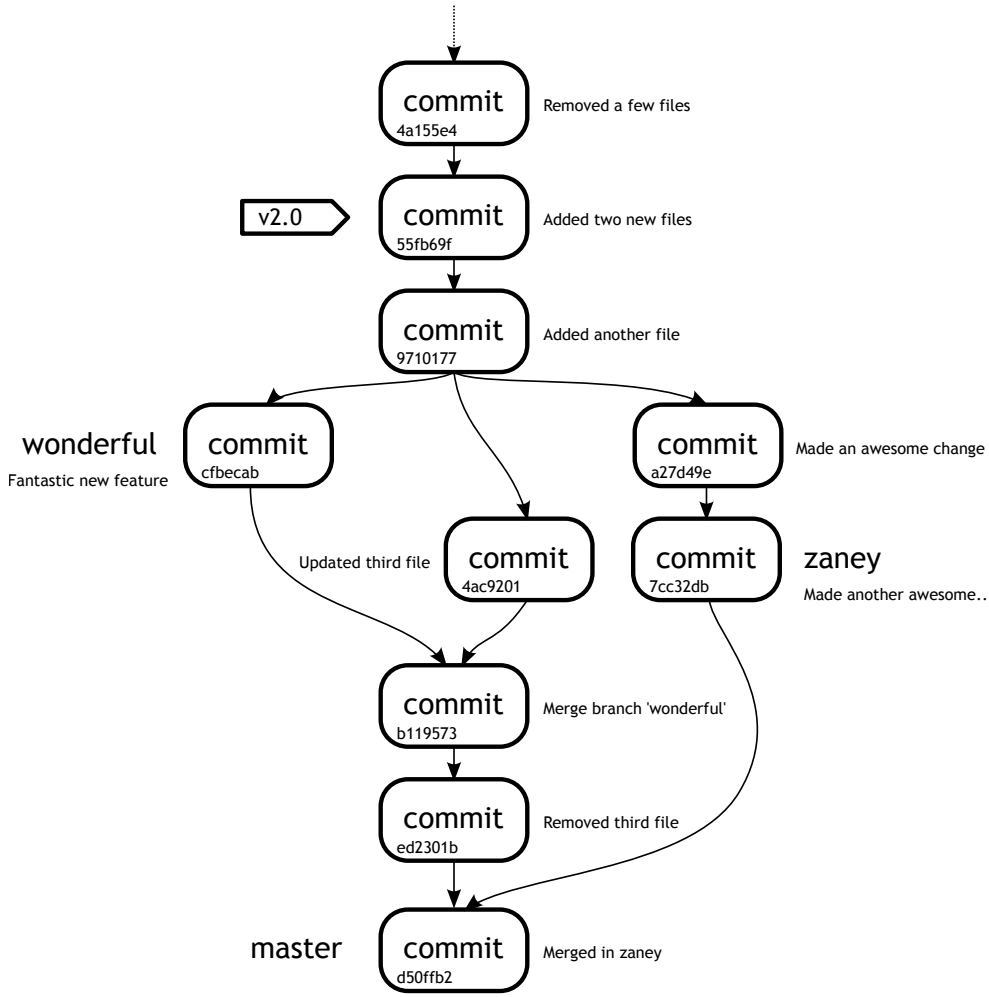


Figure 6: Repository state after **zaney** merge

be covered in a later chapter, as there are multiple ways to keep a development branch up to date.

So, at the end of this week, we have learnt how to create branches, how to merge them, and how to resolve conflicts. It should be stressed that a conflict is not a **bad**

thing. Conflicts do happen and when they do you should simply take your time and resolve the conflict in order to represent the *best of both worlds*. The *After Hours* section for Week 4, takes a closer look at merging and explains some more about what branches are at a repository level.

Summary - John's Notes

Commands

- `git branch <branch_name>` - Creates a new branch with a given name
- `git checkout <branch_name>` - Switches to a new branch
- `git log -n1` - Limits the output of the log command to only a single commit
- `git branch -v` - Verbosely list all local branches
- `git show <reference>` - Show the information pointed to by the reference
- `git merge <branch_name>` - Merge in a given branch to the current one
- `git checkout -b <branch_name>` - Create a new branch and switch into it in one go
- `git reset –hard <commit_ref>` - Reset a branch to a specific commit reference
- `git log –graph –pretty=oneline –all –abbrev-commit –decorate` - Shows a graphical view of the repository
- `git log –graph –oneline –all –decorate` - A shorter form of the above command
- `git branch -d <branch_name>` - Delete a branch that has already been merged
- `git branch -D <branch_name>` - Forcibly delete a branch
- `git branch <branch_name> <commit_ref>` - Create a new branch, starting at the commit reference
- `git reset –merge` - Abort a merge and reset back to pre-merge state
- `git diff <commit_1> <commit_2> – <files>` - Show the difference between the two commits, for certain files

Terminology

- **Conflict** - A conflict occurs when an attempt is made to reconcile multiple changes to the same section of a file at the same time.
- **Fast-Forward Merge** - The simplest merge type, which simply winds a branch forward in time to meet the tip of another.

After Hours Week 4

"Merge merge merge"

How does merging work?

To start with, we need to define which merging strategy we are talking about. In Git there are multiple ways to instruct a merge to take place. Below is a brief list of the options that you can supply to the `git merge` command along with a brief description of how each one affects the merge process. We will explain the details a little more further on.

- **Resolve** - A two headed merge strategy using a 3-way merge algorithm. This is used by default in Git.
- **Recursive** - A two headed merge strategy using a 3-way merge algorithm. This algorithm looks at situations where multiple common ancestors are eligible and creates a merged tree of the common ancestors to be used as a reference. Usually, this method has less conflicts and carries with it multiple sub-options.
- **Octopus** - This merge strategy can merge in multiple heads. When trying to pull multiple topic branches into a single merge, an **octopus** is the default method that Git will use.
- **Ours** - Another multiple head strategy, which simply ignores all changes from the other branches. At first it may sound like a useless idea, but it could be used to keep the history of a branch without actually keeping the branch itself.
- **Subtree** - A much more advanced merge strategy based on the **recursive** which trees are adjusted to result in a better merge.

Now that we are aware of the different options available to us, let us discuss what happens when we actually perform a merge. Let us consider the state of our repository at the end of Week 4, as shown in Figure 1.

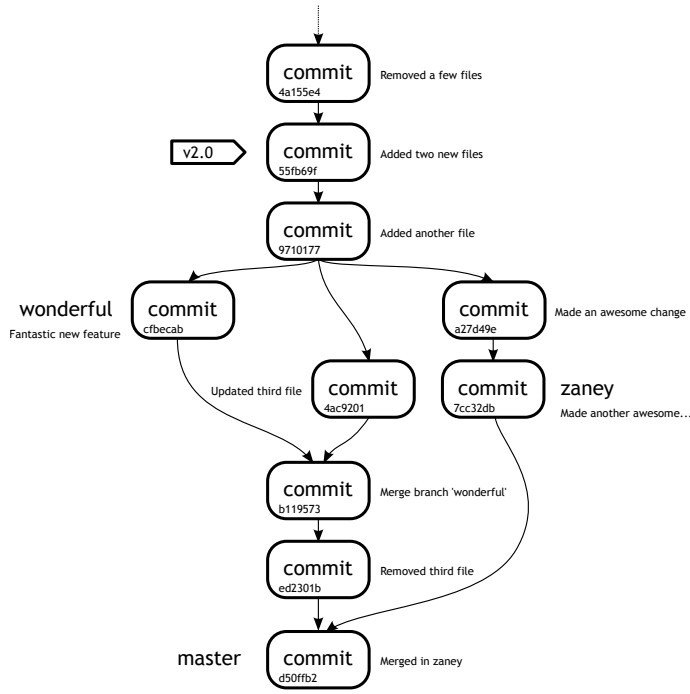


Figure 1: Repository at the end of *Week 4*

For arguments sake, let us say that we were on the **wonderful** branch, and we wanted to merge the **zaney** branch into it. Using a standard 3-way merge algorithm, this would mean taking our current branch, which we will call **A**, the **zaney** branch, which we will call **B** and a parent which we will **C**. Why do we need this parent? What is it used for?

In order to merge changes from a specific file together, we need to know exactly how that file has changed over time. In order to know that, we need to have a common starting point. By that we mean that we need to have a state in both of the branches history, where **A** and **B** were the same. In a version control system, this is most easily achieved by finding a commit that is common to the history of both branches. This commit is called the *Best Common Ancestor*.

Usually, when developing software, the code tree will be branched at various points along the way. Sometimes we may need to merge these branches together, or merge

these branches back into the master branch. Which ever way we do it, we will need to find a common ancestor. Figure 2, demonstrates a couple of possible merge proposals, and one of their common ancestors.

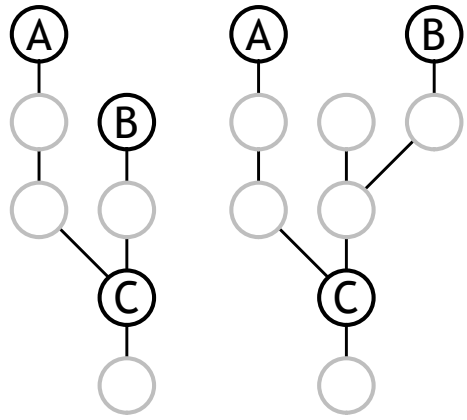


Figure 2: Finding the *Best Common Ancestor*

As you can see from the second example, branch **B** was created from a commit above the *Common Ancestor*, but the *Common Ancestor* remains the same.

Let us go back to our example above. We are attempting to merge in **zaney** to the **wonderful** branch. We need to find the **C** in our equation. To do this we could pour over Figure 1. This should yield the result to be commit **9710177**. Is there any way we can ask Git to do the same thing? The following output introduces a new command that is used to find the *Best Common Ancestor* for a given merge proposal.

```
john@satsuki:~/coderepo$ git merge-base wonderful zaney
9710177657ae00665ca8f8027b17314346a5b1c4
john@satsuki:~/coderepo$ git cat-file -p 9710177
tree 268f487e5c29a4b01c3a91637bac0024253fb77e
parent 55fb69f4ad26fdb6b90ac6f43431be40779962dd
author John Haskins <john.haskins@tamagoyakiinc.koala> 1301613377 +0100
committer John Haskins <john.haskins@tamagoyakiinc.koala> 1301613377
 +0100

Added another file
john@satsuki:~/coderepo$
```

In our example, `git merge-base` uses two parameters to speficy the two heads that we wish to merge. We can see that the commit that Git suggests as the *Best Common Ancestor* is what we suggested above, **9710177**. We have reused our `git cat-file` command to prove that the commit is the one we suggested, as you can see, the commit messages match.

This explains how a simple merge takes place, Git will find the common ancestor, and try to merge in the differences to produce a merged version. If the merge cannot take place, then as shown in *Week 4*, Git will put all the changes in the file, label them, and then raise a conflict for the user to resolve.

With the information presented here, you should now be able to reread the descriptions above and understand what is going on at a base level. If you require further information on merging, there is a wealth of documentation and papers about merge algorithms on the Internet.

"Grepping your life away"

A subtle twist on searching

As well as our `git log` tool, which we have used for searching, it is useful to know that there is actually another way to search for strings in your current directory. We can use the `git grep` tool to find all files in our working tree which have a certain pattern in them.

```
john@satsuki:~/coderepo$ git grep awesome
newfile1:and some more awesome changes
newfile2:and a new awesome feature
john@satsuki:~/coderepo$
```

This has returned two files, both containing the string `awesome`, as this is the parameter that we passed to the `git grep` command.

We can extend this a little further and ask Git to supply the context around the line that it finds. In our simple case, it results in the following.

```
john@satsuki:~/coderepo$ git grep -C3 awesome
newfile1-A new file
newfile1:and some more awesome changes
--
newfile2-Another new file
newfile2:and a new awesome feature
john@satsuki:~/coderepo$
```

This is just a short introduction to the `git grep` command. If you want to know more you should spend some time reading the manual.

Week 5

Day 1 - "This isn't working for me John"

Dealing with resistance

So, now that the team have discovered the basics of branching, they are conceptually ready to start using it in earnest. When implementing a version control system, or shifting from one to another, it is important to make sure that the users are happy with the system and know how to use it. Training is a big issue.

It would seem that the team have coped with the initial usage of Git and that they have utilised each others talents in specific areas to pull together a good learning environment. However, one thing to bear in mind is that some users may secretly be having a far worse experience than their colleagues. It is also common for these people to suffer in silence, or to wait until they are asked for their opinions on the system before they bring up any issues.

It is because of these very factors that you should probably consider employing a parallel implementation. This is exactly what John decided to do with Tamagoyaki Inc's implementation of Git. Whilst a parallel implementation does take duplicate effort in some areas, it also allows the team to return to their original system, should insurmountable obstacles present themselves. However, a parallel implementation should never be an excuse not to eventually shift over to the new system, unless serious issues are discovered.

> **In the trenches...** "I'm sorry John, but I just can't do this anymore!" Eugene was leaning over the partition wall, the keys that hung around his neck clattering loudly as he swayed. "You hard core devs may be happy with all that command line junk, but I'm a GUI kinda guy. It doesn't come as easily for me as if does for you."
>
> "You should learn how to use a computer properly then," shouted Klaus before laughing.

Terminology

Parallel Implementation

Parallel implementation means keeping your old system running, whilst bringing up a new one. It incurs extra effort in many areas, such as system administration, backups and system usage, but it allows people to evaluate a product in a real life situation. Usually people will have completed all of their preliminary testing before moving onto Parallel implementation. It would be assumed that you were fairly certain that your were going to move forward with a final implementation, before taking the time of setting up and training people on the new system.

It does however allow people to continue with the day jobs, without the risk of issues with the new system completely blocking them from working. This is often a critical factor when implementing a new system. If the system is successful, over time, the users will migrate away from the old system and start using the new system exclusively. Care should also be taken that whilst in the midst of parallel implementation, both systems remain up to date at all times, this is often the trickiest part. In Tamagoyaki Inc's case, because they were just using tarballs of their code base, it is easy for them to tar up a folder, as well as commit it to the repository.

Eugene was livid, "You're such a damn elitist Klaus. I'm so glad I don't have to share a pod with you anymore, you zealot." With that, Eugene was off, flinging open the door to the office area and stomping off to his desk.

"Nice one Klaus," said John, "You know, you could be a little more tactful. We do still need sign off from him to complete this project."

Klaus shrugged.

* * *

"Listen Eugene, I think I have a way to help you out. There is a GUI

> component to Git that you can use." John was trying his best, but five minutes of grovelling to Eugene, hadn't exactly paid off.
>
> "I'll try your GUI, but if I don't like it, I'm not signing off." He was serious too. "I don't have time to waste learning this system, I never needed versioning before, why should I need it now."
>
> "We have to work together on things now Eugene," began John, "You know there is a merger looming, right?"
>
> Eugene looked up, a little stunned.
>
> "We have to show we can function well as a team, that we have everything in hand. Let's leave the integration till the end of the week, to give you a little more time to get used to things."
>
> "OK" said Eugene, "I'll give it my best shot"

Sometimes, dealing with resistance to new systems is hard. In Tamagoyaki Inc, John was blessed with the fact that only one developer didn't like the system he had picked. Fortunately, the developer in question was only really concerned with the lack of a GUI, something that Git actually provides anyway.

It is very important to listen to users issues and questions. Often they may discover a big hole in your initial planning which you would never have seen. Never dismiss a concern before looking into it as it can be difficult for one person to understand the entire process in place during development, no matter how well documented it is. Going through a period of User Acceptance Testing is crucial before complete adoption is even considered.

Let us take a while to explore the built in GUI that Git comes bundled with.

A little bit of graphics

Whilst using a GUI can be faster for some operations, it is also worth noting that with very few exceptions, GUIs are often less feature rich than their CLI counterparts. It is very time consuming to write a GUI that can deal with every command option a user desires, so often the GUI will handle the most common use cases, leaving the CLI to handle special cases.

Git is no exception to this rule. Whilst the GUI component is a very capable tool indeed, it does lack most of the advanced functionality that can be found on the

command line. In fact, there are even some of the basic parameters to some of the commands that we have used earlier, that are not available in the GUI counterpart.

We can invoke the GUI by running the `git gui` command. We will be presented with a window similar to that in Figure 1. Note that in this case, we are running it against our test repository that we have been working on through previous chapters.

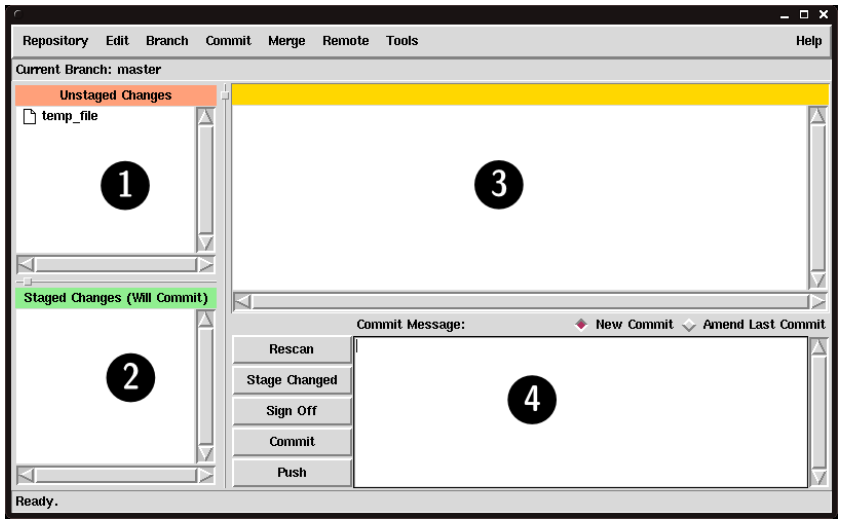


Figure 1: Initial Git GUI view

This initial window is composed of four key areas

1. **Unstaged** - This area on the screen shows all of the items present in the working copy which are unstaged, that is to say they have not yet been added to the index and so will not be included if a commit were to take place.

2. **Staged** - This area on the screen shows all of the items present in the index or staging area. Everything listed here will be included if a commit takes place.

3. **Content View** - This region of the window will show the contents of an item if it is selected in either the Unstaged or Staged areas. If the file is already tracked, then the window will show a diff between the last committed version and the chosen version.

4. **Commit** - From the commit section of the screen, the commit message can be written, the commit performed, the directory rescanned for changes, as well as other operations.

If we take a closer look at the **Unstaged** area of the screen, we see something interesting. This is shown in Figure 2.

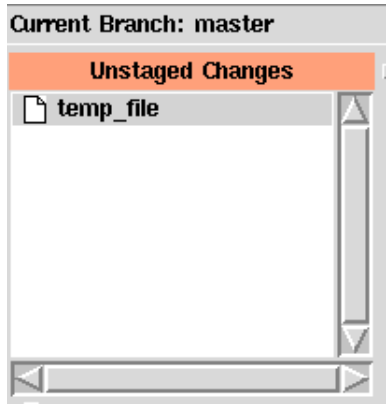


Figure 2: Unstaged section showing our `temp_file`

This is the only file in our working copy which contains unstaged changes. That should not be surprising as this file has never been added to the repository so it is not considered **tracked** by Git. If you remember, it was actually a temporary file that we piped some output to in an earlier chapter. If we click on the filename of this file, the **Content View** area changes to show Figure 3.

In the top banner section of the **Content View** we see an indication of the file's status. In this case it is **Untracked** and **not staged**. We can change this by clicking on the small blank page icon to the left of the file name in the **Unstaged** area of the screen, see Figure 4.

Now the file has moved into the **Staged** area of the screen as would be equivalent to us doing a `git add temp_file`. The file has been added to the index and is now ready for committing. We also notice a difference in the **Content View** of `temp_file`. This can be seen in Figure 5 and now shows a patch view of addition of the file, as opposed to just the contents of the file.

Now the bar reads that the file is **Staged for commit**, which is exactly what we

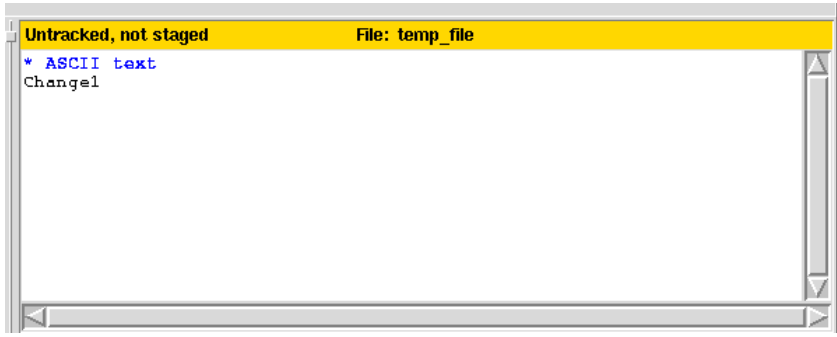


Figure 3: Content view of `temp_file`

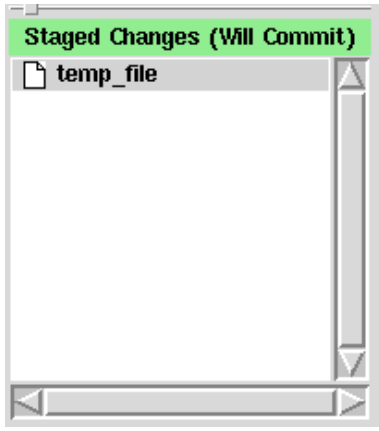


Figure 4: Staged section

expect. We are now going to fill out a commit message, which can be seen in Figure 6, and press the **Commit** button, to initialise a commit into the repository.

It should be noted here that we could have performed multiple operations here, adding several files to the staging area before pressing that all important **Commit** button. We will finish this section off by checking the status area at the very bottom of the screen. In Figure 7, you should see that our latest operation has been summarised by the string `Created commit 35243bf8: Added temp_file`

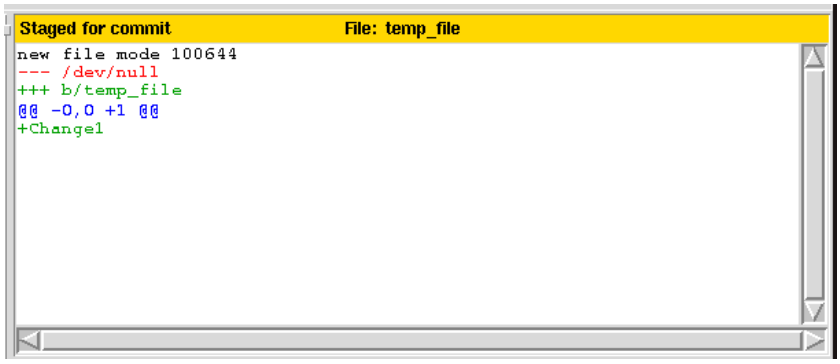


Figure 5: Content view of `temp_file`

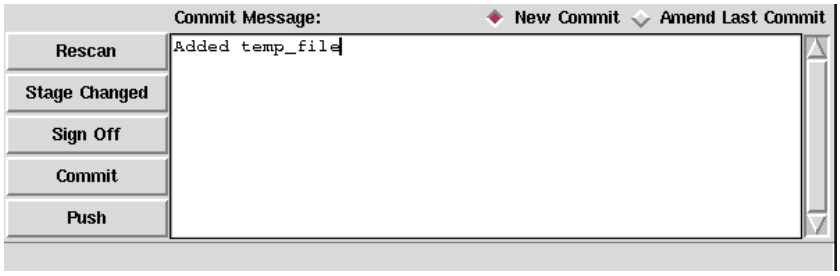


Figure 6: Preparing for commit

Figure 7: Status message showing a new commit ID

Day 2 - "Back to logging"

Visualisation to the max

With the basic operations down, as we discovered in Week 1, let us now move on to using the GUI to view the history of our database. The visualiser that is bundled with Git is

packed with features and can be invoked in one of two ways, either by running `gitk` from the command line, or by choosing ***Repository - Visualize All Branch History*** or ***Repository - Visualize master's Branch History***, the latter menu item is worded with the assumption that you are on the **master** branch of course.

Whichever way we begin an instance of `gitk`, we are likely to end up with a screen like the one in Figure 8.

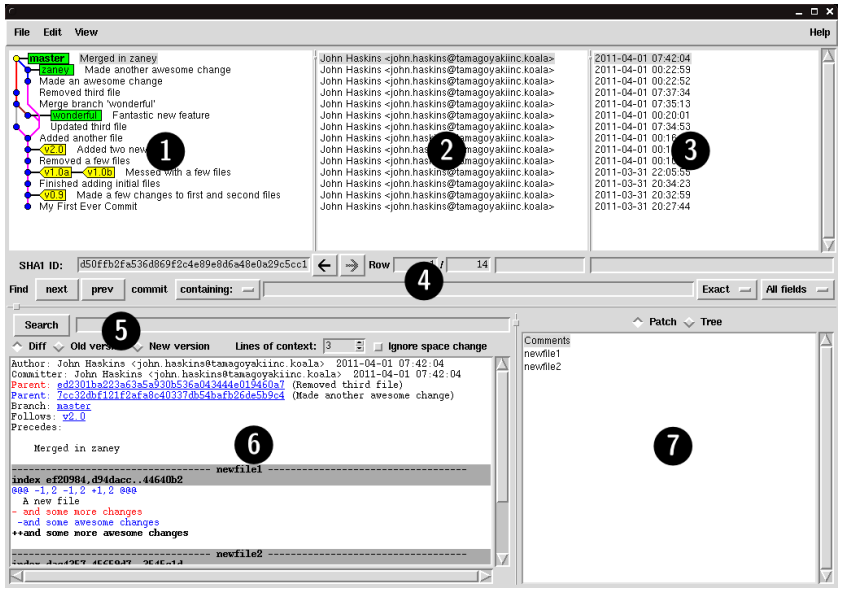


Figure 8: Initial `gitk` view of our repository

Let us spend a few minutes familiarising ourselves with the layout of the `gitk` tool. The window is split up into roughly seven different areas. We will take a brief look at each of these below.

1. **History Graph** - This area of the screen gives a graphical representation of the history of our repository. Similar to the `git log –graph` option that we used previously, the graph here is much more readable.

2. **Committer History** - The committer history tells us the person who added the commit which is horizontally adjacent in the commit graph.

3. **Date History** - The date history is very similar to the committer history section of the screen and shows us a simple view of the date of the commit which is horizontally adjacent.

4. **History Search** - This section of the screen allows you to narrow down and highlight a subset of commits which meet a certain criteria. It also allows you to navigate through these results.

5. **Commit Search** - Once you have highlighted a commit, the commit search section allows you to find specific strings within that commit, including looking at just old or new lines to the repository.

6. **Content View** - This window actually shows the data that was stored in the commit and allows you scroll through either changes or complete files.

7. **File System View** - The file system view shows you a list of files that were either modified in a specified commit, or that were present in the commit.

To start with, let us take a look at the graph that is present in the **History Graph** section of the screen in Figure 9. Essentially this is just a very simplified version of the graphs we were drawing in the previous chapter. As you can see, the only information that appears to be missing from the graph is the commit ID, which can be obtained quite easily by clicking on the relevant commit and viewing the string presented just below the graph pane.

The historical graph shows us all active branches, tags and commits. From looking at this it is easy to see where our merges occurred and where the branch HEADs point to. The branches are identified by green rectangles, and the tags as yellow labels. Each circle on the graph is a commit and is linked to the panes on the right, where you can see the committer and date information.

If we select the **master** branch HEAD, which should be the top commit, (in fact on opening `gitk` this should already be selected), we should see a pane similar to Figure 10 in the **Content View** section of the screen.

Notice in this that we have two parents listed, which is what we would expect, as the last commit we did was that of a merge from **zaney** into **master**. Notice also that below this, we get to see a diff output of exactly what changed during this merge. Interestingly, it shows that `newfile1` loses one line from parent **ed2301b** and another

Figure 9: Graphical history of our repository

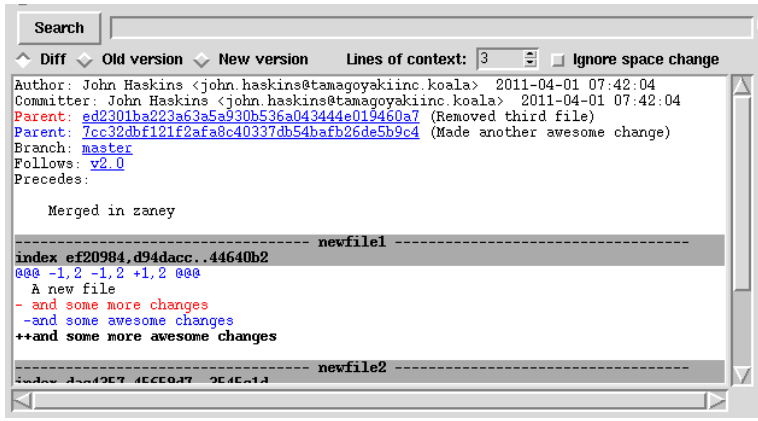


Figure 10: Content view of **master** HEAD

from parent **7cc32db**. These are both replaced by the next line, which reads `and some more awesome changes`.

The current commit is a little special as it is a merge. If you choose any normal commit, you can use the `Diff`, `Old version` and `New version` buttons to either show what the file used to look like (Old), what it looks like now (New), or the default view

which is the Diff and shows the combination of the changes.

Above the **File System View** pan is a button to switch between `Patch` and `Tree` views. By default, this is set to `Patch` and changes what the **Content View** pane displays. When set to `Patch`, this shows us the changes between the old commit and the new one. However, when we switch this to `Tree` view the **Content View** pane changes to show the contents of a file selected from its pane, at the current commit. In this way, it allows you to browse and display files from previous commits graphically and effortlessly. Simply select the commit, selected `Tree` mode, choose the file, et voila, it is presented in the **Content View** pane.

So as you can see, the `gitk` tool is already quite powerful. We are now going to take things a step further. Remember in **Week 3** we had a way of searching the repository for the introduction of a string. We can do the exact same thing with `gitk`.

We are going to drop down the box which currently reads `contains:` in the **History Search** pane and change it to `add/removing string:`. Then we are going to enter `Change1` into the field on the right. Your changes should look similar to Figure 11.

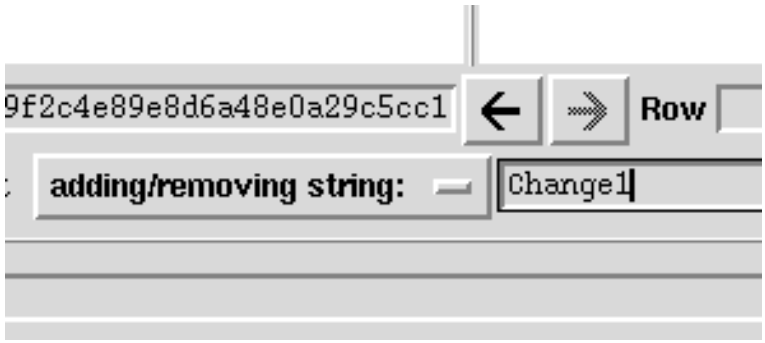


Figure 11: Searching for a string

After you have finished typing you should already have noticed a difference in the **History Graph** pane. Notice how some commits are now highlighted in bold, as demonstrated in Figure 12?

If you remember from Week 3, when we first ran the `git log -S "Change1"` command, we were presented with only one commit. That was titled, `Made a few changes to first and second files`. Our repository has moved on since then and we can see that that particular string, `Change1` was added or removed in two other commits

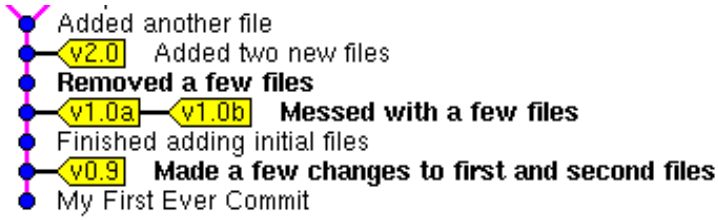


Figure 12: Search results are highlighted

as well. This was when we totally changed `my_second_committed_file` and when we removed `my_first_committed_file` a little later. The **Next** and **Previous** buttons can be used to navigate through the search results.

Now that we have found the commits that contain the change to the string we are looking for. The question is, where in the file does this change occur? We can now use the **Commit Search** pane to see this. Typing `Change1` into this search box will highlight the relevant text in the `Content View` pane below. It really is as easy as that.

Whilst it is a great idea to remember and use the command line arguments and parameters, it is also useful to know that these other tools are available. GUI tools should not be frowned upon, as some command line purists do. GUI tools are as much a part of the development process as their command line counterparts. Both have their uses and the most important lesson of all is to know when to use which. This lesson will most likely come with experience and time, or as a friend of mine used to put it, old age.

Customising the visualisation

> **In the trenches...** "Eugene," called Klaus as the tools developer walked past him. "Could I borrow you for a second?"
>
> "I guess," said Eugene coldly. "What is it that you want exactly?"
>
> "Well," Klaus began, "I started using the GUI tools a little and found something you might consider interesting. I know you spend a lot of time switching between different versions of our code looking for various functions and things and I found this cool tool in the GUI."
>
> Eugene breathed in deeply. Knowing Eugene, it wouldn't be anything amazing, but nevertheless, his interest had been piqued.

The `gitk` tool has a pretty awesome feature called *views*. Sometimes it may be necessary to keep track of everything a certain person has done. Or maybe see anything that has happened in the last week. You may even be interested in people adding certain strings to the repository. The *views* feature allows you to do just that. By setting up a view, you can filter the results that are displayed in `gitk`, by simply switching to it.

In the example we are going to go back to the search we made previously and filter the history for any changes to `my_first_committed_file` that add or remove the string `Change1`. First we are going to load the dialog box, by using the menu ***View - New View...***. The resulting dialog is displayed below and we have already filled in the required information.

We changed the **View Name** to be `Change1`. This is simply an identifier that will allow us to choose our view in the future. We have also ticked the **Remember this view**. Ticking this box ensures that the view is remembered once we close `gitk`. Sometimes, this behaviour is not desired, but in our case we want this view to be available every time we start `gitk`.

There are many options which we can set, but in our case we have chosen to simply add `Change1` to the **Changes to files:** section. This is equivalent to the search that we performed earlier. We have also filtered the files which are included in the results, by putting, `my_first_committed_file` in the **Enter files and directories to be included** box.

The result of this view is shown in Figure 14. Notice that instead of all the commits being shown, we are just shown the two that we are interested in. Both of these commits are referred to earlier in the chapter, when we ran the search manually in `gitk`

This ends our tour of the `gitk` utility. Hopefully you have seen that it can actually present a large amount of information in a very compact and usable way. As such it should not be forgotten about and should remain part of your arsenal of Git tools.

Day 4 - "Advanced Techniques"

Getting more done graphically

Whilst `git gui` can perform commits. It is interesting to note that it can do a whole lot more. One feature, which can be exceedingly useful is the **File Browser**. This is invoked by using the ***Repository - Browse Branch's Files...*** menu item.

We can now select the point at which we want to browse the repository. By default we are presented with a list of the branches present in the repository. Note we can also

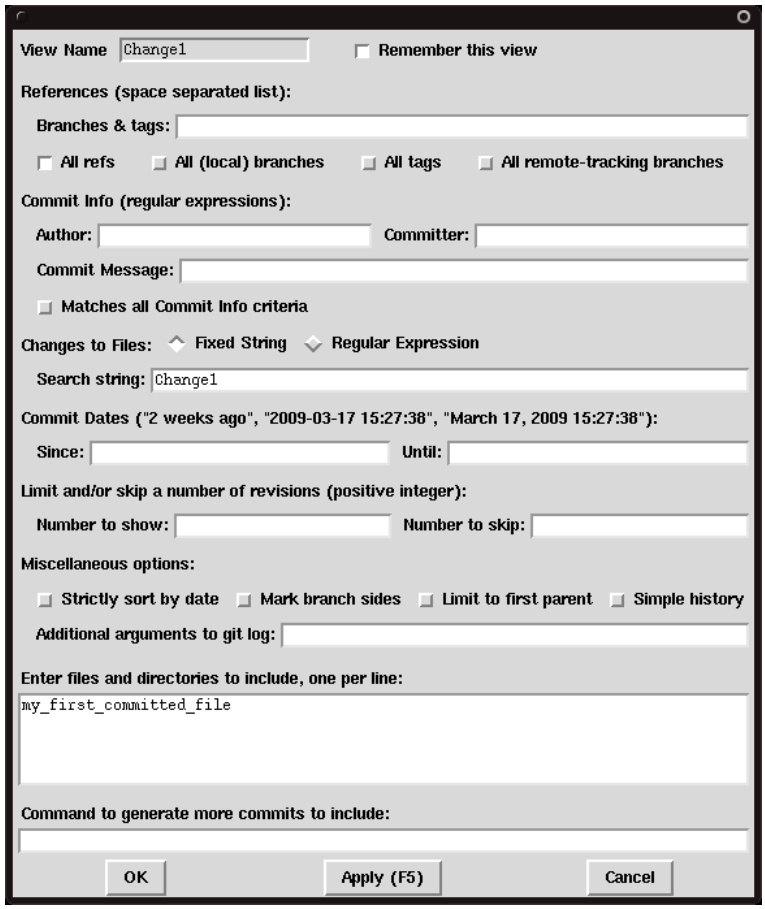


Figure 13: Setting up a *view*

choose tags, by selecting the appropriate option in the radio box. We can also put things like `HEAD` and `HEAD 1` into the **Revision Expression**: text field.

The dialog box is shown in Figure 15. Notice also that when we hover over the branch names, we are presented with a small pop up, telling us who made the last commit to the branch and what the commit message was. Clicking the **Browse** button will take us to the next window, where we can choose the file we wish to view.

The file choosing dialog is rather plain. Just double clicking on a file will start the `git`

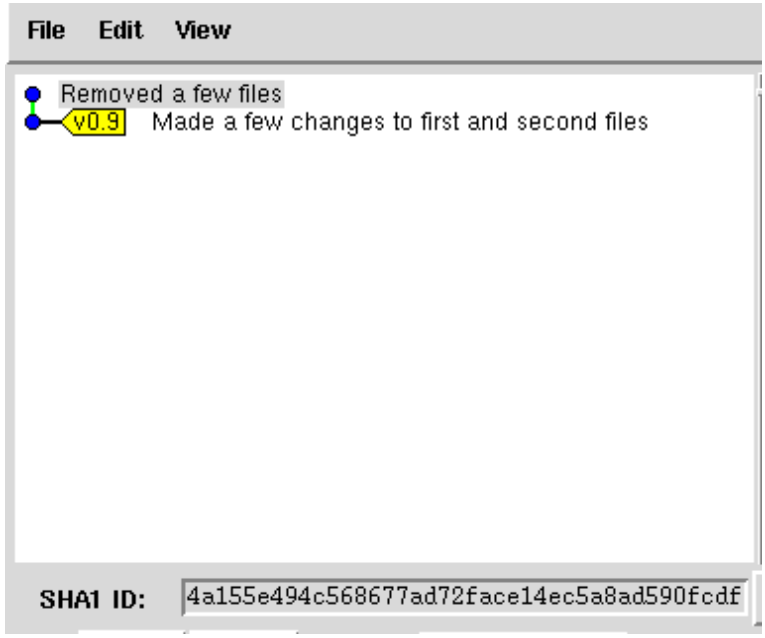


Figure 14: View results

`gui blame` tool. Though primarily used as a way to see who, or what was responsible for a specific change to a line, this tool is also useful for seeing how a file has changed over time. The file chooser dialog is shown in Figure 16.

Once selected, the file is displayed in its own window. The window is split into two sections. The top part of the window displays the actual content of the file, along with some commit IDs on the left. In the lower part of the window, more information on a specific commit is shown. This window is shown in Figure 17.

Clicking on a line in the view above, will turn that section green and will display the information related to that commit in the lower pane, along with the complete commit message. It will also highlight all other lines in the current file that were also modified in that commit. As shown in Figure 18.

Already this is a very powerful tool, much like the `gitk` too for visualising the history of the repository, `git gui blame` is also very useful for working through the events that led to the current version of a particular file. In larger files, being able to

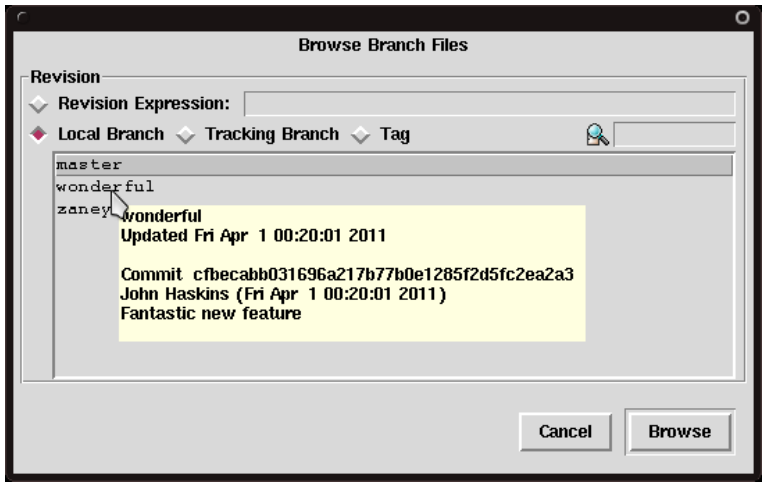


Figure 15: Point in history dialog

click on a portion and have it highlight every other line in that file that was changed in that commit can be extremely useful. By right clicking on the file, we can also choose the **Show History Context**, which will load `gitk` and move the commit history pane to show the commit we are currently interested in.

You should have noticed by now that each line has a commit ID associated with it, two in fact. These are in fact links, and by clicking on these, we can wind the history of the file, back to that point, so that it shows not the state at which it originally did, but now the state as it was in the selected commit. This is shown in Figure 19.

Again this is a hugely powerful tool. You can navigate back to the current version of the file, or if you have clicked on several to get to your current point, review a history of your path, by clicking on the green arrow at the top of the screen. You will be presented with a drop down menu, and from here you can choose which point you wish to return to.

Our last stop

We have shown a number of things that Git can do in its GUI form. We are not going to dwell on the graphical interface any further as you should already be familiar with the **Merge** and **Branch** menu items, which are present. They have a number of options which are also available from the command line.

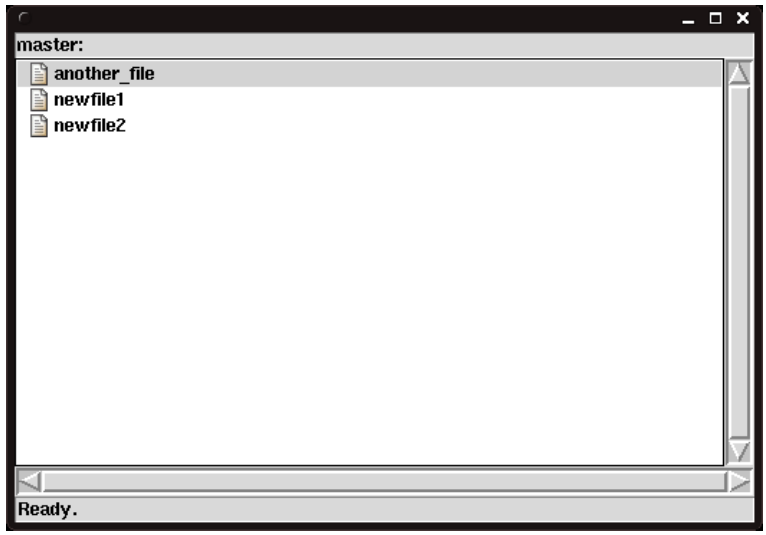


Figure 16: File chooser

The **Branch** menu allows you to do the familiar procedures, such as checkout, create, rename and delete. It also provides a limited reset feature. The **Merge** menu provides shortcuts for running a merge, as well as aborting it.

It is here that we are going part with our newest tool in order to return to the command line interface once more. We have one more trick up our sleeve and this will be presented in the *After Hours* section for the week. As previously stated, the GUI tools form an important part of your version control arsenal. Never be afraid to use them. Remember, at the end of the day, you should be focussed on getting the job done and using whichever tools you require to do just that.

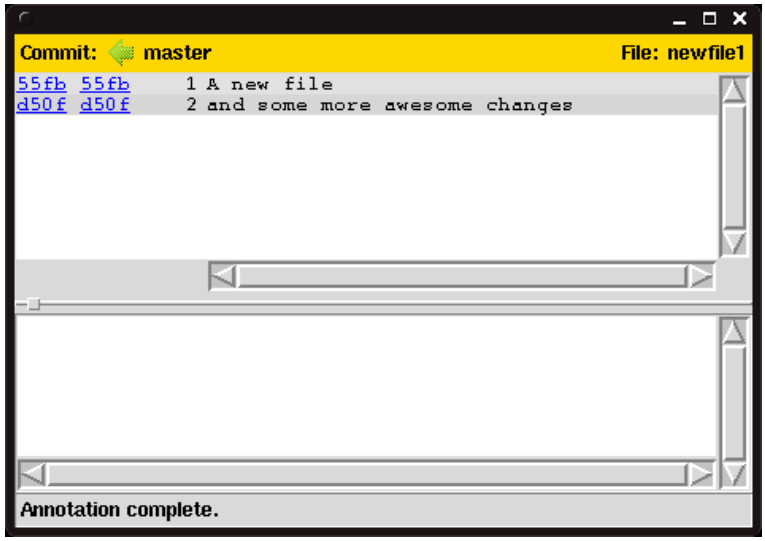


Figure 17: File viewer

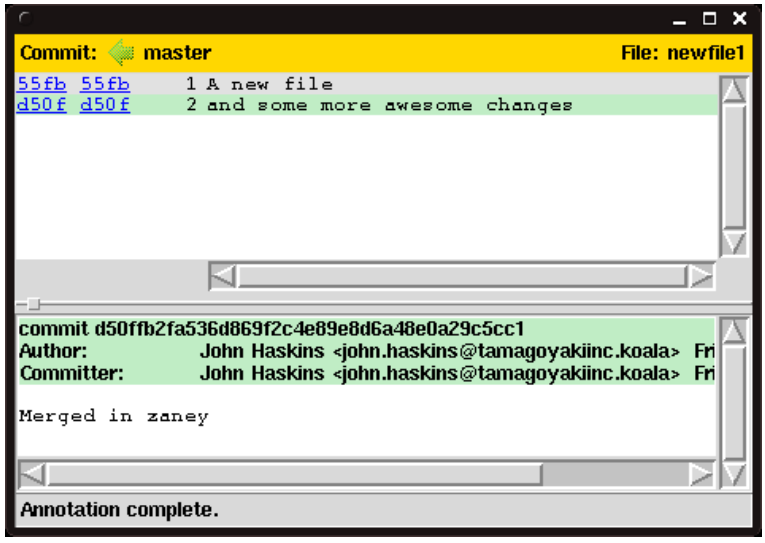


Figure 18: File viewer, highlighting a commit

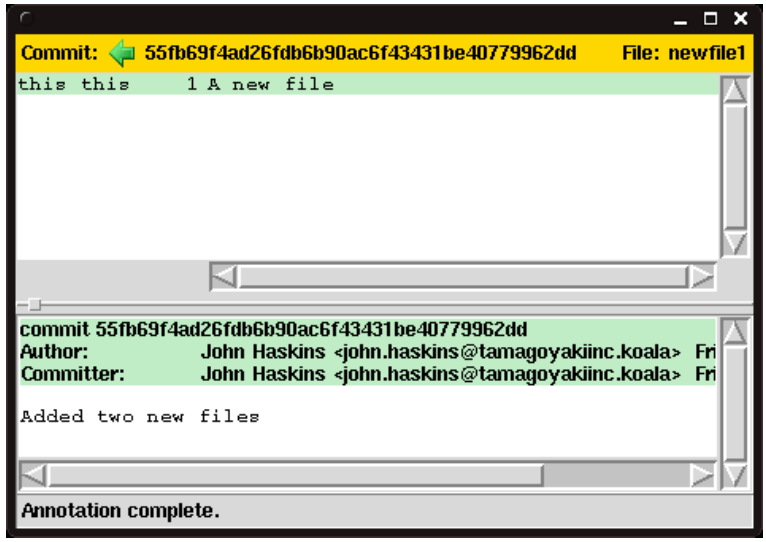


Figure 19: Going back to a previous commit

Summary - John's Notes

Commands

- `git gui` - Invokes the Git GUI tool
- `gitk` - Starts an instance of the graphical history tool for Git
- `git gui blame <path>` - Opens the blame window for a specific file

Terminology

- **Blame** - A way of finding the commit that caused an issue in the repository.

After Hours Week 5

"Splitting up commits the easy way"

Taking commits that little bit further

Sometimes, putting everything in a single commit just is not a good idea. Imagine you have pulled in number of updates to your working directory. You may want to split these up. It is true that you could simply `git add` only the files you want to include in the commit. However, what happens when you have change four or five different things in the same file, and you want to split that commit up into five different commits.

There are two ways you can approach this. The first is to copy the file in question out of the working directory, reset the working copy back to the last committed and copy your changes in line by line. This can be time consuming and frustrating and when you are working on many files, it can be totally impractical. What we need is a way to include or exclude certain lines of a file.

To demonstrate this we are going to create a new branch called **fantasy** and we are going to make several changes to a few files. We are then going to show how the same process can be achieved by using both the GUI and the command line.

So let us start by creating our branch and making some changes as shown below.

```
john@satsuki:~/coderepo$ git checkout -b fantasy
Switched to a new branch 'fantasy'
john@satsuki:~/coderepo$ echo "This is line 1" > newfile1
john@satsuki:~/coderepo$ echo "This is line 2" >> newfile1
john@satsuki:~/coderepo$ echo "This is line 3" >> newfile1
john@satsuki:~/coderepo$ echo "This is line 4" >> newfile1
john@satsuki:~/coderepo$ echo "This is a new line" >> newfile2
john@satsuki:~/coderepo$ echo "This is another new line" >> newfile2
```

Let us now just run a `git diff` to see exactly what the changes are.

```
john@satsuki:~/coderepo$ git diff
diff --git a/newfile1 b/newfile1
index 44640b2..0eccf1a 100644
```

```
--- a/newfile1
+++ b/newfile1
@@ -1,2 +1,4 @@
-A new file
-and some more awesome changes
+This is line 1
+This is line 2
+This is line 3
+This is line 4
diff --git a/newfile2 b/newfile2
index 3545c1d..40efcce 100644
--- a/newfile2
+++ b/newfile2
@@ -1,2 +1,4 @@
 Another new file
 and a new awesome feature
+This is a new line
+This is another new line
john@satsuki:~/coderepo$
```

Now, we could just do `git commit -a` and be done with it, but what if we really wanted to split this information up into four commits? We will introduce a new parameter to our `git add` tool from before. We are going to use the `git add -p` or `git add –patch`. This will allow us to interactively edit the hunks before they are committed. To begin with, let us run `git add -p` and see what it is we need to do.

```
john@satsuki:~/coderepo$ git add -p
diff --git a/newfile1 b/newfile1
index 44640b2..0eccf1a 100644
--- a/newfile1
+++ b/newfile1
@@ -1,2 +1,4 @@
-A new file
-and some more awesome changes
+This is line 1
+This is line 2
+This is line 3
+This is line 4
Stage this hunk [y,n,q,a,d,/,e,?]?
```

We are given the options of `y,n,q,a,d,/,e,?`. At first glance, this may seem rather daunting. Let us choose the `?` and see what help is presented to us.

```
Stage this hunk [y,n,q,a,d,/,e,?]? ?
y - stage this hunk
n - do not stage this hunk
q - quit, do not stage this hunk nor any of the remaining ones
a - stage this and all the remaining hunks in the file
d - do not stage this hunk nor any of the remaining hunks in the file
g - select a hunk to go to
/ - search for a hunk matching the given regex
j - leave this hunk undecided, see next undecided hunk
J - leave this hunk undecided, see next hunk
k - leave this hunk undecided, see previous undecided hunk
K - leave this hunk undecided, see previous hunk
s - split the current hunk into smaller hunks
e - manually edit the current hunk
? - print help
@@ -1,2 +1,4 @@
-A new file
-and some more awesome changes
+This is line 1
+This is line 2
+This is line 3
+This is line 4
Stage this hunk [y,n,q,a,d,/,e,?]?
```

So it appears that Git is offering us the opportunity to either

- Stage it
- Do not stage it
- Quit,
- Stage it and all remaining hunks
- Do not stage it or any of the remaining ones
- Search for a regex
- Edit the hunk

In fact, though the help mentioned a **split** command, we do not have this option available to us, due to the nature of our hunk. Instead, if we wish to split this hunk, we are going to have to edit it manually. To do this we will choose the `e` option.

Here we are left in our chosen text editor, to either add, modify or remove lines from the hunk. In our case, we are going to delete a few lines. We only want to leave the first line of additions. Just because we delete the others does not mean they are deleted from the working copy. Remember, we are not editing the actual files here. Just the hunks that are going to be staged.

```
# Manual hunk edit mode -- see bottom for a quick guide
@@ -1,2 +1,4 @@
-A new file
-and some more awesome changes
+This is line 1
+This is line 2
+This is line 3
+This is line 4
# ---
# To remove '-' lines, make them ' ' lines (context).
# To remove '+' lines, delete them.
# Lines starting with # will be removed.
#
# If the patch applies cleanly, the edited hunk will immediately be
# marked for staging. If it does not apply cleanly, you will be given
# an opportunity to edit again. If all lines of the hunk are removed,
# then the edit is aborted and the hunk is left unchanged.
```

After the deletes, the editors file should look like this.

```
# Manual hunk edit mode -- see bottom for a quick guide
@@ -1,2 +1,4 @@
-A new file
-and some more awesome changes
+This is line 1
# ---
# To remove '-' lines, make them ' ' lines (context).
# To remove '+' lines, delete them.
# Lines starting with # will be removed.
#
# If the patch applies cleanly, the edited hunk will immediately be
# marked for staging. If it does not apply cleanly, you will be given
# an opportunity to edit again. If all lines of the hunk are removed,
# then the edit is aborted and the hunk is left unchanged.
```

Once we quit our editor, we are then asked about the next hunk. In our case, we are

going to apply all of the changes to our second file during this commit. To do this we are going to use the `a` option.

```
diff --git a/newfile2 b/newfile2
index 3545c1d..40efcce 100644
--- a/newfile2
+++ b/newfile2
@@ -1,2 +1,4 @@
 Another new file
 and a new awesome feature
+This is a new line
+This is another new line
Stage this hunk [y,n,q,a,d,/,e,?]? a

john@satsuki:~/coderepo$
```

At first glance, we do not appear to have been left with any indication that anything has taken place. In order to perform a check we shall run our obligatory `git diff`, both between the working copy and the index, and between the index and the last commit.

```
john@satsuki:~/coderepo$ git diff
diff --git a/newfile1 b/newfile1
index f702b65..0eccf1a 100644
--- a/newfile1
+++ b/newfile1
@@ -1 +1,4 @@
 This is line 1
+This is line 2
+This is line 3
+This is line 4
john@satsuki:~/coderepo$
```

So above, we can see that the difference between the working copy and the index, or staging area, is the last three lines that we are not ready to commit yet. Below we can see the difference between the staging area and the last commit to the repository. This includes the three lines that we included during our interactive commit preparation.

```
john@satsuki:~/coderepo$ git diff --cached
diff --git a/newfile1 b/newfile1
index 44640b2..f702b65 100644
--- a/newfile1
+++ b/newfile1
```

```
@@ -1,2 +1 @@
-A new file
-and some more awesome changes
+This is line 1
diff --git a/newfile2 b/newfile2
index 3545c1d..40efcce 100644
--- a/newfile2
+++ b/newfile2
@@ -1,2 +1,4 @@
 Another new file
 and a new awesome feature
+This is a new line
+This is another new line
john@satsuki:~/coderepo$
```

We can now commit in the normal way, and continue to edit the working copy to stage the sections we require. The following output is shortened for brevity.

```
john@satsuki:~/coderepo$ git commit -m 'Added first line'
[fantasy 03bd20c] Added first line
 2 files changed, 3 insertions(+), 2 deletions(-)
john@satsuki:~/coderepo$ git add -p
diff --git a/newfile1 b/newfile1
index f702b65..0eccf1a 100644
--- a/newfile1
+++ b/newfile1
@@ -1 +1,4 @@
 This is line 1
+This is line 2
+This is line 3
+This is line 4
Stage this hunk [y,n,q,a,d,/,e,?]? e

john@satsuki:~/coderepo$ git commit -m 'Added second line'
[fantasy 302e3fa] Added second line
 1 files changed, 1 insertions(+), 0 deletions(-)
john@satsuki:~/coderepo$ git add -p
...
...
```

We have gone through the process of editing each hunk for each commit and have performed the commits.

```
john@satsuki:~/coderepo$ git log
```

```
commit a59e73b1dc571318a1154aa4c2fc591ab6f1f395
Author: John Haskins <john.haskins@tamagoyakiinc.koala>
Date:   Wed Apr 13 23:56:39 2011 +0100

    Added fourth line

commit 3ca3d627a54418be4c2e9d9196db6ce62e2b93ff
Author: John Haskins <john.haskins@tamagoyakiinc.koala>
Date:   Wed Apr 13 23:56:13 2011 +0100

    Added third line

commit 302e3fa5f880a2a503235667b4c96d4dcdaa11be
Author: John Haskins <john.haskins@tamagoyakiinc.koala>
Date:   Wed Apr 13 23:55:57 2011 +0100

    Added second line

commit 03bd20cb8a78a28f003ab402492cf7055f21bb2e
Author: John Haskins <john.haskins@tamagoyakiinc.koala>
Date:   Wed Apr 13 23:55:32 2011 +0100

    Added first line
...
...
```

Next we are going to see how to perform exactly the same procedure using `git gui`. For a start, we are going to reset our branch HEAD and our index back to their state before we made the last four commits, however we are going to leave the working copy files in the state they were after these last four commits. This is called a **mixed** reset, as it modifies the staging area and the HEAD, but does not touch the working files.

```
john@satsuki:~/coderepo$ git log --oneline
a59e73b Added fourth line
3ca3d62 Added third line
302e3fa Added second line
03bd20c Added first line
d50ffb2 Merged in zaney
ed2301b Removed third file
...
...
john@satsuki:~/coderepo$ git reset --mixed d50ffb2
```

```
Unstaged changes after reset:
M newfile1
M newfile2
john@satsuki:~/coderepo$
```

Now we will run a diff, just to be sure.

```
john@satsuki:~/coderepo$ git diff
diff --git a/newfile1 b/newfile1
index 44640b2..0eccf1a 100644
--- a/newfile1
+++ b/newfile1
@@ -1,2 +1,4 @@
-A new file
-and some more awesome changes
+This is line 1
+This is line 2
+This is line 3
+This is line 4
diff --git a/newfile2 b/newfile2
index 3545c1d..40efcce 100644
--- a/newfile2
+++ b/newfile2
@@ -1,2 +1,4 @@
 Another new file
 and a new awesome feature
+This is a new line
+This is another new line
john@satsuki:~/coderepo$
```

If we now run our `git gui` command, we will see the changes that are present, once we click on one of the files in the left hand portion of the screen. Let us start with `newfile2` as we want every change from that file present in this commit. Figure 1 shows what `git gui` looks like at this stage.

If we right click on one of the green lines, remembering that green is short for an **addition**, we get a menu which we have not seen before. Among other things, this menu has the ability to stage a specific line or hunk for commit. In our case we are going to hit the **Stage Hunk For Commit** and then move on to the next file, `newfile1`.

Now we are looking at `newfile1`, we can use the **Stage Line For Commit** to add specific lines into the staging area. The file is shown in Figure 2. As our file is so small, we can run into problems if we just choose lines at random and stage them. This is

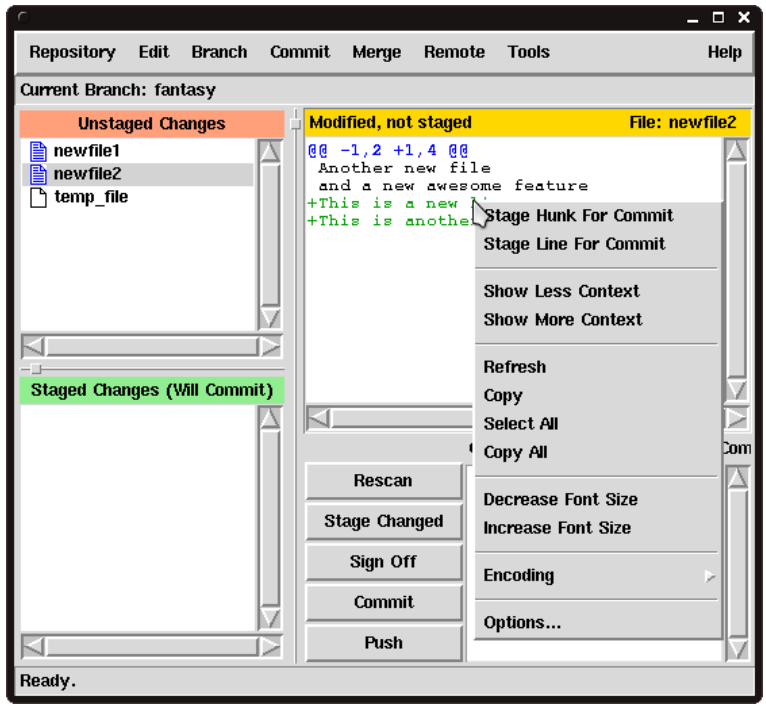


Figure 1: Changes to `newfile2`

because it is hard for Git to find context around which to associate the change. The context is the area immediately surrounding the change we are making. In order to reduce the risk of the error occurring, we are simply going to start at the top, and select the first three lines for committing, by right clicking on each on in turn and choosing the **Stage Line For Commit** option from the menu.

We have the changes that we expect ready to be committed. If we want to check one last time that they are right, we can use the **Staged Changes** pane on the left to choose the file and inspect the diff. Once we are happy we can use the **Commit** area of the window to type our commit message and emblazon our changes forever.

Figure 3, shows what our file looks like after the first commit. Interestingly, as our file is so small, you will probably find that `git gui` throws a corrupt patch error if you try to just commit the next line. As we mentioned earlier, it is always a good idea to

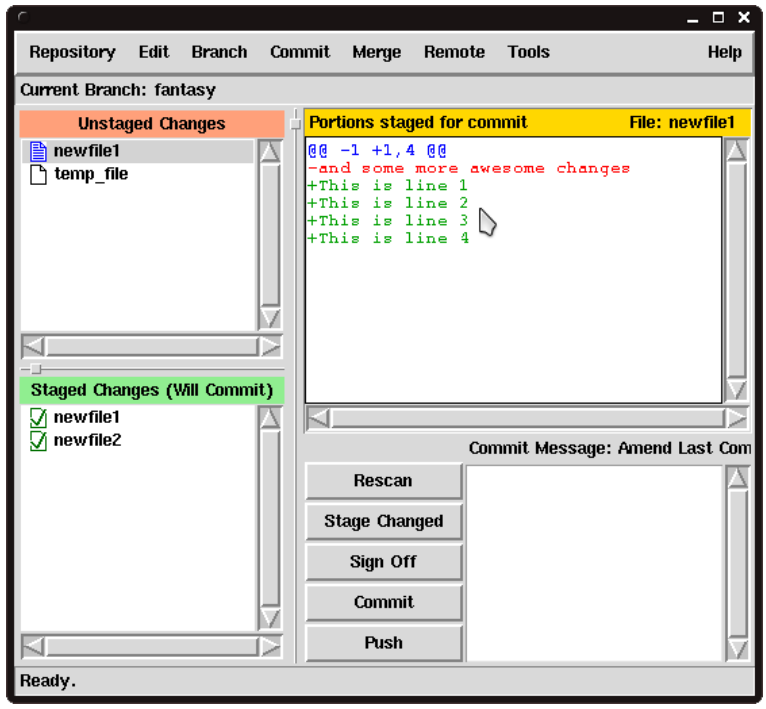


Figure 2: Changes to `newfile1`

know how to use the command line tools for precisely this reason. Often you are dealing with special cases, that the GUI just can not handle. In these cases, you may find you need to switch to the command line interface, to get the job done.

Let us now move back to our **master** branch and remove the **fantasy** branch.

```
john@satsuki:~/coderepo$ git checkout master
Switched to branch 'master'
john@satsuki:~/coderepo$ git branch -D fantasy
Deleted branch fantasy (was 29ceede).
john@satsuki:~/coderepo$
```

Notice that because we used the capital `-D` parameter, we were not asked if we were sure we wanted to delete the **fantasy** branch.

So in the *After Hours* section this week, we have found two ways to do the same complex task, one graphical and one command based. Staging commits in this way may

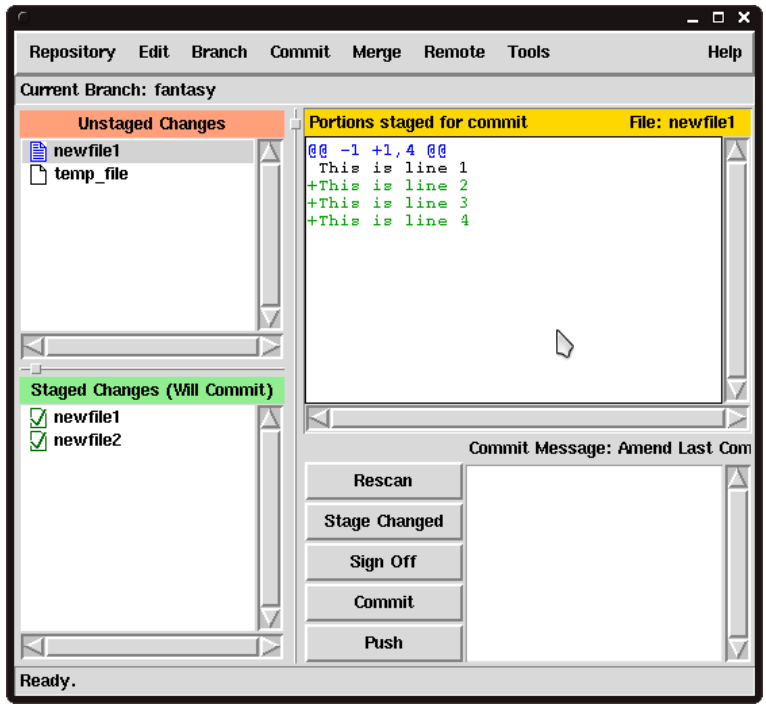


Figure 3: Changes to `newfile1` after staging

seem rather odd, but it will help you to keep your commits exceedingly exact. Whilst this may not matter on a personal project, for Tamagoyaki Inc, grouping the right lines together in a commit will be an extremely useful process to have available.

Week 6

Day 1 - "My private little stash"

Getting interrupted

We're getting close to the point where we can really start using Git as we originally intended, the team at Tamagoyaki Inc. are also getting much more acquainted with the operations of both version control and Git in particular. Unfortunately things don't always go as smoothly as we would like. The following scenario demonstrates just this.

> **In the trenches...** "Yeh but Martha, I need you to work on this fix now! Not in five minutes."
>
> "Klaus I'm kinda in the middle of something else yet, and I'm not ready to commit," said Martha, feeling a little concerned. She was used to Klaus making demands on his time, but she had spent a while working on this particular fix for John and she just wasn't ready to finish up yet.
>
> "You could always use the stash feature." It was Rob again, it seemed as if this young user had cottoned on to Git quicker than most of the seasoned developers."
>
> Klaus seemed unimpressed, "What the heck does that do?"
>
> "Allows you to move your changes to somewhere else until you are ready to finish them."
>
> "Please Rob," started Martha, "Can you show me what you mean? Sounds exactly like what I need"

Sometimes you could be in the middle of something when another really important task comes up. When this happens you are often required to drop everything and carry on with another task. Often this can be quite difficult. You may be in a development branch, but unable or unwilling to make a commit at this stage. So how can we deal with this?

Well, one way of dealing with this situation, and this is not necessarily the best way, is to do the steps outlined below;

1. Make a new branch called something like WIP
2. Pull changes into this new branch
3. Commit changes into WIP branch
4. Switch back to the branch we need to work on
5. Make our changes and commit them
6. Merge in our WIP branch on top with the `–no-commit` option
7. Delete the WIP branch
8. Continue development in our original branch

That may seem like a lot of work. OK, the benefit is a fairly awesome one, but at the cost of considerable command line hackery to get there. It would be nice if there was an easy way to do the above, and though someone of you may be screaming something like *shell script* right about now, you can rest your fingers. We actually have another tool in the Git toolbox to help us out.

The `git stash` command is used for exactly these situations. Let us make a quick example of how to use it in our test repository. To begin with we are going to make a few changes to the **master** branch, before we are interrupted.

```
john@satsuki:~/coderepo$ git checkout master
Already on 'master'
john@satsuki:~/coderepo$ echo "Number strings rule 1234" >> another_file
john@satsuki:~/coderepo$
```

So now we are interrupted and we have to make some important, and urgent changes to our **master** branch. The example that follows is one way that we could use the `git stash` command to help us out.

```
john@satsuki:~/coderepo$ git stash
Saved working directory and index state WIP on master: d50ffb2 Merged in zaney
HEAD is now at d50ffb2 Merged in zaney
```

```
john@satsuki:~/coderepo$ echo "Some mega important changes" >> newfile1
john@satsuki:~/coderepo$ git commit -a -m 'Important Update'
[master 9cb2af2] Important Update
 1 files changed, 1 insertions(+), 0 deletions(-)
john@satsuki:~/coderepo$
```

So using the `git stash` command, we have squirrelled all of our developmental changes away into a *stash*. Now we have completed the mega important change that just could not wait, we are ready to pull our changes back from the stash. First let us see just what the stash contains.

```
john@satsuki:~/coderepo$ git stash list
stash@{0}: WIP on master: d50ffb2 Merged in zaney
john@satsuki:~/coderepo$ git stash show stash@{0}
 another_file |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
john@satsuki:~/coderepo$ git stash show stash@{0} -p
diff --git a/another_file b/another_file
index dba885d..b3a5cc5 100644
--- a/another_file
+++ b/another_file
@@ -1 +1,2 @@
 New stuff
+Number strings rule 1234
john@satsuki:~/coderepo$
```

As you can see, the `-p` option to the `git stash show` command shows us exactly what is contained in the stash. We can apply this stash by running the following;

```
john@satsuki:~/coderepo$ git stash apply stash@{0}
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
# modified:   another_file
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
# my_third_committed_file
# temp_file
no changes added to commit (use "git add" and/or "git commit -a")
```

```
john@satsuki:~/coderepo$ git commit -a -m 'Continued Development'
[master 37950f8] Continued Development
 1 files changed, 1 insertions(+), 0 deletions(-)
john@satsuki:~/coderepo$ git stash list
stash@{0}: WIP on master: d50ffb2 Merged in zaney
john@satsuki:~/coderepo$
```

Our stash has been applied to the current branch, in our case **master**, and we have gone ahead and committed these changes into the repository. In this case we didn't have anything else to add to the index before we went ahead and committed it.

Interestingly though, our stash still exists. If we had used the `git stash pop` command instead of the `git stash apply`, our stash would have been removed. Of course we can have multiple stashes in our repository. For completeness sake, let us learn how to manually remove a stash, and take a look at how our repository looks with our semi-graphical `git log`

```
john@satsuki:~/coderepo$ git stash drop stash@{0}
Dropped stash@{0} (193f27172fc0df278105b981815c7718204030d8)
john@satsuki:~/coderepo$ git log --graph --pretty=oneline --all --abbrev-commit
 ↪--decorate
* 37950f8 (HEAD, master) Continued Development
* 9cb2af2 Important Update
*   d50ffb2 Merged in zaney
|\
| * 7cc32db (zaney) Made another awesome change
| * a27d49e Made an awesome change
* | ed2301b Removed third file
* |   b119573 Merge branch 'wonderful'
|\ \
| * | cfbecab (wonderful) Fantastic new feature
| |/
* | 4ac9201 Updated third file
|/
* 9710177 Added another file
* 55fb69f (v2.0) Added two new files
* 4a155e4 Removed a few files
* a022d4d (tag: v1.0b, v1.0a) Messed with a few files
* 9938a0c Finished adding initial files
* 163f061 (v0.9) Made a few changes to first and second files
* cfe23cb My First Ever Commit
john@satsuki:~/coderepo$
```

So using the `git stash drop` command, you can see that it is fairly simple to drop a single stash from the list.

Day 2 - "What?! No backup?"

Attack of the clones

We now know about basic branching and merging. At this stage, there is on important topic we must cover briefly, and this is subject of cloning. Cloning allows you to make a complete copy of your repository, including all of its history and all of the branches. Let us take a look at a situation which could make use of cloning.

In the trenches... "So John," started Rob, "Just how are we backing up the repository at the moment?"

John thought for a moment before replying. He knew what Rob was getting at, but he hadn't expected Rob to bring the question up in front of Markus. In truth he had forgotten all about it. He turned to Markus.

"I'll be honest Markus. Currently the repository isn't being backed up, but then we are running in parallel with the old system, so it wouldn't be the end of the world if we lost it."

Markus nodded and smiled. It seemed that John had gotten away with it for now and with that, Klaus shot Rob a piercing glance.

"Team," began Markus, "we need a definitive way of backing up the new Git repository, and I'd like it done before the end of the day." He pointed at a document on the table. "This project document has been approved by Wayne, and in there it states we will have a defined backup strategy. Please don't let me down."

* * *

"Rob you really got to be careful about things like that," Klaus said to one of the younger members of the team. "You really showed John up in there."

> "Yeh," said Rob, "I realise that now." He stood with his back against the wall and tapped his fingers against the painted surface. "I was wondering, do you think John would let me look at the backup system for him as a way of apologising."
>
> Klaus smiled, it seemed Rob was finally understanding things, "Go ask him," said Klaus, "He's so snowed under with the BurnForce release that he'll probably let you implement it too"
>
> "Right" nodded Rob and off he went.

So cloning is an excellent way of taking a copy of our repository, in essence it is a simple way of taking a backup of our repository and then with a little more work, we can keep that clone up to date. Obviously we could just take the files and copy them, but a better way of doing this is by utilising the `git clone` command. When cloning a repository, we take the entire structure and replicate it, creating an exact copy of the data in an alternative location. Well, that's what cloning means isn't it?

The git clone tool doesn't just copy the data though, it does several other things. Let us create a clone of our test repository to another local location. In this case, we are going to clone the repository into a folder called `coderepo-cl`.

```
john@satsuki:~$ git clone coderepo coderepo-cl
Initialized empty Git repository in /home/john/coderepo-cl/.git/
john@satsuki:~$ cd coderepo-cl
john@satsuki:~/coderepo-cl$ ls
another_file  newfile1  newfile2
john@satsuki:~/coderepo-cl$ git status
# On branch master
nothing to commit (working directory clean)
john@satsuki:~/coderepo-cl$
```

Let us take a few moments to see what has happened here. It would appear that our data has been copied successfully. If you were to run a `git log` on this dir, you would see all the previous log messages for all our previous commits. You can also see that if we run `git status` that we are on the same branch here that we were in our repository when we left it. In this case, we are in the **master** branch.

So this could be sufficient enough to serve as a backup of our repository, but we will actually find a better way to do this a little later.

Day 3 - "Is this clone for real?"

Not entirely as expected

We now have a clone of the repository and we can start to look at how we can play with branches and see how the two are different.

> **In the trenches...** "Klaus, this doesn't make sense though. I cloned my repo, but my branches have vanished," started Rob, rather worriedly.
>
> "I'm sure they haven't gone anywhere," shouted John.
>
> Rob's reply was quick and certain, "They have! Run a git branch on your clone and see for yourself"
>
> "Oh!"
>
> Rob smiled the smug smile of a man who is pleased he was right.
>
> * * *
>
> "Rob?" asked John tentatively, aware of his previous state of anguish.
>
> His reply indicated a less than stressed demeanour, "Sup, dude?".
>
> John was pleasantly surprised, "Did you sort out the branch issue?"
>
> Rob began walking over to John, "Yeh, try running git branch with the r parameter."

You may be thinking that we now have a detached copy of the other branches in our repository. That's not exactly accurate. Let us run the command that Rob suggested and see what is happening.

```
john@satsuki:~/coderepo-cl$ git branch
* master
john@satsuki:~/coderepo-cl$ git branch -r
  origin/HEAD -> origin/master
  origin/master
  origin/wonderful
  origin/zaney
john@satsuki:~/coderepo-cl$
```

Whilst we have all the objects in our repository, we do not yet have our branches set up locally. They are available, but they have not yet been set up locally. To explain this we need to introduce the concept of remote tracking branches. We are familiar with local branches. Local branches allowed us to make commits locally in a safe and separated environment. Remote tracking branches are links to remote branches which allow us to track the development of that branch and bring it in to a local one if we so desire.

Let us take a look at what we can do with these remote tracking branches. Notice in the output above, we have the word `origin` in front of the branch names. When we clone a repository, Git automatically sets up what is called a *remote*. We can view this by using the `git remote` tool.

```
john@satsuki:~/coderepo-cl$ git remote
origin
john@satsuki:~/coderepo-cl$ git remote -v
origin  /home/john/coderepo (fetch)
origin  /home/john/coderepo (push)
john@satsuki:~/coderepo-cl$
```

As you can see from the output above, the **origin** remote definition points to the original folder that our repository came from. So now we know that when referring to `origin/master`, we are really talking about the **master** branch which is located at the remote location **origin**.

We are going to spend a little while now learning about remote branches and how they differ to local branches. Let us spend a few minutes trying some of our normal operations against a remote branch. The first operation we are going to try is a diff. We are going to run a diff between our current **master** branch and the **wonderful** branch as it stands in the original repository, so we should be diffing between `master` and `origin/wonderful`. For brevity, we are also going to limit the changes shown to those which affect `newfile1` only, by appending `-- <filename>` to the command.

```
john@satsuki:~/coderepo-cl$ git diff master origin/wonderful -- newfile1
diff --git a/newfile1 b/newfile1
index f32a0e6..ef20984 100644
--- a/newfile1
+++ b/newfile1
@@ -1,3 +1,2 @@
 A new file
-and some more awesome changes
```

```
-Some mega important changes
+and some more changes
john@satsuki:~/coderepo-cl$
```

So we can see that our diff command completed successfully. Note that this command requires no connection to our **origin** repository, as all of the data has been cloned locally, more on this later.

Let us run another command now and learn a little more about what the remote reference **origin** really means.

```
john@satsuki:~/coderepo-cl$ git remote show origin
* remote origin
  Fetch URL: /home/john/coderepo
  Push  URL: /home/john/coderepo
  HEAD branch: master
  Remote branches:
    master    tracked
    wonderful tracked
    zaney     tracked
  Local branch configured for 'git pull':
    master merges with remote master
  Local ref configured for 'git push':
    master pushes to master (up to date)
john@satsuki:~/coderepo-cl$
```

Though this command introduces some terms that we are not yet familiar with yet, like push, pull and fetch, we can see that the **origin** remote has been set up to *track* three branches called, **master**, **wonderful** and **zaney**. So although these branches are not yet usable in the way a local branch would be, it is comforting to know that Git does know about them. Let us continue playing with the remote branch and run a `git log` command against it.

```
john@satsuki:~/coderepo-cl$ git log origin/master -- newfile1
commit 9cb2af2a00fd2253060e6bf8cc6c377b3d55ecea
Author: John Haskins <john.haskins@tamagoyakiinc.koala>
Date:   Tue Apr 19 16:43:59 2011 +0100

    Important Update

commit d50ffb2fa536d869f2c4e89e8d6a48e0a29c5cc1
Merge: ed2301b 7cc32db
```

```
Author: John Haskins <john.haskins@tamagoyakiinc.koala>
Date:   Fri Apr 1 07:42:04 2011 +0100

    Merged in zaney

...
...
...
```

Again, this requires no connection to the **origin** repository at all and works as expected. Let us try something really funky now, let us try checking out a remote branch so that we can view the file system. We are now going to try checking out the remote branch called **wonderful**.

```
john@satsuki:~/coderepo-cl$ git checkout origin/wonderful
Note: checking out 'origin/wonderful'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b new_branch_name

HEAD is now at cfbecab... Fantastic new feature
john@satsuki:~/coderepo-cl$
```

Interesting. This is not what we are used to seeing. When we have checked out branches before, we have always been presented with a `Switched to branch 'xxxxxx'` message. Now we are seeing a funny message about having a detached HEAD. This is because we are not actually on a branch. We can make changes as we are advised, we can even make commits, but these will not actually become integrated into any branches. In essence these commits would be left dangling and after time, because no branch points to them, they would get deleted by garbage collection routines.

Running the `git branch` command below confirms our suspicions.

```
john@satsuki:~/coderepo-cl$ git branch -v
* (no branch) cfbecab Fantastic new feature
  master      37950f8 Continued Development
john@satsuki:~/coderepo-cl$
```

So we really need to figure out how to make these branches exist locally so that we can commit to them and retain those commits. To do this we use the `git branch` tool again, but we use it in a slightly different manner.

```
john@satsuki:~/coderepo-cl$ git checkout -b wonderful origin/wonderful
Branch wonderful set up to track remote branch wonderful from origin.
Switched to a new branch 'wonderful'
john@satsuki:~/coderepo-cl$
```

We are now on a local instance of the **wonderful** branch. Let us take a minute to understand what that means. As Git is a distributed environment, our local **wonderful** branch, is not going to be synchronised with the remote end automatically. It would require effort on our part. What this does mean though, is that we are able to begin committing to this branch and experimenting with it.

Day 4 - "Help I'm no longer up to date?"

Pulling changes, not teeth

With a remote tracking branch, we can pull in the changes from the remote repository. In our case, the remote repository is in the `coderepo` directory that we created at the start of the book. In the `coderepo-cl` directory we have a separate, self-contained copy of the repository. At the moment, because no changes have occurred to either, they contain exactly the same data.

By running the `git remote` tool again, we can see which branches are set up locally to track their remote counterparts in **origin**.

```
john@satsuki:~/coderepo-cl$ git remote show origin
* remote origin
  Fetch URL: /home/john/coderepo
  Push  URL: /home/john/coderepo
  HEAD branch: master
  Remote branches:
    master    tracked
    wonderful tracked
    zaney     tracked
  Local branches configured for 'git pull':
    master    merges with remote master
    wonderful merges with remote wonderful
  Local refs configured for 'git push':
```

```
    master    pushes to master    (up to date)
    wonderful pushes to wonderful (up to date)
john@satsuki:~/coderepo-cl$
```

What we are going to do now is make some changes to our original repository and see how we can view those changes and indeed pull them into our current local working repository, or clone. Note that in the following code examples we have switched back to working on our original repository, in the `coderepo` folder.

```
john@satsuki:~/coderepo-cl$ cd ../coderepo
john@satsuki:~/coderepo$ git branch
* master
  wonderful
  zaney
john@satsuki:~/coderepo$ git checkout wonderful
Switched to branch 'wonderful'
john@satsuki:~/coderepo$ git merge master
Updating cfbecab..37950f8
Fast-forward
 another_file            |    1 +
 my_third_committed_file |    1 -
 newfile1                |    3 ++-
 newfile2                |    2 +-
 4 files changed, 4 insertions(+), 3 deletions(-)
 delete mode 100644 my_third_committed_file
john@satsuki:~/coderepo$ echo "These changes are in the origin" >> newfile3
john@satsuki:~/coderepo$ git add newfile3
john@satsuki:~/coderepo$ git commit -a -m 'Added a new file'
[wonderful 1c3206a] Added a new file
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 newfile3
john@satsuki:~/coderepo$
```

As you can see, we have switched back to working on our old **origin** and have bought **wonderful** to be in line with master and have added a new file and committed the changes. Now let us go back to our clone and see if we can see those changes.

```
john@satsuki:~/coderepo$ cd ../coderepo-cl/
john@satsuki:~/coderepo-cl$ git diff wonderful origin/wonderful
john@satsuki:~/coderepo-cl$
```

That seems a little odd at first glance. We have tried to view a diff between our local **wonderful** branch and the remote **origin/wonderful** branch. Interestingly, Git

is telling us that there is no difference. Hang on though, we just made some changes above.

Remember before we discussed the fact that our clone was disconnected from the **origin** copy? What this means is that unless we ask Git, it won't update details about what is present in the remote copy. Once this is up to date, we can pull the changes into our local copy of the remote branch? Interestingly there are actually two ways to do this.

The first method is by running a `git pull` command. If you remember the output from the `git remote show origin`, you may remember that it showed that there was a local branch configured for `'git pull'` and the details it gave for the **wonderful** branch were as follows; `wonderful merges with remote wonderful`. What this means, is that when we run a `git pull` from inside the **wonderful** branch, Git will automatically contact the remote repository, update the list of changes and merge them into the local branch if possible.

We mentioned that there were two methods to perform the procedure. Whilst the first one is to use `git pull`, the second one actually achieves an almost identical result, but by running two commands instead of one. In point of fact, these two commands are executed by the `git pull` command. Running them as two separate commands allows us to understand a little more about what is happening. Let us run the command and then explain what we have done.

```
john@satsuki:~/coderepo-cl$ git fetch origin
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From /home/john/coderepo
   cfbecab..1c3206a  wonderful  -> origin/wonderful
john@satsuki:~/coderepo-cl$ git diff wonderful origin/wonderful
diff --git a/another_file b/another_file
index dba885d..b3a5cc5 100644
--- a/another_file
+++ b/another_file
@@ -1 +1,2 @@
 New stuff
+Number strings rule 1234
...
...
...
```

```
john@satsuki:~/coderepo-cl$ git merge origin/wonderful
Updating cfbecab..1c3206a
Fast-forward
 another_file            |    1 +
 my_third_committed_file |    1 -
 newfile1                |    3 ++-
 newfile2                |    2 +-
 newfile3                |    1 +
 5 files changed, 5 insertions(+), 3 deletions(-)
 delete mode 100644 my_third_committed_file
 create mode 100644 newfile3
john@satsuki:~/coderepo-cl$
```

As you can see, the two commands that we used were `git fetch` and the more familiar `git merge`. The `git fetch` command literally visits the remote repository, in our case we asked for **origin**, and finds which objects are new and do not exist in our local clone. These are then copied to the local clone, and the HEADs of the various remote branches are updated. We now rerun our `git diff`, only this time, we see a great many more changes than before. Finally we initiate a `git merge` to pull the changes from the now up to date remote branch to our local one.

We could have used the `git pull` command here, and if we had supplied no parameters to it, it would have achieved the exact same outcome. The only difference would have been running one command instead of two. Please note, that there are occasions when you would require one over the other, but as an introduction, this should be sufficient.

Day 5 - "I'm putting my foot down"

Pushing back!

We now know a lot about remote locations right? We are ready to start collaborating with someone. Let us first see how Tamagoyaki are getting on with things.

> **In the trenches...** "But I know I have the remote right Simon, it's giving some weird error about refusing to update it." Rob was getting more than a little cross now.
>
> "Maybe you got the command wrong?" asked Simon.

> Rob slammed his hands down on the keyboard, "Well then you come over here and type it, but I don't understand how hard it is to write git push." He shook his head, "I even checked using the git remote tool and the push is apparently all set up."
>
> "Are you trying to push to a non-bare repository?" Chirped Klaus. He had been listening to the discussion escalate from mild annoyance to key shattering intrusion. "You can't push to a non-bare repository, else everything gets out of sync."
>
> "What do you mean bare?" asked Rob.
>
> Klaus smiled and his head appeared over the cubicle. "Come over here genius and I'll show you," he said with the slightest amount of gloat at being one of the only people to know something about Git that Rob didn't.

Bare? Non-bare? At first glance, this may seem confusing. It isn't exactly an intuitive word to describe a repository but what Klaus mentioned was absolutely true. In Git, if you want to push changes back to a repository, instead of pulling them, the repository that you push to should be what is called a *bare* repository. What this means is that the repository has no working copy.

At first glance this may seem like a rather odd thing to want. Why would we want a repository that doesn't have a working tree? The simple answer is that a *bare* repository allows us to be able to push changes into it. Let us recreate the error message that Rob was talking about.

```
john@satsuki:~/coderepo-cl$ echo "This is another update to newfile3" >> newfile3
john@satsuki:~/coderepo-cl$ git commit -a -m 'Added more to newfile3'
[wonderful dbf1e9a] Added more to newfile3
 1 files changed, 1 insertions(+), 0 deletions(-)
```

We have made a modification to the **wonderful** branch. So let us now try to push that back to the **origin** remote repository with the `git push` command.

```
john@satsuki:~/coderepo-cl$ git push
Counting objects: 5, done.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 335 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
```

Note

Pushing to a non-bare repository

As mentioned, pushing to a non-bare repository is not a good idea. When you push to a repository, you update the objects in the objects database. This will affect commits, blobs and trees. We should make a distinction here. We are really only worried about pushing to a branch which is currently checked out. Pushing to a non-bare repository will mean that the index will get changed, as it should reflect what the branch looks like at HEAD.

Now this causes a problem when we are trying to see what has been modified in the working directory when compared to the HEAD. As our working copy will contain older versions of the files than are in the index, it will appear as if many more changes have occurred and we risk undoing those changes. For these reasons, it is much simpler just to push only to bare repositories.

```
Unpacking objects: 100% (3/3), done.
remote: error: refusing to update checked out branch: refs/heads/wonderful
remote: error: By default, updating the current branch in a non-bare repository
remote: error: is denied, because it will make the index and work tree inconsistent
remote: error: with what you pushed, and will require 'git reset --hard' to match
remote: error: the work tree to HEAD.
remote: error:
remote: error: You can set 'receive.denyCurrentBranch' configuration variable to
remote: error: 'ignore' or 'warn' in the remote repository to allow pushing into
remote: error: its current branch; however, this is not recommended unless you
remote: error: arranged to update its work tree to match what you pushed in some
remote: error:  other way.
remote: error:
remote: error: To squelch this message and still keep the default behaviour, set
remote: error: 'receive.denyCurrentBranch' configuration
remote: error: variable to 'refuse'.
To /home/john/coderepo
 ! [remote rejected] wonderful -> wonderful (branch is currently checked out)
error: failed to push some refs to '/home/john/coderepo'
john@satsuki:~/coderepo-cl$
```

Bingo! There is the error we were looking for. In order to get round this, we are going to create another clone of the repository in `coderepo`. This time however, we are going to pass an option to it when we create it.

```
john@satsuki:~/coderepo-cl$ cd ..
john@satsuki:~$ git clone coderepo coderepo-bk --bare
Initialized empty Git repository in /home/john/coderepo-bk/
john@satsuki:~$ cd coderepo-bk/
john@satsuki:~/coderepo-bk$ ls
branches  config  description  HEAD  hooks  info  objects
packed-refs  refs
john@satsuki:~/coderepo-bk$ cd ..
john@satsuki:~$ cd coderepo-cl/
john@satsuki:~/coderepo-cl$
```

As you can see, creating the clone with the –bare option has had a definite effect on the structure of the clone itself. In fact, the contents of the `coderepo-bk` folder look remarkably similar to the `.git` folder found in our normal working tree. They are in fact, one and the same. We have removed the need for a separate `.git` folder because we do not require a working tree. Therefore, the contents of the `.git` folder and placed in the root `coderepo-bk` folder.

We are getting closer to making our push. We have created a *bare* repository, which we should now be able to push to. The problem is, we have not yet defined our new repository as a remote location in the `coderepo-cl` repository. This is something we will do now and to do this, we will use our `git remote` tool once more, this time employing the `add` parameter.

```
john@satsuki:~/coderepo-cl$ git remote add backup
 /home/john/coderepo-bk
john@satsuki:~/coderepo-cl$ git remote show backup
* remote backup
  Fetch URL: /home/john/coderepo-bk
  Push  URL: /home/john/coderepo-bk
  HEAD branch: wonderful
  Remote branches:
    master    new (next fetch will store in remotes/backup)
    wonderful new (next fetch will store in remotes/backup)
    zaney     new (next fetch will store in remotes/backup)
  Local refs configured for 'git push':
    master    pushes to master    (up to date)
    wonderful pushes to wonderful (fast-forwardable)
```

```
john@satsuki:~/coderepo-cl$
```

We have created a remote called backup. As you can see, some interrogation of the remote repository has taken place. It is already aware of the branches present in our remote location. Let us do a `git fetch` to update the local references.

```
john@satsuki:~/coderepo-cl$ git fetch backup
From /home/john/coderepo-bk
 * [new branch]      master     -> backup/master
 * [new branch]      wonderful  -> backup/wonderful
 * [new branch]      zaney      -> backup/zaney
john@satsuki:~/coderepo-cl$ git remote show backup
* remote backup
  Fetch URL: /home/john/coderepo-bk
  Push  URL: /home/john/coderepo-bk
  HEAD branch: wonderful
  Remote branches:
    master    tracked
    wonderful tracked
    zaney     tracked
  Local refs configured for 'git push':
    master    pushes to master    (up to date)
    wonderful pushes to wonderful (fast-forwardable)
john@satsuki:~/coderepo-cl$
```

Now we can see that all of the remote references for our **backup** remote have been updated. If we run a diff against our local **wonderful** branch and the remote branch **backup/wonderful**, we can see the differences.

Note

Pushing your tags

By default tags are not pushed when using the `git push` command. If we want our tags to be push to the remote repository as well, we need to append the `–tags` parameter to the `git push` command.

In contrast, using `git pull` will automatically pull down any tags which refer to any objects that have been pulled. This behaviour can be overridden by using the `–no-tags` parameter with `git pull`.

```
john@satsuki:~/coderepo-cl$ git diff wonderful backup/wonderful
diff --git a/newfile3 b/newfile3
index 7268b97..638113c 100644
--- a/newfile3
+++ b/newfile3
@@ -1,2 +1 @@
 These changes are in the origin
-This is another update to newfile3
john@satsuki:~/coderepo-cl$ git branch -v
```

There we go! Those are the changes we just committed to the local repository. The last thing we need to do is to initiate a `git push`. We are going to have to define which remote we wish to push to now for two reasons. The first is that now we have two remotes, and so Git would not know if we meant **origin** or **backup**. The second reason is that by default, the **wonderful** branch is set up to push to the **origin** remote.

```
john@satsuki:~/coderepo-cl$ git push backup
Counting objects: 5, done.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 335 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
To /home/john/coderepo-bk
   1c3206a..dbf1e9a  wonderful -> wonderful
john@satsuki:~/coderepo-cl$
```

There we go, we have now managed to push our changes to a remote location. In this case, we have pushed an update from the **wonderful** branch to the **backup/wonderful branch**. If we wanted to, we could have pushed our repository to a new branch name, or indeed any of the existing branch names.

```
john@satsuki:~/coderepo-cl$ git push backup wonderful:newbranch
Total 0 (delta 0), reused 0 (delta 0)
To /home/john/coderepo-bk
 * [new branch]      wonderful -> newbranch
john@satsuki:~/coderepo-cl$ git remote show backup
* remote backup
  Fetch URL: /home/john/coderepo-bk
  Push  URL: /home/john/coderepo-bk
  HEAD branch (remote HEAD is ambiguous, may be one of the following):
    newbranch
    wonderful
```

```
  Remote branches:
    master    tracked
    newbranch tracked
    wonderful tracked
    zaney     tracked
  Local refs configured for 'git push':
    master    pushes to master    (up to date)
    wonderful pushes to wonderful (up to date)
john@satsuki:~/coderepo-cl$
```

By specifying the local and remote branch using the `local:remote` syntax, we have told Git to push our local **wonderful** branch to the remote branch called **newbranch**. As the **newbranch** did not exist in the remote **backup** repository, it was created, as can be seen above.

Note

Killing a remote branch

Now that we have made branches on the remote end, it is a good idea to know how to delete them also. After all, if all you ever did was push new branches, pretty soon you may end up with a lot of rubbish in your repository. To delete a remote branch, all that you need to do is to use the following syntax.

```
git push <remote> :<branchname>
```

Notice that we are using the same syntax as before, but not specifying a local branch name, just the remote. Git interprets this as a call to delete the remote branch.
If you have a clone of a remote repository and a remote branch is removed, you may see a message like the following if you run `git remote show <remote>`;

```
 refs/remotes/<remote>/<branchname> stale (use 'git remote prune' to remove)
```

So we can run `git remote prune` to remove local references to remote branches that no longer exist. If you have created a tracking branch of this remote branch, this will remain unaffected.

So this ends the Week. We have learnt a lot more about how to play with branches and remotes. In the *After Hours* section, we will take a deeper look at remote branches

and how they are configured and represented in Git.

Summary - John's Notes

Commands

- `git stash` - Short for `git stash save`, creates a stash of local modifications
- `git stash apply <stash_name>` - Applies a stash back onto a branch
- `git stash drop <stash_name>` - Remove a stash from the stash list
- `git stash list` - Show a list of current stashes
- `git stash show <stash_name>` - Show information about a specific stash
- `git stash show <stash_name> -p` - Show the contents of a stash
- `git clone <local> <remote>` - Clone a Git repository, from the local to the remote location.
- `git branch -r` - Show all remote branch references
- `git remote` - Show a list of remote repositories
- `git remote -v` - Show a more detailed list of remote repositories
- `git diff <branch> <remote>/<branch> – <file>` - Show the differences between the two branches, one local, one remote, for a particular file.
- `git remote show <remote>` - Show detailed information about tracked branches for the specified remote repository
- `git log <remote>/<branch> – <file>` - Show the log of the remote branch specified for a particular file
- `git checkout <remote>/<branch>` - Checkout a remote branch, leaving the local working tree in a detached HEAD state
- `git checkout -b <branch> <remote>/<branch>` - Create and switch to a new branch, which is set up to track a remote branch specified
- `git fetch <remote>` - Fetch updates to the remote branch specified including all objects and branches

- `git pull` - When configured, fetches and merges a remote branch into the currently checked out branch.
- `git clone <local> <remote> –bare` - Creates a bare repository, suitable for pushing into
- `git push` - When configured, pushes all changes up to the configured remote branch
- `git push <remote> <local_brch>:<remote_brch>` - Pushes a local branch to a remote location with a different name

Terminology

- **Stash** - A temporary storage of local modifications that can be brought back onto the branch at a later date
- **Remote** - A copy of a repository, or part of a repository in a remote location
- **Pull** - Merging changes from a remote branch into a current local one
- **Push** - Pushing changes from a local branch into a remote one
- **Fetch** - Pull into the local repository, all new objects and update all branch HEADs to point to the same commits as the remote location

After Hours Week 6

"Tug of war"

Taking the push with the pull

We have spoken at fairly great length about how remote repositories work. We have seen how the `git remote` tool is used to create the various references to remote repositories, but we have no real understanding about what this means in terms of Git's internals. Just in the same way a branch is a single file that contains a pointer to a reference, a remote repository has to be handled within Git somehow.

As it happens, Git again uses a reasonably simplistic design when creating remote references. To take a look at this in detail, we need to once again delve into the `.git` directory. Seeing as our original repository does not contain any remotes for now, we are going to use our `coderepo-cl` folder as an example. Hopefully, if you have been following the text, you have not deleted this directory yet. If you have, do not worry, just follow the operations we completed in Week 6, or read on and use the text in the book.

If you remember, we created two clones of our original repository. One of these was a simple clone called `coderepo-cl` and the other was a bare repository called `coderepo-bk`. The `coderepo-cl` and the `coderepo-bk` repositories were both cloned from `coderepo`, but it was `coderepo-cl` that was configured to pull from one and push to the other. Running a simple `git remote -v` command, confirms this configuration.

```
john@satsuki:~/coderepo-cl$ git remote -v
backup  /home/john/coderepo-bk (fetch)
backup  /home/john/coderepo-bk (push)
origin  /home/john/coderepo (fetch)
origin  /home/john/coderepo (push)
john@satsuki:~/coderepo-cl$
```

We can get even more information by running the `git remote show` tool with the remote name as a parameter.

```
john@satsuki:~/coderepo-cl$ git remote show origin
* remote origin
  Fetch URL: /home/john/coderepo
  Push  URL: /home/john/coderepo
  HEAD branch: master
  Remote branches:
    master    tracked
    wonderful tracked
    zaney     tracked
  Local branches configured for 'git pull':
    master    merges with remote master
    wonderful merges with remote wonderful
  Local refs configured for 'git push':
    master    pushes to master    (up to date)
    wonderful pushes to wonderful (up to date)
john@satsuki:~/coderepo-cl$
```

How though, is this data set up and configured from within Git itself? Looking at the `.git/config` file, we can see a glimpse of this.

```
john@satsuki:~/coderepo-cl$ cat .git/config
[core]
  repositoryformatversion = 0
  filemode = true
  bare = false
  logallrefupdates = true
[remote "origin"]
  fetch = +refs/heads/*:refs/remotes/origin/*
  url = /home/john/coderepo
[branch "master"]
  remote = origin
  merge = refs/heads/master
[branch "wonderful"]
  remote = origin
  merge = refs/heads/wonderful
[remote "backup"]
  url = /home/john/coderepo-bk
  fetch = +refs/heads/*:refs/remotes/backup/*
john@satsuki:~/coderepo-cl$
```

As you can see there are two relevant sections, these are the `remote` and `branch` stanzas. The `remote` stanzas are there to define certain elements of the remote repository, whilst the `branch` stanzas describe elements and settings for the branches that we are

tracking. Let us look at the relevant elements that we have set out in `config` file for each type of stanza.

We will start by looking at the `remote` stanza. We have two settings here in our configuration. The first is `url` and the second is `fetch`. The `url` setting tells Git the location of the remote repository defined by the name outlined within the quotation marks. In our example, the remote we first encounter is called **origin** and has a url of `/home/john/coderepo`.

You will notice that there is also a second setting called `fetch`. This setting tells Git which branches, or HEADs we are interested in. In the default configuration, we ask Git to fetch all branch HEADs. This also has an effect on which objects are downloaded. Git will look at all of the branch HEADs and work backwards to find out which commit objects need to be downloaded.

In the `refs/` folder there are two other folders by default called `tags` and `remotes`. With the default configuration as above, these references will not be fetched, and commits that are only pointed to by these references will not be downloaded either.

Looking at the `branch` settings, we can see that there are a few more settings here that describe how Git responds to certain commands when inside a certain branch. In both the cases above, Git will automatically use remote of **origin** as the remote to use when performing operations. This means that when we run a `git pull`, we do not have to specify a remote to pull from. Obviously if we wanted to, we could change this and have **wonderful**, for example, update from a different remote repository.

The last setting in this file that we will concern ourselves with is the `merge` setting. When we are in the named branch, this setting defines the upstream version of the named branch and is used for merging and can affect pulling and rebasing too.

We can set these settings and any of the others in the `config` file manually, using the `git config` tool, which we used in a previous chapter.

"Referring to objects"

The Git spelling bee

It is now most definitely time we spoke about other ways to represent commit hashes. More details can be found about this in the Git manual, but it is definitely worth spending a few minutes looking at the ways in which we can specify objects in the repository. As you have seen in previous chapters, we have used branch names, commits hashes and tags to specify commits, but it is also possibly to use a variety of other

methods to do so. We are going to use the `git rev-parse` command again to return us an object hash from our description of an object. These descriptions are ways to *spell* objects.

- `<SHA1>` - This is most common way to specify an object. The `<SHA1>` is the 40 character identifier that Git generates for each object. As you have seen before, we do not have to specify the entire SHA1 hash, just the beginning portion that is unique.

- `<refname>` - We use this type of referencing a lot when checking out branches. The `<refname>` is a symbolic name. An example of this would be **master**, which actually refers to **refs/heads/master**, which you have seen in the `.git` directory. HEAD is also a `<refname>`. In general, Git will search through a number of directories to find the `<refname>` that is being referred to;

  - `.git/<name>`
  - `refs/<name>`
  - `refs/tags/<refname>`
  - `refs/heads/<name>`
  - `refs/remotes/<name>`
  - `refs/remotes/<name>/HEAD`

  So if we used **master** as our `<refname>`, Git would search in `.git` root directory first, then into `refs`, followed by `refs/tags`, and finishing with `refs/heads`.

- `<refname>@{date}` - Now we find a more interesting way of specifying references. We can use something like **master@{yesterday}**, to show us the closeset commit to match that date time. There are more complicated date specifications we can use as well, such as **master@{"last week"}**, or **master@{"3 hours 2 minutes and 10 seconds ago"}**. We can even put in a specific date and time like so; **master@{"2011-02-26 14:30:00"}**

- `<refname>@{<n>}` - This curious definition returns the commit that `<refname>` referred to `<n>` times in the past. It uses the *reflog*, which has been discussed before, to discover what commit `<refname>` pointed to. Be careful when using this reference. It does not mean the commit that `<refname>` pointed `<n>` commits

ago in the tree. If you have been doing resets and other things, these items show up in the *reflog*.

- `<rev>^<n>` - Is a way of asking Git to traverse an object for it's parents and so `<rev>` or `<rev>^1` means the first parent of a commit object. An example of this would be `master^2`.

- `<rev>~<n>` - Is a way of asking Git to traverse an object for it's `<n>`th grandparent, following only first parents. This will take a little understanding and it is advised that you read the man page online for more information about what this really means.

- `<rev>^{/<text>}` - This definition actually initiates a search for the youngest commit where the commit message matches the regular expression after the slash. `master^{/bug}` is an example of the usage of this reference definition.

- `<rev>:<path>` - This allows us to obtain the object hash for the file specified at the `<rev>`. We could then use the `git show` command to view that file. As an example `git show HEAD~3:readme.txt` would show us the file `readme.txt` as it was three grandparents back from HEAD.

All of these are valid ways to refer to commits and in some cases objects and trees too. Imagine a situation where you wanted to view a file called `readme.txt` that was three commits back from a tag of `v44`. Using our new knowledge, we could use `git show v44~3:readme.txt`. There are several other ways of referring to commits, but these are out of the scope of this chapter. If you would like more information, refer to the man page for `git rev-parse`.

Week 7

Day 1 - "Networking with a difference"

Pushing across a LAN

Now we have a complete copy of our repository in another location. At the moment we have created this clone on the same machine that our original is. This isn't really a very good idea for backup purposes. Git supplies several means with which to talk to a remote machine, but by far the most common of these is to utilise the SSH protocol. SSH is a secure, encrypted way to communicate with a remote repository. Which is a must for pushing to an important repository that people are going to pull information from.

If we assume that for a moment that our user *john* has now moved to another machine and now wishes to clone a repository that he had on his original machine to this new one. The commands are identical to that which we used before. We are going to assume that *john* already has SSH access to the machine. In this way, we can issue the commands as follows.

```
john@akira:~$ git clone ssh://john@satsuki/home/john/coderepo coderepo-ne
Initialized empty Git repository in /home/john/coderepo-ne/.git/
john@akira's password:
remote: Counting objects: 53, done.
remote: Compressing objects: 100% (36/36), done.
Receiving objects: 100% (53/53), 4.84 KiB, done.
Resolving deltas: 100% (10/10), done.
remote: Total 53 (delta 10), reused 0 (delta 0)
john@akira:~$
```

Now we have done exactly as before and cloned our repository to a local folder called `coderepo-ne` from the remote URL `ssh://john@satsuki/home/john/coderepo`. Notice the use of `ssh://` to denote the specification of the SSH protocol. We have also put the users name in the URL of the remote path. If the SSH server was running on a different port to usual, that is, on a

port other than 22, we could have also added a port number preceded by a colon after the hostname.

SSH isn't the only protocol that Git can use. We have already looked at two; local and SSH. In fact, Git supports a further two protocols and these are HTTP/S and Git's own GIT protocol. We are going to take a quick look at the Git protocol next, before moving on to HTTP/S.

The Git Protocol

The GIT protocol is the fastest transfer protocol out there for Git repositories. This should come as no surprise, since it was developed exclusively for use within a Git environment. It does however have a relatively large drawback. The drawback is that it provides absolutely no authentication. For this reason, enabling the GIT protocol on a repository and running the server back-end, (described later), will allow anyone who can talk to the servers port complete read access to the repository.

If you are serving a large repository on the Internet for example, this could actually be rather beneficial and will allow you to serve pulls quickly and efficiently. However, though it is possible to enable *pushing* using the GIT protocol, the lack of security would mean that anyone who could see the server and connect to the port, usually 9418, could make changes to the repository. This is usually entirely undesirable and as such people will often couple a read-only GIT protocol with a writable SSH protocol for the developers that need push access.

The HTTP/S Protocol

Just as with the GIT protocol, Git can support the HTTP/HTTPS protocol as well. Setting up this is usually as simple as creating a bare clone of your repository, keeping it up to date, usually via a post-update-hook, which is described later in the book, and simply allowing clients access to this server area.

Note that the above is only to provide read-only access over HTTP. It is possible to allow write access, i.e. pushing, over HTTPS, but this is more complicated to set up and is outside the scope of this book.

Protocol decision

Tamagoyaki are about to embark on their decision making process regarding which protocols to use and how to perform their collaboration between themselves and their external partners. They are going to have to take multiple things into consideration, such

as security, speed, administration and storage. When you begin to implement the Git system yourself, you too will have to think about these decisions and answer questions like:

- Who is going to require access to the repository?
- How many people are going to require access to the repository?
- Is the information sensitive, either from an IP perspective or from a customer point of view?
- How large is the data that we are hosting?
- How large is the change set?
- Do we need a QA area?
- Do we need a Production area?

This is just a short list of the questions that you will need to consider when implementing a full on Git environment. The beauty of the Git system though, is that it is flexible and very difficult to box yourself into a corner, where a decision made early on prohibits a different approach later on.

Day 2 - "Now let's work together"

Pure collaboration

We are now armed with a much clearer idea of how Git works and indeed we are now in a position to actually implement the developmental model that the team of Tamagoyaki need in order to collaborate on their projects. It should be noted that although we have reached the point of being able to work together on a project, this is not where our discussions about Git will end. We still have a number of topics to cover and these will be visited as required during the subsequent implementation of Git at Tamagoyaki.

> **In the trenches...** "So I still maintain that we follow the original plan," said Rob. "Each person has their own repository and is the master of their own commits"

John shook his head. "On a small scale," he began, "that might work." He paused for a breath. "But we need to think about scalability too. Whilst I don't think all development work should be in one repository, I also don't think the best way to go is to have a repo for every person."

"I know it's a rarity, but I gotta agree with John." Klaus spoke whilst idly stabbing his pen lightly into a blob of what looked like modelling clay. "Think what the chimp would say to it too."

Martha furrowed her brow, "The chimp?" she asked.

John turned to look at Martha and almost regretted having to inform her of yet another of Klaus' pet names. "That's what Klaus calls Jimmy in IT."

Martha looked a little horrified, "Klaus, that's an awful thing to say."

"What? He spends all day monkeying around in the 'datacenter'. I've never really seen him do any real work at all." Klaus was looking at Martha who now had one eyebrow raised. He had used air quotes when voicing the word 'datacenter'. "Plus he calls it a datacenter, but I've seen inside, it's more like a cupboard with a PC in it."

"You need to show a little more respect Klaus," Martha threw back, a little more aggressively than she had intended.

"Guys!! Guys!!" John shouted. "Can we get back to the topic at hand and deal with Klaus later?"

The room fell silent for a while, until the comments had transitioned from immediate to short term memory. "How about doing it by team?" It was Eugene speaking now. "You know, kinda like a Mob repository. Each team will have their own repository and the branches inside will belong to the different team members and be named accordingly so. Then we would have a company repository which would hold the projects and would have dev, qa and released branches."

"You know that's not a bad idea Eugene." Klaus said, trying to redeem himself. "You da man. How about a hi-five."

The idea of having a Mob repository is not anything new. Many people decide to split up their teams in this way especially within a company. For larger repositories

it makes a lot of sense as not only does it help to keep development altogether in one place, but it also saves on space and administration overhead. Managing many smaller repositories is often a lot more time consuming than managing several larger ones.

However you should always look at the situation and the scenario carefully to see which is going to suit you best. For Tamagoyaki, there is the prospect of the teams growing soon and so they require a way to get the data organised fairly quickly and effortlessly. Having many small repositories would likely confuse new members to the team, so the decision to move forward with team based and site based repositories makes sense.

Tamagoyaki are proposing to have a single repository which will be the *blessed* repository we discussed way back at the beginning of the book. This repository will likely hold three branches. These will relate to development, quality assurance and releases. To this end it will allow one team to gain access to features that the other teams are working on, via the development branch. It will allow a QA manager to work on the quality assurance branch and finally, it will allow someone to push these changes through to the released branch.

The idea is that all development work stays in the development branch. For Tamagoyaki Inc. this branch will likely contain mostly finished, but largely untested works. The teams will work together inside their mob repositories to create new functionality. Once they are happy, their team leader will push their changes up into the development branch on the *blessed* repository. This branch is not guaranteed to be stable. Sometimes things will break but the development branch is a place that all of the work of the various teams comes together.

Once the teams hit feature freeze, the point at which they will no longer take any more new features into the code base, they will ask the QA manager to create a **qa** branch. Bug fixes for current issues will be committed here and when all bugs have been fixed and the code is ready to be released, the release branch will be synced with a specific commit of the **qa** branch.

If this all sounds a little confusing, we will take a few minutes to digest what we have come up with and draw a few diagrams. The beauty of Git is that it is so configurable and so by design supports almost limitless workflows. Unfortunately this can also be a drawback because no two Git setups are generally alike. The set up of Git is totally customised to the situation or scenario at hand. In our case, we have merged together two of the workflow models that we presented in the early chapters.

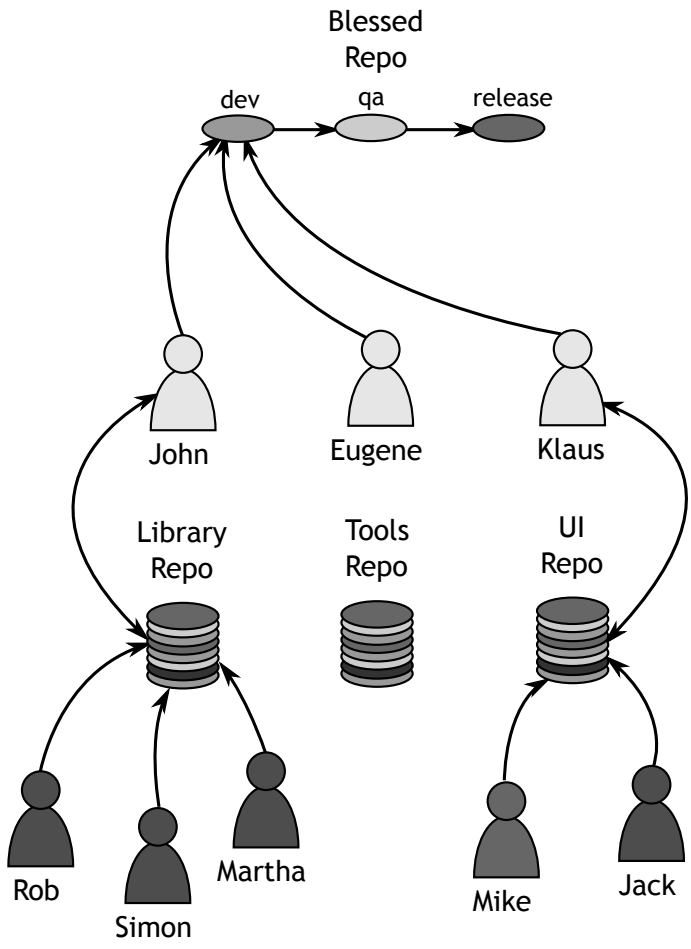


Figure 1: Our workflow

We are using a *blessed* repository with which to store our crown jewels. This is the repository that contains our main **release** branch. This branch will always contain reliable stable releases. Sure, it may contain history of how we reached the stable release, but on a checkout, it will be a solid, buildable, pristine, tagged release. This repository also contains our **qa** branch. This is the branch that upon checkout would contain a fairly polished version of the product, but it may still have a few bugs and issues. Then we

have our development branch. This is where the integration managers from the various teams would pull the code from their teams into a single code base, ready for the QA manager to begin his testing. This setup may not work for you. You may require more levels, you may require less. The fact of the matter is that Git allows you to make that distinction.

For the developers, we have the team mob repositories. These repositories will contain multiple branches that the various team members will push to, so that the integration managers are able to pull those changes into the **dev** branch on the *blessed* repository.

One item we have not yet touched upon is how to bring changes in from one branch to another. We have one method which we met earlier, called *Merging*. However there are other options open to us, such as patching and rebasing. Why are they important you might ask? Sometimes merging is not the best way to approach the situation.

> **In the trenches...** John was beginning to get a little frustrated now. Some members of the team were clearly not understanding what he was saying and he was fighting hard not to raise his voice. "It is just messy for me to review is all I am saying. I would much prefer not to have to see all those merge commits in the history when you are developing a simple feature."
>
> "We could always look at using a rebase." Martha seemed anxious to calm John down. "I haven't really played with it enough yet, but rebase could be what we are looking for."
>
> "What does rebase do when it's at home." Sneered Klaus, rolling his eyes.
>
> Martha picked up the red board pen from the board room table and gingerly took off the lid. She walked over to the whiteboard. "For small topics or features," she began, "where you are not going to have to publish it or you need to take a long time developing, it would make sense to use rebase." She began drawing little diagrams with circles on the board. "Rebasing will allow you to take your branch, pick up all the development you have done, update the branch underneath to bring it in-line with your blessed dev branch for example, and then replay your development on top."

> "Won't that change your history? Isn't that a big big no no?"
>
> "Yes it will change your history, and that's why for some things it isn't appropriate. But it may be just the ticket for what we are trying to do."
>
> John smiled. He was beginning to feel like the team members were starting to embrace Git's potential.

As you know, each time you merge you create a merge commit. Whilst this is not a problem, it can leave the tree looking messy. If you are working on a small feature in a branch, and you want to get the latest features and updates from the dev branch, but you do not want the hassle and untidiness of generating a ton of merge commits. The best way to get round this is to use the rebase tool.

As mentioned, the rebase tool can find a common ancestor between two branches, pick up the new commits on your new branch, update the branch underneath and the replay your commits on top. Figure 2, shows a fairly standard commit tree. We have made commits **A** and **B**, and then we branch off and create **C** and **D**. It could well be that the **master** branch containing **A** and **B** actually came from an external source, and we are looking to do some development in our own branch locally.

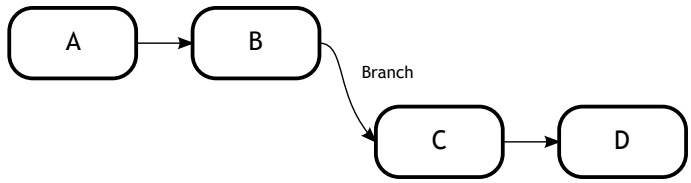


Figure 2: Standard commit tree example

Now what will inevitably happen is that some more development will continue in the source project, which is represented in the Figure below by commits **E** and **F**. This is shown below in Figure 3. In our use case, we actually want these new changes to be present during our development. Let us see how we can handle this.

We could quite happily merge the **master** branch into our own, but that would result in a merge commit being added, plus the fact that as we go further down the line, our development work would be interlaced with various updates from the **master**. A better way to handle this sometimes, is by using the rebase tool which will achieve the results displayed below in Figure 4.

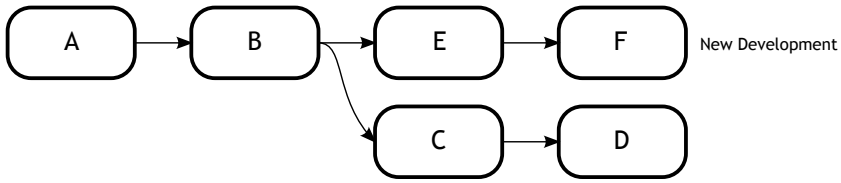


Figure 3: Continued development

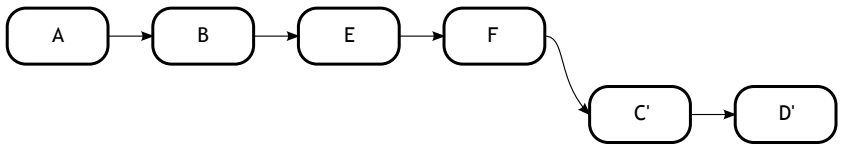


Figure 4: A rebase in action

As you can see we have a slight alteration to the diagram. Commits **C** and **D** have little ticks next to them. This is to indicate that they are actually not the same commits as before. You may be wondering why? Think about it for a few minutes. In Git, every commits SHA-1 hash is based not only on the contents of that commit, but also on the parent. In this way, if we move these commits, then their SHA-1 hashes are going to change. Our notation is simply stating that although the contents of these commits is the same, they are the **C** and **D** that we remember, their identifier will change.

Remember our discussion about changing the past? This is very relevant here. Rebasing is the ultimate way to change the past. Over the next few days, we are going to demonstrate a number of ways that rebase can aid you in your development and produce a clean and structured commit history. However, care must be taken not to rebase something which others have already pulled. Remember, with great power, comes great responsibility.

Let us now turn out attention to actually looking at an example of the rebase tool. We are going to take our example repository and create a branch. Then we are going to do some work on the **master** branch and try rebasing our changes.

Day 3 - "Rebasing our commitments"

Rebase examples

We are now going to go back to our `coderepo` folder. This is the one we cloned from in Week 6 to create the `coderepo-cl` repository. When we left this repository, we were at commit **1c3206a**. If you have played with the repository at all, it would be a good opportunity for you to bring the wonderful branch to that point. We are also going to Fast Forward the master branch to that same point too. Before reading on, try to remember what commands we would use to do this and then check below to see if you are right.

```
john@satsuki:~/coderepo$ git checkout wonderful
Already on 'wonderful'
john@satsuki:~/coderepo$ git reset --hard 1c3206a
HEAD is now at 1c3206a Added a new file
john@satsuki:~/coderepo$
```

So we begin by ensuring that we are on the **wonderful** branch. Then we perform a hard reset, to ensure that the HEAD of **wonderful** points to the commit **1c3206a**.

```
john@satsuki:~/coderepo$ git checkout master
Switched to branch 'master'
john@satsuki:~/coderepo$ git merge wonderful
Updating 37950f8..1c3206a
Fast-forward
 newfile3 |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 newfile3
john@satsuki:~/coderepo$
```

Next we use the `git merge` command as before to bring the **master** branch in line with the **wonderful** branch. As there have been no more commits to **master** since we worked on **wonderful**, it means that once again, the merge can be a simple Fast Forward merge, allowing us to simple change which commit HEAD points to within the **master** branch.

```
john@satsuki:~/coderepo$ git branch -v
  master    1c3206a Added a new file
* wonderful 1c3206a Added a new file
  zaney     7cc32db Made another awesome change
john@satsuki:~/coderepo$
```

Finally, we run a `git branch` command to confirm that the branches point to the same commit. Remember there are other ways we could have achieved this, using `git log` for example. Now that we can see they point to the same place, we can continue with our introduction to the `git rebase` command. We will start by making some commits to the **wonderful** branch and see how we can update our underlying master branch.

```
john@satsuki:~/coderepo$ echo "New Changes" >> another_file
john@satsuki:~/coderepo$ git commit -a -m 'Updated another file'
[wonderful c0e2f5b] Updated another file
 1 files changed, 1 insertions(+), 0 deletions(-)
john@satsuki:~/coderepo$ echo "More New Changes" >> another_file
john@satsuki:~/coderepo$ git commit -a -m 'Updated another file'
[wonderful 8c6a66b] Updated another file
 1 files changed, 1 insertions(+), 0 deletions(-)
john@satsuki:~/coderepo$ echo "More Super New Changes" >> another_file
john@satsuki:~/coderepo$ git commit -a -m 'Updated another file again'
[wonderful b91ec84] Updated another file again
 1 files changed, 1 insertions(+), 0 deletions(-)
john@satsuki:~/coderepo$ gitk --all
john@satsuki:~/coderepo$
```

Before we continue, there is a problem in the trenches.

In the trenches... "Ahh dang it" shouted Simon. "That's really not very helpful to anyone."

"What's up?" said Martha over the cubicle wall. She stood up, knocking a pile of papers over that were hanging precariously on the edge of the desk. They scattered across the floor creating a white dividing line down the middle of the office. She mutter something under her breath. John and Simon began to help tidy the papers up whilst Simon started stating his problem.

"Well, I commit two commits that used the same commit message, I really wasn't thinking. Plus the fact that they should really have been one commit."

"Just reset back and do the work again," shouted an eavesdropping Klaud unhelpfully. A round of raised eyebrows circled the paper-stackers.

> "You could use rebase." Martha was standing next to the desk now and was arranging the stack in a more suitable position on the desk.
>
> "How so?" John looked at Martha inquisitively.
>
> Martha giggled. "Rebase can do a lot more than just replaying commits you know. Some of the more simpler tasks are . . . well . . . let me show you."

Looking at our example commits, it would appear that we have made the same mistake as Simon. Our commits, **c0e2f5b** and **8c6a66b** have got the same commit message. Not very helpful at all. Let us see how `git rebase` can help us out here.

```
john@satsuki:~/coderepo$ git rebase -i HEAD~3
```

We have told Git to run the `rebase` command using the `-i` parameter. This will put Git into *interactive* mode. We have also asked Git to rebase the last three commits. The `HEAD 3` is used like before to tell us to go back three commits from the HEAD references. When we run this command, we are presented with a text file in our default text editor.

```
pick c0e2f5b Updated another file
pick 8c6a66b Updated another file
pick b91ec84 Updated another file again

# Rebase 1c3206a..b91ec84 onto 1c3206a
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#  x, exec = run command (the rest of the line) using shell
#
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
#
```

At the top of the file, you can see the last three commits, along with their commit message, prefixed by the word pick. We will get to what this really means in a few minutes, but as you can see there are a large number of options within the rebase

interactive system. If we closed the file and left it unchanged, the rebase would end without altering anything. The options are fairly self explanitory but they allow us to change many elements of a commit or a series of commits.

In our case, we can use this to reword the second commit, **8c6a66b**. We are going to modify the lines above to look like the ones below.

```
pick c0e2f5b Updated another file
r 8c6a66b Updated another file
pick b91ec84 Updated another file again
```

Notice that we replaced the beginning of the second line, original starting with `pick`, with the letter `r`. Using the list of available functions above, we can easily see that the `r` corresponds to the *reword* function, which will allow us to change the wording that we have used for that particular commit. On saving this and closing this file, Git quickly presents us with another editor window. This time the window contains a more familiar commit message setup. We will modify it to show the same as below, and then close the editor.

```
Updated another file 2nd time

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# Not currently on any branch.
# Changes to be committed:
#   (use "git reset HEAD^1 <file>..." to unstage)
#
#       modified:   another_file
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       temp_file
```

After Git completes working on the rebase, we are presented with the result of our operation.

```
john@satsuki:~/coderepo$ git rebase -i HEAD~3
[detached HEAD 7c35dde] Updated another file 2nd time
 1 files changed, 1 insertions(+), 0 deletions(-)
Successfully rebased and updated refs/heads/wonderful.
john@satsuki:~/coderepo$
```

So Git has completed the operation, and by running an abbreviated `git log` command, we can see that the fruits of our labour have resulted in the second commit having its working changed.

```
john@satsuki:~/coderepo$ git log --graph --pretty=oneline --all --abbrev-commit
 ↪--decorate -n 4
* aeb5679 (HEAD, wonderful) Updated another file again
* 7c35dde Updated another file 2nd time
* c0e2f5b Updated another file
* 1c3206a (master) Added a new file
john@satsuki:~/coderepo$
```

Note

Rebasing and the hashes

If you have been paying attention to the SHA-1 hashes as we have been moving through this last piece on rebasing, you may have noticed something interesting. Though there are commits that we have not altered at all, the commit IDs have indeed changed. If we take the example where we re-worded the second commit, it is clear to see that subsequent commits have their IDs changed. In the example, we modified commit **8c6a66b**, but we left **b91ec84** untouched. In the resulting tree, both have had their IDs changed.

The reason for this is simple, remember we are replaying commits. In a sense we are re-committing the changes that were made during that commit. As we have stated previously, Git is cryptographically secure. By this we mean that each commit relies on the commit that preceeds it. So, if we change the ID of a preceeding commit, all subsequent ones have to change also.

Let us try this rebase again. This time we will use the squash option to merge the two similar commits into one. We will run the same command as before; `git rebase -i HEAD~3`. This time we will use the `s` prefix to the line to choose *squashing* as our method. We could have used the word `squash` instead of `s`, but for the laziness in all of us, we will opt for the single letter versions for now.

```
pick c0e2f5b Updated another file
s 7c35dde Updated another file 2nd time
```

```
pick aeb5679 Updated another file again

# Rebase 1c3206a..aeb5679 onto 1c3206a
```

Now when we save and close the file, we are presented with a slightly different screen. As we are squashing several commits together into one, we need to choose a commit message. The message needs to be descriptive enough that it will accturately let a developer see what has been updated in this single commit. We will delete these lines and replace them with the comment `Updated another file with 2 edits`.

```
# This is a combination of 2 commits.
# The first commit's message is:

Updated another file

# This is the 2nd commit message:

Updated another file 2nd time
```

After completion, Git shows us that the rebase has been successful.

```
[detached HEAD 1ffe37f] Updated another file with 2 edits
 1 files changed, 2 insertions(+), 0 deletions(-)
Successfully rebased and updated refs/heads/wonderful.
john@satsuki:~/coderepo$
```

By using the `git log` message, we can see that the two commits have indeed been replaced by one.

```
john@satsuki:~/coderepo$ git log --graph --pretty=oneline --all --abbrev-commit
 ↪--decorate -n 4
* 4d91aab (HEAD, wonderful) Updated another file again
* 1ffe37f Updated another file with 2 edits
* 1c3206a (master) Added a new file
* 37950f8 Continued Development
john@satsuki:~/coderepo$
```

In the trenches... "Actually that's pretty cool," commented Simon, after Martha had shown him how to use rebase to squash and reword. She smiled, but said nothing. Martha had an uncanny knack for knowing when there was going to be another question. "So, just

how would we use git rebase for keeping a development tree up to date?"

Martha ruffled Simon's hair. She liked his youthful enthusiasm. "How about we take a crack at that tomorrow eh?"

Day 4 - "Starting to get rebased"

Using rebase with branches

Up until now we have used `git rebase` to work on our current branch, modifying a few things here and there. This is actually one of the simplest things that rebase can perform, and as hinted to in our workflow design, we can actually use `git rebase` to perform something called *Continuous Integration*.

As we stated before, one of the most interesting uses of using `git rebase` is to update your branch with changes from another, whilst keeping your development intact. The great part about it is that there is no messy merging, your commits appear at the end of the tree which makes things nice and tidy. Let us go back to our example and add some more commits to the **master** branch.

```
john@satsuki:~/coderepo$ git checkout master
Switched to branch 'master'
john@satsuki:~/coderepo$ touch cont_dev
john@satsuki:~/coderepo$ echo "New info" >> cont_dev
john@satsuki:~/coderepo$ git add cont_dev
john@satsuki:~/coderepo$ git commit -a -m 'Start new dev'
[master 1968324] Start new dev
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 cont_dev
john@satsuki:~/coderepo$ echo "A cool function" >> cont_dev
john@satsuki:~/coderepo$ git commit -a -m 'Finished new dev'
[master f8d5100] Finished new dev
 1 files changed, 1 insertions(+), 0 deletions(-)
john@satsuki:~/coderepo$ git logg -n5
* f8d5100 (HEAD, master) Finished new dev
* 1968324 Start new dev
| * 4d91aab (wonderful) Updated another file again
| * 1ffe37f Updated another file with 2 edits
|/
* 1c3206a Added a new file
john@satsuki:~/coderepo$
```

You may have noticed that we seem to have introduced a new command here called `git logg`. The command seems to do exactly the same as our `git log –graph –pretty=oneline –all –abbrev-commit –decorate` command. For more information about what we have done here, see the callout called **An alias in our midst**.

Information

An alias in our midst

Some times, you will get tired of typing the same old long string of parameters into Git. Though some people tend to frown upon using them, aliases can significantly increase your productivity, by allowing you to shortcut annoyingly long commands. You may have noticed our `git log –graph –pretty=oneline –all –abbrev-commit –decorate` command from before. It is awfully long. Let us create an alias called `git logg`.
There are multiple ways to do this, but we are going to get a little down and dirty and edit the `.git/config` file in our repository. You will find a few *stanzas* which start like this `[core]`, followed by a number of lines. If you don't already have one, use the editor to add `[alias]` at the end of the file, so that your file has a section looking like the following.

```
[alias]
        logg = log --graph --pretty=oneline --all --abbrev-commit --decorate
```

Once saved, we can use `git logg` just like any other Git command. As you can see from the example in the text, we can even append more parameters that the parent command `git log` usually accepts. So we can do things like `git logg -n5` to get only five entries.

So we have two development trees, our **master** which is what our development work is based on, and our **wonderful** branch, which is where we are performing our development work. The task now is to update the **wonderful** branch, with all of the changes that have taken place in **master**. To do this, we use the `git rebase` tool once more, but this time in a slightly different manner.

```
john@satsuki:~/coderepo$ git checkout wonderful
Switched to branch 'wonderful'
john@satsuki:~/coderepo$ git rebase master
```

```
First, rewinding head to replay your work on top of it...
Applying: Updated another file with 2 edits
Applying: Updated another file again
john@satsuki:~/coderepo$ git logg -n5
* 5167cce (HEAD, wonderful) Updated another file again
* 551086e Updated another file with 2 edits
* f8d5100 (master) Finished new dev
* 1968324 Start new dev
* 1c3206a Added a new file
john@satsuki:~/coderepo$
```

We started by checking out the **wonderful**. With the `git rebase master` command, we told Git to take all of the changes in our topic branch called **wonderful**, and replay them on top of the new **master**. We could have also used the command `git rebase master topic` which would actually have done the original `git checkout` for us.

Notice again, that our commit IDs have changed. The contents, in this case, remain identical, as we have not hit upon any conflicts during out rebase. If we had, we would have to have resolved that, and then run the `git rebase –continue`, or `git rebase –abort` to abort the rebase completely.

So, our wonderful branch is now sat on top of the new development changes and instead of having two diverging branch heads, we now have a single branch which **wonderful** extends from.

> **In the trenches...** "Martha!" Simon shouted across the office at the seemingly new Git expert. "HELP!!"
>
> Martha got up from her seat. It hadn't been fifteen minutes since she had shown Simon how to rebase branches and now he was calling her again. At first she had tried to ignore it, but the waves of SOS had plunged through the headphones and reverberated round her head one too many times. She placed the music on hold and put the headphones on the desk.
>
> "What's up Simon?" she asked, smiling at John who was chuckling to himself at his desk.
>
> "Well, I kinda rebased, but I wish I hadn't now, cos it all went wrong. How can I go back?"

It's an interesting question. How would we move back again? We have rewritten history, how could we possibly hope to go back again? We have discussed before how most things in Git are never really immediately gone, even if we delete them. Does this apply here too? Of course it does. Remember that a branch is really just defined by which commit the HEAD points to. Let us draw a few diagrams to show how the tree of commits looks before and after our rebase.

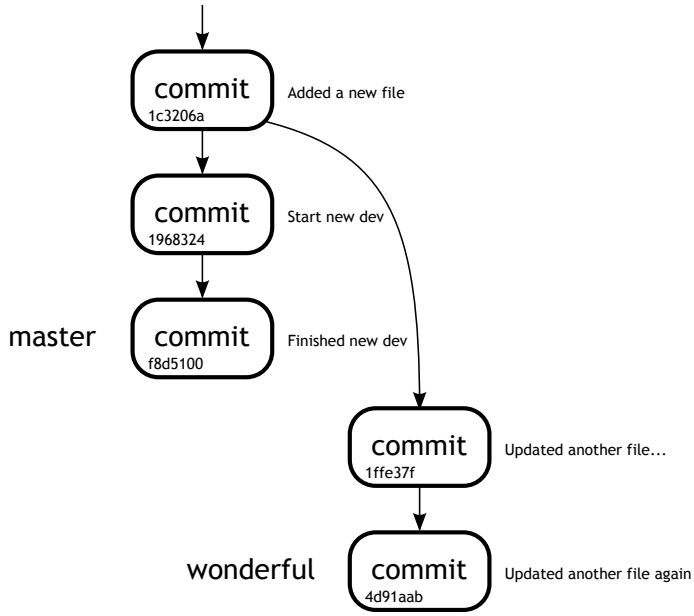


Figure 5: Our repository before the rebase

Figure 5 shows our repository before we perform the rebase. We can quite clearly see that there are two branches, diverging from the common ancestor, **1c3206a**. In Figure 6, we see the repository after the rebase has taken place. Notice that the two commits that *were* forming the **wonderful** branch are now coloured in grey.

The grey commits are no longer referenced. No branche HEADs point to them, and no other commits rely upon them. In short they have been orphaned, or are left *dangling*. A dangling commit has no references left pointing to it. Remember each commit relies on its parent. Therefore because the HEAD of **wonderful** now points to **5167cce**, and that in turn points to **551086e**. The original commit line is unchanged.

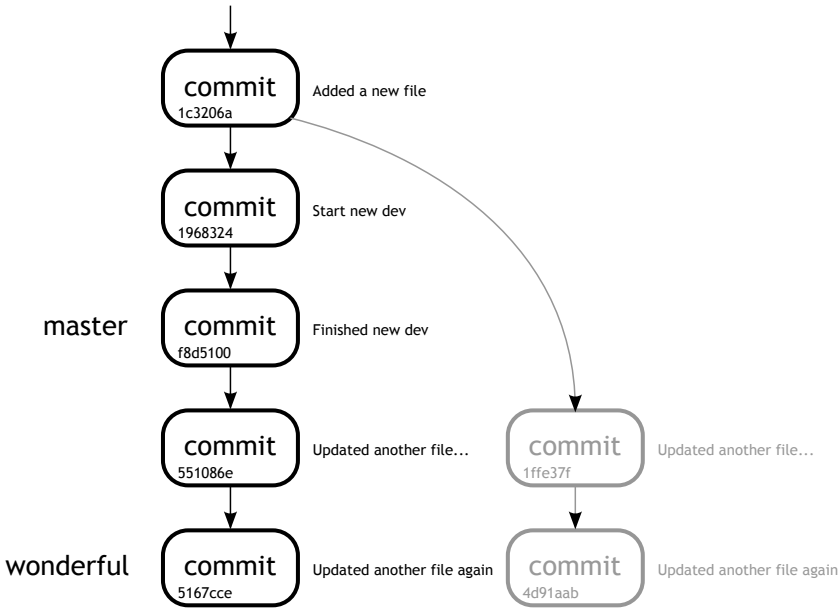


Figure 6: Our repository after the rebase

Thinking back to previous weeks, we have had a similar situation before. If we know the commit ID that we wish to return to, there really is nothing more complicated than issuing a `git reset`. The only tricky part is knowing the commit you wish to return to.

We do however have a weapon at our disposal. The `git log` tool can be used with the `-g`. This parameter forces the Git to refer to the *reflog* for entries, instead of traversing the usualy tree. If you remember, the reflog is our key to seeing what happened in the past. It holds a list of all the previous HEADs of each branch. By supplying the branch name, we can see everything that happened to the **wonderful** branch.

```
john@satsuki:~/coderepo$ git log -g wonderful -n2
commit 5167cce7864ca71420ce5dc37ec9b3f931727db3
Reflog: wonderful@{0} (John Haskins <john.haskins@tamagoyakiinc.koala>)
Reflog message: rebase finished: refs/heads/wonderful onto f8d5100142b43ffaba9bb
Author: John Haskins <john.haskins@tamagoyakiinc.koala>
Date:   Wed Jul 6 09:30:40 2011 +0100

    Updated another file again
```

```
commit 4d91aab57aaad020e62486805e25d0d6f06fdc3e
Reflog: wonderful@{1} (John Haskins <john.haskins@tamagoyakiinc.koala>)
Reflog message: rebase -i (finish): refs/heads/wonderful onto 1c3206a
Author: John Haskins <john.haskins@tamagoyakiinc.koala>
Date:   Wed Jul 6 09:30:40 2011 +0100

    Updated another file again
john@satsuki:~/coderepo$
```

As you can see, the log shows that we recently completed two rebases. The first resulted in an ID of **4d91aab** and the second resulted in **5167cce**. As it happens, we are trying to get back to the end of that first rebase, before we ran the second. So if we run the reset against that ID, we should return to our pre-rebased state.

```
john@satsuki:~/coderepo$ git reset --hard 4d91aab
HEAD is now at 4d91aab Updated another file again
john@satsuki:~/coderepo$ git logg -n5
* f8d5100 (master) Finished new dev
* 1968324 Start new dev
| * 4d91aab (HEAD, wonderful) Updated another file again
| * 1ffe37f Updated another file with 2 edits
|/
* 1c3206a Added a new file
john@satsuki:~/coderepo$
```

As you can see, even though the commit messages are identical, the history of them differs. Obviously now, it is our rebased version which is left dangling and if it continues to go unused, it will eventually be deleted. Of course it is also possible to create a branch to point to the dangling commit. In this way it would never get deleted. Let us just see how we could achieve this.

Armed with the knowledge that the commit we are looking for is **5167cce**, we can run the `git branch` command and specify a starting point.

```
john@satsuki:~/coderepo$ git branch keeprebase 5167cce
john@satsuki:~/coderepo$ git branch -v
  keeprebase 5167cce Updated another file again
  master     f8d5100 Finished new dev
* wonderful  4d91aab Updated another file again
  zaney      7cc32db Made another awesome change
john@satsuki:~/coderepo$ git logg -n7
```

```
* 5167cce (keeprebase) Updated another file again
* 551086e Updated another file with 2 edits
* f8d5100 (master) Finished new dev
* 1968324 Start new dev
| * 4d91aab (HEAD, wonderful) Updated another file again
| * 1ffe37f Updated another file with 2 edits
|/
* 1c3206a Added a new file
john@satsuki:~/coderepo$
```

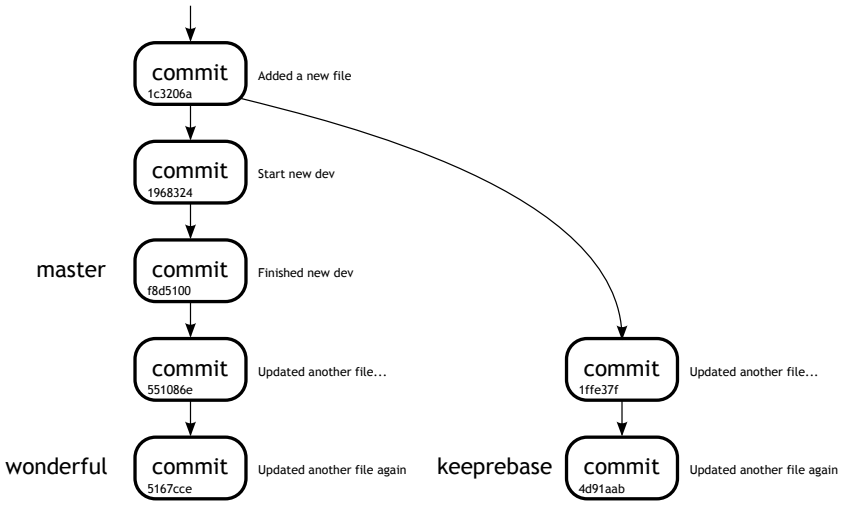


Figure 7: Our repository after the recovery

Notice our branch is now created and looking at the log, it contains the same two commits and the tree looks as we would expect. We have our `another_file` updates in both branches.

Day 5 - "I could rebase the world"

Migrating commits

We are almost at the end our our journey with the rebase tool and have one more stop before we start looking at other features of Git.

In the trenches... Rob was sitting with his head on the table. Every few seconds he would lift it up before thumping it down again. It was obvious to everyone in the office that Rob had done something wrong. In the end it was John who broke the rhythmic bass drum. He pulled up a chair next to the desk.

"Come on Rob," started John, "What's eating you?"

"Well, I have been working away, committing to a branch, but it wasn't the right one." He looked like he was almost in tears.

John frowned, "Can't you just move a new branch forward and then rewind the other one?"

"No." His voice was tired and weary. "The ancestor of the branch isn't right. I thought I had branched from master, but I'd actually branched from a dev branch. Now I can't find a way to bring my commits back. Short of cherry picking each one, but I have over fifty of them. Guess it's time to start scripting."

John smiled and shook his head, "Don't worry Rob, I have a better plan"

Cherry picking is a method of copying the contents of one commit into another and is something that we will pick up on later, but we will first look at our final use of rebase. Imagine the scenario painted above. You made a branch, have been merrily committing for hours, before realising that actually you branched from the wrong ancestor.

In Git, this isn't a problem. Things get complicated if you've been doing more advanced operations during this period, but if you have been simply adding commits, we can use rebase to migrate the tree of commits to a different ancestor.

Take the example we have been working on. Currently we have a **master** branch and a **wonderful** branch, and these differ, though they have a common ancestor. Let us say we are currently sitting on the **wonderful** branch, but we thought we were on **master** and we created a new branch called **develop**.

```
john@satsuki:~/coderepo$ git checkout -b develop
Switched to a new branch 'develop'
john@satsuki:~/coderepo$
```

Now let us commit a few files and see what we can do about fixing our problem.

```
john@satsuki:~/coderepo$ echo "new dev work" >> newfile3
john@satsuki:~/coderepo$ git commit -a
[develop eb7f633] Some new dev work
 1 files changed, 1 insertions(+), 0 deletions(-)
john@satsuki:~/coderepo$ echo "newer dev work" >> newfile2
john@satsuki:~/coderepo$ git commit -a
[develop 5e0964b] More new deving
 1 files changed, 1 insertions(+), 0 deletions(-)
john@satsuki:~/coderepo$
```

Let us use our little log alias and see the work we just did on our "**master**" branch.

```
john@satsuki:~/coderepo$ git logg -n8
* 5e0964b (HEAD, develop) More new deving
* eb7f633 Some new dev work
* 4d91aab (wonderful) Updated another file again
* 1ffe37f Updated another file with 2 edits
| * 5167cce (keeprebase) Updated another file again
| * 551086e Updated another file with 2 edits
| * f8d5100 (master) Finished new dev
| * 1968324 Start new dev
|/
john@satsuki:~/coderepo$
```

Whoops! The commits were supposed to be on a branch that stemmed from **master**, not one that stems from **wonderful**. In other words, **develop** stems from **wonderful**, when it should have stemmed from **master**. This is where `git rebase` comes to our assistance once more. With one command, we can move those commits to a branch that stems from **master**.

```
john@satsuki:~/coderepo$ git rebase --onto master wonderful develop
First, rewinding head to replay your work on top of it...
Applying: Some new dev work
Applying: More new deving
john@satsuki:~/coderepo$ git logg -n8
* aed985c (HEAD, develop) More new deving
* af3c6d7 Some new dev work
| * 5167cce (keeprebase) Updated another file again
| * 551086e Updated another file with 2 edits
|/
* f8d5100 (master) Finished new dev
* 1968324 Start new dev
| * 4d91aab (wonderful) Updated another file again
```

```
| * 1ffe37f Updated another file with 2 edits
|/
john@satsuki:~/coderepo$
```

Notice in the output above that we have used our `git rebase` tool with a new `–onto master` parameter. Basically the syntax above is stating that we would like to take all of the commits between the points **wonderful** and **develop** and place them onto **master** instead.

A little housework

To clean up our repository, we are going to delete our **keeprebase** branch.

```
john@satsuki:~/coderepo$ git branch -D keeprebase
Deleted branch keeprebase (was 5167cce).
john@satsuki:~/coderepo$
```

We have reached the end of our tour with the rebase tool. It is exceedingly powerful and is definitely one of the tools that you should understand before using. As stated before, do not forget the fact that rebasing changes history. In all cases that we have used rebase, commit IDs have been changed and therefore you must be very careful when using it.

Summary - John's Notes

Commands

- `git rebase -i HEAD 3` - Runs the rebase tool interactively for the last three commits
- `git rebase <branchA> <branchB>` - Lift all commits between the common ancestor of branchA and branchB and replay them on top of branchB
- `git branch <branch_name> <startpoint>` - Create a new branch starting from a definte start point
- `git rebase –onto master <branchC> <branchA> <branchB>` - Lift all commits between the common ancestor of branchA and branchB and replay them on top of branchC

Terminology

- **SSH** - A type of secure network protocol
- **HTTP** - The protocol that is used to serve internet pages
- **Rebase** - Used primarily for lifting commits and reapplying them to another base branch

After Hours Week 7

"Network Communicating"

The last week began by looking at some networking communication. In the After Hours section this week, we are going to look at two tools which are used for network communication.

Brewing a website - in an instant

The *Instaweb* tool allows you to spawn a web service quickly and easily, using a web daemon of your choice. The *Instaweb* tool actually uses `gitweb` which is a more permanent solution for obtaining the same functionality as *Instaweb*. If you have never played with web services before it would be worth spending a few minutes understanding a little about how web daemons work. This tool is not available on the Windows platform, but it can be used on both MacOS and Linux.

Instaweb allows us to browse our repository from the comfort of a web browser. We also have another benefit to running this from a web server. We can allow people to access our repository to look around without giving them the ability to change anything or make any commits.

Before we run `git instaweb`, we need to ensure that we have a web daemon available to us. *Instaweb* automatically creates a configuration file for the web daemon of your choice, runs the daemon on a custom network port, and loads a browser automatically pointing to the URL of the web instance you have just configured. On our example machine, we have installed lighttpd as our choice of web daemon. Once again it is advised to understand the implications of this before you do it. On Ubuntu, this can be installed by running `apt-get install lighttpd`.

```
john@satsuki:~/coderepo$ git instaweb
john@satsuki:~/coderepo$
```

The tool is invoked by running `git instaweb` and when started, you should be presented with a browser as pictured below. In our case, lighttpd has been installed, and

so Git will use that as its daemon. Firefox is also the default browser on this machine and so this is the browser that Git will choose to display the web page in. If we wanted to use an alternative browser, we could have supplied an argument with `–browser`, like `–browser chromium` for example.

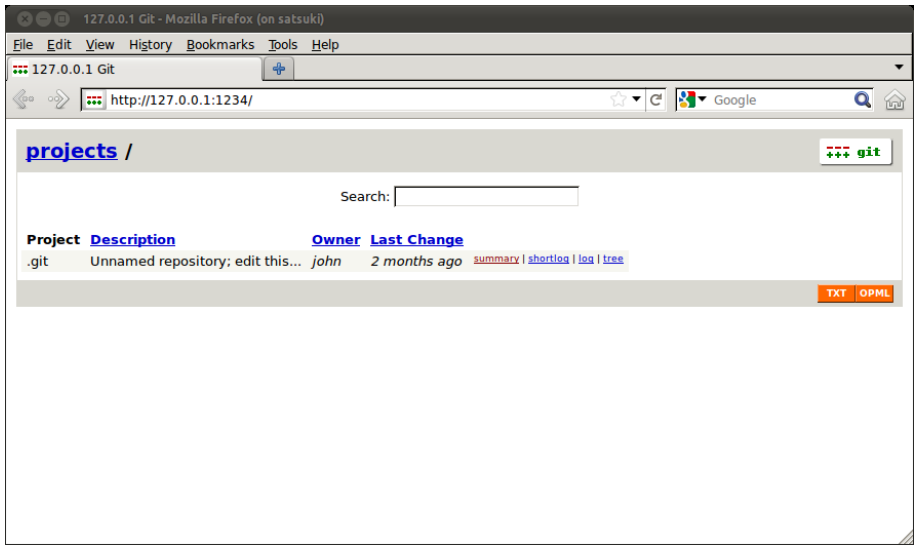


Figure 1: Instaweb's default page

The first thing to notice is that the description of the repository is unhelpful. In Figure 1, it is shown as `Unnamed repository; edit this...`. It is easy to rectify this, by editing the `.git/description` file. We are going to use the `echo` command from Linux, as we have throughout the book.

```
john@satsuki:~/coderepo$ echo "Our test repository" > .git/description
john@satsuki:~/coderepo$
```

On refreshing the page, the description will be updated. The next thing to notice is the url. The screenshot shows our url to be `http://127.0.0.1:1234/`, where `127.0.0.1` is the local address of our Git machine and `1234` is the port.

Before we start taking a look around the web interface, we should learn how to end the *Instaweb* session. If we close the web browser, it does not end the *Instaweb* process. In fact, we could load up firefox again, type in the URL `http://127.0.0.1:1234/` and

return to the home page of our Git repository. To close the instance of *Instaweb* we run the following;

```
john@satsuki:~/coderepo$ git instaweb --stop
john@satsuki:~/coderepo$
```

Now would be a good time to take a quick look at what running `git instaweb` has done to our repository. If we take a look inside the `.git` folder, we can see that there is a new folder called `gitweb`. This folder contains configuration and log files for the *Instaweb* process. The file we are most interested in is `httpd.conf`. Looking at the beginning of this file we should see something similar to the following.

```
server.document-root = "/usr/share/gitweb"
server.port = 1234
server.modules = ( "mod_setenv", "mod_cgi" )
server.indexfiles = ( "gitweb.cgi" )
server.pid-file = "/home/john/coderepo/.git/pid"
server.errorlog = "/home/john/coderepo/.git/gitweb/lighttpd/error.log"
```

Here we can see the beginning of the config file that Git has created for using with lighttpd. If we had other web daemons installed, such as apache, we could override the default of lighttpd by supplying the `–httpd apache2`. Notice the port number which as been defined as `1234`. Let us run up `git instaweb` again and see what other features the web interface offers.

Figures 2 and 3 show how the web interface looks when browsing our repository. At the top of the page we see a history of commits along with a search box which works similar to the `gitk` system which was discussed earlier. At the bottom of the page, we see our **tags** and **heads**. There are several link names which we will briefly describe now.

- **shortlog** - Gives a log of the commits, similar to that shown in Figure 2.
- **commit** - Returns a page that gives details about a specific commit. The **commit** page is shown in Figure 4.
- **commitdiff** - Shows how the chosen diff has changed since its parent. This is similar to running our `git diff HEAD 1..HEAD` command.
- **tree** - This page is a simple listing of the tree object which shows all files present in that particular commit.

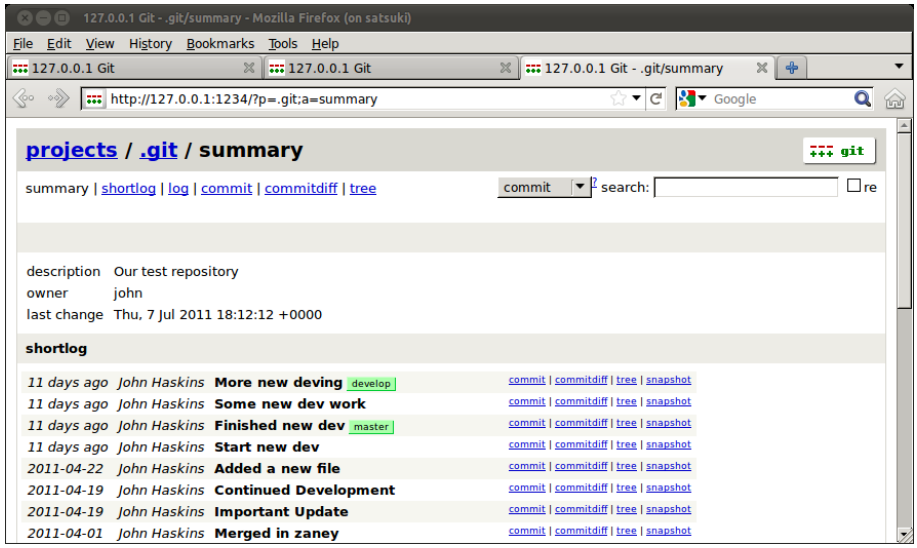


Figure 2: Instaweb's repository page (top)

- **snapshot** - Possibly one of the most useful links. Clicking on this will initiate a download of the repositories filesystem at that particular point in time.

- **log** - Gives a listing of the full log messages.

Let us take a little look around the interface. Choosing the first commit in the list on the homepage and clicking on the **commit** link moves on to the commit page. Here we can see detailed information about the commit.

The **commit**, **tree** and **parent** object hashes are displayed here for reference. The **tree** and **parent** lines are clickable links which will take us to those relevant sections. We are going to choose the **tree** link. A screenshot of the resulting page is shown in Figure 5.

Going back and clicking on the snapshot link will initiate a download of the entire filesystem, at that point in the repository's life. As shown in Figure 6.

This has been a very brief tour around the `gitweb` system. Hopefully you can see that this is a very useful tool to add to our Git suite. If you are considering setting up a permanent web based view for your Git repository, you should not use *Instaweb*. This

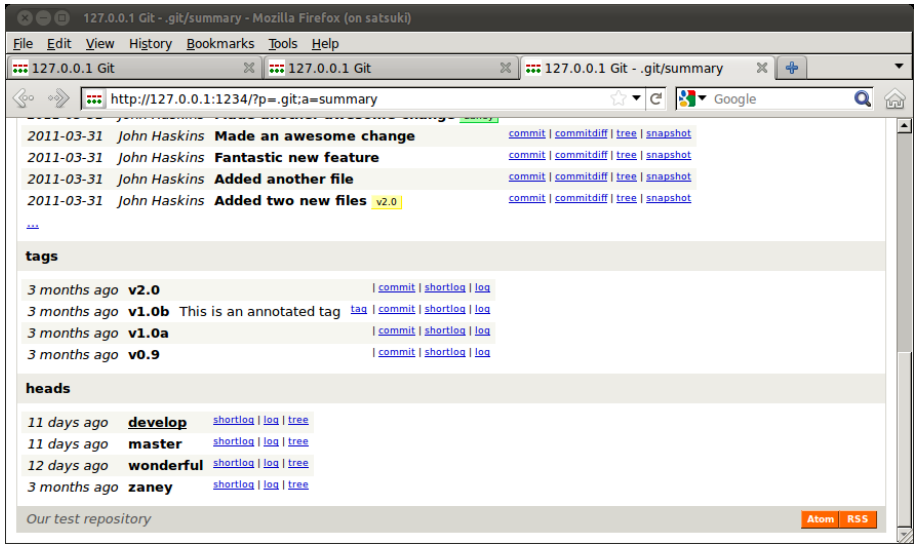


Figure 3: Instaweb’s repository page (bottom)

tool is intended for quick and easy web access to a repository. There are plenty of guides and tutorials available which explain the installation and configuration of `gitweb`.

Pushing and pulling with a daemon

The `git instaweb` daemon is very useful indeed. However, one function that it does not provide is the ability to clone or fetch from the URL. Another tool which comes with the Git package is the `git daemon` utility and it is this tool which is going to give us this functionality. We spoke earlier about the GIT protocol and it just so happens that `git daemon` uses the GIT protocol for transferring data. Let us have a look at a simple example of using `git daemon`.

```
john@satsuki:~$ git daemon --base-path=/home/john/coderepo
```

Notice how this command has not exited. This is because the process is still running, waiting for communication. We can now log into another machine on the same network and clone the repository using our `git clone` command. We will supply the host name of the machine that we configured with `git daemon`, and a directory for us to clone into. Notice the presence of `git://` instead of `ssh://` as we used in our previous network cloning example.

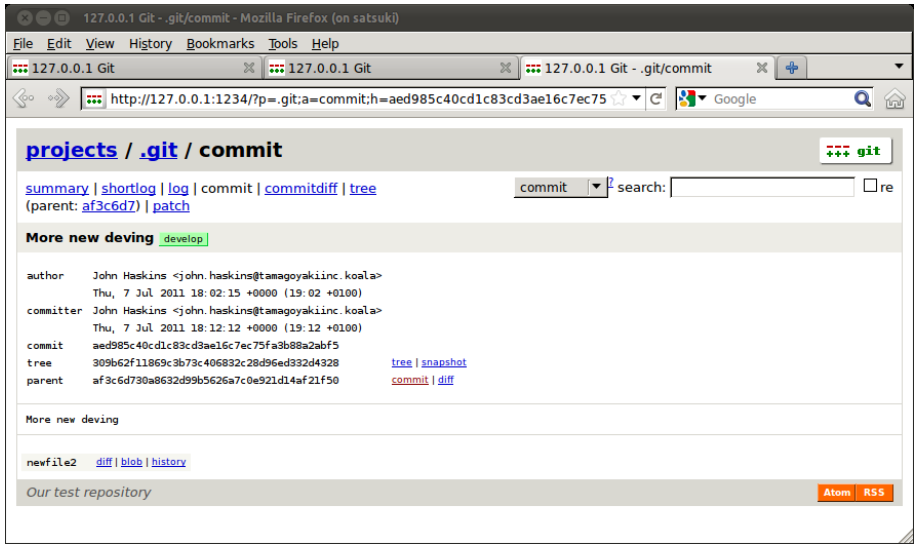


Figure 4: Instaweb's default page

```
rob@mimi:~$ cd /tmp
rob@mimi:/tmp$ mkdir source
rob@mimi:/tmp$ git clone git://satsuki/ source
Cloning into source...
fatal: The remote end hung up unexpectedly
rob@mimi:/tmp$
```

That did not go as expected and resulted in a failed clone. If we look back at the source machine we can see something interesteing has appeared.

```
john@satsuki:~$ git daemon --base-path=/home/john/coderepo
[4687] '/home/john/coderepo/.git': repository not exported.
john@satsuki:~$
```

This is probably one of the most common of all errors when dealing with `git daemon`. By default, Git tries to protect your repositories and will not allow them to be exported unless you explicitly tell it to. We have two ways of doing this. We can append the `–export-all` parameter to `git daemon`, which will allow exporting of all repositories which are under the path described in `–base-path`. The second method is to explicitly tell Git on a repository by repository, that we would like for it to be exported,

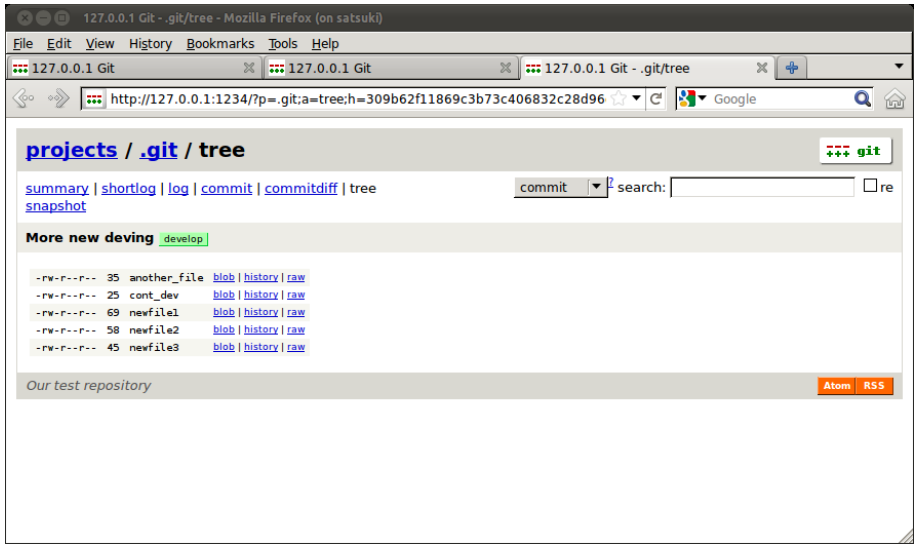


Figure 5: Instaweb's default page

or more accurately that we would like to opportunity to export it. We are going to do the latter of the two and we do this by creating a special file in the `.git` directory, called `git-daemon-export-ok`.

To add this file, we are going to need to stop the daemon by pressing the **ctrl+c** key combination. Then we ran run the commands as shown below.

```
john@satsuki:~$ touch coderepo/.git/git-daemon-export-ok
john@satsuki:~$ git daemon --base-path=/home/john/coderepo
```

Now we can go back to our second machine again and try to clone the repository as before.

```
rob@mimi:/tmp$ git clone git://satsuki/ source
Cloning into source...
remote: Counting objects: 71, done.
remote: Compressing objects: 100% (51/51), done.
remote: Total 71 (delta 16), reused 0 (delta 0)
Receiving objects: 100% (71/71), 6.47 KiB, done.
Resolving deltas: 100% (16/16), done.
rob@mimi:/tmp$
```

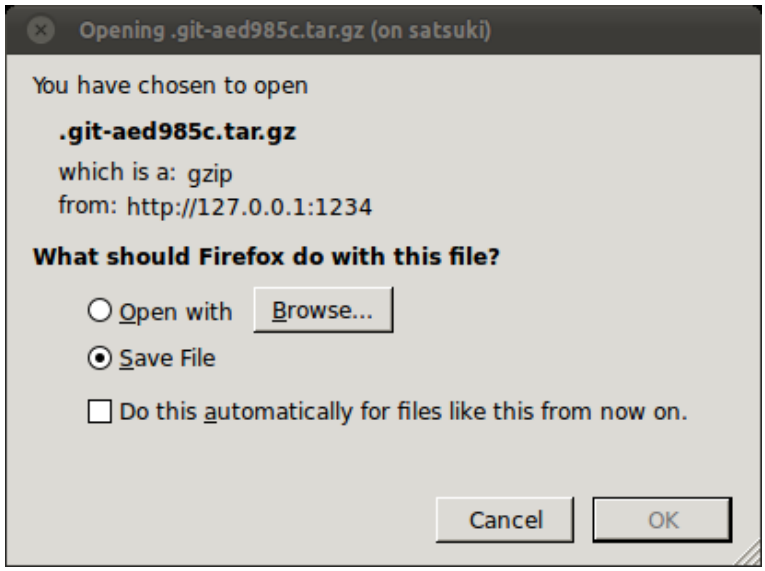


Figure 6: Instaweb's default page

Et voila! Our repository has been cloned. As we exported the root of our repository using the `–base-path` parameter, we do not need to specify to the clone command which repository we are trying to clone. If you remember we did have some other repositories in the home folder and if we had exported `/home/john/`, instead of `/home/john/coderepo`, then we could have chosen any of the repositories that lay in the home folder by appending their names to the URL. An example of this would be `git://satsuki/coderepo/` or `git://satsuki/coderepo-cl/`.

By default the `git daemon` tool only allows people to fetch objects from the repository. This is quite sensible because as you can see there is no authentication present with the GIT protocol at all. You *can* enable pushing of objects to your repository using the GIT protocol, but this is only advised within a well trusted LAN environment. Otherwise you are giving whoever else is on your network the capability to push whatever they want into your repository, which as you can understand is not a good idea.

We can run the daemon tool in the background by supplying the `–detach` parameter. However if you are unfamiliar to the Linux world, this requires some knowledge to stop

the daemon again. Running the daemon detached will result in your shell returning to the prompt, seemingly not have executed anything. In fact the process has been moved to the background and no longer supports shell interaction. The benefit of this is that you do not have to keep a shell window open to run it in. The downside is that you can no longer end the process with the **ctrl+c** key combination.

You can hopefully see that by using the `git daemon` tool, it is possible for us to allow other people on our network to have access to our repositories. Whilst the `git daemon` tool is very useful, it does have its disadvantages as stated, primarily in areas of version control. However, seeing how efficient the GIT protocol is compared to its other counterparts, it often makes a fantastic device for making a repoitory available to pull from.

There are much more complicated configurations that can be performed with the daemon tool, but these are out of the scope of this chapter. The manual page for `git daemon` has some examples to get you started, and even goes into the areas of virtual hosting to allow multiple Git *sites* to exist on one server.

Week 8

Day 1 - "Give a man a patch"

Collaborating with outsiders

We have spoken at great length now about rebasing and have seen that it is a very very powerful tool. It can form part of your workflow in your development cycle. However, always heed that warning that should send alarm bells ringing in the back of your mind about rebasing. Rebasing changes the past. Rebasing changes history. As such, it should be used a) with caution, and b) only by people who understand exactly what they are doing.

We are going to leave rebasing for a while now, take a quick look at a feature you really should know about and then focus on some of the more advanced features of Git. The following situation occurs fairly regularly for some people.

> **In the trenches...** John was stroking his chin and looking pensively out of the window when Simon approached his desk. The manager hadn't seen him yet and Simon instinctively swayed a little back and forth, try to make himself known in as subtle a way as possible. Klaus, who was watching from the corner of his eye took a more direct approach. He took the out of date org chart down from the office divider, screwed it up into a ball and launched it at John's head. It struck the manager squarely in the jaw causing him to almost tip from his awkwardly balanced chair.
>
> John noticed Simon standing there and looked a little surprised. He then noticed Klaus and in an instant understood the chain of events that had just taken place. "Sorry Simon," started John, "I've been trying to figure out a problem all morning."
>
> "It's no problem." Simon pulled up a chair and sat down. "I was wondering if you had a few minutes to discuss Luigi?"

* * *

> "Well as Luigi is a contractor, he's not going to get access to our repository here to perform commits directly. And he doesn't have the capability, nor do I really want him, making our code available on the internet. But he does have a clone of our repository from last week." John understood the problem.
>
> "Right!"
>
> "Have you heard of patching in Git?" asked John.
>
> Simon looked at his shoes, "Can't say I have John, sorry."
>
> John smiled, "No worrys. What we can do is get Luigi to generate a patch of his changes. We can then take that patch and apply it to our codebase. Luigi can then just reset his clone when he comes into the office." Simon nodded as John continued, "Go and ask Martha about it. I think she's pretty hot on these types of things."
>
> Klaus giggled, "Think she's hot eh John?"
>
> The paper was returned.

It is a good question though. Sometimes you may have a repository that is either publically available, or made available to a group of people. You do not necessarily want to set up a remote tracking branch and pull changes in from every single contributor. There are two primary reasons for this;

1. There are a large number of people submitting small changes to the code.
2. There are difficulties in communicating between the two repositories either for security or general reasons.

In these cases we need another way to apply changes from one branch into another. Many larger open source projects allow contributors to email in patches. Git does have some rather advanced ways of dealing with these types of scenarios. We are going to scratch the surface and look at using three commands `git apply`, `git format-patch` and `git am`.

First, let us find a way of generating a patch. Let us take the example we have currently in our repository. Imagine that the **develop** branch exists on another computer

in a clone of our repository. At some point in time, someone cloned our repository. They have the HEAD of our repository at the same point as we do, but they have continued to do some development in a new branch called **develop**. Now they are ready to give those changes back.

Firstly we are going to look at using the `git diff` tool to generate a patch file which we can apply.

```
john@satsuki:~/coderepo$ git checkout develop
Already on 'develop'
john@satsuki:~/coderepo$ git diff master develop
diff --git a/newfile2 b/newfile2
index 3545c1d..ff59f55 100644
--- a/newfile2
+++ b/newfile2
@@ -1,2 +1,3 @@
 Another new file
 and a new awesome feature
+newer dev work
diff --git a/newfile3 b/newfile3
index 638113c..2e00739 100644
--- a/newfile3
+++ b/newfile3
@@ -1 +1,2 @@
 These changes are in the origin
+new dev work
john@satsuki:~/coderepo$
```

That will generate us a diff from the `develop` to the `master` branch. We could copy and paste that information from the terminal window into a file, but Linux offers us an easier way of doing this.

```
john@satsuki:~/coderepo$ git diff master develop > our_patch.diff
john@satsuki:~/coderepo$ cat our_patch.diff
diff --git a/newfile2 b/newfile2
index 3545c1d..ff59f55 100644
--- a/newfile2
+++ b/newfile2
@@ -1,2 +1,3 @@
 Another new file
 and a new awesome feature
+newer dev work
diff --git a/newfile3 b/newfile3
```

```
index 638113c..2e00739 100644
--- a/newfile3
+++ b/newfile3
@@ -1 +1,2 @@
 These changes are in the origin
+new dev work
john@satsuki:~/coderepo$
```

So we can see that the file itself has the information we are looking for. Now we can use the `git apply` tool to actually modify the files in **master** and bring in the changes that have happened in **develop**.

```
john@satsuki:~/coderepo$ git checkout master
Switched to branch 'master'
john@satsuki:~/coderepo$ git apply our_patch.diff
john@satsuki:~/coderepo$ git diff
diff --git a/newfile2 b/newfile2
index 3545c1d..ff59f55 100644
--- a/newfile2
+++ b/newfile2
@@ -1,2 +1,3 @@
 Another new file
 and a new awesome feature
+newer dev work
diff --git a/newfile3 b/newfile3
index 638113c..2e00739 100644
--- a/newfile3
+++ b/newfile3
@@ -1 +1,2 @@
 These changes are in the origin
+new dev work
john@satsuki:~/coderepo$ git commit -a -m 'Updated with patch'
[master 81eee9f] Updated with patch
 2 files changed, 2 insertions(+), 0 deletions(-)
john@satsuki:~/coderepo$ git diff develop master
john@satsuki:~/coderepo$
```

Of course doing things this way means that we still have to commit our changes. Plus, all of the changes that we have made in the patch are committed in one block. Sure, we could split that using some of the techniques in the After Hours sections, but then we may not always be aware of what should be split where.

Can we have some order please?

There is another tool that can come to our rescue here. It is primarily used for working with mailboxes, but it also has some other uses which we will describe here. Would it not be nice to be able to have each commit that we want to use as a patch in a separate patch file. The file `our_patch.diff` above contained two commits worth of data. We have access to another tool in our fight against disparate systems. This is the `git format-patch` command.

First we will undo the changes we made previously by resetting the **master** branch back to its older position and deleting the `our_patch.diff` file.

```
john@satsuki:~/coderepo$ git reflog show master -n 4
81eee9f master@{0}: commit: Updated with patch
f8d5100 master@{1}: commit: Finished new dev
1968324 master@{2}: commit: Start new dev
john@satsuki:~/coderepo$ git reset --hard f8d5100
HEAD is now at f8d5100 Finished new dev
john@satsuki:~/coderepo$ rm our_patch.diff
john@satsuki:~/coderepo$
```

We used the `git reflog` command to show what the last four **master** HEAD values were. Then we reset the branch back to the point before the `git apply`. Finally we deleted the patch. Now let us see how to use the `git format-patch` command to create multiple patch files.

```
john@satsuki:~/coderepo$ git format-patch master..develop
0001-Some-new-dev-work.patch
0002-More-new-deving.patch
john@satsuki:~/coderepo$
```

It would appear that the result of this command is that two files have been generated. Let us confirm our suspicions and `cat` the contents of them to ensure that they contain the data we expect.

```
john@satsuki:~/coderepo$ cat 0001-Some-new-dev-work.patch
From af3c6d730a8632d99b5626a7c0e921d14af21f50 Mon Sep 17 00:00:00 2001
From: John Haskins <john.haskins@tamagoyakiinc.koala>
Date: Thu, 7 Jul 2011 19:01:59 +0100
Subject: [PATCH 1/2] Some new dev work

---
```

```
 newfile3 |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)

diff --git a/newfile3 b/newfile3
index 638113c..2e00739 100644
--- a/newfile3
+++ b/newfile3
@@ -1 +1,2 @@
 These changes are in the origin
+new dev work
--
1.7.4.1

john@satsuki:~/coderepo$
```

Woah! Hold on a minute. This does not seem to be a normal diff file at all. In fact, that is absolutely right. This is a patch file and the two are not the same. The patch file contains much more information than the simple diff file. For a start we get information about which commit this patch came from, who created it, when and a subject. In fact this looks almost like an email. In fact it is created to resemble a format that would be easily emailable.

We have specified a range of commits to the `git format-patch` command with the parameter `master..develop`. The format of that parameter should be familar from earlier chapters when we utilised it for commands like `git diff` and `git log`. We could now take those files, email them to someone else and they could apply them. Let us learn one more tool, and see how we would apply those patches when they had been received at the other end.

```
john@satsuki:~/coderepo$ git am 0001-Some-new-dev-work.patch
Applying: Some new dev work
john@satsuki:~/coderepo$ git am 0002-More-new-deving.patch
Applying: More new deving
john@satsuki:~/coderepo$ git diff master..develop
john@satsuki:~/coderepo$
```

Of course this is just a simple example case and in actual usage there may be cases where conflicts and other complications occur. Looking at a log output, we can see that the original dates and times of the commits are maintained and are not updated. We can ignore this if we wish and use the `–ignore-date` parameter to use the current date when committing the patch to the repository.

```
john@satsuki:~/coderepo$ git log -n4
commit 30900fe1b7e72411dabab8b02070f36e2431f704
Author: John Haskins <john.haskins@tamagoyakiinc.koala>
Date:   Thu Jul 7 19:02:15 2011 +0100

    More new deving

commit a8281fb589e36389cc8cb0da7ebee225b4d1adfc
Author: John Haskins <john.haskins@tamagoyakiinc.koala>
Date:   Thu Jul 7 19:01:59 2011 +0100

    Some new dev work

commit f8d5100142b43ffaba9bbd539ba4fd92af79bf0e
Author: John Haskins <john.haskins@tamagoyakiinc.koala>
Date:   Thu Jul 7 08:39:29 2011 +0100

    Finished new dev

commit 1968324ce2899883fca76bc25496bcf2b15e7011
Author: John Haskins <john.haskins@tamagoyakiinc.koala>
Date:   Thu Jul 7 08:39:07 2011 +0100

    Start new dev
john@satsuki:~/coderepo$
```

Interestingly if we use our alias for the log command we see something maybe a little unexpected.

```
john@satsuki:~/coderepo$ git logg -n6
* 30900fe (HEAD, master) More new deving
* a8281fb Some new dev work
| * aed985c (develop) More new deving
| * af3c6d7 Some new dev work
|/
* f8d5100 Finished new dev
* 1968324 Start new dev
john@satsuki:~/coderepo$
```

Notice that the branch **master** has not been simply fast forwarded to that of commit of **develop**. This is because we have not performed a merge, but in a sense we have manually made that changes to the files and created separate commits for them. In this way the commits **30900fe** and **a8281fb** are not the same as their **develop** counterparts.

If you intend to use this workflow, it is worth spending some time reading the man page for `git am` and `git format-patch` as both of them hold valuable information regarding the customisation and handling of patches and emails. Tamagoyaki Inc. are not going to use this workflow often and so just applying a few patches here and there from contractors using the methods is prefectly acceptable to them. If you were a large open source establishment, or any company that accepts a large number of patches, you may want to take a closer look at how to work these. Now it is time to move on to some more advanced topics within Git, but first a little cleanup.

```
john@satsuki:~/coderepo$ rm 0001-Some-new-dev-work.patch
john@satsuki:~/coderepo$ rm 0002-More-new-deving.patch
john@satsuki:~/coderepo$
```

Day 2 - "Looking for problems"

A problem shared is a problem bisected

During most software development, bugs are introduced. Sometimes these bugs are fixed immediately and sometimes they sit there in the code festering away for months on end until someone tests a specific case. Of course it is always best to have test suites and run them regularly against the code base, but on occasions either the test case itself has a bug, or the test case is written in such a way that a particular bug would never present itself. Tamagoyaki Inc. have a fairly rigorous testing procedure. Unfortunately it would seem that one particularly nasty bug has slipped through the cracks. Cue a difficult discussion.

> **In the trenches...** "But what I don't understand John, is that you now know what happened at every step in the process. How can something like this break and you not know about it?" As always Markus was getting snappy and as always John was having to bite his lip.
>
> "It's not a question about not knowing about it," began John, "The difficulty is knowing what change introduced the problem. We are on such a rapid development schedule that too many things are changing at once."

"Well, this is one of the reasons you guys have spent the last two months getting this version control system running." Markus got up and opened the door. "I suggest you fix it."

* * *

"Markus is blaming us for introducing a bug?" Rob was pretty shocked as he and Simon chatted at the water cooler.

"More like, Markus believed that a version control system was going to solve all of our problems," replied Simon.

Rob squinted his face up as a car drove into the buildings car park, showering the room with reflected sunlight. He shielded his eyes. "You know I heard there was a tool in Git for helping to find bugs. Think I may take a look over lunch, you know, be a real hero."

They both chuckled.

It is true that Git does have a very powerful tool for helping to detect revisions that introduced bugs into the system. The tool is called `git bisect` and it is used to successively checkout revisions from the repository, check them to see if the bug is present and then use that information to determine the revision that is most likely to have introduced the bug.

Let us assume that the bug in our repository is a fairly simple one. For some bizarre reason our codebase is broken unless the word `Addition` is present in one of the files. If we run a simple Linux `grep` command across the files, we can see that the word we are after is not there. However, if we go back to tag **v1.0a** and run the same command, we can see that the word is there.

```
john@satsuki:~/coderepo$ grep "Addition" *
john@satsuki:~/coderepo$ git checkout v1.0a
Note: checking out 'v1.0a'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
```

```
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b new_branch_name

HEAD is now at a022d4d... Messed with a few files
john@satsuki:~/coderepo$ grep "Addition" *
my_third_committed_file:Addition to the line
john@satsuki:~/coderepo$
```

Notice the warning about checking out a non-branch. This is perfectly normal and should not worry you but please be aware that it is obviously best to have a clean working directory before starting any type of `bisect` commands. We can see that the string we are looking for is present in the file called `my_third_committed_file`. As our repository is very small, it would not take us long to go through and check each revision to see when this string was deleted. In fact we have other tools available to search for the adding and removal of strings. For now let us assume that the *bug* is more complicated than this.

Let us go back to the facts. We know that the repository was **good** at tag **v1.0a**. We also know that the repository is bad in its current state. By feeding these details to the `git bisect` command, we can begin a search for the bug. What will happen at each stage is that Git will checkout a revision that it wants us to test and we tell Git if we think that revision is good or bad.

```
john@satsuki:~/coderepo$ git bisect start
Already on 'master'
john@satsuki:~/coderepo$ git bisect good v1.0a
john@satsuki:~/coderepo$ git bisect bad master
Bisecting: 9 revisions left to test after this (roughly 3 steps)
[ed2301ba223a63a5a930b536a043444e019460a7] Removed third file
john@satsuki:~/coderepo$
```

So we invoke the tool by running `git bisect start`. After this we tell Git the things that we know. It was good at **v1.0a**, `git bisect good v1.0a`. However, it was bad at **master**, our current revision, `git bisect bad master`. After this, Git checks out revision **ed2301b** and tells us that there are `9` revisions between the two points and that it should take only `3` more steps to complete. Now we run our test again.

```
john@satsuki:~/coderepo$ grep "Addition" *
john@satsuki:~/coderepo$
```

As we have no result here, this would be classed as a bad revision and so we mark it as so.

```
john@satsuki:~/coderepo$ git bisect bad
Bisecting: 3 revisions left to test after this (roughly 2 steps)
[9710177657ae00665ca8f8027b17314346a5b1c4] Added another file
john@satsuki:~/coderepo$
```

Git now presents us with a new choice and you can see that the number of revisions left to check has decreased dramatically from `9` to `3`. We continue marking our revisions as good and bad.

```
john@satsuki:~/coderepo$ grep "Addition" *
my_third_committed_file:Addition to the line
john@satsuki:~/coderepo$ git bisect good
Bisecting: 2 revisions left to test after this (roughly 1 step)
[cfbecabb031696a217b77b0e1285f2d5fc2ea2a3] Fantastic new feature
john@satsuki:~/coderepo$ grep "Addition" *
my_third_committed_file:Addition to the line
john@satsuki:~/coderepo$ git bisect good
Bisecting: 0 revisions left to test after this (roughly 1 step)
[b119573f4508514c55e1c4e3bebec0ab3667d071] Merge branch 'wonderful'
john@satsuki:~/coderepo$ grep "Addition" *
my_third_committed_file:Addition to the line
john@satsuki:~/coderepo$ git bisect good
ed2301ba223a63a5a930b536a043444e019460a7 is the first bad commit
commit ed2301ba223a63a5a930b536a043444e019460a7
Author: John Haskins <john.haskins@tamagoyakiinc.koala>
Date:   Fri Apr 1 07:37:34 2011 +0100

    Removed third file

:100644 000000 68365cc0e5909dc366d31febf5ba94a3268751c6
 ↪0000000000000000000000000000000000000000 D  my_third_committed_file
john@satsuki:~/coderepo$
```

Oh! Something different. Git has actually finished the bisect and has suggested to us that this commit was responsible for introducing the bug in our code. If we look at the comment it was in this revision that we removed a particular file. This file was the one that contained our special `Addition` string. Git was right! We screwed up then. At this point we can go back to our **master** branch and decide what to do about it.

```
john@satsuki:~/coderepo$ git branch -v
* (no branch) b119573 Merge branch 'wonderful'
  develop     aed985c More new deving
  master      30900fe More new deving
  wonderful   4d91aab Updated another file again
  zaney       7cc32db Made another awesome change
john@satsuki:~/coderepo$ git checkout master
Previous HEAD position was b119573... Merge branch 'wonderful'
Switched to branch 'master'
john@satsuki:~/coderepo$
```

Notice that at the end of the bisect, Git does not return us to the master branch. We are left in the last tested checked out revision.

Automating the process

So bisecting is a very powerful way of quickly and efficiently finding the point at which bugs were introduced or regression testing. Git was spot on when it suggested that that revision was the one responsible for the mistake. Sometimes you may not be able to test a revision that Git checks out for you for other reasons. In this case you can always run `git bisect skip` to skip that revision. It is all very well being able to run this at each revision Git asks us to but to be honest, if you have 30-40 steps to test and you have to compile code to see if the bug is present it can get a little bit boring.

Git has a way of allowing us to test automatically. The example we are going to use is obviously based on a Linux environment, but if you are a developer on a Windows platform, you should have no trouble understanding what is happening here. We are going to create a small shell script that will automatically run our grep test. If the string is found we will exit with a status code of `0`, indicating that it was successful and if the string is not found, we will exit with a status code of `123`, indicating that the test was unsuccessful.

Git will use these status codes and interpret a code of `0` as **good** and a code of `123` as **bad**. Below is a copy of our shell script which we have saved as `test.sh` and have given relevant permissions to allow it to run etc. Notice we have had to exclude our `test.sh` file from the test, else the string `Addition` would have been found there which would have returned true every time.

```
john@satsuki:~/coderepo$ cat test.sh
#!/bin/bash
```

```
if grep -q Addition * --exclude=test.sh
  then echo "Good"
  exit 0
else
  echo "Bad"
  exit 123
fi
john@satsuki:~/coderepo$
```

Now we invoke `git bisect` slightly differently by asking it to start and itterate over the revisions `master` to `v1.0a`. At this point we have not told Git anything about which revisions are good or bad.

```
john@satsuki:~/coderepo$ git bisect start master v1.0a
Bisecting: 9 revisions left to test after this (roughly 3 steps)
[ed2301ba223a63a5a930b536a043444e019460a7] Removed third file
john@satsuki:~/coderepo$
```

Now we ask Git to continue testing, but to run our script at each iteration to determine the success or failure of each checked out revision.

```
john@satsuki:~/coderepo$ git bisect run sh ./test.sh
running sh ./test.sh
Bad
Bisecting: 3 revisions left to test after this (roughly 2 steps)
[9710177657ae00665ca8f8027b17314346a5b1c4] Added another file
running sh ./test.sh
Good
Bisecting: 2 revisions left to test after this (roughly 1 step)
[cfbecabb031696a217b77b0e1285f2d5fc2ea2a3] Fantastic new feature
running sh ./test.sh
Good
Bisecting: 0 revisions left to test after this (roughly 1 step)
[b119573f4508514c55e1c4e3bebec0ab3667d071] Merge branch 'wonderful'
running sh ./test.sh
Good
ed2301ba223a63a5a930b536a043444e019460a7 is the first bad commit
commit ed2301ba223a63a5a930b536a043444e019460a7
Author: John Haskins <john.haskins@tamagoyakiinc.koala>
Date:   Fri Apr 1 07:37:34 2011 +0100

    Removed third file
```

```
:100644 000000 68365cc0e5909dc366d31febf5ba94a3268751c6
 ↪0000000000000000000000000000000000000000 D  my_third_committed_file
bisect run success
john@satsuki:~/coderepo$
```

The parameters after the `git bisect run` tell Git which command we wish to run at each stage. In our case it is `sh ./test.sh`. You can see Git invoking our `test.sh` script in each case, and the result of our script, either `Good` or `Bad` depending on which was echoed from the result of the grep test. Git has arrived at exactly the same result, but we have had to do nothing other than write a small script. For larger tests, this would have saved us a large amount of work.

In the trenches... "Simon could I have a word?" It was Rob and he wasn't looking happy.

Simon turned to him and grinned, "Sure buddy what's up?" His face dropped when he saw Rob's expression.

"I think we'd better go grab the meeting room."

Simon looked confused.

"I used the bisect tool to find the bug. But you're not gonna like what I found."

* * *

"Simon how could you have done that?" John was asking the questions and they were coming thick and fast. "I mean changing the API key for the web sevice whilst developing was not a great idea to start with, but committing that to the repository was rediculous." Simon sat there with his head in his hands. "You know how secret that API key is right?" Simon nodded. "Simon we were supposed to be releasing this repository publically in a few weeks but now that the API is in there we can't do that."

"John I'm really sorry OK." Simon was kicking himself for his mistake.

John sighed, he had been really angry to begin with but now he was calming down, "It's OK Simon, we're all getting used to the repository and version control. Do you think we can fix it?"

Day 3 - "Filtered repos"

Looking at a repo with rose tinted glasses

It does happen. Sometimes when people are under pressure, mistakes are made, just like earlier when we accidently deleted our branch from the repository. This time the mistake is a little more crucial but again it does happen and it sometimes goes a long time before it is noticed.

> **In the trenches...** "So it's been in there for how long?" asked John.
>
> Simon looked pretty sheepish as he mouthed the words, "Weeks."
>
> John bit on the end of the pen in his hand. His teeth chewed into the plastic, deforming the blue lid. "Did you find a way of sorting it out yet?"
>
> "I think so. It's not ideal, but I think so."

It would be useful if we could rewrite the history to remove the information that we wanted to. As it turns out there is a tool that we can use to do this. The `git filter-branch` allows us to run operations on a branch to rewrite its history. Hopefully you are already remembering about the care we need to take when rewriting history, but sometimes there is a real need to perform some of these operations. Let us take a look at a few examples to see how this can work. We are going to assume that our file `newfile1` contains some very sensitive information and we wish to remove it completely from the repository.

```
john@satsuki:~/coderepo$ git checkout master
Already on 'master'
john@satsuki:~/coderepo$ ls -la
total 40
drwxr-xr-x  3 john john 4096 2011-07-27 19:54 .
drwxr-xr-x 32 john john 4096 2011-07-27 19:00 ..
-rw-r--r--  1 john john   35 2011-07-22 07:15 another_file
-rw-r--r--  1 john john   25 2011-07-22 07:15 cont_dev
drwxrwxr-x  9 john john 4096 2011-07-27 19:54 .git
-rw-r--r--  1 john john   69 2011-07-27 19:54 newfile1
-rw-r--r--  1 john john   58 2011-07-22 07:15 newfile2
-rw-r--r--  1 john john   45 2011-07-22 07:15 newfile3
```

```
-rw-r--r--  1 john john    8 2011-03-31 22:15 temp_file
-rwxrwxr-x  1 john john  114 2011-07-21 21:17 test.sh
john@satsuki:~/coderepo$
```

As you can see, currently we have `newfile1` in our tree. We can also use the `git log` tool to see each commit which has touched that path.

```
john@satsuki:~/coderepo$ git log --pretty=oneline master -- newfile1
9cb2af2a00fd2253060e6bf8cc6c377b3d55ecea Important Update
d50ffb2fa536d869f2c4e89e8d6a48e0a29c5cc1 Merged in zaney
a27d49ef11d9f0e66edbad8f6c7806510ad5b2be Made an awesome change
cfbecabb031696a217b77b0e1285f2d5fc2ea2a3 Fantastic new feature
55fb69f4ad26fdb6b90ac6f43431be40779962dd Added two new files
john@satsuki:~/coderepo$
```

So there were five commits in the past which have touched that path. In our example we require the removal of this path from the entire history of the repository. As this is a destructive operation that works on the current branch, meaning it will rewrite our branch HEAD, we are first going to switch into a new branch.

```
john@satsuki:~/coderepo$ git checkout -b remove_file
Switched to a new branch 'remove_file'
john@satsuki:~/coderepo$
```

Now we need to run the `git filter-branch` tool.

```
john@satsuki:~/coderepo$ git filter-branch --index-filter 'git rm --cached
 ↪--ignore-unmatch newfile1' HEAD
Rewrite 55fb69f4ad26fdb6b90ac6f43431be40779962dd (6/21)rm 'newfile1'
Rewrite 9710177657ae00665ca8f8027b17314346a5b1c4 (7/21)rm 'newfile1'
Rewrite 4ac92012609cf8ed2480aa5d7f807caf2545fe2f (8/21)rm 'newfile1'
Rewrite cfbecabb031696a217b77b0e1285f2d5fc2ea2a3 (9/21)rm 'newfile1'
Rewrite b119573f4508514c55e1c4e3bebec0ab3667d071 (10/21)rm 'newfile1'
Rewrite ed2301ba223a63a5a930b536a043444e019460a7 (11/21)rm 'newfile1'
Rewrite a27d49ef11d9f0e66edbad8f6c7806510ad5b2be (12/21)rm 'newfile1'
Rewrite 7cc32dbf121f2afa8c40337db54bafb26de5b9c4 (13/21)rm 'newfile1'
Rewrite d50ffb2fa536d869f2c4e89e8d6a48e0a29c5cc1 (14/21)rm 'newfile1'
Rewrite 9cb2af2a00fd2253060e6bf8cc6c377b3d55ecea (15/21)rm 'newfile1'
Rewrite 37950f861a3cc0868c65ee9571fc6c491aa689ea (16/21)rm 'newfile1'
Rewrite 1c3206aac0fb012bfdaf5ff00e320b565bb89e7d (17/21)rm 'newfile1'
Rewrite 1968324ce2899883fca76bc25496bcf2b15e7011 (18/21)rm 'newfile1'
Rewrite f8d5100142b43ffaba9bbd539ba4fd92af79bf0e (19/21)rm 'newfile1'
Rewrite a8281fb589e36389cc8cb0da7ebee225b4d1adfc (20/21)rm 'newfile1'
```

```
Rewrite 30900fe1b7e72411dabab8b02070f36e2431f704 (21/21)rm 'newfile1'

Ref 'refs/heads/remove_file' was rewritten
john@satsuki:~/coderepo$
```

We have passed a few parameters to `git filter-branch` and we should take a few seconds to discuss this as the syntax may seem a little strange. Firstly we are invoking the `git filter-branch` tool, that should not be anything new at all. Next, we are passing three parameters to it. The first of these is the type of filter we wish to use. In our case we have used the `–index-filter` option. More information is available in the Git manual, but in a nutshell we have asked Git to work on the *index* at each commit stage. There is another similar option called `–tree-filter`, however care must be taken to distinguish between the two as using `–tree-filter` checks out the commit at each point in history. This may not sound like a problem, until you discover that as well as checking each revision out, it also automatically adds any untracked files in the working tree and commits them.

The next parameter is the actual command that we wish Git to perform on each revision. In this case we want to `git rm –cached –ignore-unmatch newfile1` each time. We have enclosed the command we wish to run inside quotes so that Git does not get confused with which parameters are part of the `filter-branch` and which are part of the `rm`. Using these options we have asked Git to work on just the *index* and not to complain if it can not find the file to delete.

Lastly we list the commit range we wish to filter. In this case we have specified the target revision as `HEAD`. Git will interpret this as meaning everything up to the `HEAD` revision. As such Git will be rewriting the entire history of the branch.

Now if we list the files in the directory, we can see something important has happened. The file that we wanted removed, has gone and `newfile1` is no more.

```
john@satsuki:~/coderepo$ ls -la
total 36
drwxr-xr-x  3 john john 4096 2011-07-27 19:53 .
drwxr-xr-x 32 john john 4096 2011-07-27 19:00 ..
-rw-r--r--  1 john john   35 2011-07-22 07:15 another_file
-rw-r--r--  1 john john   25 2011-07-22 07:15 cont_dev
drwxrwxr-x  9 john john 4096 2011-07-27 19:53 .git
-rw-r--r--  1 john john   58 2011-07-22 07:15 newfile2
-rw-r--r--  1 john john   45 2011-07-22 07:15 newfile3
-rw-r--r--  1 john john    8 2011-03-31 22:15 temp_file
```

```
-rwxrwxr-x  1 john john  114 2011-07-21 21:17 test.sh
john@satsuki:~/coderepo$
```

Re-running the log command we ran earlier against our new branch confirms our operation. However checking out the **master** also confirms that the file is still present elsewhere.

```
john@satsuki:~/coderepo$ git log --pretty=oneline remove_file -- newfile1
john@satsuki:~/coderepo$ git checkout master
Switched to branch 'master'
john@satsuki:~/coderepo$ ls -la
total 40
drwxr-xr-x  3 john john 4096 2011-07-27 19:54 .
drwxr-xr-x 32 john john 4096 2011-07-27 19:00 ..
-rw-r--r--  1 john john   35 2011-07-22 07:15 another_file
-rw-r--r--  1 john john   25 2011-07-22 07:15 cont_dev
drwxrwxr-x  9 john john 4096 2011-07-27 19:54 .git
-rw-r--r--  1 john john   69 2011-07-27 19:54 newfile1
-rw-r--r--  1 john john   58 2011-07-22 07:15 newfile2
-rw-r--r--  1 john john   45 2011-07-22 07:15 newfile3
-rw-r--r--  1 john john    8 2011-03-31 22:15 temp_file
-rwxrwxr-x  1 john john  114 2011-07-21 21:17 test.sh
john@satsuki:~/coderepo$
```

It should be stressed at this point how destructive the `git filter-branch` command can be to your repository. The **master** and **remove_file** branches have diverged from the point where `newfile1` was first introduced. Consequently all of our other branches, such as **zaney** and **wonderful** still refer to the **master** branch. We would also have to rewrite those branches too, but because of the rewriting of commit objects, we could lose the relationships between the branches and their ancestors. In short, though it is exceedingly powerful, this type of filtering can cause huge distress to other people working on the project.

> **In the trenches...** "So what do we do?" asked John. "We can't push out the repo as it is because it contains the API key." He massaged his forehead moving down to his eyebrows. "But we seem to be introducing a real headache if we filter the branch. Any suggestions?"
>
> "Well the project is going to be finished in a few weeks right?" Simon

> was sitting at the end of the table. He was ashamed and was talking through a pair of hands deperately trying to conceal his identity.
>
> "Yeh, but what the hell has that got to do with it?" snorted Klaus.
>
> "I'm just thinking that we leave the repo like it is until all development has finished," he paused to run his hands through his hair, "then we filter the branch just before we release it." He looked over at John, "At that point there shouldn't be any test or dev branches, and we can just get everyone to clone the repo if we need to do anything else."
>
> John nodded. "You know Simon I think you may have just redeemed yourself."

The idea being proposed here is only really viable because of Tamagoyaki's situation. The code is due to be finished soon and once that happens, the team have decided to push a rewritten branch into the public domain and to resync all of their development repositories to this new branch. It should be noted that the `filter-branch` tool can be used in other circumstances too. We are going to take a look at just one of these. However, let us first clean up our repository a little and move some things around.

```
john@satsuki:~/coderepo$ mkdir tester
john@satsuki:~/coderepo$ ls
another_file  cont_dev  newfile1  newfile2  newfile3  temp_file  tester  test.sh
john@satsuki:~/coderepo$ mv test.sh tester/
john@satsuki:~/coderepo$ git mv newfile* tester
john@satsuki:~/coderepo$ git add tester/test.sh
john@satsuki:~/coderepo$ rm temp_file
john@satsuki:~/coderepo$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# renamed:    newfile1 -> tester/newfile1
# renamed:    newfile2 -> tester/newfile2
# renamed:    newfile3 -> tester/newfile3
# new file:   tester/test.sh
#
john@satsuki:~/coderepo$ git commit -a -m 'Moved testing suite'
[master f08ac57] Moved testing suite
 4 files changed, 9 insertions(+), 0 deletions(-)
```

Since you've been gone

Note

Even though we have rewritten our tree, the fact that another branch still has the file present means that our potentially senitive data still exists somewhere inside the repository. In order to truly get rid of the file we would need to not only remove the file from all branches, or delete the branches that contained the file, but also run a few more steps if we wanted to ensure the file was gone *now*. Be aware that these steps are potentially very destructive to a repository. The best way to remove the file completely would be to remove ALL references to the file and then clone the repository. Git will not clone objects into a new repository if nothing references them. Alternatively if you absolutely must work on the current repository, you would need to do the following.

Delete the `filter-branch` backup using `git update-ref <refname> -d`. (See the callout on *More backups*)

Expire all reflogs with `git reflog expire –expire=now –all`

Repack all of the pack files with `git repack -ad`

Prune all unreachable objects with `git prune`

As you can see some of these are quite scary procedures and so it is important that you understand all that you are doing before you do it.

```
 rename newfile1 => tester/newfile1 (100%)
 rename newfile2 => tester/newfile2 (100%)
 rename newfile3 => tester/newfile3 (100%)
 create mode 100755 tester/test.sh
john@satsuki:~/coderepo$
```

We have reverted back to our **master** branch and in doing so have regained `newfile1`. After that, we deleted our rewritten branch and moved `test.sh` along with all of the `newfiles` into a new folder called `tester`.

Day 4 - "Let's make a library"

Splitting the atom

Sometimes, after a project has been running for a while certain components actually grow rather useful. When this happens, people often want to move it outside of the original project and maintain it as a separate library. Of course the easiest way to do this is to just copy and paste the files out of the main project and into a subdirectory. In doing this we would lose or disconnect all of the development history of that subproject up to this point.

Using the `git filter-branch` we can actually pull out a folder and retain all of its history. The methodology behind this is that we rewrite the history to a new branch, but we only pull across changes to a particular folder and we store those in the root of the branch. Let us see how this works with a quick example. Remember we created the `tester` folder? We are going to make a few commits to the files in this folder to give it some history.

```
john@satsuki:~/coderepo$ echo "More development work" >> tester/newfile1
john@satsuki:~/coderepo$ git commit -a -m 'Work on tester nf1'
[master 1a4956b] Work on tester nf1
 1 files changed, 1 insertions(+), 0 deletions(-)
john@satsuki:~/coderepo$ echo "More dev work" >> tester/newfile2
john@satsuki:~/coderepo$ git commit -a -m 'Work on tester nf2'
[master 7156104] Work on tester nf2
 1 files changed, 1 insertions(+), 0 deletions(-)
john@satsuki:~/coderepo$ echo "Even more dev work" >> tester/newfile3
john@satsuki:~/coderepo$ git commit -a -m 'Work on tester nf3'
[master 1433223] Work on tester nf3
 1 files changed, 1 insertions(+), 0 deletions(-)
john@satsuki:~/coderepo$
```

Now we are going to split that off into a separate branch which we will then clone into a new Git repository. After we have copied the history of the `tester` folder to a new branch, see if you can run through in your head, the steps we would need to take to pull this branch into a new repository.

```
john@satsuki:~/coderepo$ git checkout -b tester_split
Switched to a new branch 'tester_split'
john@satsuki:~/coderepo$ git filter-branch --subdirectory-filter tester
Rewrite 1433223d9c8a8abc35410d12cf78128c318b6e42 (4/4)
```

```
Ref 'refs/heads/tester_split' was rewritten
john@satsuki:~/coderepo$ git branch
  develop
  master
* tester_split
  wonderful
  zaney
john@satsuki:~/coderepo$ ls
newfile1  newfile2  newfile3  test.sh
john@satsuki:~/coderepo$ git checkout master
Switched to branch 'master'
john@satsuki:~/coderepo$ ls
another_file  cont_dev  tester
john@satsuki:~/coderepo$
```

So now the directory has been split away from the original source code into a new branch. Have a think about what steps you would take to bring this into an entirely new repository.

Note

More backups

Git likes to make things easy for you. You may not have noticed it before, but when using the `git filter-branch` tool to rewrite a branch, Git keeps a backup of the value of HEAD before you started rewriting your branch. This backup is kept in `refs/original/refs/heads/<branch_name>`. This file will contain a commit ID which we can use to revert our branch back to its original state, if the filter does horribly wrong.

In the trenches... "So John, I managed to split the Atom library out into a new branch like you said, but I have no idea how to pull this into a new repo." Jack was finally feeling like he had gotten to grips with Git, but his latest task had left him feeling a little dejected. He idly stabbed at his leg with a pen whilst waiting for John to finish his tapping away.

John lifted his keys from the keyboard and turned his chair. "You really can't think of a way to coopy what we have in one repo into another?"

> Suddenly it was like a light bulb had exploded with light inside Jack's skull. "CLONES!" he shouted.

We actually have at least four methods we can use to do this.

1. Copy the data from one repo to another with a simple copy and paste
2. Clone our repository, delete all of the branches other than **tester_split** and then rename it to **master**
3. Initialise a new repository, setup a remote to the original and then fetch our **tester_split** branch
4. Create a bundle of the **tester_split** and then clone from the bundle into a new repository

The first of these will leave us with no history of development at all, so let us ignore it, as it is not what we require. The second of these is trivial and should require no explanation at all. We simply clone and then using the usual tools, we delete all unnecessary branches. However this first method does have its disadvantages, namely the fact that when we clone the repository, we take every single object from the source repository into the new one. Whilst this is generally not a problem it would mean that we would have to run some fairly aggressive garbage collection to remove all of these unwanted objects. This would happen natually over time as the objects aged and were no longer referenced, but it would result in a repository that was initially much larger than it needed to be.

The other two methods deserve a little more consideration as they both perform much better in this respect. The third method you should be familiar enough with previous material to be able to perform right now. However, using the fetch command as we have done so before would again pull in many more objects than we require. As such we are going to do a subtle twist to this command in the following output.

```
john@satsuki:~/coderepo$ cd ../
john@satsuki:~$ mkdir subrepo
john@satsuki:~$ cd subrepo/
john@satsuki:~/subrepo$ git init
Initialized empty Git repository in /home/john/subrepo/.git/
john@satsuki:~/subrepo$ git remote add source /home/john/coderepo
```

```
john@satsuki:~/subrepo$ git fetch source +tester_split:master
fatal: Refusing to fetch into current branch refs/heads/master of non-bare repository
john@satsuki:~/subrepo$ fatal: The remote end hung up unexpectedly
john@satsuki:~/subrepo$
```

What we have asked Git to do is to pull only the branch **tester_split** from the remote we called **source** and place it into **master** locally. Think of the `+<branch>:<branch>` as `+<source>:<destination>` and all will make sense. As you can see Git is not too happy about our intentions here as it does not like overwriting the **master** branch of a non-bare repository. That is OK, we have another way around this.

```
john@satsuki:~/subrepo$ git fetch source +tester_split:tmp
remote: Counting objects: 15, done.
remote: Compressing objects: 100% (14/14), done.
remote: Total 15 (delta 3), reused 0 (delta 0)
Unpacking objects: 100% (15/15), done.
From /home/john/coderepo
 * [new branch]      tester_split -> tmp
john@satsuki:~/subrepo$ git branch -m tmp master
john@satsuki:~/subrepo$
```

So we have almost deceived Git a little here, but I think we can live with ourselves. By first pulling the branch into a **tmp** branch, we were then allowed to rename it as **master**. Notice the number of objects required for this branch `15`. If you remember when we cloned our repository a few *weeks* ago, this value was a lot higher than this. It was the subtle `+<source>:<destination>` which prevented us from pulling every last object from the source repository into our new slim *sub*-repository.

```
john@satsuki:~/subrepo$ ls
john@satsuki:~/subrepo$ git checkout master
Already on 'master'
john@satsuki:~/subrepo$ ls
newfile1  newfile2  newfile3  test.sh
john@satsuki:~/subrepo$
```

Notice that there are no files in the repository until we have checked out. This is because all the fetch did was to *fetch* the objects and place them in the repository object directory. It did not place anything in the working directory. If you remember this is same behaviour we saw with fetching before. So now we have a complete copy of our `tester` component of our repository from the source into a new repository. If we do a `git log`, we can see the history of the development.

```
john@satsuki:~/subrepo$ git log --format=oneline
590e0eb79bc5ba0bc09f611392e643f676b00a04 Work on tester nf3
785b86d877d2a5c0679d98181a23d06ed2ba7652 Work on tester nf2
1ff89f787438f081a0d74de2d26eb2d831c9c738 Work on tester nf1
a5a0d9762dd4b50d8f3228e37b315f6056d5a034 Moved testing suite
john@satsuki:~/subrepo$
```

Unfortunately since some of our development work on these files happened outside of this directory, this was lost when splitting and this is something to keep in mind should you ever perform this kind of operation.

Little bundles of joy

Git has so many ways to do things. This is in part what makes it a little daunting for those just starting but after you have gained a little experience, you begin to understand just what is happening in the background. When this realisation hits, you are able to almost immediately think of at least two different ways of performing the same thing. There have been numerous examples throughout the book, where there have been multiple ways to complete the same task. Here we are going to look at just one more way that we can create a new repo from our **tester_split** branch.

The tool we are going to introduce here is `git bundle`. The `bundle` utility allows us to export a set of revisions and archive them to a file. This file then becomes a resource that can be updated and pulled or fetched from. This is especially useful if you have no physical connection between two computers and wish to sync some of the data from one to the other. Let us take a quick look at how we could use the bundle tool in this case.

```
john@satsuki:~/coderepo$ git bundle create ../tester.bundle tester_split
Counting objects: 15, done.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (15/15), 1.50 KiB, done.
Total 15 (delta 3), reused 0 (delta 0)
john@satsuki:~/coderepo$ cd ..
john@satsuki:~$ git clone tester.bundle subrepo-b
Cloning into subrepo-b...
warning: remote HEAD refers to nonexistent ref, unable to checkout.

john@satsuki:~$
```

The syntax is fairly simple. The word `create` is used to tell Git to create a new bundle. After this we specify a filename and then the tip of the branch that we want to

archive. However, as can be seen above, there is a problem. When we created the bundle, the branch which was checked out at the time was **master**. The objects we pulled from the source repository and placed in the bundle were all from the **tester_split** branch. As such the HEAD of the working tree at the time of the bundle creation, pointed to an object in the **master** branch. Obviously this object does not exist in our bundle and so Git complains. If we had checked out **tester_split** before creating the bundle, there would have been no complaints.

So all we have to do is to remap the HEAD of **master** to that of the HEAD of **tester_split**. As you can see below, it seems as if there are no branches at all and when we try to checkout master it does not exist. What actually happened is that the objects were cloned into the repository, but as the object that the source HEAD pointed to was unavailable, no branch was created. With a little `git reset` trickery, we can create our **master** branch in our new repository.

```
john@satsuki:~$ cd subrepo-b/
john@satsuki:~/subrepo-b$ git branch
john@satsuki:~/subrepo-b$ git checkout master
error: pathspec 'master' did not match any file(s) known to git.
john@satsuki:~/subrepo-b$ git reset --hard origin/tester_split
HEAD is now at 590e0eb Work on tester nf3
john@satsuki:~/subrepo-b$ git checkout master
Already on 'master'
john@satsuki:~/subrepo-b$ ls
newfile1  newfile2  newfile3  test.sh
john@satsuki:~/subrepo-b$
```

Now we have our repository complete as before and we have successfully reampped the **master** branch so that it points to **origin/tester_split**.

> **In the trenches...** Martha and John were sitting together in the office. The rest of the team had left hours ago and it was getting really late. Martha broke the silence, "So we've pulled the Atom library out," she giggled before continuing, "but how the heck do we put it back in again?"
>
> "I'm really not sure said John," taking another swig of coffee before placing the mug back down on the desk. On the side was written the word GIT in large marker pen, a gift from Klaus.

Martha sighed. "It's getting pretty late John. I think I'm gonna head out."

"Yeh, I know what you mean," started John, "I think I'll get going too. Thanks for the help Martha."

"Anytime John."

Day 5 - "Shhh....we're in a library"

Nuclear fusion

OK, so we are not quite at the stage of nuclear physics, but it would be nice to know how to bring our library back into our repository. Git offers a tool called `git submodule`. This tool allows you to link a remote repositories branch and store it under a subdirectory of the project. It does have some nuances which must be learnt, but can be very useful. Let us add our testing suite from the `subrepo` repository into the directory called `tester` in our main `coderepo` repository. First we must remove our `tester` directory.

```
john@satsuki:~/coderepo$ git checkout master
Already on 'master'
john@satsuki:~/coderepo$ git rm tester/*
rm 'tester/newfile1'
rm 'tester/newfile2'
rm 'tester/newfile3'
rm 'tester/test.sh'
john@satsuki:~/coderepo$ git commit -a -m 'Removed tester - will be replaced by
 ↪submodule'
[master 5698499] Removed tester - will be replaced by submodule
 4 files changed, 0 insertions(+), 20 deletions(-)
 delete mode 100644 tester/newfile1
 delete mode 100644 tester/newfile2
 delete mode 100644 tester/newfile3
 delete mode 100755 tester/test.sh
john@satsuki:~/coderepo$
```

We need to define what a submodule actually is. Submodules are tricky to understand and often people use them once and conclude that they are more trouble than they are worth. However, if you take some time to understand what a submodule really is, then they can be very useful to you. A submodule is the inclusion of a repository branch at a specific commit. It is not intended to track the development of the upstream library or module, (see the callout box for an explanation of *upstream*).

Terminology

Upstream

Upstream refers to the source of a project which may have one or more derivatives which are also distributed. Take the package that was used to build this book for example, LaTeX. LaTeXis distributed by the people who developed it as open source software, but it is also included with a number of Linux distributions. The location of the software created by the LaTeXdevelopers is referred to as the *upstream* project. The projects which include it within their own are what is referred to as *downstream*. Think of it like a river which flows from the source further *upstream*.

As we will see, though it can be a little longwinded to actually change the version of the code that the submodule refers to, it actually makes a lot of sense to handle them in this way. If the code in the submodule is being included in your repository, you do not want to run the risk of a change upstream resulting in a broken build for your project. This is why submodules always refer to a single commit.

Let us go ahead, create a submodule and then discuss the steps we have taken.

```
john@satsuki:~/coderepo$ git submodule add /home/john/subrepo tester
Cloning into tester...
done.
john@satsuki:~/coderepo$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# new file:   .gitmodules
# new file:   tester
#
john@satsuki:~/coderepo$ git commit -a -m 'Added submodule (subrepo)'
[master 2aadc11] Added submodule (subrepo)
 2 files changed, 4 insertions(+), 0 deletions(-)
 create mode 100644 .gitmodules
 create mode 160000 tester
john@satsuki:~/coderepo/tester$
```

As you can see we had to perform a number of steps before we obtained the source for the **subrepo** library in our `tester` directory. We had to begin by using `git submodule` to add the upstream repository. The upstream repository is really just like any remote repository we have been using, but we will use the terminology *upstream* to make a distinction. The command `git submodule add /home/john/subrepo tester` creates a special file in the root of our project called `.gitmodules`, plus it clones the upstream repository into the folder we specified, in this case `tester`.

Notice that when we ran `git status`, we saw two new entries, one for `.gitmodules` and one for `tester`. Next we have to commit those entries using the standard `git commit` command. When we do, we see that there is a code in front of `tester` which is special and tells Git to treat this directory as a submodule.

Though the submodule has now been added, it has not yet been initialised. To do this, we run our next set of steps.

```
john@satsuki:~/coderepo$ git submodule init
Submodule 'tester' (/home/john/subrepo) registered for path 'tester'
john@satsuki:~/coderepo$ git submodule update
john@satsuki:~/coderepo$
```

Now our submodule has been added and initialised. The update command is used to ensure that the directory `tester` contains the version of the submodule that we committed earlier.

```
john@satsuki:~/coderepo$ cd tester/
john@satsuki:~/coderepo/tester$ ls
newfile1  newfile2  newfile3  test.sh
john@satsuki:~/coderepo/tester$ git log --format=oneline
590e0eb79bc5ba0bc09f611392e643f676b00a04 Work on tester nf3
785b86d877d2a5c0679d98181a23d06ed2ba7652 Work on tester nf2
1ff89f787438f081a0d74de2d26eb2d831c9c738 Work on tester nf1
a5a0d9762dd4b50d8f3228e37b315f6056d5a034 Moved testing suite
john@satsuki:~/coderepo$
```

Looking in the directory we can see two things. The first, is that the files present in the **subrepo** upstream project have now been added. The second, may appear a little suprising to begin with. The `git log` command actually shows a log for the upstream project, not for the local root project stored in `coderepo`. In all honesty, the submodule repository is actually just a clone of the upstream project, with a few subtle differences.

The information about which upstream url to use for the project can be found in the `.gitmodules` which we committed earlier. Below is an example of what the file looks like in our current repository.

```
john@satsuki:~/coderepo$ cat .gitmodules
[submodule "tester"]
  path = tester
  url = /home/john/subrepo
john@satsuki:~/coderepo$
```

Changes down the river

So what happens when we want to pull in changes from the upstream project? Well, you can make your submodule point to whatever commit you like and stay there. As long as you commit your changes in the super project, Git will always allow you to return to that point using the `git submodule update` command.

Let us take a look at how we could pull in some changes into our `tester` submodule. First, we are going to make a change to our upstream project.

```
john@satsuki:~/coderepo$ cd ..
john@satsuki:~$ cd subrepo
john@satsuki:~/subrepo$ ls
newfile1  newfile2  newfile3  test.sh
john@satsuki:~/subrepo$ echo "Added a new function" > newfile4
john@satsuki:~/subrepo$ git add newfile4
john@satsuki:~/subrepo$ git commit -a -m 'Added a new library file'
[master 94ad27e] Added a new library file
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 newfile4
john@satsuki:~/subrepo$ cd ..
john@satsuki:~/subrepo$
```

Now that we have a new version of the project, let us try to pull those changes into our superproject.

```
john@satsuki:~$ cd coderepo
john@satsuki:~/coderepo$ cd tester
john@satsuki:~/coderepo/tester$ git status
# On branch master
nothing to commit (working directory clean)
john@satsuki:~/coderepo/tester$ git fetch origin
```

```
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From /home/john/subrepo
   590e0eb..94ad27e  master     -> origin/master
john@satsuki:~/coderepo/tester$ git checkout master
Already on 'master'
Your branch is behind 'origin/master' by 1 commit, and can be fast-forwarded.
john@satsuki:~/coderepo/tester$
```

As you can see, we are told that our branch is currently one commit behind that of **origin/master**. If we want to update our **master** branch in the submodule, we need to *pull* our changes in, just like a **real** Git repository.

```
john@satsuki:~/coderepo/tester$ git pull
Updating 590e0eb..94ad27e
Fast-forward
 newfile4 |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 newfile4
john@satsuki:~/coderepo/tester$ ls
newfile1  newfile2  newfile3  newfile4  test.sh
john@satsuki:~/coderepo/tester$ cd ..
```

Now let us see what happens if we try to update the module.

```
john@satsuki:~/coderepo$ git submodule update
Submodule path 'tester': checked out '590e0eb79bc5ba0bc09f611392e643f676b00a04'
john@satsuki:~/coderepo$ cd tester
john@satsuki:~/coderepo/tester$ ls
newfile1  newfile2  newfile3  test.sh
john@satsuki:~/coderepo/tester$
```

Our new changes have disappeared. How odd! Well actually not really. As we stated earlier, when we committed our `.gitmodules` file along with the `tester` directory, we not only committed the fact that we required a submodule, we also committed the exact point we wanted that submodule to point to. If we want to change this, then we must commit that as a change. It may seem a little odd that we have to jump through these hoops to get an update to an upstream project, but if you think about it, it actually makes a lot of sense. It means that anyone cloning our repository is sure to get a version of the submodule that we have decided is right for the project. So keeping this in mind,

let us walk through a quick example of how we would finish the job and commit a new version of the submodule.

```
john@satsuki:~/coderepo$ cd tester/
john@satsuki:~/coderepo/tester$ git pull
You are not currently on a branch, so I cannot use any
'branch.<branchname>.merge' in your configuration file.
Please specify which remote branch you want to use on the command
line and try again (e.g. 'git pull <repository> <refspec>').
See git-pull(1) for details.
john@satsuki:~/coderepo/tester$
```

Interesting! What has happened here is that by performing the `git submodule update` command, we effectively asked Git to checkout a commit. Remember in the past we talked about detached HEAD? This is exactly what Git has done. A submodule spends most of it's life in a detached HEAD state. As we tell Git that we must have the submodule at a specific commit, it means that Git checks out a commit, rather than a branch. If you think about it, this makes sense, we do not want the contents of the module *changing*.

So to bring our module up to date, we need to first checkout master. Then we can issue our `git pull`.

```
john@satsuki:~/coderepo/tester$ git checkout master
Previous HEAD position was 590e0eb... Work on tester nf3
Switched to branch 'master'
john@satsuki:~/coderepo/tester$ git pull
Already up-to-date.
```

Oh? Should we not have seen some commits pulled in here? Actually, no. We pulled the changes into master earlier, when we ran the `git pull`. When the module reverted to the earlier commit, **590e0eb**, it did not affect the master branch at all, as we simply checked out a single commit. So by swiching to **master**, we have already altered the contents of the submodule directory, as can be seen below.

```
john@satsuki:~/coderepo/tester$ ls
newfile1  newfile2  newfile3  newfile4  test.sh
john@satsuki:~/coderepo/tester$ cd ..
john@satsuki:~/coderepo$ git status
# On branch master
# Changes not staged for commit:
```

```
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
# modified:   tester (new commits)
#
no changes added to commit (use "git add" and/or "git commit -a")
john@satsuki:~/coderepo$
```

All we need to do now is to commit the submodule changes into the repository and check that the update yields the new file.

```
john@satsuki:~/coderepo$ git commit -a -m 'Up revd upstream module'
[master 022a163] Up revd upstream module
 1 files changed, 1 insertions(+), 1 deletions(-)
john@satsuki:~/coderepo$ git submodule update
john@satsuki:~/coderepo$ cd tester/
john@satsuki:~/coderepo/tester$ ls
newfile1  newfile2  newfile3  newfile4  test.sh
john@satsuki:~/coderepo/tester$ cd ..
john@satsuki:~/coderepo$
```

As you can see, submodules can be rather useful. You can even make changes to the repository in the submodule and commit them locally to perhaps keep changes that you want to make to the submodule. As this is a Git repository in its own right, you can merge *upstream* changes in too! Remember though that if you did make changes, and you committed them to the submodule, if you then issued a `git submodule update` without first committing your changes in the superproject, your commit would be lost. Of course nothing in Git is ever really lost, but it would be prudent of you to always keep changes you make to submodules in a branch, that way they are easy to bring back if you make a mistake like the one described.

With that all said and done, we have finished our tour of the major portions of Git. What follows in the next chapter are some other points that are added more for information on what **can** be done with Git.

Summary - John's Notes

Commands

- `git apply <filename>` - Applies a patch to the working tree
- `git reflog show <branch>` - Show the reflog only for the specified branch
- `git format-patch <ref1>..<ref2>` - Create a set of patches of each commit between two points
- `git am <filename>` - Apply a specific patch containing a *format-patch* file
- `git bisect start` - Begin a bisect session
- `git bisect good <ref>` - Mark a reference as good, during a `git bisect`
- `git bisect bad <ref>` - Mark a reference as bad, during a `git bisect`
- `git bisect start <ref_recent> <ref_old>` - Start a bisect session between two known points
- `git bisect run <command>` - Start an automated run of the bisect tool
- `git filter-branch –index-filter 'git rm –cached –ignore-unmatch <file>' HEAD` - Rewrites the current branch to remove file
- `git filter-branch –subdirectory-filter <directory>` - Rewrites the current branch to make subdirectory directory the root of the branch
- `git fetch <remote> +<remote_branch>:<local_branch>` - Creates a local branch from the remote branch existing in a remote repository
- `git branch -m <old_branch> <new_branch>` - Move or rename a branch from old to ne
- `git bundle create <filename> <branch>` - Create a bundle file in filename, containing all the objects and references from branch.
- `git submodule add <repo> <path>` - Add a submodule at the directory specified by path

- `git submodule init` - Initialise any submodules in the super project
- `git submodule update` - Pull all submodules back to the points that have previously been committed to

Terminology

- **Patching** - A method of distributing changes from someone elses repository without having a line of communication between the two, or without a user having access to commit into the destination repository
- **Bundle** - A type of archive file that hold objects and commits and can be pulled from
- **Bisect** - A way of progressively searching through a repository to find where bugs were introduced
- **Filtering** - Takes a branch and rewrites it according to a set of rules
- **Submodule** - Incorporating a remotely reachable project as a subdirectory of a superproject
- **Superproject** - A Git repository containing one or more submodules

After Hours Week 8

"Fishing for beginners"

Hooking your scripts up

So we come to the last After Hours section of the book and this time we are going to take a look at hook scripts. Hook scripts are tiny pieces of code that you are allowed to have executed at certain points during Git's processes which can affect the outcome of those same processes. Imagine for example that whenever you committed to a repository you wanted an email to be sent to the project manager. That is a job for a hook script.

The are a fairly large number of hook points available. They are called hooks because it allows you to hook something else into the Git process. Below is a list of a few of the available hook points along with a very brief description of when they are triggered and how they can affect the process. For a full list, you should check out the manual for githooks.

- **applypatch-msg** - Invoked by `git am`, it is provided with the name of the proposed commit message for the patch and allows for sanitisation. Exiting with a non-zero status will cause `git am` to abort the patch. Will run the **commit-msg** hook if enabled.
- **pre-applypatch** - Invoked by `git am`, it runs immediately after the patch is applied, but before a commit is made. In this way, checks can be made to the working tree before committing. Exiting with non-zero status aborts the commit. Will run the **pre-commit** hook if enabled.
- **post-applypatch** - Invoked by `git am`, it runs after the patch is applied and the commit is made. It does not affect the flow of `git am` and as such is useful for actions such as notifications.
- **pre-commit** - This hook is invoked by `git commit` and can be bypassed with the `–no-verify` option. Exiting with non-zero status aborts the commit.

- **prepare-commit-msg** - This hook is invoked by `git commit` and is run immediately after the commit message has been prepared, but before the message editor has been loaded. It takes three parameters, the file containing the log message, the source of the commit message and an SHA hash. Exiting with non-zero status aborts the commit. It should not be used to replace **pre-commit**.

- **commit-msg** - This hook is invoked by `git commit` and can be bypassed with the `–no-verify` option. The hook is used to normalise messages in place before commit. Exiting with non-zero status aborts the commit.

- **post-commit** - Invoked by `git commit`, it runs immediately after a commit is made. It can not affect the flow of `git commit` and as such is useful for actions such as notifications.

- **pre-rebase** - Invoked by `git rebase`, it is useful for preventing a branch from being rebased.

- **post-checkout** - Invoked by `git checkout` after it has updated the working tree. The hook is given the ref of the previous HEAD, the new HEAD and a flag indicating if it was a branch or file checkout. It is also invoked after `git clone`, but can be overidden by the use of `–no-checkout`. The script is most useful for performing sanity and validity checks on the working tree post checkout. It does not affect the process of `git clone` or `git checkout`

- **post-merge** - Invoked by `git merge` when run as part of a `git pull`. It does not affect the outcome of the merge, but does not get executed if the merge fails due to conflicts.

So we have a large arsenal of hooks that we can employ to assist us in administrating our repository. We are going to set up a fairly simple example here of ensuring that we do not delete one of our core files. To begin with, let us add a `corefile` to the repository in a new branch and play a little with a new hook script.

```
john@satsuki:~/coderepo$ git checkout -b hooktest
Switched to a new branch 'hooktest'
john@satsuki:~/coderepo$ echo "Very Important File" > corefile
john@satsuki:~/coderepo$ git add corefile
john@satsuki:~/coderepo$ git commit -a -m 'Added Core File'
```

```
[hooktest 96fe8b9] Added Core File
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 corefile
john@satsuki:~/coderepo$
```

Now let us create a hook script called `.git/hooks/pre-commit`, with the following code.

```
#!/bin/bash

if [ -e "corefile" ]
then
  echo "File OK"
  exit 0
else
  echo "File Deleted Not Committing"
  exit 2
fi
```

Now let us see what happens if we make a commit that does not affect the `corefile`.

```
john@satsuki:~/coderepo$ echo "New Development" >> cont_dev
john@satsuki:~/coderepo$ git commit -a -m 'Added a commit'
File OK
[hooktest 73656f2] Added a commit
 1 files changed, 1 insertions(+), 0 deletions(-)
john@satsuki:~/coderepo$
```

We see the text `File OK` in the output, which indicates that the commit is allowed to go ahead. What happens if we try to remove the core file and commit the result?

```
john@satsuki:~/coderepo$ git rm corefile
rm 'corefile'
john@satsuki:~/coderepo$ git commit -a -m 'Removed a corefile'
File Deleted Not Committing
john@satsuki:~/coderepo$
```

We have been prevented from committing because we removed the all important `corefile`. To be able to commit again, we must return the file.

```
john@satsuki:~/coderepo$ git checkout HEAD -- corefile
john@satsuki:~/coderepo$ ls
another_file  cont_dev  corefile  tester
```

```
john@satsuki:~/coderepo$ echo "New development" >> cont_dev
john@satsuki:~/coderepo$ git commit -a -m 'Some more development'
File OK
[hooktest 6786dd5] Some more development
 1 files changed, 1 insertions(+), 0 deletions(-)
john@satsuki:~/coderepo$
```

So you see hooks can be really simple, or they can do highly complex things. Imagine a hook that would compile the code in the working tree prior to the commit and only allow it to continue if the code compiled cleanly. That could be pretty useful in ensuring that there is always good clean buildable code in the repository.

A little extra help

Taking things further

We have completed our journey through eight weeks of intensive version control usage at Tamagoyaki Inc. However, there are probably still many questions that you have regarding Git and its usage. This chapter focusses on taking a few of the topics that were discussed a little further but does not go into as much detail as was presented in the rest of the book. The reason for this is purely that as the tasks you are trying to achieve become more complex, they often require greater knowledge about the implementation details and are often heavily customised to fit a particular scenario. Also, some of the topics are rather specialised and are not of use to everybody, so taking a large amount of time discussing them may be a little redundant for the majority of the people reading the book.

If you have read through weeks 1-8, and supplemented it with the After Hours sections, you should have a very good understanding of the Git system. As with most things practice makes perfect and it is important to play with any new system before using it in earnest. If you are planning to migrate away from an existing version control system, take some time to test your current processes and procedures in Git to ensure that it will fit your needs. Always keep in mind that no matter which system you try to introduce, there will nearly always be resistance from someone.

So let us touch on a few more topics and just wrap up a few loose ends.

Collaborating with a larger audience

We looked at a few tools within the Git system to allow access to a Git repository, but what if you want to share your repository with a larger audience. As stated earlier, the implementation of the tools in our testing was suitable for merely that, testing. In a production environment one would should never use the instaweb tool for a providing a permanent setup where people can browse a repository over HTTP.

There are two distinct methods for handling the distribution of Git repositories to larger audiences. Generally people will either, set up the server themselves or use a third

party system to host their repositories.

GitHub

No book on Git would be complete without at least a small section on GitHub. GitHub is a third party site for hosting and interacting with Git repositories. All of the information about GitHub is correct at the time of going to print.

GitHub offers the ability to host both public and private repositories. Public repositories can be hosted for free, whereas private repositories require a subscription fee, however the pricing is tiered to allow you to start small and gradually grow your usage. Repositories can be pulled via HTTPS, SSH and Git protocols which allows you to be entirely flexible in the methods you use to keep up to date. Pushing to the repositories can be achieved via HTTPS or SSH, which again is rather flexible.

The real beauty of GitHub is how it is presented. GitHub has a tagline of *Social Coding* and it really does live up to that name. People can fork your project, which basically means they have cloned it to their account in GitHub. Whenever they make commits to their fork, you can see these changes pop up in a section of the site called Network.

GitHub also allows you to edit files directly on the site and commit them. This means that wherever you are, as long as you have a connection to the Internet and a browser, you can edit files in your repository and commit them. When you return home, you can issue a pull and the changes will appear on your local system.

For open source projects, GitHub really does offer a wealth of features for collaborative coding. When a collaborator has made enough changes to fix a bug or a feature, they can issue a pull request. This sends a notification to you that Person X would like you to merge their changes into your branch. If there are no conflicts these pull requests can be manage straight from GitHub, without requiring you to checkout the remote branch, merge it and push it back again. You can also comment on lines in the diff and many more features.

Indeed during the writing of this book, several contributors used GitHub to make pull requests containing spelling mistakes and design suggestions.

Hosting yourself

Many people choose to host themselves, be it for security or control reasons. Sometimes people do not want to have their data *online* and prefer to keep it internal to the company, this may be due to IP, license or other restrictions. However, it should be

noted that this option does not pose a problem to the implementation of Git. Though you may not have access to the fancy interface features of GitHub, you still get the raw power of Git in your workplace.

Apache is a common implementation of Git for companies as it is a robust webserver which offers the ability to serve and receive data through HTTP and HTTPS with a Git repository. We are not going to go into the implementation details of such a set up as each one is different. There are many tutorials on the Internet that deal with such set ups in different situations and for different purposes.

SSH is also a good choice for implementing a Git repository and there are several means of securing the Git repository to allow only certain users access to certain paths etc. Git by default does not really have a security access control layer but then that is realy by design more than anything else. Git was intended to be used by open source projects to share and collaborate on code. However, this does not mean it can not be used on closed source projects, just that sometimes some modifications need to be made if more advanced features like access control are required. Many people rely on the work flow processes to ensure that things are done in a secure manner. If you have a blessed repository and have some dictators who are responsible for integrating code into it, it should be up to them to ensure that Developer X does not try to commit changes to Module Y.

Taking out the garbage

Something that we glossed over is the subject of garbage collection. Git handles this for us, routinely deleting objects that are older than a certain time period from the object database, as long as they are no longer referenced by any branches. So by this we mean that an object that exists only in one branch will only remain imune to garbage collection whilst that branch is alive. As soon as we delete the branch, all objects that are not longer referred to in any tree are left in a *dangling* state and may be deleted when garbage collection runs. Of course as stated earlier, Git holds on to objects for a period of time for you, to make sure you do not do anything silly, like delete a branch you actually really wanted. You can invoke a garbage collection manually, but it is often better to let Git handle these things for you, unless you have a specific reason for doing so.

But how do you really know....you know?

When obtaining source code via an insecure means, over the Internet for example, there are times when you would really like to be as sure as possible that someone has not tampered with it. Let us take an example of someone distributing their content using a Git repository. An attacker takes control of the repository, puts in some malware and tags the commit as release version 1.0. People start do download it and as you can imagine all hell breaks loose.

It does not have to be this way. Many developers and Internet users use GPG to encrypt and sign their emails and documents. GPG is a tool which can be employed to both encrypt files so that they can only be read by the intended recipient, or to sign documents and files, such that you can be reasonably sure that document is the same as when the person signed it. The private GPG *key* of the signing party is used along with the file to generate a code which is sent along with the email. When it reaches the recipient a check can be made against the users public key and the content of the email, to validate that said user did actually sign that particular document or file.

Git allows you to use GPG signing for tags. This gives people piece of mind that unless an attacker has gained access to both the repository and the developers private GPG key, if the sign a tag, that tag is exactly as it was intended. To use Git in this way, you must first set up a GPG key, which is out of scope of this book. Once you have this key, you can change your config like the example below, to tell Git to use a particular key when signing.

```
git config user.signingkey 0xABCDEFG
```

Then when you want to sign a tag, simply append the `-s` parameter to the `git tag` command. You can also verify someone elses signed tag by running `git tag -v <tagname>`

The end of the journey

Well we have reached the end of our journey. We have taken a look at many of the aspects of both Git and the mechanisms by which it can be used. As the workflow of actually using Git is so open, you will have to decide for yourself how you want to collaborate with others. Do not worry if you do not get it right first time. Try a test repository and make up scenarios, just like we have in the book, to test your workflows.

Using a version control system is not an exact science and should never be something that people dislike so much that they resent using it. A good implementation of a version

control system invents minimal work for the users, whilst giving themall the benefits we have desribed throughout. Remember if your users dislike the system, the chances are they will not use it properly and sometimes will try at every stage to avoid it. Help them to understand why you are using a version control system and get them onside early. You will find the procedure will be much less painless.

Before we sign off for good, let us take one more trip back to the trenches to see how things panned out.

> **In the trenches. . .** "Well I have to say guys there were points during this project that I didn't think we'd ever complete it" said Markus with a grin on his face. The rest of the team all smiled, including Klaus.
>
> "It wasn't the easiest thing we've ever done," started John, "but it sure has helped us out in so many ways." Markus grinned, "Well ya big bunch of gits, hows about we head off early today and take a trip to the pub. First round is on me."
>
> The room roared with shouts and cheers as they began to file out of the room, gather their things and head towards the Dog and Duck.
>
> * * *
>
> The team had been in the pub for well over two hours and several of Tamagoyaki Inc's were beginning to feel the effects of the consumption of alcohol. Martha and John had broken away from the rest of the pack and were sat on a table on their own. It hadn't been intentional, but their conversation hadn't seemed to fit with the rest of the team.
>
> "Thanks for all the help John," started Martha, "You've been really good to me."
>
> "Hey! I'm just glad you're on my team." he replied. "We certainly couldn't have done it without you."
>
> At that point the effects of the alchopop were too much for Martha. Her cheeks began to flush a bright shade of red, and in a totally out of character manouver she leant across the table and kissed John.
>
> The rest of the team caught a glimpse of the events taking place in the isolated environment and cheered loudly.

"Geee, it's about time," jeered Klaus.

Acknowledgements

"A Huge Thank You"

Proofing and Ideas

These people gave their time and effort to do forking and proofing, and giving marvellous suggestions about the book during development:

Johan Sageryd
Og Maciel
Alistair Buxton
Miia Ranta (Myrtti)
Hassan Williamson (HazRPG)
(Synth_sam)
Jonas Bushart

LaTeX Support

These people provided invaluable LaTeX support:

Johan Sageryd
Kevin Godby
Matthew Johnson
Ben Clifford

Git Support

#git on freenode.net

GitHub.com for hosting the Git repository of GITT
Wikipedia.org for providing information on the history of version control systems

Index

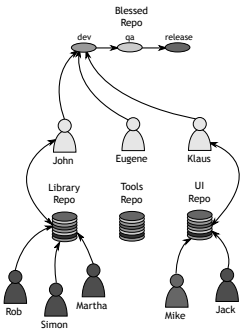
Master the most powerful version control system

Covers all the topics below and more:

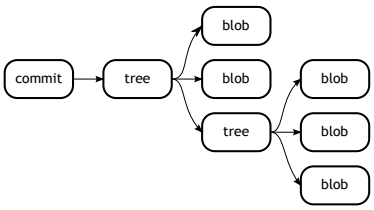
Committing	GUI usage
Resetting	Stashing
Staging	Cloning
Logging	Networking
Diffing	Rebasing
Tagging	Patching
Branching	Bisecting
Merging	Filtering Branches
Conflicts	Recovering Branches

This book is designed to help you both apply and understand the subtleties of Git, perhaps the most powerful version control system in use today. This book is not supposed to be purely a technical reference. GITT follows the lives of some developers at a fictional company called Tamagoyaki Inc. a small software outfit who write bespoke software for people.

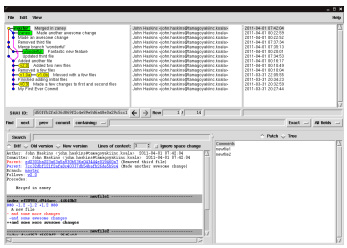
GITT follows developer John, as he works to implement their first version control system. The chapters are presented as weeks during the implementation of the project. Each chapter spells a new week in the project helping you to learn the tricks of the Git trade.



Detailed workflow analysis



Diagrams detailing the Git methodology



Inclusion of screenshots of Git GUI

- Covers all 21 of the standard Git commands
- Split up into easy to manage chunks
- Includes After Hours sections for greater understanding
- Complete Git output from all new commands
- Scenario based for associative learning

1 234567 891231