CAB302, *Software Development*
Week 4 – Coming to grips with Git

In the lectures this week, you were introduced to version control, some concepts and history and then to the practical usage of Git on a project source tree. In this prac exercise, we will work with Git mainly on a local repository which we will create. The exercise is designed to reinforce a couple of issues considered in the lecture demos, but also to incorporate some of the other facets of version control which are crucial when dealing with complex projects.

Installing Git

Instructions for installing Git can be found in the Portable Software Installation Guide under Guides and Other Resources on Blackboard, if you have not yet installed it.

Getting Started:

If you are unfamiliar with Git, you should begin by working through the demo in the lecture. You should be able to undertake everything except the push to the remote repository. Towards the end there is some guidance on how to push to a remote repository on GitHub. Once you are sure you have mastered these basics, begin with the clean repository described below. By the time you reach this, you may already have set up the `user.name` and `user.email`. If you would like a refresher or to learn more, the GitHub tutorials are a fantastic resource (https://guides.github.com/).

Common Unix and Vi Commands

When setting up Git, you may have selected the Vi text editor as your default editor. In addition, the Git Bash environment, which is preferred for working with Git from the command line features a Unix-like environment. In order to use these, you will need to know some common commands of the Unix operating system and the Vi text editor. Some of the Unix commands are similar to the cmd.exe equivalents, however, Vi can be a bit opaque to those who have never used it before. Please ask your tutors for assistance.

Note there will be times during this practical when you will be re-entering previous commands. As a shortcut, you can use the up and down arrows to cycle forward and backward through your recent commands.

Unix Task	**Command**
List files in directory	`ls`
Output current directory to the screen	`pwd`
Change directory	`cd dirname`
Make directory	`mkdir dirname`
Remove Directory	`rmdir dirname`
Remove a file	`rm filename`
Output a text file onto the screen	`cat filename`
	`more filename`

Vi Task	**Command**
Switch to command mode	Press the esc button
Save (write) a file	`:w` (press enter)
Quit vi	`:q` (press enter)

Branches and Merging:
As in the lecture, we will work from a directory which is initially empty. Most of the commands will match a sequence that we have seen before. Change to a directory in which you have write access, and create a subdirectory called week4. This will be the project directory and we will make it a Git repository from scratch.

```
$ mkdir week4
```

Change into this directory.

```
$ cd week4
```

At this stage, follow our other examples, and create an empty repository:

```
$ git init
```

If you have not done so, set the local git configs as in the lecture:

```
$ git config --global user.name "My Name"
$ git config --global user.email me@example.com
```

Now the focus of this prac is to reinforce some of the concepts introduced in the lecture, but also to introduce the crucial issues of branching and merging. A branch on a source tree is, as the name suggests, a variation on the code base which differs in some sense from the core project code base, usually referred to as the trunk. Branching has already been introduced to you through the Git status at the command line:

```
hogan@SEF-EEC-056940 /c/Data/week4 (master)
```

Here the repository is aligned with the master branch, the trunk. Git maintains successive commits as a linked list, with the branch ID pointing to the most recent version. Here we will work with a very simple repository. With the editor of your choice, create a file called README.txt in the week4 directory. We will successively add lines to this file to indicate the version. In the first version, enter the line:

```
This is version 0 of the file
```

As before, save, stage and commit this file, with an appropriate message. So, for the initial commit, the commands are:

```
$ git add README.txt
$ git commit –m “Version 0”
```

Iterate through this approach four times. After this stage, your readme file should have the contents (each on a separate line):

```
This is version 0 of the file
This is version 1 of the file
This is version 2 of the file
This is version 3 of the file
```

And the use of the git log command yields:

```
$ git log
commit bbc6fcbb9139249d234f5a4bb12b2813ecdf921d
Author: James M Hogan <j.hogan@qut.edu.au>
Date:   Thu Apr 17 00:10:30 2014 +1000

    Version 3

commit 0f4d71251670a283391b4c185263532600f02173
Author: James M Hogan <j.hogan@qut.edu.au>
Date:   Thu Apr 17 00:10:13 2014 +1000

    Version 2


commit 4ec97255eeab3ddb694cb178cdaccc8912ea6633
Author: James M Hogan <j.hogan@qut.edu.au>
Date:   Thu Apr 17 00:09:51 2014 +1000

    Version 1

commit 45ddefb1020456c592628a943341cf3e033bb719
Author: James M Hogan <j.hogan@qut.edu.au>
Date:   Thu Apr 17 00:09:18 2014 +1000

    Version 0
```

This means that we now have a basic trunk, consisting of a list of commits, each showing a minor variation on the README file. ***NOTE: The commit hashes that you will see will almost certainly differ from those shown in my log above. This is of course how it should be.***

A short note here on the hash codes associated with each commit. Being long hashes, they uniquely identify each commit in a way that a comment cannot. But they are ridiculous to type, and even to copy and paste. Fortunately, git accepts shortened versions of them which is useful for operations such as revert.

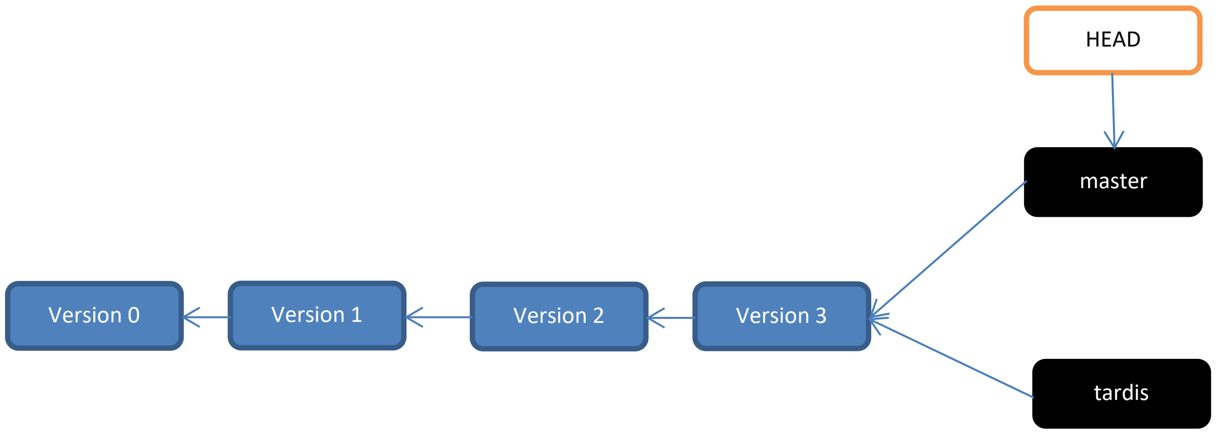
We are now going to go back in time, removing some of these silly lines in the file. We choose to explore this option on a branch which reflects our intention. The choice of name is arbitrary; I have chosen this one to be cute – many others are possible.

[If you don't know what a TARDIS is, see: http://en.wikipedia.org/wiki/TARDIS].

We first create the branch:

```
$ git branch tardis
```

At the moment, both the tardis and main pointers refer to the same commit, but the current HEAD pointer – the indicator of the current branch – points to the master:

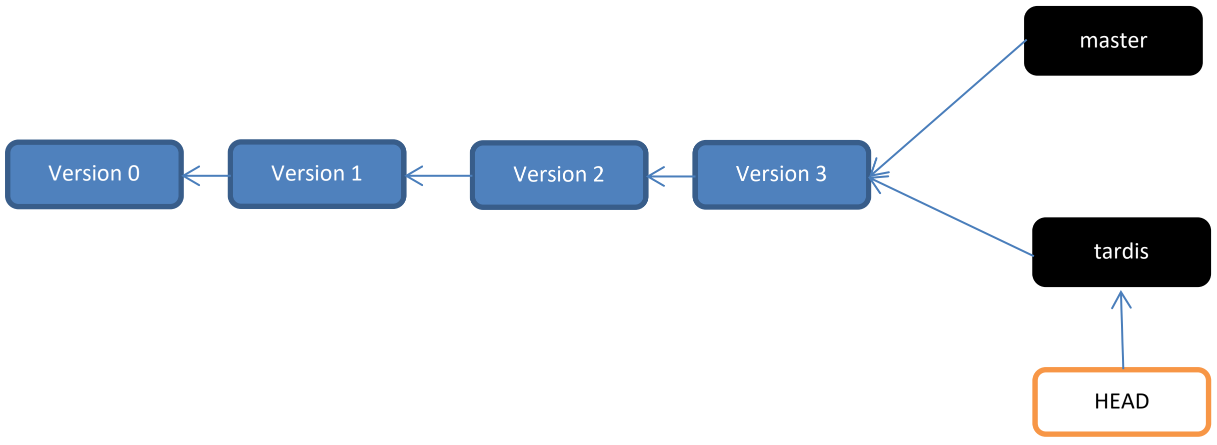


When we switch to the tardis:

```
$ git checkout tardis
```

We see some crucial changes indicating a new branch, but no real content changes:

```
hogan@SEF-EEC-056940 /c/Data/week4 (tardis)
```



We now operate on the tardis branch, and we do some other commits, winding back time. Each Tardis commit removes a line, but we keep it new by also adding a line of the form:

```
TARDIS 1
TARDIS 2
```

depending on the iteration. So after 2 such changes we have, using commands such as:

```
$ git add README.txt
$ git commit –m “Version T1”
```

the following results:

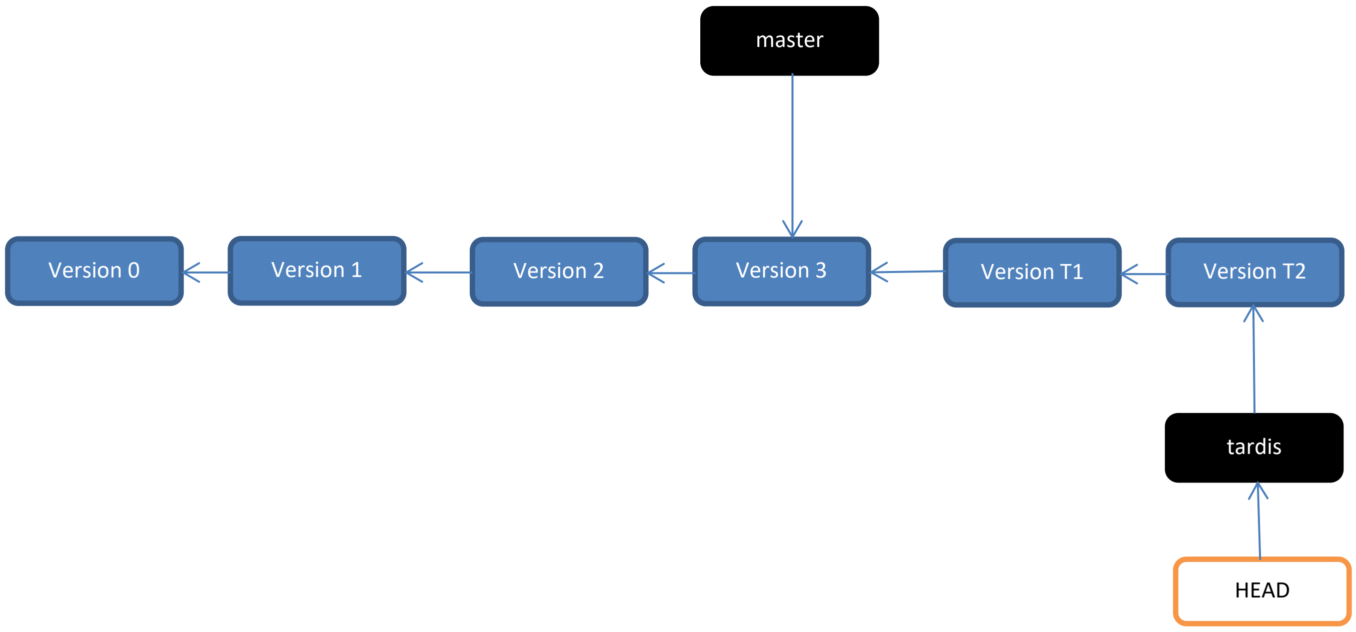
README.txt:
```
This is version 0 of the file
This is version 1 of the file
TARDIS 2
TARDIS 1
```

Screen log:
```
[Edits here on the tardis branch]

hogan@SEF-EEC-056940 /c/Data/week4 (tardis)
$ git add README.txt

hogan@SEF-EEC-056940 /c/Data/week4 (tardis)
$ git commit -m "Version T1"
[tardis 6605651] Version T1
 1 file changed, 1 insertion(+), 1 deletion(-)

hogan@SEF-EEC-056940 /c/Data/week4 (tardis)
[edits here]

hogan@SEF-EEC-056940 /c/Data/week4 (tardis)
$ git add README.txt

hogan@SEF-EEC-056940 /c/Data/week4 (tardis)
$ git commit -m "Version T2"
[tardis 495431d] Version T2
 1 file changed, 1 insertion(+), 1 deletion(-)
```

So we now have a branched repository for no particularly good reason, and we think about whether we can now collapse this back onto the trunk. Usually you will branch for one or more of a number of good reasons, with the most common being:

- To try code to resolve a bug
- To maintain a separate development branch which does not infect the stable master

The situation is as shown below, with the HEAD still pointing to the tardis branch.



A quick demonstration: open up the README.txt file and look at the contents. Now switch the branch back to the master:

```
$ git checkout master
```

As well as the expected change back to the master in the prompt:

```
hogan@SEF-EEC-056940 /c/Data/week4 (master)
```

we also find that the README file has been overwritten with the old version:

```
This is version 0 of the file
This is version 1 of the file
This is version 2 of the file
This is version 3 of the file
```

We will now extend the master branch as well, before coming back to sanity and merging our two branches. Try the commands below (the * in the results indicates the HEAD branch):

```
$ git branch
$ git branch --no-merged
```

The `--no-merged` filter shows only those branches which have not yet been merged with the current branch.

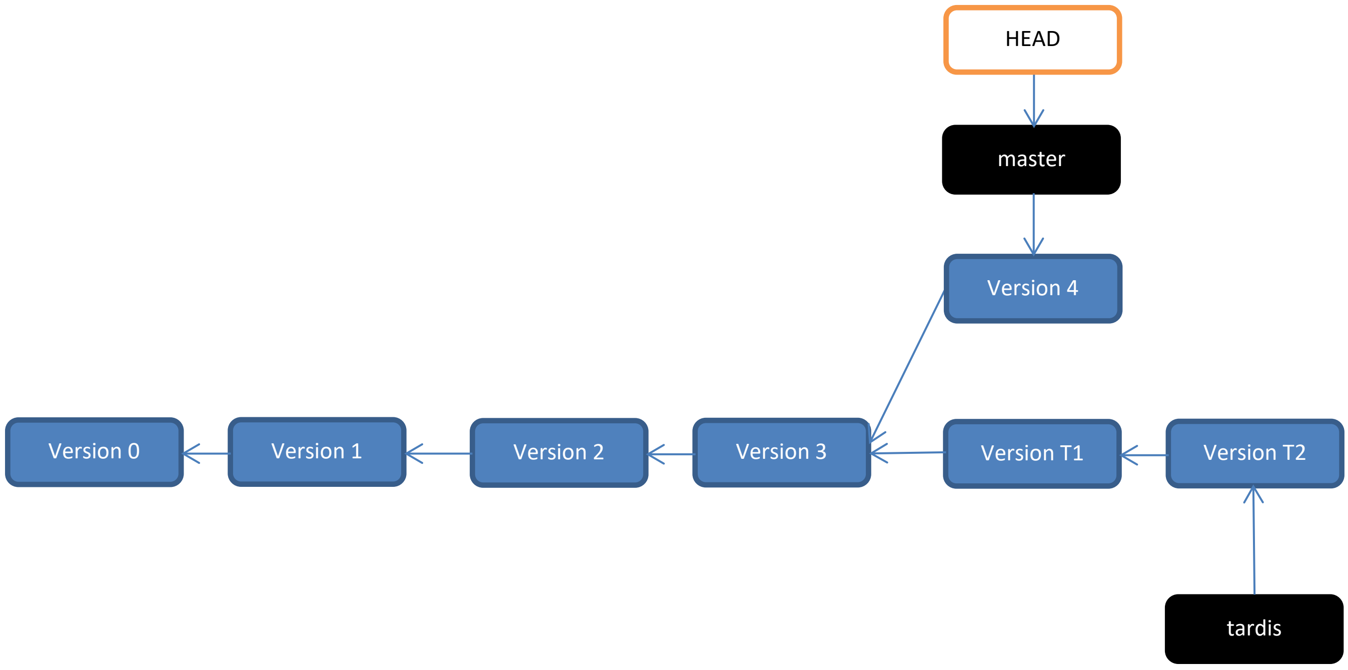
We edit README.txt, and delete the line:

```
This is version 1 of the file
```

Replacing it instead with a line as follows:

```
This is a post tardis update on the master :)
```

We undertake the usual add and commit operations, leaving us with the repo as shown:



A simple examination of the files shows for the master:

```
This is version 0 of the file
This is a post tardis update on the master :)
This is version 2 of the file
This is version 3 of the file
```

And similarly, for the tardis:

```
This is version 0 of the file
This is version 1 of the file
TARDIS 2
TARDIS 1
```

We now look to try a merge. We are on the master already, so we do not have to switch back. The command is straightforward, but the overlap kills us:

```
hogan@SEF-EEC-056940 /c/Data/week4 (master)
$ git merge tardis
Auto-merging README.txt
CONFLICT (content): Merge conflict in README.txt
Automatic merge failed; fix conflicts and then commit the result.

$ git status
On branch master
```

```
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

        both modified:      README.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

The system creates a partial merge, including the two versions in the one file while we proceed to resolve the problems on the temporary branch. The HEAD tag indicates the current branch, with the competing version shown underneath:

```
hogan@SEF-EEC-056940 /c/Data/week4 (master|MERGING)
$ cat READ*
This is version 0 of the file
<<<<<<< HEAD
This is a post tardis update on the master :)
This is version 2 of the file
This is version 3 of the file
=======
This is version 1 of the file
TARDIS 2
TARDIS 1
>>>>>>> tardis
```

We have no option but to fix the task manually and make the change:

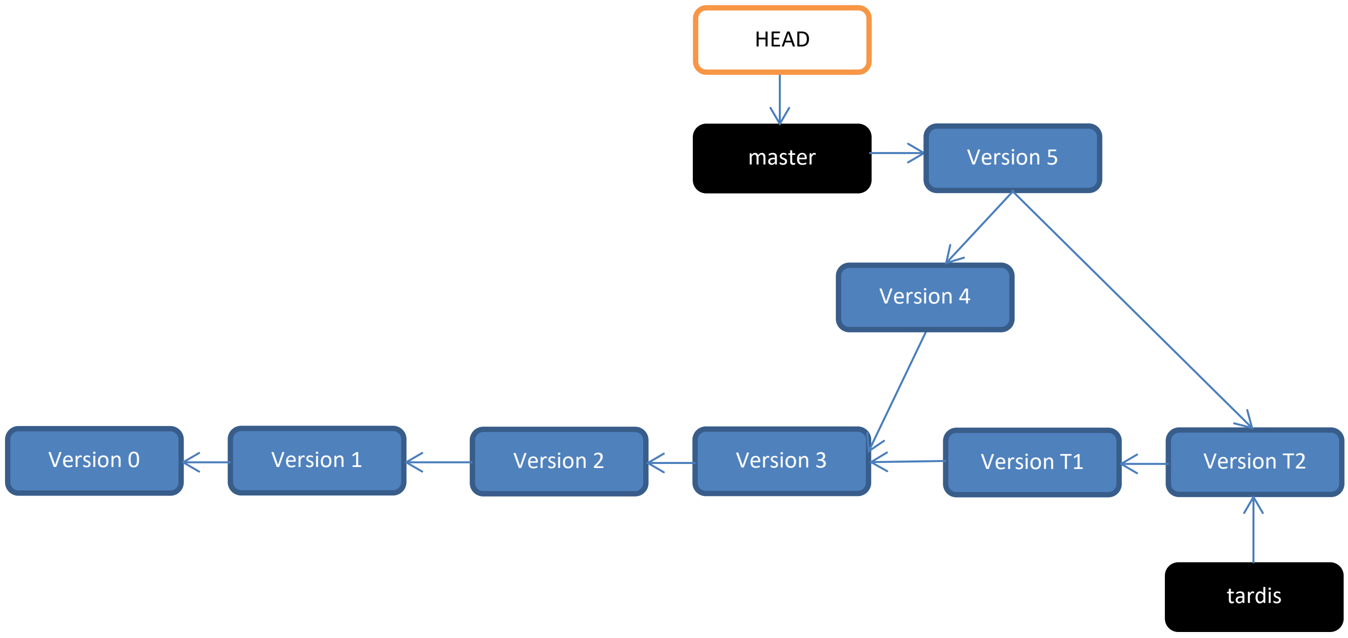
```
This is version 0 of the file
This is a post tardis update on the master :)
TARDIS 2
TARDIS 1
```

We then add the file to mark it as resolved, and commit to incorporate the merged file:

```
$ git add README.txt
$ git commit -m "Version 5: Tardis merged"
```

With the result as shown below:

```
[master 9bf1717] Version 5: Tardis merged
```



The branch tardis still exists, but is now somewhat unwanted. We can remove the branch simply, by using the command:

```
$ git branch –d tardis
```

This works, as the branch is merged. (Use `–D` if not). The result has the form:
`Deleted branch tardis (was 495431d).`

For a deeper understanding of these issues, see the discussion of branching workflows in section 3.4 of the git book. http://git-scm.com/book/en/Git-Branching-Branching-Workflows The earlier section, 3.3, gives a mix of simple and complicated examples of branch creation and merging.

In assignment 2, I doubt that you will end up using a vast number of branches. The most likely approach is that you will maintain a `master` and a `development` branch. The master will contain the stable, largely correct versions of the code, and you will commit your work more commonly on the development branch. When you are comfortable with this code, perhaps after a number of commits, you will then merge onto the master and then diverge again.

A Simpler Merge:
Having now completed the merge, create a new branch called simple:

```
$ git branch simple
```

We are now going to create two new versions of README.txt which can clearly be resolved by the merge command. The difference from the earlier case is that we will maintain a common line in the middle of the file, here in **bold**. Change to the simple branch (via checkout) and make the following edits.

Edit for simple:

```
This is version 0 of the file
This is an upper introduction
This is a post tardis update on the master :)
TARDIS 2
TARDIS 1
```

As usual, stage the file and commit. Now change to the master branch and do a similar, but different edit:

```
This is version 0 of the file
This is a post tardis update on the master :)
This is a lower introduction
TARDIS 2
TARDIS 1
```

Once again, stage the file and commit, and then try a merge, which this time happens automatically:

```
hogan@SEF-EEC-056940 /c/Data/week4 (master)
$ git merge simple
Auto-merging README.txt
Merge made by the 'recursive' strategy.
 README.txt | 1 +
 1 file changed, 1 insertion(+)
hogan@SEF-EEC-056940 /c/Data/week4 (master)
$ cat README.txt
This is version 0 of the file
This is an upper introduction
This is a post tardis update on the master :)
This is a lower introduction
TARDIS 2
TARDIS 1
```

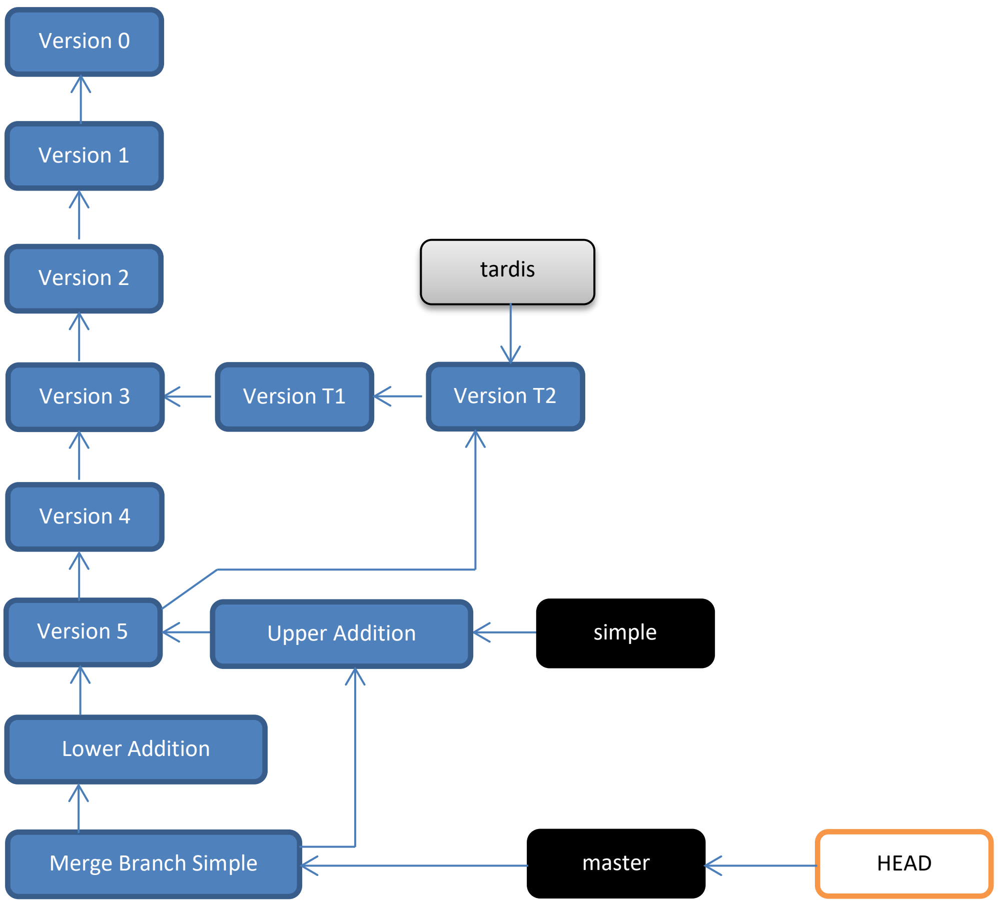
The difference of course is that the system is able to ‘split’ the file around a common centre. There are changes above, and changes below, but the changes do not take place in an overlapping fragment of code, and so Git can compare *both* new versions to their common ancestor, the README.txt created when we merged the tardis branch back on to the master in the previous exercise.

Note you may arrive a screen that asks you to enter a message via the default edition(vi). You can just use the default message by saving/writing (type in :w and press enter) and then quit (:q and enter).

Reverting Changes:

In the previous example we went back in time in a very manual way – actually editing the files and creating a separate branch. But sometimes we want have really made a mess of things, and so want to retreat completely to a previous “Fortress of Working Code”.

Now, if you have followed the exercise through in its entirety, you should have a commit history that looks something like this one:



Here of course we know that the tardis branch has been deleted and this is shown as a greyed out pointer. We are now going to assume that the last merge commit is something we want to avoid. We will go back twice. We will first get rid of the merge, and then we will return to Version 5. In each case, we will stay with the master.

Going to the log, we look up the hash for the last commit and execute the following:

```
$ git revert -m 1 b61a132244d86cf35025c8705658fc6c45a3a514
```

Note that the hash here refers to the merge commit, and NOT the commit to which you are reverting. As mentioned above, you may use a shortened version of the hash – 5 or 6 characters will usually be quite enough. The reversion specifies that we are getting rid of a merge (-m) and that we are staying with the primary branch version (1). The remaining argument is the hash. We are then offered the chance to change the commit message:

```
Revert "Merge branch 'simple'"

This reverts commit b61a132244d86cf35025c8705658fc6c45a3a514,
reversing changes made to bc6eb2fd785016f7bd05c4d6ec42dd68f9562b1c.

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#       modified:   README.txt
#
```

Save and quit from vi as before. Having stuck with this default message, we see the response at the end of the command:

```
$ git revert -m 1 b61a132244d86cf35025c8705658fc6c45a3a514
[master f498a1f] Revert "Merge branch 'simple'"
 1 file changed, 1 deletion(-)
```

And examination of the README shows that we are at the point “Lower Addition”:

```
$ cat README.txt
This is version 0 of the file
This is a post tardis update on the master :)
This is a lower introduction
TARDIS 2
TARDIS 1
```

Finally, we will revert directly to the previous commit, Version 5. As before, we will need to look up the hash from the log. But first, we will do an edit to the README.txt to show how we can keep some uncommitted changes. Edit this file now to look like:

```
This is version 0 of the file
This is a post tardis update on the master :)
This is a lower introduction
This is a post reversion edit
TARDIS 2
TARDIS 1
```

If you like, do a git status to show that the file is modified but uncommitted. Performing the hard reversion command at this stage would clobber the uncommitted changes. Fortunately, git lets us make a temporary copy:

```
$ git stash
Saved working directory and index state WIP on master: f498a1f Revert
"Merge branch 'simple'"
HEAD is now at f498a1f Revert "Merge branch 'simple'"
```

We can now revert to Version 5 as we wished to, using a hard reset:

```
$ git reset --hard 9bf1717e9f80d5d4f90111cfb7cbf8a2985d9f32
HEAD is now at 9bf1717 Version 5: Tardis merged
```

A quick look at the README file yields:

```
$ cat README.txt
This is version 0 of the file
This is a post tardis update on the master :)
TARDIS 2
TARDIS 1
```

Which is consistent with the older version of the file displayed earlier in this guide. To use the changed version of the file, we have only to use the stash command. This comes in two flavours: `git stash pop` and `git stash apply`. The difference is that pop discards the stash, apply keeps it around until you use a `git stash drop`. Here we will go ahead and use a `git stash pop`, which this time means there is a conflict to be resolved:

```
$ git stash pop
Auto-merging README.txt
CONFLICT (content): Merge conflict in README.txt
```

As before, we have to handle it manually:

```
This is version 0 of the file
This is a post tardis update on the master :)
<<<<<<< Updated upstream
=======
This is a lower introduction
This is a post reversion edit
>>>>>>> Stashed changes
TARDIS 2
TARDIS 1
```

After editing, we decide to keep everything:

```
This is version 0 of the file
This is a post tardis update on the master :)
This is a lower introduction
This is a post reversion edit
TARDIS 2
TARDIS 1
```

And after adding and committing the changes, we have our final version “The End”.

```
$ git commit -m "The End"
[master bdce384] The End
 1 file changed, 2 insertions(+)
```

These approaches are considered in gory detail in an excellent exchange on Stack Overflow [See the winning answer - with 9469 votes at the time of writing]:
http://stackoverflow.com/questions/4114095/revert-to-previous-git-commit

Remote Repository:
It is recommended that you use a remote repository as well as your local repository for 1) backup and 2) collaboration with teammates. Here are some steps that will allow you to upload your repository to GitHub – a well known cloud based repository service that is free. Another popular choice is BitBucket – consider checking them both out.

Step 1: Creating a GitHub Account

- Go to the GitHub website and follow the steps to create your own account.
- https://github.com/

Step 2. Create a new repository

- On the GitHub website click the plus sign in the upper right corner and choose New Repository
- Add a suitable name into the Repository Name text field
- Make sure that the option to have a private repository is clicked.*
- Click “Create Repository”

Step 3. Add files to the repository

- Go back to your Git bash and type in:

- `git remote add origin https://username@github.com/username/reponame.git` (where username is the user name you chose in step 1 and reponame is the name you chose in step 2)
- `git push origin master`

Step 4. Check the remote repository

- Go back to your repository in your browser and make sure that the files have been uploaded correctly.

*When you work on your assignment, if you are using a remote repository (which is highly recommended) you are **required** to ensure that it is private and that nobody outside your team is granted access. GitHub free accounts have a limited number of contributors on private accounts, so you might find consider applying for a GitHub Student Developer Pack which comes with free GitHub Pro access while you are a student: https://education.github.com/pack

Quick Reference Guide and Cheatsheet:

http://git-scm.com/docs