

Informatics 1: Object Oriented Programming

Tutorial 09

Self-Study: Version Control

Volker Seeker (volker.seeker@ed.ac.uk)

Vidminas Vizgirda (s1750767@ed.ac.uk)

1 Introduction

The teaching semester is finally almost over, and hopefully everyone is excited about the upcoming holiday. If the end of the semester has been harsh for you with deadlines or revision, then at least you are almost there, there's not much more left!



Figure 1: Skateboarding dog wishes you the best of luck with exams! “Skateboard Dog, Pet Expo, California” by lora_313 is licensed under CC BY 2.0.

In your programming journey you will have likely encountered words like Git, Subversion or Mercurial (no worries if you haven't yet - this is exactly what this tutorial is about). Perhaps you might have even used them before - if so, you can skip the introductory exercises and jump right into the fun part (literally, the section called “the fun part”).

All these tools (Git, SVN and others) are version control systems, sometimes also called revision control or source code management. Even if you aren't familiar with the concept yet, you have almost surely employed your own version control system already at some point. Does something like this seem familiar?

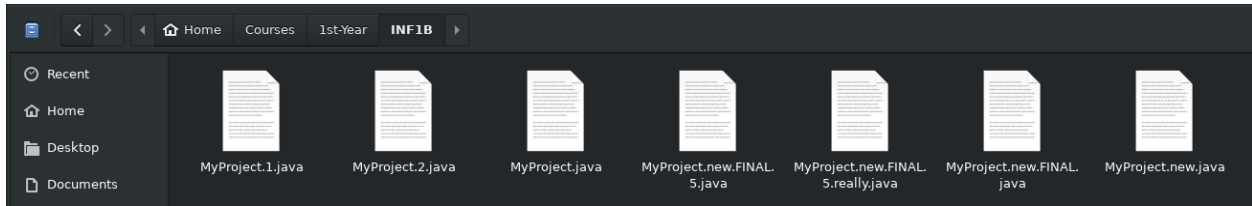


Figure 2: Here is an example of manual version control

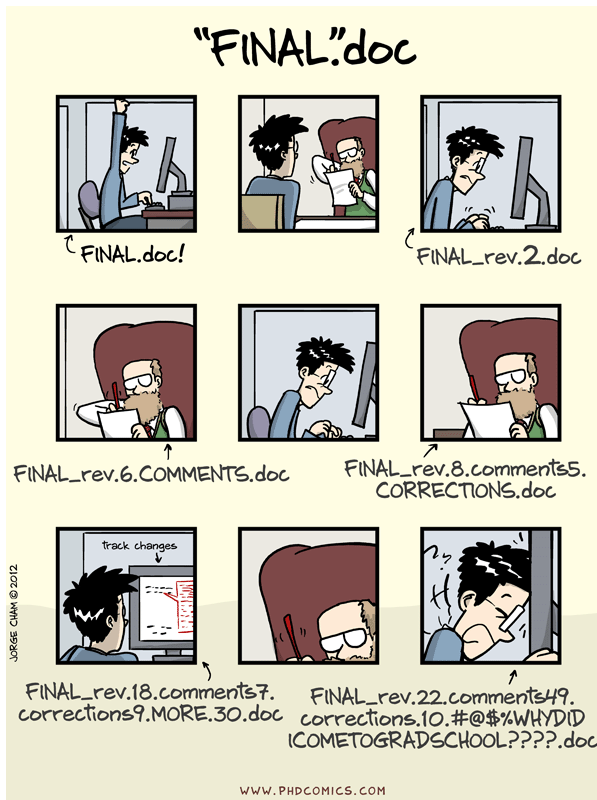


Figure 3: This situation is revision control too! “Final.doc” from “Piled Higher and Deeper” (PHD Comics) is by Jorge Cham. © 1997-2008 Jorge Cham

Why on earth would you do something like this? Well, to mention some reasons, having previous versions of your files allows to see how they have developed over time or go back to previous versions to recover something that was useful.

Formal version control does exactly that, but automatically! Even better, it has more features and most importantly, it allows multiple people to work on the same projects without (too much) hassle! We can:

- tag code versions for later reference (via tags).
- record a unique identifier for the exact code version used to produce a particular plot or result (via commit identifiers).
- roll back our code to previous states (via check-out).
- identify when/how bugs were introduced (via diff/blame).
- keep multiple versions of the same code in sync with each other (via branches/merging).
- efficiently share code and collaborate with others (via remotes/online hosting).

See more at <http://smutch.github.io/VersionControlTutorial/index.html>.

Many of the advantages of version control are not limited to just managing code. For example, it can also be useful when writing papers. Here we can use version control to:

- bring back that paragraph we accidentally deleted last week.
- try out a different structure and simply disregard it if we don't like it.
- concurrently work on a paper with a collaborator and then automatically merge all of our changes together.

In this tutorial, we will be learning how to use a version control system called Git, and then using it to create a team project around finding cat videos on YouTube.

2 Name's Git, good to meet ya, partner

Why Git? For one, it is one of the most widely used modern version control systems in the world, which means there are tons of helpful documentation, tutorials and Q&A pages online. For bonus points, many IDEs are integrated with Git (e.g. IntelliJ, Visual Studio, VS Code, and many more) and GUI tools to help with using Git (e.g. GitKraken or SourceTree).

We won't be using these tools in this tutorial, because for simple applications, the command line is just as good (in fact, for very complex things, it is often more useful than graphical interfaces too, but we won't need that just yet) and you won't need to install all the extras.

Why is Git called Git? There's some funny backstory, which you read about at <https://en.wikipedia.org/wiki/Git#Naming>

Anyways, to proceed beyond, you will need to have Git installed. If you haven't done this yet, you can *git* it following this neat tutorial: <https://www.atlassian.com/git/tutorials/install-git>

Task 1 - Setting up your first repository

◀ Task

If you've now got Git, you can always check if it works correctly by opening terminal and running `git --version`, which on DICE would output something like:

```
1 $ git --version
2 git version 1.8.3.1
```

Hopefully on your computer you will have a newer version, since Git 1.8 has been unsupported for a long time already. If running the command gives an error, try troubleshooting by going through the installation tutorial above - perhaps you missed a step? The tutors are also here to help you.

Once Git works, you will need to configure it (don't worry, you only have to do this once) by running (making the obvious substitutions):

```
1 $ git config --global user.name "The ROCK"
2 $ git config --global user.email rock@legend.ed.ac.uk
```

Next you need to tell Git what editor you want to use when Git needs you to type something:

```
1 $ git config --global core.editor vim
```

You should replace `vim` with whatever your favorite editor is (e.g. `notepad`, `emacs`, `nano`, `subl`, etc.).

You can also make things a little easier on the eyes by telling Git to add some color to its messages by running:

```
1 $ git config --global color.ui true
```

With Git set up and ready, create a directory for a temporary project, navigate to it and run `git init` to create a new repository versioned by Git (a repository is the collection of all your files - code, documentation, cat pictures, whatever it may be). This might look like this (for example):

```
1 $ cd ~/Courses/1st-Year/INF1B/
2 $ mkdir git-good
3 $ cd git-good
4 $ git init
```

```
5 Initialized empty Git repository in ~/Courses/1st-Year/INF1B/git-good/.git/
```

Now if you check inside your project directory, there will be a new hidden ".git" sub-directory with all sorts of configuration files (which you don't need to touch). This has not added anything to your project yet, but it means that it will be possible to use Git to manage all files in this directory from now on.

Task 2 - Your new best friend - "git status"

< Task

Like `ls` and `pwd` are helpful if you get lost when navigating directories, `git status` is possibly the most useful command for getting a view of what's going on with Git.

This command gives you a summary of what has happened since the last revision in your repository. If you run it in the empty repository you just created, it should say something like this:

```
1 $ git status
2 # On branch master
3 #
4 # Initial commit
5 #
6 nothing to commit (create/copy files and use "git add" to track)
```

This message mentions branches. If you look at the image on the right - this is a tree structure showing directories and files contained within them. If you were manually versioning an article, you might create separate directories like "final_proof" and "submitted" to categorise your article versions. Branching is like these directories - they categorise versions.

The master branch is a default branch that Git starts you with, but you can create others if you like.

It also talks about commits - these are analogous to versions or revisions. Each commit is a saved state of your repository (project).

Especially while still getting used to Git, you might want to run "git status" in between every command just to see what has happened in the repository from Git's point of view, it can help to understand how things work.

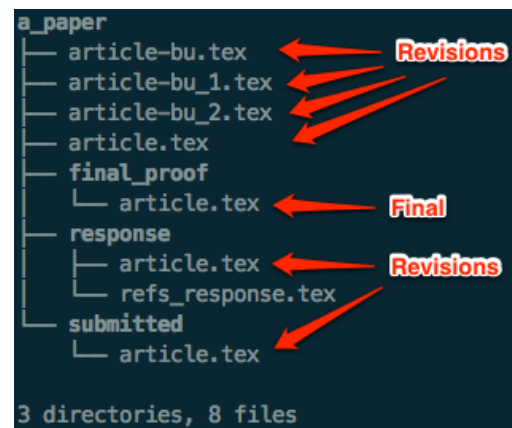


Figure 4: Aren't you glad you don't need to write a dissertation anytime soon yet? Image by Simon Mutch is available under the [CC BY-SA 3.0 Unported](https://creativecommons.org/licenses/by-sa/3.0/) licence.

Task 3 - Making *commitments*

< Task

Try creating a new file and running `git status`. What happens? An easy way to create a new file from terminal is using the `echo` command, e.g. like this:

```
1 $ echo "\"10 years a slave\" was a really good film" > favourites.txt
```

Run `git status` and you'll see our text file is now an "untracked file" - sounds a lot more exciting than a five word file really is! Untracked just means that Git does not care about the contents of this file at the moment.

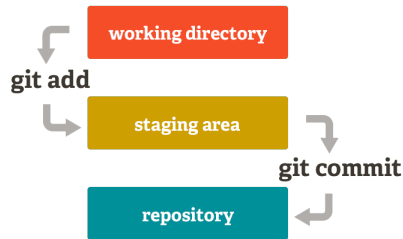


Figure 5: Git commit stages¹

But it's Git's job to care! To change this, let's add the file to the staging area. The staging area is a collection of files with changes that are tracked by git.

To do this, use `git add favourites.txt`

Run your best friend again (`git status`, remember?) and you'll see our file has entered the second stage of the pipeline - it's a "change to be committed". (If you're curious how to remove files from the staging area, you can use `git reset HEAD favourites.txt`)

Now you can commit your changes using `git commit -m 'your_commit_message'`. One way to think about this is that your code is like a messy room that you are tidying up. The staging area is a box for changes, you can place things in this box using `git add`. Finally, you can seal the box and put a label on it using `git commit`, which will make it easier to find things later on.

Commit messages are optional, but they're an easy way of telling your other collaborators (or yourself in the future) what you've done for the project with a particular set of changes. It's customary to use an imperative mood to write commit messages - instead of 'added example.txt', a typical commit message would be 'add example.txt'. Think of it as you telling git to do what you want it to.

A complete example might look like this:

```

1 $ echo "\"10 years a slave\" was a really good film" > favourites.txt
2 $ git add favourites.txt
3 $ git commit -m "Add my favourite films file"
4 [master (root-commit) 019c676] Add my favourite films file
5 1 file changed, 1 insertion(+)
6 create mode 100644 favourites.txt

```

	COMMENT	DATE
○	CREATED MAIN LOOP & TIMING CONTROL	14 HOURS AGO
○	ENABLED CONFIG FILE PARSING	9 HOURS AGO
○	MISC BUGFIXES	5 HOURS AGO
○	CODE ADDITIONS/EDITS	4 HOURS AGO
○	MORE CODE	4 HOURS AGO
○	HERE HAVE CODE	4 HOURS AGO
○	AAAAAAA	3 HOURS AGO
○	ADKFJSLKDFJSOKLFJ	3 HOURS AGO
○	MY HANDS ARE TYPING WORDS	2 HOURS AGO
○	HAAAAAAAAANDS	2 HOURS AGO

AS A PROJECT DRAGS ON, MY GIT COMMIT MESSAGES GET LESS AND LESS INFORMATIVE.

Figure 6: XKCD #1296: "Git commit" is licensed under CC BY-NC 2.5

Congratulations! Now you know the basics of working with Git. There is a lot more to it than just making commits and adding files, but in many cases this is all you will need.

¹Image source: <https://medium.com/@lucasmaurer/git-gud-the-working-tree-staging-area-and-local-repo-a1f0f4822018>. Original author unknown.

3 The fun part

Since it's already near the end of the semester and you have developed lots of skills throughout, let's put them to good use by creating a project that uses an external API. API stands for Application Programming Interface - in a nutshell, it's the specification for how you can use a program in other programs.

You can read more about what APIs are and what they're good for here: <https://readwrite.com/2013/09/19/api-defined/>

In this project, we will be using the YouTube API to find interesting YouTube videos for people based on their interests.

Since this tutorial is all about version control, let's get into teams of 2-3 pairs of people (who may be sitting with you at your desk, or from around the whole room, especially if you skipped to this section), and develop it collaboratively.

The project is bigger than anything we have had in the previous tutorials, so don't worry if your team can't finish it in the tutorial. Try to get as far as you can, and you can always experiment with it later on - nothing beats practice when it comes to getting good at programming.

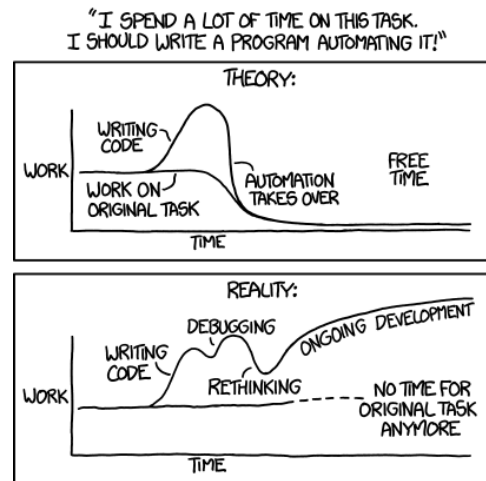


Figure 7: XKCD #1319: "Automation" is licensed under [CC BY-NC 2.5](https://creativecommons.org/licenses/by-nc/2.5/)

Task 1 - The specification


◀ Task

The task in this project is to create a program that gives YouTube video suggestions for someone, given their interests. For simplicity, let's represent interests by standardised tags, e.g. "functional programming", "funny cats", "backflip", "rainbow lipstick".

How can we find someone's interests? Some information about their past can help. For example, since this project is about suggesting videos, what better way is there of finding what people like to watch than seeing what they have watched previously?

For this project you are given an input file with user names and a list of videos from their viewing history of the last 2 days (it's all completely fictional, although in the real world, YouTube, Netflix and many others actually have something similar), formatted like this:

```
1 name: Alice Webster
2 history: video1.mp4, video2.mp4
3 name: Stephen King
4 history: video4.mp4
```

 You can download the videos themselves from the Inf1B course materials page: <https://course.inf.ed.ac.uk/inf1b>

You will need to create an associative array linking videos to topics by watching them and coming up with some tags to describe the themes in them manually. For example, if video1.mp4 was showing a birthday party with lots of kids running around with balloons, tags like ["birthday", "balloons", "fun"] might be appropriate, or if video2.mp4 was an extract from a wildlife documentary, ["nature", "documentary", "animal"] might fit. Something like ["wildlife documentary about nature"] might be not so good, long tags are more precise but it becomes harder to find similar to them.

With tags like these, you can merge them to get a particular user's interests, for example, since Alice Webster

watched both video1.mp4 and video2.mp4, her interests might be ["birthday", "balloons", "fun", "nature", "documentary", "animal"].

Now, you can find some search results for these terms using the YouTube data API and recommend these videos to Alice.

The expected output is a file with recommended video names and URLs, in this format:

```
1 name: Alice Webster
2 videos: ["Metallica - One [Official Music Video]", https://www.youtube.com/watch?v=WM8bTdBs-cw],
          ["Mein Lieblingsspiel 2019: Trant | Game Two Adventskalender #10",
           https://www.youtube.com/watch?v=ifc0wZyJpoA]
3 name: Stephen King
4 videos: ["CITIZENFOUR - Official Trailer", https://www.youtube.com/watch?v=rHaWhUjV96M]
```

Task 2 - Splitting up teamwork

◀ Task

This project is quite large, but you also have your teammates by your side to tackle it. It's a good opportunity to think about how will you all work on this project at the same time. This is up to your team, but a suggestion would be to split it into modular tasks:

- Identifying topics in the provided videos
- Parsing the input file
- Linking each user to their interests
- Finding video suggestions
- Writing the output file

These are mostly independent steps - you don't need to have parsed an input file to get started with looking for video suggestions with the YouTube API. Instead, you can leverage interfaces to help you write your code.

For example, to separate parsing the input file and linking users to interests, you can start by creating an interface like this:

```
1 interface InputParser {
2     String readFileIntoString(String filePath);
3     void parseUsersAndHistory(String input);
4     List<String> getUserHistory(String user);
5 }
```

Now the teammates working on the file parser can create a class that implements the methods of this interface.

```
1 class FileInputParser implements InputParser { //...
```

And the teammates working on linking interests can create a function that takes an InputParser instance and use its methods, without needing them to be implemented just yet:

```
1 class InterestLinker {
2     List<String> getUserInterests(InputParser parser, String user) {
3         // Assuming file has been read and parsed before
4         List<String> userVideos = parser.getUserHistory(user);
5         // ...
}
```

Finally, when everything is ready, you can create concrete class instances in a main method and call all the appropriate functions to put it all together.

When it comes to finding video suggestions, there is a lot of scope for your own exploration - the tags are subjective, it is up to you to come with something that fits well. Once you have a user's interests, you can find a video for each interest, or find one that matches as many interests as possible, or something in between.

Task 3 - Team git repo setup

◀ Task

You will need to choose one person to create an online project repository and add everyone else as collaborators. It would be best if this person was someone who has used GitHub, GitLab, BitBucket (or another remote repository hosting service) before for this role. If no one on your team has such experience, then everyone should go through this tutorial first <https://lab.github.com/githubtraining/introduction-to-github>.

Once you've got a repository set up online, everyone in the team will need to clone it, using `git clone`.

One person (or one pair) should set up a new Gradle project with IntelliJ in the project repository, commit it and use `git push` to push the changes online.

Then everyone else in the team should run `git pull` to get the skeleton project on their machines.

To avoid lots of hassle in merging changes, before you start, it is a good idea for each person (or pair) to create a new branch for themselves using `git branch branch-name` (just make sure that everyone uses different branch names). You can then run `git branch` to check whether it worked. For example, this might look like:

```
1 $ git branch legend
2 $ git branch
3 * master
4   legend
```

This created a branch called `legend` alongside the already existing `master` branch. The asterisk indicates that we are currently still working on the `master` branch.

To switch to another branch, use `git branch branch-name`, e.g.:

```
1 $ git checkout legend
2 Switched to branch 'legend'
3 $ git branch
4   master
5 * legend
```

For this exercise, it will be easiest if everyone works in their own branches. Then, at the end, all changes can be merged online using pull requests.

Task 4 - Go go go

◀ Task

Some helpful links to help you get started:

- Parsing input files - <https://javabeat.net/parsing-input-using-scanner/>
- Parsing input files - <https://stackoverflow.com/questions/13198393/how-to-parse-an-input-file>
- YouTube data API documentation - <https://github.com/googleapis/google-api-java-client->

`services/tree/master/clients/google-api-services-youtube/v3`

- YouTube Java API start guide - <https://developers.google.com/youtube/v3/quickstart/java>
- YouTube Search API - <https://developers.google.com/youtube/v3/docs/search/list>
- Writing output files - <https://www.baeldung.com/java-write-to-file>

Don't worry if you get stuck, you can always ask your teammates, and if no one knows what to do, then the tutors for help!

Hint for the YouTube API: you don't need an OAuth signature, since we are only looking at publicly available videos. An API key is enough.

4 Further learning

If you found the topics of version control interesting, you can learn a lot more about it by taking "INF2C-SE: Introduction to Software Engineering". You can learn a lot more about Git from the links listed here: <http://try.github.io/>

If using the YouTube APIs was fun, you could learn about similar work by taking on practical courses (e.g. Informatics Large Practical, Compiling Techniques, ...)

Finally, you will find out a lot more about making clever recommendations in "INF2 - Foundations of Data Science" and any future machine learning courses. If you are up for it, you can try a project at home yourself, using the [Google Cloud Video Intelligence API](#) to detect tags describing videos automatically.

Shout-out to InfPALS for creating the content that was borrowed from their Introduction to Git workshop.

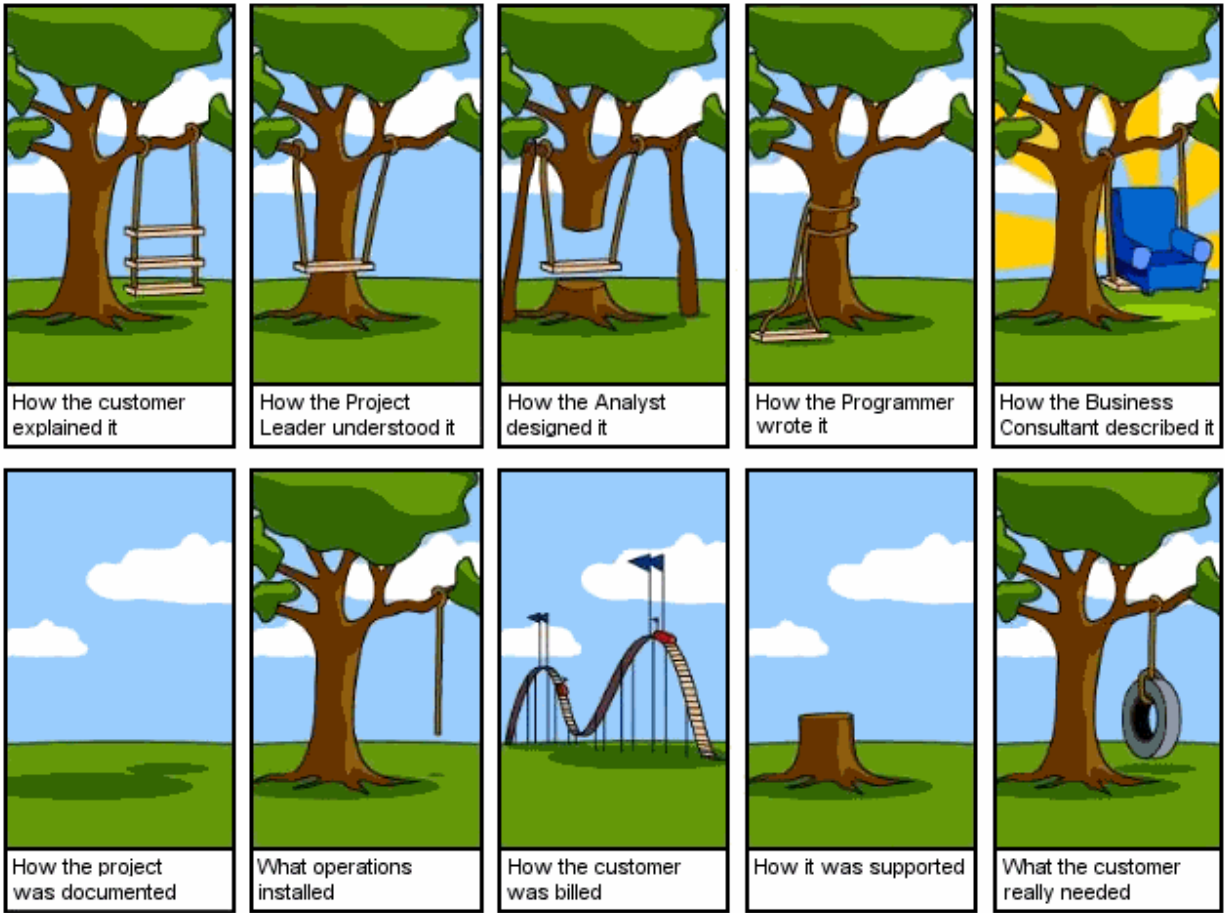


Figure 8: Image from [Coding Horror blog](#). Originally by [Alex Gorbatchev](#) (source no longer available)