EEEE2076 - Software Development Group Design Project

Worksheet 2 - Collaborative Software Development with Git

P Evans

February 5, 2026

Contents

1 Collaborating with Git and GitHub

1.1 Sign up for the Github student pack

Github is an online service that you will use to store the code that you write. Before you read any further, sign up for the Github student pack (https://education.github.com/pack). It takes some time to process, so better to do it now. You can use Github without signing up for the student pack but you will run into difficulties later - you won't have access to all features and any code repositories you create will be public (visible by anyone online).

1.2 Problems

Naturally, code changes throughout the development process. Hopefully these changes will result in features being added and bugs being fixed but sometimes fixes to one problem can introduce issues elsewhere. These new issues can remain hidden for a while and only come to light weeks later, this can make understanding when and how bugs were introduced difficult.

All of this makes creating software a difficult exercise and sometimes there is no better solution than going back in time and inspect the code at the last working version. Even though most modern operating systems offer general, automated backup solutions, these solutions are not necessarily the best for source code management: they give very little information when it comes to the actual content of the files, they do not allow quick retrieval of old versions, and they do not store any meta-data that can give further insight in how the code base evolved over time.

Neither do these OS supplied systems provide branching, i.e. the ability to maintain and track several parallel versions of the code base (one for the next release, one for the latest bugfixes, one for an experimental feature, one for a paying customer who desired bespoke modifications,. . .). It goes without saying that an OS supplied backup system is not portable (not accessible by other users who may be running different operating systems / operating system versions) and thus cannot be used within the terms of reference of our project.

A related problem is that of collaboration. Indeed, assuming that you cannot remember all the modifications you made over time (and you cannot remember all modifications you made over time), working with several people is very similar to working with multiple versions of the software. In this case the goal is of course to, on a very regular basis, merge all existing versions into a single working copy that benefits from all the code you and your colleagues have been producing. No backup system I know about provides any help in this regard.

1.3 The solution

To solve the above problem we will use a tool called git to keep track of all code you will develop for this project. Git is a so-called code versioning system. Other systems are around (such as svn and mercurial). We will stick with git because it is probably the most widely used and something you might already have heard of or be familiar with.

Before you go any further, find Git for Windows online and install it.

Git manages what are known as repositories. From a user point of view, these are simply directories containing code and their history / different versions. Git itself is a command line tool, but GUI interface tools for the various OSs exist. I strongly advise to use the CLI (command line interface) tool to begin with, because some of these GUI tools are quite pro-active in guessing what you mean, even if you didn't mean what they guessed. Its also very useful to know how to use the native Git interface so you know what the GUI interfaces are actually doing if you choose to use them later.

After installing git somewhere on your system you can invoke it as any other programme. That means that you can either supply the (usually very long) complete path to the executable, or you can add it to the system

path. it is a good idea to put the path to git.exe (on Windows) in the system path environment variable. Git is already installed on all of the VR PCs and has been added to the path.

Don't put repos on a network drive or in a OneDrive/Dropbox folder. The reason is that git uses a lot of small files to keep track of history and the constant updating of these files can generate a lot of network traffic, and the OneDrive/Dropbox update programs can sometimes conflict with Git (they can block access to the file while Git is trying to update something). You do not need the backup feature of these drives when using Git - all your work will be saved in the cloud anyway.

2 Creating a Repository

Create a subdirectory that will contain your new repository. Use *2025_STUDENTID* as your repository name so I can easily identify it later, a repository called 'work' by username 'Smurf123' causes me problems! The 2025 (rather than 2026) is because its the 25/26 academic year. This repository will become your individual repository where you can create sub-directories for the worked examples you will be asked to complete in these worksheets. You should use this repository for all individual worksheet exercises.

We now create a repository by:

```
C:\Users\ezzpe>mkdir 2025_ezzpe

C:\Users\ezzpe>cd 2025_ezzpe

C:\Users\ezzpe\2025_ezzpe>git init
Initialized empty Git repository in C:/Users/ezzpe/2025_ezzpe/.git/

C:\Users\ezzpe\2025_ezzpe>
```

A few words of caution:

- When you type `git init`, git creates a hidden folder called .git that it uses to store the history of the your repository. Do not modify this folder.
- Do not create a new git repository inside an existing repository. For example: you create a repository and then decide it has the wrong name, then create a repository with the correct name inside the old, wrong name, repository. This will cause a lot of problems and Git to behave erratically.
- Be very careful copying files between git repositories in the filesystem (i.e. copy and paste using Windows explorer). If you copy and paste the .git folder from one repository to another it will cause you problems (this is Git's database for the code, you will copy the history for other code into the new repository). Or if you try an copy the contents of a repository from one place to another and don't copy the hidden .git folder you'll break the repository because the history is lost. It is best to create the repository in one place and leave it there. If you need another copy, clone a copy from Github (see instructions later).

Now create a text file 'README.md' using notepad. Place some text in this file: your name is a good idea to its easier to see who the respository belongs to. You now need to tell git to track the file we just created. This is done by:

```
C:\Users\ezzpe\2025_ezzpe>echo Hello > README.md

C:\Users\ezzpe\2025_ezzpe>git add README.md

C:\Users\ezzpe\2025_ezzpe>git status
On branch master

```

```
No commits yet

Changes to be committed:
(use "git rm --cached <file>..." to unstage)

new file:   README.md


C:\Users\ezzpe\2025_ezzpe>
```

Here we used the git status command to get a summary of the current state of our development directory. This provides us with some important pieces of information:

- The branch we are working on, this can be thought of as the version of the code we are working on. More details later.
- A list of files that are modified but not staged to be committed.
- A list of files that are modified and whose modifications are staged to be committed.

This is something where a lot of novice git users get confused. Modifications first need to be staged before being committed committed. Staging means making Git aware of a list of changes that you would like to be saved. Committing means actually saving the changes into Git's database to create a permanent record.

Let's now actually commit the change:

```
C:\Users\ezzpe\2025_ezzpe>git commit -am "Initial Commit"
[master (root-commit) 9515bb2] Initial Commit
Committer: Paul Evans (staff) <paul.evans@nottingham.ac.uk>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly. Run the
following command and follow the instructions in your editor to edit
your configuration file:

git config --global --edit

After doing this, you may fix the identity used for this commit with:

git commit --amend --reset-author

1 file changed, 1 insertion(+)
create mode 100644 README.md

C:\Users\ezzpe\2025_ezzpe>git status
On branch master
nothing to commit, working tree clean

C:\Users\ezzpe\2025_ezzpe>
```

A commit needs an obligatory commit message. It is often challenging to enter something meaningful here, especially if you do a lot of commits (and you should do a lot of commits). Take some time to make sure the message is representative of what you where working on at the time. **I don't want to see all of your commits with the same 'Initial Commit' message at the end of the semester!**

Immediately after doing a commit, the status shows that no modifications are present in the current working tree, as expected. Let's follow this initial commit with another one.

Create a new file hi_and_date.cmd with a command line script that prints a welcome message and the current time and date (see the date command) (.cmd is an alternative file extension for a windows batch file, you need to write a batch file that displays the date). Change the test in README.md to "Read this very carefully". Add and commit the modifications. The git add command followed by a directory name add all modified files in that directory and all its subdirectories to the stage. In particular git add . adds all modifications.

After finishing this we can query our two commit history:

```
C:\Users\ezzpe\2025_ezzpe>git log
commit aa0880be9ba80d27a13f88c8681e21c9b6d1ad1b (HEAD -> master)
Author: Paul Evans (staff) <paul.evans@nottingham.ac.uk>
Date:   Thu Sep 8 10:18:55 2022 +0100

Added hi_and_date file

commit 9515bb2a31527112b778ee177ce83d247b53e27a
Author: Paul Evans (staff) <paul.evans@nottingham.ac.uk>
Date:   Thu Sep 8 10:13:53 2022 +0100

Initial Commit

C:\Users\ezzpe\2025_ezzpe>
```

We can retrieve the modifications that were applied at every commit using the show command. The argument for the show command is the commit you want to visit. All commits have a SHA hash code, this is a hexadecimal number that forms a unique fingerprint or signature that can be used to identify the commit. This is the long hexadecimal string you find next to commit in the output of the log command. Of course this is not a practical way of identifying the commit. Fortunately it suffices to enter the first couple of digits making up the SHA. As long as this uniquely determines the commit no further specification is required.

```
C:\Users\ezzpe\2025_ezzpe>git show aa088
commit aa0880be9ba80d27a13f88c8681e21c9b6d1ad1b (HEAD -> master)
Author: Paul Evans (staff) <paul.evans@nottingham.ac.uk>
Date:   Thu Sep 8 10:18:55 2022 +0100

Added hi_and_date file

diff --git a/hi_and_date.cmd b/hi_and_date.cmd
new file mode 100644
index 0000000..a8f9fd8
--- /dev/null
+++ b/hi_and_date.cmd
@@ -0,0 +1 @@
+Hello

C:\Users\ezzpe\2025_ezzpe>
```

As you can tell from this output only the modifications are reported, not the entire file content. The show command allows deep customisation, but by default you get the commit in so called patch format. The + and - signs in front of the lines indicate which lines are removed and which ones are added. A modified line simply is recorded as a combination of a line removed and a line added.

There are a number of short hands to indicate commits, called references or refs for short. In particular, HEAD always refers to the most recent commit on the current branch. Entering a new commit will automatically advance the HEAD reference to point to the correct location. The most recent commit on any branch can be referred to using the branch name. In our current one branch scenario the following commands are synonymous

(since aa088 is the ID of the previous commit, HEAD also points to the most recent commit, and master is the name of the default branch which is where the most recent commit was made):

```
>git show aa088
>git show HEAD
>git show master
```

The commit HEAD is pointing to just prior to submitting a new commit will be the parent of that commit. In general to refer to the parent of a commit, you can use the tilde notation. This means the following are synonymous:

```
>git show HEAD~
>git show HEAD~1
>git show master~1
>git show 9515b
```

3 Going back in time

In this section we will discuss four ways you can go back in time using git:

- Undoing a staging operation using git reset
- Disregarding changes in the working tree using checkout
- Checking out a previous version using checkout
- Undoing a commit using reset

Git is extremely conservative when it comes to overwriting you work. Most command will generate new commits or add to the history of your code development without actually overwriting data. From the four options above, only the second can lead to loss of data.

Git may seem very user unfriendly. It is true that the names of some commands do not really seem to make a lot of sense or are used for several task that at first sight do not seem very much related at all. Only when you learn more about the philosphy behind the design of git you will see that there is actually a lot of consistency.

3.1 Scenario 1: Undoing an add operation

At mentioned above, the whole reason there is a staging area is to allow you to submit multiple changes in a single commit. If you accidentally staged (add'ed) more changes than you originally were planning to commit you can undo these operations on a per file level as show in the following example. Note that git actually tells you what this command is as part of the output of git status.

```
C:\Users\ezzpe\2025_ezzpe>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:   README.md

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

C:\Users\ezzpe\2025_ezzpe>git add README.md

```

```
C:\Users\ezzpe\2025_ezzpe>git status
On branch master
Changes to be committed:
(use "git reset HEAD <file>..." to unstage)

modified:   README.md


C:\Users\ezzpe\2025_ezzpe>git reset HEAD README.md
Unstaged changes after reset:
M       README.md

C:\Users\ezzpe\2025_ezzpe>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:   README.md

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

C:\Users\ezzpe\2025_ezzpe>
```

Make changes to README.md Add (but don't commit) the file and undo the staging using the reset command. At this point you remove the changes from the staging area (incidentally, the staging area is sometimes also called the index). The changes you made however are still present in your working directory, as the output of git status will remind you of.

3.2 Scenario 2: Discarding changes made in the working tree

Discarding changes made in the working tree are undone by overwriting files with versions that were previously stored in the history managed by git. You can check out these versions using the git checkout command.

```
C:\Users\ezzpe\2025_ezzpe>git checkout -- README.md

C:\Users\ezzpe\2025_ezzpe>git status
On branch master
nothing to commit, working tree clean

C:\Users\ezzpe\2025_ezzpe>
```

This is one of a few instances where git overwrites your work. Use git checkout in conjunction with file names very carefully! Gone is Gone!

Run this command and check to see that the last change you made to Readme.md has been undone.

3.3 Scenario 3: Checking out a previous version using checkout

Related to the previous but more common (and safe) is to checkout a different version of your code base. This does not delete anything from the history. It simply places an older version in your working directory (HEAD then points to this previous version).

```
C:\Users\ezzpe\2025_ezzpe>git log
commit aa0880be9ba80d27a13f88c8681e21c9b6d1ad1b (HEAD -> master)
```

```
Author: Paul Evans (staff) <paul.evans@nottingham.ac.uk>
Date:   Thu Sep 8 10:18:55 2022 +0100

Added hi_and_date file

commit 9515bb2a31527112b778ee177ce83d247b53e27a
Author: Paul Evans (staff) <paul.evans@nottingham.ac.uk>
Date:   Thu Sep 8 10:13:53 2022 +0100

Initial Commit

C:\Users\ezzpe\2025_ezzpe>git checkout HEAD~1
Note: checking out 'HEAD~1'.

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 9515bb2 Initial Commit

C:\Users\ezzpe\2025_ezzpe>git log
commit 9515bb2a31527112b778ee177ce83d247b53e27a (HEAD)
Author: Paul Evans (staff) <paul.evans@nottingham.ac.uk>
Date:   Thu Sep 8 10:13:53 2022 +0100

Initial Commit

C:\Users\ezzpe\2025_ezzpe>git checkout master
Previous HEAD position was 9515bb2 Initial Commit
Switched to branch 'master'

C:\Users\ezzpe\2025_ezzpe>git log
commit aa0880be9ba80d27a13f88c8681e21c9b6d1ad1b (HEAD -> master)
Author: Paul Evans (staff) <paul.evans@nottingham.ac.uk>
Date:   Thu Sep 8 10:18:55 2022 +0100

Added hi_and_date file

commit 9515bb2a31527112b778ee177ce83d247b53e27a
Author: Paul Evans (staff) <paul.evans@nottingham.ac.uk>
Date:   Thu Sep 8 10:13:53 2022 +0100

Initial Commit

C:\Users\ezzpe\2025_ezzpe>
```

`git log` will only give you information about the commit history up to the commit you currently have checked out (i.e. the commit pointed to by HEAD). In particular this means that our history seems to contain only two commits, not three.

git checkout master resets HEAD so that it points to the most recent commit again.

3.4 Scenario 4: Undoing a commit using reset

The git reset command changes where HEAD and the current branch name (e.g. master) refer to. By resetting it to refer to an older commit, you effectively forget about the newer commit. Note that it does not get deleted from disk, it just disappeared from git's history books.

```
C:\Users\ezzpe\2025_ezzpe>git reset HEAD~1

C:\Users\ezzpe\2025_ezzpe>git log
commit 9515bb2a31527112b778ee177ce83d247b53e27a (HEAD -> master)
Author: Paul Evans (staff) <paul.evans@nottingham.ac.uk>
Date:   Thu Sep 8 10:13:53 2022 +0100

Initial Commit

C:\Users\ezzpe\2025_ezzpe>git reset head@{1}

C:\Users\ezzpe\2025_ezzpe>git log
commit aa0880be9ba80d27a13f88c8681e21c9b6d1ad1b (HEAD -> master)
Author: Paul Evans (staff) <paul.evans@nottingham.ac.uk>
Date:   Thu Sep 8 10:18:55 2022 +0100

Added hi_and_date file

commit 9515bb2a31527112b778ee177ce83d247b53e27a
Author: Paul Evans (staff) <paul.evans@nottingham.ac.uk>
Date:   Thu Sep 8 10:13:53 2022 +0100

Initial Commit

C:\Users\ezzpe\2025_ezzpe>
```

This will have HEAD and the branch it points at (master) point at the commit prior to master. Even though this technically does not remove the latest commit, it is a bit tricky to undo the reset operation. Immediately after the reset command it can be undone by:

```
C:\Users\ezzpe\2025_ezzpe> git reset HEAD@{1}
```

You should read this as: set HEAD (and the branch it points at) to wherever HEAD was pointing at before the last time it moved.

4 Branches

4.1 Background

A branch in git should be thought of as a separate version. So far, you have one branch - the 'master' branch.

You can have more than one branch, a common reason for this is if you want to try something a bit risky - you will modify the code to add a new feature but are aware that you might break the code. In this case you add second branch where you develop the new code, and new commits will be added to this branch leaving the 'master' branch containing the old, stable code for normal users to download.

You then might want to make changes to either the master branch (e.g. to make minor updates to the stable code on master), or the development branch (to work on the new development code).

At some point the development code will mature and you will have confidence that the new feature works properly, at this stage the new changes on the development branch can be merged into the master branch. Sometimes this merge can be problematic because updates made in the master branch might conflict with changes made in the development branch and these conflicts need to be resolved.

Git has features designed to manage this process and these are what we will look at in the following section.

4.2 Creating a Branch

Usually we want to immediately start working on a newly created branch and so the checkout command allows also for the creation of branches using the '-b' flag:

```
C:\Users\ezzpe\2025_ezzpe>git checkout -b devel
Switched to a new branch 'devel'

C:\Users\ezzpe\2025_ezzpe> git branch -v
* devel aa0880b Added hi_and_date file
master    aa0880b Added hi_and_date file

C:\Users\ezzpe\2025_ezzpe>
```

You can read off a number of things from this output:

- Two branches exist and we are on the one marked by an asterisk
- Both branches point (for now) to the same commit

Next you can add something new to the 'devel' branch so that it begins to differ from 'master': **Add a file called 'git-cheat-sheet.txt'** containg a few words that summarise what each of the commands 'show' 'log' 'checkout' 'init' 'reset' do. Add and commit this file on the current branch ('devel').

5 Merging Branches

Say you are extremely happy with how your git-cheat-sheet.txt is getting along. In that case, you will probably want to merge it into the master branch so that it will be part of your next release. At this point the history looks like this:

```
C:\Users\ezzpe\2025_ezzpe>git log --graph --oneline
* e4e03a3 (HEAD -> devel) Added cheatsheet
* aa0880b (master) Added hi_and_date file
* 9515bb2 Initial Commit

C:\Users\ezzpe\2025_ezzpe>
```

This output is telling you that there was one initial commit ('9515bb2'), and then after this two more commits but each made to a different branch ('aa0880b' on 'master' and 'e4e03a3' on 'devel'). The 'HEAD -¿' text in from of the branch name for the last commit is telling you that 'HEAD' points to this commit. HEAD is just a pointer that points to the active commit, the one you are currently working on - in this case its the last commit made to the devel branch.

Now switch back to the master branch.

```
C:\Users\ezzpe\2025_ezzpe>git checkout master
Switched to branch 'master'

C:\Users\ezzpe\2025_ezzpe>
```

Notice that after you execute the 'checkout master' command the cheatsheet document disappears because it only exists on the devel branch.

```
C:\Users\ezzpe\2025_ezzpe>git merge devel
Updating aa0880b..e4e03a3
Fast-forward
git-cheat-sheet.txt | 1 +
1 file changed, 1 insertion(+)
create mode 100644 git-cheat-sheet.txt

C:\Users\ezzpe\2025_ezzpe>git log --graph --oneline
* e4e03a3 (HEAD -> master, devel) Added cheatsheet
* aa0880b Added hi_and_date file
* 9515bb2 Initial Commit

C:\Users\ezzpe\2025_ezzpe>
```

Now master and devel are pointing at the same commit. Not only does this mean the share the same content, but they also have the same history. Even though merge seems to imply some very smart algortihm was employed to commensurate the two version, all that really happened was that the master reference was moved to point at the same commit as the devel reference. Hence the name fast-forward merge as reported by git.

Switch to devel Add a line saying *Priority: devel* to 'README.md', add and commit. Switch back to 'master' and add a line saying *Priority: master* to 'README.md' and commit. Inspect the log (provide the '–all' flag to get the history of all branches in one view) as above, does it look more complicated? Switch to 'master' and merge 'devel' into the master branch. What happens?

6 Resolving Conflicts

The log command you used just now show a splitting of the history. This is a result of work being done on the two branches after they were created and before we merged them. Merging the branches by fast-forwarding is not an option in this case. Instead git uses a more advanced algorithm to try to combine the work done in both branches. When changes were made in the exact same location, this algorithm gives up and asks for our help. From our point of view, a conflict has emerged.

If you open a file that has a conflict in it, you see conflict markers like below:

```
>git merge devel
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the result.
```

Now look in the README.md file where the problem exists:

```
Hello

<<<<<<< HEAD
```

```
Priority: master
=======
Priority: devel
>>>>>>> devel
```

You get the two versions encountered in the two branches included in the same file and enclosed by conflict markers. The conflict markers show where the two files differ and it is your job to choose how to proceed: you can keep one of the version, or modify the file in anyway you see fit, and remove the conflict markers. To indicate to git that the conflict is resolved, add the file to the index and commit. This will be recorded as a merge commit.

```
C:\Users\ezzpe\2025_ezzpe>git merge devel
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the result.

C:\Users\ezzpe\2025_ezzpe>git add README.md

C:\Users\ezzpe\2025_ezzpe>git commit -am "Resolved conflict"


C:\Users\ezzpe\2025_ezzpe>git log --graph --all --oneline
*   ef24aa2 (HEAD -> master) Resolved conflict
|\
| * 7ee6392 (devel) devel modified readme
* | 3127313 Master modified readme
|/
* e4e03a3 Added cheatsheet
* aa0880b Added hi_and_date file
* 9515bb2 Initial Commit

C:\Users\ezzpe\2025_ezzpe>
```

And history is unified again! Being in conflict with yourself is fairly rare, but Conflicts with other developers are more common as you often do not know which part of the code they are working on.

7 The .gitignore file

Git is well suited to keep track of source code and other simple text format based content. The things it is not good at are:

- Dealing with content where a simple semantic modification leads to file-wide changes in the representation of this information. E.g. don't try to have git manage your word or excel files.
- Large files. Git is a distributed versioning system. This means everyone needs to have a complete history. If large files are part of this history, this causes a lot of network traffic and corresponding delays. Github also has repository size limits.
  A very common mistake in this project is accidentally adding Visual Studio project directories to git. These directories include build files which can be many 100's of MB when building VTK and Qt linked applications. If you do this, Github will reject your code, and the problem is quite difficult to fix - you don't just need to delete the large files, or remove them from your current git image, you need to make git completely forget that they ever existed!

This poses no serious restrictions on software development. In fact, when developing portable projects, you typically want to keep the entire shared code base free of any binary or system specific file types.

Keeping these system specific files out of reach of git is not always a simple task. For example while building the software (i.e. compiling, linking, . . .), many build systems pollute the code base with project files, and intermediate/final binaries.

Git has a simple but effective way to exclude these files. We can provide a file called .gitignore listing all types of files and names of subdirectories that should be exluded from version management. An example of such a file is:

```
# Contents of .gitignore
*.com
*.class
*.dll
*.exe
*.o
*.so
build/
```

The example above is quite simple and it won't correctly ignore all of the project file types that advanced software development IDEs, such as Microsoft Visual Studio, create. If you search on the internet you can find examples of .gitignore files that can be used or adapted for different types of repository and software development tools. A good place to start is here: https://github.com/github/gitignore. You need to add the gitignore before it will be effective.

Make sure you have a suitable .gitignore file(s) in your repositories. You should only be storing code on github, not binary files / executables. You will lose marks if you have uploaded files generated by your compiler to Github.

Important: As mentioned previously, Visual studio can generate 100s of MB of data when building a large project, if you accidentally commit this alongside your code it will cause you lots of problems! Github (see later section) will reject the repository as it is too large. Removing the excessive data after a commit is actually quite difficult. After you have added your changes, use git status to check what has actually been added. If you have added something you didn't mean to (perhaps because the .gitignore didnt work as expected) then use git reset to fix the problem before git committing. Once you have commited something you didnt mean to, it is difficult to completely undo.

8 Closing remarks on Git

This is a lot to take in. Git was designed to manage extremely large projects. Actually it was designed for the Linux kernel. You will learn git gradually. My advice is to create a repo just for messing around where you can experiment with what each command does.

9 Remote Repositories and Github

To work together with others it is important that there is a central repository that is kept up to date and that is accessible to everyone on the team. In it most primitive form you could set up a shared drive and simply put the repository there.

Github is a free service (for open source projects and for academic use) that provides a place to put your communal repository but on top also offers many useful features:

- Issue trackers

- A Project website
- An in-browser visualisation of your code-base
- Pull request support
- Permission management
- An API that other services such as continuous integration tools can hook into

Your application for the student pack should ave come through by now. Log into Github and click the plus button to generate your first online repository (use '2025_STUDENTID' for your repository name).

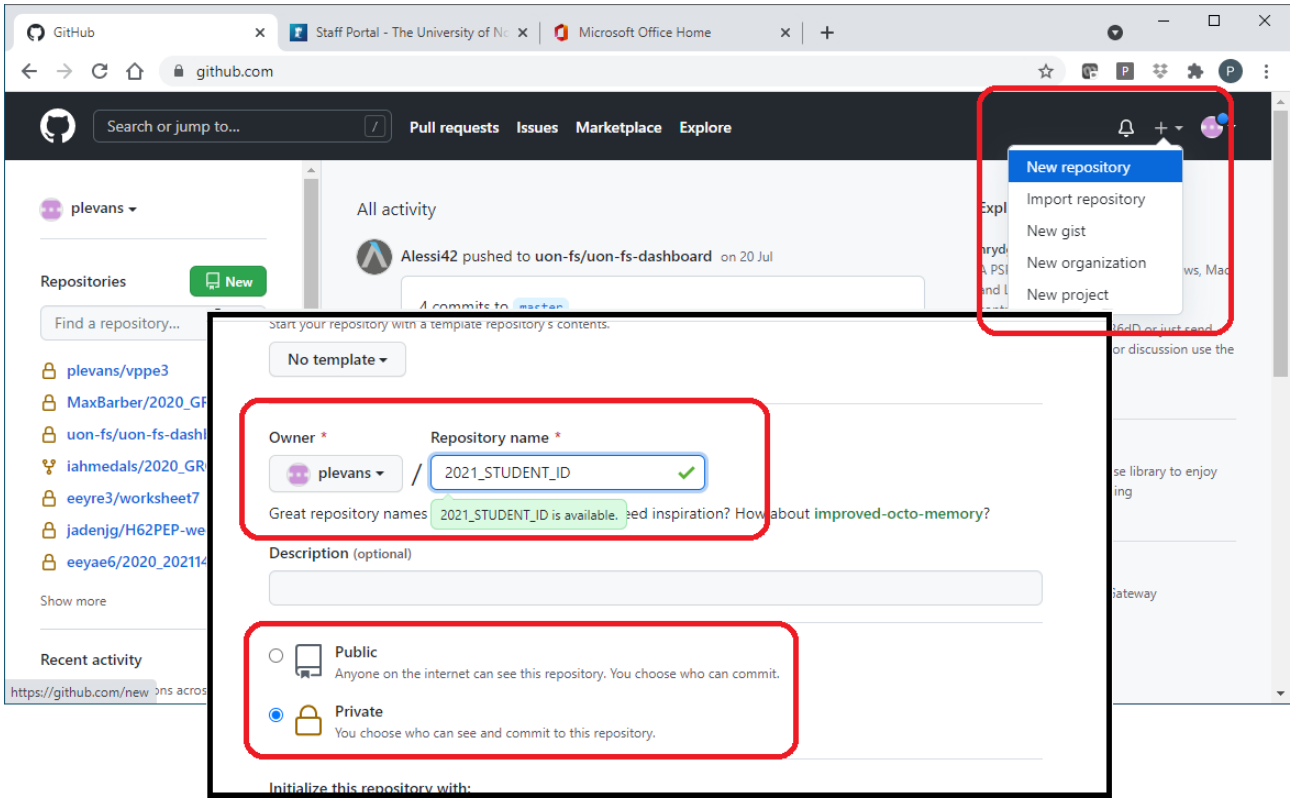

Figure 1: New Repository

As suggested by the dialog, we can connect our local repository to the Github repository by defining it as a so called remote:

```
C:\Users\ezzpe\2025_ezzpe>git remote add origin https://github.com/username/reponame.
    git

C:\Users\ezzpe\2025_ezzpe>git push -u origin master
Counting objects: 22, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (16/16), done.
Writing objects: 100% (22/22), 2.21 KiB | 755.00 KiB/s, done.
```

```
Total 22 (delta 3), reused 0 (delta 0)
remote: Resolving deltas: 100% (3/3), done.
To https://github.com/username/reponame.git
* [new branch] master -> master
Branch master set up to track remote branch master from origin.
```

It is possible you will be asked for your Github credentials. The name 'origin' is arbitrary but it is common practice to use this name for the *upstream* repository. Upstream in this context means that you consider it the repository that all your work needs to end up in.

The git push command is used to actually copy over your commits so far to Github. The flag '-u' is only required the first time and help git remember this is the upstream repository for the current branch. Later invocations of push do not require you to mention origin; it is understood that a push executed whilst on the master branch is to copy your data to 'origin'.

Let's pretend we are another user now (even though you will work under the same Github credentials). Make a directory otheruser and execute the following command from inside that directory:

```
C:\Users\ezzpe\otheruser>git clone https://github.com/username/reponame.git
```

You will see that the entire history of the repository is cloned or copied to this new directory.

- **Verify you have the entire history using** git log.
- **Ask for a list of remote repositories** by issuing git remote –v. Note that the repo you pulled from is automatically included, named origin, and set as the upstream destination for the master branch.
- **Pretend to be user 1 again**. Move into the user 1 repo. Make some changes to README.md (for example "User 1 says hello!").
- **Add, commit, and push** to the Github based repository (a simple git push whilst on 'master' sufffices). Move to the repo of user 2. Pull down the new work added by user 1 by issuing the git pull command. origin.

Just like with a normal merge, this could potentially lead to conflicts. Investigate this in the following exercise:

- **(Add,Commit,Push) as User1 a line in README.md saying "I want it this way"**.
- **(Add,Commit)** as User2 a line in README.md saying "I want it that way".
- **Try** pushing.
- Does this work? Why not? **Follow the instructions git gives you to resolve the situation**.

Your history should look something like this now:

```
content...
C:\Users\ezzpe\2025_ezzpe>git log --all --graph --oneline
* b450297 (HEAD -> master, origin/master, origin/HEAD) We found middle ground
|\
| * 1cac67e User 1 wants it this way
* | 31bddb2 User 2 wants it that way
|/
* 3fb6cfb User 1 says hello
*   ef24aa2 (HEAD -> master) Resolved conflict
|\
| * 7ee6392 (devel) devel modified readme
```

```
* | 3127313 Master modified readme
|/
* e4e03a3 Added cheatsheet
* aa0880b Added hi_and_date file
* 9515bb2 Initial Commit
```

10 The fork, pull, push, pull request model

Choose a team leader and then get the team leader, create a new, empty online repo. It doesn't really matter who this is, but you only need one group repository! This will be used to hold your group work so give it a sensible name like *2025_GROUP_X* where X is your group number. The repository should be private so other groups can't see what you are doing! Now you need to make sure all group members can contribute to the group repository.

Work through the following process as a group:

- Team leader, go to the Collaborators page (in settings) of your repo and add your minions to the list of users with read access.
- Minions, surf to the website of your leader's repo and fork it (there is a fork button on the page, just click it). This creates a copy of the repo in your account. This is a completely separate copy that you can edit without changing the original repository, the original repository is known as the upstream. The fork retains a link to the original so you have the option of sending your changes back to the upstream later. your Github account.
- Minions, clone your fork to a local directory (browse the the fork on your Github account, copy the web address, git clone on your PC). Add a file named ¡username¿.txt and add some content to it. Add, commit, and push. This push will be to your own fork, so no conflicts are possible.
- Minions, go to the Github page of your own fork and click the New Pull Request button. Check the settings and submit the pull request. A Pull Request is a method for you to alert the Team Leader that you have changes in your fork that you would like merging into the upstream repository.
- Leader, navigate to the *Pull Requests* tab on your Github repo's site. Check that you are happy with the changes you are asked to pull in and confirm. You are approving the changes and allowing them to be merged into the upstream.

A pull request is essentially an automatic email send to the leader to perform a pull from one of the minions. Just like any merge operation, this could result in conflict if non-compatible work has been done in the origin.

It is typically considered the contributor (let's use that instead of minion) responsibility to resolve these conflicts. The process is:

- A minion defines the leader's repo as a remote, usually called upstream.
- Upon conflicts a minion pulls from origin. This results in conflicts, which can be resolved locally. A merge commit is pushed.
- The pull request is automatically updated and should now notify the absence of conflicts
- The leader can now confirm the pull request, which will be performed by simple fast forwarding

If another user merges changes into the upsteam, and you then need to get these changes reflected in your fork, you have a number of options. The easiest of which is to use the *Sync Fork* button on Github. There are also a number of command line variations, including the traditional approach shown below. First you will need to add the upstream repository URL as a remote, this step only needs to be done one.

```
C:\Users\ezzpe\2025_ezzpe>git remote add upstream <original repo URL>
```

You can then fetch and merge the upstream into your fork:

```
C:\Users\ezzpe\2025_ezzpe>git fetch upstream
C:\Users\ezzpe\2025_ezzpe>git merge upstream/master
```

11 Software Releases on Github

Commits usually represent relatively small additions or changes to source code, but at some point you might want to mark the software as ready for download and use. This can be done as a Github release, there is a *Create new release* option on the right hand side of the Github repository page. Click this, give the release a *tag* (version number) and a name, and a description. There is also an option to upload binaries here, this allows you to provide a compiled version of the code for download (e.g. as a .exe, a zip, or some other form of installer). The release will then be listed on your Github page and it will allow someone to download the binaries that you uploaded, and the source code at that point in time as a zip. Try this feature out, you will use this feature in this module.

12 Final actions required to configure your repositories

Finally, make sure you do the following before moving on to the next worksheet:

- **Populate the root directory of the group repository with a 'README.md'**. Give an overview of the project in README.md and list your team members. Create a few test files and practice contributing to get familiar with the pull request dynamics.
- **Add a suitable .gitignore file**, you will get marked down if you upload build artefacts (.exe, .lib, etc) files when you start compiling code in the following worksheets.
- **Add 'plevans' to the collaborator list** for all of your repositories (individual and group) - remember I will need to see them to be able to mark them!

13 Git with GUIs

We have covered the basic commands that are required to use Git. You can continue to use Git from the command line, in fact this is often the easiest way as you have complete control cover what you are doing. There are other options that use a GUI though, you can download Github Desktop or GitKraken - these are applications that allow you to manage Git repositories using a typical Windows GUI. There is also usually a Git interfece built into most modern IDEs too - e.g. Visual Studio, Qt Designer, CLion, etc (if you haven't come across these yet - they are basically just modern versions of Code::Blocks and we will use Visual Studio and Qt Designer in this project).

The reason we looked at Git from the command line is because you are able to clearly see what each command does. If you have this understanding, you can quickly learn to use any of the GUIs. You are free to use Git in any way you choose - it is only your repository on Github that is marked, how you create this and commit/push to it doesn't matter.

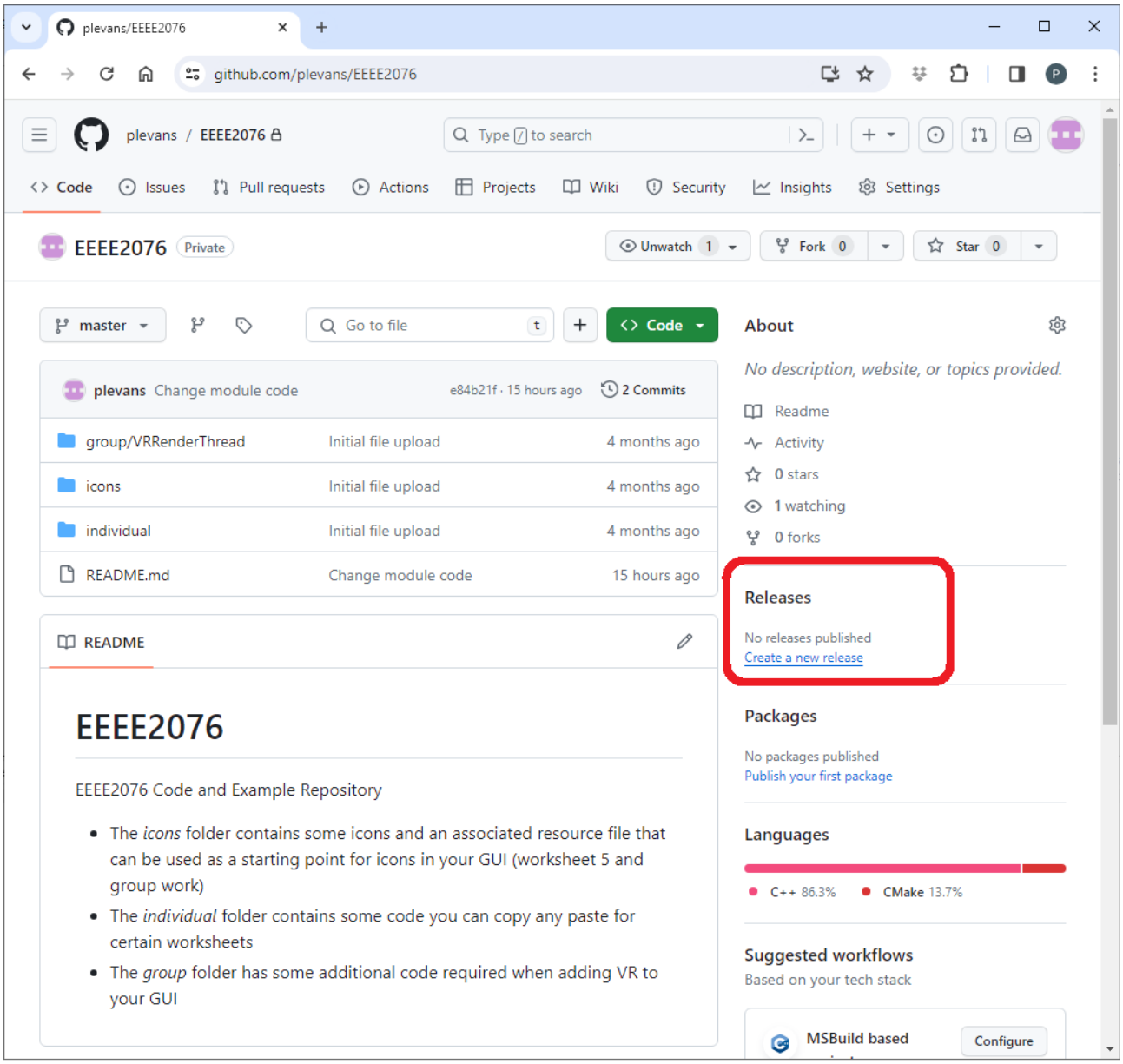

Figure 2: Creating a release

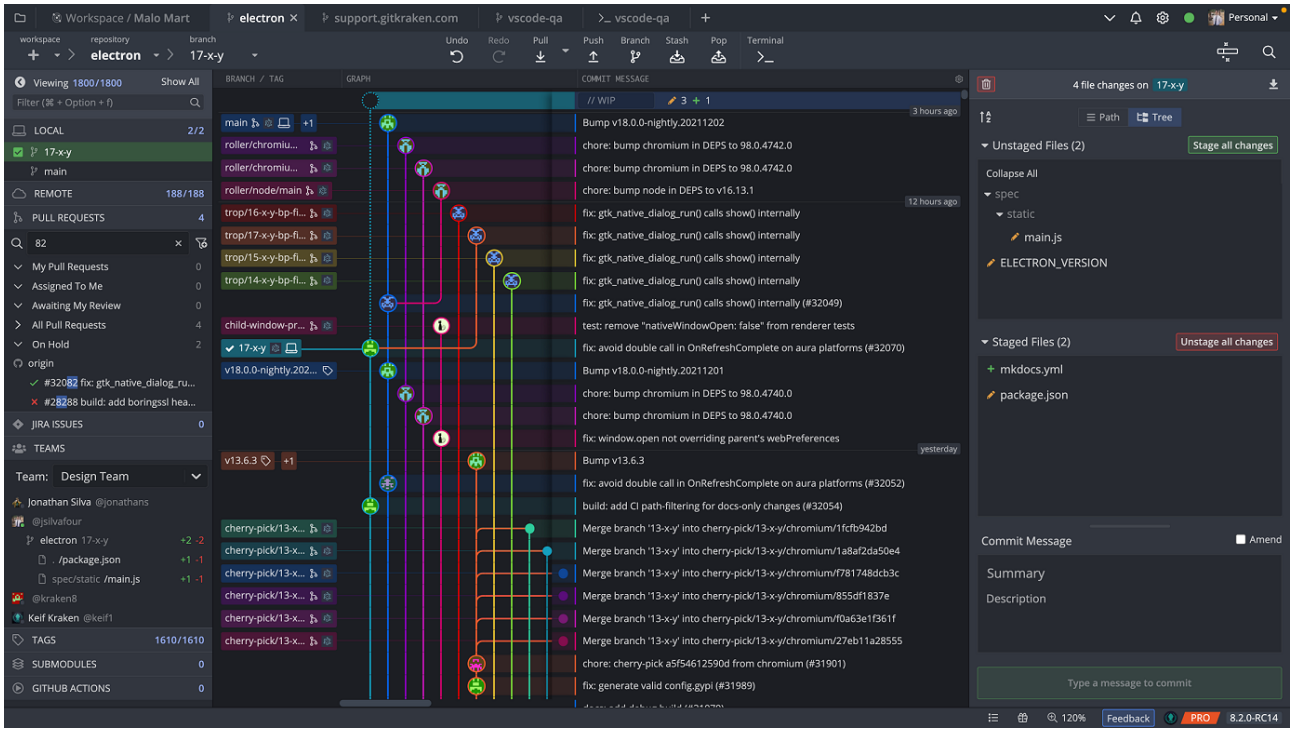

Figure 3: Gitkraken