Week 5 Lab

Assertions, Exceptions, Javadoc, and Git

Objectives

In this week's lab, you will:

- use a Git repository to manage your code
- use assertions and exceptions to enforce specifications
- use a Map
- document code using Javadoc

UML Class Diagram

You must draw a UML class diagram showing the system that you plan to implement *by the end of the lab*. Note that this system will incorporate work you have done in multiple tasks, so **read the entire lab sheet** carefully.

This diagram should be quick and simple. It only needs to show the classes you plan to create and the relationships (*associations in UML*) between them. You don't need to show fields or methods. You are strongly encouraged to draw this diagram by hand — you don't need to use a UML diagramming tool.

You must get your design in UML approved by your demonstrator before you start coding. Both the design and the UML syntax must be complete and correct.

Task 1. Set up your Monash GitLab account and link it to your project for this week

Follow the instructions in the "Getting started with Git" section at the end of this document. Get your demonstrator to check that you have got it all set up and working correctly before attempting the following tasks.

Task 2. Use assertions and/or exceptions to enforce specifications

You have been informed that valid Minion IDs consist of a letter in the range A–Z, followed by seven integers, e.g. `A1234567` or `K2233445`. If you have used an integer to represent an ID in `Minion` and its subclasses up to now, you will need to refactor it so that it uses a `String`.

Modify the constructor of `Minion` so that it throws an exception when an attempt is made to construct a `Minion` object where the above specification for valid IDs is violated.

Add code to `Lair` to verify that this works as expected before proceeding to the next task.

Commit all changed and/or added files, and push to the upstream repository that was created for you on the Monash GitLab server.

Task 3. Add a `Team` class, and refactor `LairLocation` to use it

At present, minions are assigned to an `LairLocation` individually. The analysts have now determined that this is inadequate. A complete team needs to be assigned to a `LairLocation` at once, so that the rules for a valid combination of staff can be enforced.

Create a `Team` class. It will need an attribute to store references to the `Minion` objects in the team. It will need methods for

- adding a minion to the team
- removing a minion from the team
- getting a `List` containing references to all the `Minion` objects in the team
- returning the number of minions in the team that have a given `MinionSkill` (passed in as a parameter)

Refactor `LairLocation` so that the `minions` attribute is of type `Team` (rather than a Java library class such as `ArrayList<Minion>`).

Remove the method `assignMinion(...)` and replace it with one called `assignTeam(...)` that takes a `Team` object as its parameter, and sets the `minions` attribute accordingly.

All other existing methods of `LairLocation` must be refactored to use the new `minions` attribute type internally, but have unchanged behaviour from the point of view of the clients of the class.

Add code to `Lair` to verify that this works as expected before proceeding to the next task.

Commit all changed and/or added files, and push to the upstream repository that was created for you on the Monash GitLab server.

Task 4. Establish required skills for `LairLocations`

Different parts of the lair require different skillsets in order to be operated correctly, and some require larger teams than others. Before we can assign `Teams` to `LairLocations`, we'll need to be sure that there are enough qualified people on the `Team`. `LairLocation` will thus need an attribute to represent the number of qualified personnel in each skill that it will need.

This attribute must be a reference to a data structure that is able to represent the number of qualified `Minions` (an integer) that the `LairLocation` needs for any given `MinionSkill`. A data structure implementing the Java `Map` interface would be suitable to meet these requirements. Choose a suitable concrete implementation of `Map` and add an appropriate attribute – the keys should be `MinionSkills` and the values should be integers. This attribute should not be accessible from outside `LairLocation` and won't need to change after it has been created. Create the map in `Lair` and pass it into `LairLocation`'s constructor so that you won't need to implement any accessor or mutator methods: the constructor's signature will need to change, but no other parts of the class's interface should be affected.

Add a comment to your code to explain why you chose the particular concrete `Map` implementation that you used. Commit all changed and/or added files, and push to the upstream repository once more.

Task 5. Modify `assignTeam(...)` to enforce the rules for a valid `Team`

The analysts tell you that for a team of minions to be valid, it must obey the following rules

- it must have at least one member

- there must be at least one support minion for every researcher minion in the team[1]

In order to assign a `Team` to a `LairLocation`, the following additional criterion must be met:

- the team must have enough qualified minions in each skill that the `LairLocation` requires (e.g. if the `Map` in the `LairLocation` maps `MinionSkills.SCUBA` onto 3, the `Team` must contain *at least* 3 minions that have the `MinionSkills.SCUBA` skill.)

Modify `assignTeam(...)` so that it throws an exception when an attempt is made to assign a `Team` to a location when the rules above are violated.

Given that clients should be able to check whether they are about to violate the rules, you should give `LairLocation` a method called something like `isValidTeam(...)` that takes a `Team` as a parameter and returns a boolean indicating whether the `Team` meets the rules for the location. Use this method in the precondition check for `assignTeam(...)` to avoid repeated code.

Add code to `Lair` to verify that this works as expected before proceeding to the next task.

Commit all changed and/or added files, and push to the upstream repository that was created for you on the Monash GitLab server.

Task 6. Document class APIs using Javadoc

Code can be worked on by many people and have a very long lifespan. It is thus vitally important to document your code properly.

The easiest and best place to document the details of an individual class is in the class itself. Documentation in the class is essential when we need to maintain or extend the class. Other programmers using your class, however, should not be required to look at the source code — it's slow and can lead to them relying on implementation details you want the freedom to change later.[2] For library classes, the source code may not even be available. It is thus vitally important to be able to produce separate documentation for the *interface* of the class: those class members that are visible from outside the class.

We can use Javadoc to simplify this process — if you format your comments correctly, you can generate well-formatted API documentation automatically.[3] This is how the Java API documentation is generated.

On Moodle, we have provided you with two guides to writing Javadoc documentation: one on the syntax of Javadoc, and another on the principles of what to write in code comments. We have also provided you with a documentation standard for code written for FIT2099.

Document all the classes you have written to handle minions and teams following the documentation standard provided. Generate HTML documentation using the instructions in the guides. Make sure you comment each method as well as the classes themselves.

Commit all changed and/or added files, and push to the upstream repository that was created for you on the Monash GitLab server.

Getting marked

Once you have completed these tasks, call your lab demonstrator to be marked. Your demonstrator will ask you to do a "walkthrough" of your program, in which you explain what each part does. Note that to receive full marks, **it is not sufficient merely to produce correct output**. Your solution must implement all structures specified above *exactly* — as software engineers we must follow specifications

[1] Look up the Java `instanceof` keyword. Later in the semester we will discuss why the use of `instanceof` is often considered a code smell, but it is fine to use it for this exercise.

[2] Can you think of ways in which a client of your class could rely on implementation details even if the attributes and methods involved are `private`?

[3] API is an abbreviation for Application Programming Interface.

precisely. If you have not done so, your demonstrator will ask you to fix your program so that it does (if you want to get full marks).

If you do not get this finished by the end of this week's lab, you can complete it during the following week and get it marked at the start of your next lab — but make sure that your demonstrator has at least given you feedback on your design before you leave, so that you don't waste time implementing a poor or incorrect design. That will be the last chance to get marks for the exercise. You cannot get marked for an exercise more than one week after the lab for which it was set.

Marking

This lab is worth 2% of your final mark for FIT2099. Marks will be granted as follows:

- 0 marks if you do not attempt the tasks, or if you could only complete them once given the solution
- 1 mark if some tasks are incomplete, if the design or the implementation is flawed, or if you needed to see a significant part of the solution to complete them
- 2 marks if you were able to complete all tasks independently, with good design and correct implementation

Appendix: Getting started with Git

You will be required to use Git to manage all the code and other documents you produce throughout your three assignments in this unit (and others). From this week, you will start using Git in your lab exercises too. You must have completed the set reading on Git available via the Week 5 section before commencing this lab.

In labs, you will use Git via your IDE. Nearly all modern IDEs (including Eclipse) have Git integration. It is also possible to use Git on the command line, or via Git clients such as GitKraken.

For both these lab tasks and your assignments, you will use a GitLab server run by Monash. You can read about using GitLab at `https://gitlab.com/help/#getting-started-with-gitlab`. (Note: you will be using repositories created for you and hosted at Monash (`https://git.infotech.monash.edu.au/`), **not** `https://gitlab.com`.)

Your code *and all other documents* will be stored on the Monash GitLab server. In the assignments, we will use your project's commit log to work out who's been contributing to the project and who has not. That means that it is *very important* that you commit and push your work frequently—*including your design documentation*. It's also good practice to do this as it makes it much less likely that you'll lose work to a crash or accidentally deleting it: laptops get lost, hard disks become corrupt, and cloud services go offline at crucial moments. If Monash GitLab goes down, however, it's our problem, not yours.

In this lab, you will use Git and a repository on the Monash GitLab server for the first time in this unit. These instructions were developed in 2018 by David Squire, so the details of the lab tasks are different, but we are still using the same version of Eclipse and GitLab so the important details are the same.

Setting up your account on Monash GitLab

If you have never used the Monash GitLab server before, the first thing you need to do is log into the server, and set up a *Personal Access Token*:

- Login to the Monash GitLab server using SAML (your Monash login details).
- Go to `https://git.infotech.monash.edu/profile/personal_access_tokens` and create a Personal Access Token, with the API option checked. Set the expiration date to 2019-12-31
- Copy the Personal Access Token and save it somewhere you can always find it. (You can always create a new one if the token expires or you didn't save it.)

Use your Monash email address as your username and your Personal Access Token as your password whenever using a repository hosted on the Monash GitLab server.

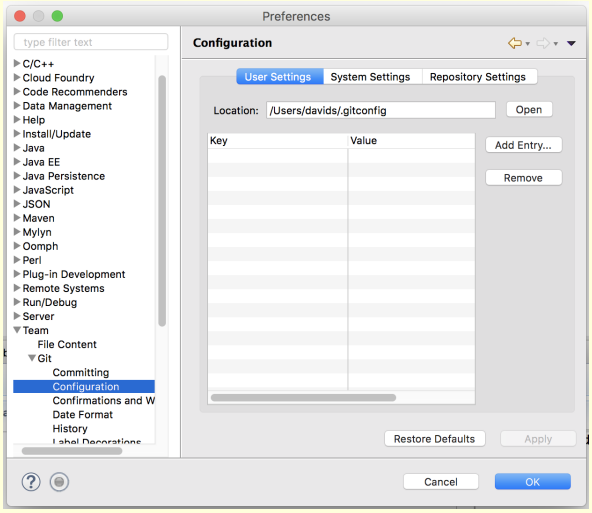
Connecting your IDE to your repository

A project (and thus repository) for your lab code has been created for you. It has the name:
`FIT2099-2019-S1-LabTasks-your_monash_email_username`
(e.g. if your Monash email address is `abcd0001@student.monash.edu`, then your project is `FIT2099-2019-S1-LabTasks-abcd0001`). You will see it if you click on Projects | Your Projects at the Monash GitLab website. You *must* use this repository. **Do not create another**.

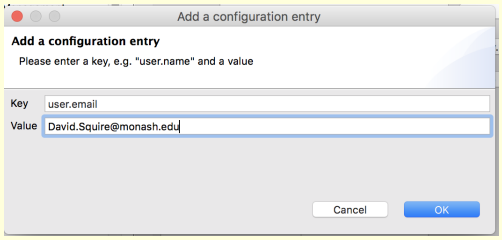
Note: The following instructions are for Eclipse. If you are using another IDE, you will have to work out how to do this yourself (there are certain to be instructions on the web). You can also consult `https://wiki.eclipse.org/EGit/User_Guide#Basic_Tutorial:_Adding_a_project_to_version_control` for something similar to the following, using MS Windows. Remember, however, that this is different, because you *must* use the Monash GitLab repository already created for you.

If you have not previously configured Git on your computer, you first need to set up the name and email address that will be used for as you Git identity.

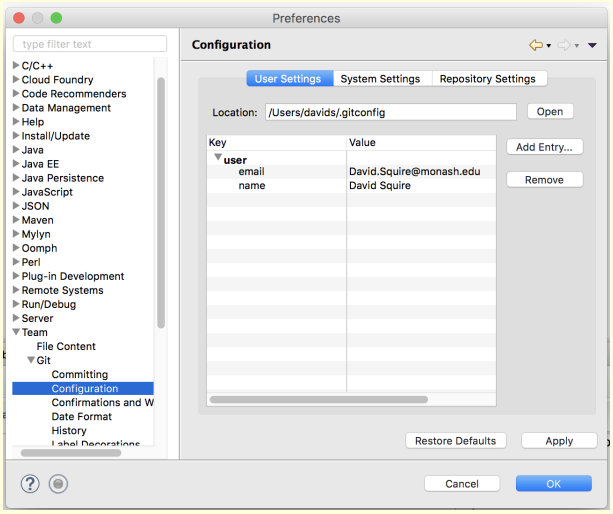
- Open Eclipse preferences. Go to Team | Git | Configuration



- Click Add Entry... and add an entry for `user.email` using your Monash email address.



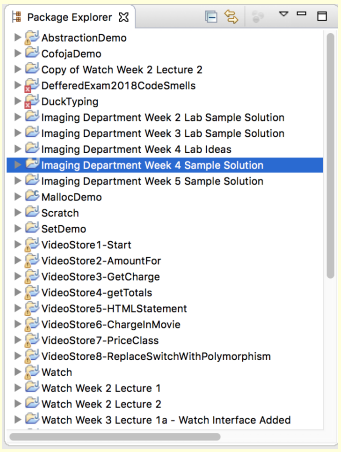
Then add an entry for `user.name` with the name you want to use for Git. The result should be like



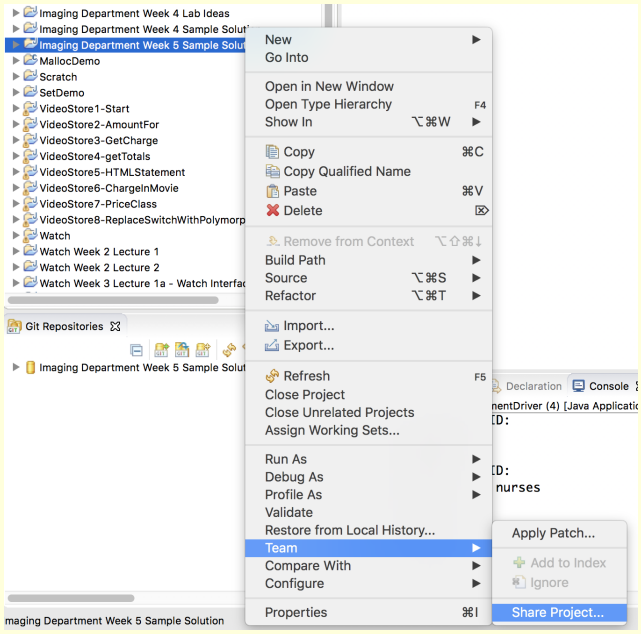
Click OK.

The next task is to create a new project containing your Week 4 code. In this example, mine is named `Imaging Department Week 5 Sample Solution`. Once we have created it, we will create a local Git repository. Finally we will merge it with the state of the repository created for you on the Monash GitLab server.

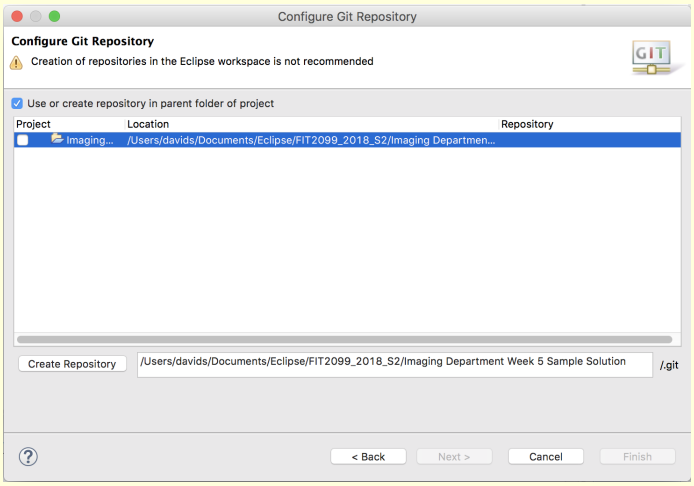
- Start by simply copying and pasting your Week 4 project in Project Explorer at top left in the Eclipse window.



- Right-click on the newly created project, and select Team | Share Project...

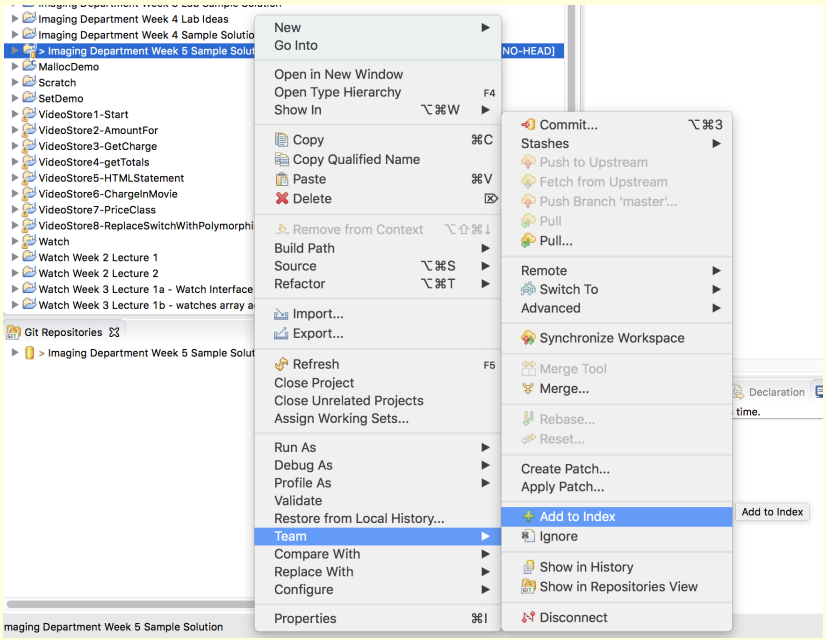


- Select Git in the dialog box and click Next.
- Tick the box Use or create repository in parent folder of project and select the project you have just created.



Note that you are first creating a local Git repository on your computer. Next we will connect this with the repository on the Monash GitLab server. Git is a *distributed* version control system, so everyone has a fully-fledged repository.

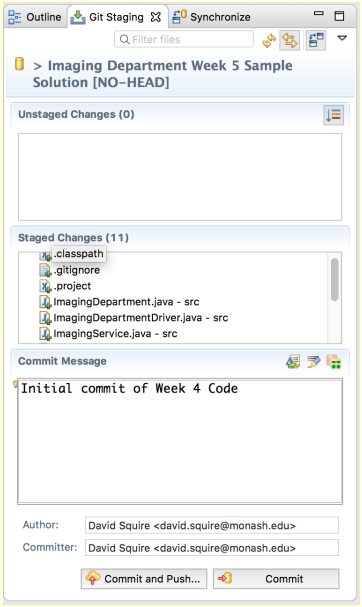
- Click on Create Repository, and then on Finish.
- Right-click on the project, and select Team | Add to Index.



This adds everything currently in the project to the list of things to be added to the git repository.

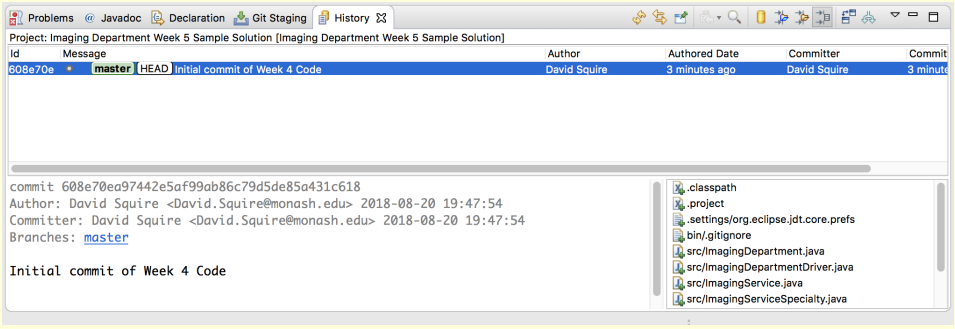
- Right-click on the project, and select Team | Commit... This will cause a Git Staging tab to appear (it could be at the right of the Eclipse window, or at the bottom, depending on your layout).

- Enter a commit message in the Git Staging tab at

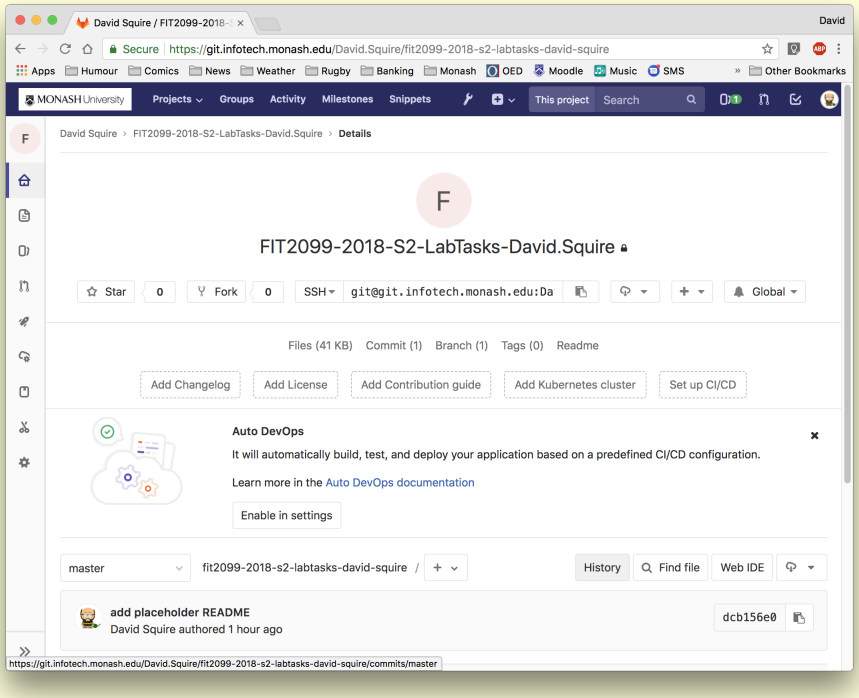


  All commits must have meaningful commit messages. These form an important part of the documentation of your project, and are a valuable tool for communicating with other team members.

- Click Commit to commit your files to the repository for the first time.
- Right-click on the project, and select Team | Show in History. You will see something like this

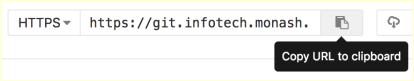


- Go to your project in the Monash GitLab web interface in your web browser. You will see something like

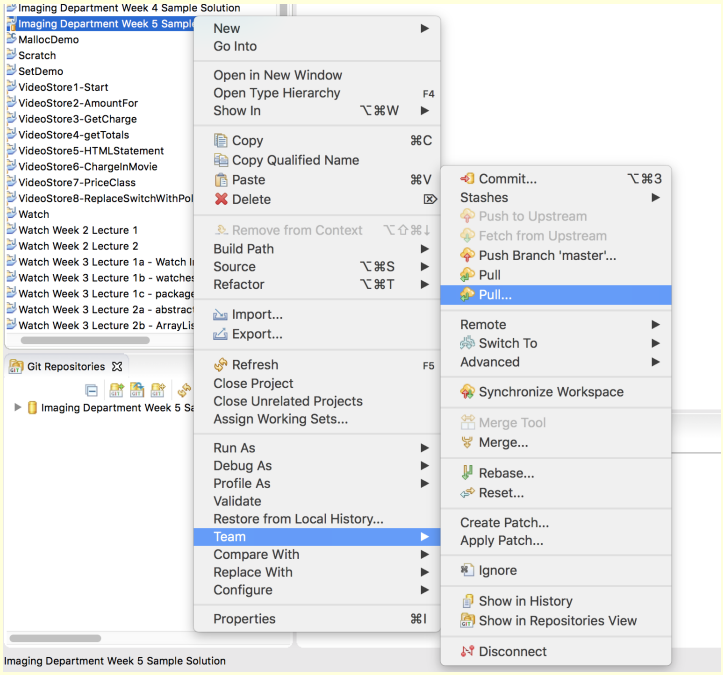


  Notice that below the project name, the repository address appears, with SSH beside it. The Monash Gitlab server is behind a AWS Load Balancer which only supports HTTP/HTTPS. If you try to use SSH, it will time out. You must thus *always* use the HTTPS address of your repository instead.

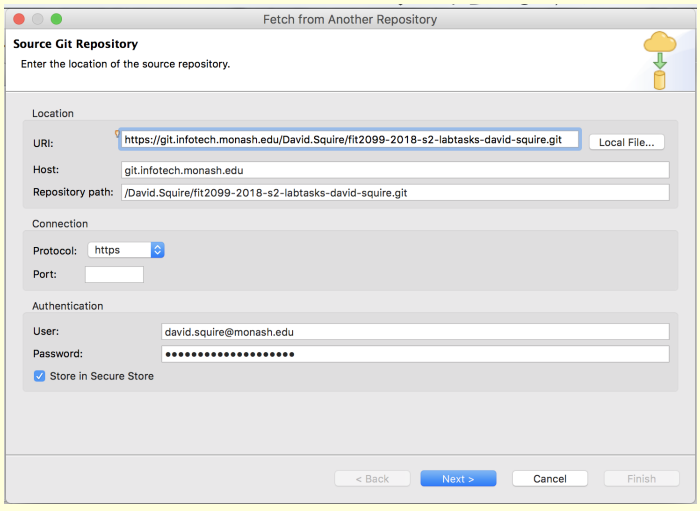
- Use the drop-down menu to the left of the repository address to select HTTPS.
- Click on the icon to the right of the repository address to copy the address to the clipboard:



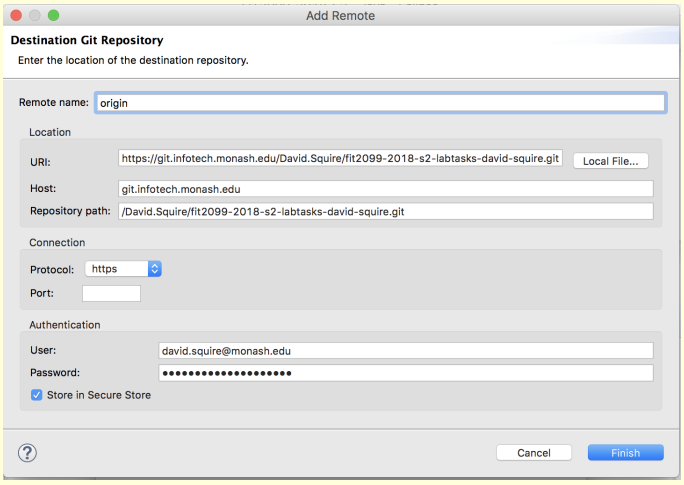
- Now, in Eclipse, Right-click on the project again, and select Team | Pull...



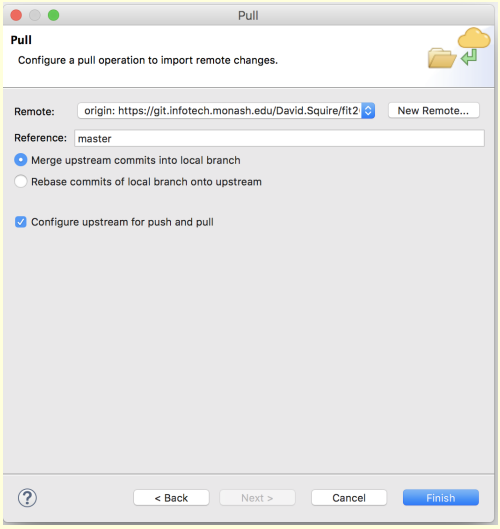
- Paste the repository address in the URI field of the dialog box. Enter your Monash email address in the User field, and the Monash GitLab Personal Access Token you created earlier in the Password field. Click Next.



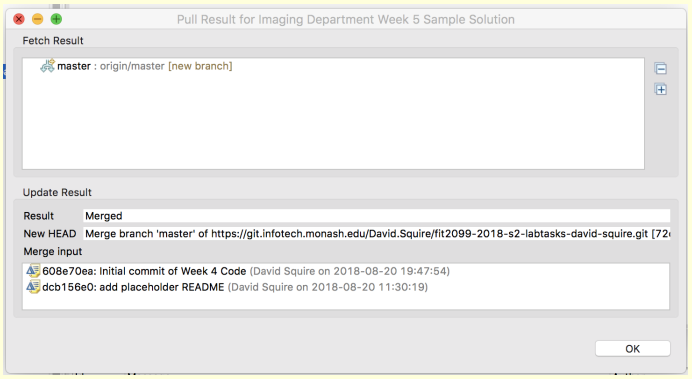
- In the next page of the dialog, click on New Remote.... The dialog that comes up should already be populated with the required details:



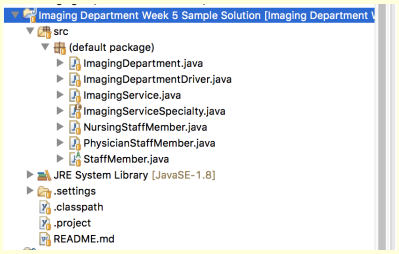
- Make sure the Merge upstream commits into local branch and Configure upstream for push and pull options are selected



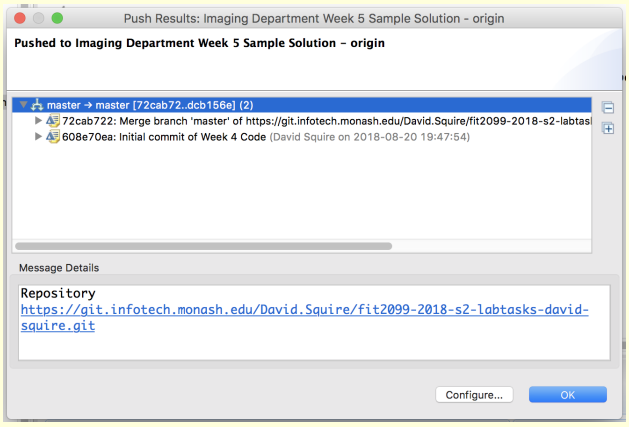
- Click on Finish. You should see a Pull Result window like this



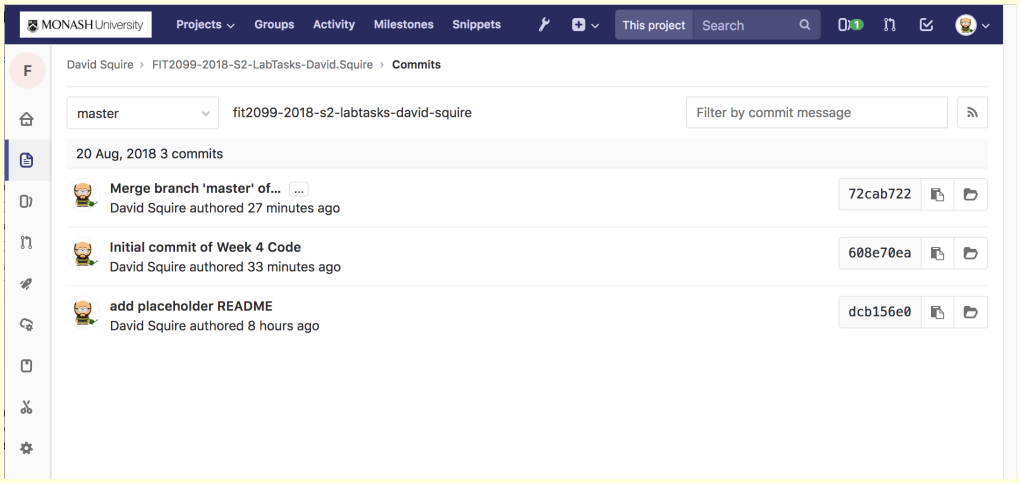
- Click OK. Look at the files in your project in the Project Explorer. You will see that the `README.md` that was already on your repository has been merged into your project.



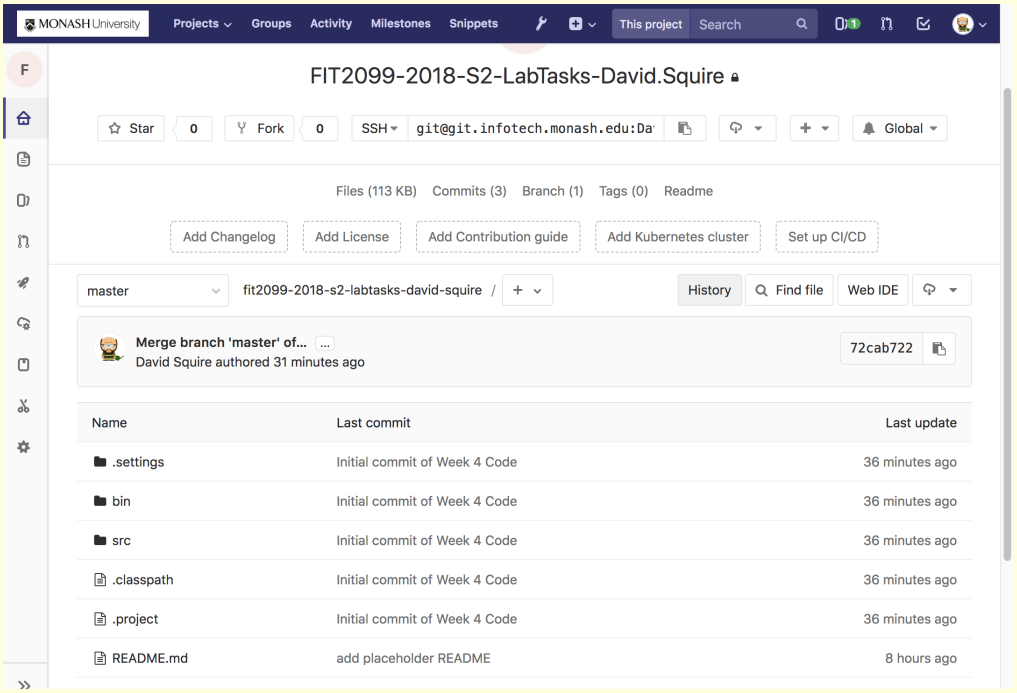
- Now that we have merged our local repository state with that of the upstream repository on the Monash GitLab server, we can push the local repository state to the Monash GitLab server. Right-click on the project and select Team | Push to Upstream. A status dialog like this will appear



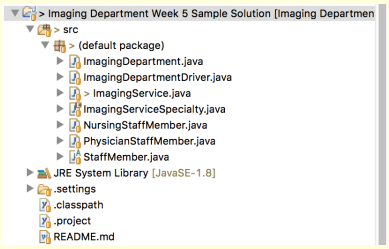
- Go back to the Monash GitLab web page for your repository in your web browser. Click on the History button. You should see something like this



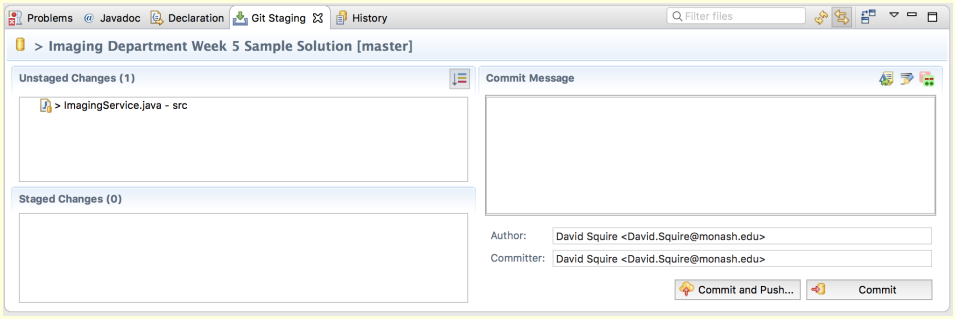
This indicates that the `master` branch on the Monash GitLab repository has indeed been merged with that from your local computer. Go back to the repository page and refresh it. You will see that the files from your Eclipse project have been pushed to the server



- Make a change to one of your source files on Eclipse (e.g. add a comment). Save the file. You will see that a symbol that appears beside the changed filename in the Project Explorer, as well as beside the package, directory, and project it is in. This is an indication that the file has changed, and the changes have not been committed to the Git repository

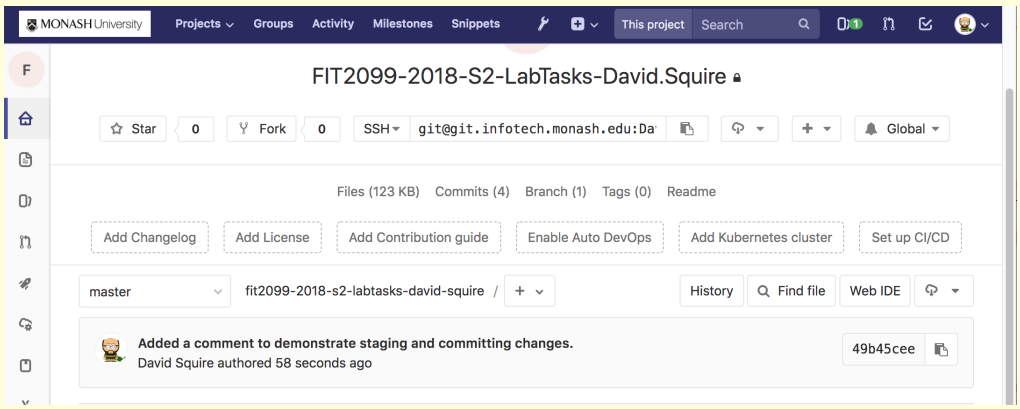


- To commit this change, right-click on the project, and select Team | Commit... This will take you to the Git Staging dialog. You will see that the changed file appears in the Unchanged Changes section.

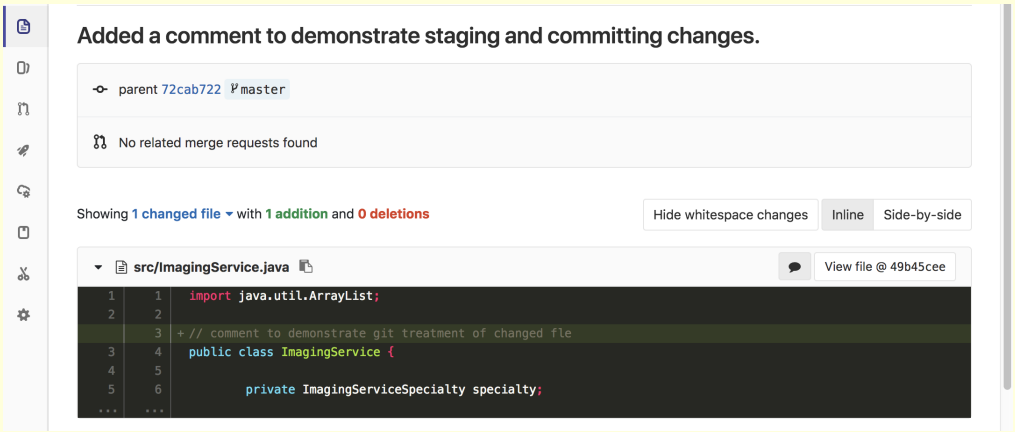


Before changes are committed, they must be staged. To do this, drag the file into the Staged Changes section. Add a commit message. Notice that there are two buttons in the bottom right of the tab: Commit and Push... and Commit. If you select Commit, the changes will be committed to your local repository. If you select Commit and Push..., the changes will be committed to your local repository, and also pushed to your upstream repository (on the Monash GitLab server).

- Click on Commit and Push.... Go to the Monash GitLab web page for your repository in your web browser, reload the page, and check that your new commit appears on the server. You should see something like



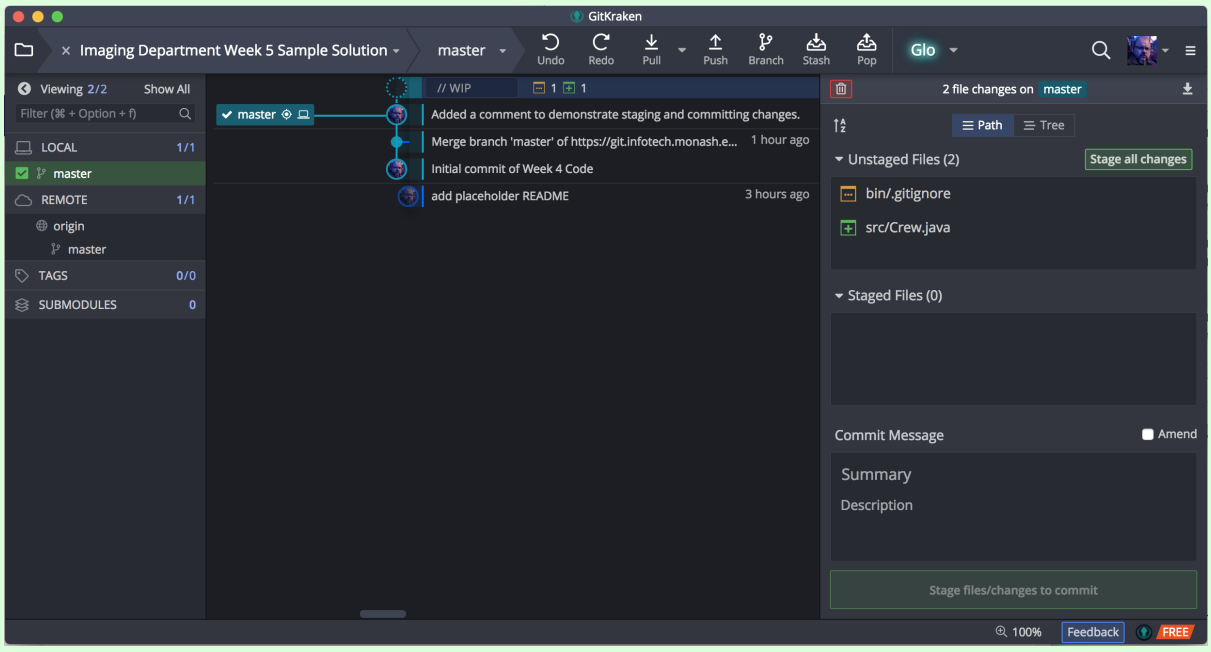
- Click on the commit message for the latest commit. For my example, that is **Added a comment to demonstrate staging and committing changes.**. This will take you to a page where you can see what changed. For my example, this is



You are encouraged to explore the GitLab interface to see what is available.

That brings us to the end of this example. In the remainder of this lab, you are required to do a commit and push after completing each task. In fact, it is a good idea to do a commit immediately after getting something working. It gives you a state you can revert back to from any time in the future.

Note: You can use GitKraken[4] in parallel with other Git tools, even if you're not using it to do commits, pushes, etc. It provides a really nice visual representation of who has committed what on various branches, and where merges have occurred. For the example above (after a couple of more local changes I made), we have



[4] https://www.gitkraken.com/git-client