Software engineering

Suzanne Embury and team

Contents

Welcome

Welcome to COMP23311: software engineering at the University of Manchester.

0.1 Making better software

The development of software systems is a challenging process. Customers expect reliable and easy to use software to be developed within a set budget and to a tight deadline. As we come to depend upon software in so many aspects of our lives, its increasing size and complexity, together with more demanding users, means the consequences of failure are increasingly severe. The stakes for today's software engineers are high!

Experience over the last few decades has taught us that software development failures are rarely caused by small scale coding problems. Instead, failures result from the difficulties of writing software that customers actually need, of keeping up with constantly changing requirements, of coping with the scale of large developments, of getting many different people with very different skill sets to work together, and of working with large bodies of existing code that no one on your team may fully understand. Being a good coder is an important part of being a good software engineer, but there are many other skills - including soft skills - that are needed too.

In this course unit, you will get the chance to expand and broaden the programming skills you gained in your first year course units by applying them in a more realistic context than is possible in a small scale lab, see figure **??**. Instead of coding from scratch, you will be working in a team to make changes to a large open source software system, consisting of thousands of classes and tens of thousands of files - and all without breaking the existing functionality.

You will fix bugs in the codebase and add new features, as well as performing larger scale refactorings to maintain or improve on non-functionality properties of the system. You will perform all this using an industry strength tool set. We will complement the hands-on experience-based learning with an understanding of the core concepts underlying current notions of software engineering best

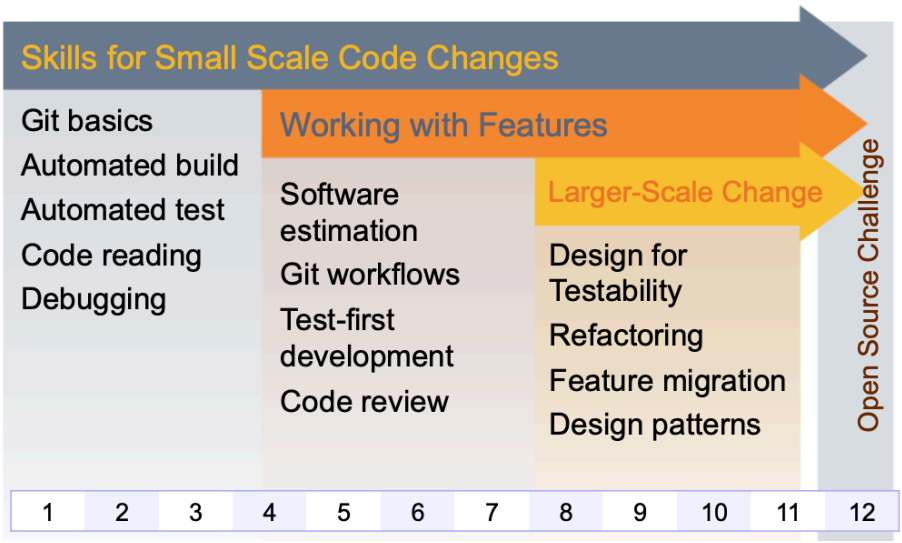

Figure 1: Course unit roadmap. This twelve week course will take you from small scale code changes (shown in grey), through to working with features (shown in orange) and on to larger-scale change (shown in yellow). We finish with an open source challenge in chapter **??**. The skills you will develop on this course are fundamental to modern software engineering.

practice. Volunteer mentors from local industry will help you to put your learning into context, and to understand the key importance of being not just a good coder, but a good software engineer.

This course unit detail provides the framework for delivery in 20/21 and may be subject to change due to any additional Covid-19 impact. Please see Blackboard / course unit related emails for any further updates.

0.2 Aims

This unit aims to help students appreciate the reality of team-based software development in an industrial environment, with customer needs, budget constraints and delivery schedules to be met. Through hands-on experience with an industry-strength development toolkit applied to a large open source software system, students will gain an appreciation of the challenges of green and brownfield software development, along with an understanding of the core software engineering concepts that underpin current best practice. Students will have the core skill set needed by a practicing software engineer, and will be ready to become productive and valuable members of any modern software team.

0.2.1 Learning outcomes

On successful completion of this unit, a student will be able to:

- make use of industry standard tools for version management, issue tracking, automated build, unit testing, code quality management, code review and continuous integration.
- write unit tests to reveal a bug or describe a new feature to be added to a system, using a test-first coding approach.
- explain the value of code reviews, and to write constructive and helpful reviews of code written by others.
- make use of basic Git workflows to coordinate parallel development on a code base and to maintain the quality of code scheduled for release.
- explain the role of software patterns (design and architectural) in creating large code bases that will be maintainable over the long term.
- explain why code that is easy to test is easy to maintain, and make use of test code smells in identifying and correcting design flaws (design for testability)
- apply basic software refactorings to maintain or improve code quality
- explain the challenges inherent in cost estimation for software development, and create defensible estimates with the help of work breakdown structures

0.3 Recommended reading

The following books are recommended course texts, they are all available from the University of Manchester library if you clickthrough to the references:

1. Pro Git (**?**)
2. The pragmatic programmer : from journeyman to master (**?**)
3. Effective unit testing : a guide for Java developers (**?**)
4. Clean code : a handbook of agile software craftsmanship (**?**)
5. The clean coder : a code of conduct for professional programmers (**?**)
6. Beginning software engineering (**?**)

These and any other references cited are listed in chapter **??**.

0.3.1 Requirements

The compulsory pre-requisites for this course are the first year programming units:

1. COMP16321: Programming 1
2. COMP16412: Programming 2

0.3.2 Overview of course

The following is an outline of the topics covered in COMP23111.

- Team software development
- Software project planning and issue tracking
- Greenfield vs brownfield software development
- Git best practices and common Git workflows
- Automated build tools and release management
- Automated unit, integration and acceptance testing
- Test code quality and test coverage tools
- Continuous integration and testing tools
- Best practices and tool support for code review, including source code quality tools
- Design patterns and common architectural patterns
- Design for testability
- Refactoring for code quality
- Safely migrating software functionality
- Basic risk management techniques
- Working with open source software systems

0.4 Using the lab manual

We expect that the web-based version of this manual will be the one you'll use most at software-eng.netlify.app. You can search, browse and link to anything in the manual. It was last updated on 25 September, 2021.

However, if you'd prefer, the manual is also available as a single pdf file softeng.pdf and an epub as well softeng.epub. Having said that, the content is optimised for viewing in a web browser, so while the pdf and epub are OK, the web version is the best.

0.5 Contributing to this manual

If you'd like to contribute this laboratory manual, we welcome constructive feedback. Once you're familiar with git and markdown you can github.com/join and:

- Raise new issues at github.com/dullhunk/softeng/issues/new
- Click on the `Edit this page` link, which appears on the bottom right hand side of every page published at software-eng.netlify.app when viewed with a reasonably large screen (not a phone)
- Contribute at github.com/dullhunk/softeng/contribute and help with existing issues at github.com/dullhunk/softeng/issues
- Fork the repository, make changes and submit a pull request github.com/dullhunk/softeng/pulls. If you need to brush-up on your pulling skills see makeapullrequest.com
- From the command line, clone the repository and submit pull requests from your own setup:

```
git clone https://github.com/dullhunk/softeng.git
```

Most of the guidebook is generated from RMarkdown, that's all the `*.Rmd` files. So markdown files are the only ones you should need to edit because everything else is generated from them including the `*.html`, `*.tex`, `*.pdf` and `*.epub` files.

0.6 Acknowledgements

This course has been designed, built and written by Suzanne Embury at the University of Manchester with support from a team of academics, industry club members, support staff, graduate teaching assistants (GTAs) and summer students including (in alphabetical order):

Muideen Ajagbe, Mohammed Alhamadi, Aitor Apaolaza, Gerard Capes, Martina Catizone, Sarah Clinch, Peter Crowther, Sukru Eraslan, Gareth Henshall, Duncan Hull, Caroline Jay, Nikolaos Konstantinou, Kamilla Kopec-Harding, Kaspar Matas, Chris Page, Dario Panada, Steve Pettifer, Liam Pringle, Julio Cesar Cortes Rios, Sara Padilla Romero, Viktor Schlegel, Stefan Strat, Jake Saunders, Federico Tavella, Mokanarangan Thayaparan, David Toluhi, Karl Tye and Markel Vigo.

Academic staff on the course for 2021/22 include:

- Mercedes Argüello Casteleiro
- Thomas Carroll
- Suzanne Embury
- Duncan Hull
- Sandra Sampaio
- Anas Elhag

We'd also like to thank all the 1,500+ students who have done the course since its first iteration in 2016 and given us feedback on how to improve.

Thanks also to our industrial mentors, the Institute of Coding (IoC) instituteofcoding.org and the Office for Students (OFS) for their ongoing support.

0.7 Licensing

See the full license (CC-BY-NC-ND) for details.

0.7.1 Your privacy

This site is hosted on netlify.com, see the netlify privacy policy. This site also uses Google Analytics to understand our audience better which is compliant with the General Data Protection Regulation (GDPR). If you want to, you can opt out using the Google Analytics Opt-out Browser Add-on.

Some of these services use cookies. These can be disabled in your browser, see allaboutcookies.org/manage-cookies

So now that we've dispensed with the formalities, you can start using this laboratory manual.

Expectations

While you are studying on this software engineering course, you are part of a team and a wider community:

- Your immediate team members
- The community of all second year students

Your learning community is supported by a group of graduate teaching assistants (GTAs), mentors and academic staff.

0.8 Expectation engineering

It's important that you know we expect of you and what you'll get in return. That's what this page describes.

0.8.1 Our expectations of you

We need you to read ALL of the messages we send you via GitLab, Jenkins, Microsoft Teams, the Piazza forum, emails and in this guidebook. Before you ask for help, make sure you have Read The Friendly Manual(s). RTFM.

0.8.1.1 Workshops vs. Team study

There are two main sessions:

1. Team study sessions
2. Workshops

Team study sessions are for you to get together as a team to work alongside each other. You can also get help from GTAs and staff on coursework.

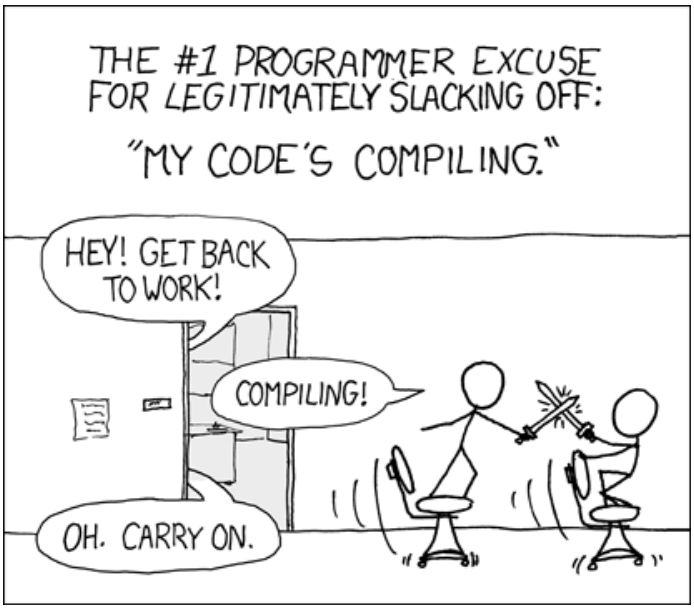

Figure 3: There are legitimate reasons for slacking off, such as compiling (and building) your code. Falling out with your team members, not returning their messages or just being busy doing other things are not legitimate reasons for letting your team down. Compiling (xkcd.com/303) by Randall Munroe is licensed under CC BY-NC 2.5

Each workshop has a specific theme that we need you to focus on. This means **we won't discuss coursework with you during the workshops**, otherwise we risk not getting through workshop material.

0.8.1.2 Physical vs online sessions

For the live (physical) sessions you'll need to be in the appropriate lab in the Kilburn building. For online sessions (e.g. some marking and mentoring) it is especially important that you turn up on time by being at a computer with access to:

- A working pair of headphones
- A working microphone that you can mute if you're somewhere noisy
- A webcam (ideally) but see section **??**

Please note, this may mean the best place to work is *not* necessarily the Kilburn building. Go and find a quiet spot, use your laptop (if you have one) or use your phone (there are good mobile clients for Teams) or work from home. It will really help if *at least* one of your team is at a desktop computer.

We have deliberately scheduled online activities so they aren't immediately after physical activities (like a lecture) so that you have time to get setup BEFORE the meeting starts.

0.8.2 What your team expects of you

For this course to run smoothly your team will expect that you:

- Turn up to all the bi-weekly team study sessions, **especially the marking sessions**
- Participate in the all workshops
- Contribute to your team by:
  - Respecting your team members, no bullying. Assume good faith by default. It's your responsibility to make your team work. Team work makes the dream work.
  - Getting along with your team members. You may not like all of them (that's life) but your team members are crucial to your teams success. While you can get away with being a "lone wolf" coder on other course units, (see figure **??**), you are now expected to behave like a sociable engineer as part of a professional team
  - Encouraging people who might be slacking off to make contributions, see figure **??**
  - Communicating with your team if you have difficulty contributing

Figure 4: Normally a social pack animal, wolves sometimes act alone. While being a lone wolf on other course units may be a reasonable strategy for studying, it won't work well for this one. Don't be a lone wolf because sociable teams usually make better software than loners. CC-BY-SA Image of Winter wolf by ForestWander.com on Wikimedia Commons w.wiki/45Vj

- Reporting issues where necessary, either to a GTA or academic member of staff

What do you get in return for our expectations and those of your team?

0.8.3 What to expect of GTAs

This course is supported by a team of Graduate Teaching Assistants (GTAs), they are here to help you. They have lots of other people to help too, so please treat them with respect. If you're waiting for support from a GTA, make sure you've Read The Friendly Manual, see section **??**.

Our GTAs have Read Their Friendly Manual to (the GTA wiki) so they will know how to help you, or can quickly find out how to. They won't give you the answer, but will be able to help you find your own way.

The GTAs have scheduled marking sessions that we expect them to stick to. The second year timetable is incredibly crowded, and the team study sessions are the **ONLY TIMES IN THE WEEK** when we can guarantee that everyone in your team is available.

0.8.4 What to expect from mentors

You have been assigned a mentor who will meet with you online for two one hour meetings, see chapter **??**. These meetings are a bit like code review meetings, they have access to your private code repository and can see what your team is up to.

Our mentors are all professional software engineers, who can give you advice on how to manage the process of making better software. so please treat them with the respect they deserve. They have volunteered to help by sharing their engineering wisdom with you and taken time out of their busy schedules to do so.

0.8.5 What you can expect from academic staff

The academic staff on this course include Suzanne Embury, Anas Elhag, Duncan Hull, Thomas Carroll and Sandra Sampaio. We're here to help but please remember, we're often very busy and have other teaching commitments too besides this course. We're here to ensure that the course runs smoothly and we aim to give feedback on coursework to you within the two week window.

0.9 Your camera

We would normally expect participants in small meetings (not large ones like lectures) to turn their cameras on but we understand that there are good reasons why people may not be willing/able to and won't explicitly ask you to.

0.9.1 Camera on?

There has always been a question around whether to turn cameras on during online meetings but it is even more obvious with online meetings becoming the norm rather than the exception. There is a direct benefit in using cameras in small, personal meetings where many of us make use of visual cues to aid the flow of conversation – at the very least it's easier to identify who is talking. Additionally, it can help people get along – people might feel more 'listened to' if they can see somebody listening and your team will find it easier to remember names etc if they have a face to match the names to.

0.9.2 Camera off?

There are lots of legitimate reasons why you might turn your camera off. Most obviously, if you don't have access to a camera. But you may also be in an environment which you prefer others not to see, you may have anxiety around the issue, or your connection might be too slow. There are many other perfectly reasonable reasons for you not to put your camera on and you should not feel pressured to do this. If you simply say "Sorry, I can't turn my camera on today" then nobody will ask any further and they should never explicitly ask you to turn it on.

0.9.3 Being appropriate

You should already be treating online meetings like physical ones e.g. turning up on time, being prepared, listening, engaging etc. Similarly, if people can see you then you should ensure you are wearing appropriate clothes (wearing clothes is the absolute minimum here!) and in an appropriate place (the bathroom is probably not appropriate) as you would for a physical meeting.

0.9.4 Respecting others

If other people have decided to turn their cameras on then we ask that you show them respect by not recording anything without their explicit permission. We won't touch on the legality of this as we believe that basic respect for each other

should be enough to prevent any issues. You will take part in larger meetings where recording may be standard and in such cases this should be made explicit.

(Thanks to Giles Reger and Sarah Clinch for the text above)

Weekly timetable

The weekly schedule for autumn 2021 is shown in table **??**, based on timetables.manchester.ac.uk, see also manchester.ac.uk/discover/key-dates key dates.

- Other than the introductory lecture in week 1, there are no lectures. Instead we have workshops which are more like labs and may contain mini-lectures
- Workshops are on Tuesday or Friday afternoon depending on your lab group
- Team study sessions are on Tuesdays at 10am and Thursday at 11am

0.10 Configuring

Events in the week starting 20th September:

1. Configuring, you will receive a notification from gitlab so that you are ready for week 1

0.11 Automating

Events in the week starting 27th September:

1. **Team Study Tuesday**: Work on individual coursework 1 described in chapter **??**
2. **One off lecture** to introduce the course unit at 9am on Wednesday 29th September, Simon Engineering building check timetables.manchester.ac.uk
3. **Workshop**: Automated build and test with Duncan Hull
4. **Team Study Thursday** Work on individual coursework 1 described in chapter **??**

Table 1: The weekly schedule for this twelve week course, please note we are using the week numbering from the timetables.manchester.ac.uk where week zero is welcome week, and week one is the first teaching week

Week no.	Subject	Deadlines
0: 20th Sept	Configuring, see section **??**	
1: 27th Sept	Automated build and test, see see section **??**	IndCwk2, Fri 1st Oct,
2: 4th Oct	Reading large codebases, see see section **??**	
3: 11th Oct	Debugging, see see section **??**	IndCwk2, Fri 15th Oct
4: 18th Oct	Cost estimation, see see section **??**	
5: 25th Oct	Test first development, see see section **??**	TeamCwk1, 29th Oct,
6: 1st Nov	Reading week see see section **??**	
7: 8th Nov	Git workflows, see see section **??**	
8: 15th Nov	Software refactoring, see see section **??**	
9: 22nd Nov	Design for testability, see see section **??**	
10: 29th Nov	Design patterns, see see section **??**	TeamCwk2, Fri 3rd De
11: 6th Dec	Risk management and practice exam, see see section **??**	
12: 13th Dec	Open source challenge, see see section **??**	

5. **Coursework deadlines**: Individual individual coursework 1 can be premarked (automatically) if you submit by **6pm Tuesday 28th September** and finally by **6pm on Friday 1st October**

0.12 Reading

Events in the week starting 4th October:

1. **Team Study Tuesday**:
2. **Workshop**: Reading large code bases
3. **Team Study Thursday**
4. **Coursework deadlines**:

0.13 Debugging

Events in the week starting 11th October:

1. **Team Study Tuesday**:
2. **Workshop**: Debugging

3. **Team Study Thursday**
4. **Coursework deadlines**:

0.14 Estimating

Events in the week starting 18th October:

1. **Team Study Tuesday**:
2. **Workshop**: Cost estimation with Duncan Hull
3. **Team Study Thursday**
4. **Coursework deadlines**:

0.15 Testing

Events in the week starting 25th October:

1. **Team Study Tuesday**:
2. **Workshop**: Test first development
3. **Team Study Thursday**
4. **Coursework deadlines**:

0.16 Pausing

Events in the week starting 1st November (reading week). Take a break if you're ahead, or catchup if you've fallen behind. There are no activities in reading week.

1. **Team Study Tuesday**:
2. **Workshop**: Git workflows
3. **Team Study Thursday**
4. **Coursework deadlines**:

0.17 Workflowing

Events in the week starting 8th November:

1. **Team Study Tuesday**:
2. **Workshop**:
3. **Team Study Thursday**
4. **Coursework deadlines**:

0.18 Refactoring

Events in the week starting 15th November:

1. **Team Study Tuesday**:
2. **Workshop**: Refactoring
3. **Team Study Thursday**
4. **Coursework deadlines**:

0.19 Testing

Events in the week starting 22nd November:

1. **Team Study Tuesday**:
2. **Workshop**: Design for testability
3. **Team Study Thursday**
4. **Coursework deadlines**:

0.20 Patterning

Events in the week starting 29th November:

1. **Team Study Tuesday**:
2. **Workshop**: Design patterns
3. **Team Study Thursday**
4. **Coursework deadlines**:

0.21 Managing

Events in the week starting 6th December:

1. **Team Study Tuesday**:
2. **Workshop**: Risk management and practice exam
3. **Team Study Thursday**
4. **Coursework deadlines**:

0.22 Challenging

Events in the week starting 13th December:

1. **Team Study Tuesday**:
2. **Workshop**: Open source challenge
3. **Team Study Thursday**
4. **Coursework deadlines**:

0.23 Tools

We'll be using the following tools:

- Team study sessions take place on Microsoft Teams, login using your `@student.manchester.ac.uk` email address at teams.microsoft.com or download a native teams client
- Other course materials (coursework submission, slides and videos) can be found on blackboard online.manchester.ac.uk

Part I

Weekly Workshops

Chapter 1

Building and testing

1.1 Introduction

In this workshop, we will be building and testing a system called Marauroa. We will look at some essential processes for working on an existing team-developed software system. We'll be assuming that, after this workshop, you are capable of carrying out the following tasks for yourself, without needing much guidance:

- Acquire the right version of the source code on which to work.
- Create an executable version of the source code using an automated build tool.
- Test the system, prior to making changes.
- Use a test suite to find functional regression in the system.

- Run a piece of software consisting of multiple distributed subsystems.

In this, and some later workshops, we'll be working with the code of the *Marauroa games engine* for constructing on-line multi-player games.

Marauroa is an open source *framework* and engine to develop games. It provides a simple way of creating games on a portable and robust server architecture. Marauroa manages the client server communication and provides an object orientated view of the world for game developers. It further handles database access in a transparent way to store player accounts, character progress and the state of the world.

You should already have begun to practice some of these skills, through the GitLab Access Check activity. In this workshop, we will build on that activity to carry out these basic skills on a large open source software system. During the workshop, you will:

1. Use an IDE to clone a local copy of the Marauroa repository.
2. Build executable versions of the client and server components, using the Ant build tool.
3. Run the test suite provided for Marauroa
4. Use a code coverage tool to assess the strength of the test suite.
5. See how the test suite can help us pinpoint errors in the code.

You may work at your own pace, but you should try to complete step 4 by the end of the workshop if you can. You will need to finish the exercise in your own time if you don't manage it in the workshop, as you'll need to use these techniques for the team coursework. **If you are not up-to-speed with them, then you could slow your team down.**

1.2 Acquiring Marauroa

First, you'll need to acquiring a local copy of the Marauroa Project.

1.2.1 Run the IDE

The Department provides a range of Integrated Development Environments (IDEs) for use by students. You are welcome to use any of these IDEs to carry out the work for this workshop. However, we are only able to provide technical support for Eclipse, specifically `2020-03`. If you do want to use one of the other IDEs, we will do our best to help should you get stuck, but we can't guarantee to be able to fix all problems. At the bare minimum, you should feel confident that you can do all the tasks listed in the introduction in your chosen IDE, before you finalise the decision.

The instructions that follow assume you are using `2020-03` on the Department of Computer Science Linux image or on the Linux Mint VM provided by the Department.

You can start Eclipse from the Applications menu (under Programming) or from the command line, by issuing the command:

```
/opt/eclipse-2020-03/eclipse &
```

1.2.2 Select the Workspace

Eclipse calls a folder containing one or more Eclipse projects a `workspace`. On start-up, Eclipse will ask you which workspace you want to use for the session. You can either accept the default location or use the File Browser to locate or create a different one. (Depending on when you do this workshop, you may

already have created a workspace for the GitLab Access Check activity. You can either choose to use the same workspace for this activity, or create a new one.

If you choose to create a new workspace, Eclipse will show the Welcome View when it loads. Uncheck the box at the bottom right of the window (labelled `Always show Welcome on start up`) and close it down, as we do not need this view for this workshop. (You can get it back whenever you want by selecting the `Welcome` option from the `Help` menu.)

1.2.3 Organise Workspace

You'll need to organise your main Eclipse workspace window and you should now see a window that looks something like figure **??**.

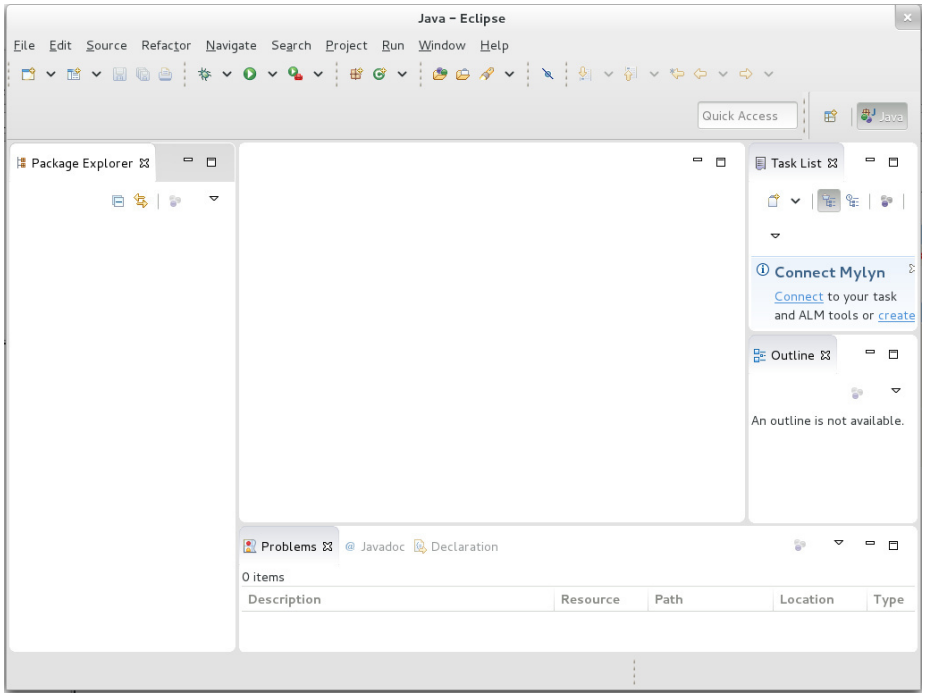

Figure 1.1: Your main eclipse window should look something like this

If you used the same workspace you created for the individual coursework exercises, you'll see the project for that in the Package Explorer view. If you used a new workspace, it will be empty like the one shown above.

This is the standard layout for working on Java projects. The central empty space is where we will use the various Eclipse editor tools and views to work

on individual files. It is empty at the moment, as we are not working on any specific file. Around it are a number of other views. We'll talk about the main ones and what they tell us later.

I find this screen rather cluttered, and would immediately delete all the views I don't need regularly, to free up space for the ones I do, and move the views I do use to more convenient locations. You might want to do the same. You can experiment with moving the views around by clicking and dragging on their tabs. Delete any views you don't think you'll need, but **make sure you keep the Package Explorer view, the Outline view and the Problems view open**, as we'll be making use of those very soon.

Note that you can always get any views you delete back again, using the `Window > Show View` menu option.

1.2.4 Create a New Project by Cloning

Next, we're going to pull down (git clone) the public Marauroa source code into a local repository where we can work on it. You've already had experience of working with Git from the command line. In this course unit, we ask you to use your IDE for (at least) your basic interactions with Git and GitLab. This will help you to understand the strengths and weaknesses of both approaches, if you are not already familiar with them.

The first step is to ask Eclipse to import the Marauroa project for us, from a public Git repository.

Select the `File > Import` menu option. Then choose `Git > ImportFromGit` shown in figure **??**

You can either double-click on `Projects from Git`, or single-click on it and press `Next`.

A dialogue box appears showing the two ways in which you can import a project from Git. We're going to **clone a project from a URI**, so select that option shown in figure **??**

Next, we need to tell Eclipse which URI to clone from. The team behind Marauroa have set up their own Git server, which we'll connect to anonymously. Enter the following into the URI field:

```
git://git.code.sf.net/p/arianne/marauroa
```

Eclipse **Should fill in the rest of the fields automatically**. If it doesn't, it's likely that something went wrong when copying the link from this PDF: try typing it instead. Check that your dialogue looks like figure **??** before proceeding.

If everything looks okay then select `Next`.

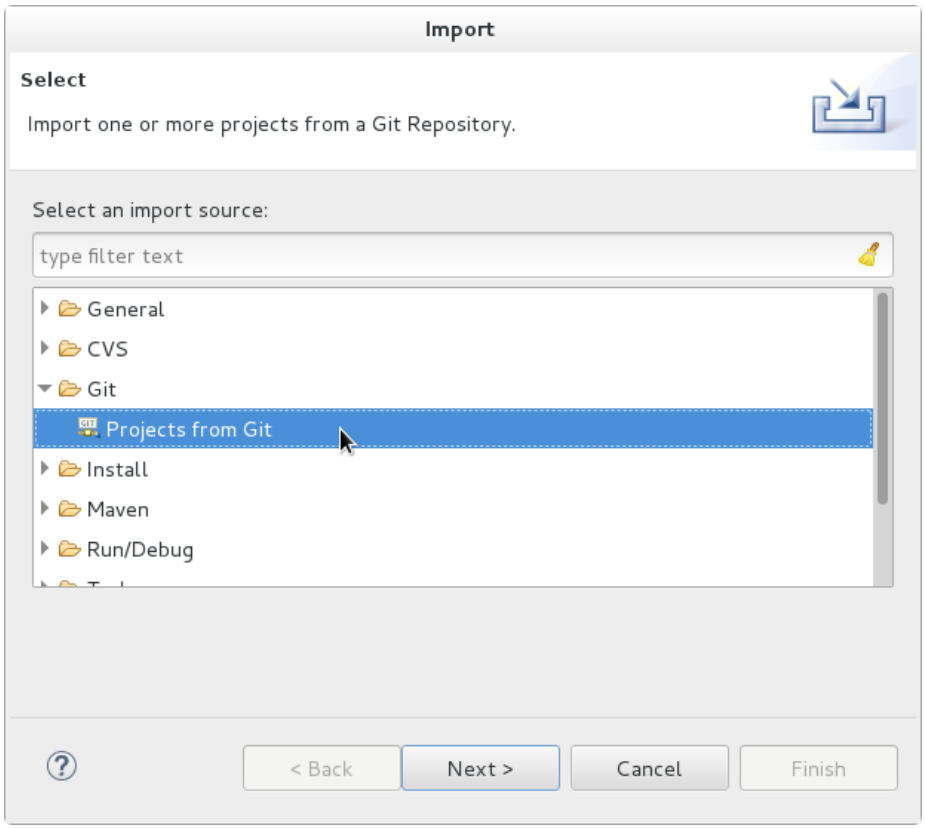

Figure 1.2: Your main eclipse window should look something like this

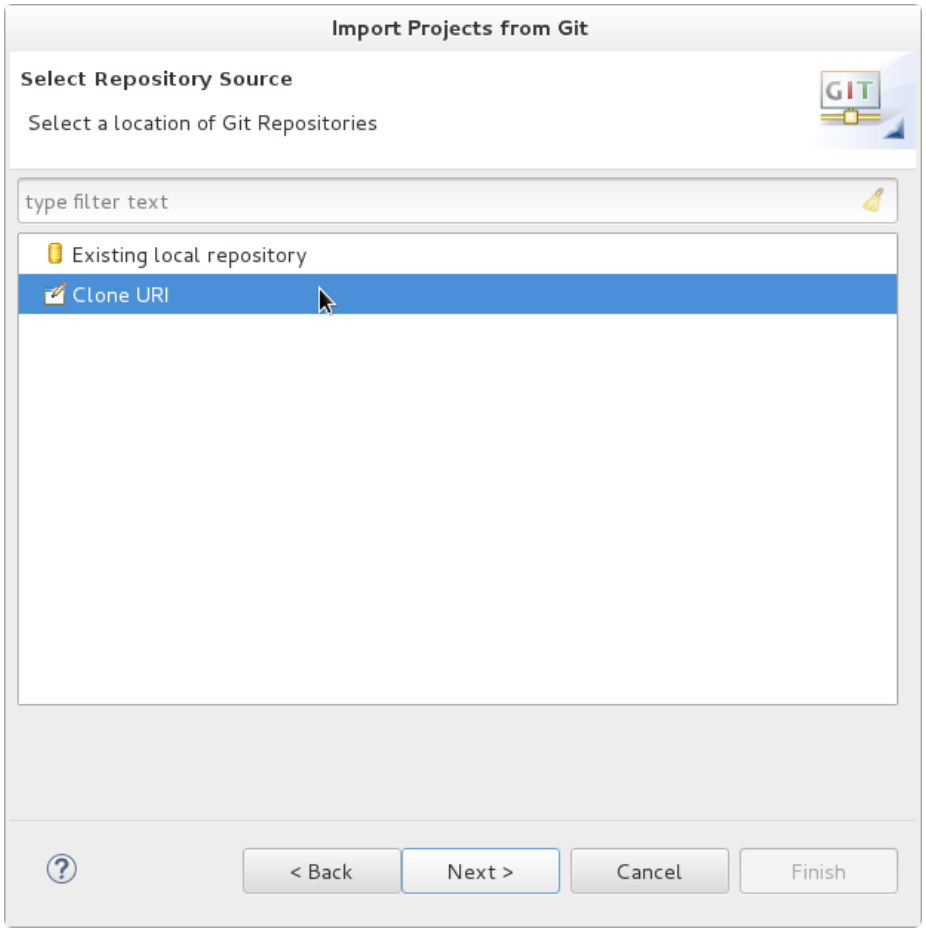

Figure 1.3: Your main eclipse window should look something like this

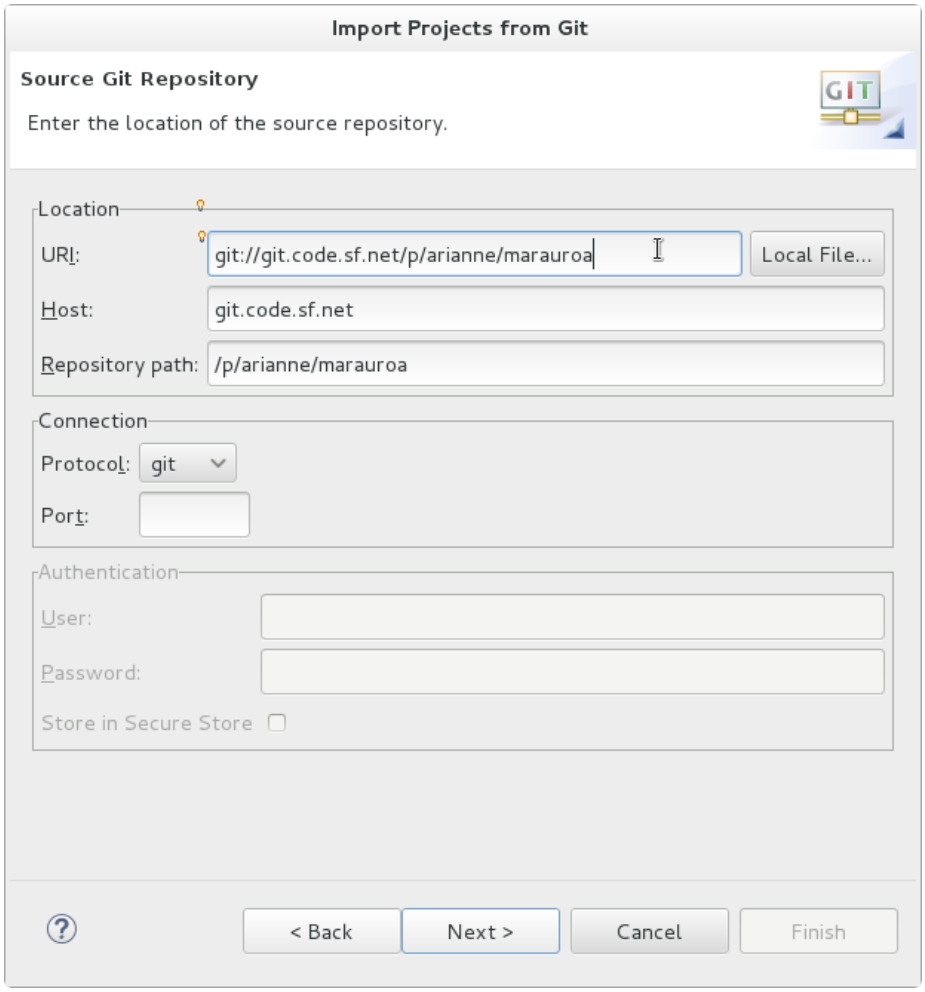

Figure 1.4: Your Import Projects from Git dialogue box should look like this

Does your Clone Attempt Fail With An Error?

If so, the Arianne project Git server may be temporarily down. If you can't clone using the URI given above, you can try using this GitHub repository URI instead:

```
https://github.com/arianne/marauroa.git
```

Eclipse will now communicate with the remote Git repository specified in the URI. It will ask us which branches we want to work with locally, that is, which branches we want to create local remote tracking branches for. Note that this is not the same as asking us which commits we want to include in our clone. A standard Git clone will always include all the commits in the cloned repository, regardless of which branches we select here. And it is not asking us which remote branches we want to have in the repository. Again, a standard Git clone will include all the remote branches by default. The question Eclipse is asking here applies only to the question of which tracking branches should be created in the clone.

We're not going to be making any serious changes to the Marauroa code base in this workshop, so we will just ask for a remote tracking branch to be created for the `master` branch of the repository, see figure **??**.

Make sure that the `master` branch is selected, and press `Next`.

As in the GitLab Access Check activity, we need to tell Eclipse where we want the cloned repository to be stored before it can issue the Git command to create it see figure **??**

You can use the default location suggested, or you can use the Browse button to use the file selector to create a new directory in a different location. Here, I've followed the standard convention of putting the repository inside my personal `git` folder.

When you have selected your preferred location, select `Next`.

Eclipse now issues the commands to clone the project.

The next step is to import the Marauroa project from your local Git repository into Eclipse, so you can start to work on it.

What is a project in this context?

One of the confusing things about IDEs when we first start to use them is the notion of a `project`. When we code from the command line, we tend to organise our work in directories. Sometimes these directories relate to specific tasks we are carrying out (like coding up the solution to a lab exercise) and sometimes they relate to the structure of the code we are creating (like different directories for source code and object code, or for libraries or documentation).

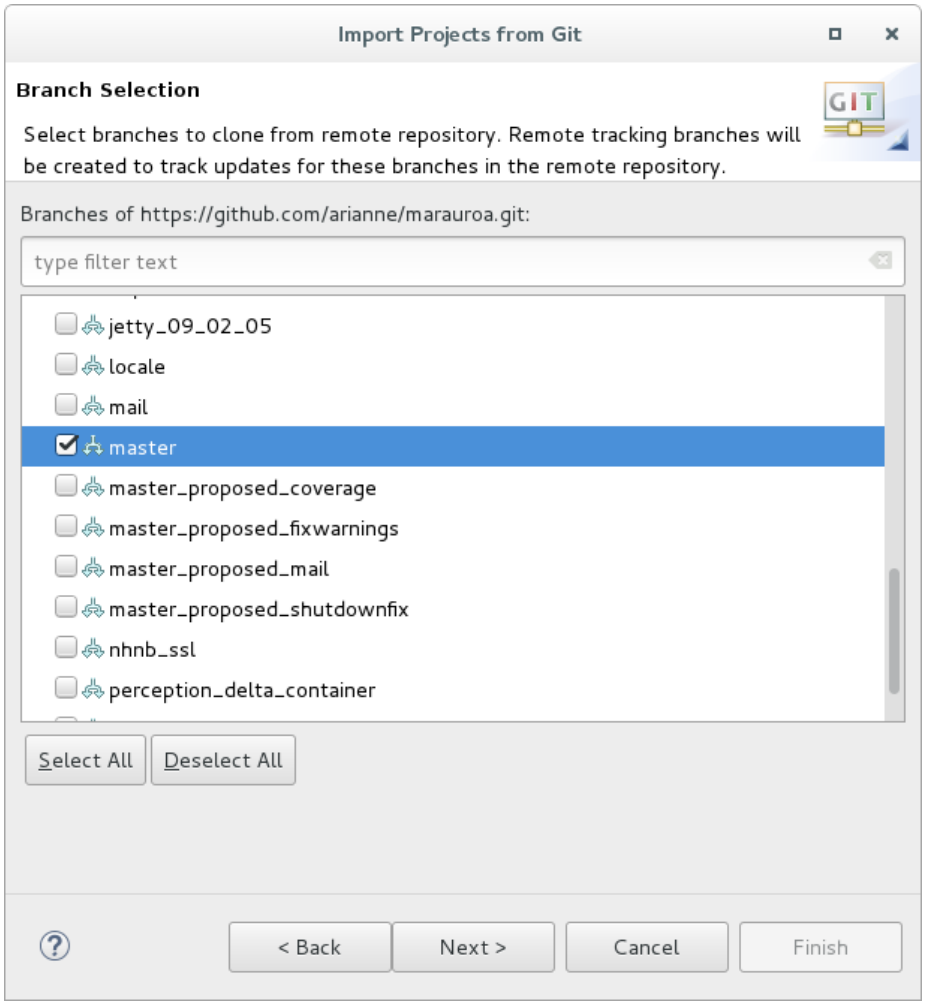

Figure 1.5: Take a look at the list of branches contained in the project, by scrolling up and down the list. You'll see that the Marauroa project uses separate branches to describe specific releases, as well as other development branches. Another common approach is to have a single release branch and to use tags to distinguish specific releases on that branch.

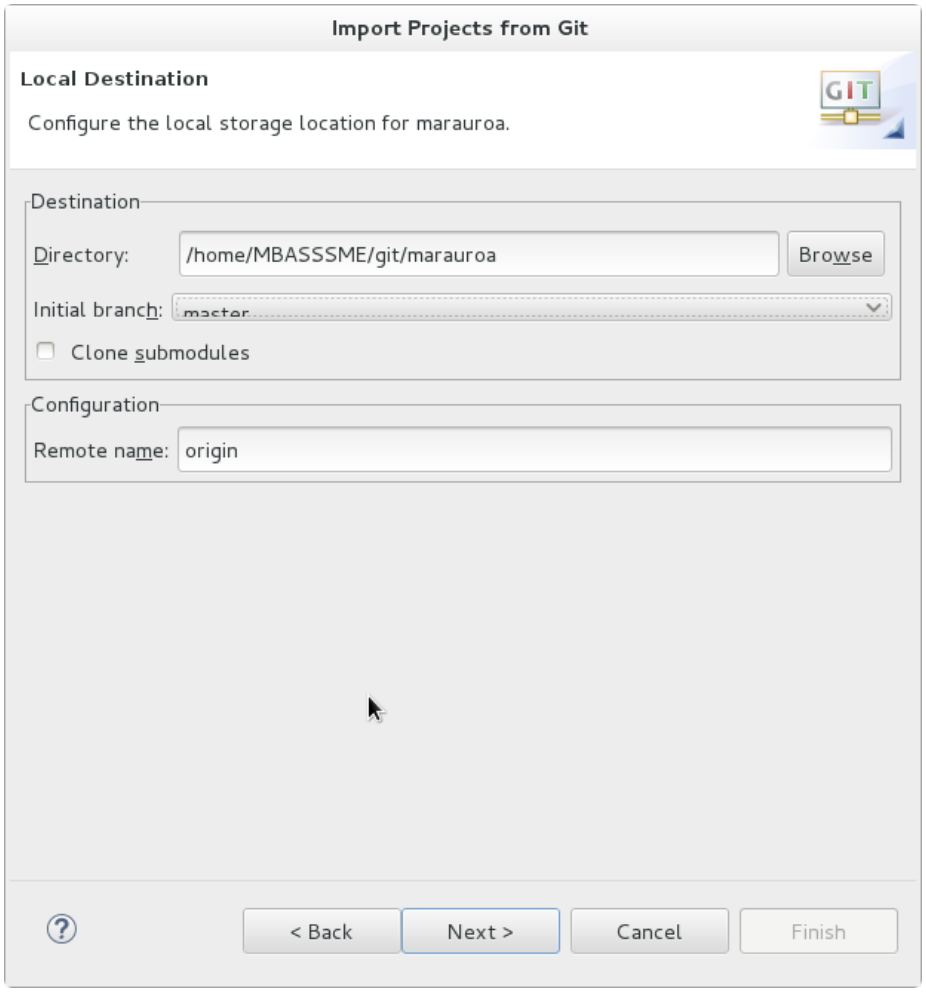

Figure 1.6: Cloning from git dialog box

We use directories for all these purposes when we code in an IDE as well, but in order to be able to support us well, the IDE needs to know the *root* directory of a piece of software that we are building. That way, it can perform useful tasks for us, like automatically setting the `classpath` for us, and automatically compiling code and reporting on errors while we type. This root directory is typically referred to as a *project*. IDEs use the concept of a project as a means of recording metadata about the project. For example, Eclipse will remember that a specific project is a Java project, and will then know to apply the set of tools appropriate to Java projects, and not (for example) tools relating to Ruby or Python.

As in the GitLab Access Check activity, we have to tell Eclipse which wizard to use to import the project for us. Since the Marauroa team uses Eclipse, we can use the wizard that looks for existing Eclipse projects in the repository, see figure **??** If we were loading a project built in another IDE, we would need to use one of the other wizards.

Click on `Next` when the correct wizard has been chosen.

Eclipse will now scan the local Git repository looking for anything that it recognises as an Eclipse project. It looks through all the folders, searching for the metadata files that Eclipse creates and stores in the root directory of a project. In this case, it finds just one (called `newmarauroa`) see figure **??**

Since there is just one project in the repository, we have an easy decision here. Click on the `newmarauroa` project to select it, and then click on `Finish` (finally!).

Eclipse can now import the project into your workspace. When that is done, you'll be taken back to the main Eclipse work screen (strictly speaking, we're taken back to what Eclipse calls the 'Java Perspective'). You should see that a project has appeared in the Package Explorer view, and that the Problems view has now been populated with information see figure **??**

1.2.5 Checkout a Specific Commit

Although we asked for the `master` branch to be checked out locally when we cloned the repository, we are actually going to be working with a different commit, one that is not pointed to by `master`. This is partly to make sure everyone in the workshop uses the same commit for the exercise, even if `master` gets updated between the creation of these notes and the running of the workshops. But it is also to give you confidence in working with non-head commits (that is, commits that are not pointed to by a branch or tag).

For this activity, we are going to work with the commit with the short SHA of `f30e098`.

The easiest way to check out a commit, branch or tag from within Eclipse is to use the History View. To open it, right click on the `newmarauroa` project name

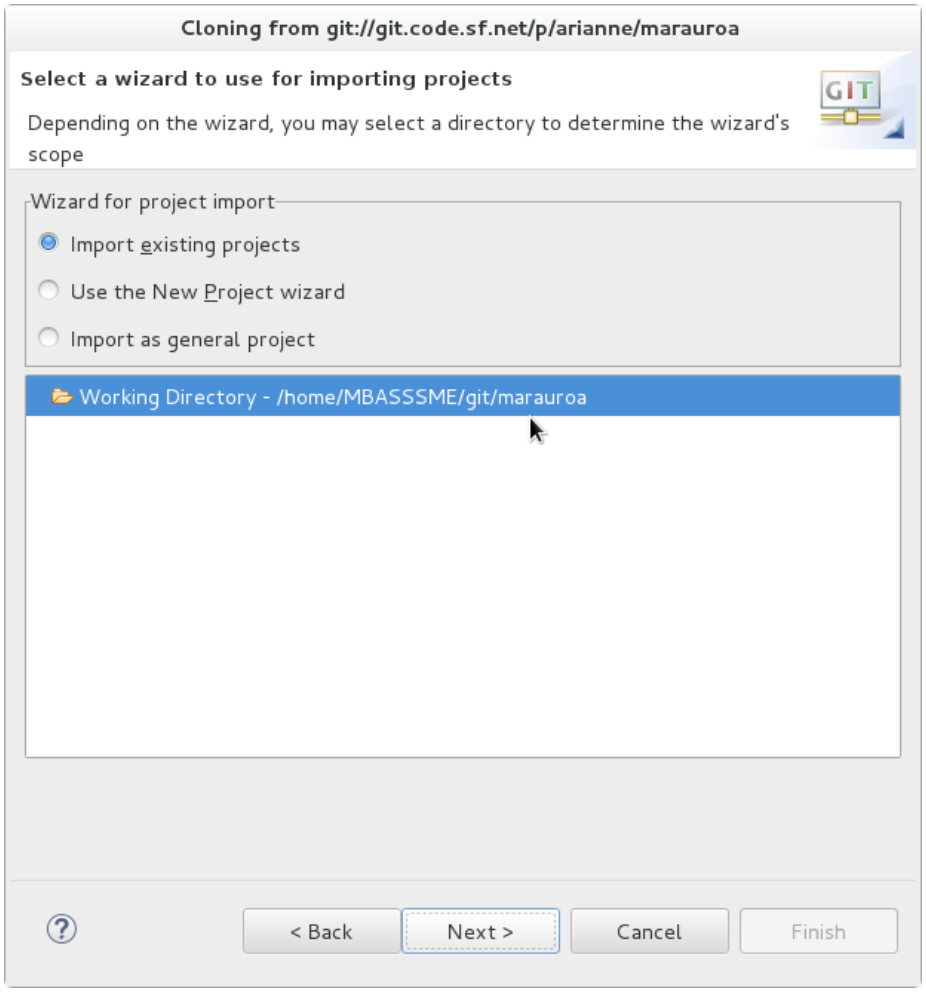

Figure 1.7: Cloning from git dialog box

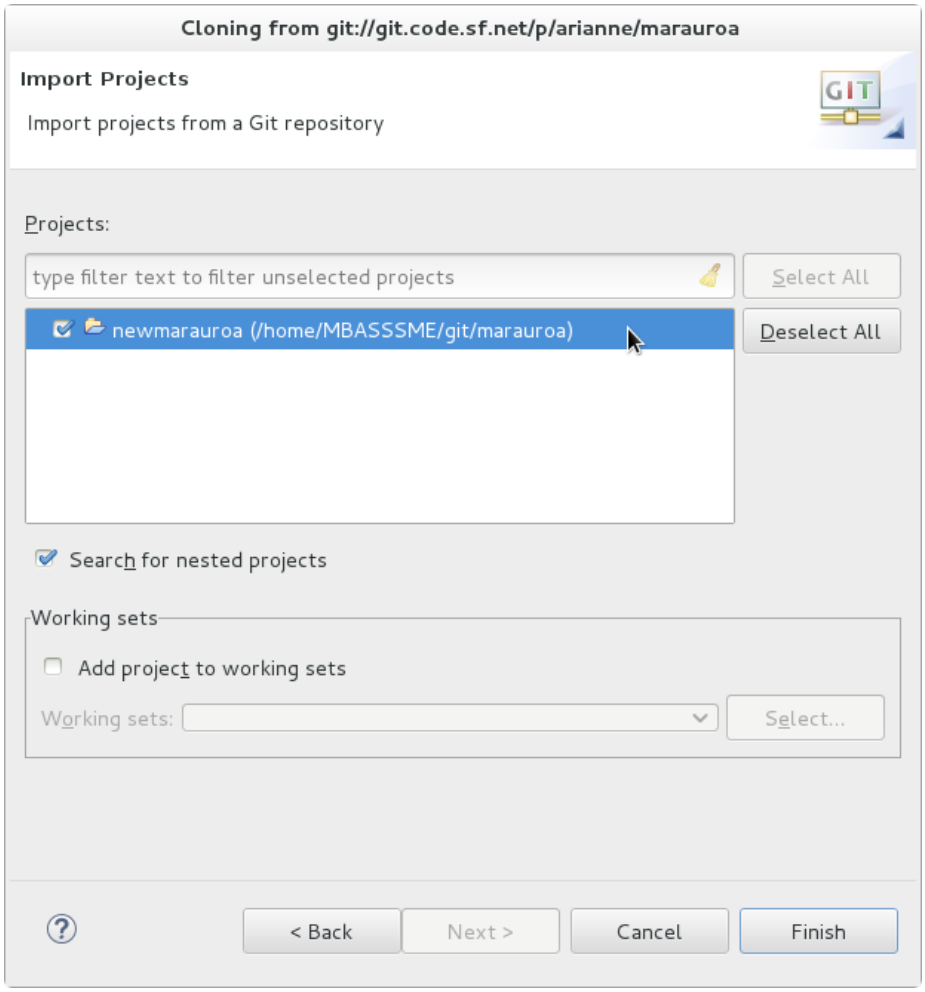

Figure 1.8: Import projects from a git repository and the newmarauroa project

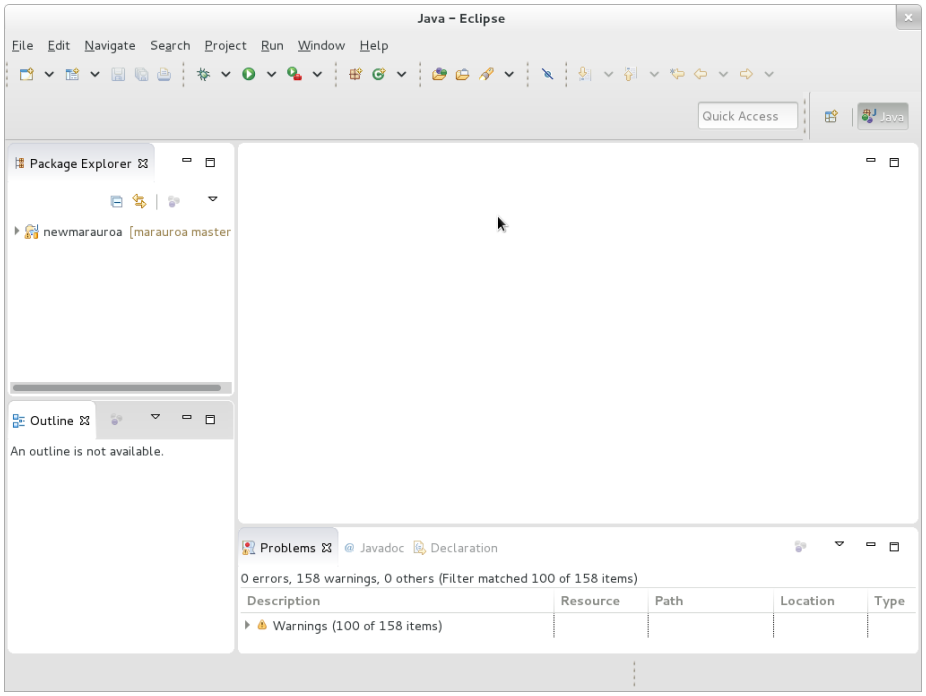

Figure 1.9: Import projects from a git repository and the newmarauroa project

in the Package Explorer view. Select `Team` > `Show in History` from the menu that appears. The History View shown in figure **??**, should now be visible in the bottom panel of your Eclipse window. You may wish to double click on the view tab to expand it, so that the contents are more easily seen.

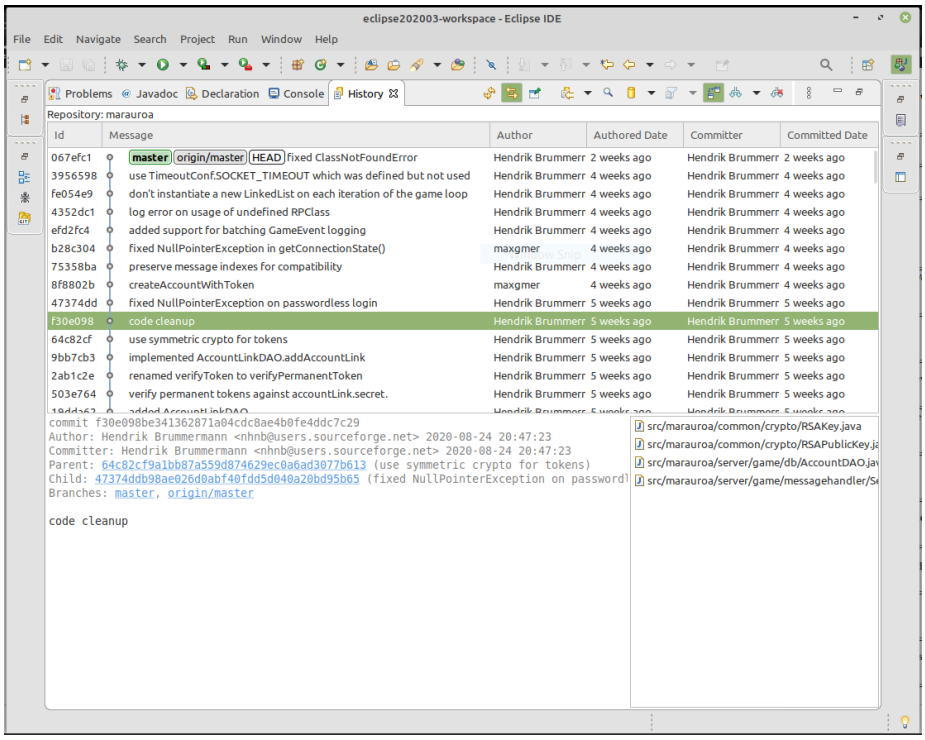

Figure 1.10: In this view, you should now see the most recent part of the network of the Marauroa project repository. You can scroll down to see the full commit log. As you can see, the history is significantly more complex than the simple repository we looked at in the GitLab Access Check. Marauroa has been under development since 2003, and its history reflects its age. Note that your view of the repository may be a little different than that shown in the screen shot. We are working with a live repository, and new commits are being made on a regular basis.

Look for the commit with SHA `f30e098`. It should have the (not terribly helpful) commit message `code cleanup`. Right click on it, and select `Checkout` from the menu that appears.

At this point, Eclipse will warn you that you are in a `detached HEAD` state shown in figure **??**

This just means that we have checked out a commit that is not pointed to by any current branch or tag. The `HEAD` in Git is the currently checked out commit.

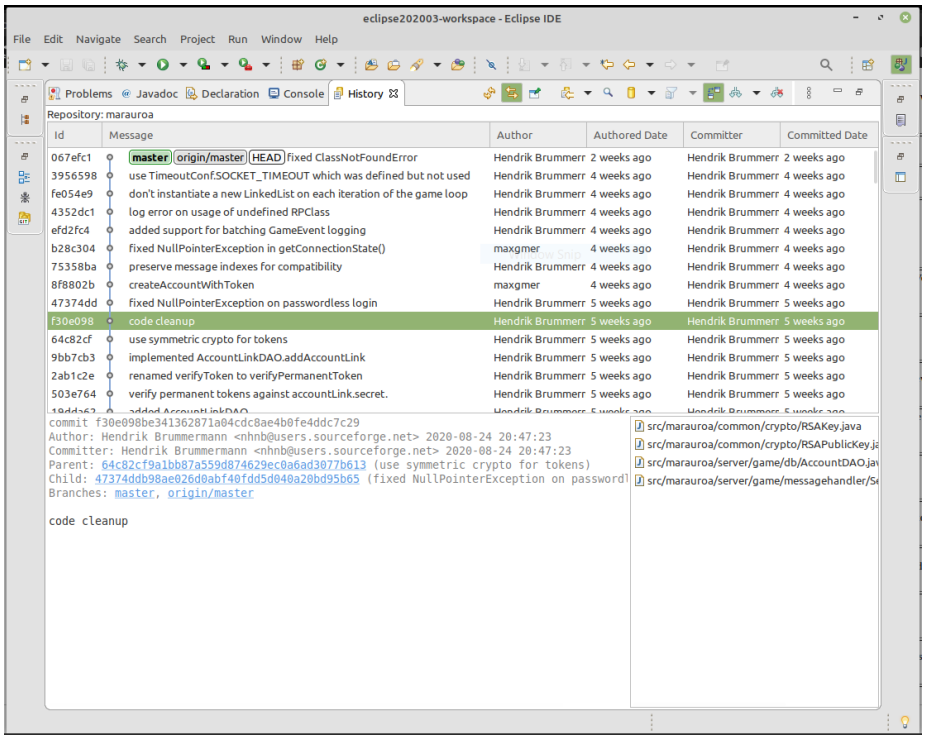

Figure 1.11: A warning of the detached HEAD state which reads: "You are in the `detached HEAD` state. This means you don't have a local branch checked out. You can look around but it's not recommended to commit changes. The reason is that these commits would not be on any branch and would not be visible after checking out another branch."

Eclipse (and Git) are warning us about this because any changes we make and commit from this point will also not be pointed to by any branch or tag (unless we create one specifically). In fact, they will be unreachable from any branch or tag, and so will be treated by Git as if they had been deleted. They will be scheduled for garbage collection, the next time that takes place. We're not going to commit any changes for this exercise, so we don't care whether the `HEAD` commit is detached or not. We can safely ignore this warning for now.

Checkout and Detached Heads

If you're interested to learn more about checking out a detached head, you could read this article: What's a "detached HEAD" in the Git FAQ

Press `OK` and double-click on the History View tab, to shrink the view back to its original size and location, now that we have finished working with it.

1.2.6 Explore your Project

You now have your own copy of the Marauroa project source to play with and look around a little. Take a few minutes to look around and explore what is inside it before moving on to the next step. Look at the way the contents of the project are organised into folders. Can you guess at the contents of each folder from its name?

Explore around some of the folders. Can you find some Java class files? What clues did you use to track them down?

Notice the icon that Eclipse has placed next to the project name. Quite a lot of information is packed into this small symbol. The folder symbol indicates that this is a project. The small J just above it indicates that this is a Java project. The small orange drum under the J indicates that this project is under version control. Eclipse also tells us the name of the Git repository the project is stored under, and which branch or commit of the project we current have checked out, in the text following the project name: `[marauroa f30e098]` (or similar). Finally, the small yellow road sign with the exclamation mark in the middle tells us that when Eclipse used its internal builder on the Java code in the project, it encountered some compiler warnings.

You might be surprised to see that the Marauroa team have released code that produces compiler warnings. Let's take a look at what the warnings are, using the Problems view. You'll notice that this view has already been populated with some information about the project, without us having to ask for it to be generated. IDEs will commonly provide services like this, performing key analyses of the project source and letting you know about problems without you having to explicitly request it. After all, if we have introduced a compilation error, we want to know about it as soon as it happens, and not much later when we finally remember to ask Eclipse to compile the code.

Because the Marauroa team have configured this project as a Java project, Eclipse already knows how to find the Java source files, and it uses its internal Java build tool to compile them. In fact, it will recompile every time we make even a small change to the code, as well as when we import new code. From the Problems view, we can see that this automatic compilation produced no compiler errors (good!) but 158 compiler warnings (eek!).

If you have time, you can take a few minutes to explore the compiler warnings generated, by clicking on the small triangle beside the warn `Warnings` in the Problem view. Take a look and see if you think these are serious problems or whether the Marauroa creators were making a reasonable decision not to fix them.

STEP 1 of 4 COMPLETED

You've now completed the first step, and have a code base to explore. But, that is only the beginning. Please proceed to the next step, where we'll look at how to **use the automated build scripts** provided by the Marauroa team to build an executable version of the Marauroa engine.

1.3 Building the Marauroa Engine

If we are going to make changes to an existing body of code, we have to be able to create an executable version of it. There is no point in making changes to source code if we can't actually run the new version of the code.

In this step, you're going to be introduced to the Apache Ant automated build tool, which is the tool chosen by the Marauroa team for use on their project. You'll learn how to use it to create executable code from the source we've just downloaded.

Note: we will not cover a full tutorial on the use of the Ant tool in this workshop — nor indeed in any workshop to follow in the semester. One of the key skills we need when working with large existing software systems is *the ability to keep moving forward* even when we don't have much of a clue about what is going on. We have to accept that we will never know everything there is to know about the tools used by the system, or the source code of the system, or any other aspect of the system.

For our purposes today, you just need to know how to run an Ant script to create an executable project. We'll take a look at the build script, to get an idea of how it works, but there will be a lot that we ignore or skip over very briefly. Becoming comfortable with this approach is one of the skills you need to develop over the course of this semester. (Many of you will already possess this skill, of course!)

1.3.1 Locate and Examine the Build Script

Open up the `newmarauroa` project in the Package Explorer (if you have not already done so), and scroll down until you see a file called `build.xml`. This is the default name for Ant build scripts. Double click on it, to get Eclipse to load the file into an Editor view, so that we can see its contents shown in figure **??**

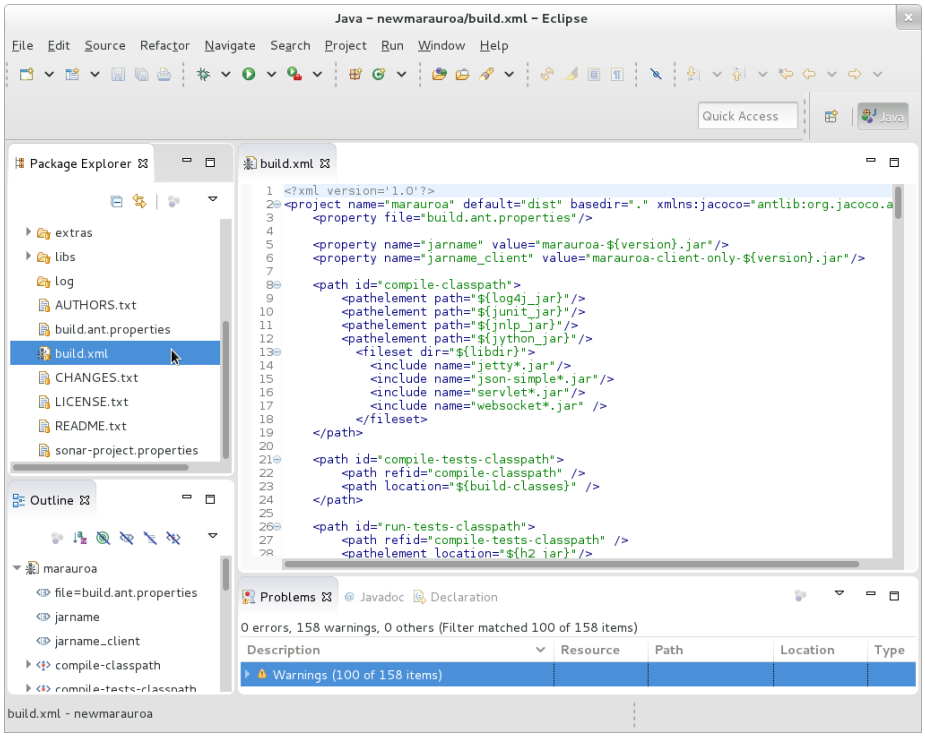

Figure 1.12: An XML build file

If the filename wasn't already enough of a clue, you'll see from this that Ant build scripts are XML files. XML tags are used to define the things that the file knows how to build, and the steps involved in building them, as well as key configuration information, such as class paths (show in the screen shot above).

Notice that the Outline view has also now been populated. This very useful view gives a high-level summary of the contents of a file, by listing its main components as a tree view. In the case of a Java file, the Outline view shows the classes defined by the file, and their members (fields and methods). In the case of XML files, like our build file, the Outline view shows the hierarchy of tags defined by the files.

We can use the Outline view to run Ant builds, by right clicking on the XML tags that represent descriptions of how to build things. But an even more useful

view is the **Ant View**. This is a view that has been created with knowledge of how the Ant build tool works, and added in to Eclipse as a plugin. Open it by selecting `Window` > `Show View` > `Ant` from the top level menus.

Now open `build.xml` from the view by clicking on the `Add Buildfiles` icon in the view toolbar. It looks like an ant with a green plus on its left shown in figure **??**

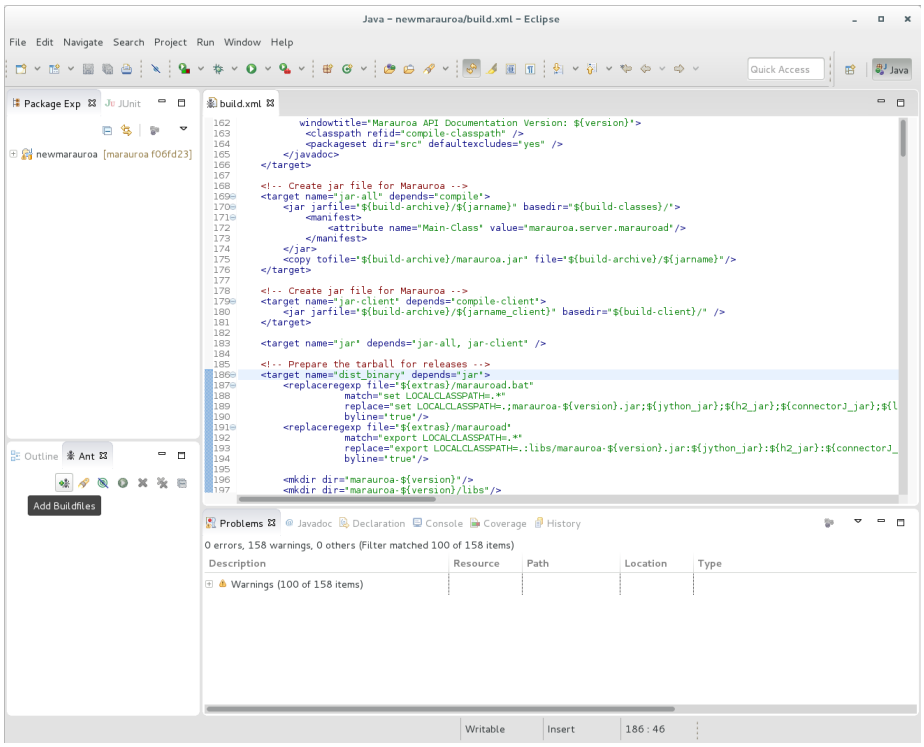

Figure 1.13: An XML build file

This will open a new dialogue that allows you to select a Buildfile. Select `build.xml` and click `OK` shown in figure **??**

Notice that the Ant view has been populated. Instead of listing all the top-level XML tags, this view knows just to list the **build targets**. These are the things the Ant script knows how to build. The user of the script can request which target she or he wishes to build.

Scan down the targets and see if you can guess from the name what each one builds. Hint: `dist` here stands for `distribution`.

Let's take a look at the definitions of some of the targets. Right click on the name of any of the targets, and select `Open In` > `Ant Editor` from the context menu that appears. You will see that the contents of the `build.xml` editor

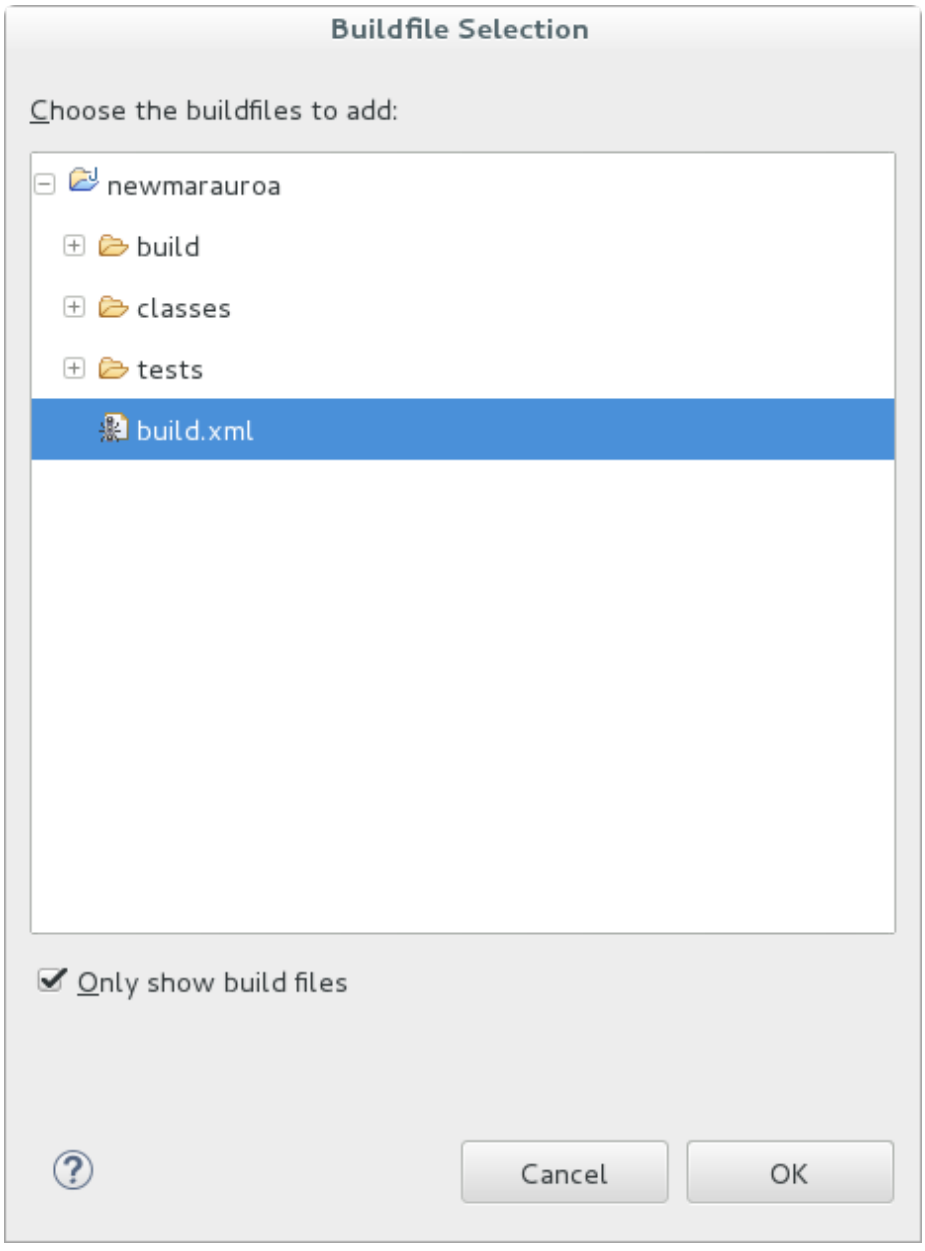

Figure 1.14: Buildfile Selection

window are changed, so that the definition of the target we have clicked on is displayed. For example, in figure **??**, we've clicked look at the `jar-all` target and take a look.

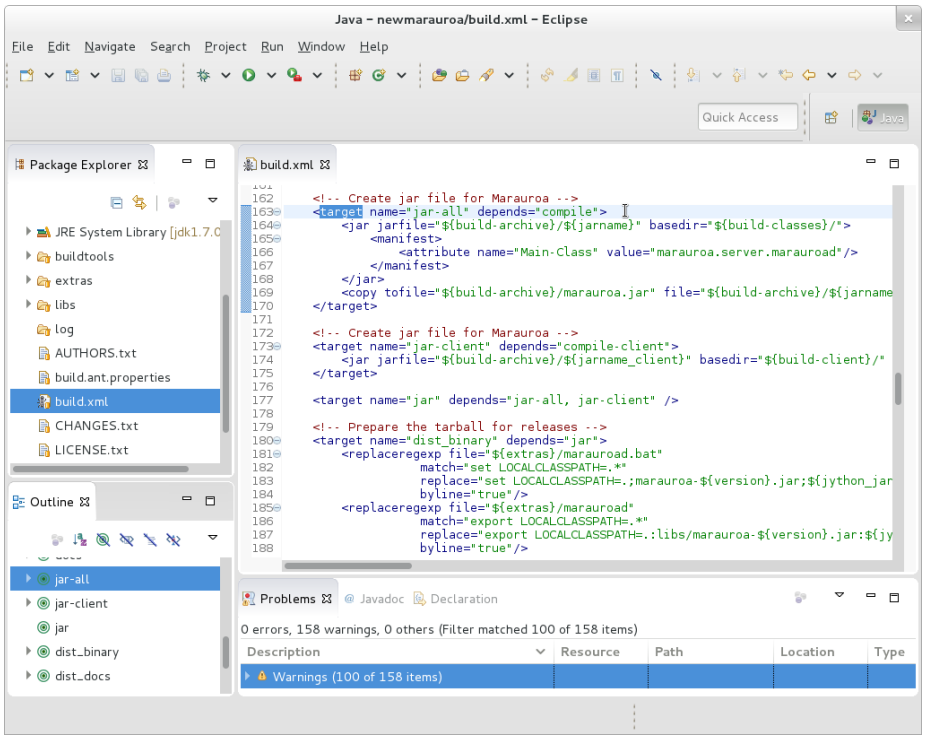

Figure 1.15: The jar-all target

We can get a rough idea of what this definition is telling us. First, note the `depends` attribute in the target tag. This states that before we can build the jar file for the project, we must have built the `compile` target. This makes sense as we need to have compiled Java code before we can create a Jar file.

These dependencies between targets are one of the key strengths of build tools such as Ant. We can describe individual steps in the build process, and state the other steps that they depend on. At build time, Ant will examine all the targets and their dependencies and find an order of execution that means that no target is built before the targets it depends on.

It's important to note, again, that you don't need to understand everything about the Ant build file to be able to make some educated guesses about what it is doing. We don't need a detailed understanding just now. We are just looking for easy-to-absorb clues as to what the various targets do.

A Note on Automated Build

At this point, you might be wondering why we are bothering with this complicated build script when the Eclipse internal Java builder already seems to be doing a good job of compiling all the Java classes for us, without us needing to do anything at all.

The answer is that there is typically more to turning source code for a non-trivial system into deployable software than just compiling the Java code. The Eclipse internal builder creates class files for all the Java files. But when was the last time you downloaded an app or application and what you got was a folder full of class files?

Quite what *deployment* means differs from application to application. Simple Java applications may simply be wrapped up into a jar file, but even then we often need to supply a shell script for setting the class path and executing the main method of the entry point class. If we are building a Web application then deployment typically means packaging up the components of the application in a `*.war` file (web archive file) and copying it into a particular directory (the one used by the container manager our web server provides). Or, we might need to prepare a zip archive of files, or to package up the files ready for use by an install tool.

As these examples show, the steps needed to deploy a system are often very simple, but they are also quite fiddly and fussy. One wrong key stroke and we end up with something unusable. Explicitly documenting the deployment steps in an automated build script make the deployment process quick, easy and reliable for anyone on the development team to carry out, even the newest team member. That is very important, as it means that tests can be run on the deployable form of the system (even if it is not, at that point, deployed in the live environment). As we have seen, the closer our test environment can be to the live environment, the more chance there is that we'll find errors before they reach the customer rather than afterwards.

1.3.2 Build the System Using the Build Script

Now that we have seen something of the build script, we are going to use it to build the whole Marauroa distribution. That is, **we are going to ask Ant to build the "dist" target**.

Right click on the target we want to build and select *RunAs* from the menu. You'll see that the IDE recognises the file we have clicked on as an Ant Build target and offers the option of running it as an Ant build.

Note: you can build a target from both the Outline view and the Ant view in the same way

Select the first of the two Ant Build options. The second takes you to a wizard, but we don't need that at this stage.

A Console tab will appear (figure **??**) in the bottom section of the Eclipse window, showing the output that Ant is sending to the standard output and standard error streams while it works. Double click on the tab of the Console view, and take a look at what Ant is doing.

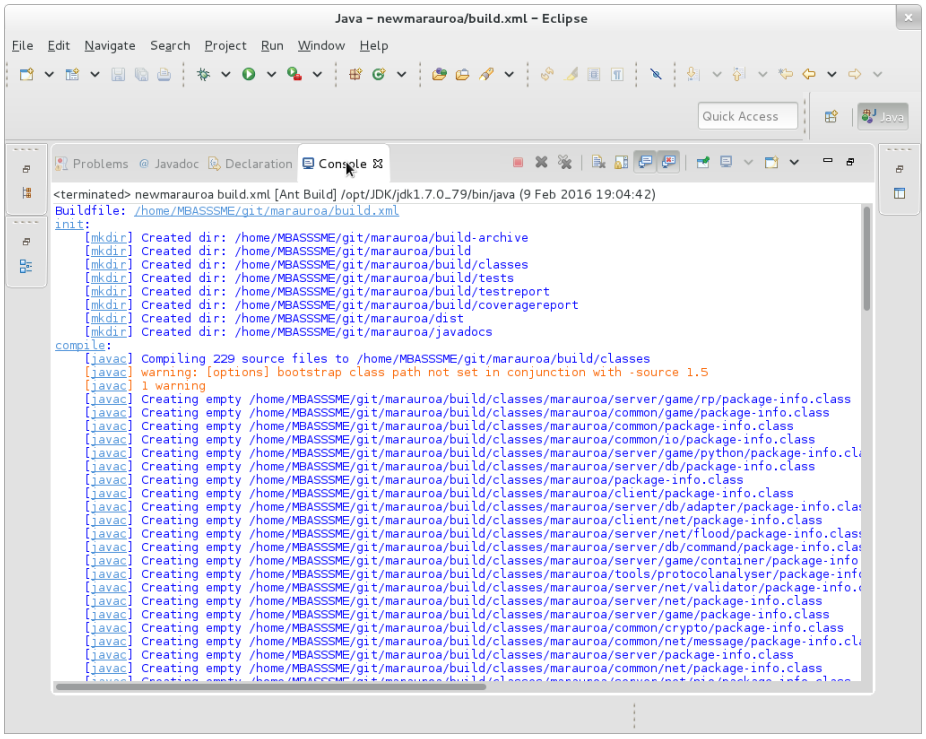

Figure 1.16: Console output

The console output shows the various targets that Ant creates, as it works through the dependencies specified in the build script. The targets are shown at the left of the window, followed by a colon (init and compile, in the above screenshot).

Beneath the target, the names of the tasks invoked are shown, in square brackets.

The most important part of the output, of course, is shown at the end, when the process finishes:

We can see here that the build was successful. We have built the executable version of the code, just by right clicking on a target! Building Marauroa would be a lot more work if we had to carry out all these steps ourselves, manually, every time the code changes, and the chances of getting a step wrong would have been much higher. This shows one of the strengths of automated build tools. The Marauroa team have encapsulated their expertise in building their games engine into this build script file. It now becomes possible for anyone,

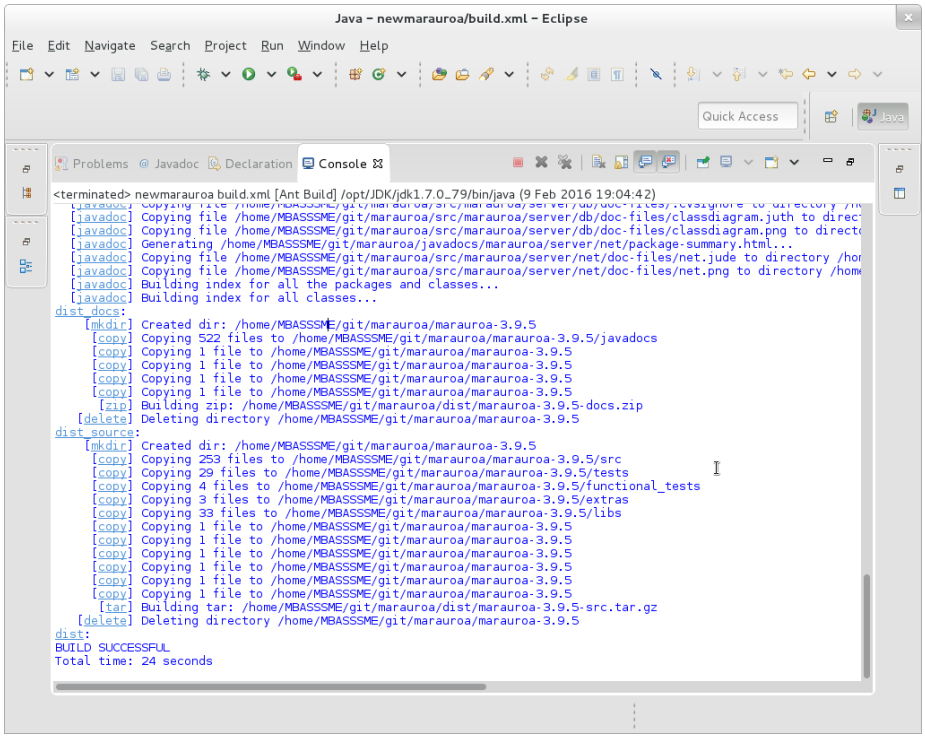

Figure 1.17: Console output

with or without expertise in Ant, or Marauroa, to build the system in the same way.

In other words, the build tool has made the build process *repeatable*. A source of potential error in working with the code (and in deploying to the user) has been removed.

Take a moment to look through the full console output from the build command we have just run. Look for the actions the build script is taking that are vital to creating a deployable product, but which are not about compiling individual class files.

1.3.3 Examining the Results of the Build

We'll finish this step by taking a brief look at what the build process has achieved.

Right click on the `newmarauroa` project name and select Refresh from the drop-down menu. Eclipse know about any file changes you make using Eclipse tools (such as the Java editor or the internal Java builder) and can update the view of the project you see through its GUI automatically for you. But Ant is not part of Eclipse. It is a separate tool that Eclipse is running for us. When an Ant script creates new files and folders, or moves things about, Eclipse doesn't know anything about it, and so the view of the project it shows to us can get out of date. The Refresh menu option tells Eclipse to go and look at the directory structure and files in the project directory, and to update the GUI to show the effects of any changes.

When you refresh, several new folders should appear: build, build-archive, dist and javadocs.

Take a few moments to look at the contents of these folders, and see if you can form any hypotheses as to their role in the deployment process. If we were going to share the Marauroa engine we have just built with a friend, what would we need to do?

STEP 2 of 4 COMPLETED

You have now completed the build step of the process. Now we need to find out whether the engine we have built does what we expect it to. Next, we will learn how to run the automated test suite that the Marauroa team have created.

1.4 Testing the Marauroa Engine

Having created an executable version of the system, the next step is to check whether it is working correctly. In this part of the activity, we'll take you through the process of running the automated test suites created by the Marauroa team.

You saw one way to run JUnit tests in Eclipse in the GitLab Access Check activity. But there we just had one test class with just a few test methods to worry about. The Marauroa test suite is much larger than this, and we need a different approach.

We'll take a first look at how these suites are organised and implemented in this step, though this is a topic we'll be coming back to in future workshops, too.

1.4.1 Finding Out What Tests There Are to Run

Before we run the tests, it is helpful first to take a high level look at the test suites provided by the developers of the system we are working with. One way to do this is to look at the source folders in the project. The source code for any large project, nowadays, is typically split into two halves: the production code (the part that the user will use and the customer will pay for) and the test code (the part that the development team use to work out whether they are delivering the right thing). It's important not to get these two parts of the code mixed up, and therefore it is common practice to split test code off into its own folders (and sometimes its own packages).

Another source of useful information about the test suites is the Ant build script. Although called a `build` script, we have seen that these scripts do a lot more than just compiling code. Their task is not just to create an executable version of the system, but to create a verified executable that is ready for the user to take away and use. Therefore, these scripts more normally follow a three step process:

1. build
2. test
3. deploy

The Marauroa build script is unusual in that the target that produces the distribution doesn't also run the tests. But it (the build script) does contain instructions for running the test suites.

Take a look at the targets in the build script. We can see that there is one called `test`. That sounds promising. Let's take a look at figure **??**

Let's take a quick look at that target before we look at the rest of the `test` target shown in figure **??**

The target in figure **??** depends on the `compile` target. In other words, the Marauroa team are saying here that if you want to compile the test code, then you have first to compile the production code that it tests (which makes sense, because the test code will make use of lots of classes and methods from the production code).

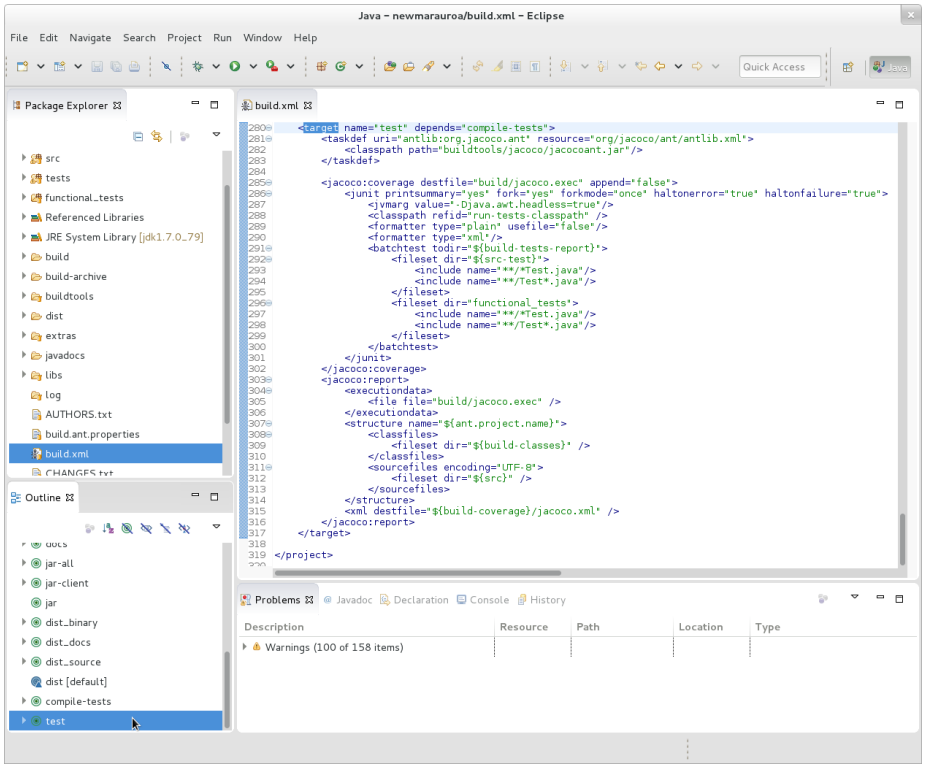

Figure 1.18: We can see that this target is dependent on another target, called `compile-tests`. That makes sense as we would expect to have to compile the test (and production) code before we can run the tests.

```
<!-- testing targets -->
<target name="compile-tests" depends="compile">
    <javac srcdir="${src-test}" source="1.5" target="1.5" destdir="${build-tests}"
        debug="true" debuglevel="source,lines"
        includes="**/marauroa/**" excludes="${exclude.python};${exclude.junit}"
        includeantruntime="false">
        <compilerarg line="-encoding utf-8"/>
        <classpath refid="compile-tests-classpath" />
    </javac>
    <javac srcdir="functional_tests" source="1.5" target="1.5" destdir="${build-tests}"
        debug="true" debuglevel="source,lines"
        includes="**/marauroa/**" excludes="${exclude.python};${exclude.junit}"
        includeantruntime="false">
        <compilerarg line="-encoding utf-8"/>
        <classpath refid="compile-tests-classpath" />
    </javac>
    <copy todir="${build-tests}">
        <fileset dir="${src-test}">
            <include name="**/*.*" />
            <exclude name="**/*.java" />
            <exclude name="**/*.ini" />
        </fileset>
    </copy>
    <copy file="${test-server-ini}" tofile="./server.ini" />
</target>
```

Figure 1.19: Testing targets

In the description of the `compile-tests` target, we can see two calls to `javac`, and a couple of file copy commands. The `javac` commands are compiling code in the folder specified by the `$\{src-test\}` property and the `functional_tests` folder.

A string of the form `\$\{something\}` in an Ant script is a reference to the value the property called `something`. They can be defined in the Ant script itself (using the `property tag`), but the `src-test` property has its value set in the `build.ant.properties` file, which the build script imports. If we look in that file, we can see that this property is set to the path to the `tests` folder.

So, we can see from this small section of the build file (without bothering to look any further) that there are two kinds of test in the Marauroa system: functional tests and another kind of test. It is a fairly safe bet that this other kind of test are unit tests.

Forgotten what unit tests and functional tests are?

This was covered in COMP16412. Unit tests are short snappy tests that (strictly speaking) just test the behaviour of a single code unit. In Java, we normally think of individual classes as the units for unit testing. Functional tests are tests of the major functions that the system offers, and will typically involve the execution of many classes working together.

In practice, it's quite hard to write true unit tests, and many of the tests in the `test` folder will in fact be *integration tests*, i.e., tests that assess the behaviour of a small number of units, working together.

Now that we understand something of what is happening in the dependent tasks, we can go back to the `tests` target. Its body contains a couple of tasks that appear to be calling a tool called `jacoco`.

JaCoCo is a test coverage tool. We'll look at what it does in more detail later in this activity, but for now all you need to know is that it is a tool that runs the tests, and works out what proportion of the production code statements are executed by the tests.

We can also see a call to a `junit` Ant task embedded in the `jacoco` task definition. That must be where the tests are actually run. It is run inside a `jacoco:coverage` task, suggesting that JaCoCo will be collecting the coverage information while JUnit is running the test suite.

The other target is called `jacoco:report`. The name suggests that it has the job of taking all the coverage logs gathered from running the tests, and producing a coverage report from that information.

1.4.2 Run the Tests

Now that we know a little about what is happening inside the test-related targets, we'll run them. Just as we did with the `dist` target when building the

code, we're going to right click on the `test` target, and select `Run As` > `Ant Build`.

Please try that now.

As before, you should see a log of what Ant is doing appearing in the Console view.(Note that this time, the compile target and its predecessor targets are run but seem to do nothing. This is because Ant knows that the production code source hasn't changed since these targets were last built. So, there is no point wasting any time recompiling them, when we can just use the object files that were created the last time.)

The build should succeed as before, indicating that all tests pass successfully.

1.4.3 Examining the Test Results

It's useful to find out more details about the results of running the test suite, but the summary output we get on the console from the build is not very helpful. We need a better way to get more details about the results of running the test suite. But where are the results stored?

Line 37 of the `build.ant.properties` file tells us that the `build-test-reports` are in the `build/testreport` folder. We'll need to refresh the project to allow Eclipse to show us these new folders and files, so right click on the project name in the Package Explorer View and select `Refresh`.

You should now be able to examine the contents of the `build/testreport` folder shown in figure **??**

The icon next to these test result files tells us that they are JUnit test results, and the suffix tells us that they are XML files. You can double-click on any of these XML files, and Eclipse will open them in the special JUnit viewer. For example, if you select the file:

`TEST-marauroa.clientconnect.ClientConnectTest.xml`

you'll see the JUnit view shown in figure **??**.

We can see that there were four test cases in this class:

1. `clientconnectTest`
2. `createCharacterTest`
3. `joinGame`
4. `wrongPwTest`

All are shown with a small green tick next to them, indicating that they passed. The numbers in brackets after the test name indicate the execution time of the test. (You can see that they are all running in a fraction of a second, which is

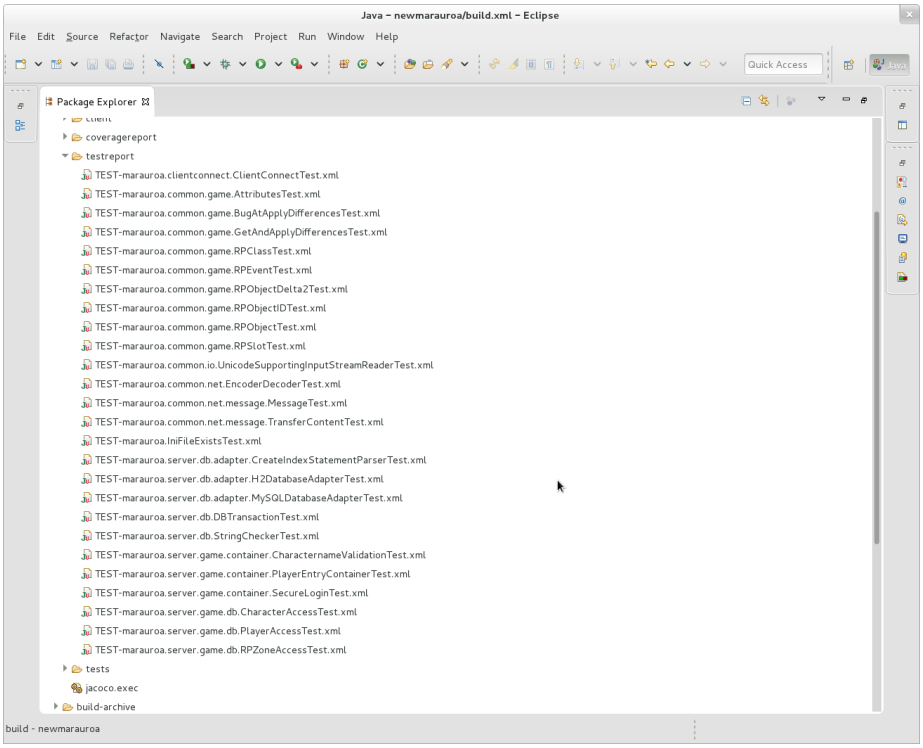

Figure 1.20: The testreport

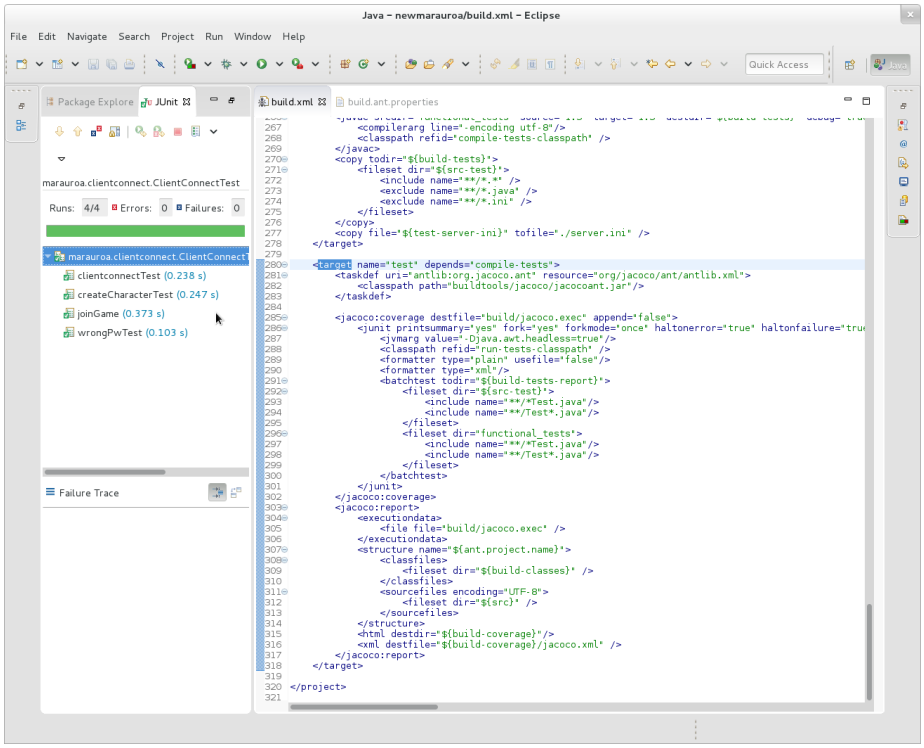

Figure 1.21: The JUnit view

what we need for tests of this kind, as we would expect to be running them very regularly as a developer - after every small code change, in fact.)

The green bar at the top of the JUnit view also indicates that all the tests in this test class passed.

We can double click on any of the tests to find out more about them. Try this with the first test: `clientconnectTest`. You should see the source of the test case, loaded in the Editor view shown in figure **??**

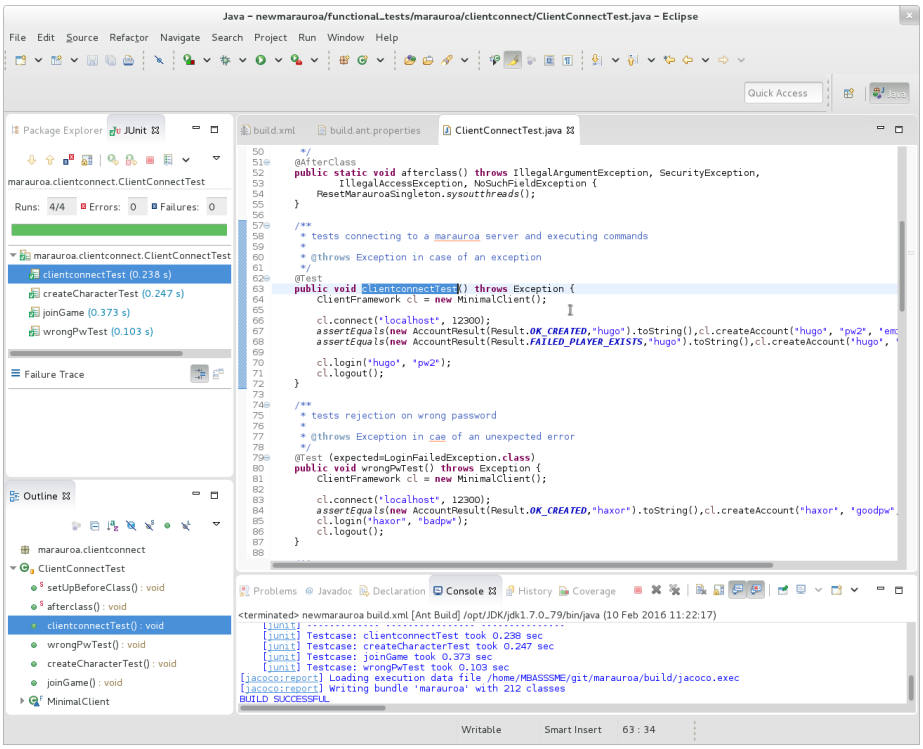

Figure 1.22: The Editor view

You can see that the test in **??** is short, and quite descriptive. This is typical of unit test code. Did you also notice that this is the first time we have looked at Java code in our exploration of the Marauroa system. Test cases make a great starting point for understanding what an unfamiliar code base does, as we shall see in another COMP23311 workshop.

If you have time, you can explore some of the other test results, and the test cases associated with them. Try not to get too bogged down in the details, though. We're just trying to get an overview of what the system is doing here, rather than drilling down into the details of any one feature.

1.4.4 Examining the Test Coverage Results

We can now see the results of the JUnit tests. But what about the code coverage results produced by JaCoCo? How do we get to see those? The results are stored in two places: a file called `build/jacoco.exec` and another called `build/coveragereport/jacoco.xml`.

IMPORTANT NOTE Don't try to open the jacoco.xml file in Eclipse! It is huge and Eclipse will spend a lot of time trying (and probably failing) to grab enough memory for it. If you want to see what it contains, then use a lightweight text editor or a command like `head -c` from the command line.

It's not clear what use the Marauroa team make of these results, but we would normally prefer to have the results of the code coverage in a more human-friendly format than a giant XML file. Jacoco provides the facility to create a report as a web page, as well as in XML form. So, we're going to modify the build.xml file, to create this more useful form for us.

All we need to do is add one extra line to the test target in the build file, just after line 320:

```
<html destdir="${build-coverage}" />
```

Note that the parameter is `destdir`, not `destfile`, like in the line that follows. Figure **??** below shows how the edited build file should look, with the new line highlighted.

The line to add has been highlighted in the editor window in figure **??**. When you have added it and saved the file (Control-S is the keyboard short cut, or you can click on the small floppy disk in the tool bar), run the Ant test target again. When this completes, **refresh** the project to pull in the extra report files we have asked JaCoCo to create.

Expand the `build/coveragereport` folder. You should see lots of extra folders inside it, with names that look like they could be Java packages. You should also find a file called `index.html` down towards the bottom. This is the root file of the HTML report that JaCoCo has created for us.

Eclipse has a Web browser plug-in that you can use to look at this report (by double-clicking on the `index.html` file). This plug-in was pretty buggy in previous releases. If you find this to be the case under Eclipse 2020-03 too, you can open your preferred Web browser and look at the files in that. (You'll need to use a file browser to locate the file in your file space. Searching for the directory called `coveragereport` is a quick way to do this.)

The report in figure **??** shows, for each package, the degree to which the test suite exercised the source code. For now, we'll just focus on the first four result columns. For the second package, `marauroa.common.game`, we can see from the report that the test suite executed 68% of the instructions in the package. (An

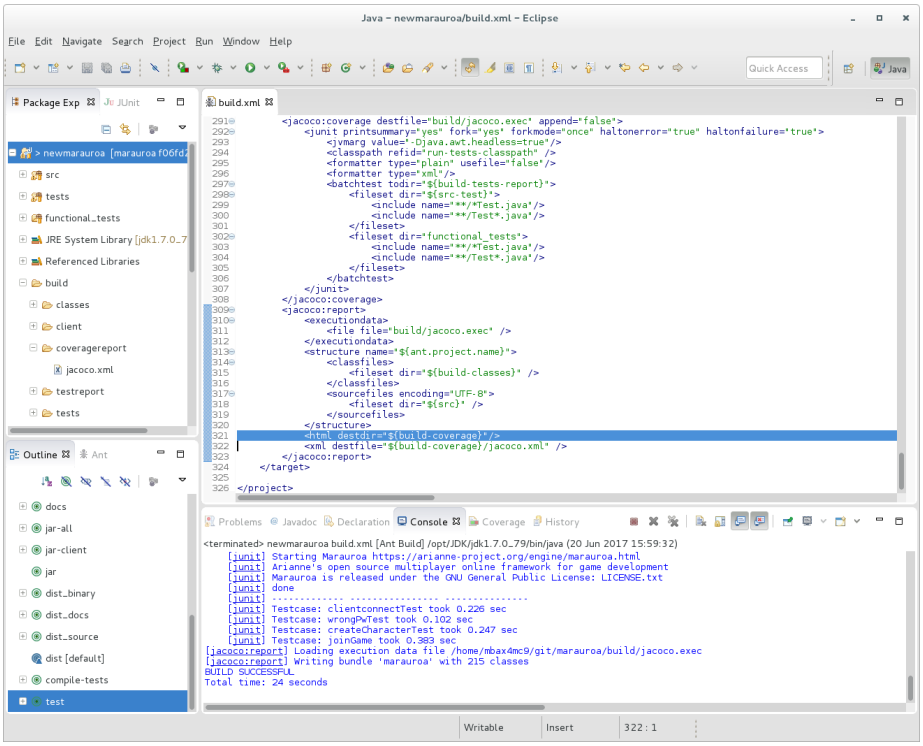

Figure 1.23: The JUnit view

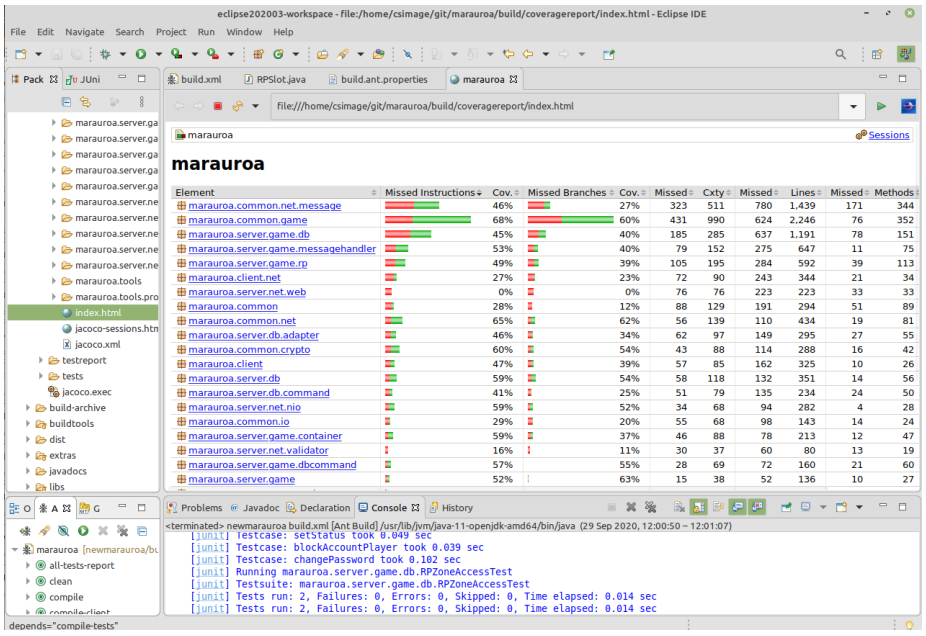

Figure 1.24: A test report

instruction, here, is a single Java byte code instruction.) But, 32% (around a third) were not executed at all by the test suite. Any bugs in instructions not covered by the test suite will not be caught by it.

The next two columns show how many "branches" in the code were covered by the test suite. A "branch" in this context means a conditional point in the code, where execution could follow one of two paths based on the value of the condition. JaCoCo currently computes branch coverage only for `if` and `switch` statements, though theoretically loops also introduce branches into code. We would like our test suite to exercise all exits from all branches. That is, if there is an if-statement, we would like our test suite to execute the if-statement with a true condition (so that the then-body is executed) and with a false condition (so that the else-body is executed). These columns assess how far the test suite has met this goal. In the case of our example, just 60% of the branches in this package are exercised by the test.

Aside: Coverage by Instrumentation

Code coverage tools like JaCoCo need to find out which source code statements were executed when a test (or suite of tests) are run. They typically do that by "instrumenting" the code. That is, they convert the code of the system so that every statement is accompanied by a second statement that logs the occurrence of the first statement in some file somewhere. For example, the code fragment:

```
int x = Math.random();
System.out.println(x);
```

would be converted into:

```
int x = Math.random();
coverage_log("int x = Math.random();");
System.out.println(x);
coverage_log("System.out.println(x);");
```

(Obviously, this is a simplified picture of what is actually going on.) When the instrumented code is run, as well as executing the main code, a log is gradually built up of which statements were executed and when.

You can drill down further by clicking on the package names to see code coverage reports for each class. If you click on the classes, you'll get a breakdown of the coverage per method. You can even get reports on the coverage of individual lines of code (by clicking on the methods in the coverage report). For example, figure **??** shows the detailed coverage report for the method `marauroa.common.game.RPObject.size()`.

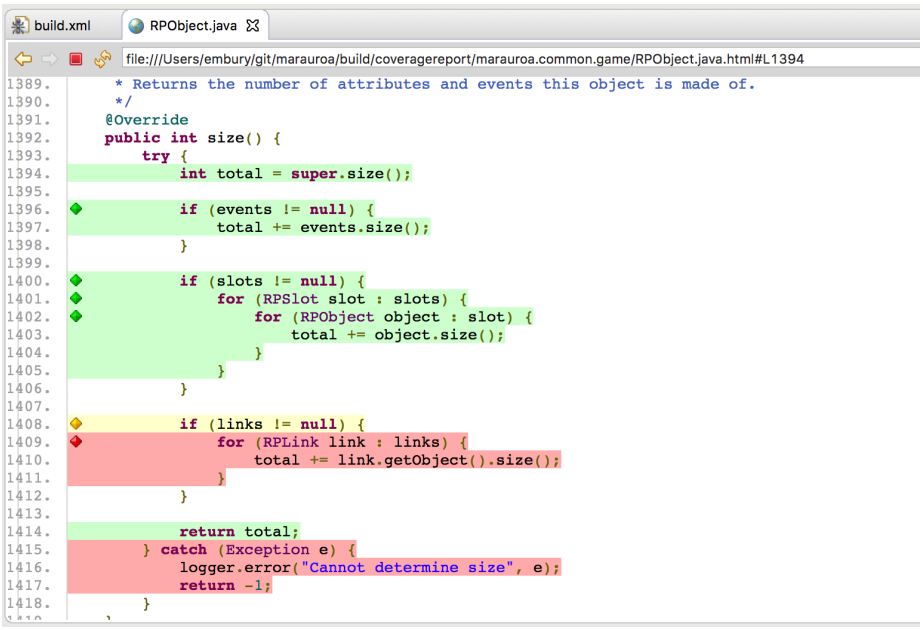

Figure 1.25: A more detailed coverage report

In figure **??** the green lines were executed by the test suite. We can see that the beginning of the method has been covered well by the tests. All the early

instructions were executed, and the two if-statements have been executed with both a true and a false condition in different test cases.

Towards the end of the method, the coverage looks less good. The yellow colouring of the if statement on line 1408 indicates that only one of the two exits from it were exercised by the test case. Since the body of the if-statement is coloured red, it seems that the code has only been executed in scenarios where the `links` variable is set to null. We can also see that the exception handling code has not been tested.

Hopefully, it is now obvious how useful this kind of tool is. If we see that important and complex parts of the code are not covered by the test suite, we can write test cases that explicitly target the missed branches and instructions (using white-box testing design techniques). In this way, we can gradually build up a test suite that covers all the important cases, while not wasting time on covering parts of the code that are seldom executed or of little importance.

STEP 3 of 4 COMPLETED

You have now run the test suite for the Marauroa engine, and have begun the process of understanding how it works. We'll be coming back to look at the tests in more detail in a future workshop. For now, we're going to spend whatever is left of the workshop looking at how the test suite can help us to detect when bugs are introduced into the code.

1.4.5 Using the Test Suite to Find Bugs

We're going to end the workshop by making a quick experiment to show the power of this kind of automated test suite. We're going to make a change to the code, and then we'll see if the tests can indicate that something has been broken. For example, let's make a small change to the method:

```
marauroa.common.game.RPSlot.setDeletedRPObject()
```

First, we need to get the source of this method loaded into the Editor view. You can do that by expanding the src folder tree in the Package Explorer, or by using the Search facility from the main menu bar. (File search is the easiest to use, but Java search would be a sensible way to run this search, too.)

Double click on the class or method to load it into the Editor window. You can use the Outline View to locate the method quickly once you have the class loaded into the Editor. Now we can make the change. Comment out line 649, as shown in figure **??**

Save the file and run the build test target.

This time, you should see a failed build like the one shown in figure **??**. This is because one (or more) of the tests has failed, because of the change we introduced. The failing test(s) caused the process of executing the test suite to come

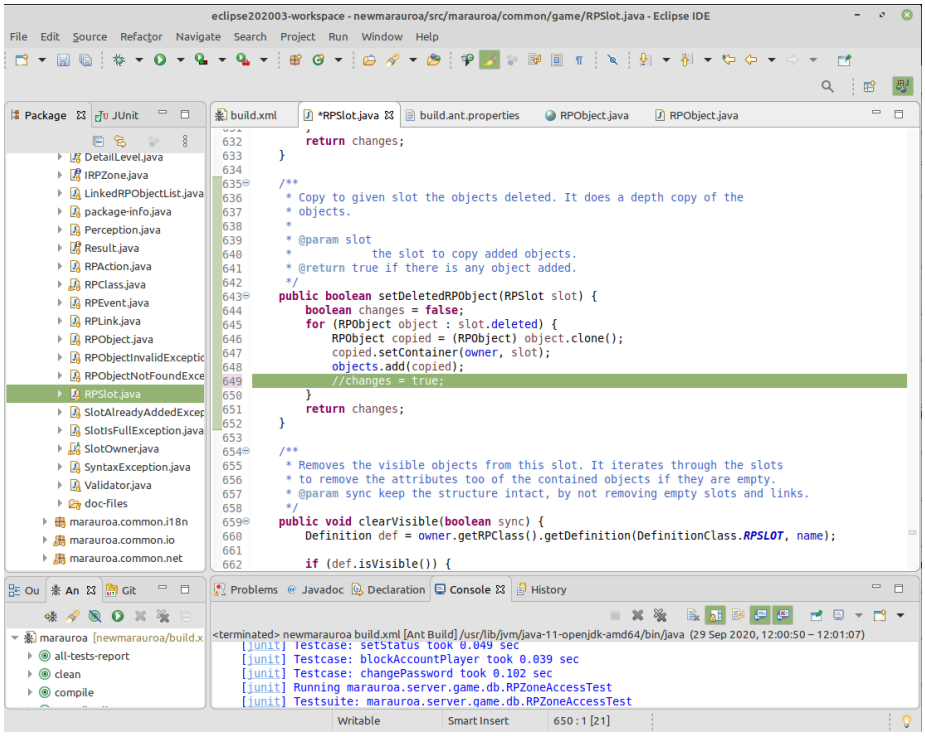

Figure 1.26: A more detailed coverage report

to a full stop. This is because the Marauroa build script tells JUnit to stop as soon as a failing test is encountered.

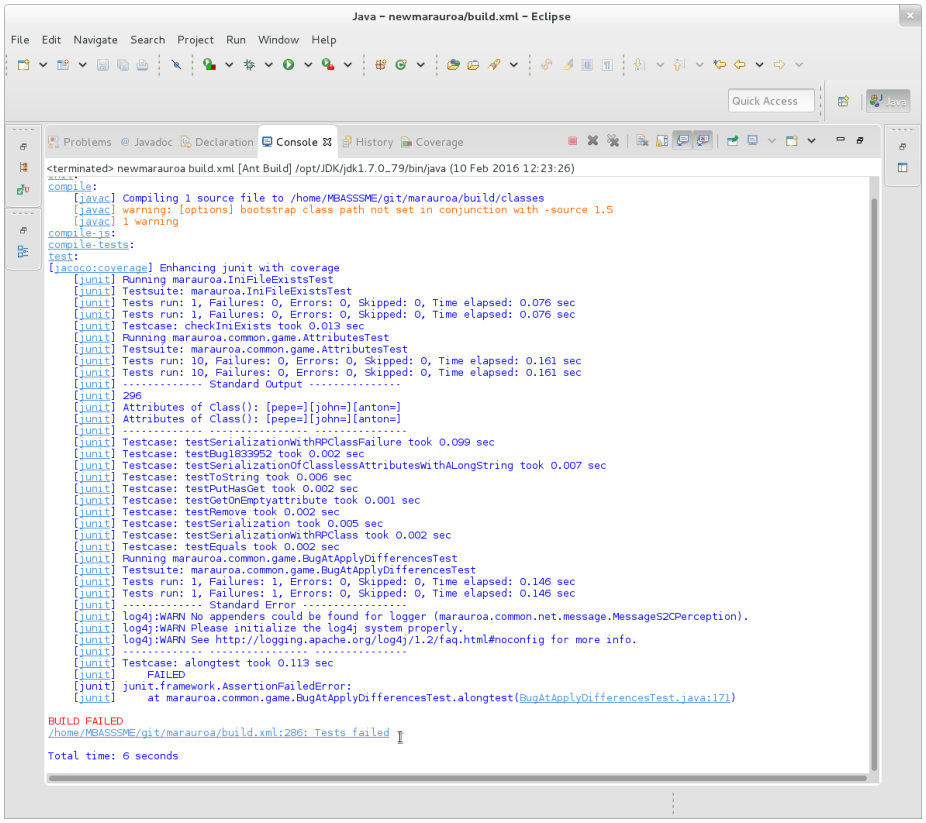

Figure 1.27: A more detailed coverage report

In some situations this is useful (it leaves the details of the failing test clearly visible in the console window), but in other situations we want to see the results of the whole test suite, whether there are failures or not. In other words, we want the build process to carry on despite the failing tests.

To allow this, open up the `build.xml` file, and go to line 292. On this line, change the values of the `haltonerror` and `haltonfailure` parameters to both be `false`. Then run the tests again, from the Ant View. You should now see a much larger number of test results scrolling by in the Console View.

Then run the test build target again. You should see output similar to figure **??**.

Notice the important line at the bottom in red. One or more of the tests failed. To see the details, open up the JUnit results file:

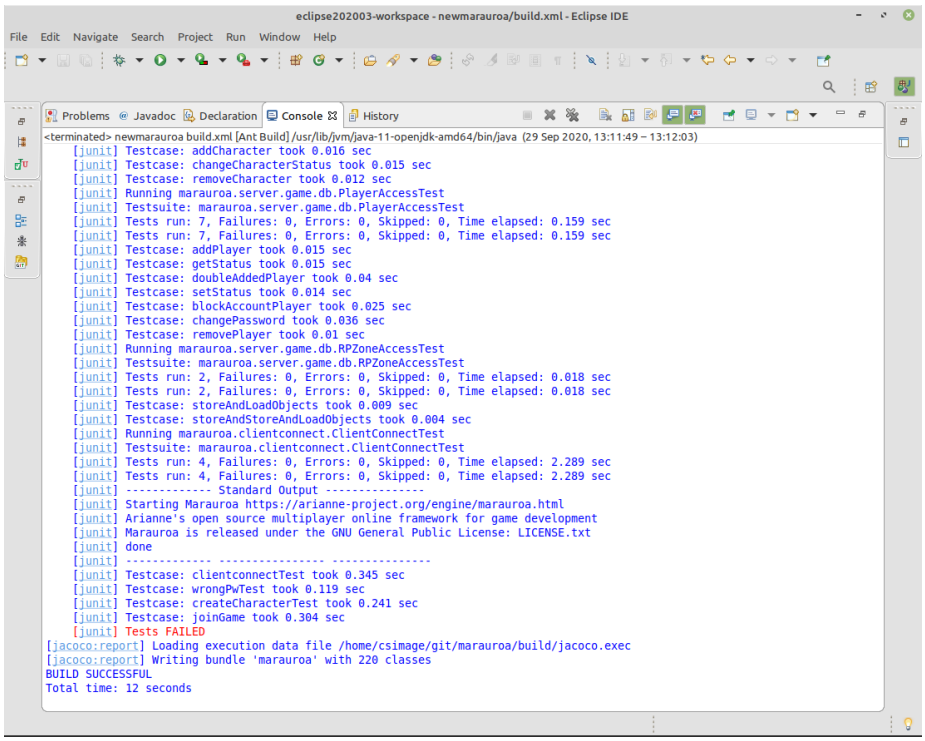

Figure 1.28: Tests FAILED

```
build/testreport/TEST-marauroa.common.game.RPObjectDelta2Test.xml
```

Double click on the names of the test cases that failed in the JUnit View. You will see that the test class is automatically loaded into the Editor window, with the cursor placed at the assertion that failed.

This shows that the error we introduced in this case was spotted by the tests.

You may be thinking at this point that I must have spent ages looking for a line of code that I could comment out and that would cause a test to fail. In fact, this line was the first one I tried — honest! I did cheat a little bit, as I made sure to look for a line to change that was in code that was well covered by the test suite. If I'd made a change in code that was less well covered, then I'd probably have had to look harder to find a change that the tests could spot.

Aside: Writing Tests to Trap Bugs

You may notice that the name of the test that failed here contains the word "bug". Even with a fairly comprehensive test suite, it is still possible for bugs to slip through. When this happens, it is good practice to write a new test that fails due to the bug. That is, the test describes what the correct behaviour of the system should be, and its failure tells the developers that the bug is still in the system. Then, when the developer thinks she has fixed the bug, she can run the bug test and find out whether she has or not.

When the bug is fixed, the test we wrote to make it visible can enter the normal pool of tests that we run regularly over the system. That way, if any future code change causes the bug to reappear, we'll have a test that will catch it.

In whatever time is left, try making your own changes to the code. Run the tests, and see if they were able to detect the error you had introduced. Try making changes in code that is well covered by the tests and in code that is less well covered. How did the test suite do?

You won't have to try this for long before you start to want a better way of looking at the test results than scanning through the build output on the console. It is usual to set up the build script so that it creates a summary report of all the test results for the project, so that you can see at a glance which tests are failing. The Marauroa team have not done this (perhaps because they prefer to look at test results through their continuous integration and test system—we will look at these tools, and use them for the coursework, later in the course unit).

If you want to get a summary of all the test failures, you can add the following target to the end of the build script (before the closing "project" tab):

```
    <target name="all-tests-report" depends="test">
        <property name="test-summary-report" value="${build-tests-report}/summary"/>
        <mkdir dir="${test-summary-report}"/>
```

```
        <junitreport todir="${test-summary-report}">
            <fileset dir="${build-tests-report}">
                <include name="TEST-*.xml"/>
            </fileset>
            <report format="frames" todir="${test-summary-report}"/>
        </junitreport>
    </target>
```

You'll now see a new target appearing in the Ant View for the `build.xml` file. Run this target to generate the summary report in your test reports folder. Once built, the file to look at is: `build/testreport/summary/index.html`.

When you have injected some bugs that your test suite catches, why not challenge your colleagues to see if they can use the information provided by the failed tests to work out which line you changed. (Don't forget to make a note of the class file and line you changed.)

STEP 4 of 4 COMPLETED

All done with the activity and nowhere to go?

If you have raced through all the above, and still have time left in the workshop, you could try to run the code we have built so far. Marauroa is a game engine rather than a game itself. It provides functionality to be used by other software (a game), and so if we run it by itself, there isn't much to see. We need to have a game of some sort to run, and then we can see the engine at work.

The Marauroa team provide a tutorial describing how to use Marauroa to create a simple "chat" game using the engine, which does allow us to run the engine we have built. Head over to the Marauroa wiki and follow the tutorial instructions if you want to try this out (it's an optional exercise, and is not important for the coursework or the remainder of the workshops). It is very short, and just involves the creation of 5 classes in total. All code for the classes is provided, but there are a couple of tricky elements to making it all fit together in Eclipse.

You should start by making a new Java project in Eclipse, with a `lib` folder (an ordinary folder, not a source folder). Import the Marauroa jar file that you built in the previous steps into this directory, and add it to the build path for your project. This is done by right clicking on the imported jar, and selecting `Build Path` > `Add to Build Path`. You can then add the files from the tutorial to the `src` folder.

You'll need to add a couple of other libraries to the `lib` folder as you progress. They can all be imported/copied from the `newmarauroa` project.

When you are ready to run, you'll need to create a new Run Configuration. The drop down menu associated with the small green circle and white triangle in the task bar (i.e. the run button) will take you to the screen for this.

Do not hesitate to ask if you are stuck!

Chapter 2

Large systems

During your summer internship, year-long placement or after you leave University, chances are you will work on LARGE software systems. So it is crucial to be able to understand large software systems. You need to develop strategies for working with unfamiliar and large codebases. The codebase you're working on here is probably much larger than things you've worked with previously, though its relatively small compared to larger well known software projects, see figure ??.

Figure 2.1: Like telescopes, software projects can get quite large, so you'll need to develop strategies for working with large codebases. Telescope Names (xkcd.com/1294) by Randall Munroe is licensed under CC BY-NC 2.5

2.1 Purposes of the workshop

In this workshop, we will look at strategies to better comprehend unfamiliar large codebases. These strategies are supported by code reading techniques (**?**) and the functionalities offered by your Integrated Development Environment (IDE). We'll be assuming that, after this workshop, you will be capable of carrying out the following tasks for yourself, without needing much guidance:

- Navigate large codebases using the most effective strategy for comprehending the code.
- Use the views and functionalities provided by the Eclipse IDE to acquire a better understanding of the code.

- Read and write unit tests to understand the codebase.

In this workshop, you will:

- Use Eclipse to explore the codebase of Marauroa.
- Use the functionalities of Eclipse to carry out top-down reading strategies.
- Build a simple calculator to remove the fear to unit testing.
- Write unit tests to understand different components of Marauroa.

2.2 Learning Large Codebases

2.2.1 What activity takes most of a maintenance programmer's time?

Having to work with a large unfamiliar codebase is a very common challenge that you will have to face at some points during your career. Hence, developing the required skills to deal with this challenge is imperative. In this course unit so far, you have already faced this challenge (twice!).

Large codebases keep changing all the time. Normally, there are a handful of people working on the same project at the same time. Hence, it's very difficult for anyone to claim that they have full knowledge of all parts of the codebase. What really matters is to learn how to *build a mental model of the large codebase that helps you navigate* and find your way around the large codebase you are trying to work with whenever you need.

2.2.1.1 Tips for learning large codebases

It can be challenging finding your way around a large codebase that you are new to. Here are some strategies you can apply:

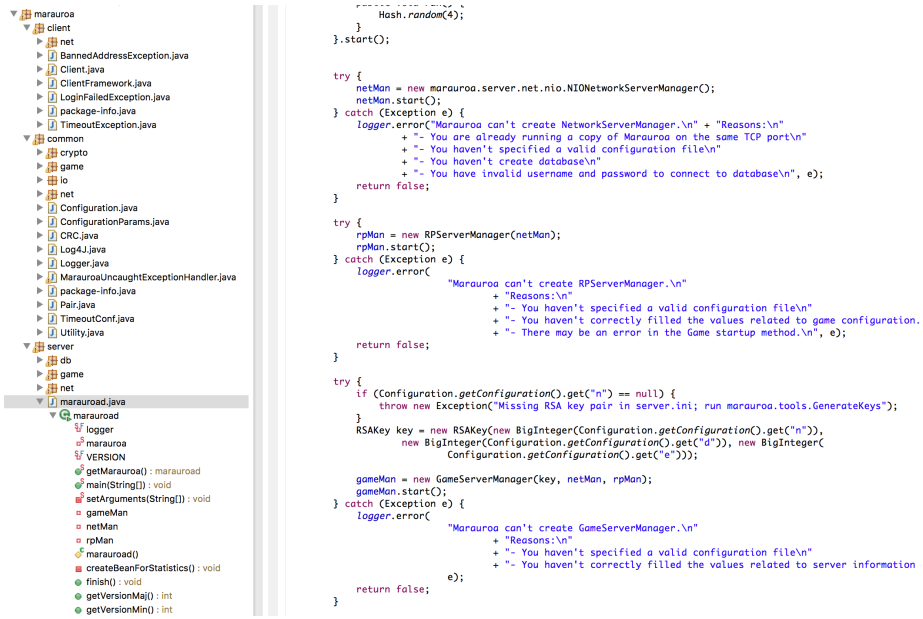

Figure 2.2: marauroa is a large codbase, how can you find your way around it?

2.2.1.1.1 Tip 1: Develop general knowledge Develop your general programming knowledge. Most of the large codebases follow similar well-known design patterns. The more you know about these, the easier things will be for you. Does the codebase you are trying to debug use an MVC pattern for instance? if you already know the MVC pattern, your life will be much easier dealing with that codebase.

2.2.1.1.2 Tip 2: Develop domain knowledge Develop you application domain knowledge. The more you know and understand about the application domain, the better your understanding of the codebase will be.

2.2.1.1.3 Tip 3: Be systematic Packages and classes are hierarchical! Use systematic reading strategies such as top-down and bottom-up strategies. In top-down, you use the context of the application together with some previous assumptions in order to gain an overall understanding of the codebase. In the bottom-up strategy, you start with individual statements and build up picture incrementally.

2.2.1.1.4 Tip 4: use your IDE Use the functionalities of the IDE. Different IDEs (such as Eclipse) offer different options that can be very helpful in learning large codebases. On your Eclipse IDE, select an attribute, function

or object and with a right mouse click, explore the options you have on the resulting menu.

2.2.1.1.5 Tip 5 Always assume that previous coders were sensible and honest. Every part of the code was written to serve a purpose and it was written in a very logical and methodical way. If you do not understand something, don't just dismiss it. Of course, there is always the chance that someone made a mistake, but do not rely on that.

2.2.1.1.6 Tip 6: read the tests Read the Tests! Every large codebase comes with a large test suite that contains hundreds or even thousands of tests. These tests are usually organised in a structure that mimics the source code. These tests are typically a very important resource for learning what various parts of the code are meant to do. Reading through the tests gives you a very good opportunity to understand the expected behaviour of a specific part of the source code, and then try to match that with what the actual code. Even better, if you go ahead and modify some of the tests or even write your own! This will dramatically increase your understanding of the codebase.

2.2.1.1.7 Tip 7: Take your time Take your time, and don't hesitate to ask. As we discussed above, large codebases keep changing all the time. Even the most experienced software engineers can't claim they have detailed knowledge of every single part of the codebase all the time. So, take your time and don't put a lot of pressure on yourself. Take a breath and work in a logical and a methodical manner. If you ever get stuck, always feel free to ask a more experienced member of your team.

2.2.1.2 Comprehending Marauroa

Now, using the tips outlined in section **??**, try to apply them to improve your understanding of the Marauroa codebase.

Let's start with the application domain. What do you know about Marauroa's application domain? By this point you should already know that it is a game engine that is used to develop Massive Multiplayer Online Role-Playing Videogames MMORPGs. How does/did this piece of information helped your understanding of the codebase?

Secondly, in the IDE, follow the gradual expansion of the hierarchy and read package and class names shown in figure **??**. This gives you a bird's eye view of the general structure of the codebase. Do you think the names are meaningful? Did that contribute to your understanding of the codebase?

Now try to identify the classes that play a central role. Once you identify a few of those, try to do more digging. Open these classes and skim through them.

Figure 2.3: A screenshot from marauroa

Check the import clause to see their dependencies. Do you notice anything in common? (Use the Outline view of Eclipse). For instance, check the class `marauroad.java` within the `marauroa.server` package. How do you think this approach helps you improve your understanding of the codebase?

Classes can also be read gradually using a top-down strategy. Look at the icons provided by the environment. Do you know what each icon means? Skim through the comments inside the class – use Javadoc. Do you think the comments were useful? Or are they just adding more chaos?

Skim the attributes and methods of the class. You don't need to understand every word to understand overall meaning. Can you establish any hypotheses about the "marauroad.java" class? which one of the following statements do you agree with:

- It's a daemon running on the server side.
- It's a thread that throws 12 threads.
- It follows a singleton pattern.

2.3 Unit Testing Overview

2.3.1 Definition

A unit test is an automated test that quickly verifies a small piece (unit) of code in an isolated manner.

- What is **a small piece of code**? The convention is that the piece of code under test should be a single class, or a single method inside that class. A common beginner's mistake is to try to test more than one class at once. In general, you should always strive to keep the guideline of unit testing one class at a time.
- **Quickly** refers to any amount of time that is acceptable within a given domain (but normally, it should only be fractions of a second). However, as long as the execution time of your complete test suite is good enough for you, it means that your tests are quick enough.
- **Isolated manner** refers to the fact that unit tests should be written such that they can be run in isolation from each other. This way, unit tests are not dependant on each other, and they can be run in any order at any time.

2.3.2 The AAA pattern: Arrange-Act-Assert

The AAA pattern is a simple and intuitive approach that provides an elegant structure for unit tests. It helps reading and writing unit tests that are easily understandable and maintainable.

- **Arrange**: in this section, we set up the objects to be tested (these are usually referred to as the `System Under Test` – or `SUT`). The `SUT` is configured to take a specific state, along with any other variables that are required for the test.
- **Act**: this is where you invoke the code (unit) you would like to test. The SUT that has been prepared in the previous section is used to call one of its methods. The output of the method is captured and saved.
- **Assert**: this is where the output of the action is verified. Based on the arrangement in the first section, the action that was invoked in the second section is expected to produce a specific result or manipulate the state of the SUT in a specific way. In this section, you assert that the result(s) of the action meet the expectation(s).

2.3.3 A simple example

Assume that we have a class called `StringUtils` which includes a method called `reverse` that return the reverse of an input string. For instance, if this method receives a string `xyz`, it should return a string `zyx`. Now, without knowing the exact implementation of that method, we can write the following unit test using the AAA pattern:

```
@Test
public void testReverse () {
    // Arrange
```

```
    String input = "abc";
    StringUtils sut = new StringUtils(); // sut = system under test

    // Act
    String result = sut.reverse(input);

    // Assert
    assertEquals("cba", result);

}
```

It is important to note that you do not need to know the implementation of the method being tested in order to write the required unit tests. As in the example above, all you need to know is the signature (inputs and outputs) of the method and what it is expected to do. In fact, it is encouraged that you write unit tests for methods before you implement them. This way, your tests will not be written for a specific implementation, but rather they will be written based on what the method is supposed to do. This is a common practice referred to as Test-Driven-Development (TDD). (**?**)

2.3.4 Tips and tricks

Some tips and tricks for test-driven development.

2.3.4.1 Tip 1: Arrange section is largest

The `Arrange` section should always be the largest of the 3. In this section, all the required variables and objects are created and given the desired state required for the test. Sometimes multiple tests require the same `Arrange` section. Hence, it's a common practice to extract the arrange section into a private method(s) inside the test class and then simply call these methods from the arrange section of any test that require them.

```
String input;
StringUtils sut; // sut = system under test

// this method will be used in the arrange section of the tests
private void initialize(){
    input = "abc";
    sut = new StringUtils();

}
```

```java
@Test
public void testReverse () {

    // Arrange
    initialize(); // everything we need is arranged using this single line

    // Act
    String result = sut.reverse(input);

    // Assert
    assertEquals("cba", result);

}
```

2.3.4.2 Tip 2: Act is usually a single line

The Act section should normally be a single line of code that invokes the method being tested. In most cases, if the Act section is more than one line of code, you should immediately think about refactoring the unit test (breaking it down into smaller unit tests). The main reason for this is that, if the test fails, it is usually difficult to tell which part of the act section is responsible for the failure of the test.

2.3.4.3 Tip 3: Order is important

When using the AAA pattern, you do not have to start writing your test code at the beginning: the 3A sections should always come in that order; `Arrange` > `Act` > `Assert`. However, you do not have to start writing them in that order. Sometimes, it makes more sense to start either from the act or the assert section, based on the system and unit you are testing and the way you understand it.

2.3.4.4 Tip 4: Avoid multiple AAA

Avoid multiple Arrange-Act-Assert sections in a single test. In some cases you find a test that repeats each section more than one time. It could look something like this:

`Arrange` > `Act` > `Assert` > `ActAgain` > `AssertSomethingElse` > `ArrangeAgain` > `ActOnceMore` > `Assert`

If you are struggling to understand the previous line, imagine trying to understand the code in the test that actually follows that pattern! If a test contains more than one Act sections mixed with multiple `Assert` and/or `Arrange` sections, it means that the test is trying to verify more than one unit of code. This

indicates that the test is no longer a unit test, but rather an integration test, since it tries to verify the interaction of more than one units of code.

If you ever come across a test that contains multiple `Act-Assert` sections, it is always a good idea to think about refactoring it by breaking it into more than one test, each with one Act and Assert sections.

2.3.4.5 Tip 5: Avoid if statements in tests

Avoid using `if` statements in unit tests. Unit test with conditional statements are difficult to read and understand. A unit test is supposed to be a simple sequence of instructions that contains no branching. if statements in unit tests should also make you think about refactoring, i.e breaking the test into more than one test each of which verifies the outcomes of one of the branches in the original test.

2.3.4.6 Tip 6: Annotate

It's a common practice to annotate each section with it's name as a comment, as shown in the example above. Annotated tests are much easier to read and maintain.

2.3.5 Building a simple application with simple tests

Let's put the knowledge you have gained so far into practice. you will build a class with two methods and then write some unit tests to check that all is working as expected. Follow these steps:

1. Create a new project: `File` > `New` > `Java Project`
2. The `src` source folder has been created by default. Create another source folder by right-clicking on the root element of the hierarchy: `New` > `Source Folder`.
3. One of the folders will contain the code under test, while the other one while have the tests.
4. On the code under test folder, create a class `New` > `Class` with two methods: one that returns the sum of two integers and another one that :
   - One of them returns the sum of two integers. `IN: 5,3; OUT: 8`
   - The other one, gets two strings and returns a string that which is the product of linking them. `IN: aab, bbc; OUT: aabbbc`
5. Create the Test case by *right clicking* on the project, `New` and then select `JUnit Test Case`.
6. Now we are going to fill out some fields of the *New JUnit Test Case* dialogue menu.

- **Name**. As a convention for naming test classes, you have to append *Test* to the code under test class name, see figure **??**
- **Class under test**. You can specify which is the class under test in the last field of the dialogue menu. Click `Browse...` see figure **??**
- A dialogue will ask about whether you want to include the JUnit libraries. Say `Yes` to this.
- Click on *Finish*.

7. You could test many things:

   - On the sum method, check whether the output is `8` when the inputs are `3` and `5`
   - On the string concatenation method, check the output is `aabbbc` when the inputs are `aab` and `bbc`
   - On the string concatenation method, check the output is `35` when the inputs are `3` and `5`
   - etc.

8. Some hints to create the tests:

   - Use `@Test` to indicate a method is a test
   - `assertEquals(String message, expected, actual)` tests if two values are equal.

A possible solution to the above exercise will be discussed during the workshop and it will be published in Blackboard by the end of week 2.

2.4 Unit Testing Reading and Writing in Marauroa and Stendhal

Here are four exercises to understand Marauroa better through the use of tests. Worlds, zones and objects are fundamental concepts in many videogames. Therefore, Marauroa provides some classes to facilitate the development of such concepts. The classes we are going to deal with can be found in:

- Object: `Java marauroa.common.game.RPObject`
- Zone: `marauroa.server.game.rp.MarauroaRPZone`
- World: `marauroa.server.game.rp.RPWorld`

We will create a new JUnit Test Case for the following four exercises. Based on what we discussed today think about what would be its best location under the `tests` package.

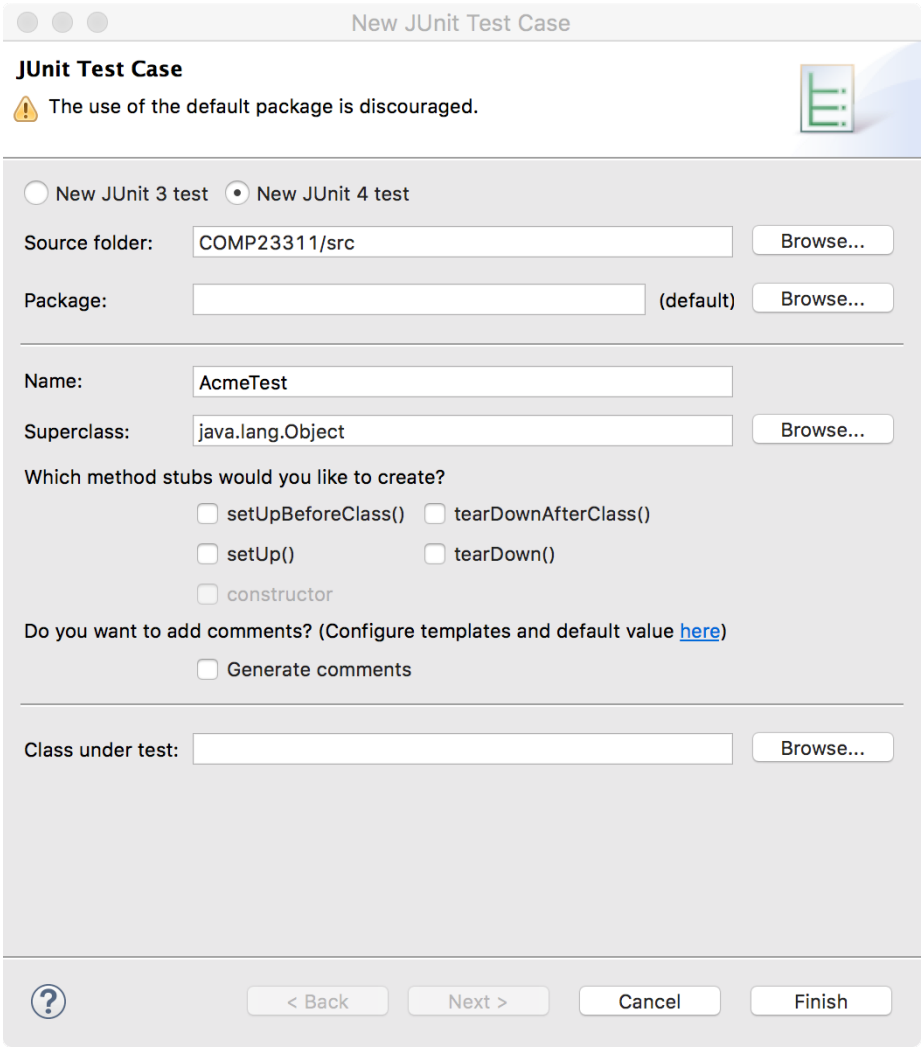

Figure 2.4: JUnit Test Case, with Test appended to the Acme class

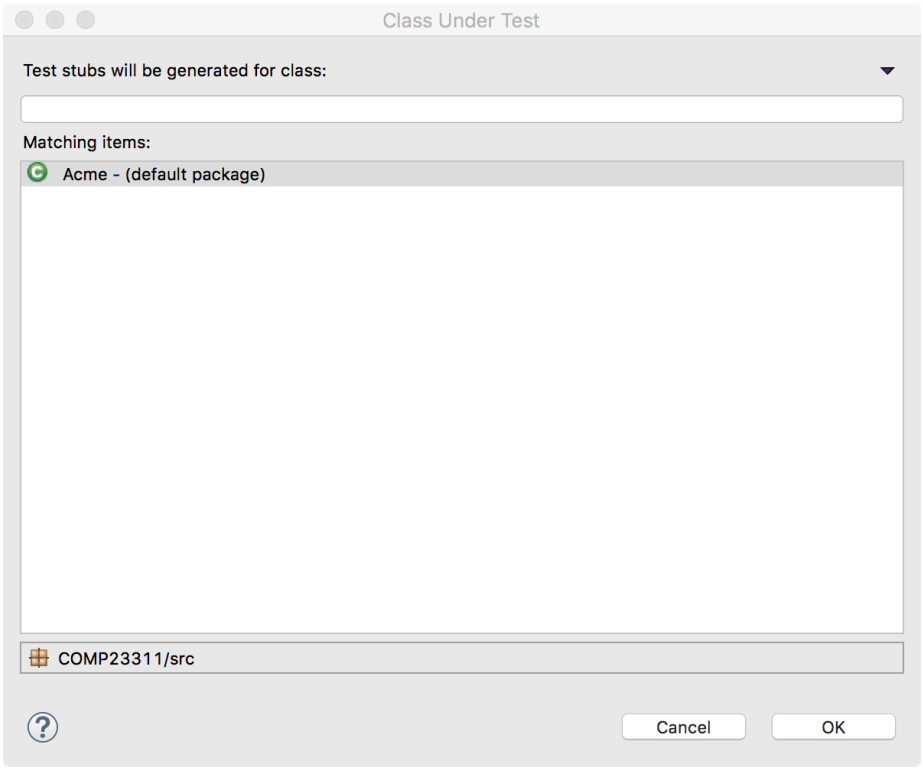

Figure 2.5: The Class Under Test in this example is Acme

2.4.1 Exercise 1: There is only one instance of World

The strategy here should be to:

- Get two instances of the `World` class
- Use a JUnit a statement to compare whether the two variables refer to the same object.

2.4.2 Exercise 2: Zones are actually added to Worlds

- Get an instance of the `World`
- Create a new `Zone`
- Add the new `Zone` to the `World`
- Use a method from the `World` class to check if our `Zone` belongs to the `World`
- Use a JUnit a statement to check the above

2.4.3 Exercise 3: Objects are actually added to Zones

- Create a `Zone` and create and `Object`
- Set an identifier to the `Object`. Tip: the `Zones` class has a method for that.
- Add the object to `Zone`
- Use a method from the `Zone` class to check if our `Object` belongs to the `Zone`.
- Use a JUnit a statement to check the above

2.4.4 Exercise 4: Objects are destroyed when removed from Zones

- Same as in Exercise until step 5.
- Remove object from zone
- Use a method from the `Zone` class to check if our `Object` belongs to the `Zone`. Use a JUnit statement.

2.4.5 Exercise 5: Reading and refactoring Stendhal tests

Note: this task is not part of your team coursework and you are not expected to commit or push the results of this task as part of your current coursework.

1. In Stendhal codebase, use the tips from the previous section to find tests for Quests.
2. Skim the tests to identify those that follow the AAA pattern and and those that do not.
3. Find and skim the largest test method in this class. What do you think of the approach used to write this test. How many act sections are there in this test.
4. Try to refactor this test to make it more readable and understandable?

Solutions to the above exercises will be discussed during the workshop and will be published on Blackboard by the end of week 2.

2.5 JUnit Cheatsheet

A cheatsheet for JUnit:

2.5.1 JUnit annotations

```
@Test
```

Identifies a method as a test method.

```
@Test(expected = Exception.class)
```

Fails if the method does not throw the named exception.

```
@Test(timeout=100)
```

Fails if the method takes longer than 100 milliseconds.

```
@Before
```

This method is executed before each test. It is used to prepare the test environment (e.g., read input data, initialize the class).

```
@After
```

This method is executed after each test. It is used to cleanup the test environment (e.g., delete temporary data, restore defaults). It can also save memory by cleaning up expensive memory structures.

```
@BeforeClass
```

This method is executed once, before all tests start. It is used to perform time intensive activities, for example, to connect to a database.

```
@AfterClass
```

This method is executed once, after all tests have finished. It is used to perform clean-up activities, for example, to disconnect from a database.

2.5.2 JUnit statements

```
fail(String message)
```

Let the method fail. Might be used to check that a certain part of the code is not reached or to have a failing test before the test code is implemented.

```
assertTrue(String message, boolean condition)
```

Checks that the boolean condition is true.

```
assertFalse(String message, boolean condition)
```

Checks that the boolean condition is false.

```
assertEquals(String message, expected, actual)
```

Tests that two values are the same. Note: for arrays the reference is checked not the content of the arrays.

```
assertEquals(String message, expected, actual, tolerance)
```

Test that float or double values match. The tolerance is the number of decimals that must be the same.

```
assertNull(String message, object)
```

Checks that the object is null.

```
assertNotNull(String message, object)
```

Checks that the object is not null.

```
assertSame(String message, expected, actual)
```

Checks that both variables refer to the same object.

```
assertNotSame(String message, expected, actual)
```

Checks that both variables refer to different objects.

Chapter 3

Debugging a Codebase

Much of your time as a software engineer will be spent debugging code, either other people's or your own. That code will often be unfamiliar to you so it is important to be able develop strategies for debugging an unfamiliar codebase. In this course we will use Stendhal as an example to help you develop better debugging skills, see figure **??**

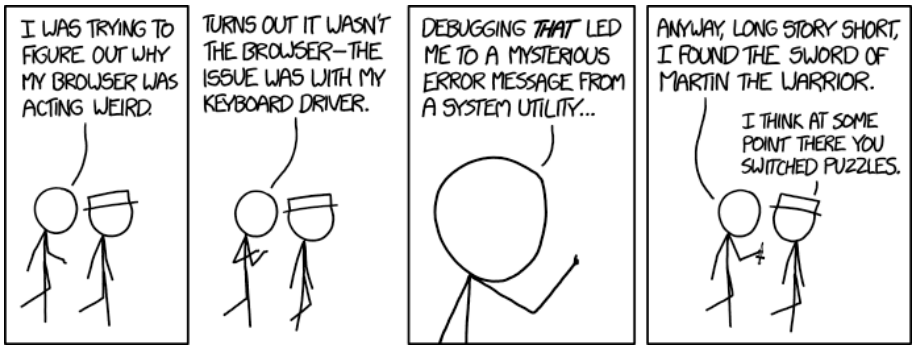

Figure 3.1: Debugging unfamiliar codebases is a routine part of software engineering. Debugging (xkcd.com/1722) by Randall Munroe is licensed under CC BY-NC 2.5

3.1 Preparing for the workshop

Welcome to the COMP23311 workshop on *Debugging an Unfamiliar Code Base*.

Today, after a short lecture introducing the core concepts, we'll be working through a number of activities in which you will be undertaking some debug-

ging tasks. Before the workshop begins, please follow the instructions below to prepare your machine for the activities we will do in the workshop today.

We are going to use the Stendhal code base to illustrate the topics under discussion. This will involve you reading the Stendhal code, and making some small changes. In order not to put your coursework at risk, we're going to use a slimmed down version of the Stendhal code repository containing the Stendhal code but without the extensive revision history.

To prepare for the workshop, you need to clone the repository and import it into your preferred IDE.

The HTTPS protocol URI of the repository is:

```
https://gitlab.cs.man.ac.uk/suzanne.m.embury/stendhal-playground-2019.git
```

If you are using Eclipse and need a reminder of what to do, you can follow the steps we took in the Week 1 workshops, when we cloned and imported the Marauroa code base. The instructions are on Blackboard, under `course content` > `Week 1`.

Eclipse users will need to create a new workspace to import this project into. This is because Eclipse doesn't allow two projects with the same name in a single workspace.

3.1.1 Introduction to the Workshop Activity

In this workshop, we will look at techniques for debugging unfamiliar codebases such as those encountered throughout the COMP23311 (`marauroa` and `stendhal`), when contributing to open source projects or when working with other legacy codebases (e.g. as part of an industry development role).

Note that unlike the other COMP23311 workshops so far, this workshop will focus on debugging the `stendhal` codebase rather than `marauroa`.

This workshop should have direct application in the first team coursework exercise.

The workshop builds on techniques given in Workshop 2 for navigating large, unfamiliar codebases. In this workshop you will:

- Systematically develop your understanding of a reported error in the Stendhal codebase through execution only.
- Develop test cases that verify the problem by closely replicating gameplay.
- Use code navigation skills developed in Workshop 2 to identify possible causes of the error.
- Use a range of debugging tactics to eliminate causes, such that a possible fix can be proposed.

- Use code navigation skills developed in Workshop 2 to identify other similar areas of code that may contain analogous flaws.

We'll be assuming that, after this workshop, you are capable of carrying out the following tasks for yourself, without needing much guidance:

- Develop understanding of a reported or observed fault in a large and/or unfamiliar codebase.
- Identify appropriate strategies and tactics to identify the likely location of a reported or observed fault in a large and/or unfamiliar codebase.
- Read and write test cases as part of a selected debugging strategy.

As in prior workshops, there will be scope to work through the task at your own pace. This is an ambitious workshop task. However, you should aim to have completed steps 1-3 by the end of the workshop, or at least to have narrowed down the source of the problem considerably. Ideally, you would have a clear idea what fix should be applied. Implementing (and testing) the fix itself should add very little extra work after this, and is the only real way to prove that you've successfully completed step 3.

3.2 Workshop Activity: Working Through a Bug Report

Definition: Debugging is the process of understanding and reducing the number of "bugs" (errors or defects) in a computer system (software, hardware or a combination of the two) such that the system behaves as expected.

We'll work through a systematic process to get from bug report to resolution as follows:

1. Start with a problem
2. Stabilise the problem
3. Isolate the source of the problem
4. Fix the problem
5. Test the fix
6. Look for similar errors

Note that although these are represented as six distinct steps, the reality is that there are times when some of the steps may overlap. For example, if you decide to add test cases to help you stabilise your understanding of the problem (#2); this will probably require you to take some steps towards isolating the source of the problem (#3) as you will need to decide which portions of the code should be subjected to the test cases you are going to write.

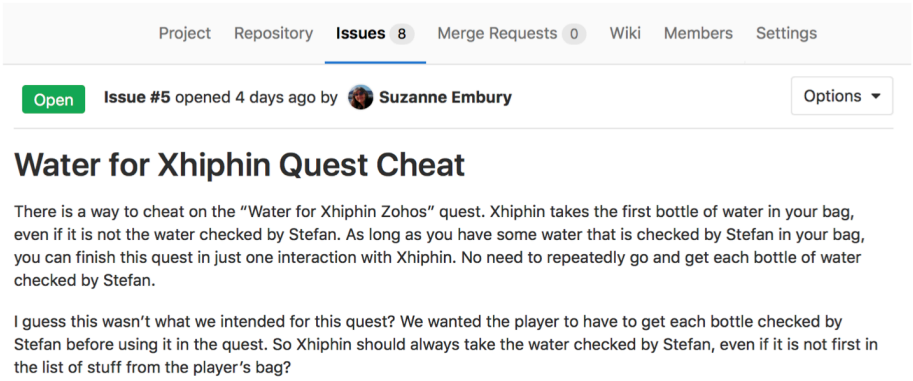

Figure 3.2: An example bug report as an issue in GitLab. GitLab issues are very similar to GitHub issues if you're familiar with those: guides.github.com/features/issues/

Figure **??** is an example bug report similar to those you should already have seen in your team coursework.

In this workshop you'll be working systematically through the steps to confirm the reported error, to understand the cause and (if possible in the time available) implement changes to the `stendhal` codebase that address the issue.

3.2.1 Start With a Problem

We'll begin this workshop by confirming that the reported bug is genuinely a problem, and understanding how to trigger it in regular gameplay. We'll do all of this **without looking at any code**.

It should be relatively intuitive to figure out how this quest should work. However, we want to avoid assumptions and so you are strongly encouraged to view the `stendhal` documentation for this quest: stendhalgame.org/quest/water_for_xhiphin_zohos.html.

Hint Some useful `stendhal` commands (you'll need to make yourself an admin user to use these):

- `/summonat [player] bag [quantity] [item]` - Add some quantity of an item to a player's bag.
- `/teleportto [player|NPC]` - Move your player to the location of another player or NPC.
- `/alterquest [player] [questslot]` - Sets the specified quest to `null` (not accepted, not completed) for a given player. \end{smitemize}

For a full list of admin commands (and details of how to make a player an admin user) see: stendhalgame.org/wiki/Stendhal:Administration

To replicate this bug, you'll want to teleport to the characters `Xhiphin Zohos` and `Stefan`. You'll want to summon the item `water`. You may also find it helpful to summon additional items to act as clear separators in your player's bag, e.g. `chicken` or `potion`. The name of the quest slot is `water_for_xhiphin`

To get the quest from Xhiphin, you'll need to engage him in conversation: *"hi"*, *"quest"*, and *"ok"* (in that order) should accept the quest. Likewise *"hi"* and *"water"* should result in Stefan checking the water for you.

The item at the top-left of your bag grid is the "first" item in your bag.

WARNING: Problems Becoming an Admin User?

Make sure to edit the `data/conf/admins.txt` file before running the Stendhal server. The Server won't pick up changes to this file while it is running.

`[OUTPUT]` To demonstrate completion of this step, write a statement that summarises your current understanding of the reported problem.

3.2.2 Stabilise the Problem

Following an initial confirmation that there is some odd behaviour with this quest, we now need to develop a more detailed understanding of the problem. What are the cases in which this quest behaves as expected? When does it not behave as expected?

`[OUTPUT]` Now that you have played the quest through a couple of times you should refine your original statement to something more precise that represents your new understanding of the problem.

You may find that your understanding hasn't changed much at all – if this is the case, compare with others around you to be sure that this is simply because you had a really precise statement of the problem to start with. Note also that your problem statement will likely continue to be revised as you work through the rest of the process.

Since we don't want to have to repeatedly play the game every time we make a change to the code, we'll look to develop a set of test cases that demonstrate a variety of behaviours related to this error. In this case, the problem relates to the quest *Water for Xhiphin Zohos*, so we should find the correct place to locate tests related to the behaviour of quests: `tests/games/stendhal/server/maps/quests`.

`[OUTPUT]` Write a set of test cases that demonstrate both correct and incorrect behaviour of the quest (some tests that succeed and some that fail).

3.2.3 Isolate the Source of the Problem

To locate the parts of the computer system (in this case software) that are causing the problem, we first need to select one or more strategies and tactics that we will use as tools in our investigation. For the purposes of debugging, you should think of a *strategy* as a broad approach, and *tactics* as the set of specific actions or equipment you will use to follow the strategy. Note that not all *strategy–tactic* pairs will make sense to use together.

For this workshop, we're going to suggest that you avoid the *brute force* strategy. You can also rule out the *architectural* strategy (this error is definitely in the server, not the client). Tactics-wise, the *profiler* is definitely not appropriate here. **No matter what tactics you pick, you should ultimately aim to write tests on suspected method calls to demonstrate that you have correctly identified the source**.

To try and isolate the source of the problem you should work with at least one other person. Check in regularly with your partner as you learn new things about the problem. A suggested approach is as follows:

1. Choose one strategy and tactic to isolate the source.
2. Compare with a partner – find someone taking a different approach to you.
3. Work through the code for no more than 15 minutes.
4. Discuss with your partner – what have you learned about the problem. If you've not made progress towards isolating the source, consider if you've picked good strategies/tactics.
5. Repeat as needed.
6. Try to keep brief notes as you go to record your progress.

`[OUTPUT]` Evidence of your developing knowledge about the source of the problem.

`[OUTPUT]` Once you are confident you have identified the source, if you have not already done so you must write test cases for the suspected method calls to demonstrate that you have correctly identified the source.

3.2.4 Fix the Problem

Having accurately isolated the source of the problem, the fix is usually fairly straightforward. In this case, you should be able to figure out some relatively trivial modifications to the codebase that would allow you to ensure correct quest behaviour.

`[OUTPUT]` Modify the codebase to alter, replace, or add to the problematic method call associated with this problem.

3.2.5 Test the Fix

Rerun the tests developed during Steps 2 (Stabilise the Problem) and 3 (Isolate the Source of the Problem). Do the tests now pass? If not, return to an earlier step in the process (usually Step 2 or Step 3) and try again.

`[OUTPUT]` You should be able to successfully demonstrate that both sets of tests complete without errors.

Play the game – does the quest now behave as expected? If not, why did the tests not pick this up for you? Return to Step 2 to revise your understanding of the problem and be sure that your tests accurately reflect the correct and incorrect behaviour of the quest (NOT of some specific aspect of the code that the quest uses).

`[OUTPUT]` You should be able to successfully demonstrate that the quest behaves correctly in all cases.

3.2.6 Look for Similar Errors

So far our debugging process has been *reactive* – that is, someone has reported a problem and we've tried to respond to this by identifying and fixing the flaw in the computer system that was responsible. We're now going to finish the debugging process with one final *proactive* step – we're going to go and deliberately look through the codebase to see if there are other likely parts of the computer system with similar behaviour that may also be problematic.

Using the code navigation skills developed in Workshop 2, you should work through the `stendhal` codebase to find other places in which the original (unmodified) method implicated in the reported bug is called. For each of these calls, you should try and establish what the expected and actual behaviours are.

`[OUTPUT]` Your debugging log for this workshop should contain a list of candidate calls to the implicated method, your predictions about their expected behaviours and a comparison with the actual behaviour seen during execution.

3.2.7 The 11 Truths of Debugging

Nick Parlante at Stanford University (cs.stanford.edu/people/nick) has enumerated *eleven truths of debugging*, which encapsulate some of the strategies discussed above:

1. Intuition and hunches are great – you just have to test them out. When a hunch and a fact collide, the fact wins. That's life in the city.

2. Don't look for complex explanations. Even the simplest omission or typo can lead to very weird behaviour. Everyone is capable of producing extremely simple and obvious errors from time to time. Look at code critically – don't just sweep your eye over that series of simple statements assuming that they are too simple to be wrong.
3. The clue to what is wrong in your code is in the flow of control. Try to see what the facts are pointing to. The computer is not trying to mislead you. Work from the facts.
4. Be systematic and persistent. Don't panic. The bug is not moving around in your code, trying to trick or evade you. It is just sitting in one place, doing the wrong thing in the same way every time.
5. If you code was working a minute ago, but now it doesn't – what was the last thing you changed? This incredibly reliable rule of thumb is the reason you should test your code as you go rather than all at once.
6. Do not change your code haphazardly trying to track down a bug. This is sort of like a scientist who changes more than one variable in an experiment at a time.
7. It makes the observed behaviour much more difficult to interpret, and you tend to introduce new bugs.
8. If you find some wrong code that does not seem to be related to the bug you were tracking, fix the wrong code anyway. Many times the wrong code was related to or obscured the bug in a way you had not imagined.
9. You should be able to explain in Sherlock Holmes style the series of facts, tests, and deductions that led you to find a bug. Alternately, if you have a bug but can't pinpoint it, then you should be able to give an argument to a critical third party detailing why each one of your methods cannot contain the bug. One of these arguments will contain a flaw since one of your methods does in fact contain a bug. Trying to construct the arguments may help you to see the flaw.
10. Be critical of your beliefs about your code. It's almost impossible to see a bug in a method when your instinct is that the method is innocent. Only when the facts have proven without question that the method is not the source of the problem should you assume it to be correct.
11. Although you need to be systematic, there is still an enormous amount of room for beliefs, hunches, guesses, etc. Use your intuition about where the bug probably is to direct the order that you check things in your systematic search. Check the methods you suspect the most first. Good instincts will come with experience.
12. Debugging depends on an objective and reasoned approach. It depends on overall perspective and understanding of the workings of your code. Debugging code is more mentally demanding than writing code. The longer you try to track down a bug without success, the less perspective you tend to have. Realise when you have lost the perspective on your code to debug. Take a break.

Chapter 4

Cost estimation

Cost estimation material will go here, see figure **??**

4.1 Course content

Figure 4.1: Accurately estimating how long things will take can be hard. The author of the windows file copy dialog visits some friends: "I'm just outside town, so I should be there in fifteen minutes" … "Actually, it's looking more like six days" … "No Wait, thirty seconds". Estimation (xkcd.com/612) by Randall Munroe is licensed under CC BY-NC 2.5

Chapter 5

Test first development

Material to be added here

5.1 Course content

Chapter 6

Git workflows

There are several workflows you can use in git, figure **??** should not be one of them.

This workshop will look at different workflows you can use.

Figure 6.1: Deleting your project and downloading a fresh copy is not the best workflow. Telescope Names (xkcd.com/1597) by Randall Munroe is licensed under CC BY-NC 2.5

Chapter 7

Software Refactoring

Software refactoring and migration

7.1 Preparing for the workshop

Welcome to the COMP23311 workshop on *software refactoring and migration*.

Today, after a short lecture introducing the core concepts, we'll be working through a number of activities in which you will be undertaking some refactoring and migration tasks. Before the workshop begins, please follow the instructions below to prepare your machine for the activities we will do in the workshop today.

7.1.1 Prepare your IDE

We are going to use the Stendhal Playground code base in today's workshop. so you can make changes without putting your coursework at risk. To prepare for the workshop, please run your IDE and load this project.

If you did not attend the earlier workshops where we used the Stendhal Playground codebase, you'll need to look at the workshop instructions in chapter **??**, to see how to acquire it.

7.1.2 Run the Regression Test Suite

You can't do effective refactoring without running the regression test suite frequently. Make sure you can run the test suite in this project, using the run configuration that comes with the Stendhal Playground project.

If your run configuration does not work, and you can't find and fix the problems yourself, get the help of a TA promptly, so you can get on with the activity.

7.2 Activity: Literals and Magic Numbers

First, we're going to look at a very basic code smell: the presence of literals in production code. Literal values in production code are problematic because they tend to be duplicated throughout code, and when they need to be changed it can be difficult and time consuming to identify all the places in the code that need to be updated. Literal values can also make code harder to read, as we have to guess at the special meaning of the literal.

Of course, test code is different. Literal values are expected in test code (though we should still take steps to avoid duplication of literals in test code).

To see an example of this code smell in action, search for and open the following file:

```
src/games/stendhal/server/maps/quests/revivalweeks/FoundGirl.java
```

Look at the literal values in this class. There are many, and several are repeated in a number of places.

Choose one literal (repeated or not) that you think would benefit from being represented as a constant, rather than being repeated throughout the file.

Double click on one example of your selected constant (so that the whole literal is highlighted).

Right click on the highlighted constant and select `Refactor` > `Extract Constant` from the menu that appears.

A dialogue box will pop up, asking you what name the new constant should have. The convention in Java is that constants have names in upper case, with underscores to separate words. Like this:

```
A_CONSTANT_IN_JAVA
```

Type your name for the constant in that form in the dialogue box. You also need to specify the access modifier (private will be fine for now), but you can accept the other defaults.

You should now see that the literal you had selected has been removed from this code, and replaced with a constant, defined towards the top of the class definition.

Use the undo option in your IDE and run the refactoring again, this time requesting that duplicate literals all be replaced by the new constant.

Hopefully, you can see how this refactoring has improved the design of the code, and so removed some technical debt (if you chose your literal well). Can you see how making this change could mean that some future changes will be easier because of it?

Don't forget to run the test suite after making the change, to make sure you haven't caused more problems than you have fixed by it.

A variant on this smell is the "magic number". This is a number that has a meaning that is important for the code, which which is hard to infer just by looking at the number itself. You can find several examples of the magic number code smell in the following class:

```
tests/games/stendhal/client/sound/system/ToneGeneratorTest.java
```

On the other hand, this next class has an example of a magic number that has been neatly turned into a constant, greatly improving the readability of the code it is involved in. Can you find the magic number:

```
src/games/stendhal/client/sound/system/processors/ToneGenerator.java
```

Can you find any other magic numbers in the Stendhal code? Or magic strings?

7.3 Activity: Long Methods

This code smell is based around a simple but surprisingly powerful idea: short methods are easier to understand than long ones. Take a look at the following classes to see this point in action.

First, look at:

```
src/games/stendhal/server/actions/pet/OwnAction.java
```

See if you can figure out what the methods here do. By contrast, look at:

```
src/games/stendhal/server/actions/pet/NameAction.java
```

How did the experience of trying to figure out what the `NameAction.onAction()` method does compare with the experience of reading the much shorter methods in `OwnAction`?

A good rule of thumb is that your method bodies should be no longer than a single screen's worth of text. If you can keep your methods that short, then you should be able to easily digest them. It is also a good discipline for the

developer, as it forces us to think in terms of small, self-contained chunks of logic, rather than long rambling sequences of code.

So how can we shorten the `NameAction.onAction()` method? We can't take any of the functionality out. It is all needed. So what are our options?

Once all unnecessary and duplicated code has been removed, the most useful tool we have is the refactoring called `Extract Method`. We can use this to take some of the lines of code out of this method, and put them into a smaller method. This simple change can have a dramatic effect on readability.

An opportunity in `NameAction.onAction()` is on lines 62–65. We could wrap these lines up in a method called `removeQuotes()` (or something similar). We could do the work of converting these lines into a method, but our IDE can do this job for us almost automatically, including working out what the parameters are and what the result type should be. Let's try it.

Select lines 62-65 (from beginning to end) and right click on the highlighted code. Select `Refactor` > `Extract Method` from the menu that appears. In the dialogue box, give the name you want the extracted method to have. The name `removeQuotes` seems okay to me. We can always rename it later, if we need to. We will make a private method, and will accept all the default parameters the dialogue gives us. So we can select `OK`.

Look at the code of the class after the refactoring. The IDE has created a new method, at the bottom of the class, with the name we specified. It has worked out that we need to pass the name to be processed as a parameter, and has declared the method with the appropriate signature. The `onAction()` method is now slightly shorter, and the name of the method tells the code reader exactly what the intent of the code we have extracted into it is. This makes for a double readability improvement in one simple step.

Helper Methods

Methods like this one (private methods, called perhaps just once within a class) are sometimes referred to as *helper methods*. They are there to help us write readable clear code. In the early days of computing, it would have been considered ridiculous to define a function or method that was called just once. Methods, by definition, were used to group together code statements that would be called many times. With older compilers, method calls incurred a performance overhead: the variables of the calling scope had to be put onto the heap, to be preserved while the method was called, and restored afterwards.

Modern compilers, however, cope easily with methods that are called in just one place. We no longer have to worry about the performance aspects of creating new methods, and can make as many as we need to create readable clear code.

The next essential task is to run the tests. If we have broken something, we want to find out now, while the refactoring is still in our IDE's undo buffer and while the changes we have made are still fresh in our mind.

Can you see any more opportunities to shorten this method by extracting shorter helper methods? Experiment with the `Extract Method` refactoring and see if you can reduce the size of this method even further.

A hint is given is commented out `<!-- hint -->` below, if you get stuck you can `view the source`: either the `*.html` or `*.Rmd` to see the hint.

`Extract Method` is a really powerful tool, and it is worth learning the keyboard short cuts for it, along with the short cuts for *Rename*, so you can apply it quickly and easily when opportunities for code improvements present themselves. Often, when we extract a method, a useful domain concept is showing itself that the developers had not identified before. The private helper methods we create with this refactoring can sometimes prove so useful that they become public methods, or even get grouped to form their own class (using the *Extract Class* refactoring).

7.4 Activity: Excessive Comments

Another code quality issue that can be effectively dealt with by `Extract Method` is the smell of excessive comments. While a few judicious comments, well placed, can be extremely useful in helping developers to read code accurately and quickly, anything more than this is now viewed as an indication that the code itself may not be of the highest quality. Rather than fixing the quality problems, and making the code self-documenting, the developer has felt it necessary to include lots of explanatory comments. This is a quicker solution for the developer in the short term, but leads to technical debt in the long term, as the comments age and grow out of step with the changing code.

You can see an example of this code smell in the `execute()` method of the following class:

`src/games/stendhal/server/actions/equip/EquipAction.java`

There are comments spread throughout this method, and they don't always seem to add very much value. Some of them seem to be duplicated by the calls to the logger. Others seem like they could be conveyed more effectively through extracted methods (which would also deal with the fact that this method is too long).

Can you see any chances to extract methods from this class?

The comments here help us see where we might add in helper methods, but the situation is complication by the structure of the code. We have a number of if statements, which look like they could make good methods, but they have a `return` statement in their body, which only makes sense when executed in the current method. We can't pull that statement into a helper method, and expect it to work.

In this case, you need to do a little manual refactoring first. Can you see how to move the return statement outside the body of the if-statement, but still have its execution dependent on the value of the condition in the if-statement?

A hint is given is commented out `<!-- hint -->` below, if you get stuck you can `view the source`: either the `*.html` or `*.Rmd` to see the hint.

Once you've refactored the return statements out of the if-statements, can you see any opportunities for removing the need for comments by extracting code into well-named helper methods? The goal is to produce code that reads as clearly as natural language, and that explains what the code is doing in as plain and obvious a way as possible.

When you reach this point, if you want help, let the lecturer know. She or he will demonstrate this technique on the screen share.

7.5 Activity: Applying the Refactorings Together

If you finish all the other activities before we move on to the topic of migration, you can have a try at this last activity.

In this activity, you are asked to refactor some test code. People who are learning to refactor often forget to apply it to their test code as well as their production code. But readability and ease of modification are just as important for test code as for production code - maybe even more important, because the test code is an essential tool that guides us in changing the production code. We don't have that safety net for the test code, so it is vitally important that it is clear and simple and can be seen to be correct.

Take a look at the code that tests the Ice Cream for Annie test:

```
tests/games/stendhal/server/maps/quests/IcecreamForAnnieTest.java
```

You should see a couple of the code smells we have been looking at in this code. And should hopefully have some idea of the refactorings you can use to help you improve it.

By way of contrast, take a look at this test code:

```
/stendhal/tests/games/stendhal/server/maps/quests/FindRatChildrenTest.java
```

This shows the kind of organisation we want to get the `IceCreamForAnnie` tests to follow. In the `RatChildren` test, each part of the quest is tested in a separate test case method. This means that if one fails, the other tests will still be run,

and we'll get to see which parts of the whole quest are working and which are not.

In the `IceCreamForAnnie` test, on the other hand, everything is written in one long test case. JUnit will stop at the first failing assertion in each test case, so if an assertion fails in this test somewhere near the beginning, the rest of the assertions won't get run, and we won't get the diagnostic information we need from them.

See if you can use the three refactorings we have used in this workshop to improve the diagnostic capabilities of the `IceCreamForAnnie` test, without changing its meaning.

Question if we refactor test code, what tells us when we have made a mistake and changed the behaviour of the system?

Good luck!

Chapter 8

Design for Testability

Software refactoring and migration

8.1 Preparing for the Workshop

Welcome to the COMP23311 workshop on *design for testability*.

Today, after a short lecture introducing the core concepts, we'll be working through a number of activities in which you will be using the Stendhal code base to learn some basic design for testability concepts.

We are going to use the Stendhal Playground code base in today's workshop. so you can make changes without putting your coursework at risk. To prepare for the workshop, please run your IDE and load this project.

If you did not attend the earlier workshops where we used the Stendhal Playground code base, you'll need to look at the workshop instructions for week 3, to see how to acquire it.

8.2 Understanding Test Doubles: Dummies

The simplest kind of test double is a *dummy*.

We use a dummy when the code under test is required to pass an object to some part of the fixture, but we know that the object itself will not be used during test execution. In this case, we just need a test double object that has the same interface as the required fixture object. We don't care what the dummy does, because it will never be used.

This is easiest to see by looking at some examples.

8.2.0.1 Example Dummy No. 1

In your IDE, find the code for the following method:

```
games.stendhal.server.core.config.zone.NoTeleportInTest.testConfigureZone()}
```

The test class and method name and the comments make it clear that this test case is checking that whole zones can be correctly configured to disallow teleporting in. The test checks that teleporting in is not allowed at two locations in the zone (top left and bottom right edge squares) but that teleporting out is still allowed.

The fixture for this test is a zone that is configured to disallow inwards teleports. We don't care about the other attributes of the zone. But the signature of the method for adding a configuration to a zone requires two arguments: the zone to be configured *and* a map containing attributes to be used in guiding the configuration. (Hover over the call to the `configureZone()` method to see it's JavaDoc.) The `NoTeleportIn` configuration does not need any special attributes to be set, but the method signature is inherited and the attributes must be provided whether they are needed or not.

There's no point wasting time and resources (and lines of code) setting up some fake attributes for this configuration if the test isn't going to use them. So, the coder of this test has sensibly chosen the simplest possible object compatible with the method signature: a `null` value.

In Java, as in many object oriented languages, `null` is an instance of the class `Object`, the class which is at the root of the object inheritance hierarchy and which all other classes are a sub-class of. An instance of `Object` can be used anywhere an instance of any other class is needed. This means that `null` is a match for the required data type (`Map<String, String>`) and can be given as the simplest possible attribute map to satisfy Java's strong typing requirements.

8.2.1 Example Dummy No. 2

Let's look at another example, this time in the client code. Search for the method:

```
games.stendhal.client.gui.ItemPanelTest.testCursors()
```

The very first line of this method contains an example of a `null` value being used as a dummy. The `itemPanel()` constructor takes two arguments: a slot name and a placeholder sprite. We don't care much about either value in this test, but need the `ItemPanel` instance as part of the fixture. The name is easy enough, we can give any value. (Note the carefully chosen value used in the test, which

clearly indicates to the reader of the code that the name is not important.) But we also don't care about the sprite. So the coder has used the simplest possible `Sprite` instance to fulfil the fixture requirements: a `null` value.

8.2.2 Example Dummy No. 3

Staying with the `testCursors()` test case from the previous example, can you see another dummy being used in this test—one that is not just a `null` value this time?

See if you can find it, then check your answer with a staff member or a graduate teaching assistant (GTA). Remember that a dummy is a test double that represents the simplest possible, most vanilla object that can allow the necessary fixture to be found.

8.3 Understanding Test Doubles: Stubs

What do we do when our fixture needs to be more complex than the simple dummy objects we have looked at so far? Most test doubles need to behave more like the production objects that they mimic, and have real behaviour that is invoked in the test.

Sometimes, we need to be able to control the values returned from method calls. When we can hard code a simple return value into the object, then we call the test double a `stub` object.

A stub is a version of the desired fixture class that has the same interface as that class, but returns simple, hard-coded values from its methods, rather than doing any actual game processing. This removes randomness and unpredictability from our fixture, while also giving the results we need for the test.

In older languages, stub classes have to be defined in full, just as ordinary production classes are, and we need to include some mechanism in the code to say when to use the stub class (when testing) and when to use the production class (when in production use). But OO languages, with sub-classing and inheritance, give us a more convenient way of defining stubs, right inside the test code itself.

8.3.1 Example Stub No. 1

Look at the test case methods in:

```
games.stendhal.client.entity.EntityTest
```

The methods create an instance of a class called `MockEntity`. But if you look for this class in the code base, you will not find it, and no import pulls a class with this name into the class we are defining.

Instead, this class is defined at the end of the `EntityTest` class, as a private class (lines 156–176).

This new class inherits from the `Entity` class, and so has all the same behaviour as this production code class, apart from where the behaviour is overridden and added to in this definition. The changes define the extra control we need over `Entity` instances in order to write this test effectively.

In this case, the changes are:

- Adding a new private field called `count`.
- Overriding the superclass constructor behaviour to create a Marauroa RPObject instance and give it a type.
- stubbing the method that returns the area of the entity so that all `MockEntity` instances will return a null `RectangularArea` (an example of a dummy used inside a stub).
- Overriding the `onPosition()` method so that as well as doing everything the production superclass does when this method is called, we also increment the `count` variable.

For each of these, look at the test methods on this test class, and see if you can work out roughly why the stub class is designed to have these behaviours.

Notice that this stub class both controls the state (returning a hard-coded value when the area of an entity is requested) *and* adds special control behaviour needed by the test code but not the production code (counts the number of times the `onPosition()` method is called).

8.3.2 Example Stub No. 2

Another useful Java mechanism for creating stub classes is the anonymous subclass. This is used widely throughout the Stendhal test suite for creating test doubles, and is a popular technique in general for getting code under test.

To see an example of this in the Stendhal code base, find the method:

```
games.stendhal.server.actions.admin.SummonActionTest.setUP()
```

Make sure you find the server version of this test class. There is another class with the same name in the client code, which does not contain an obvious stub.

Take a look at this simple method and see if you can work out what it does. (Anonymous sub-classes are a feature of Java that you were not taught in our

first year programming course units. You can either ask a member of staff or a GTA to explain, or you can research for yourself how this Java feature works. But don't leave the workshop without understanding this language construct and how it can be used to create test doubles.)

The `SummonActionTest.setUP()` method creates an anonymous sub-class of the `StendhalRPZone` class, and overrides one of the methods on that class: the `collides()` method that we have met in previous workshops. Instead of using the collision layer to decide whether the player or objects can be placed at the location given, this over-ridden version of the method just always returns `false`. It does not matter which location in this zone you give as parameters, this method will always say there is no collision at that point, and the object can be placed there.

Hopefully, you can see that this is a much quicker and more elegant way of ensuring we have no collisions in our zone than having to setup and configure an actual collision layer for our test zone. The stub object is created at the time the test is run, and has exactly the same behaviour as the `StendhalRPZone` class, except that it will never report any collisions for any zone created with it.

Take a look at how the `zone` instance created from the anonymous sub-class is used, and how this stub test double allows us to write the test more simply.

8.4 Test Doubles Scavenger Hunt

Work in pairs or small groups to find more examples of the different types of test double in Stendhal. Some guidance on how to do this is given below. Can you find at least:

- One additional example of a dummy
- One additional example of a stub

Write the name of the class, and the line where the test double occurs, on a sheet of paper or in a file.

When you have found an example of each type of test double (or given up) check your answers with staff or a TA, and share examples with neighbouring students.

8.4.1 Finding Dummies

To find some candidate code to examine, you can use File Search in your IDE to search for the string `null` in test code. (A good shorthand way to search through only the test classes is to use the regular expression `*Test.java` in the file name section of the search dialogue box.)

You are looking for places in the fixture setup part of a test case where a null is passed as a parameter when preparing the class under test for test execution, or when preparing a dependent object.

For other dummies, look for the use of no argument constructors, where simple instances are created and passed as parameters to the class under test, or when setting up a dependent class.

A good sign that you have found a dummy is that you could replace it with another more complex object, and the test behaviour would not change.

8.4.2 Finding Stubs

Stubs will also normally be found in test classes. Look for anonymous subclasses created during the set-up stages of a test case, where literal values are used to specify return values from methods.

Stubs can also be implemented as named private classes. This normally happens when we need to create several instances of the same stub, for use across multiple test cases, perhaps. If we only need one instance of the stub, we don't need to refer to it in other parts of the code, and it is fine for it to be anonymous.

Sometimes stubs need to be used by several classes under test. In this case, they can't be declared as private classes, and must be declared in their own file. Look for such classes wherever test classes are declared, but also in places where test helper code is located.

Can you find the packages containing test helper code in Stendhal?

Look at the names used for the non-anonymous stubs you find. Have the authors of the code made the role of these classes as test doubles clear from the name?

8.5 Understanding Test Doubles: First Experiments with Mock Objects

For this short activity, you are asked to look through the test code for the HandToHand class:

`games.stendhal.server.entity.creature.impl.attack.HandToHandTest.java`

Below is a list of the names of each of the test case methods in this class. Take a sheet of paper and draw a line down the middle. On one side, write the names of the methods that are using mocks, and on the other side write the names of the test methods not using the mock objects framework.

- `testAttack()`

- testCanAttackNow()
- testCanAttackNowBigCreature()
- testFindNewTarget()
- testHasValidTarget()
- testHasValidTargetDifferentZones()
- testHasValidTargetInvisibleVictim()
- testHasValidTargetNonAttacker()
- testHasValidTargetvisibleVictim()
- testNotAttackTurnAttack()

What do the methods that use mocks have in common, compared with the methods that don't use mocks?

Next, we're going to look at what happens when tests using mock objects fail.

Starting from the first test method, testAttack(), use your IDE's navigation facilities to jump to the definition for the method that that test case is checking. (Hint: double click on the method name and press Function key 3 (F3) in Eclipse, or right click on the method name and select `Open Declaration`.)

Comment out line 26, like this:

```
public void attack(final Creature creature) {
    if (creature.isAttackTurn(SingletonRepository.getRuleProcessor().getTurn())){
        //creature.attack();
    }
}
```

Now run the tests.

Take a look at the error message you get. Can you tell what it means?

8.6 All Finished and Nowhere to Go?

If you have finished the other activities, you can try this more challenging exercise.

The Daily Item Quest contains an annoying bug. This quest asks you to find an item set for you by the Mayor of Ados. If you can't find the item, after a week, the Mayor will allow you to request a different item. But, the bug in the code allows the quest class the possibility of giving you the same impossible-to-find item again.

Work in pairs or small groups to make the Daily Item Quest functionality testable, using the test double techniques we have covered in the class, so that this bug can be made visible.

You do not have to create a complete implementation. Just sketch out the changes you would make, in sufficient detail to understand the costs and benefits.

There is no single right answer to this. Several approaches could work. If you are unsure, just try one and see how it looks when it is sketched out. Discuss your answer with staff if unsure.

Chapter 9

Software design patterns

9.1 Introduction

In this workshop, we will look at design patterns, and their application in refactoring. A software design pattern describes a general, reusable solution to a commonly occurring problem in a specific design context. They're often useful when designing new pieces of software as they both allow us to reuse best practice from prior experience, and provide a means to discuss the design with others (a shared vocabulary). However, design patterns can be equally useful when refactoring existing codebases.

The workshop builds on techniques given in previous workshops for working with large codebases, in particular extending those in Workshop 8 for refactoring existing code. In this workshop you will:

- Be introduced to 6 of the 23 Gang of Four (GoF) Design Patterns (**?**)
- Refactor a small existing code base to apply Behavioural, Structural and Creational patterns

We'll be assuming that, after this workshop, you are capable of carrying out the following tasks for yourself, without needing much guidance:

- Identify portions of existing codebases that could be improved through the application of Design Patterns
- Describe a refactoring using a design pattern vocabulary and, where appropriate, supporting UML
- Apply design patterns to an existing codebase

As in prior workshops, there will be scope to work through the tasks at your own pace – in particular, each of the three workshop exercises is divided into

multiple stages that address first one design pattern, and then a second. You should (at a minimum) aim to have completed all tasks related to the first design pattern of each exercise.

9.2 Workshop Exercise 1 - Behavioural Patterns

This first section of the workshop focuses on applying the two Behavioural patterns introduced: Strategy, and State.

For this exercise, you'll be working with a small-scale Java codebase that's loosely inspired by some classes in the Stendhal codebase. For this first exercise you'll be focussing on a set of classes that represent pets. The main classes and their members can be represented in a UML class diagram shown in figure **??**

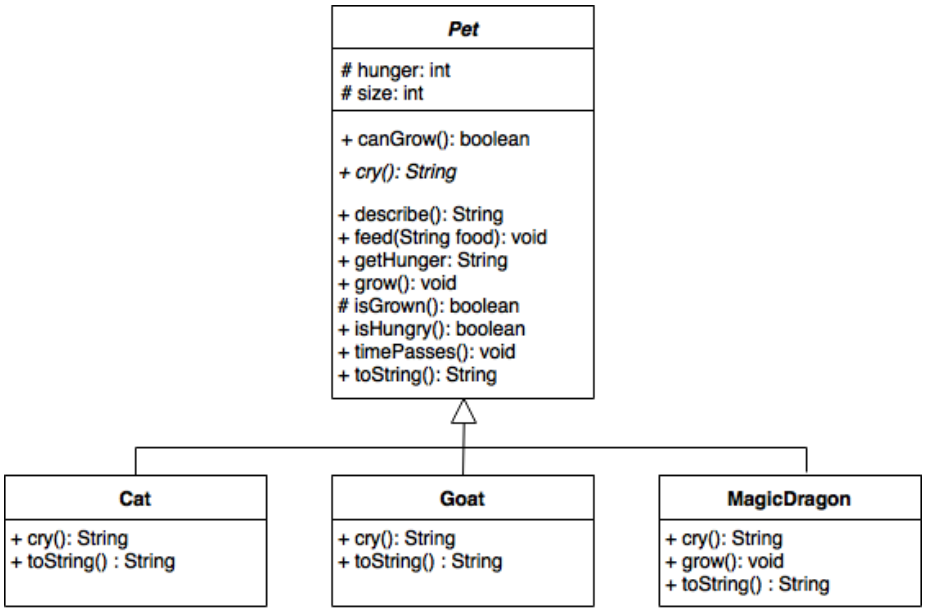

Figure 9.1: Pets, Cats, Goats and Magic Dragons

In this workshop you'll be extending and then refactoring the codebase to explore how behavioural patterns can simplify the process of adding new functionality, and can remove the need for duplicate code.

9.2.1 Exercise 1a - The Strategy Pattern

In this part of the exercise we'll be focusing on the Strategy pattern. You will modify the code in five stages:

- Add a new `Pet` class `CuddlyToy` that requires new algorithms for the growth, feeding, hunger, and crying.
- Consider how one might use sub-class/super-class relationships to avoid duplicate code.
- Implement an abstract `GrowthStrategy` that provides method signatures for growth-related algorithms.
- Implement the three concrete implementations of `GrowthStrategy` encountered so far.
- Modify the existing `Pet` classes to use the newly created strategy classes.

9.2.1.1 Stage 1 - Add a new Pet class

You've been asked to add a new Pet, `CuddlyToy`, for players that (for example) have allergies or just don't want the effort of looking after a real-life creature. The requirements for `CuddlyToy` are as follows:

- A `CuddlyToy` should not grow, they are `ADULT_SIZE` at instantiation.
- A `CuddlyToy` does not eat, and should not get hungry.
- A `CuddlyToy` squeaks, its cry is generated by a plastic squeaker

`[ACTION]` Implement a new `Pet` subclass that complies with the above requirements, and modify `PetDriver.java` to demonstrate your new Pet subtype.

9.2.1.2 Stage 2 - Design sub-class/super-class relationships to avoid duplicated code

It's clear that many of our Pets have quite different algorithms for growth. Some, like `Goats` and `Cats` grow steadily, increasing by a fixed amount over a constant time interval. Others, like `MagicDragons`, increase by a fixed amount but at irregular intervals – their growth stagnates for a while and then they undergo a growth spurt. Some `Pets`, like `CuddlyToys`, don't grow at all.

If we wanted to introduce more `Pet` types, we could quickly end up having to duplicate the code for steady, irregular or no growth across multiple `Pet` subclases. Alternatively, we could add layers of subclassing shown in figure **??**

However, this could quickly become difficult to manage, and doesn't always avoid duplicate. For example, suppose you're now asked to add a new `Bird` subtype. Birds can fly (like `Dragons`) but grow steadily (like `Cats` and `Goats`). The resulting class structure might look something like that shown in figure **??**

So, we've now potentially duplicated our steady growth code in two superclasses `SteadilyGrowingGroundPet` and `SteadilyGrowingFlyingPet`, and some new code for flying behaviours in two superclasses `SteadilyGrowingFlyingPet` and `RandomlyGrowingFlyingPet`. This definitely isn't great design.

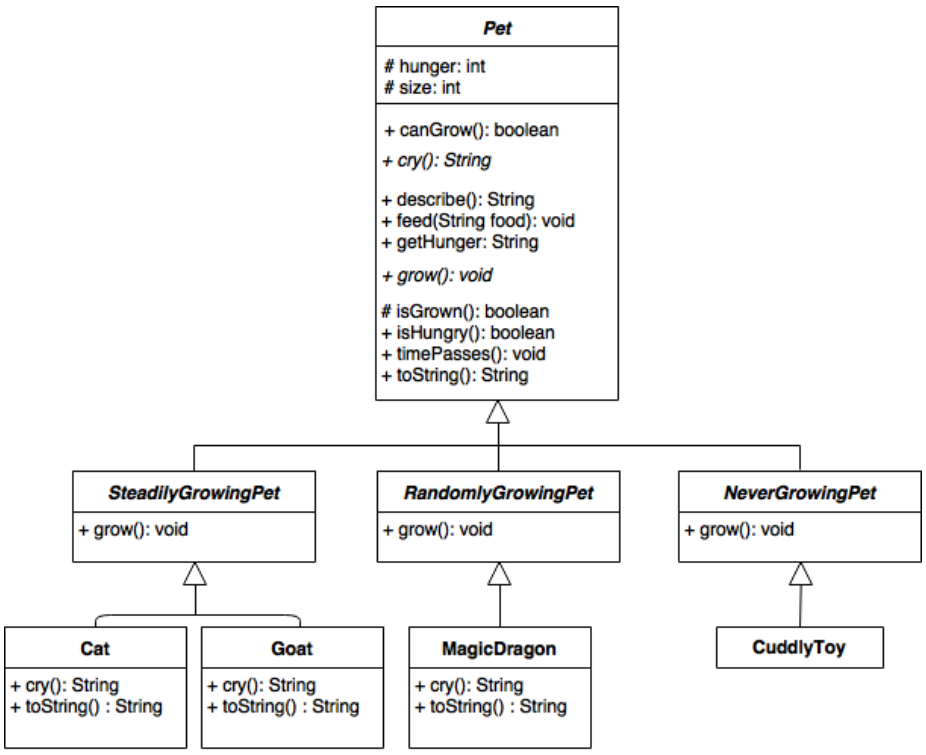

Figure 9.2: Possible subclasses of Pet

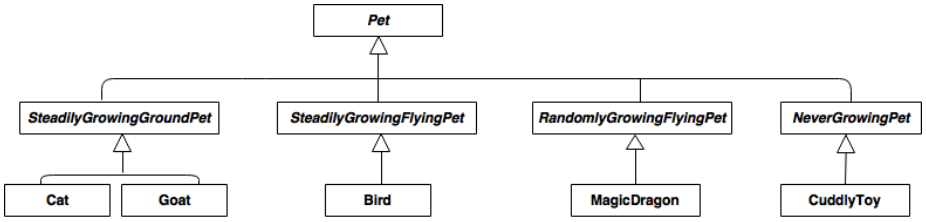

Figure 9.3: A badly designed hierarchy

9.2.1.3 Stage 3 - Introduce a GrowthStrategy

A Strategy pattern defines a encapsulates a family of interchangeable algorithms – here, our interchangeable algorithms describe different patterns of growth.

The first step in refactoring to a Strategy will be to create an abstract class `GrowthStrategy`.

[ACTION] Create a new `GrowthStrategy` class, with abstract method signatures for `canGrow()` and `Grow()`.

9.2.1.4 Stage 4 - Implement concrete growth strategies

You now need to create concrete implementations of your `GrowthStrategy` class, each representing a different growth algorithm. So far, we've encountered three growth algorithms:

- Steady growth – Grows by a fixed amount every time the grow method is called.
- Random growth – Grows by a fixed amount some random subset of times that the grow method is called.
- No growth – Does not grow, even when the grow method is called.

[ACTION] Create three subclass implementations of `GrowthStrategy`, one for each of the growth algorithms encountered so far.

9.2.1.5 Stage 5 - Modify the codebase to use our GrowthStrategy

Now we have a selection of implemented `GrowthStrategy` classes, we need to modify the `Pet` subclass to utilise these new classes. To do this, we'll add an attribute `growthStrategy` of type `GrowthStrategy` to the `Pet` class. We'll also need to add a set method for the new attribute, and modify the existing `canGrow()` and `grow()` method in `Pet` and it's subclasses to make calls to the new strategies.

[ACTION] Make the remaining code changes needed to have `Pet` and its subclasses use `GrowthStrategy`. This should now mean that there is no special-case `grow()` implementation in `MagicDragon` and `CuddlyToy`. Verify that `PetDriver.java` still behaves as expected.

The UML diagram in figure **??** should be a good representation of your codebase at the end of this migration task.

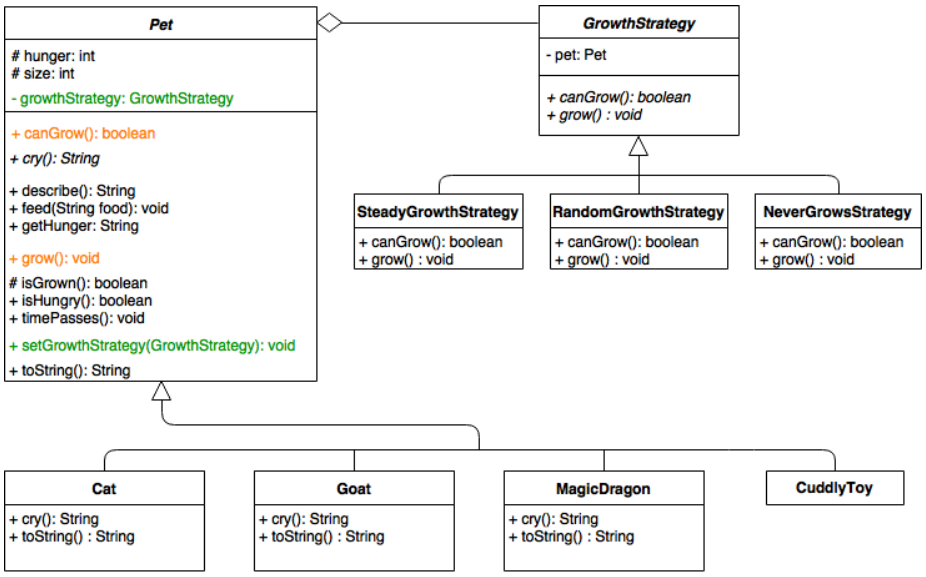

Figure 9.4: Your codebase should look something like this at the end of this migration task

9.2.2 Exercise 1b - The State Pattern

In this part of the exercise we'll be focusing on the State pattern. You will continue to modify the `Pet` codebase.

In this task, you should look to apply the State pattern to store attributes related to hunger, and algorithms that depend on those attribute values.

Note that this is the extension/secondary task for "Exercise 1 - Behavioural Patterns". Detailed instructions are therefore not provided, but a suggested approach might break the modification down into the following four further stages:

1. Identify hunger states and their dependant behaviours.
2. Implement an abstract `HungerState` that provides method signatures for dependant behaviours.
3. Implement a concrete implementations for each of the hunger states identified previously.
4. Modify the existing `Pet` classes to use the newly created state classes

You may find it helpful to make brief UML sketches as needed as you refactor the code towards the State pattern.

9.3 Workshop Exercise 2 - Structural Patterns

This first section of the workshop focuses on applying the two Structural patterns introduced: Composite, and Adapter.

For this exercise, you'll be working with a set of classes that represent habitats – places that pets might want to live. The main classes and their members can be represented in a UML class diagram in figure **??**

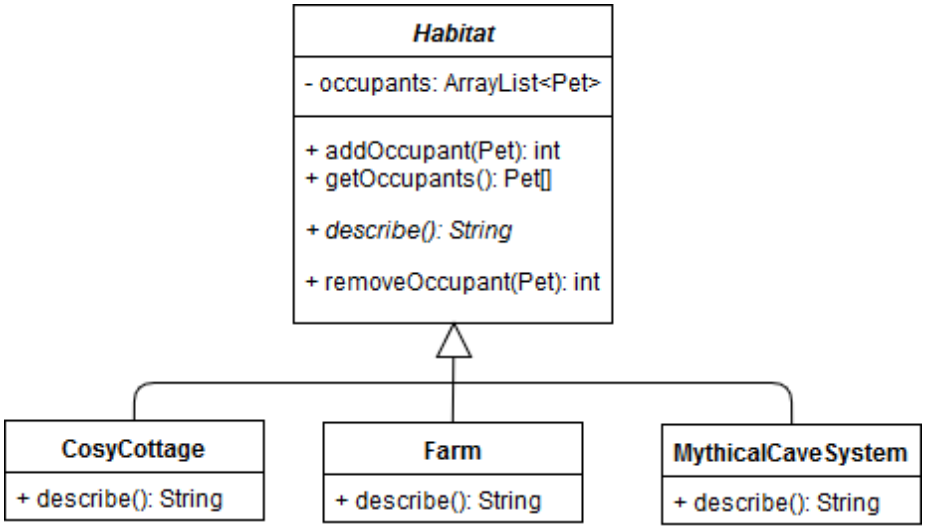

Figure 9.5: Subclasses of Habitat

In this workshop you'll be extending and then refactoring the codebase to explore how structural patterns can simplify the process of adding new functionality, and can remove the need for duplicate code.

9.3.1 Exercise 2a - The Composite Pattern

In this part of the exercise we'll be focusing on the Composite pattern. You will modify the code in 3 stages:

1. Add new `Habitat` classes, `Cave`, `Field`, and `MuddyPuddle`
2. Modify `Habitat` such that it can (optionally) contain a number of child `Habitat` objects.
3. Modify the `describe()` and `getOccupants()` methods to include the values of child objects.

9.3.1.1 Stage 1 - Add new Habitat classes

You've been asked to add some new `Habitat` classes to represent more specific places that Pets might choose to spend time. The current description for `MythicalCaveSystem` already indicates that the cave system is actually composed of three separate `Caves`. Likewise, the `Farm` is described as containing multiple fields and a barn.

You've been asked to add three specific new Habitat classes:

- `Cave` - A single cave for dragons to hide in.
- `Field` - A field with grass that goats might eat.
- `MuddyPuddle` - A patch of muddy water – goats love splashing in puddles.

`[ACTION]` Implement three new `Habitat` subclass as above, and modify `HabitatDriver.java` to demonstrate your new `Habitat` subtypes.

9.3.1.2 Stage 2 - Modify Habitat to contain child Habitat objects

We already know that `MythicalCaveSystem` contains three `Caves`, and that `Farm` contains a `Field`. We're going to use the Composite pattern to make this relationship an integral part of our class structure.

To start this refactoring, you'll need to modify `Habitat` to have a list of children; children should be of type `Habitat`.

`[ACTION]` Modify the `Habitat` class to add the new element.

`[ACTION]` Create new methods to add, remove and get children to a `Habitat`.

`[ACTION]` Modify `HabitatDriver` to demonstrate that multiple `Caves` objects can be added as a child of a `MythicalCaveSystem`, and that a `Field` can be added as the child of a `Farm`.

`[ACTION]` Modify `HabitatDriver` to demonstrate that an instance of `MuddyPuddle` can be added as a child of the `Field` (which is itself a child of `Farm`).

9.3.1.3 Stage 3 - Modify Habitat to call child methods

The final stage of our refactoring is to make sure that the descriptions of each `Habitat` are as complete as possible, and that the occupancy counts are correct (i.e. they include occupants in any part of the `Habitat`). To do this, we need to make sure that the `describe()` and `getOccupants()` of `Habitat` recursively call the same methods on any children.

[ACTION] Modify `describe()` to recursively call `childHabitat.describe()` for every `childHabitat` in the list of children for this habitat. You will need to store the result and build a new formatted description string in the parent.

[ACTION] Modify `getOccupants()` to recursively call `childHabitat.getOccupants()` for every `childHabitat` in the list of children for this habitat. You will need to store the result to build one complete list of every `Pet` in parts of the top-level `Habitat`.

[ACTION] Modify `HabitatDriver` to demonstrate that your new `describe()` and `getOccupants()` methods work as expected. In particular you should confirm that:

- A call to `aMuddyPuddle.describe()` shows only the description for the `MuddyPuddle`.
- A call to `aField.describe()` shows the description for the `Field` and the `MuddyPuddle`.
- A call to `theFarm.describe()` shows the description for the `Farm`, the `Field` and the `MuddyPuddle`.

Likewise, you should check calls to `getOccupants()` for each of the above, and check both `describe()` and `getOccupants()` for `theCaves` and `aCave`.

[OPTIONAL EXTRA] Modify `removeOccupant()` to remove an Occupant from this child `Habitats` if they aren't found in the parent.

The UML diagram in figure **??** should be a good representation of your codebase at the end of this migration task.

9.3.2 Exercise 2b - The Adapter Pattern

In this part of the exercise we'll be focusing on the Adapter pattern. You will continue to modify the `Habitat` codebase.

In this task, you should look to apply the Adapter pattern to make a legacy class `FieryMountains.java` available as a possible `Habitat`. `FieryMountains` was implemented many years ago for a previous game but has lots of neat graphics that the team want to reuse. You should use the Adapter pattern to `FieryMountains` to be used as is, as a new `Habitat`. You absolutely must not modify `FieryMountains.java`, and it must be used in your final solution (i.e. you can't just copy and paste a few values out and then just ignore it).

Note that this is the extension/secondary task for "Exercise 2 - Structural Patterns". Detailed instructions are therefore not provided, but a suggested approach might break the modification down into the following four further stages:

1. Create a new Java stub `FieryMountainsAdapter` that extends Habitat and stores a new `FieryMountains` instance as one of its attributes.

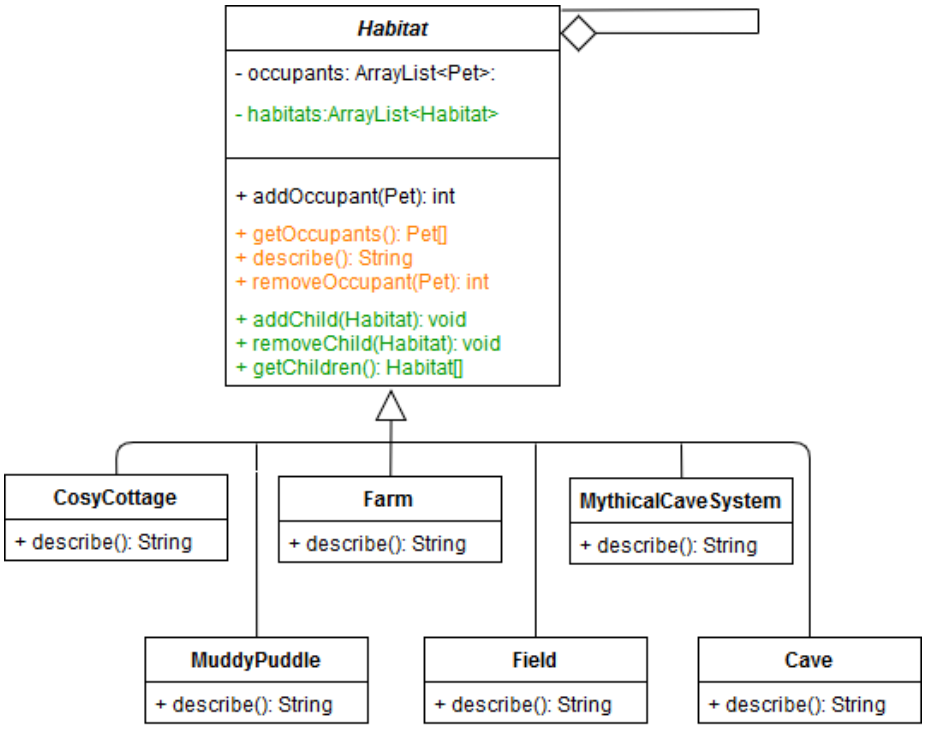

Figure 9.6: Your codebase should look something like this at the end of this migration task

2. Write a new implementation for `FieryMountainsAdapter.describe()`, that complies with the signature provided for this method in `Habitat` and calls relevant functionality from `FieryMountains`.
3. Modify `HabitatDriver` to demonstrate that fiery mountains can be added to the `ArrayList` of Habitats, and that `Pet` instances (maybe a `Dragon`?) can be added as an occupant of `FieryMountains`.

You may find it helpful to make brief UML sketches as needed as you refactor the code towards the Adapter pattern.

9.4 Workshop Exercise 3 - Creational Patterns

This first section of the workshop focuses on applying the two Creational patterns introduced: Factory Method, and Singleton.

These two patterns should be more familiar to you, from your experiences in this and other courses. For example, you've previously looked at Stendhal's own `Singleton` class `RPWorld` in one of the early workshops.

For this exercise, you'll be working with the `Pet` and `Habitat` classes you've already seen. This time we're using these classes together as part of a `Tamagotchi` application – a simple text based application that lets users look after a virtual pet for a while.

In this workshop you'll be extending and then refactoring the codebase to explore how creational patterns can allow users to control instantiation of `Pets` and `Habitats`.

9.4.1 Exercise 3a - The Factory Method

In this part of the exercise we'll be refactoring **towards** the Factory Method to instantiate different `Pet` and `Habitat` classes at runtime[1]

This is a much simpler change than previous changes, and can most likely be achieved in 3 stages:

1. Add a new `PetCreator` class that creates `Pet` objects in response to a `String` parameter.
2. Add a new `HabitatCreator` class that creates `Habitat` objects in response to a `String` parameter.
3. Modify the `Tamagotchi` class to use the new classes, passing user input in as the `String` parameters.

[1]Note that it would also be possible to achieve this using Reflection, but in this case we'll be demonstrating how a Factory Method might be applied.

You should now be able to carry out these changes without the more detailed instructions of previous exercises.

9.4.2 Exercise 3b - The Singleton Pattern

In this final part of the exercise you should consider if there is a sensible application of the Singleton pattern in any of the application code you have worked with in today's exercises.

Chapter 10

Risk management

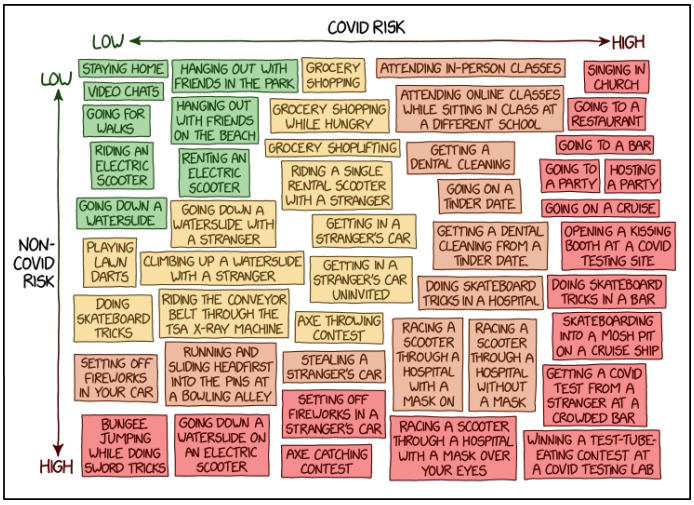

Figure 10.1: Just like life duringa global pandemic, software engineering is inherently risky COVID Risk Chart (xkcd.com/2333) by Randall Munroe is licensed under CC BY-NC 2.5

10.1 Course content

See workshop

Chapter 11

Open source challenge

11.1 Introduction

Contributing to open source software is a great way to get experience, it looks great on your CV (so will improve your chances of being invited to job interviews) and also helps you to develop your skills and knowledge. (**??**)

In this workshop, you can work alone, in pairs or in small groups to put everything you have learnt this semester into practice, aiming to fix an issue (and create a pull request) for an open source system of your choice.

The workshop builds on all the techniques given in previous workshops for working with large and/or unfamiliar codebases. Hopefully any pull request you put together today will be the first of many …

11.2 The challenge

Working alone, in pairs, or in small groups… **find and fix an issue for an open source system of your choice**. You may work on a project in any programming language you feel comfortable coding in.

11.3 Identify an appropriate project

Make sure you have a project that:

- accepts new contributions
- has a license / contribution policy that you're happy with.

  - Do you need to complete a contributors agreement?
  - Is there a specific programming style you need to stick to?
- you can easily download, build and run
- you can run relevant tests for
- is in a programming language you're familiar with
- uses a toolchain that you're (mostly) familiar with
- has an issue tracker with open issues

Some Java projects that might work for you:

- jitsi.org - A Java audio/video/chat client. Uses Ant and JUnit, hosted at github.com/jitsi/jitsi. As of July 2021 there are 46 open issues tagged with help-wanted.
- junit.org - The next generation of JUnit (Java testing). Hosted at github.com/junit-team/junit5, as of July 2021 there are 14 open issues tagged with up-for-grabs.

You'll also find Java projects listed on up-for-grabs.net. You may also find ideas on at twitter.com/yourfirstpr. Some more suggestions for open source projects, including many in Python and other languages see *Coding Your Future*. (**?**)

11.3.1 Find an interesting issue to work on

You'll need to identify an issue for your group to tackle. You should check that:

- Your issue is one that you can replicate (i.e. you should be able demonstrate to yourselves that there really is a problem).
- Your issue is one that the project team are open to contributions for.
  - Some issues may have been addressed but not released yet
  - Other issues may already be in progress
  - Some may be issues that the project team are choosing not to address (e.g. for backward compatibility reasons, or because they consider the issue to be a feature, not a bug).
- Your group agree that your chosen issue is something that you have the skills to address.

11.3.2 Fork and clone the repository

You'll need a copy of the codebase to work on. Exactly how to achieve this may vary based on how the project is hosted.

For GitHub hosted projects you'll usually fork a repository, and then clone your forked version.

See help.github.com/articles/fork-a-repo

11.3.3 Branch, change and push

Work on the issue as a team. Once you're happy that you've addressed the issue, run any relevant tests again. If all the tests pass and you're confident that you've met all the requirements for contributors, then now's the time to push changes up to your forked repository.

11.3.4 Notify the repository creator (make a pull request)

Once everything is complete, you'll want to feed your work back to the original codebase.

For Github-hosted projects you do this by making a pull request: help.github.com/articles/creating-a-pull-request. You may also want to: post to relevant mailing lists/forums, contact the project owner on twitter.

Part II

Team study materials

Chapter 12

Starting with Stendhal

Stendhal (stendhalgame.org) is a massively multiplayer online role-playing game (MMORPG) with an old school feel. You can explore cities, forest, mountains, plains and dungeons. A screenshot of some of the the game shown in figure **??**.

Figure 12.1: A screenshot from stendhalgame.org showing a selection of the 300 non-player characters (NPCs)

In this course, we use the open source code base for the Stendhal game to practice a range of software engineering skills. This chapter describes an activity that you can work through with your team to help you get started on working with the code base. You'll need to carry out many of the same steps described here as you tackle the issues set for your team for the 1st team coursework

exercise.

12.1 Introduction

In the team study sessions from week 2 onwards, you'll be working on your team coursework. The focus initially is on writing tests to make the issues your team has been asked to fix visible: that is, you will write tests that fail when the bug reported by the issue is present in the code base, and pass when it is not present. The process is:

1. First, understand the bug by replicating it manually in your local copy of the game.
2. Check to see whether any existing tests fail because of the presence of the bug.
3. If not, you need to write a test that has this property. Use the existing test suite as a source of inspiration and ideas for this.
4. Check that the test fails on the original version of the code base (i.e., it reveals the presence of the bug).
5. Figure out what is causing the bug and fix it in the production code.
6. Check that all the tests are now passing, including the one that reveals the presence of the bug.

12.2 Manually Replicating The Issues

You will find the code base much easier to navigate, and the code relating to your issue much easier to find and understand, if you first replicate the issue manually. This means running the game on your local server, to see the bug happening in the context of game play. This will give you additional keywords for searching (to add to the keywords you can extract from the issue description itself) and will help you locate the relevant tests and the source of the bug.

This, however, raises a problem. Some of the issues would require many hours of play to get a player character to the required level to be able to acquire the objects necessary and get to the NPCs involved in the issues. You don't have that time to spare if you are going to get your fix tested and sorted in the timescales given for the coursework.

Following common practice, the Stendhal team have provided facilities for supporting targeted replication of issues through manual testing. These facilities allow you to create a player character with `admin` status. This gives the player special powers — including the ability to teleport right to any NPC, to summon any item or creature, and to interrogate the internals of any item from within the game.

Information on these useful features can be found on the Stendhal wiki at:

https://stendhalgame.org/wiki/Stendhal:Administration

Think carefully within your team about how you will use these admin features, and especially what test accounts you will use. The admin accounts are specified through the contents of a file in the source code base, and therefore something that is potentially under version control. You don't want to mess up your Git history by continually committing and changing everyone's favourite test accounts in this file. A little coordination at the beginning can keep your Git history clean, while allowing everyone to be able to access at least one admin level player account.

12.3 Getting Inspiration for Writing your Own Test Cases

Most people find writing test cases for bugs in large unfamiliar code bases very challenging. This is normal and to be expected. Writing test code is quite a different style of coding than most of you will be accustomed to, and you are working blind, since you don't have much existing experience of working with this particular code. The idea of the first team coursework exercise is for you to get experience in writing fairly simple tests. It will be difficult at first, but remember that we are on hand to give help whenever you need it. Don't sit stewing in silence because you don't know what you are doing. Just ask!

A major source of inspiration for your test cases is the existing test suite. A good starting point for writing your own test case is to look through the code base for test cases that are similar to the one you want to write.

In some cases, you may find that test cases already exist for the piece of functionality you are working on, and you just need to add some additional cases to cover the specific functionality affected by the bug you are solving. That is the easiest case. If there is no test at all for you to work from, you can look at similar tests for ideas. For example, if you need to write a test that checks the properties of an object, you can look for other tests on the `Item` class and get ideas from those. Or, if you are writing a test for a quest, then you can look at tests for other similar quests, to get an idea of how these tests are structured, and what testing utility code the Stendhal team have provided to help you get started quickly.

Sometimes you might need to put ideas together from two different places to write the test you need. For example, if you are dealing with an issue that describes an error in how pets are affected by stings from poisonous creatures, you might look for tests that deal with poisoning of player characters, and tests that deal with the health of pets. Putting the ideas from these two tests together

helps you write the new test you need. You can also look at the production code for ideas when working with functionality that is not well covered by tests.

Existing test cases will also give you lots of useful tips on where to locate your new test code, whether in an existing test class or whether you need to create a brand new test class for your issue.

You should not feel embarrassed about copying and pasting existing test code and modifying it to fit your own issue. This is a normal survival technique for software engineers in the wild. Do make sure though that you understand the code you have copied, at least at a high level. Don't leave bits of code in there when you are not sure what they are for. That will just lead to brittle and slow tests.

Of course, since this is a coursework exercise, you *should* feel embarrassed about copying and pasting solutions written by the members of other teams for your issues. That would be plagiarism. Just use your own version of the Stendhal code base, and the work of your team members, to build your own tests on.

Chapter 13

Industrial mentoring

For mentors, welcome and thanks for your interest in our software engineering mentoring program. This chapter is aimed at **mentors not students** so if you're a student on this course you can go straight to chapter **??** to find out more about how mentoring works.

Figure 13.1: An outline of software engineering mentoring at the University of Manchester. Watch the full 13 minute introduction at youtu.be/H3rnYhd5hx8

A short video explaining the scheme is shown in figure **??** which describes:

- what the scheme is
- who is currently involved

- what we ask of volunteer mentors
- what you get in return
- how to sign up

Before we delve into the details of the course, here's some background on us.

13.1 About the Department of Computer Science

The Department of Computer Science at the University of Manchester www.cs.manchester.ac.uk/ is one of the oldest in the UK. The worlds first stored-program computer was developed here in 1948, by the academics who would go on to found the Department, followed by the first floating point machine, the first transistor computer and the first computer to use virtual memory. This history of innovation continues today with cutting-edge research projects like SpiNNaker (part of the Billion Euro Human Brain Project) which has built a million core ARM-powered neural High Performance Computer (HPC).

In the most recent government ranking of all research across the UK, the School was ranked 4th in the UK (based on GPA), and was assessed as having the best environment in the UK for computer science and informatics research. Since awarding the first undergraduate degrees in Computer Science in 1965, the school has awarded 10,000 degrees in Computer Science at Bachelors, Masters and Doctoral level. Our students are sought after by employees, and are active (and successful) in taking part in major coding competitions and hackathons.

As of 2022, our entry tariff is A* A* A* with an A * in Maths, and a minimum of one Science subject at A*.

13.2 About the Course Unit You Will Support

If you volunteer, the course unit you will support is our second year compulsory course on Software Engineering (course code COMP23311). This is a year-long course unit that is taken by students on all of our undergraduate programmes. The course focusses on the skills and expertise needed to be able to work with a large body of code. Students will gain experience of fixing bugs in code written by other people, adding new features to code without breaking the existing functionality, and making larger scale architectural changes to improve non-functional properties of the system - all while keeping the system up and running for its users.

For the 2021/22 academic year, we have a cohort of more than 400 students taking Software Engineering. They have already worked on individual assignments in the first semester, but in this semester, they will be working in teams of around 8. They will undertake 2 team-based coursework assignments across the semester, as well as keeping an individual reflective journal focussed on developing their personal software process, and a final examination in the summer.

As well as learning about the academic discipline of software engineering, students take this course unit to gain key employability skills, to prepare them for interviews for industrial placements and graduate positions, and to allow them to hit the ground running when they do start work.

13.3 About the Mentoring Scheme

As a mentor, you are asked to meet with your team of students twice, to work with a team of students for around an hour each time using Microsoft Teams. The visits take place in specific weeks, during time when the teams are scheduled to be working on their Software Engineering coursework. The dates/times for the visits are described in the eventbrite invitation you have received. Both sessions are one hour long.

13.4 Meeting agenda

We suggest the following ice breaker questions may be useful for getting to know your team

- What degree programme are you studying?
- What ideas do you have about your career?
- What interests you about computers/building software?
- Are you thinking of doing an industrial year, or a summer placement?
- What is the largest piece of software you have built/worked with so far?

Challenging and guiding the team:

- What are you working on at the moment?
- How are you coordinating work within your team?
- What sorts of challenges are you facing at the moment?
- What team working issues have you faced so far?
- How did you divide the work between the team members?
- How do you think your team is performing? How do you know?
- Are you on target to meet your next deadline? If yes, how do you know that?

Questions the Students Might Ask You

Questions about the mentor:

- Can you give a brief overview of your career up to this point?
- How did you get into the job you are doing today?
- What do you enjoy about your current role?
- Was there anything that surprised you about working in industry compared to being a student?

Questions about employability:

- What skills do I need to be competitive in job applications?
- What skills do you look for when you are hiring people?
- What do you know now you wish you'd known as a student?
- What are the current trends in software development?
- What up-and-coming topics do you recommend we should know about?
- What can I do to make my CV stand out when applying for placements/jobs?

Questions about team-working:

- How do you resolve technical disagreements in development teams?
- How do you deal with personality clashes within your team?
- How do you encourage people to recommit to the team?
- One of our team members isn't contributing. Would this happen in industry? How would you resolve these problems?

Questions about the process of developing software:

- What processes/methodologies do you use in your company?
- What software tools do you use and why?
- What process do you use to release software in your company?
- What code review practices do you use?
- How big is the software system you are working on now?
- What techniques do you use when working with code written by other people?
- How can we avoid getting into a mess when using Git (or other version control systems)?
- We're having a lot of trouble fixing this bug/making this change? Do you struggle with this too? How would you go about dealing with this sort of problem?

Chapter 14

What to expect of your mentor

Information for student on their mentors will be included here.

Chapter 15

Synchronising

When you're working in a team, you need to synchronise with your team repository.

15.1 Introduction

In this activity, we'll take you through the process of synchronising your local Git repositories when your fellow team members have pushed new commits to your remote. This is a team activity, which you should work through together in the team study session.

In the activity:

- One team member will make a commit to their local repository.
- That team member will push the commit to the team repository.
- Everyone else will fetch the commit down into their own local repository.
- Everyone else will synchronise their local branches with the remote tracking branches.

This document will guide you through the steps needed to achieve all of these goals, explaining some of the core Git concepts as we go.

These instructions assume that no one has yet pushed any commits to your team repository. If that's not the case for your team, you'll need to carry out steps 3 and 4 before getting started on these instructions. Check the `Commits` page for your GitLab project to find out if any commits have been made since your repository was created.

15.2 Making a Local Commit

Choose one team member who will make and push the commit. That person will share their screen with their team members, who will observe and make notes of the process, giving feedback and suggestions where needed.

We're going to make a change to the following file:

```
/data/conf/admins.txt
```

This file is empty at present, but it has an important function for us as developers of the code base. If the username of a Stendhal player is added to this file, that player becomes an administrator for the game, with access to lots of capabilities that ordinary players don't have. These capabilities will be very important for replicating issues and testing the changes you need to make to the game, without having to actually play the game for long periods.

The capabilities available to admin level players are described on the Stendhal wiki at stendhalgame.org/wiki/Stendhal:Administration

You should familiarise yourself with the main ones, so you can use them in testing your code changes in game.

The first step when making any change to a code base under version control is to decide where the commit should be made. The commit we are going to make is a small self-contained change that is needed by everyone in the team. So, we are going to make the commit directly onto the main development branch. Obviously, if the change had been more complicated or affected more files, we would want to first make the change on a feature branch. But the simple Git workflow the Stendhal team use allows for commits to the development branch, and the commit is small and simple. Most importantly, the change is not to the program code of the system; we can't introduce compile errors with this change and it seems unlikely that it would cause any existing tests to fail. It is therefore probably safe to make directly onto the development branch, provided we check the effects of the change carefully before committing and pushing.

We begin by checking out the correct branch: the `master` branch. If you have done this correctly, you should see the name of this branch next to your project name in the `Package Explorer` view.

Our commit will set up the admin level players that your team wants, by editing the `admins.txt` file. In the `Git Staging View`[1], enter a description of the commit we are about to make as the commit message:

[1] Use menu option `Window` > `Open View` > `Other` > `Git` > `Git Staging View` to open this view.

```
Set up admin player for manual testing by dev team
```

Now we'll prepare the code change that will become the commit.

Every team member will need access to an admin level player on their own test server, but you will all need to use the same `admins.txt` file that is checked into your team repository. So you need to decide your strategy for this now. You can do one of the following:

- Add one user name to the `admins.txt` file (such as 'testplayer'). Everyone must create a player with this username in their local server for their own testing. Or,
- Everyone tells the person making the commit their preferred admin user name, and all those names are added to the `admins.txt` file now. The file should be formatting with one name on each line, and no punctuation surrounding them. Everyone creates a player with one of these names in their local game server for their own testing.

When you have created an `admins.txt` file that fits your team's requirements, save it.

The next step is to run the test suite, and the build process, to check that the change has not broken something unexpected. If all tests pass (or, at least, no new failing tests have appeared) you can go ahead and make the commit.

The `admins.txt` file should now appear as an "unstaged change" in your `Git Staging View`. Drag it into the "staged changes" box (taking care to leave any other files you may have changed in the "unstaged changes" — they are not related to this commit, and we don't want them to be included in it). Make the commit (but *do not push* at this stage).

WARNING Don't forget to check that the code compiles and the tests all pass before committing the code changes. We haven't made any changes to Java source code in this commit, but we have made changes to configuration information that could potentially cause some part of the build process or some tests to fail. So, it is still important to run through the build and test check before making the commit.

Remember that if you commit broken code to your team repository, all your team members will fetch the errors into their local repository, and their own build and test process will be affected for that branch too. If this is the development branch, that can cause a lot of extra work for your whole team, and if done at the last minute before a release may even break the code that the customer sees (or, in our case, affect your team mark). So, it is a good idea to get into the habit of checking the build and test results regularly, and *always* before making a commit.

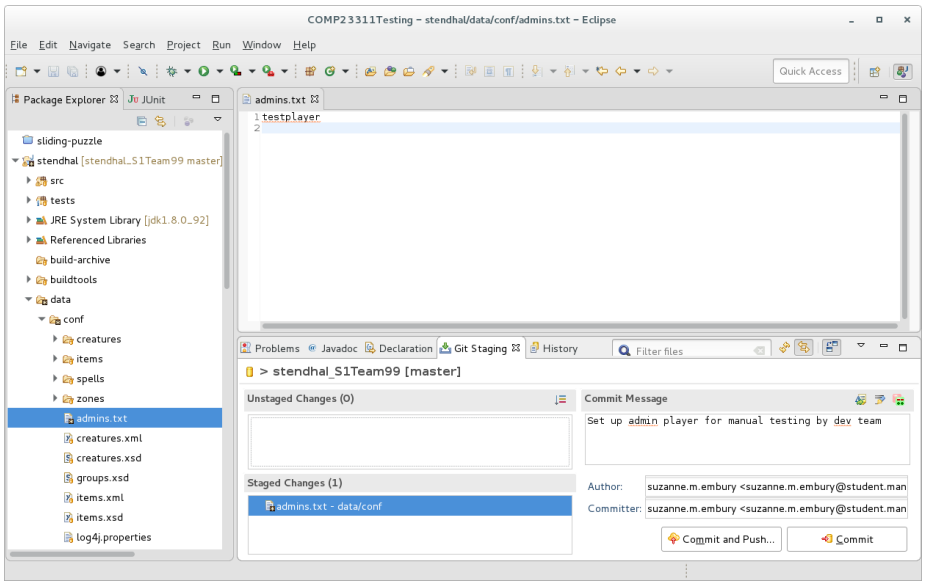

Figure 15.1: Your main eclipse window should look something like this

15.3 Step 2: Pushing the Commit to the Team Repository

The next step is to check that the commit looks okay when viewed in the context of your project history. Right click on the Stendhal project name in the `Package Explorer` view and select `Team` > `Show in History`. You should see something like figure **??**

Click on the commit you just created and check that the right files and changes have been included. We should see a small commit on the `master` branch that changes only the `admins.txt` file.

Let's compare the state of this repository with that of the team's remote repository on GitLab at this stage. In your web browser, view your team's GitLab project. Use the `Repository` > `Graph` option from the menu on the left hand side to see the commit graph. It should look something like figure **??**

You can see that the team's project is now lagging behind the state of the local repository where the commit was made. It doesn't yet have the new commit. This is reflected in the `History` view in Eclipse, where the position of the `master` branch in the remote repository `origin/master` is shown as being at the parent commit of the one we have just made.

Remote Tracking Branches

The `origin/master` branch is a special kind of branch called a "remote tracking

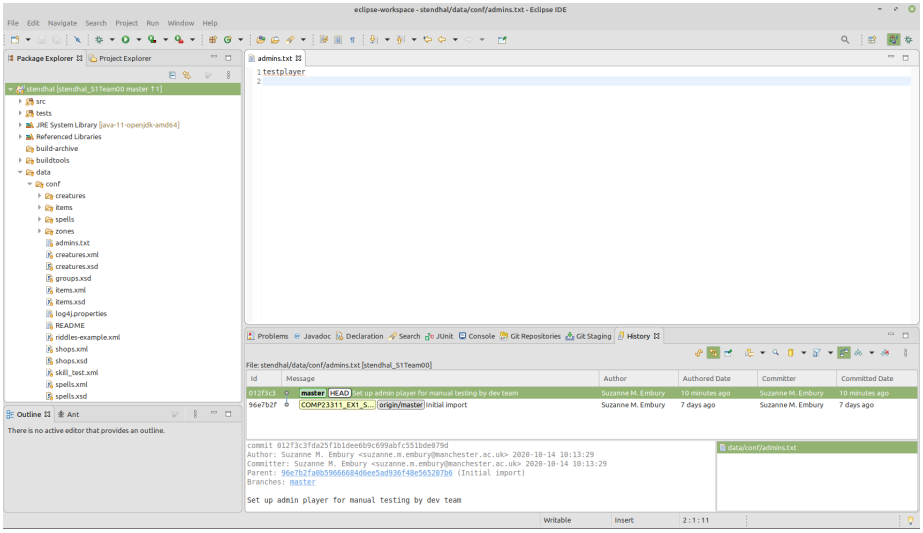

Figure 15.2: Your setup should look something like this

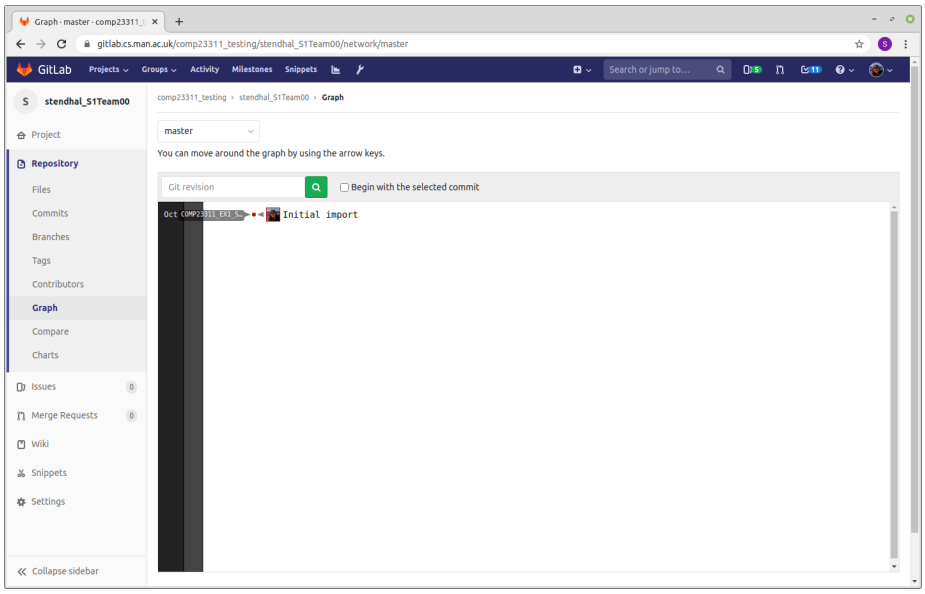

Figure 15.3: Your gitlab repository (gitlab.cs.man.ac.uk) should look something like this.

branch". It is not a normal branch that we would use for development. Instead, its role is to remember the positions of branches in the remote repository. We have two `master` branches in play here: the `master` branch in the developer's local repository and the `master` branch in the remote repository. Some of the time these branches will point to the same commit, but a lot of the time they will be pointing to different commits. So we can't use one branch to represent them both; we need one branch for the local repository position and another to track the position of the branch in the remote repository. Hence the name: remote tracking branch.

Remote tracking branches are easily identified in commit graphs because they have the name of the remote repository prepended to them. In this case, our remote uses the default name `origin`, so the remote tracking branch for the `master` branch in our team repository is `origin/master`.

Eclipse colours these branches grey in the `History` view, to indicate that they are present for information but are not for us to actively work on. If you check out a remote-tracking branch, you'll see that it is treated as a *Detached Head* checkout: you won't be able to move the position of the branch forward by making commits on it.

When you are happy that the commit contents and location in the commit graph are correct, you can go ahead and push your code to the team remote repository. Right click on your project name and select `Team` > `Push Branch 'master'...` from the menu. Use the `Preview` to check that Git is going to do what you expect (push the one commit we just made), and then make the `Push`. Eclipse will confirm the results of the operation, then you can `Close` the window.

If no commits have been made to your team repository since it was created, then this push should succeed.

It is a good habit to check that your changes have reached the team repository, and look as you expect. Refresh the commit graph page of your team project in GitLab. It should now look the same as the graph in the `History` view, shown in figure **??**

You can also check the `History` view in Eclipse, where you should see that the remote-tracking branch for `master` has now moved forward to match the position of your local `master` branch. The tag that marks the starting point of the coursework (shown in yellow in the Eclipse History view) is unaffected by any of the changes we have made and remains at its original location.

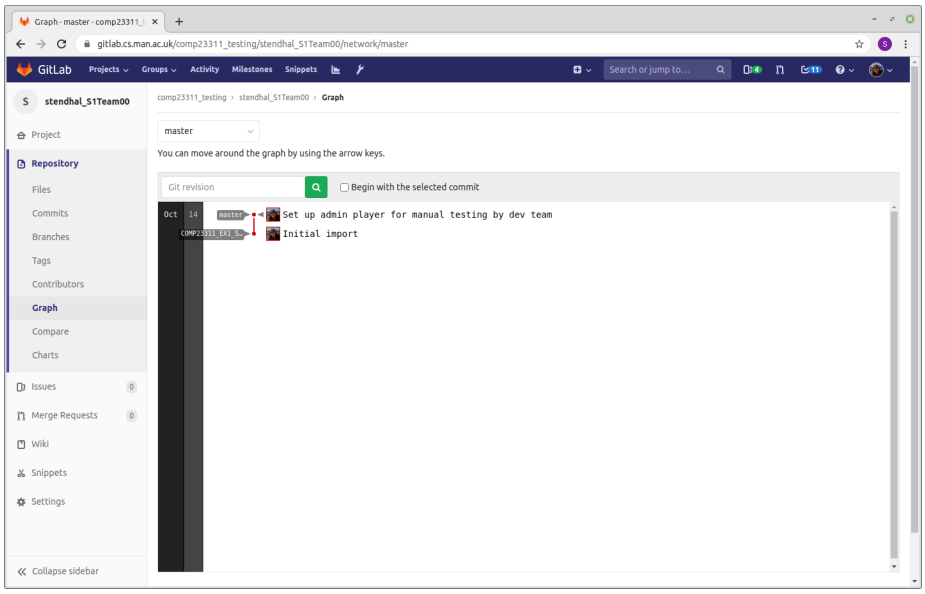

Figure 15.4: Your commit graph should look something like this

15.4 Step 3: Fetching the New Commit from the Team Repository

The remaining steps are to be carried out by all other team members. The person who made the commit that changed the `admins.txt` file should just observe from this point. Perhaps one team member who is carrying out the steps could share their screen for this part of the activity.

At this stage, the commit exists in the team repository and in the local repository of the person who made the commit, but it is does not yet exist in the local repositories of the other team members. We need to synchronise these local repositories with the team repository, so you can see and build on the work of other team members.

Synchronising your repository with a remote repository requires two basic steps:

1. First, we bring any commits and branches that have appeared or changed in the remote since we last synchronised with our local repository.
2. Then we integrate the work you have on your local repository with the work of your colleagues that you've just fetched down into your repository.

We'll carry out the first of these steps now.

Bring up the `History` view of your project in your IDE, and open the commit graph view of your project in GitLab. (The GitLab commit graph should look

the same as in the screenshot at the end of the instructions for Step 2.) If you compare the two commit graphs, you should see that there is an additional commit in GitLab that you don't have in your repository — the commit that adds the admin players. You will see that your `master` branch is "behind" the position of the `master` branch on the remote repository shown in figure **??**.

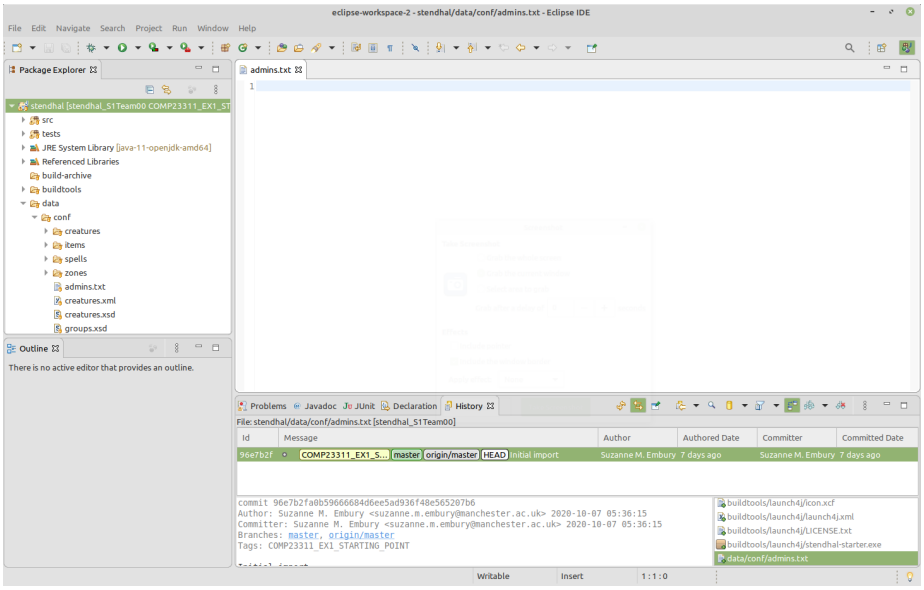

Figure 15.5: Your commit graph should look something like this

In the `Package Explorer` view, find and open the `data/conf/admins.txt` file. It should be empty. The changes made by your team mate are not yet visible to you.

Now we're going to "fetch" any new commits and branch/tags down from the remote repository. Start by checking out your `master` branch. Then, right click on the Stendhal project name in the `Package Explorer` view, and select `Team` > `Fetch from origin`. You should see a dialogue box summarising the commits that have been brought into your repository and confirming that the fetch operation succeeded like the one shown in figure **??**

Your `History` view should also have updated to show the results of the fetch operation. It should now look something like figure **??**.

If you don't see this same commit graph, make sure you have selected the option to "Show all branches and tags"[2]. If you don't select this option, you'll only see the history that is visible from your currently checked out branch (in this case, your `master` branch) and not any commits that happened after that.

[2]It is the button to the right of the green "compare mode" button in the toolbar for the `History` view. Its tool tip text begins "Change which commits to show.".

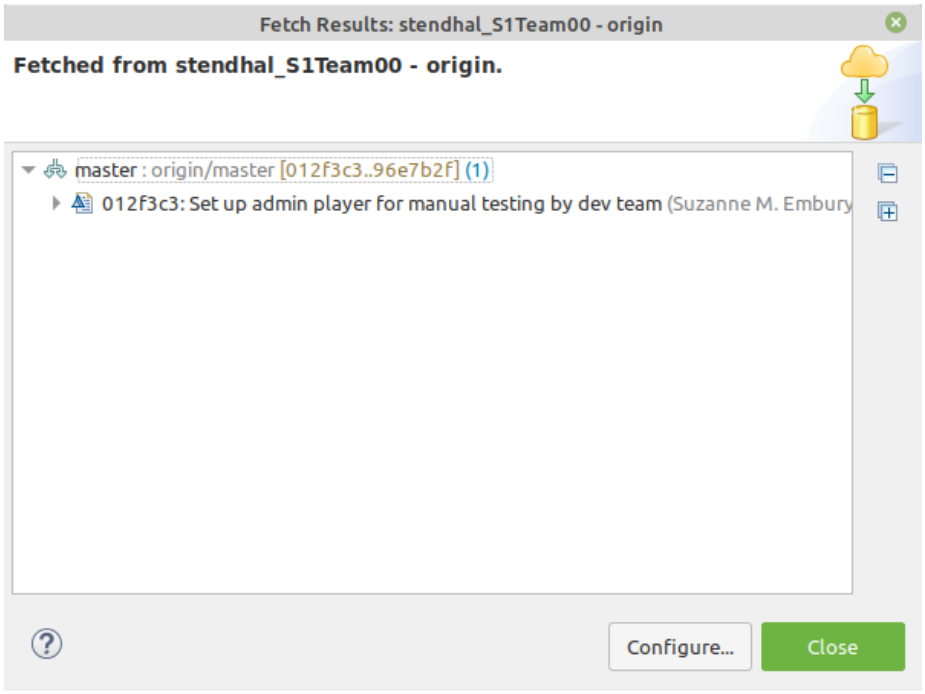

Figure 15.6: The fetch results dialog box

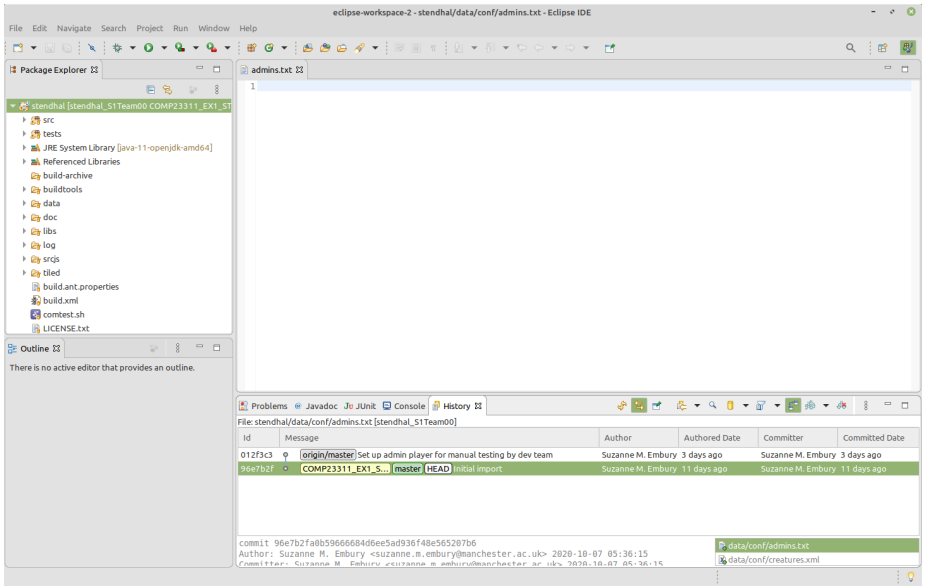

Figure 15.7: An updated History view showing the results of the fetch operation

Notice that the new commit is now present in your local commit graph. But, your `master` branch has not moved to include it. Instead, the remote tracking branch called `origin/master` has moved to point to the new commit, whereas previously it was at the same commit as your `master` branch. The fetch operation gave Git the chance to update the position of the remote tracking branch, to match its current position in the remote repository.

Note that the remote tracking branch only tracks the position of the branch in the remote repository at the time we last asked about the state of the remote: that is, at the time of the last fetch or push operation. Remote tracking branches are not magic — they don't always follow the position of the branch in the remote whenever any changes are made to it. But whenever we synchronise our repository state with the remote, the remote tracking branch will be updated.

You should never try to check out and make changes to the position of a remote tracking branch. When we make commits on a local branch, the position of the branch moves forward to point to the new commits without us having to ask. But the position of a remote tracking branch should only move when changes have been made in the remote repository. Similarly, you should not try to reset the position of a remote tracking branch, or to merge commits from other branches into it. Leave Git to keep its position updated, and concentrate on controlling the position of your local branches. They are the ones that record the state of work you are doing.

15.5 Step 4: Incorporating the Commit into Your Local Branch

Now we need to integrate the changes we have just brought from the remote into our own local branches, so that we can build on top of the work of our team mates with our own code changes.

In this case, this means we need to get our local `master` branch to point to the same commit as the remote tracking branch, so that we can see the new commit and make our own changes on a code base that includes the change it makes (the specification of the new admin player).

Because we haven't yet added any commits ourself to the `master` branch, this process is easy[3]. We can just use a merge operation. Git merge is used to bring changes from one branch into another. It always changes the branch that we have checked out. We have the `master` branch checked out[4] so that's the branch that will change its position as a result of the merge.

[3] It is a little trickier if you have made your own commits onto `master`. In this case, you're advised to use rebase rather than merge. See chapter **??** "Integrating Your Commits with your Team's Commits" for an explanation of how to do that.

[4] Checked out branches are shown in bold on the `History` view, and also have the symbolic branch `HEAD` next to it. `HEAD` is not a real branch — it is just some syntactic sugar that Git supports to give a really quick way to refer to the currently checked out branch.

Next we need to work out which branch contains the code changes (commits) that we want to include in the checked out branch. In this case, we want to bring the changes from the `origin/master` branch into the local `master` branch. So, we right click on the commit labelled with `origin/master` and select the `Merge` operation.

If this sounds like a contradiction with our earlier instructions regarding merges and remote tracking branches, then it is worth noting that Git merge operations involve two branches, but *only one of the branches is changed by the merge*. Here, we have the local `master` branch checked out, so this is the branch that will change. The other branch involved in the merge is left unchanged by it.

So, suggesting that we merge a remote tracking branch *into* a local branch is completely consistent with our earlier advice not to try to change the position of remote tracking branches. Only merges that change the position of a remote tracking branch are problematic.

In this case, the merge operation is a simple one: Git just has to move the `master` branch forward along the chain of commits until it reaches the same place as the `origin/master`. This kind of merge is called a "fast forward merge", because it is very quick for Git to do (it just changes the commit the `master` branch points to, rather than doing any actual mucking about with creating new versions of the code base) and because it involves moving the branch forward along the chain of commits.

Your `History` view should now look something like figure **??**.

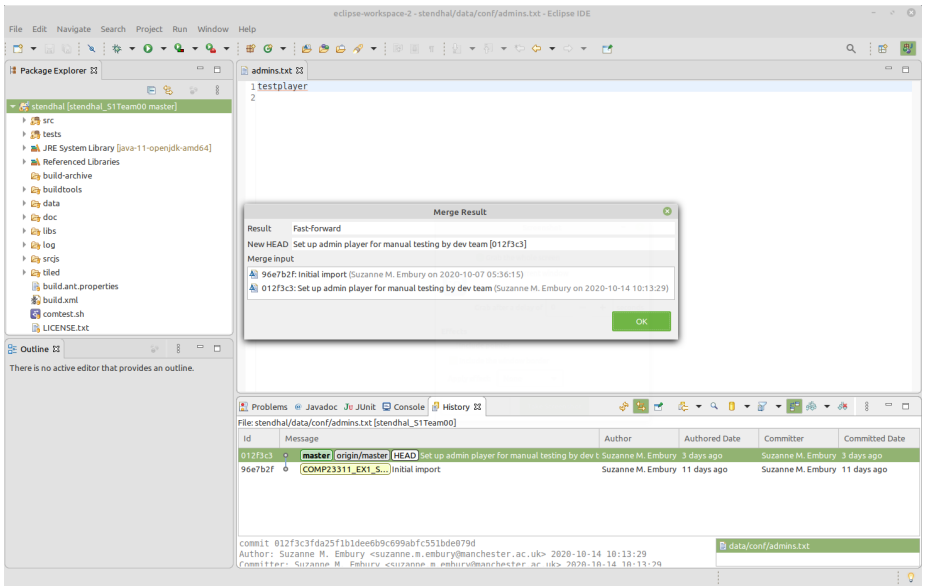

Figure 15.8: Your commit graph should look something like this

Notice how the two versions of the `master` branch are now at the same commit, which is also the checked out commit. If you look again at the contents of the `data/conf/admins.txt` file, you should see that the changes made by your team mate are now included. When you start work from this commit, you'll be building on top of the changes made by your colleague.

At this point, before beginning to make changes yourself, you should check that the code can be built and that the tests all run. If you find a problem, track down the author of the commit that introduced the error and work with them to correct it. You'll need to run this whole process again, so that the fix can get copied into everyone's local repository.

15.6 A Final Word

This illustrates the basic workflow we will use in the project, except that you will be making most of your changes on feature branches rather than on the `master` branch. The basic steps are the same. Only the branch names change.

You will find the whole process of collaborative coding using Git goes more smoothly if you get into the habit of synchronising your code base with the team remote on a regular basis. That means carrying out steps 3 and 4 described in this document. You should synchronise your code base whenever you start work on the code for the day, whenever you are about to create a new feature branch, whenever you are about to push work to the remote and (most importantly) whenever you are about to integrate work on a feature branch into the development branch.

Of course, in this activity, we covered only a very simple synchronisation scenario. As your team begins to push more code to your remote, you'll quickly encounter scenarios where the simple approach described in this document doesn't work. These more involved scenarios (and how to handle them) are described chapter **??** on "Integrating Your Commits with your Team's Commits".

Good luck!

Part III

Coursework

10 10 10 10 6.00pm, Friday 1st October 2021 6.00pm, Friday 1st October 2021 9 9 2020-03 Eclipse 2020-03 gitlab.cs.man.ac.uk/COMP23311_2020/sliding_puzzle_your-username.git https://gitlab.cs.man.ac.uk/comp23311_2021/sliding_puzzle_

Chapter 16

Individual Coursework 1

16.1 Introduction

The first piece of coursework for COMP23311 is an individual exercise designed to help you warm up your Git and Java skills after the long summer holidays, so that you are ready to collaborate with your team on the team-based coursework. It takes you through the simple Git workflow we'll be using in the team coursework later in the semester and introduces some basic Java testing and debugging concepts. You'll carry out the following steps:

- Clone a GitLab repository.
- Compile the code and run it.
- Test the code using an automated test suite to reveal a bug.
- Make a new branch in the repository.
- Fix the bug and see the tests pass.
- Commit the fix to the repository.
- Merge your branch with the development branch.
- Push your changes to your remote repository.
- Update the issue tracker to record the project status.

Detailed instructions for carrying these tasks out from within the Eclipse IDE are given in this document. We focus on Eclipse as that is the IDE used for the team coursework. You are free to use any IDE that you wish to carry out this individual coursework exercise, but we can currently only provide instructions and technical support for Eclipse.

Once the exercise is completed, you should be ready to use the same workflow on your team's repository in the first team coursework exercise.

Trouble-shooting: If you experience problems when completing this exercise, you can find a trouble-shooting guide on the Department's wiki at:

wiki.cs.manchester.ac.uk/index.php/LabHelp:Main_Page

We've provided two indexes into the trouble shooter, to help you find your way around. One is an index of error messages:

wiki.cs.manchester.ac.uk/index.php/LabHelp:Errors

If a specific error message is being reported alongside the problem you are experiencing, then you can look for it in this index, and find suggested solutions that have worked for students with this problem in the past.

The second index contains descriptions of more general symptoms:

wiki.cs.manchester.ac.uk/index.php/LabHelp:Symptoms

Use this index when something is going wrong but you do not have a specific error message to help you track down the cause of the problem.

Please report any problems you encounter that are not covered in the troubleshooter, giving details of specific error messages and screenshots where appropriate. You can report problems on the course unit forum (Piazza) or through the Live Help Queue in Team Study Sessions. We'll do our best to help!

16.2 About the Coursework

16.2.1 Key Information

The exercise will be marked out of 10, and will count towards 10 percent of your total mark for the course unit.

The deadline for the exercise is: 6.00pm, Friday 1st October 2021

You'll submit your work through your own private GitLab repository. This is created for you, and should be visible in your personal project list through the GitLab web interface at

gitlab.cs.man.ac.uk

At the deadline, we'll make a clone of your repository and run the automated marking code. You just have to make sure you have pushed your Git branches and commits to your GitLab repository by then, and make a comment in your issue tracker to let us know the work is ready to mark. There are no additional submission steps.

16.2.2 Submission Procedure

To submit your work for marking, you must add a comment to the coursework issue in the issue tracker of your project, saying:

Table 16.1: The Mark scheme for the first piece of individual coursework

Criterion
At least one new commit has been made by the student and pushed to the GitLab repository
The new commits have the author and committer e-mails set to the student's University e-mail address
A feature branch with the correct name has been created and pushed to GitLab
The feature branch appears to have been merged with the development branch
The tests all pass on the development branch
The tests all pass on the feature branch
The issue has been closed if the bug fix has been merged into the development branch and the tests all pass, and l
Total

Project ready for marking.

The time of submission will be the time at which this comment was added to the repository.

If this exact phrase is not present as a comment on the coursework issue at the deadline, we will assume that you were not ready to submit the work and you'll get a mark of 0.

You may delete and re-add the comment as many times as you like up until the formal deadline for the coursework. Once your work has been marked, we will ignore any further changes to your issue tracker; it will not be possible to request marking for a second time once marking for your project is complete and the feedback has been uploaded to your issue tracker, even if the deadline has not yet passed.

Any changes to your repository made after the final eligible marking request comment will be ignored by the marking process. So make sure you have definitely finished all your work, before you add the comment, especially if you are submitting after the deadline.

16.2.3 Marking Scheme

This coursework exercise consists of 9 steps. If you complete them all correctly, then you should earn full marks against the marking scheme shown in table **??**.

Note that we can only mark work that is present in your GitLab repository, while the earlier steps involve doing work in your local Git repository. We won't be able to see that work until you get to the final step, and push your commits and branches to your GitLab repository. If you reach the deadline with some of the steps incomplete, and want us to mark what work you have done, you'll need to jump forward to step 7 and work through as much of it as you can before the deadline to allow this.

16.2.4 Pre-Deadline Feedback

To give you a chance to see how well you have understood and applied the principles underlying this coursework exercise, we will run the marking code a little ahead of the deadline, to generate provisional marks and feedback on the work completed by that time. The feedback and a provisional mark will appear on the GitLab issue tracker for the coursework repository.

The provisional marking will take place on **Tuesday 28th September 2021, after 6.00pm**

You will then have 3 days to make corrections before the final marking process takes place, shortly after the coursework deadline on the Friday.

You don't have to do anything specific to request this provisional marking. We will mark all the repositories at this time, and provide what feedback we can based on whatever work you have done at that point.

16.2.5 Late Submissions

This coursework uses the University's standard policy on marking work that has been submitted late.

A penalty of 1 mark will be applied for each 24 hour period following the deadline that the work is late, up a total of 10 such periods. Note that for the purposes of this calculation, weekends and evenings are counted. This means that, since this coursework's deadline is on a Friday, a submission on the following Monday morning will receive a penalty of 3 marks.

These penalties will be applied until all marks earned by the student have been removed. Marks will not go below zero.

Work which is submitted more than 10 calendar days after the deadline will be considered a non-submission and will be given an automatic mark of 0. At the discretion of the course leader and the Department, we may be able to give the mark the work would have achieved if not submitted late, along with feedback explaining it. Contact the course team leader if you want to discuss the possibility of doing this.

For this coursework, the submission time will be the date and time at which you place the comment saying the work is ready for marking on the issue for the exercise. All work that has been pushed to GitLab by that date will be marked. Any commits or references that are not pushed to GitLab until after marking is requested will not be considered during marking, even if they were created or modified in your local Git repository before this.

Once the initial deadline has passed, we'll only be running the automated marking software every couple of weeks. So, there may be a delay in receiving your mark and feedback for a late submission.

16.2.6 Plagiarism

This coursework is subject to the University's standard policy on plagiarism:

wiki.cs.manchester.ac.uk/index.php/UGHandbook21:Academic_Malpractice

16.2.7 How to Get Help

Help with this exercise will be available in the two team study sessions in the week before the deadline for submission. Team study sessions are scheduled on Tuesdays at 10.00am and on Thursdays at 11.00am. Since the team coursework has not yet started, these sessions are run on a clinic basis: you only need to turn up if you need help with the individual coursework. GTAs and academic staff will be available to provide help and advice.

See Blackboard online.manchester.ac.uk for details of how to access these sessions if you are joining them online.

Help is also available through the Piazza discussion forum.

16.3 The Coursework Instructions

16.3.1 Step One: Start Eclipse

First we need to run Eclipse (or the IDE you have chosen to use for this activity, if using a different one). If working from home, you should use the VM provided by the department.

Start `2020-03`. From the command line, this is done by typing:

```
/opt/eclipse-2020-03/eclipse
```

You can also find it in the `Applications` menu, under `Programming`.

If it is the first time you have run this version of Eclipse, you will be prompted to create or select a *workspace*. This is just a folder where your Eclipse projects will live. Choose the default offered, or use the file browser to choose a location that you prefer and make a folder with an appropriate name (such as `EclipseProjects`). Or you can make a workspace that will contain only projects relating to COMP23311, such as `COMP23311Workspace`

Once you have told Eclipse which workspace you want to use, Eclipse will load.

If this is the first time you have accessed this workspace, then you will see a Welcome view, giving links to tutorials on using Eclipse. Click the cross on the tab to remove it. You should now see the basic Eclipse window, with default code views opened and ready to be used.

16.3.2 Step Two: Clone a GitLab Repository in Eclipse

The next step in the activity is to ask Eclipse to clone the required GitLab repository, and import its contents as a Java project.

Choose the `File` > `Import` menu option.

In the wizard that appears, select the `Git` > `Projects from Git` option and click `Next`.

There are two ways to import a project from Git. You can import from a local Git repository or clone a remote repository. We're going to work with a project that is currently stored remotely in GitLab, so select `Clone URI` from the list of options and click `Next`.

In the form that appears, you need enter only the URI of the remote repository that you want Eclipse to clone. Eclipse will fill in the other fields, using the components of the URI. For this activity, you should clone the repository with the URI:

gitlab.cs.man.ac.uk/COMP23311_2020/sliding_puzzle_your-username.git

where `<your-username>` is replaced by your University username. This is a personal repository that has been set up just for use by yourself, for this activity. No other students can see its contents (though the course team and GTAs can see it).

Caution Note that the URI you need to give when cloning a remote Git project is *not* the same as the URL of the GitLab page describing the project, even when using the HTTPS protocol. Make sure you have the right URI by checking against the one given on your project's main GitLab page, via the `Clone` button. It should end in the string `.git`.

Once the URI is entered, Eclipse will fill the other fields for you. (If this doesn't happen, it's likely that something went wrong when copying the link text from this pdf. Try copying it directly from values given with the `Clone` button on the GitLab page for the project, instead. If you are still getting an error, get help in one of the team study sessions.)

Click `Next`. Eclipse will connect to GitLab to authenticate your connection. Since we are using the HTTPS access protocol here, you will need to enter your University username and password at this point. (You will also be given an option to save the details, so you don't need to enter them again.)

An Aside on Protocols: GitLab can authenticate through two protocols, HTTPS and SSH. In previous academic years, we've found the HTTPS protocol to be most stable on our lab machines, but at present we expect both protocols to work well for our students on the VM you are asked to use for coursework, or from your own machine directly, in Eclipse or with other IDEs.

If you have set up an SSH key for the machine you are using and have uploaded it to your GitLab account, and want to use the SSH protocol for this coursework,

just copy and paste the SSH URI for your repository into the `Repository URI` field in place of the HTTPS URI. Note that you *must* also tick the checkbox labelled `Accept and store this key, and continue connecting?` for the SSH connection to work.

Information about GitLab SSH setup within the Department, including how to create and register a PGP key with our GitLab server, can be found on the Department wiki pages:

wiki.cs.manchester.ac.uk/index.php/Gitlab/Git_Setup

After you have entered correct login details, Eclipse will fetch some information about the remote repository. It will then ask you which branches you might want to work with in the cloned repository. In fact, all branches will be included in the clone regardless of what you select here. But Eclipse will create local versions of the branches you select (so-called `remote tracking branches`), in addition to the branches already present in the remote.

There is only one branch in this repository (the `master` branch) and that is the one we will be working with. So ensure that that branch is selected, as in figure **??**, and click `Next`.

Eclipse will now ask where you want the clone of the repository to be located (i.e., which folder in your file space you want it to be put in). This can be located anywhere you like in your file space. A common convention is to have a folder in your home directory, called `git`, in which all your local Git repositories live. If you have such a folder, you'll need to create a folder inside it, to hold the repository itself. So, you might request the clone to be placed somewhere like this:

```
/git/sliding-puzzle-activity
```

After choosing or creating a suitable location for the clone, click `Next`.

At this point, Eclipse issues the commands to create the local clone of the remote repository in the folder you selected above. (You will need to enter your login details again at this point, if you did not save them earlier, so that Eclipse can send another request to the GitLab system.) Eclipse also checks out the branch you selected, into the folder you selected, so you should also see a `src` folder, a `test` folder and some configuration files for Eclipse in the folder you created, when viewed through a file browser or at the command line.

The next step is for Eclipse to import the checked-out files (and repository) as an Eclipse project that you can work with in the IDE. Ensure 'Import existing projects' is selected as shown in figure **??**, then press `Next`.

This wizard looks for existing Eclipse projects in the cloned repository. It finds just one project (hopefully, because that is how many we put in there) so all we need to do is make sure it is selected, and then press `Finish`.

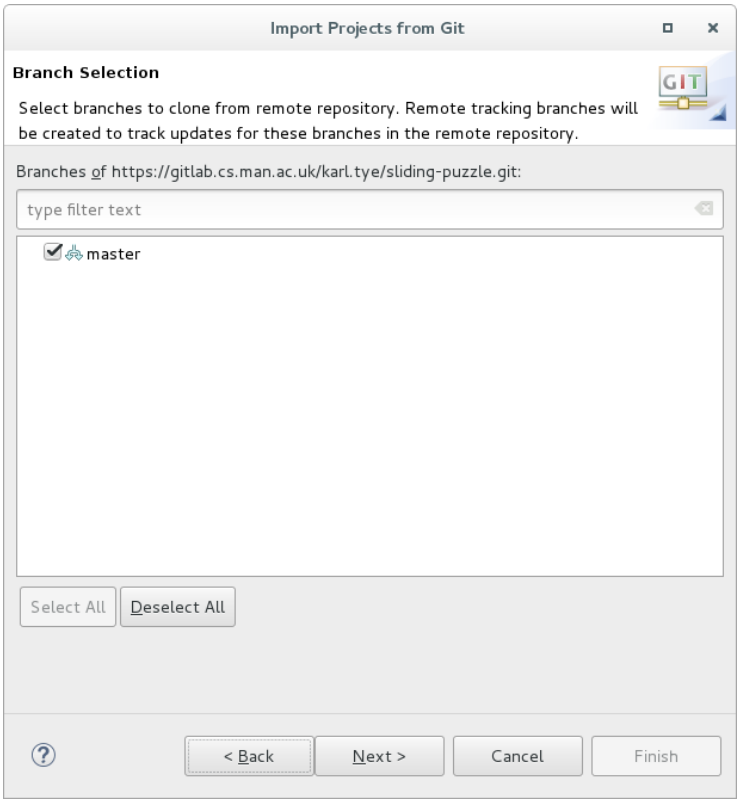

Figure 16.1: Selecting a branch

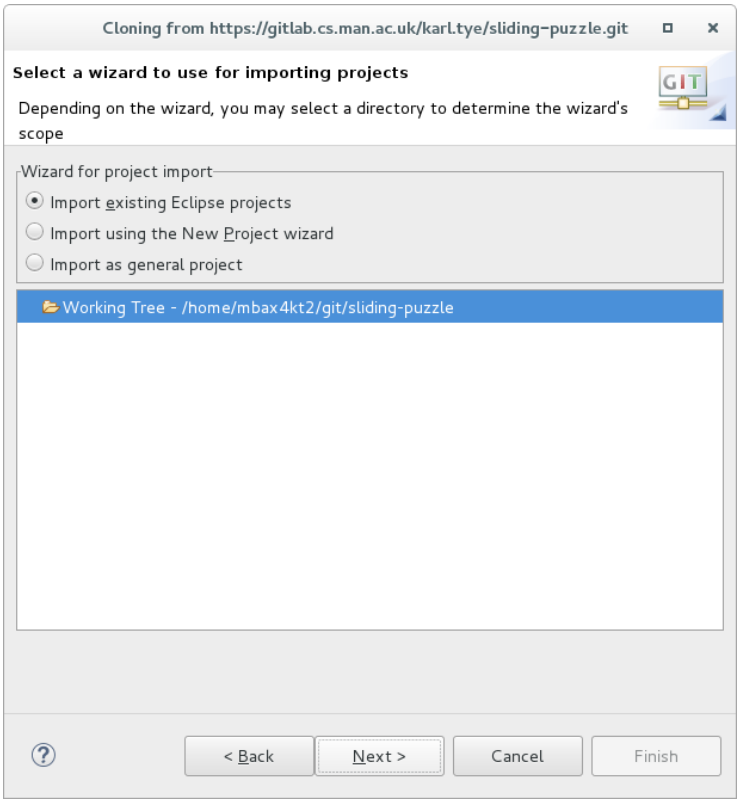

Figure 16.2: Import existing Eclipse projects

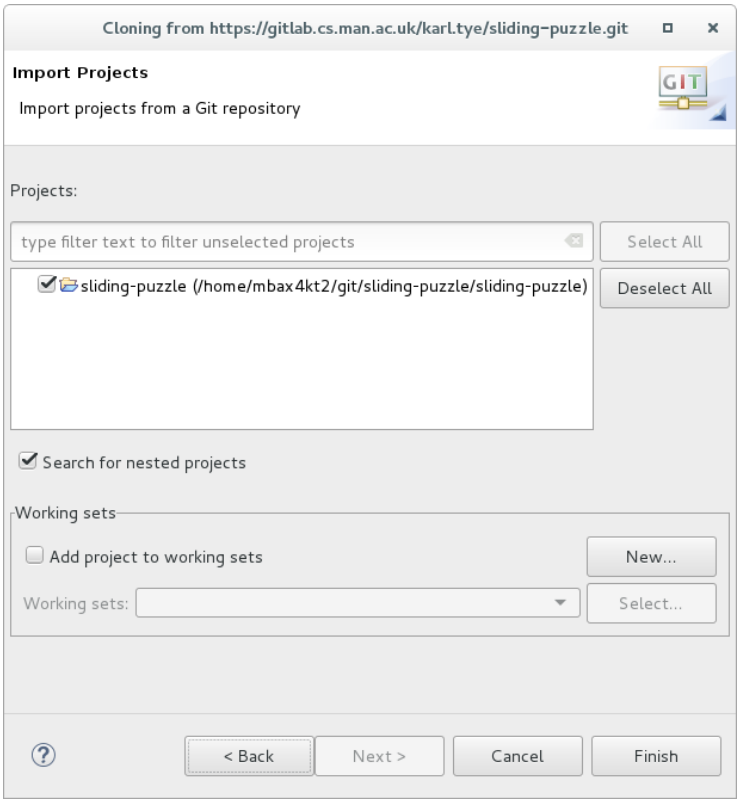

Figure 16.3: Selecting Eclipse Project to import

The project should now have been imported, and should be visible in the Package Explorer View (on the left in figure **??** along with any other projects you may have created in this workspace. You can double-click on the project to see the contents.

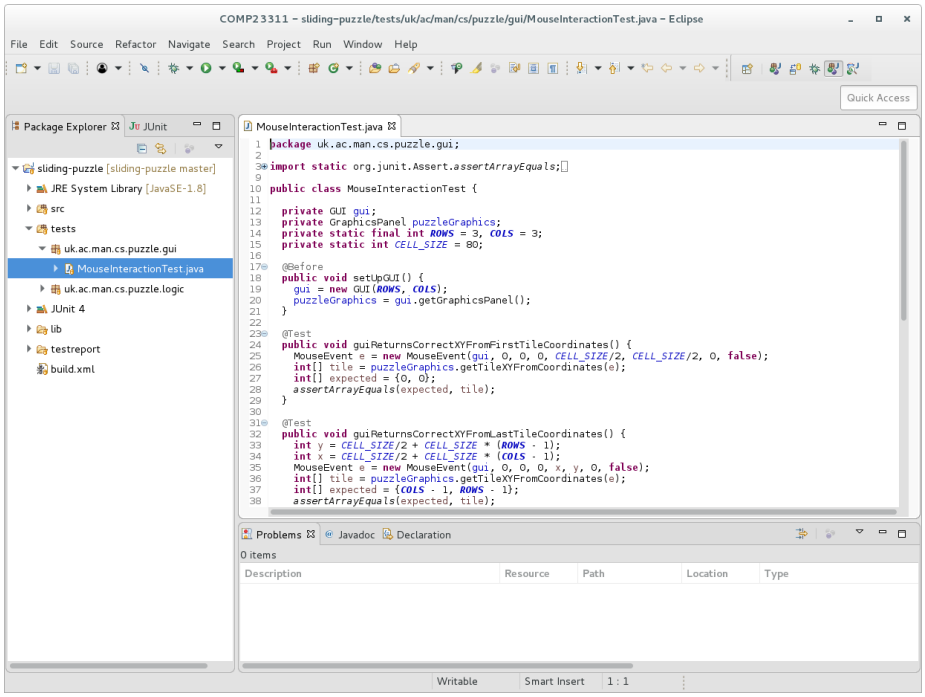

Figure 16.4: An Eclipse project loaded and visible in package explorer

16.3.3 Step Three: Identify the Bug

In your web browser, open the GitLab page for the repository created for you for this exercise and go to the issue tracker (using the menu on the left). You should see a single open issue. This issue describes a bug within the program which you are going to fix.

Your first task is to see if you can replicate this issue within the program itself. To launch the puzzle, find the `Puzzle` class in the `Package Explorer` view: it is located in the:

```
uk.ac.man.cs.puzzle
```

package within the `src` folder. Right click on it, and choose `Run As` > `Java Application`. The game should now appear in a window on your screen.

The game is a simple 8-tile sliding puzzle where the goal is to get the tiles in ascending order, left to right, top to bottom. Tiles can only be moved if they are adjacent to the free space. Players click on the tiles to move them into the space. The game changes the outlines of the tiles to bright green when the game is finished.

Play around with the puzzle for a while and see if you can replicate the bug which the issue describes. After you are done, close the program and return to Eclipse to continue the bug fixing process.

The code base we are working with follows a common (and useful) convention of making a clear separation between test code and production code (that is, the code that actually implements the functionality needed by the client). It contains two source folders. The one called `src` contains the production code, while the `test` source folder contains the test code.

Note A `source folder` in Eclipse is simply a folder that is on the build path for the project. In the case of a Java project, like this one, that means that the folder is on the Java class path. Java classes and methods stored within folders that are not on the build path will not be found by the Eclipse Java compiler and will not be executed when the code is run. Eclipse uses a special folder icon with a small package symbol overlaid on it, to distinguish source folders from ordinary folders.

The orange drum symbol indicates that the file or folder is under version control

If you double-click on the `test` folder, it will open up to show you its contents: two packages. The packages themselves contain a single class each.

Double-click on the `MouseInteractionTest` class to open it in the Editor. You should see something like figure **??**:

This class contains 5 JUnit test case methods, each testing a slightly different aspect of the program. A separate document, available on Blackboard, explains how test cases are written in JUnit. You should work through that document before starting your team coursework, but for now, we are just going to see how to run these tests in Eclipse.

There are several ways to run a JUnit test suite in Eclipse, depending on the complexity of the program you are working with. This is a very simple project, so we can just right click on the name of the test class in the Package Explorer view, and choose `Run As` > `JUnit Test` from the menu that appears. A new view containing the test results should appear, alongside your Package Explorer. It's a bit cramped there, and we can't see much of what it is telling us. Double-click on the tab of the JUnit view to expand it to fill the Eclipse window. You should see something like figure **??**

On my system, three of the tests passed (labelled with a green tick) and two failed (labelled with a blue cross). These tests give an indication as to which classes could be responsible for the bug. The bar at the top of the view is red,

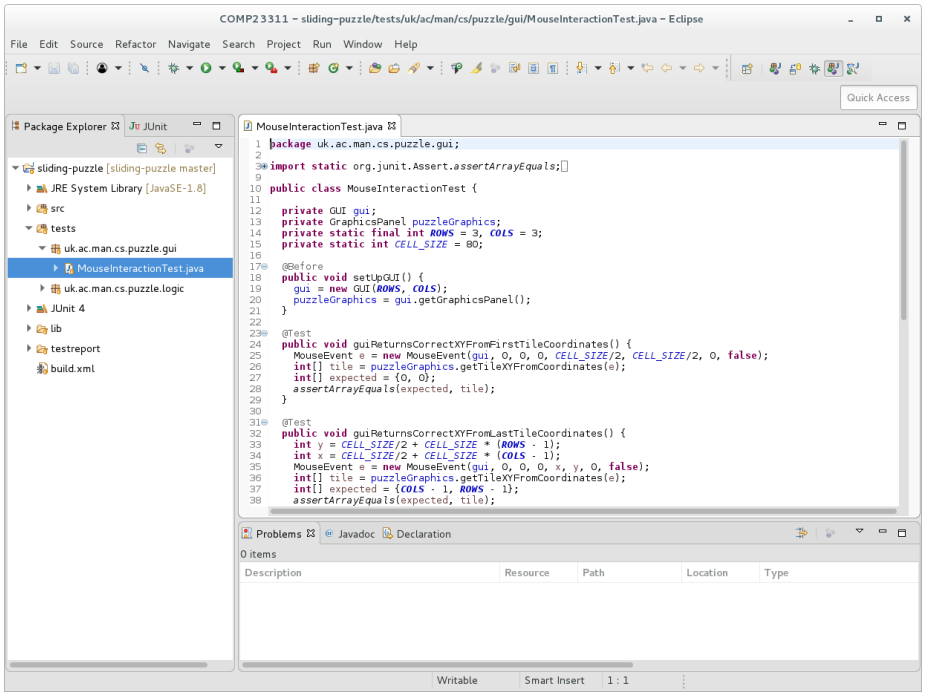

Figure 16.5: Viewing the test code

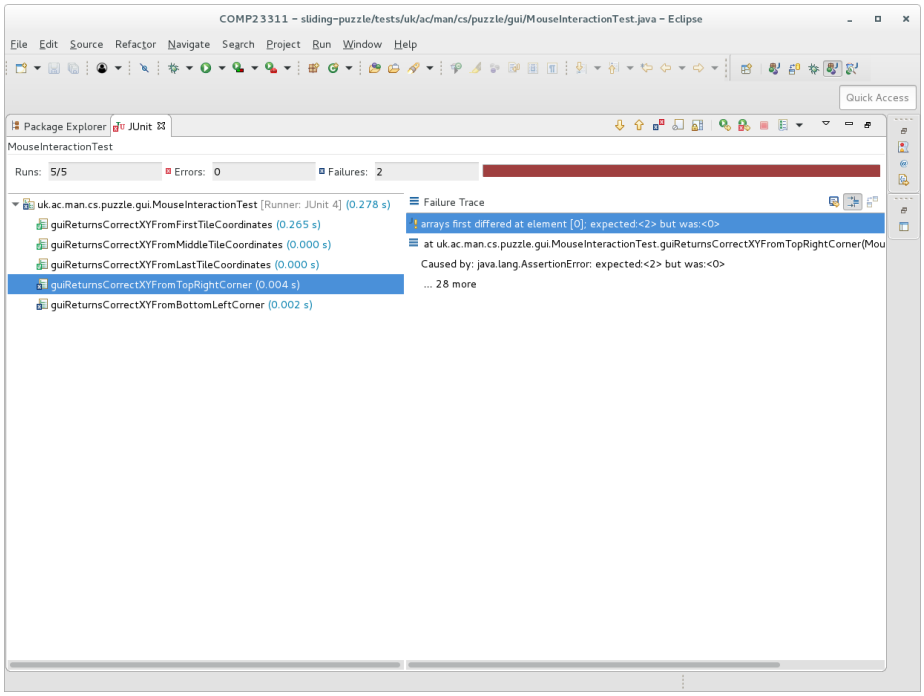

Figure 16.6: After running the tests you should see some that fail

showing that some tests failed. It is good practice to keep all tests passing, if possible, so our next step will be to make changes to the code, to make the tests pass. Hopefully, this will also fix the bug in the game.

Double-click on the tab of the JUnit view once more, to restore it to its earlier position and allow us to see the other Eclipse views again.

16.3.4 Step four: Create a Branch for Your Own Changes

We're going to make a simple change to this project, and commit it to the Git repository. In this course unit, we ask you to use a very simple Git workflow, called *Feature Branches*. This workflow uses separate Git branches to hold your changes initially, so multiple developers can work on the code at the same time without interfering with one another and so that changes from one developer can be checked for correctness before they are merged into the main development branch. So, before we make any changes, we have to create a feature branch in our local Git repository.

To do this in Eclipse, right click on the name of the `sliding-puzzle` project in the `Package Explorer` view. From the menu that appears, select `Team` > `SwitchTo` > `New Branch`. Eclipse then asks you to give the branch a name.

Branch names should describe the functional change that the branch will contain. We're going to make a change that will fix the failing tests related to the mouse interaction, so we will call the new branch `mouse-interaction-fix`.

Enter this exact name (without the quotes) into the dialogue box as the branch name. Make sure the `Check out new branch` option is selected and press `Finish`.

Caution We will use an automated process to mark this exercise, in order to provide feedback on any errors quickly before you start work on your team coursework. So, it is important that you use the exact text given for the name of this branch, to ensure the automated process can find and mark your work.

Eclipse will now ask Git to make a new branch in your local Git repository, with the name you have given. This command creates the new branch at whatever commit was previously checked out. In this case, we had checked out the `master` branch, so the new branch will be created at the same commit as `master`.

Once the branch is created, Eclipse checks out the contents, which become visible in the package explorer. Since the new branch is at the same commit we were already at the contents of the project should look exactly the same as before. The important difference, though, is that any changes you now make to these files and folders will appear on the branch you have just created. The contents of the `master` branch will remain in their original state.

One difference you should see, however, is that your new feature branch now appears in the annotation next to the project name in the Package Explorer

view. The annotation (in square brackets) shows the repository name and the branch that is currently checked out.

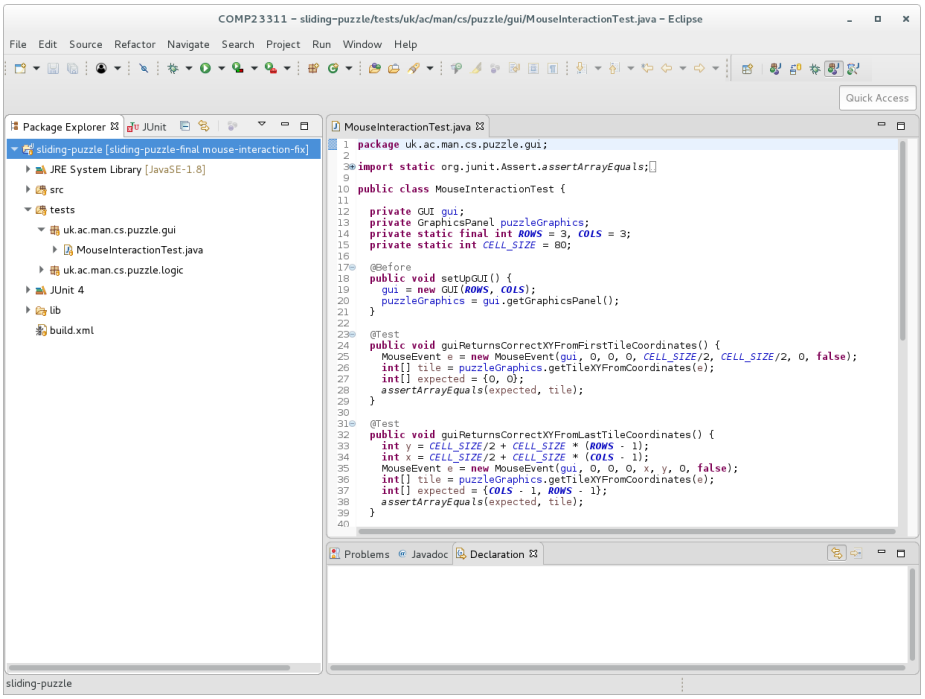

Figure 16.7: After creating branch note the annotation change in the package explorer

If you don't see this, then something has gone wrong. If you can't work out what it is, then you can get help from staff or a GTA in one of the team study sessions for this exercise.

Note *The Team Menu*

Most of the Eclipse commands for interacting with Git come under the Team menu we used here to create a new branch. You can explore around the various options to see what Eclipse allows you to do with your Git repository.

16.3.5 Step five: Commit a Change to the New Branch

Next, we're going to make a change to one of the files in the project, and commit it to our new branch. Eclipse provides a number of views and commands to help with making commits to a Git repository. One of the most useful of these is the Git Staging view. To open this, select the following option from the menus at the top of the Eclipse window:

Window > Show View > Other > Git > Git Staging

A new view with this name should now appear. Notice that there is a box on the right for the message you will associate with the commit you are about to make. It is good practice to write this commit message before you begin to make any code changes. This helps us think about the change we are about to make, and helps to keep our commits small and focussed. In this case, please add the following text to the `Commit Message` box in the Git Staging view:

```
Fix bug with wrong tiles sliding on mouse click
```

Your view should look like figure **??**.

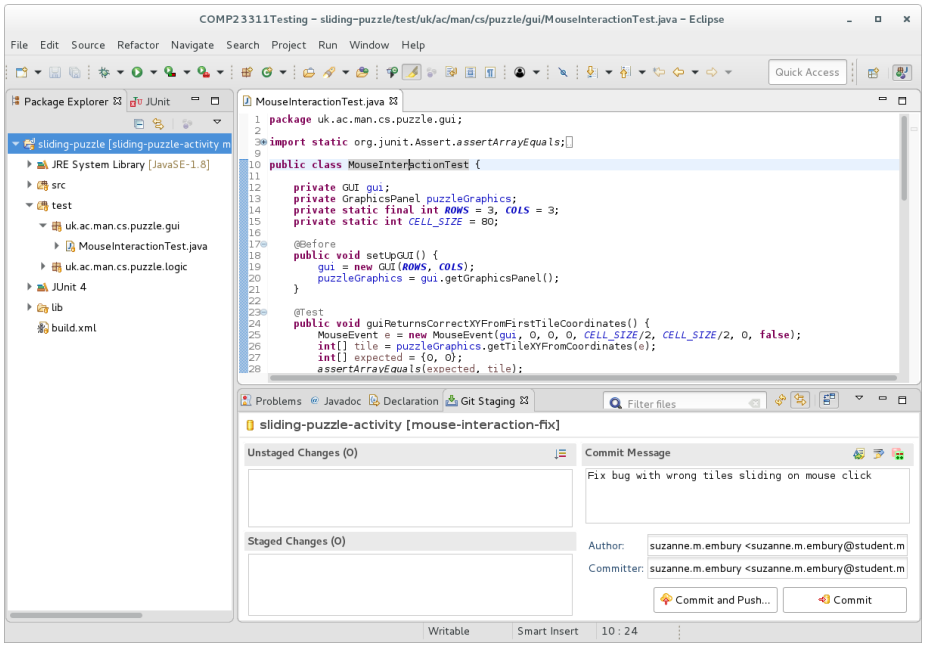

Figure 16.8: Git staging view before changes with comment

Note that, at present, the "Unstaged changes" and "Staged changes" boxes are empty. This is because (as yet) no changes have been made.

You're going to do something about that now!

For the next part of the exercise, you need to identify the bug in the production code: that is, in the files under the `src` folder. Use the information provided by the failing tests to point you in the right direction. JUnit also gives a trace of why the tests fail. These can be viewed by clicking on any of the tests labelled with a blue cross in the JUnit view. You can double click on any test, and you'll be taken directly to the point where the test failed in the Editor view.

When you think you have identified the part of the code that is causing the bug, correct it and save the file (using the floppy disk icon on the toolbar or with the standard Ctrl-S shortcut). You will notice that as soon as the changes are saved, the file appears in the Git Staging view as an unstaged change.

Figure **??** illustrates a file appearing as an unstaged change for a different issue than the one you are working on. (Obviously, we can't show screenshots of the change that fixes the bug you are working on, as that would give the answer to the exercise away.)

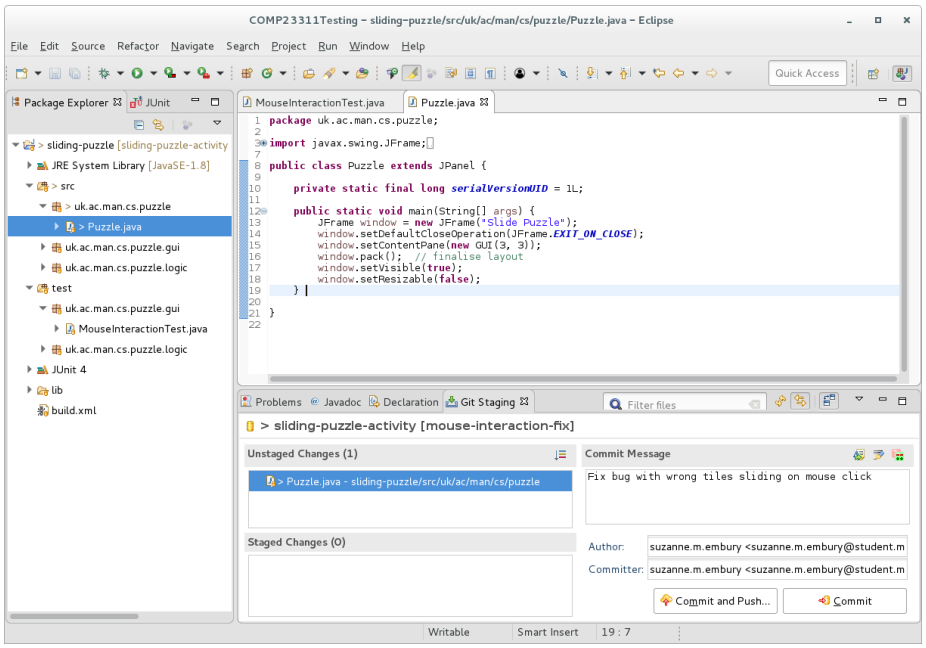

Figure 16.9: Some unstaged changes showing in staging view

Before committing the change, it is important to check that the tests now pass. Run the tests again to check that have fixed the bug, and haven't introduced any other problems. When you have correctly fixed the bug, all the tests should pass, in both test packages, and you should see a green bar in your JUnit window shown in figure **??**

If the tests do pass, **run the puzzle and check that it now behaves as intended**. If your tests keep failing, or the puzzle still isn't working as it should, and you can't find the problem, you'll need to come to one of the team study sessions to get help from staff or GTAs.

Once the code compiles, passes all the tests, and the puzzle runs as expected we can commit our changes to our *local* Git repository. To tell Eclipse that the change we've made should be included in this commit, drag the file from the

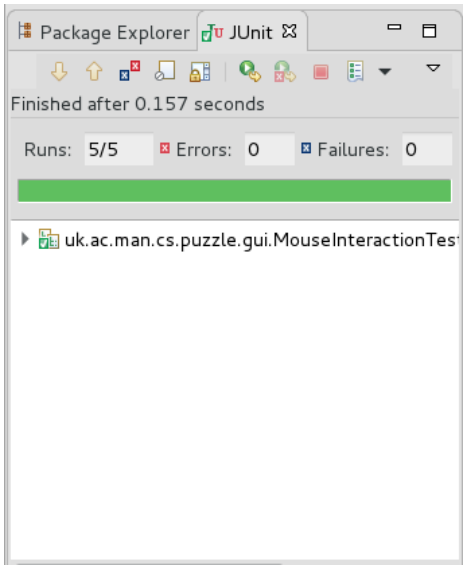

Figure 16.10: The JUnit view

'Unstaged Changes' box to the 'Staged Changes' box in the Git Staging view. Figure **??** illustrates how this should look for a different change than the one you are making.

Before making the commit, you need to check that Git is configured correctly on the machine you are working on, so that it assigns the correct author and committer information to all your commits. We make use of a lot of automated marking code in this course unit. Our code will not be able to find your commits if you have not configured Git correctly.

Git will use the values you set for the `user.name` and `user.email` parameters to set the author/committer details for your commits. When the commits are pushed to GitLab, the GitLab server tries to guess which project member made the commits, using this information. If it cannot, the commits will still exist but they will not be linked to your GitLab user. This will make it much harder for our automated marking tools to find your work. To make sure you get credit for all the work you do, please take care in configuring Git on *all* the machines you code on.

To check that Git is configured correctly on your machine, look at the Author and Committer information in the Git Staging view. If these show your correct name and (most importantly) your University e-mail (ending in the domain `student.manchester.ac.uk`, with no brackets or quotation marks) then everything is okay. If they do not, then **you will need to exit Eclipse** and reconfigure Git using the information on the Department wiki:

wiki.cs.manchester.ac.uk/index.php/Gitlab/Git_Setup

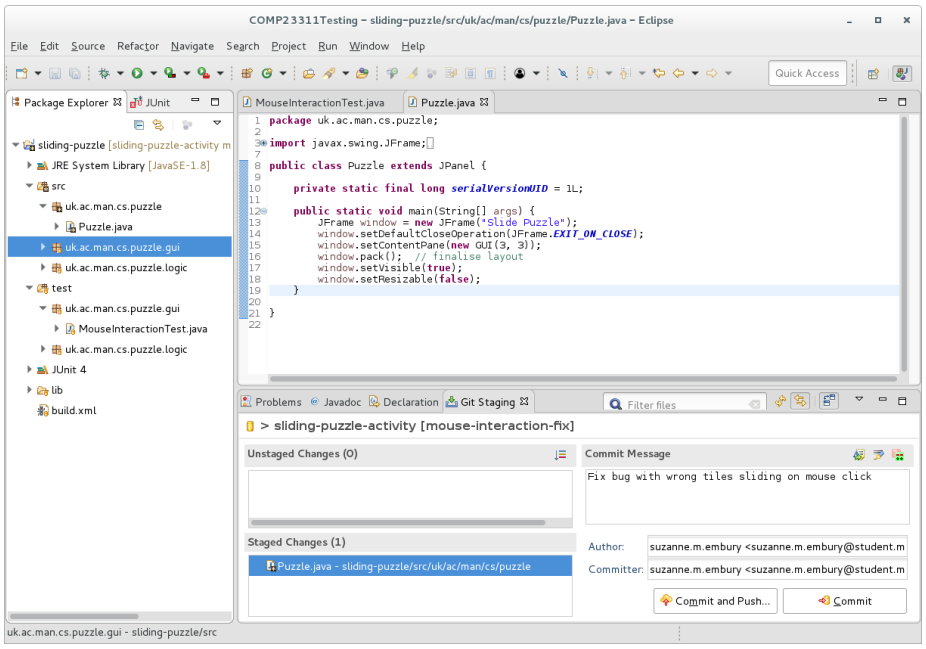

Figure 16.11: Stage the changes now that the tests pass

When you re-enter Eclipse, the correct author and committer details should be shown.

Caution If your name and e-mail are not shown correctly in the Author and Committer fields, you may be tempted to edit them directly in Eclipse, rather than fixing your Git configuration. That will fix the problem as far as this one commit is concerned, but you'll need to remember to make the same change for every commit you make for this course unit.

This is a particular problem for the team-based coursework, where the only way our marking systems know if you have made any commits is if we can find ones linked with your GitLab account. GitLab uses the author e-mail address of commits to link them to the account with the same e-mail address. Students with no commits linked to their GitLab account for each exercise will automatically receive a mark of 0. Therefore, it's really important that you take the time now to configure your local Git installation correctly now, to avoid losing marks in the future.

Now you can press the `Commit` button. **Important: do not push your commit at this time. Just make the commit.**

Why are we asking you not to push the commit at this stage? It is a good idea, especially when new to Git, to commit all your changes locally first, so you can check them out before you push them to the remote Git repository.

In general, it is easy to fix Git errors in your own local repository. But it is much harder to fix problems once they have been pushed to a public or team repository, and pulled into your team mates' local repositories. Getting into the habit of checking your commits before pushing them can save you a lot of time, frustration and embarrassment in the future, as well as sparing your team mates from losing marks due to your error.

So, before we do anything else, we're going to check that the commit went through as we expected. This is quick and easy to do using the `Git History` view. To bring this up, right click on the project name in the `Package Explorer`, and select `Team` > `Show in History`. You should see a new view appear next to the Staging view. It's a bit small, so figure **??** shows the effect of double-clicking on the view tab, to make it fill the screen.

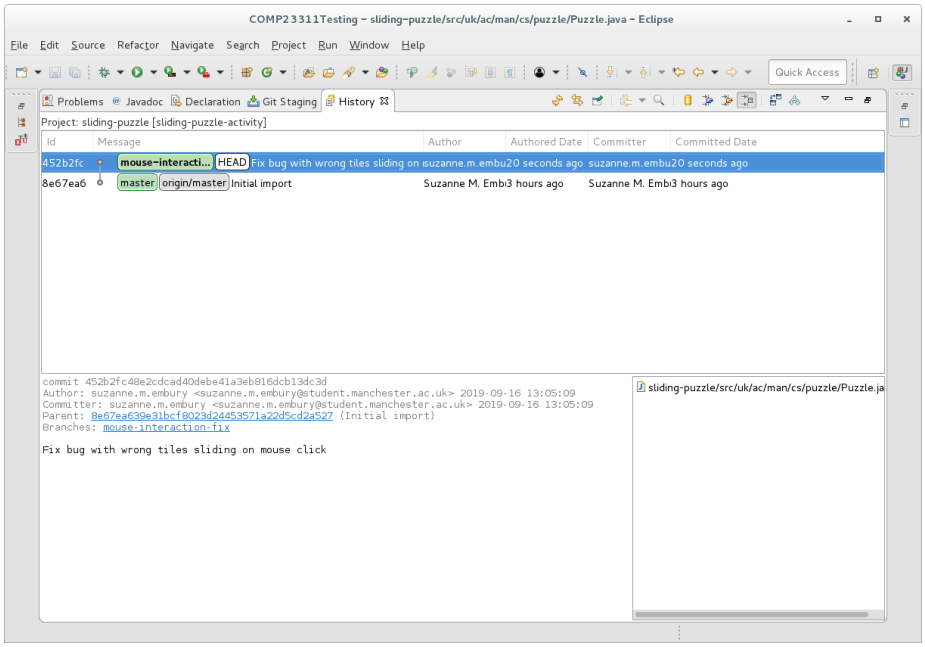

Figure 16.12: check commit in history view

This view shows the history of commits that are visible from the currently checked out commit. It shows that the branch we are on is one commit ahead of the local `master`, as well as the `master` branch in the remote repository (called `origin/master` because `origin` is the default name for the remote repository). The history view also tells us which branch is currently checked out, by bolding the name of the branch (and also putting the label `HEAD` next to it).

This looks okay, so we will go ahead and push the changes to the remote. Before doing this, take a look at the network for your remote repository on GitLab shown in figure **??**

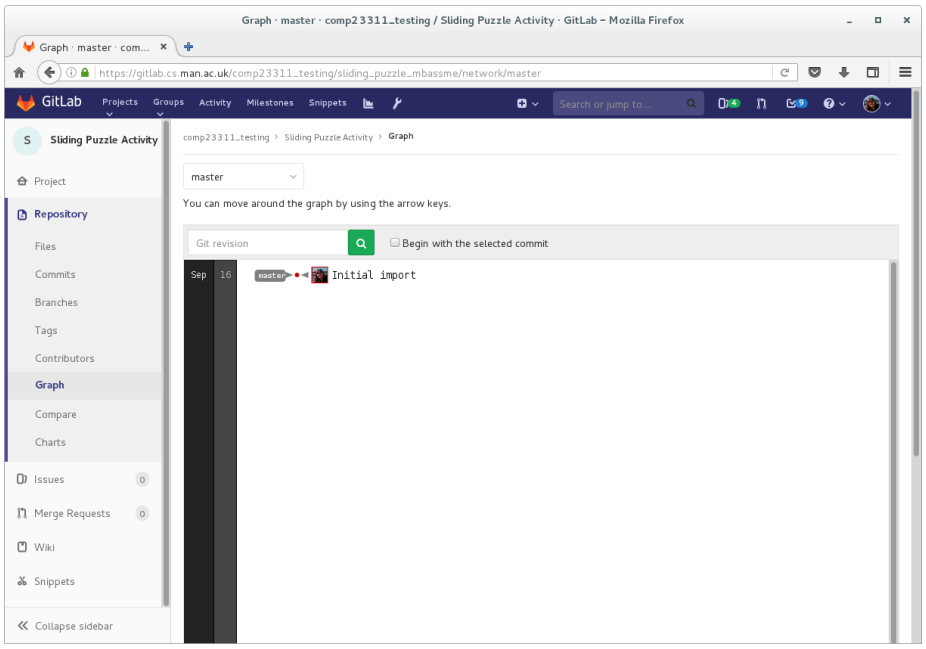

Figure 16.13: Screenshot showing gitlab before pushing

You should see only the `master` branch and the original lone commit. The changes you have made in your local repository are, as yet, not present in your remote (and therefore not visible to any collaborators you may have on the project).

Now push the changes, by right clicking on the project name, and selecting `Team` > `Push to origin` from the menu that appears. Whenever you perform remote operations, like push, Eclipse will need your GitLab credentials. If you didn't save them earlier, you'll be asked to provide them again at this point.

At this point, you may be asked to configure your repository for pushing. This sounds complicated but in fact is simple. Git is just asking you to tell it how it can map local branches to remote branches. Click on `Advanced`. You should see the dialogue appear in figure **??**

Press the "Add All Branches Spec button in the middle of this dialogue. This will add the default *ref specs* (reference specifications) to your repository. Select "Finish". Your repository should now be configured for push.

Before it asks Git to push, Eclipse will give you a summary of what the push will do and ask you to confirm that you want to go ahead, looking something like the dialogue shown in figure **??**

Press `OK` to initiate the push.

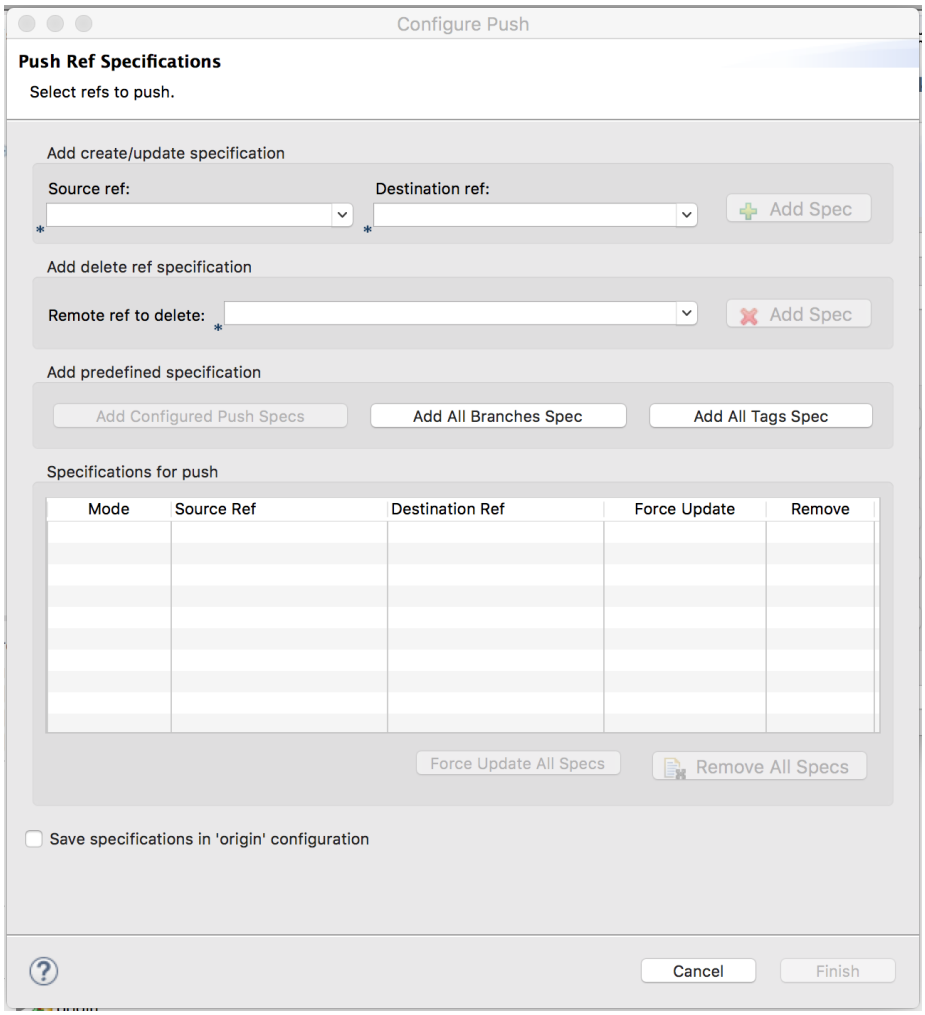

Figure 16.14: Configure push dialog

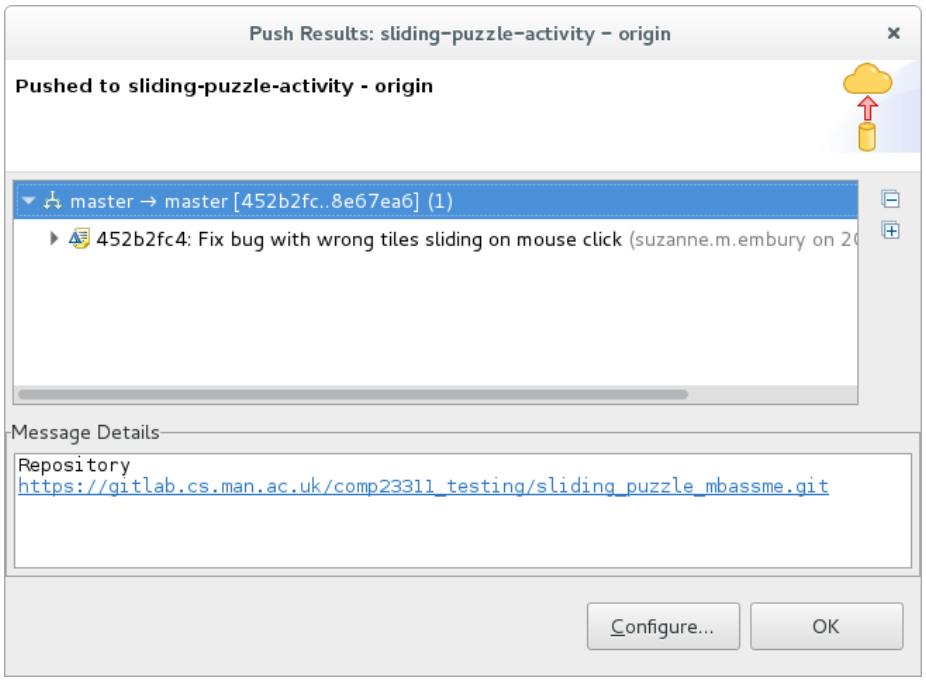

Figure 16.15: Confirmation of push to remote dialog

When the push is successfully completed, look at the contents of the History View. Can you see what has changed, as a result of the push?

You should also look at the commit graph in your remote repository. How has that changed since we looked at it before the push operation?

Note

As a rule of thumb, you should only commit working code. This means: code that compiles and passes the tests. It is especially embarrassing to push code with compile errors to your team's repository. Remember to check *before* you push!

16.3.6 Step six: Merge your Changes

In the Git workflow we use for this exercise, the development branch is called `master`.

Once a change to the code base has been tried out in a feature branch (and passes the tests), we can incorporate it into the main development branch, so that (when pushed) other team members can see it and build on top of it.

This is called *merging*.

In this case, we want to merge the changes in our feature branch *into* our `master` branch. The Git merge commands works by bringing the changes *into* the currently checked out branch. So we need to start by checking out `master`.

You can switch branches very easily from the History view. Just right click on the commit you want to check out (the one labelled with the `master` branch in our case), and select `Checkout`.

Eclipse will notice that there are two branches at the commit you want to check out. It will ask you which one you want to check out. In our case, that will be the local version of the `master` branch and not `origin/master`, the branch that is tracking the contents of the `master` branch on the GitLab remote. (Broadly speaking, you should *never* check out a remote-tracking branch.)

Select the local branch (`refs/heads/master`) and click `OK`.

When the checkout completes, the contents of the History view will change. It will look now as if the commit on the `mouse-interaction-fix` branch has been deleted, along with the branch it was on. Don't worry—the commit is still there. By default, the History View shows only commits reachable from the checked out branch, which is now `master`—a parent of the commit we just made. Note also that `master` is now shown in bold text, and the `HEAD` label has also moved to this commit: both signs that this is the checked out commit.

Now we can go ahead and make the merge. Merging is a tricky part of Git, and it is easy to make mistakes. We're going to do the merge in our local repository

first, without pushing any of the commits, so we can check out the result and fix things before anyone else in our team pulls our mistakes into their local repository. This is a really good habit to get into.

To request the merge, right click on the project name in the Package Explorer, and choose the `Team` > `Merge...` option. (If you expanded the History View, as we did, you'll need to double-click on its tab to get back to the normal Eclipse view layout before doing this.)

This will bring up a dialogue box giving you a choice of branches to merge with shown in figure **??**

Select the feature branch we have been working on. Leave the other options with their default settings and click on `Merge`.

Note When learning Git, it can be tricky to remember which branch merges into which, when using the merge command.

The key thing to remember is that, unless you've said otherwise, Git will make changes to the checked out branch. When merging, start by checking out the branch that does not yet have the new changes in it. The branch specified in the merge command is the branch that contains the changes we want to pull in to the checked out branch.

After the merge is complete, the checked out branch will change, but the branch given in the merge command will be unaffected by the merge.

Remember: try your merges out locally and check them before pushing them. It is easy to fix merge problems locally, by using the Git reset command. Merge errors that have been pushed to a remote repository are much harder to correct.

When the merge is finished, Eclipse will show a summary of the results shown in figure **??**.

In this case, the merge was successful. Git had only to perform a *fast forward merge8. That is, it just had to push the `master` branch forward by one commit to bring in all the changes made in the `mouse-interaction-fix` branch. It did not need to create a merge commit, and there are no merge conflicts to resolve.

The next step is to check that the tests still pass in the merged code. This is not so important after a fast-forward merge, if you have checked that the tests pass on the feature branch before merging. But merging is not always straightforward, and it is good to get into the habit of checking that tests pass after merging as well as before.

The History View (visible in the above screenshot) shows the new Git network after the merge. Now `master` and `mouse-interaction-fix` are at the same commit, with `master` still checked out. The remote tracking branch for the `master` branch in your remote repository, however, is still at the commit it was at when we first imported. This will change once we push the changes to the remote repository.

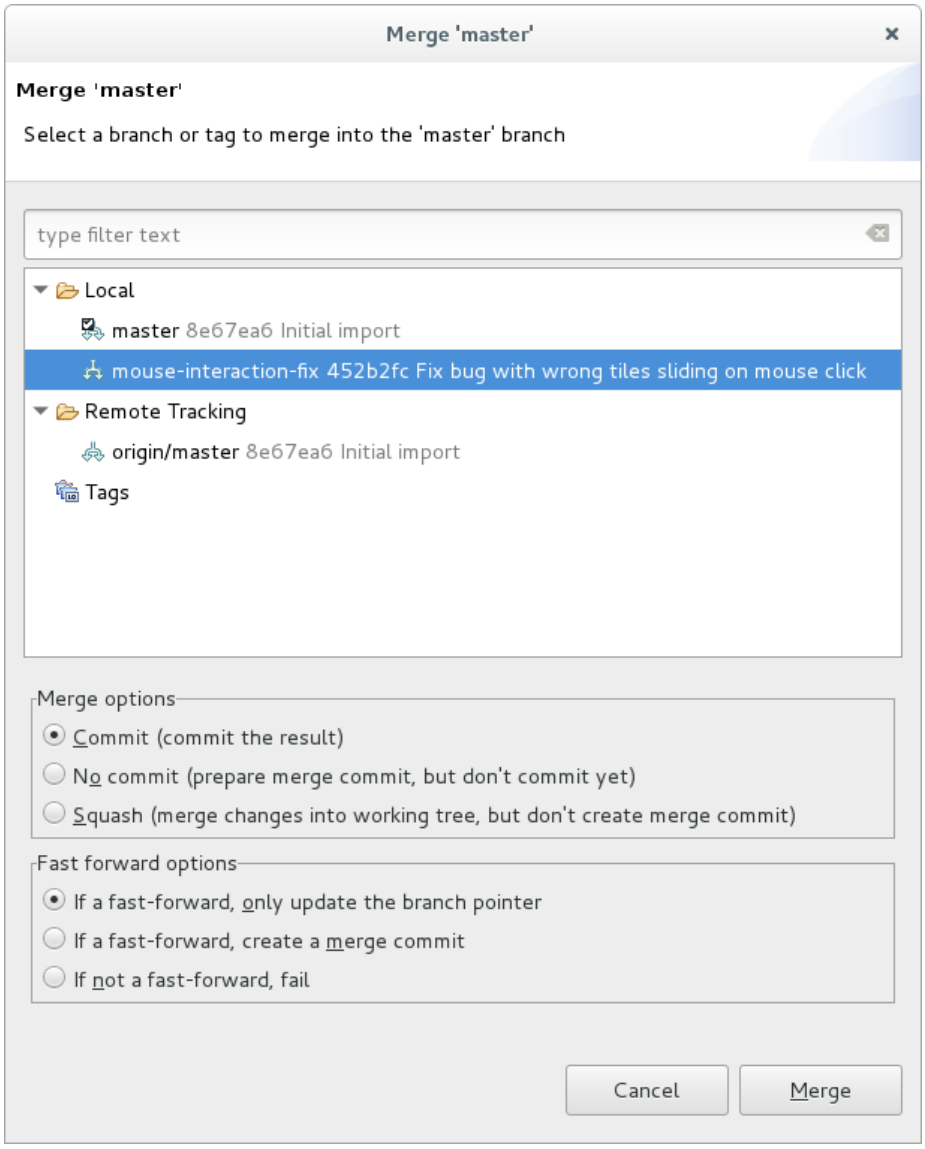

Figure 16.16: Request theM merge into master and select branch to merge in dialog

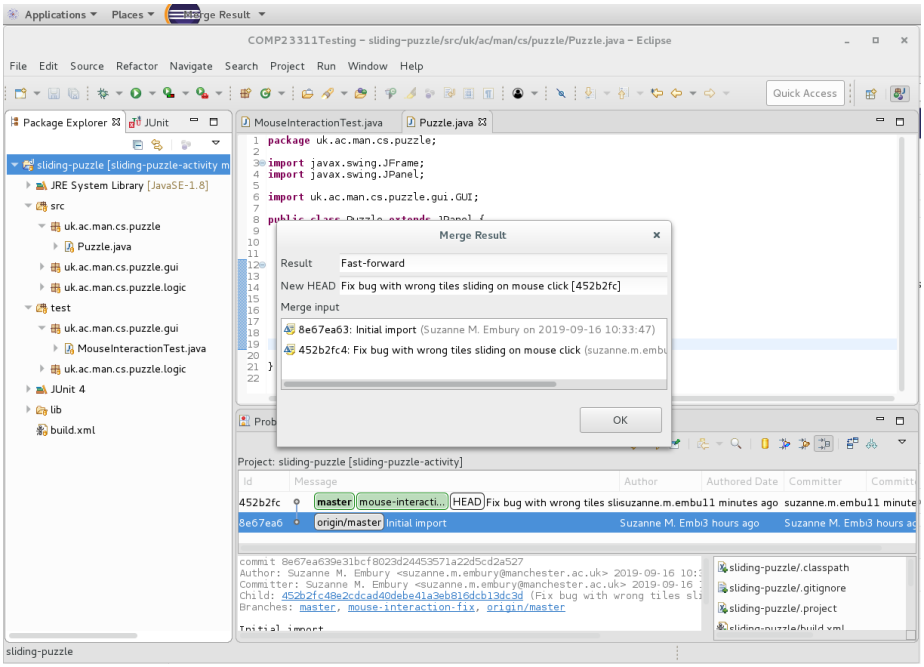

Figure 16.17: Screenshot showing merge results

16.3.7 Step seven: Push the Changes to the Remote Repository

If the merge looks okay in your History view, and the tests all still pass, we can push the changes to the remote repository.

As before, we do this by right clicking on the project name, and selecting `Team > Push to origin` from the menu that appears. Again, if you haven't saved your GitLab credentials in Eclipse, you'll need to supply them again for this operation to complete.

Press `OK` to initiate the push.

When the push is successfully completed, look at the contents of the History View. Can you see what has changed, as a result of this second push?

If you have merged and pushed correctly, you should see both branches pointing to the same commit on GitLab as shown in figure **??**

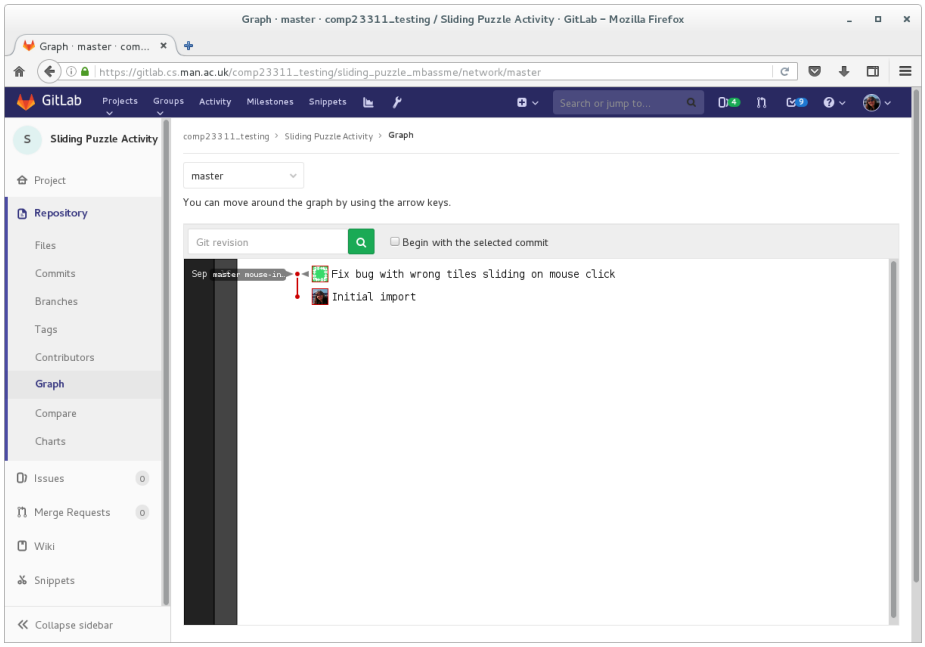

Figure 16.18: Screenshot showing check gitlab network after feature branch is pushed

16.3.8 Step 8: Record the Project Status in the Issue Tracker

Once the bug fix has been merged into the development branch, and the tests all still pass, we can close the issue. Open the `Wrong Tiles Sliding` issue in the issue tracker and click on the `Close issue` button. (Of course, if the feature branch hasn't been merged into the development branch and the tests still fail, then we can't close the issue, as the bug wouldn't be properly fixed yet.)

When you are ready for your work to be marked, add a comment to the issue, to let the automated marking system know. The comment should contain the following text:

```
Project ready for marking.
```

This comment will be detected by the automated marking code, and your work will be scheduled for marking at the next automated marking point. Because of this, it's important that you use this exact text string in your issue comment. You can delete and re-add the comment at any time before the coursework deadline, if you want to make further changes. It's also important that you add the comment to the correct issue. In the past, students have created new issues for this marking-request comment and even, in a few cases, have made dummy commits with this string as the commit message. None of these actions will be picked up by the automated marking system, so it's important to check that the comment is appearing on the issue we created for you and that all your changes are visible at your remote *before* the deadline.

Caution It is your responsibility to check that your work has been successfully pushed to the GitLab project repository *before* the deadline for coursework submission. We will not mark your local repository contents, only the contents of your GitLab repository. Therefore, make sure you check that the full set of commits and branches you expect to see are visible in your GitLab repository, and report any problems to the course team *before* the deadline.

If you submit your work late for whatever reason, add this comment to your issue to let us know that you are ready for the work to be marked. Late marking will be rerun every couple of weeks, so you won't receive feedback immediately.

16.3.9 Coursework Complete

This completes the instructions for the first individual coursework exercise for COMP23311. If you managed to complete it in full, then you can be confident that your Git/GitLab set up is as needed on this machine for the workshops and the coursework. Perhaps even more importantly, you have gone through the basic cycle of steps we'll expect you to follow when carrying out the team coursework for this unit. If you were unfamiliar with this approach to coding,

then it might have felt long-winded and complicated. But with a little practice, this pattern of work will become easier and more natural. And, with this experience under your belt, you'll find it much easier to adapt to variants of it, if you join a software engineering team on placement or in employment outside University.

In the next individual coursework, we'll look at a more complicated merge case, when Git can't handle the merge by itself and needs help from you to intervene.