Software engineering

Suzanne Embury and team

Contents

Welcome **9**

0.1 Making better software 9

0.2 Aims 11

0.3 Recommended reading 12

0.4 Using the lab manual 13

0.5 Contributing to this manual 13

0.6 Acknowledgements 13

0.7 Licensing 14

Weekly timetable **17**

0.8 Tools 17

I Weekly Workshops **19**

1 Marauroa **21**

1.1 Introduction 21

1.2 Acquiring Marauroa 22

1.3 Building the Marauroa Engine 38

1.4 Testing the Marauroa Engine 46

2 Understanding large systems **65**

Purposes of the workshop 65

2.1 Learning Large Codebases 66

2.2 Unit Testing Overview 69
2.3 Unit Testing Reading and Writing in Marauroa and Stendhal . . 76
2.4 JUnit Cheatsheet 77

3 Debugging a Codebase **81**
3.1 Preparing for the workshop 81
3.2 Workshop Activity: Working Through a Bug Report 83

4 Cost estimation **89**
4.1 Course content 89

5 Test first development **91**
5.1 Course content 91

6 Git workflows **93**
6.1 Course content 93

7 Software Refactoring **95**
7.1 Preparing for the workshop 95
7.2 Activity: Literals and Magic Numbers 96
7.3 Activity: Long Methods 97
7.4 Activity: Excessive Comments 99
7.5 Activity: Applying the Refactorings Together 100

8 Design for Testability **103**
8.1 Preparing for the Workshop 103
8.2 Understanding Test Doubles: Dummies 103
8.3 Understanding Test Doubles: Stubs 105
8.4 Test Doubles Scavenger Hunt 107
8.5 Understanding Test Doubles: First Experiments with Mock Objects 108
8.6 All Finished and Nowhere to Go? 109

9 Software design patterns **111**

9.1 Introduction 111

9.2 Workshop Exercise 1 - Behavioural Patterns 112

9.3 Workshop Exercise 2 - Structural Patterns 117

9.4 Workshop Exercise 3 - Creational Patterns 121

10 Risk management **123**

10.1 Course content 123

11 Open source challenge **125**

11.1 Introduction 125

11.2 The challenge 125

11.3 Identify an appropriate project 125

II Team study materials **129**

12 Stendhalgame.org **131**

12.1 Introduction 131

12.2 Manually Replicating The Issues 132

12.3 Getting Inspiration for Writing your Own Test Cases 132

13 Industrial mentoring **135**

13.1 Your mentors 135

13.2 Getting the most of your mentor 135

14 Synchronising **137**

14.1 Introduction 137

14.2 Making a Local Commit 138

14.3 Step 2: Pushing the Commit to the Team Repository 140

14.4 Step 3: Fetching the New Commit from the Team Repository . . 143

14.5 Step 4: Incorporating the Commit into Your Local Branch 146

14.6 A Final Word 148

III Coursework **149**

15 Individual Coursework 1 **151**

16 Individual Coursework 2 **153**

17 Team Coursework 1 **155**

18 Team Coursework 2 **157**

IV Self study materials **159**

19 Testing Stendhal **161**

19.1 Introduction 161

19.2 Manually Replicating The Issues 162

19.3 Getting Inspiration for Writing your Own Test Cases 162

20 Integrating commits **165**

20.1 Introduction 165

20.2 Git has Rejected my Push 165

20.3 Integrating the New Commits using Merge 168

20.4 Integrating the New Commits using Rebase 172

20.5 A Final Word 180

21 Continuous integration **181**

21.1 Introduction to Continuous Integration 181

21.2 Logging onto Jenkins CI Server: 183

21.3 Accessing your Coursework Builds 185

21.4 Drilling Down on the Development Branch Build 188

21.5 What Happens When Jenkins Builds a Job 190

21.6 A Look at Feature Branch Builds 192

21.7 Confused? Stuck? Need Help? 194

22 Code review **197**

22.1 Why Review Code? 197

22.2 Types of Code Review 199

22.3 Good Practice for Code Reviewers 201

22.4 Code Review Facilities in GitLab 203

22.5 Code Review in COMP23311 205

23 Unit testing **209**

23.1 Introduction 209

23.2 What is Automated Testing and Why Do We Need It? 210

23.3 Automated Testing in JUnit: a Simple Example 212

24 References **215**

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 1. 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

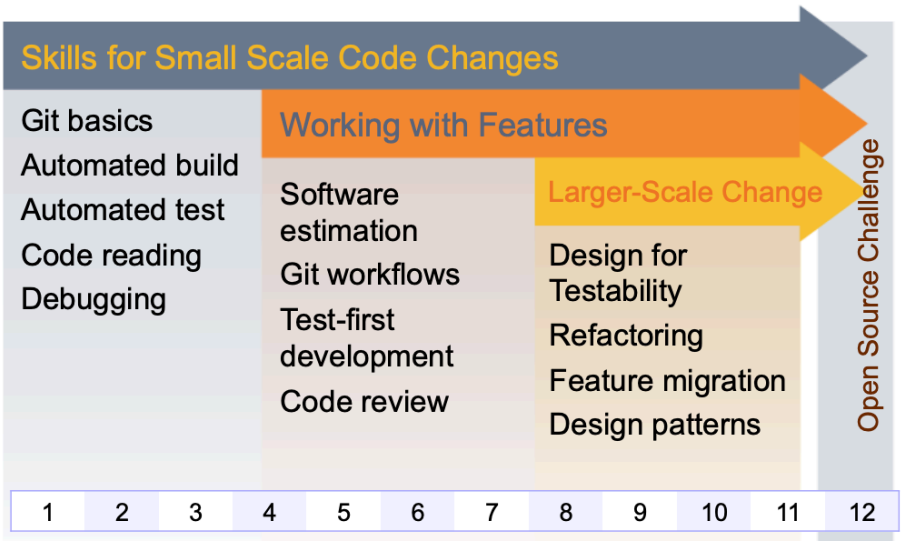

Figure 1: Course unit roadmap. This twelve week course will take your from small scale code changes (shown in grey), through to working with features (shown in orange) and on to larger-scale change (shown in yellow). We finish with an open source challenge in chapter 11. These skills are fundamental 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 listed in chapter 24.

1. Pro Git (Chacon and Straub, 2014)
2. The pragmatic programmer : from journeyman to master (Hunt and Thomas, 2004)
3. Effective unit testing : a guide for Java developers (Koskela, 2013)
4. Clean code : a handbook of agile software craftsmanship (Martin and Feathers, 2009)
5. The clean coder : a code of conduct for professional programmers (Martin, 2011)
6. Beginning software engineering (Stephens, 2015)

These and any other references cited are listed in chapter 24.

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. You can search, browse and link to anything in the manual. If you'd prefer, the manual is also available as a single pdf file and single epub as well.

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 softeng.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, Mercedes Argüello Casteleiro, Gerard Capes, Martina Catizone, Sarah Clinch, Peter Crowther, Anas Elhag, Sukru Eraslan, Gareth Henshall, Duncan Hull, Nikolaos Konstantinou, Kamilla Kopec-Harding, Kaspar Matas, Chris Page, Dario Panada, Liam Pringle, Julio Cesar Cortes Rios, Sara Padilla Romero, Viktor Schlegel, Stefan Strat, Jake Saunders, Federico Tavella, Mokanarangan Thayaparan, David Toluhi and Markel Vigo.

We'd also like to thank students who have done the course since its first iteration.

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

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.

Weekly timetable

The weekly schedule for autumn 2021 is shown in table 1, see also:

- studentnet.cs.manchester.ac.uk/ugt/timetable full timetable
- manchester.ac.uk/discover/key-dates key dates

0.8 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

Table 1: The weekly schedule for this twelve week course

Week no. starting	Workshop	Activities
0: 20th Sept	Welcome week	details tbc
1: 27th Sept	Automated build and test	details tbc
2: 4th Oct	Reading large codebases	details tbc
3: 11th Oct	Cost estimation, see chapter 4	details tbc
4: 18th Oct	Test first development	details tbc
5: 25th Oct	Git workflows	details tbc
6: 1st Nov	Software refactoring	details tbc
7: 8th Nov	Design for testability	details tbc
8: 15th Nov	Design patterns	details tbc
9: 22nd Nov	Risk management and practice exam	details tbc
10: 29th Nov	Open source challenge	details tbc
11: 6th Dec	confirm when reading week is	details tbc
12: 13th Dec	tbc	details tbc

Part I

Weekly Workshops

Chapter 1

Marauroa

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.

1.1 Introduction

In this workshop, we will be building and testing 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.

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 1.1.

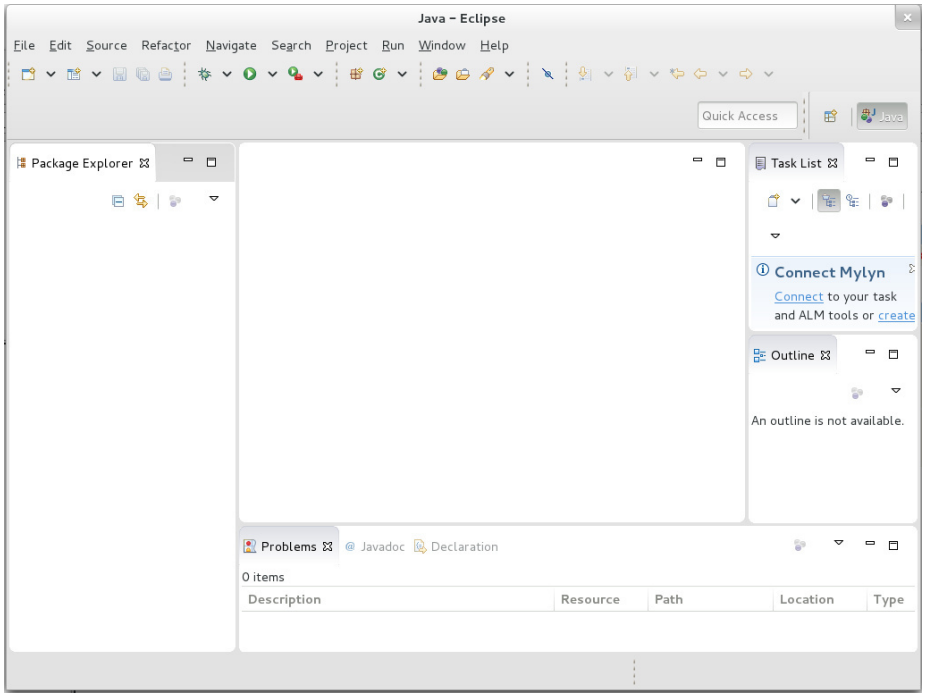

Figure 1.1: Your main eclipse window should look something like this

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 1.2

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 1.3

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 1.4 before proceeding.

If everything looks okay then select `Next`.

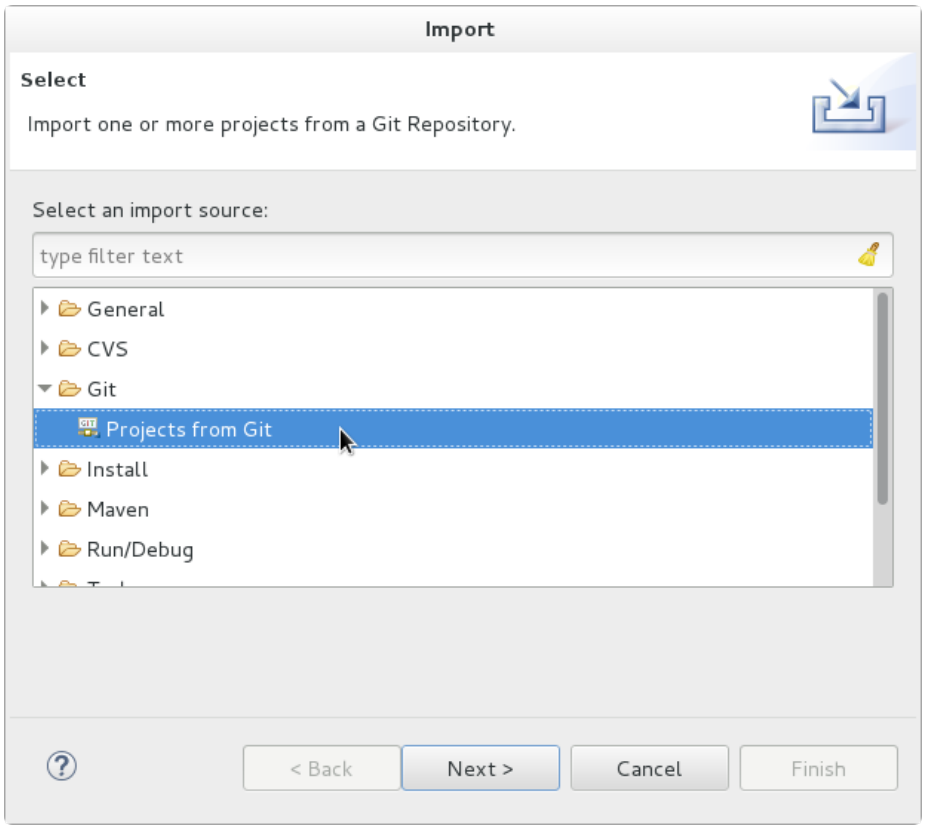

Figure 1.2: Your main eclipse window should look something like this

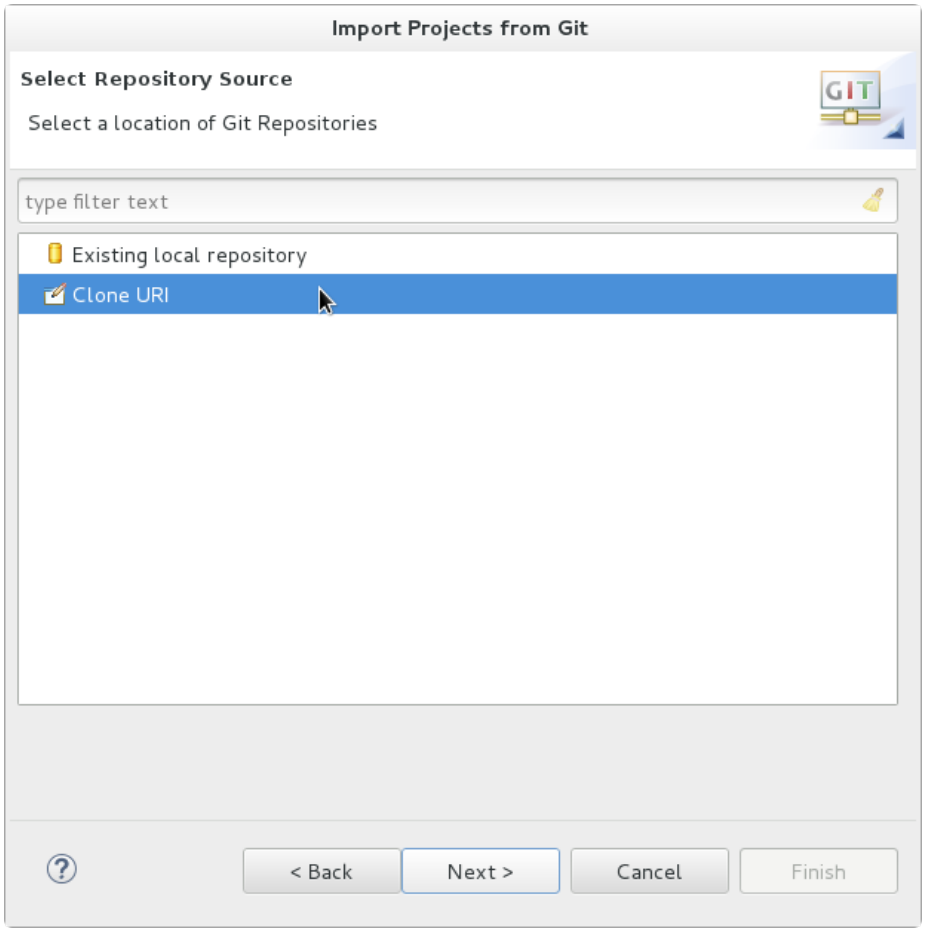

Figure 1.3: Your main eclipse window should look something like this

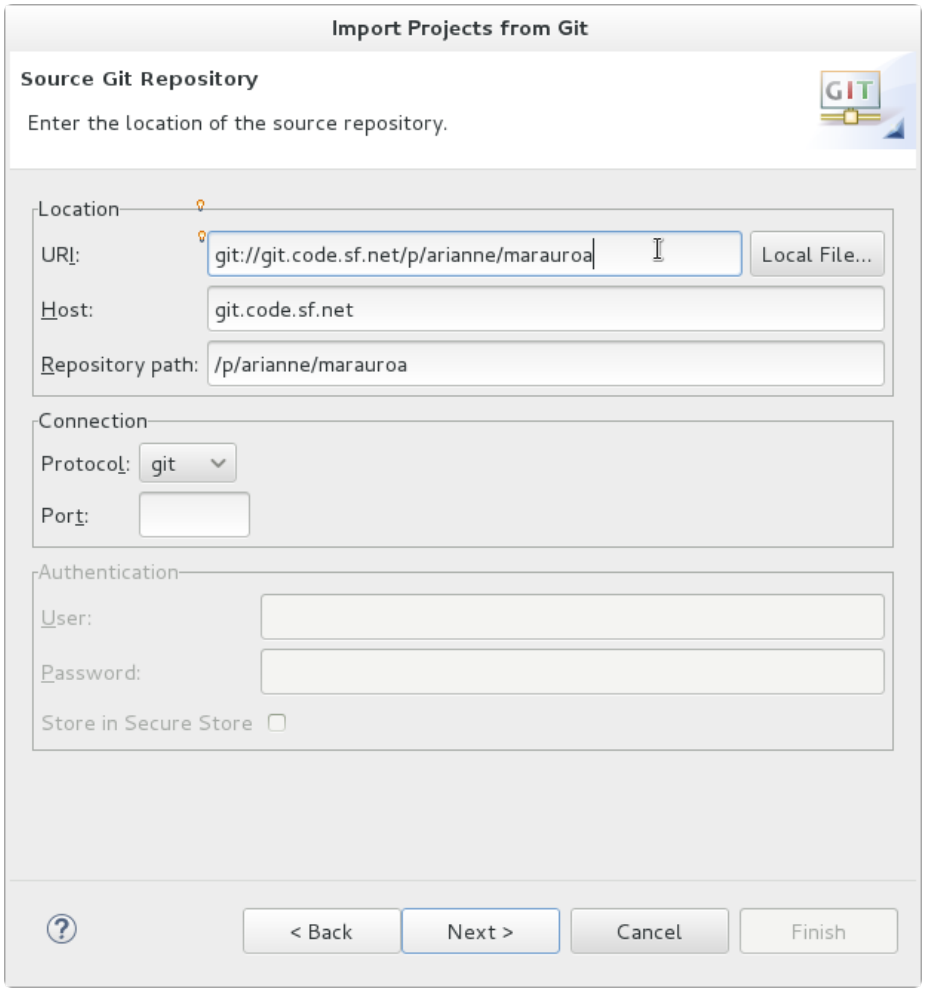

Figure 1.4: Your Import Projects from Git dialogue box should look like this

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 1.5.

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 1.6

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).

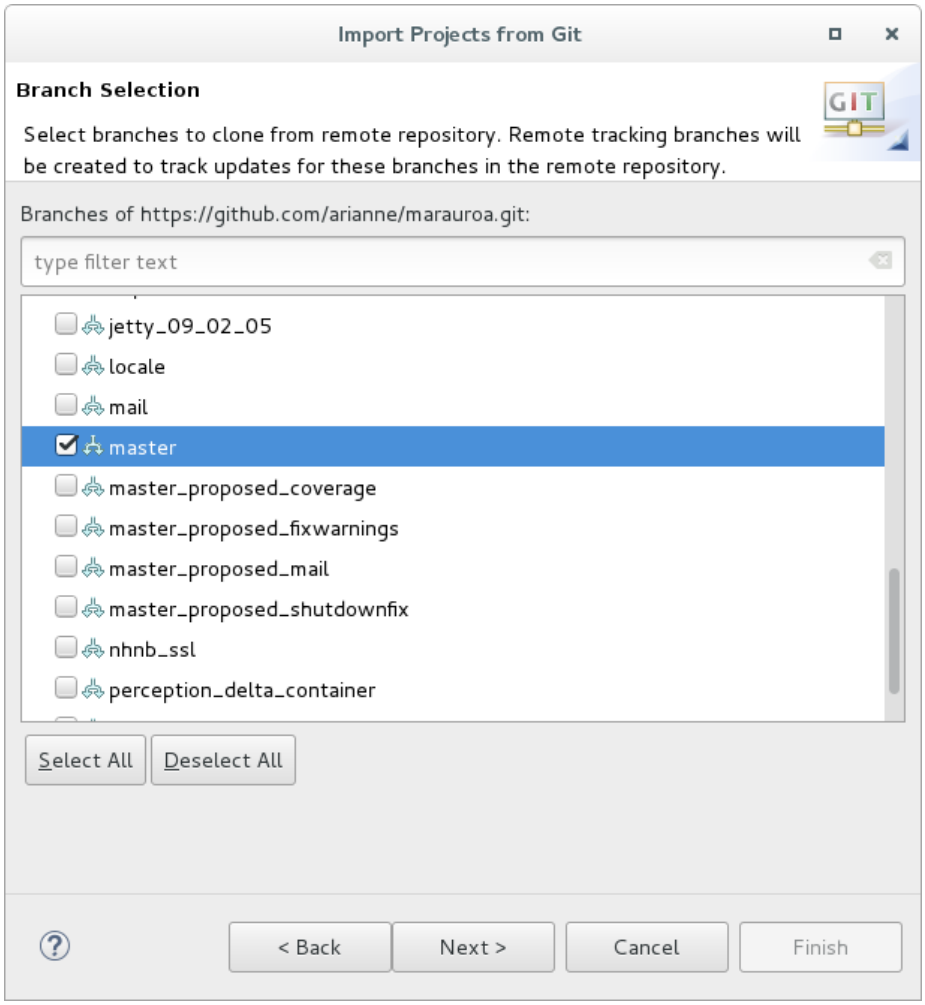

Figure 1.5: Take a look at the list of branches contained in the project, by scrolling up and down the list. You'll see that the Marauroa project uses separate branches to describe specific releases, as well as other development branches. Another common approach is to have a single release branch and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

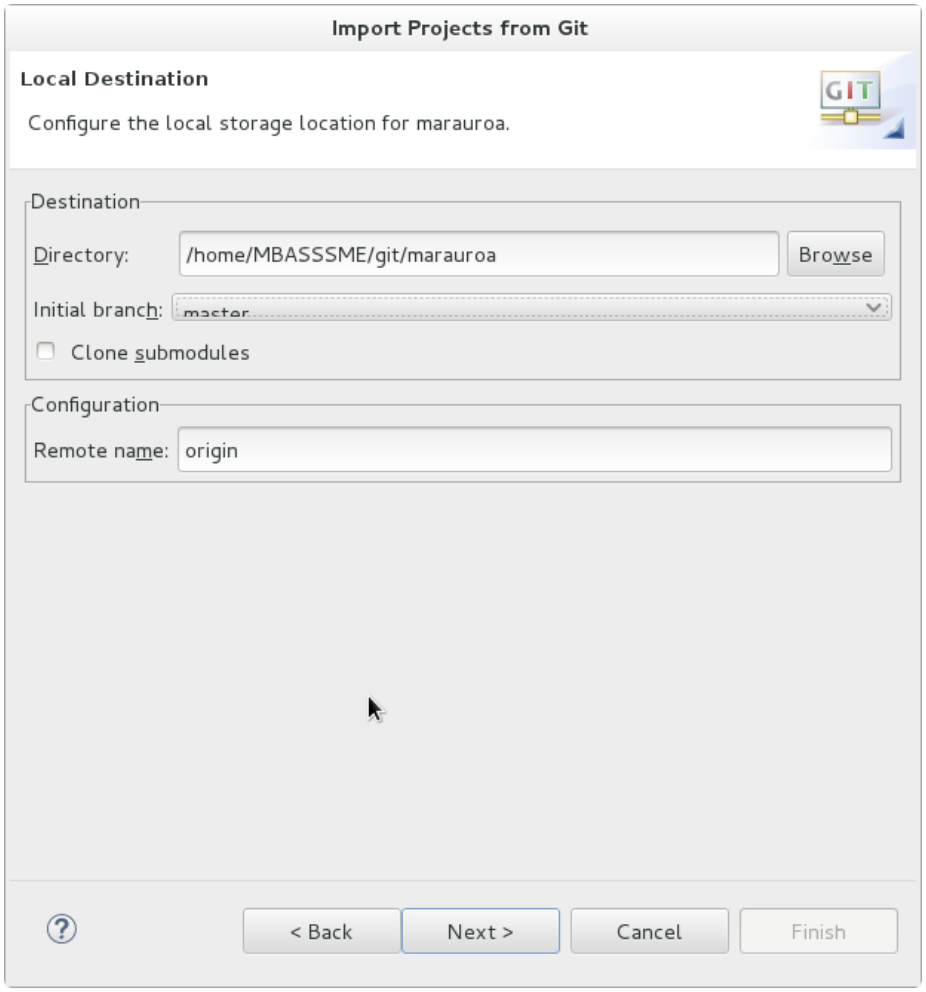

Figure 1.6: Cloning from git dialog box

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 1.7 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 1.8

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 1.9

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

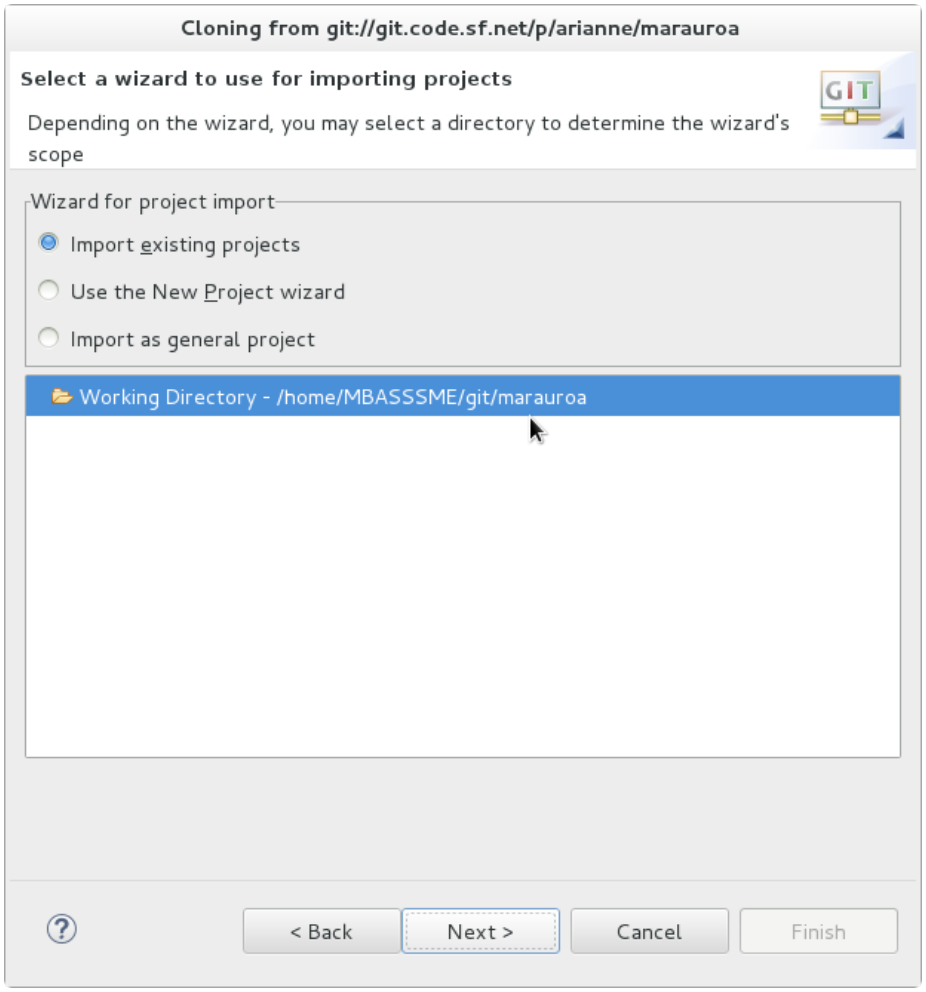

Figure 1.7: Cloning from git dialog box

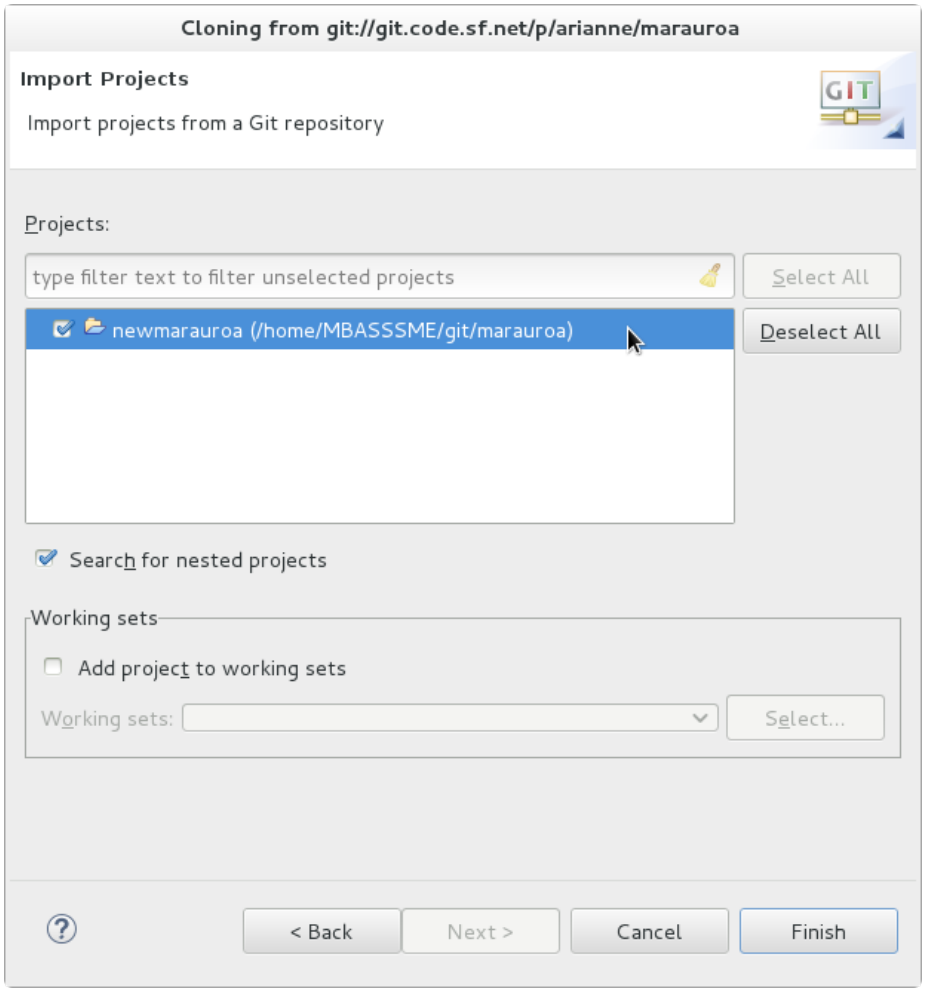

Figure 1.8: Import projects from a git repository and the newmarauroa project

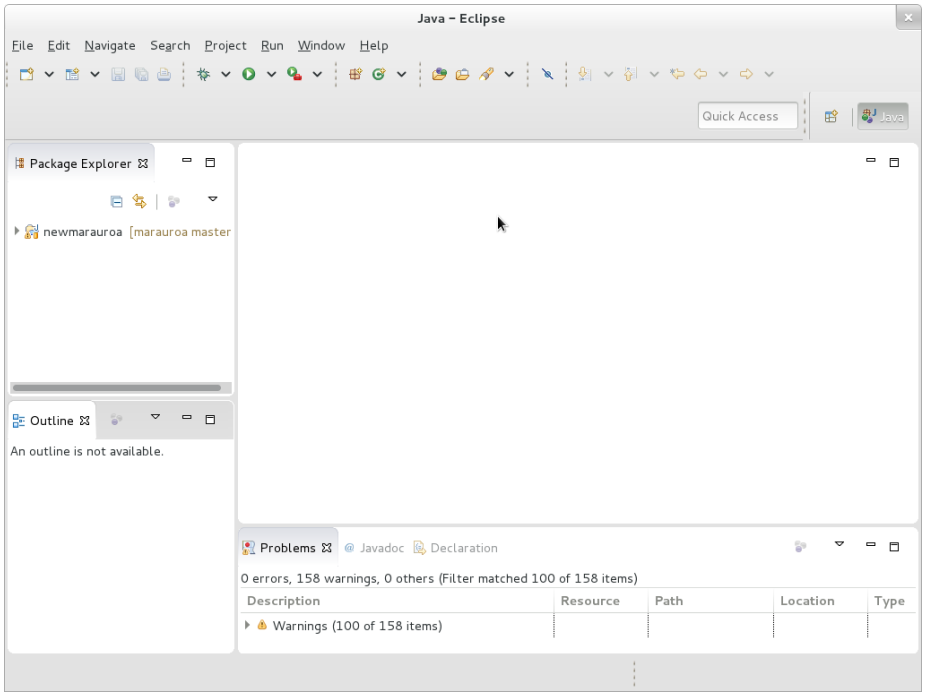

Figure 1.9: Import projects from a git repository and the newmarauroa project

in the Package Explorer view. Select `Team` > `Show in History` from the menu that appears. The History View shown in figure 1.10, 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.

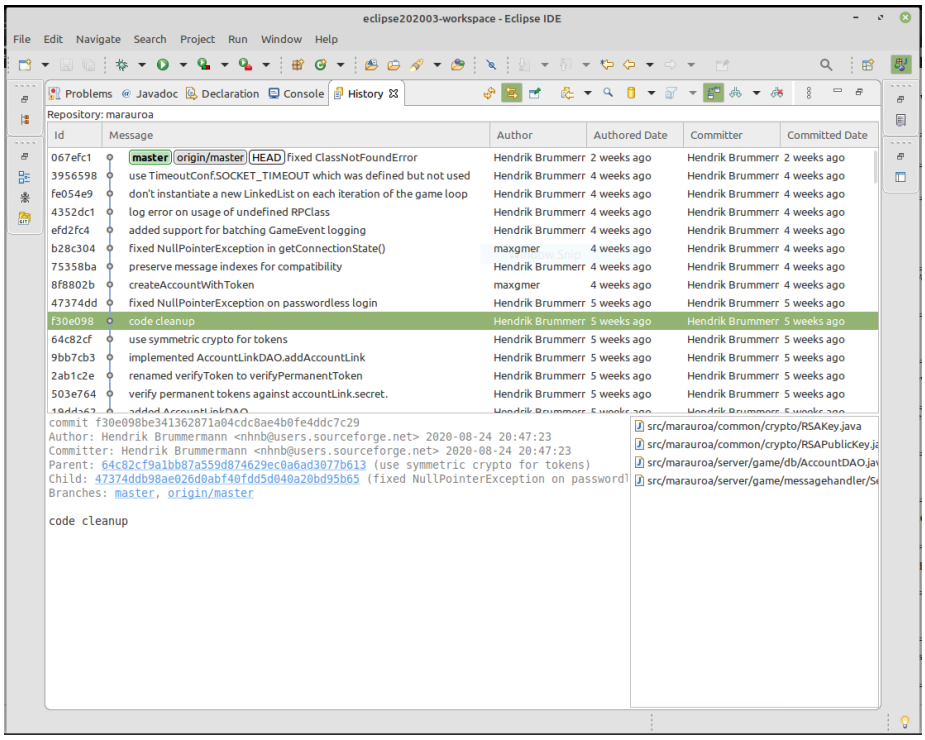

Figure 1.10: In this view, you should now see the most recent part of the network of the Marauroa project repository. You can scroll down to see the full commit log. As you can see, the history is significantly more complex than the simple repository we looked at in the GitLab Access Check. Marauroa 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 1.11

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.

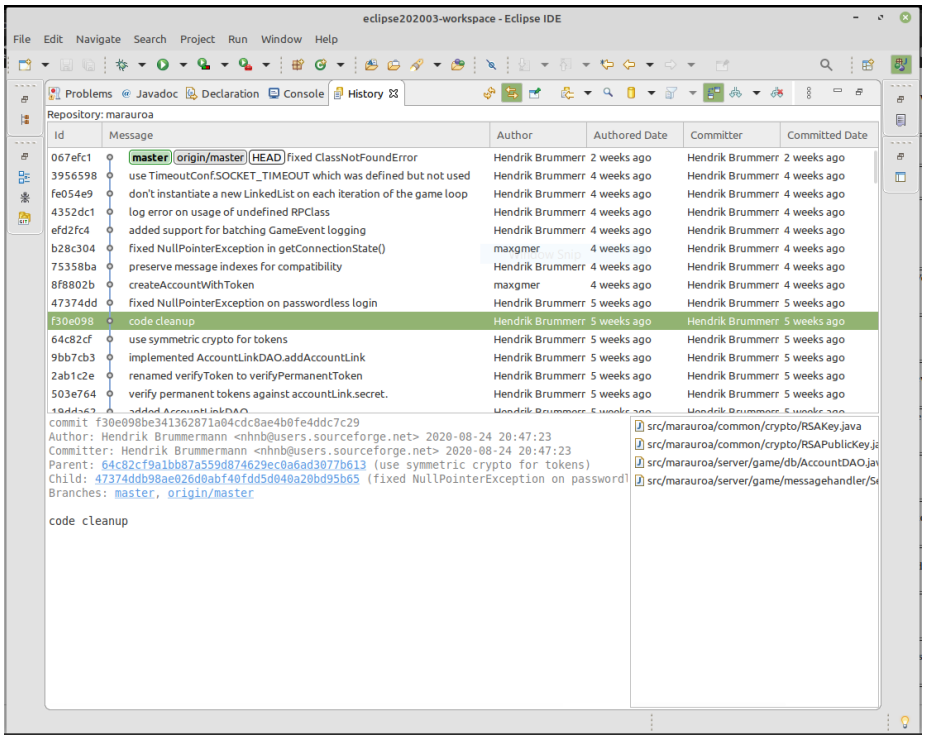

Figure 1.11: A warning of the detached HEAD state which reads: "You are in the `detached HEAD` state. This means you don't have a local branch checked out. You can look around but it's not recommended to commit changes. The reason is that these commits would not be on any branch and would not be visible 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 1.12

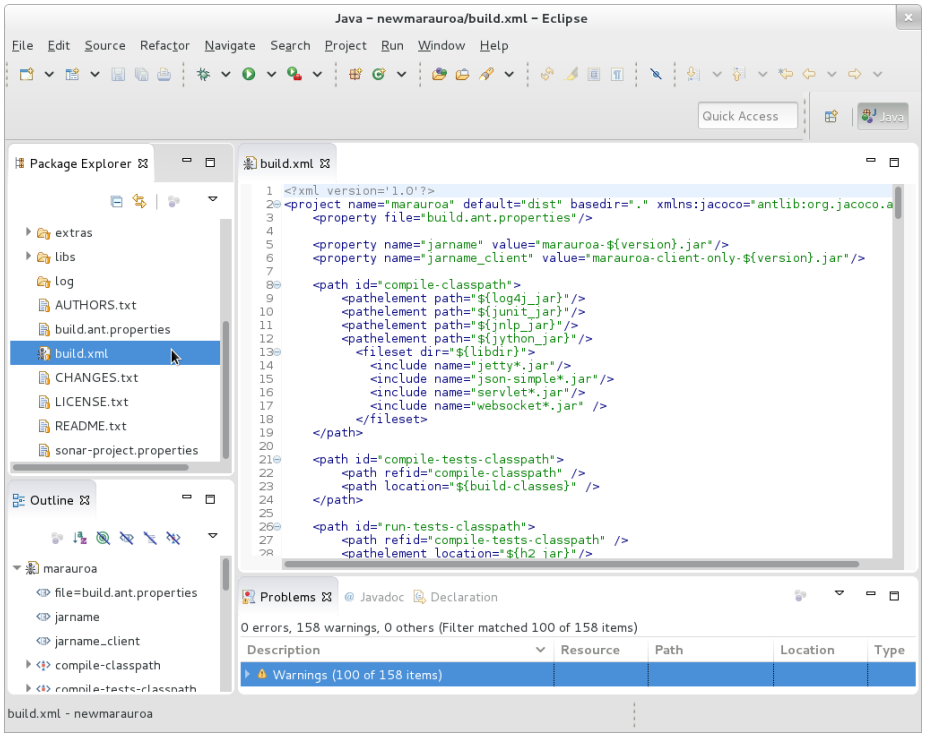

Figure 1.12: An XML build file

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 1.13

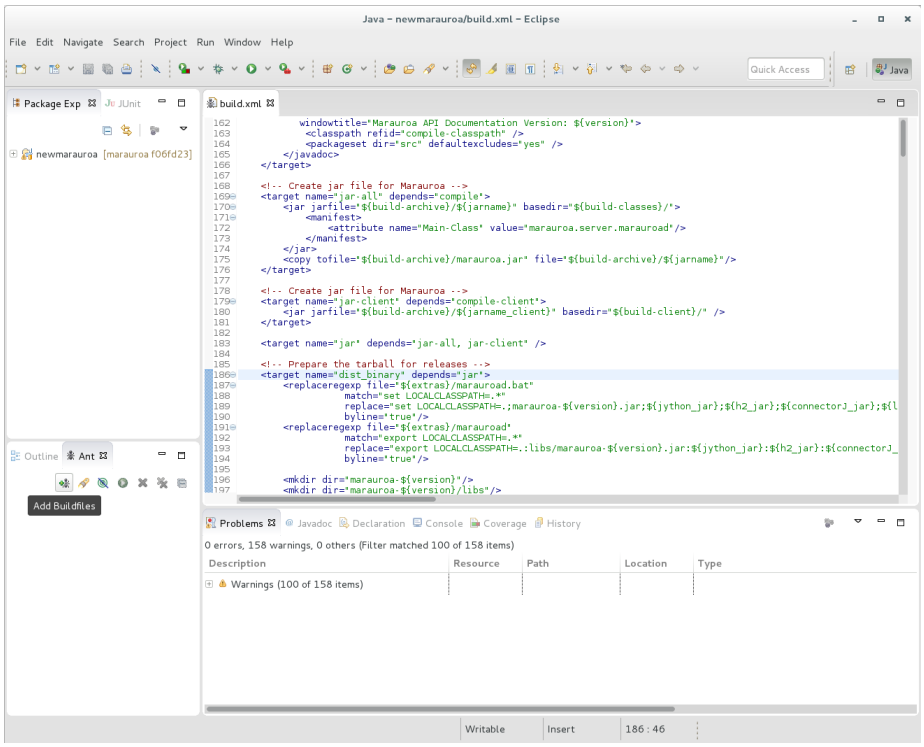

Figure 1.13: An XML build file

This will open a new dialogue that allows you to select a Buildfile. Select `build.xml` and click `OK` shown in figure 1.14

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

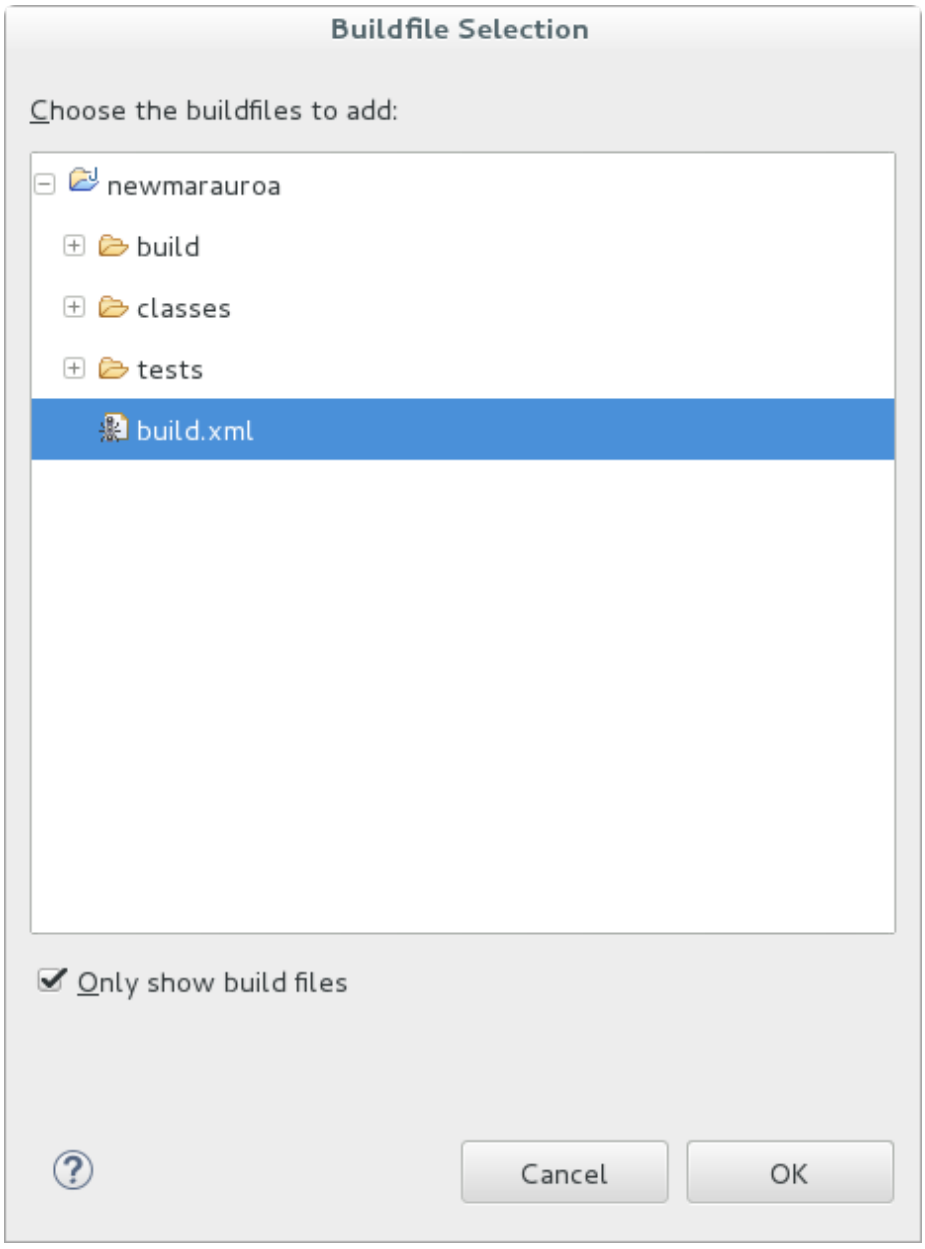

Figure 1.14: Buildfile Selection

window are changed, so that the definition of the target we have clicked on is displayed. For example, in figure 1.15, we've clicked look at the `jar-all` target and take a look.

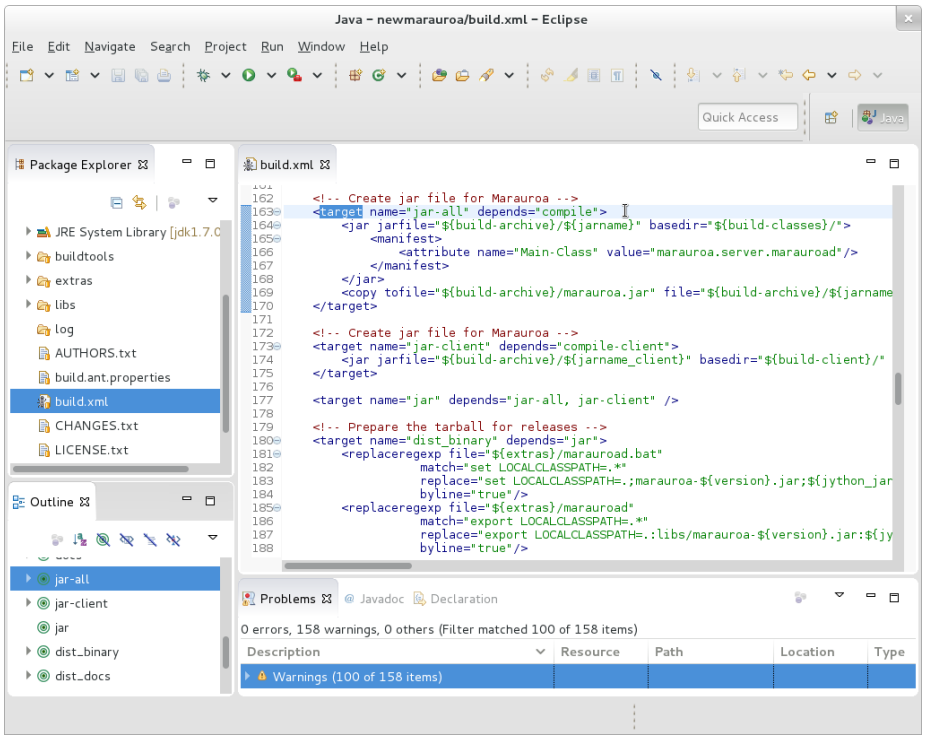

Figure 1.15: The jar-all target

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 1.16) 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.

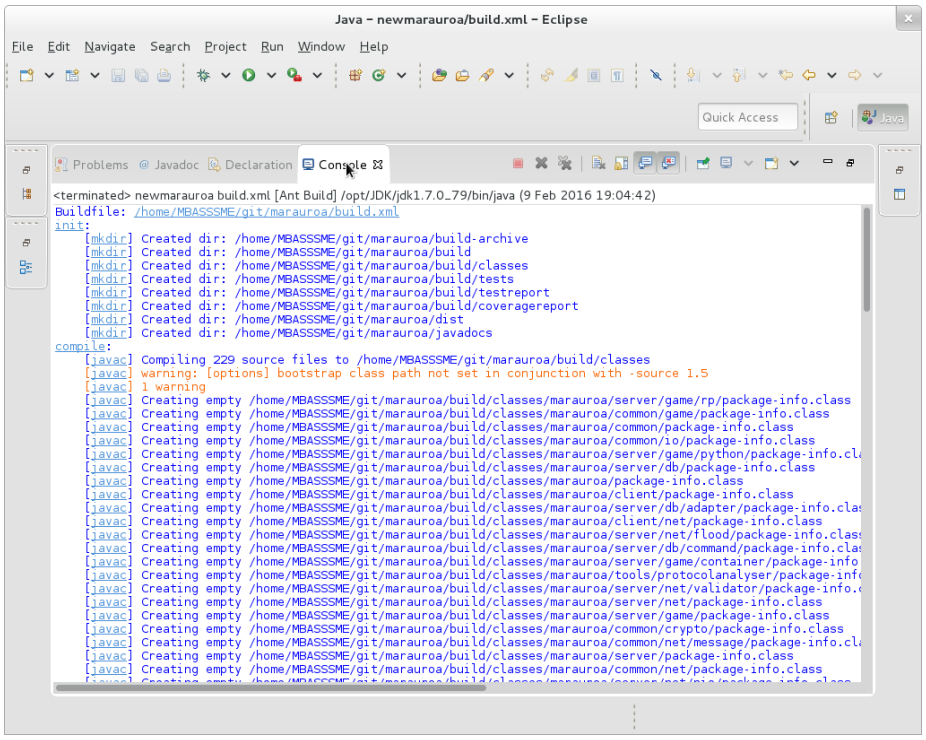

Figure 1.16: Console output

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,

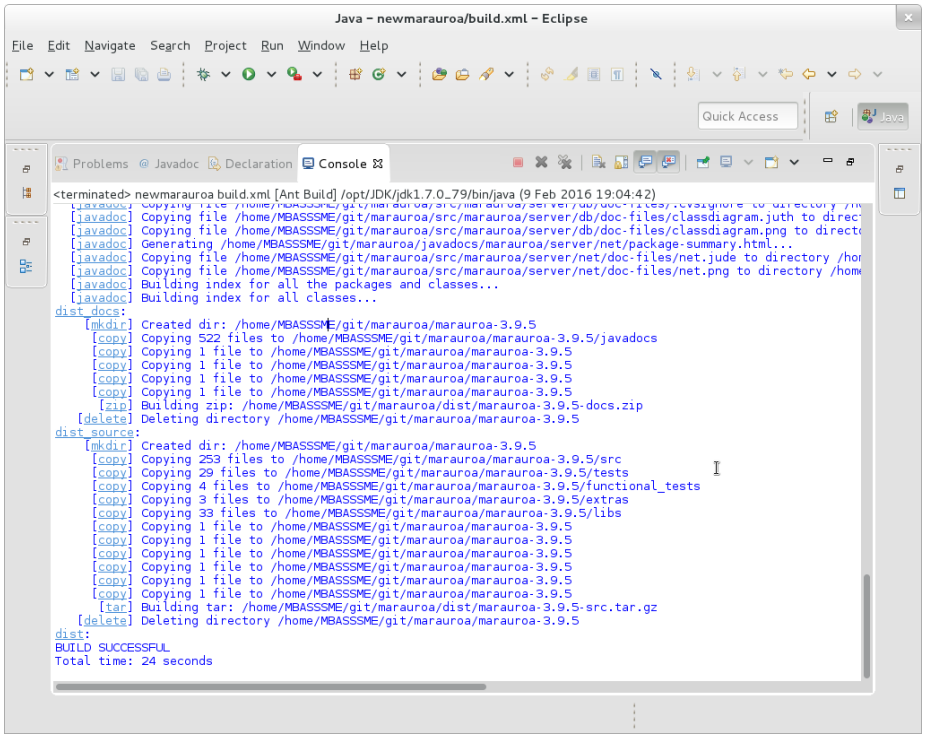

Figure 1.17: Console output

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 1.18

Let's take a quick look at that target before we look at the rest of the `test` target shown in figure 1.19

The target in figure 1.19 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).

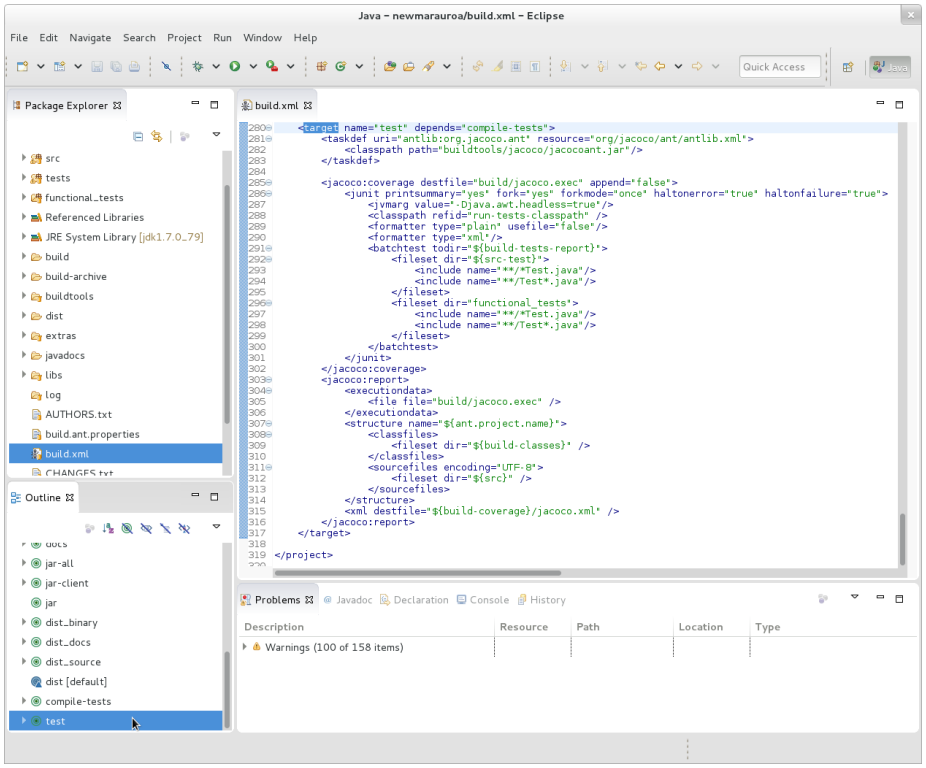

Figure 1.18: We can see that this target is dependent on another target, called `compile-tests`. That makes sense as we would expect to have to compile the test (and production) code before we can run the tests.

```
<!-- 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 1.20

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 1.21.

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

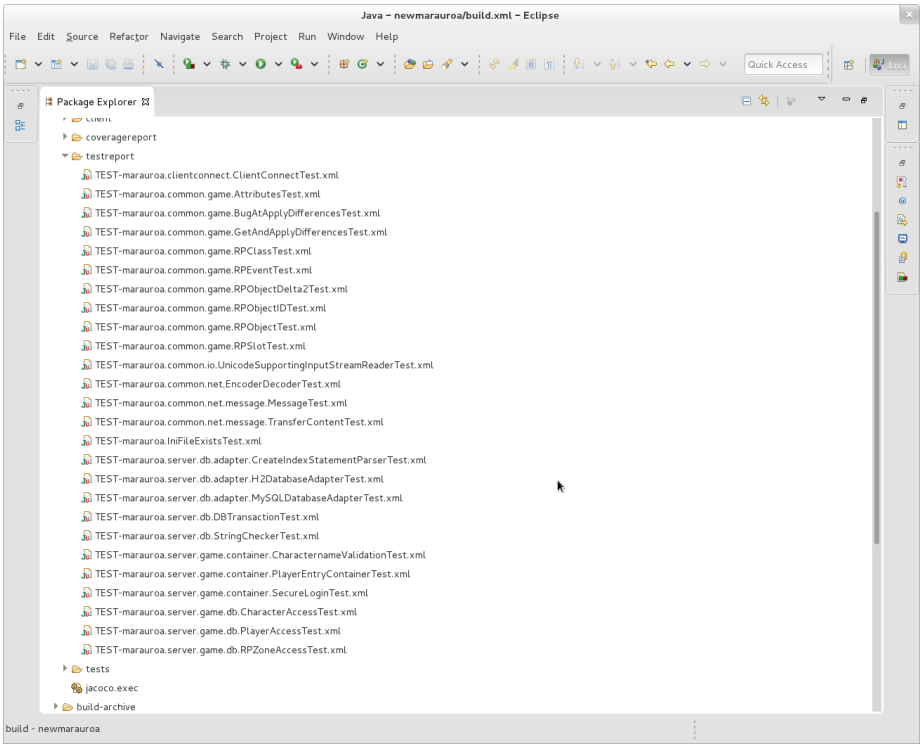

Figure 1.20: The testreport

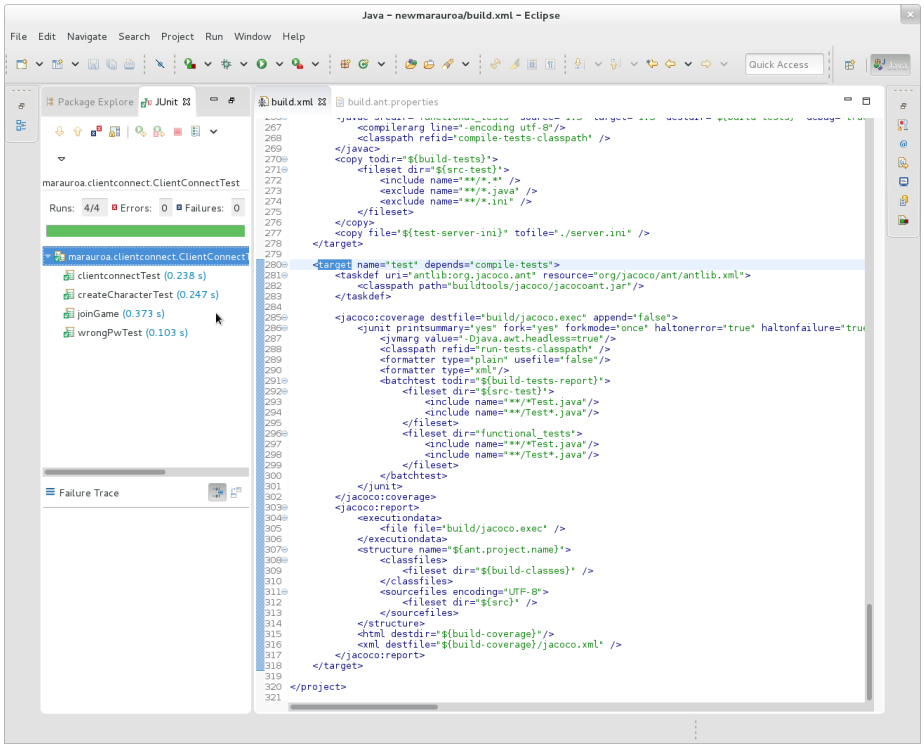

Figure 1.21: The JUnit view

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 1.22

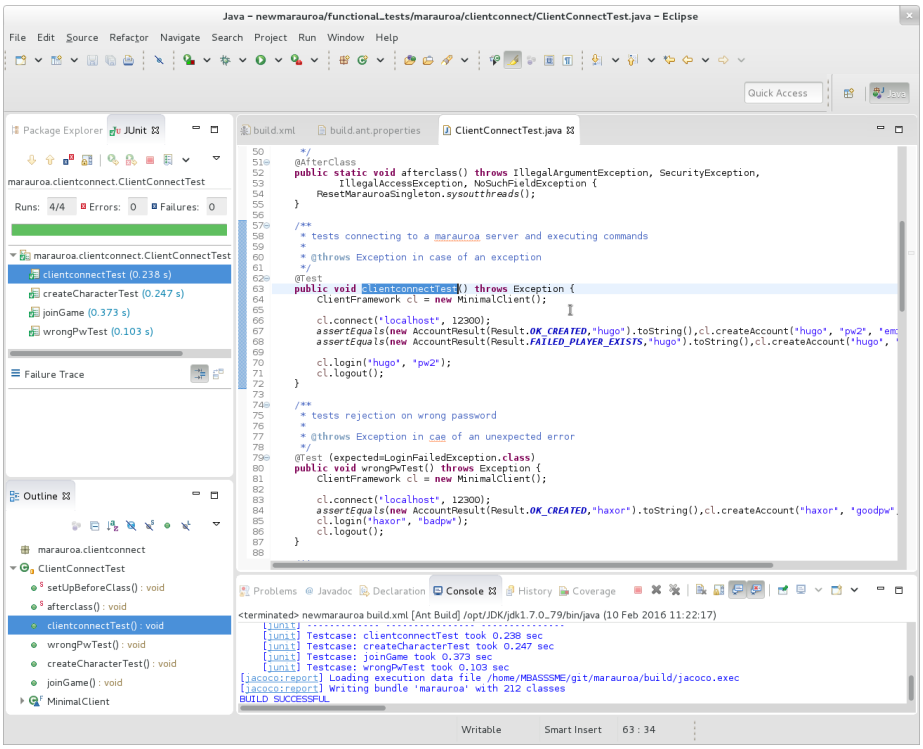

Figure 1.22: The Editor view

You can see that the test in 1.22 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 1.23 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 1.23. 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 1.24 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

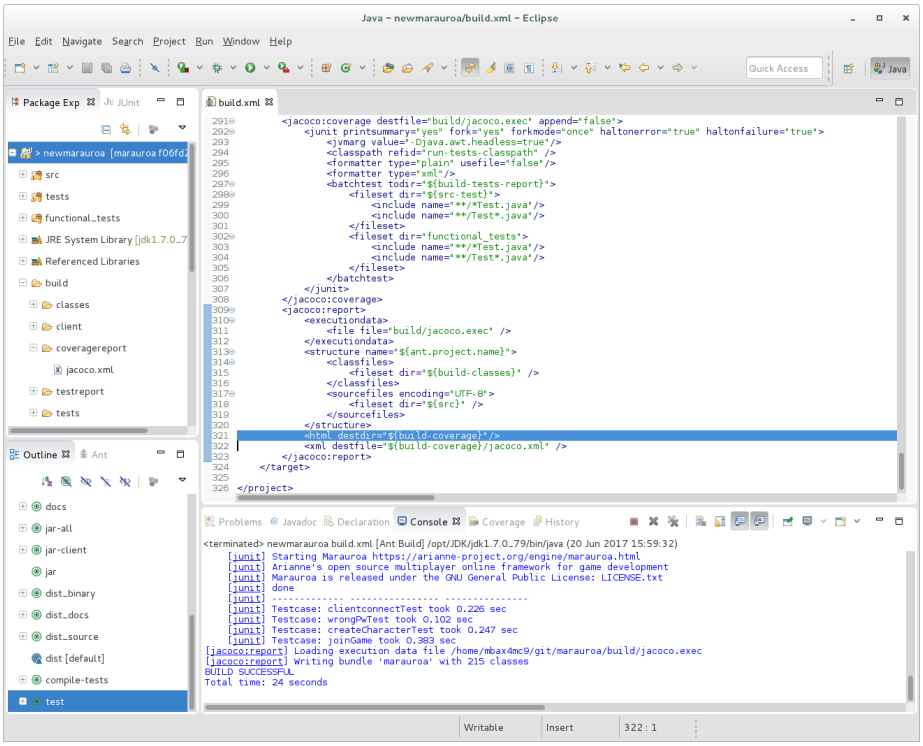

Figure 1.23: The JUnit view

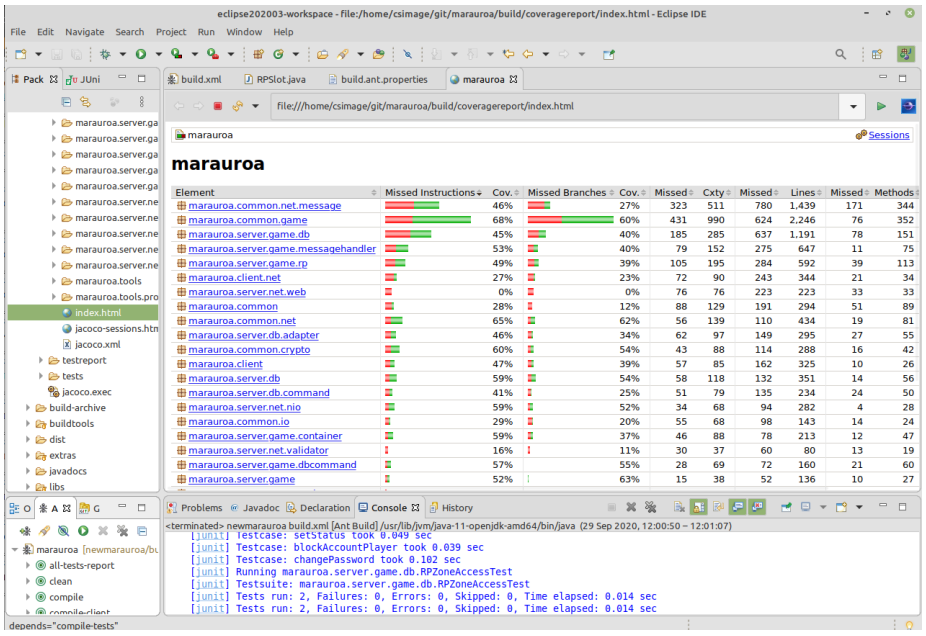

Figure 1.24: A test report

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 1.25 shows the detailed coverage report for the method `marauroa.common.game.RPObject.size()`.

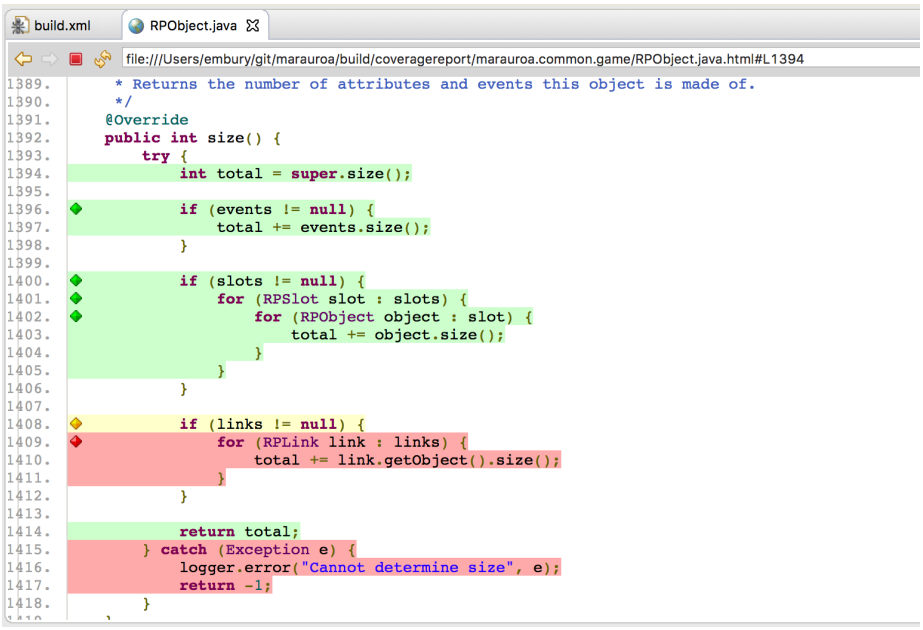

Figure 1.25: A more detailed coverage report

In figure 1.25 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 1.26

Save the file and run the build test target.

This time, you should see a failed build like the one shown in figure 1.27. 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

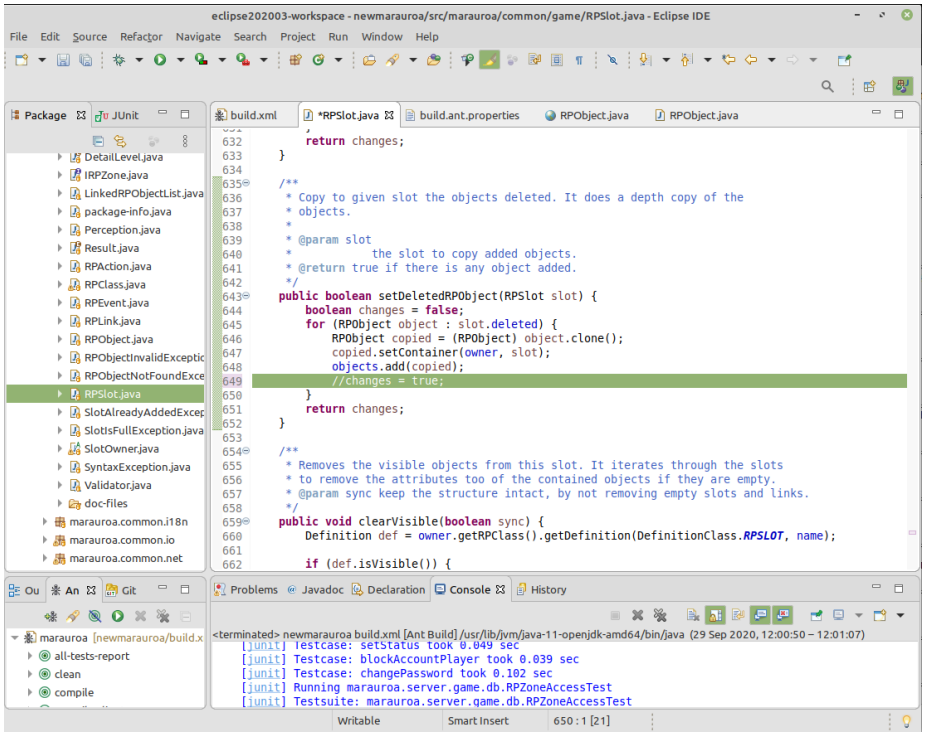

Figure 1.26: A more detailed coverage report

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

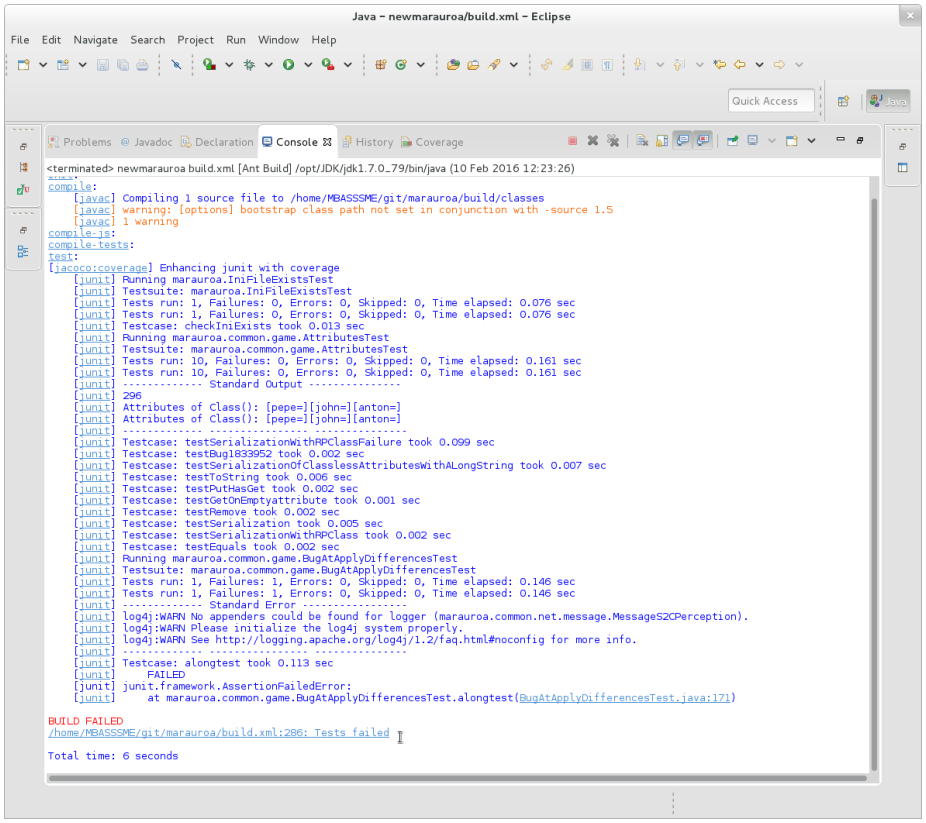

Figure 1.27: A more detailed coverage report

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 1.28.

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:

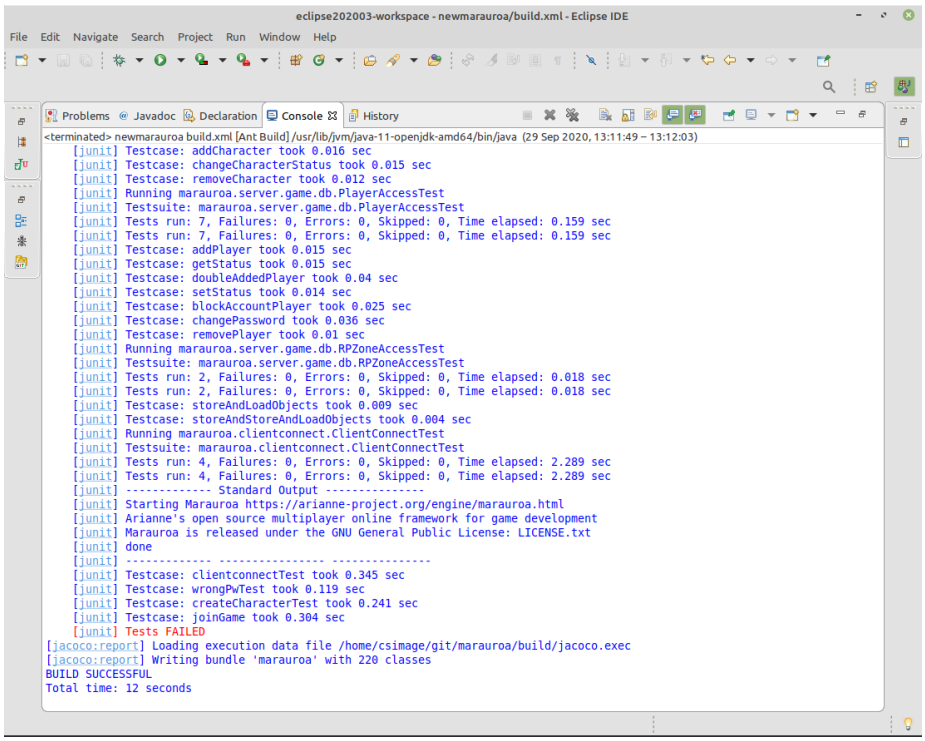

Figure 1.28: Tests FAILED

```
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

Understanding large systems

After you leave University or during an internship or placement, many of the systems you'll work will be large so its crucial to be able to understand large systems. You need to develop strategies for understanding unfamiliar systems.

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 the 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.

2.1 Learning Large Codebases

2.1.1 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.

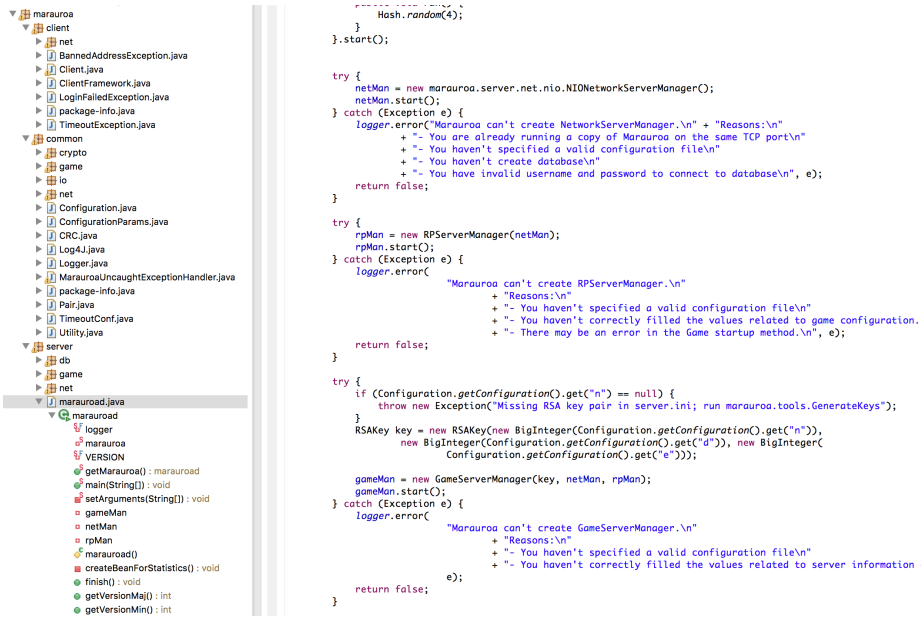

Figure 2.1: marauroa is a large codbase, how can you find your way around it?

2.1.1.1 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:

2.1.1.1.1 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.

2.1.1.1.2 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.

2.1.1.1.3 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.

2.1.1.1.4 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.

2.1.1.1.5 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.

2.1.1.1.6 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.

2.1.1.1.7 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.

2.1.1.2 Comprehending Marauroa

Figure 2.2: A screenshot from marauroa

Now, using the tips outlined in section 2.1.1.1, 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 2.1. 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. 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.

2.2 Unit Testing Overview

2.2.1 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.

2.2.2 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).

2.2.3 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). (Koskela, 2013)

2.2.4 Tips and tricks

Some tips and tricks for test-driven development.

2.2.4.1 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();

}

@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);

}
```

2.2.4.2 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.

2.2.4.3 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.

2.2.4.4 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.

2.2.4.5 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.

2.2.4.6 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.

2.2.5 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 2.3
   - **Class under test**. You can specify which is the class under test in the last field of the dialogue menu. Click `Browse...` see figure 2.4
   - 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.

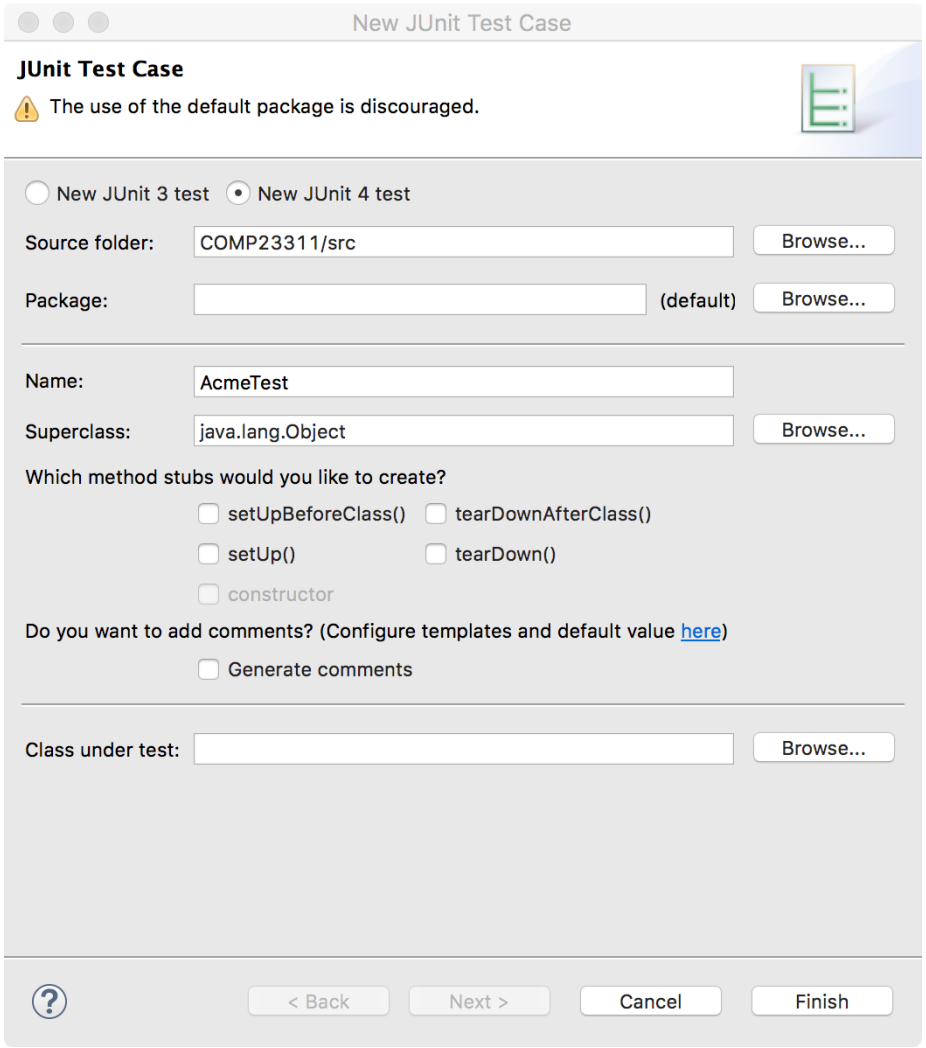

Figure 2.3: JUnit Test Case, with Test appended to the Acme class

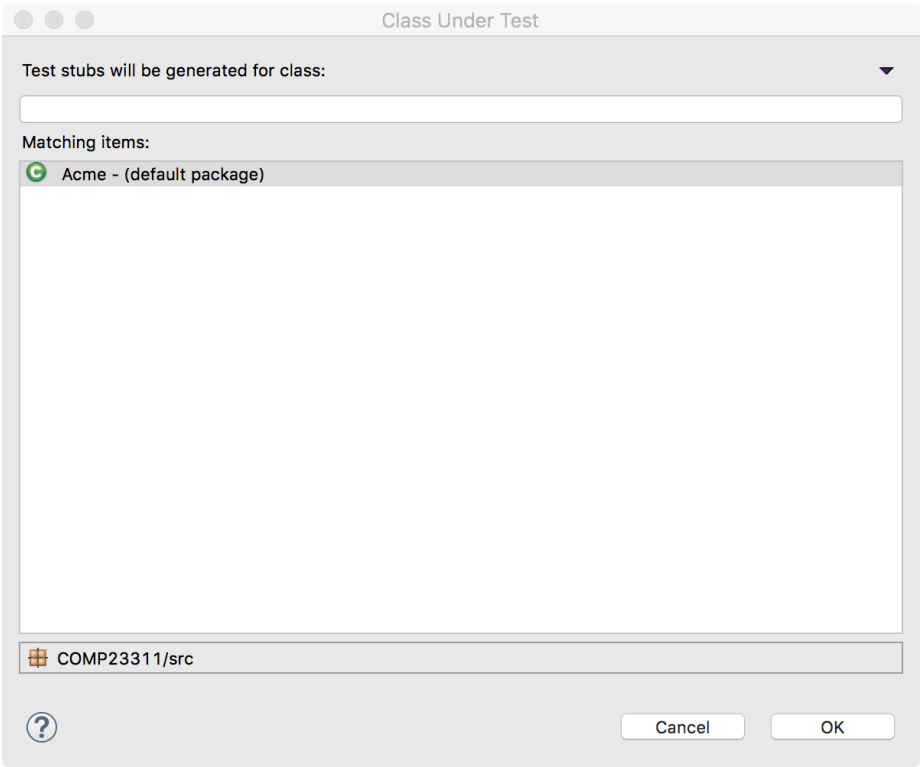

Figure 2.4: The Class Under Test in this example is Acme

2.3 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.

2.3.1 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.

2.3.2 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

2.3.3 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

2.3.4 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.

2.3.5 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.

2.4 JUnit Cheatsheet

A cheatsheet for JUnit:

2.4.1 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.

2.4.2 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.

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 debugging 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 3.1 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.

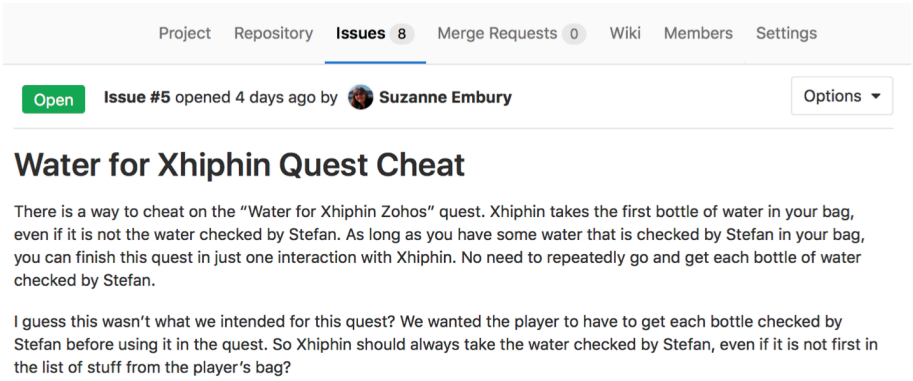

Figure 3.1: An example bug report as an issue in GitLab. GitLab issues are very similar to GitHub issues if you're familiar with those: guides.github.com/features/issues/

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

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

Material to be added here

6.1 Course content

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 code base, you'll need to look at the workshop instructions for week 3, 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 (Gamma et al., 1994)
- 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 9.1

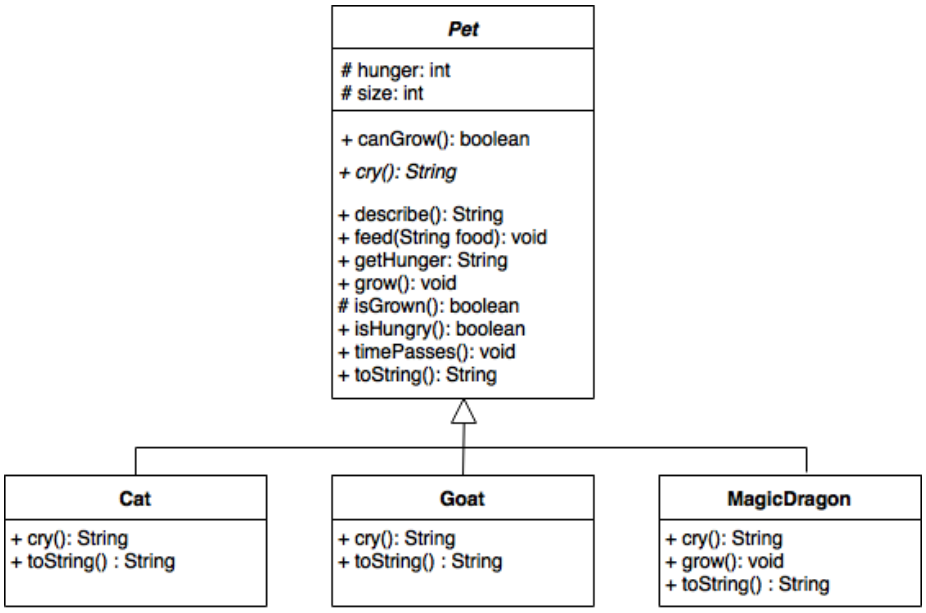

Figure 9.1: Pets, Cats, Goats and Magic Dragons

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 9.2

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 9.3

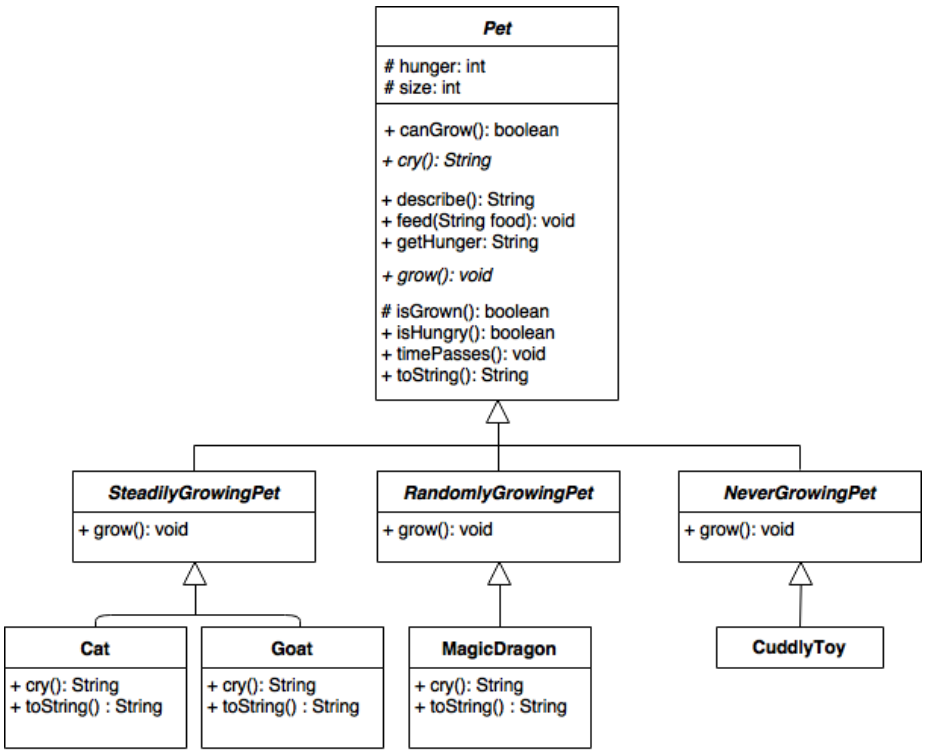

Figure 9.2: Possible subclasses of Pet

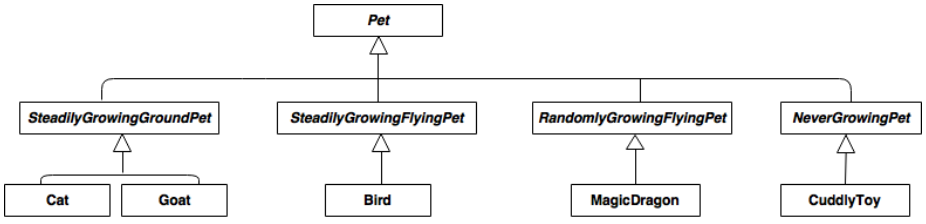

Figure 9.3: A badly designed hierarchy

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.

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 9.4 should be a good representation of your codebase at the end of this migration task.

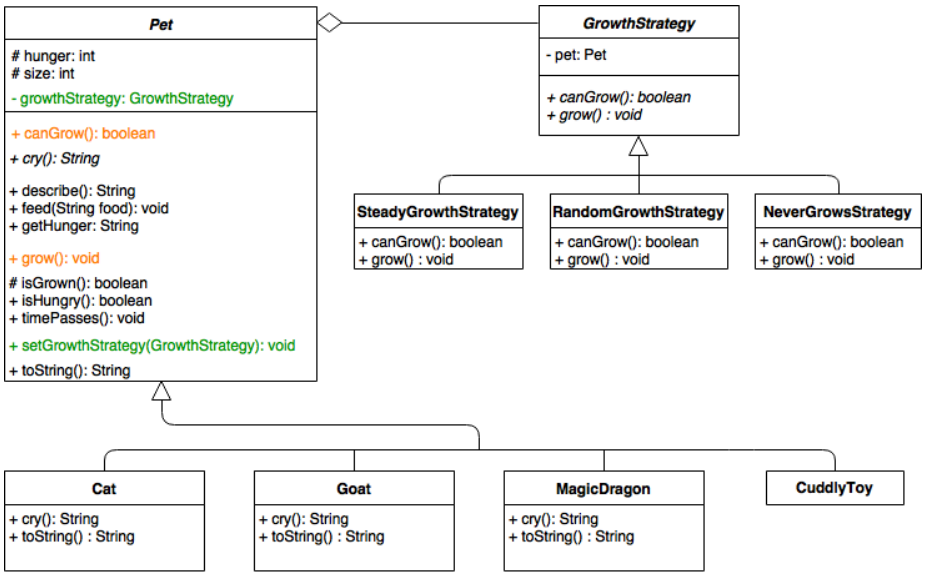

Figure 9.4: Your codebase should look something like this at the end of this migration task

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 9.5

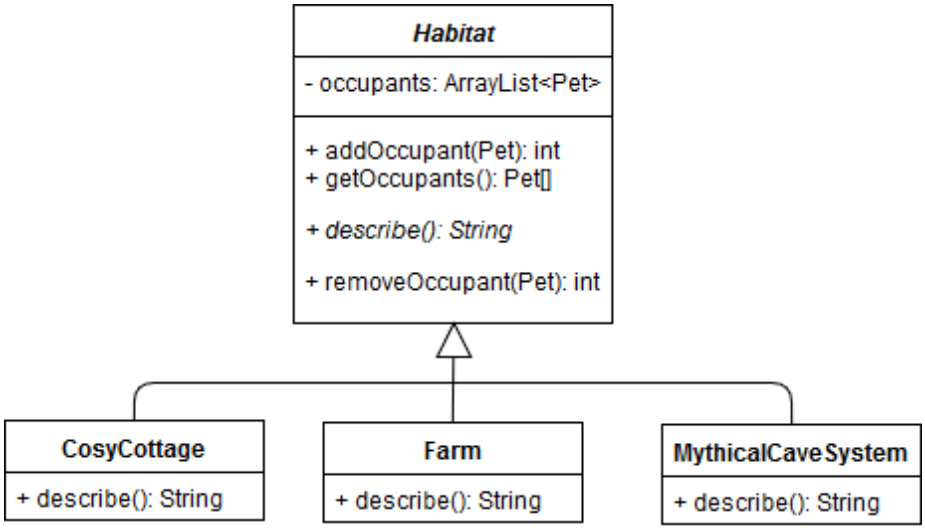

Figure 9.5: Subclasses of Habitat

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 9.6 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.

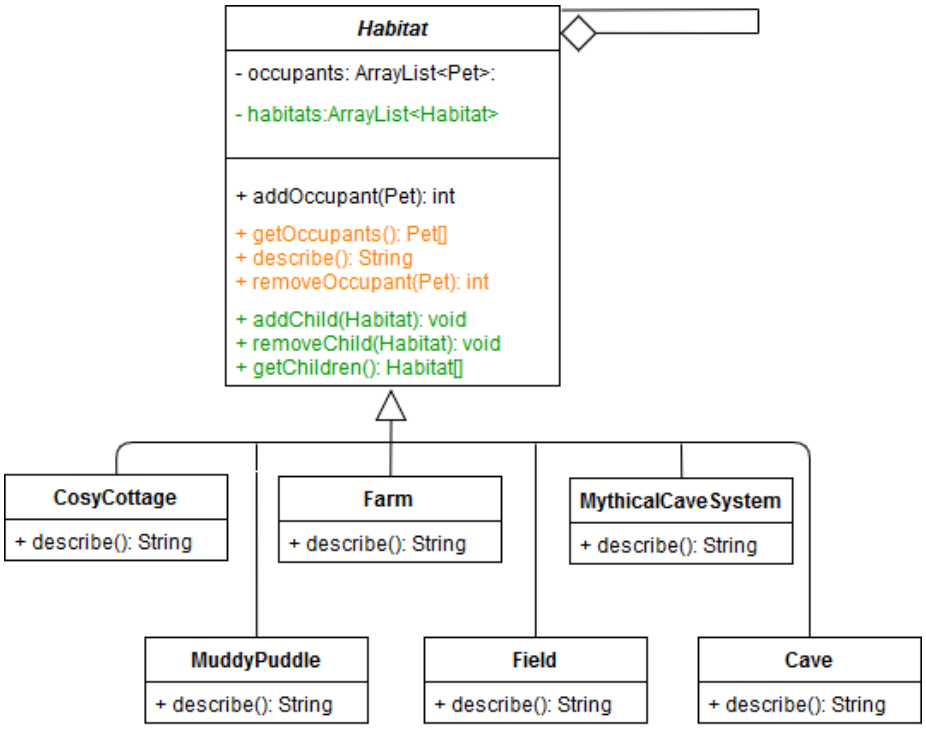

Figure 9.6: Your codebase should look something like this at the end of this migration task

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

Content to be added here

10.1 Course content

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. (Hull, 2021; Spinellis, 2021)

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 langauges see *Coding Your Future*. (Hull, 2021)

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

Stendhalgame.org

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.

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 in Stendhal 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

13.1 Your mentors

Information on mentors here

13.2 Getting the most of your mentor

Chapter 14

Synchronising

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

14.1 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.

14.2 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.

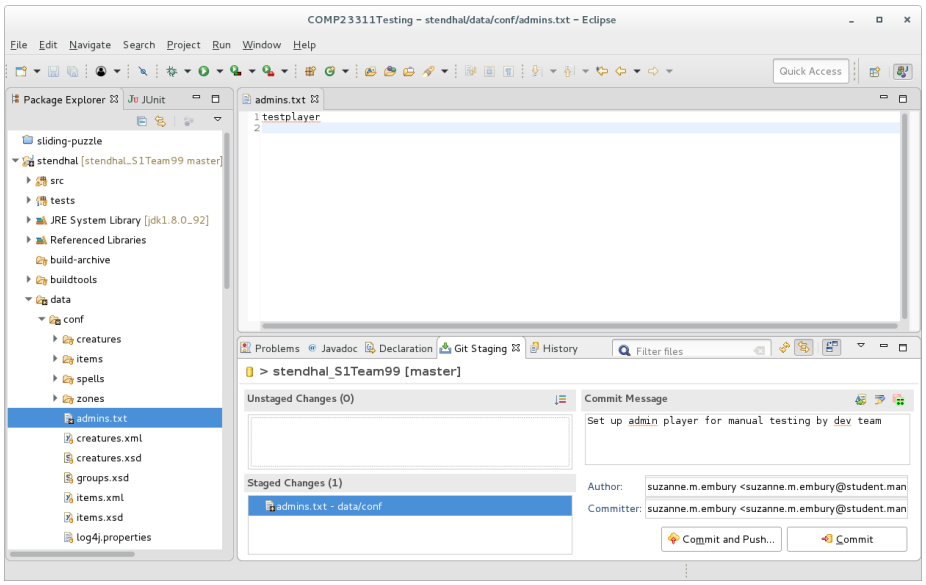

Figure 14.1: Your main eclipse window should look something like this

14.3 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 14.2

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 14.3

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

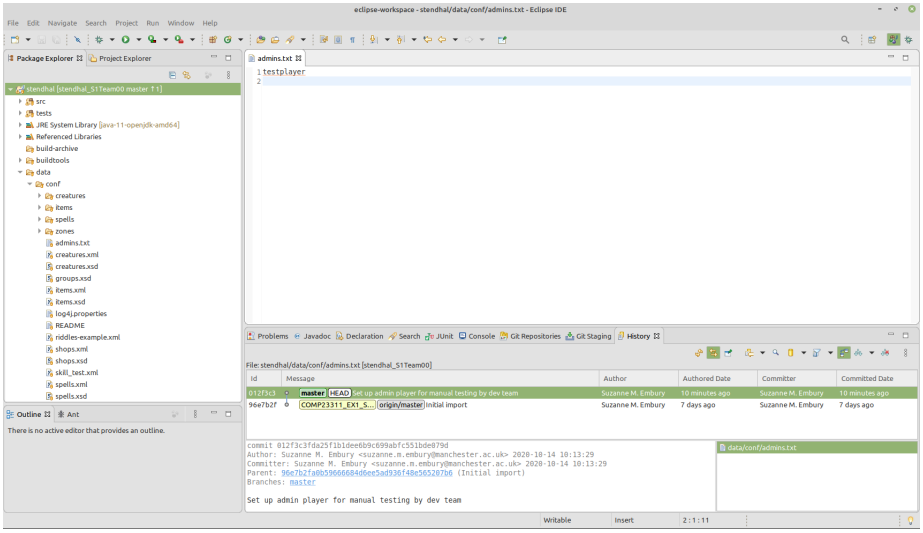

Figure 14.2: Your setup should look something like this

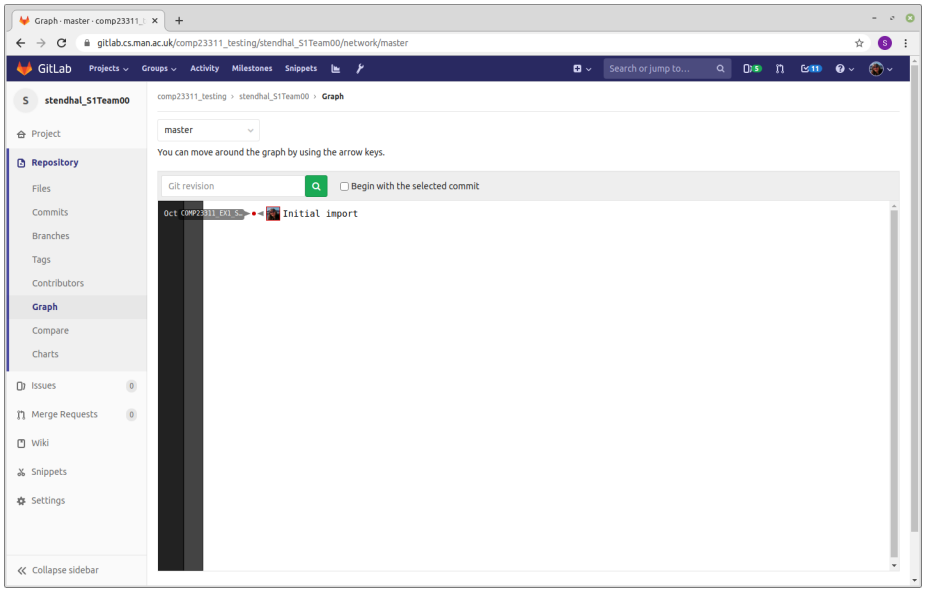

Figure 14.3: Your gitlab repository (gitlab.cs.man.ac.uk) should look something like this.

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 14.4

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.

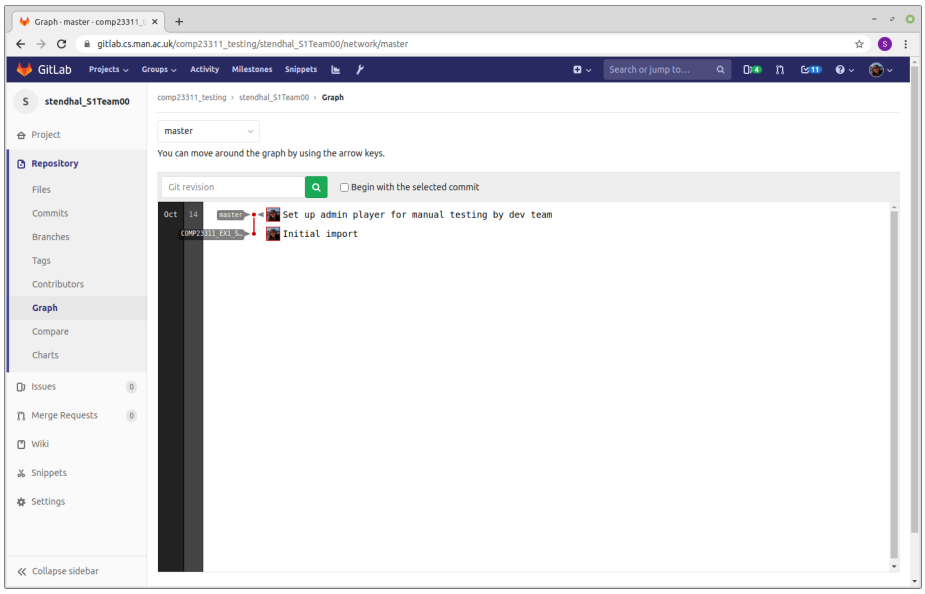

Figure 14.4: Your commit graph should look something like this

14.4 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 14.5.

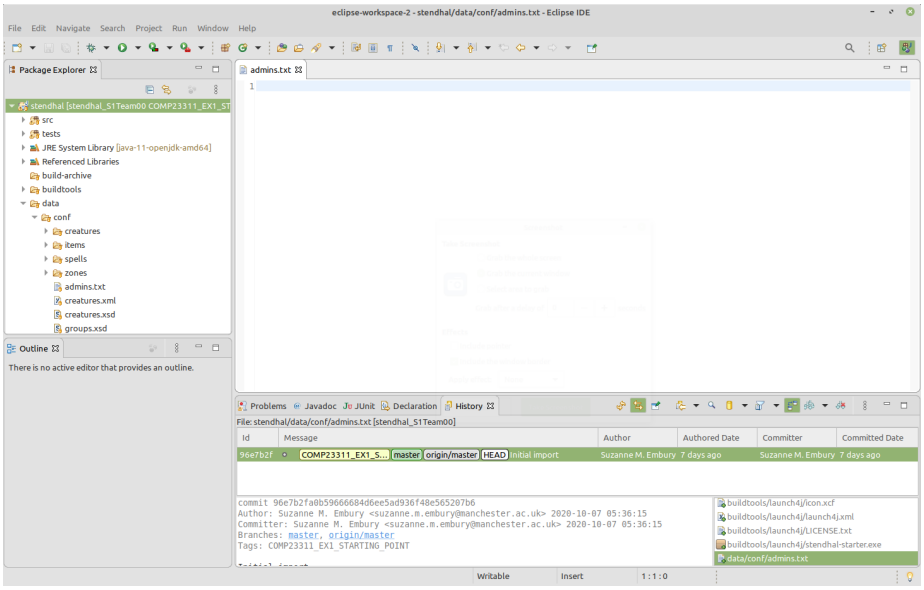

Figure 14.5: Your commit graph should look something like this

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 14.6

Your `History` view should also have updated to show the results of the fetch operation. It should now look something like figure 14.7.

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.".

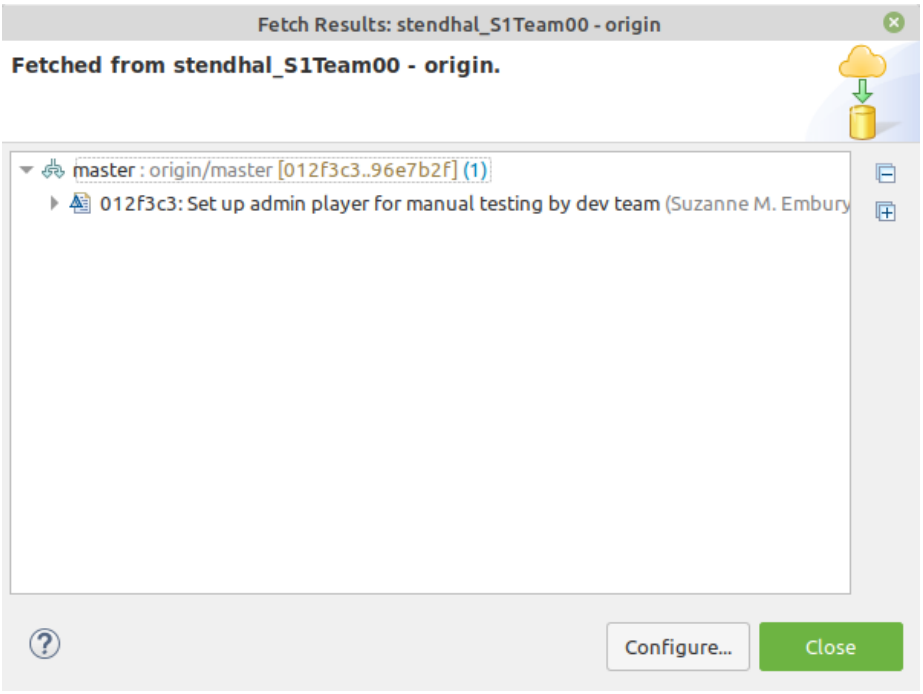

Figure 14.6: The fetch results dialog box

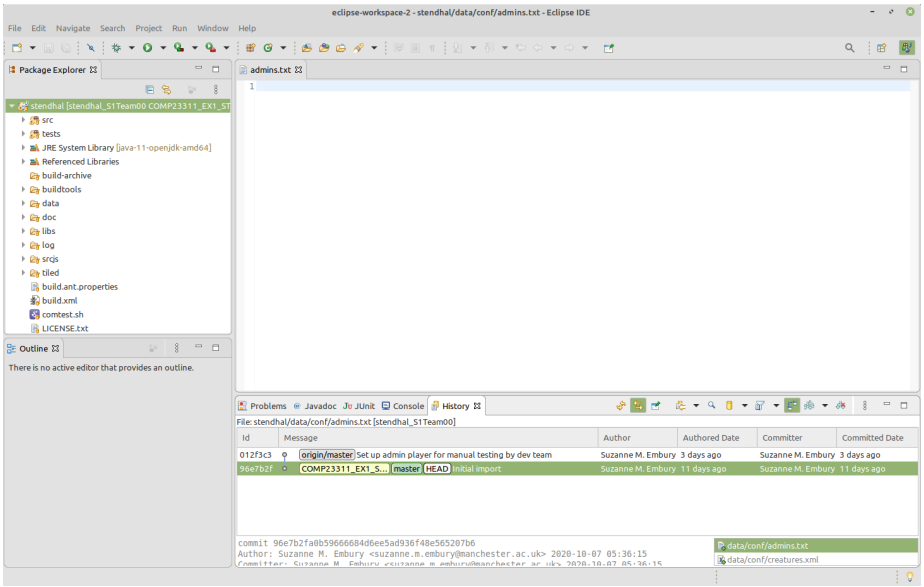

Figure 14.7: An updated History view showing the results of the fetch operation

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.

14.5 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 20 "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 14.8.

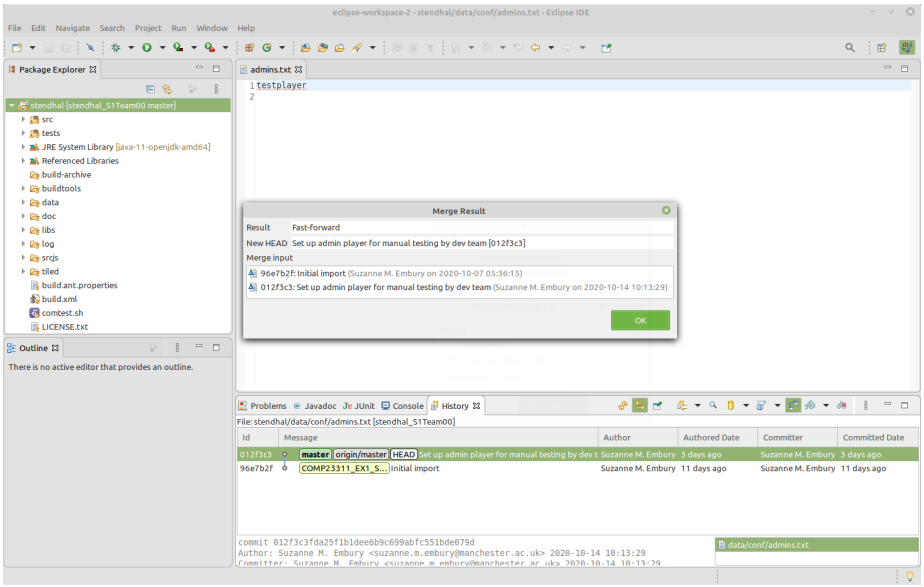

Figure 14.8: Your commit graph should look something like this

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.

14.6 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 20 on "Integrating Your Commits with your Team's Commits".

Good luck!

Part III

Coursework

Chapter 15

Individual Coursework 1

will go here (basic git)

Chapter 16

Individual Coursework 2

Exercise 2: conflicts in git

Chapter 17

Team Coursework 1

Team coursework exercise 1: Dealing with Small Scale Code Changes

Chapter 18

Team Coursework 2

Working with Features

Part IV

Self study materials

Chapter 19

Testing Stendhal

Guidance for Testing Stendhal.

19.1 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 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.

In this approach, we start off with a green JUnit bar (because although the bug is present, no test reveals it), before moving to a red JUnit bar (because we have added a test that reveals the bug) and then back to a green JUnit bar (because we have a test that reveals the presence of the bug, but the bug is now fixed).

This document contains some suggestions for carrying out the first few of these steps more efficiently. You'll look at techniques for carrying out the full process in the workshops in week 3.

19.2 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 targetted 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 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.

19.3 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 20

Integrating commits

Integrating Your Commits with Your Team's Commits

20.1 Introduction

In the team study activity, Syncing with your Team Repository, we practised how to synchronise your local repository with your team's when someone in your team had pushed work to the remote since you last synchronised. In this follow on self-study document, we'll cover the procedure we recommend you to follow when you have work on your local `master` branch that you need to integrate with work at the team remote. It covers more complex cases than the team study activity. but ones which are very common when working on a collaborative coding project using Git.

Note that this document is an explanation of various Git concepts, using a running example based on the example used in chapter 14 *Syncing with your Team Repository*. You won't be able to follow along with these exact steps on your own project, but you'll be able to use the concepts and ideas we describe in your own work afterwards.

20.2 Git has Rejected my Push

Suppose you and a team mate have both made commits on your local `master` branches, from the starting commit of the project. You worked on a feature branch, and merged it into the development branch (using a fast-forward merge) so your local history looks like figure 20.1.

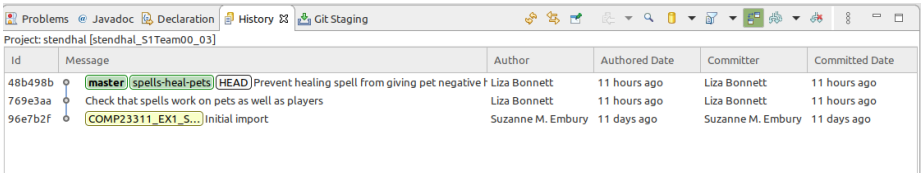

Figure 20.1: Your local history

Your team mate made a single commit directly onto the development branch and pushed their work to the team repository, so that the remote commit graph looks like figure 20.2.

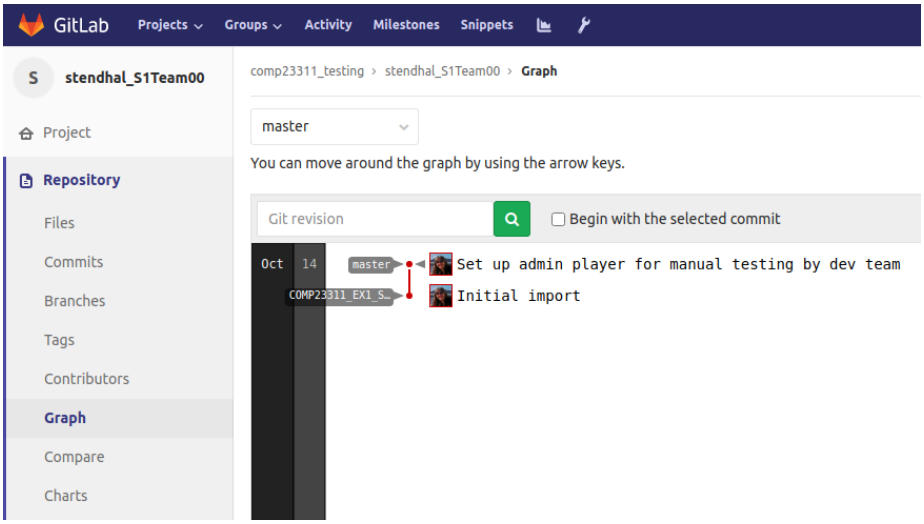

Figure 20.2: Your remote commit graph

When you push your work, Git rejects the attempt, shown in figure 20.3.

Can you see why?

When you push your development branch to the remote, Git has to try to work out how to combine these two graphs. The only commit that both graphs have in common is the starting commit (`96e7b2f`), so that will appear once in the combined graph, with two lines of commits extending from it—the line leading to your local `master` branch and the line leading to your team mate's `master` branch, at its position when it was last pushed to the team's remote repository.

Combining these commit graphs is easy and Git can do it for us automatically. The problem comes when Git tries to work out the position of `master` branch in the combined commit graph. Git needs to reconcile the current position of `master` (at commit `012f3c3f`) with the new position you are suggesting for it

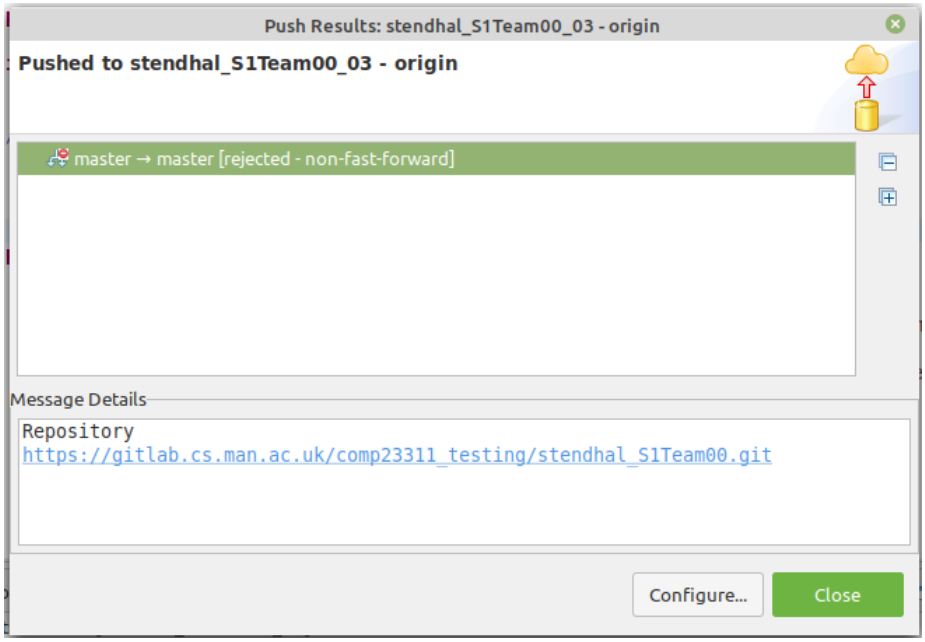

Figure 20.3: Can you see why git has rejected this push?

(commit `48b498b`). It tries to do this using a fast-forward merge, and if that isn't possible it will reject the attempt. In this case, there is no path through the combined graph linking the two `master` branch positions that does not involve going backwards in time. Therefore, a fast-forward merge is not possible, and Git rejects the push.

When Git rejects a push, it means that we need to synchronise the state of our local repository with the team repository. Then we can push again (and hopefully be successful—provided no work has been pushed to the repository in the meantime).

As before, synchronisation follows two steps:

1. Fetch down the new commits from the remote repository.
2. Integrate your existing commits with the new commits.

20.2.1 Fetching the New Commits from the Remote

This step is straightforward. Just right click on the project name in the `Package Explorer` view, and select `Team` > `Fetch from origin`. Your local commit graph now looks like figure 20.4

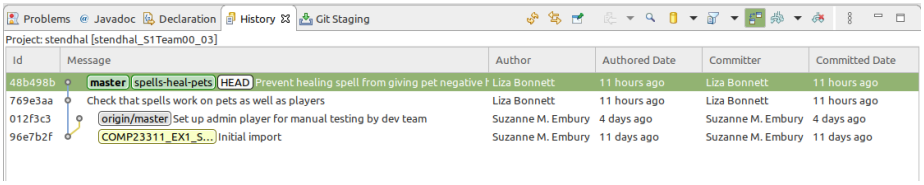

Figure 20.4: Your local commit graph

We can see from this why a fast-forward merge of the remote tracking branch for `master` into our local `master` branch was not possible.

The next step is to integrate the work we pulled down from the team remote with our own, so that the result can be pushed to the remote and be visible to other team members.

Git provides two mechanisms for integrating separate lines of development work: merge and rebase. They are both useful, but have different strengths and weaknesses. Different workflows recommend they be used in different ways. For example, GitFlow mandates the use of non-fast-forward merges, even in situations where fast-forward merges are allowed, while Cactus Flow requires the use of rebase for all code integration events. Other workflows mix merge and rebase, as we do in the simple Feature Branches workflow we ask you to use in this course unit.

In the situation we are looking at now, synchronising your work with the work of your team, **we recommend that you use rebase rather than merge**. In the remainder of this document, we'll look at both strategies, and explain why we make this recommendation.

20.3 Integrating the New Commits using Merge

The Git merge operation takes two branches, one pointing to the line of development that needs to be extended with the new changes (the *target*} branch), and the other pointing to the line of development containing the new changes that need to be integrated (the *source*} branch). There are three possible options for any merge:

1. The changes are already on the target branch (because the source and target both point to the same commit, or because the source branch points to a commit that is an ancestor of commits on the target branch). In this case, the merge succeeds without Git having to do anything.
2. The changes are on the same line of development as the target branch, but ahead of it. The target branch just needs to be moved forward along the line of commits, to reach the same point as the source branch. This is the fast-forward merge case.

3. In all other cases, we need to create a new merge commit, with two parent commits, the source branch and the target branch. The target branch is moved forward so that it points to the new merge commit, and so includes the commits in the source branch as well as all the commits it pointed to before. This is the non-fast-forward merge case.

Looking at our example and the combined graph after the Fetch operation, we can see that the 3rd of these cases applies.

To make the merge commit, **first make sure that you have checked out the branch you want to integrate the new commit into**. In this case, this is our local `master` branch. We can see from the `History` view contents that it is already checked out.

Right click on the commit pointed to by the branch we want to integrate from: in this case, the commit pointed to by branch `origin/master`. Select `Merge`.

This succeeds, and produces the summary shown in figure 20.5

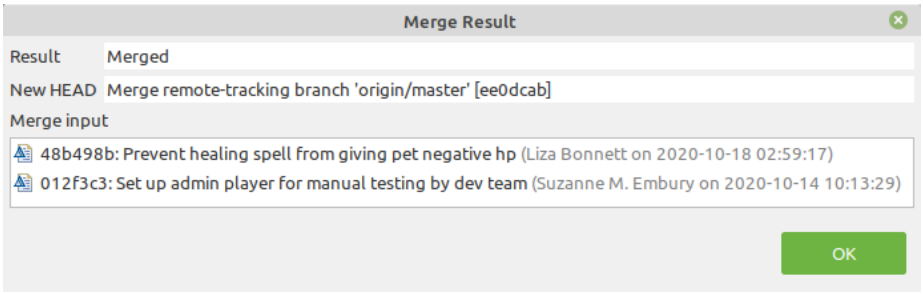

Figure 20.5: Merge result dialog box indicating success

This tells us that the merge was not a fast-forward merge, that the checked out branch (`HEAD`) has been moved forward to point to the new merge commit, and reminds us of which commits were merged by this operation. The `History` view shown in figure 20.6 us the new local commit graph after this operation has completed:

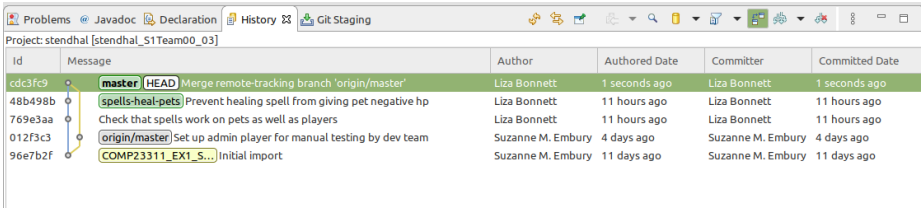

Figure 20.6: The History view shows us the new local commit graph

Here you can clearly see the new merge commit, with its two parent links. The code base it represents contains the changes made by us (the fix of the healing

spell bug) and the changes made and pushed by our team mate (the addition of an admin player to the `admins.txt` file.

Notice that only the checked out branch (`master`) changed its position as a result of this operation. Our feature branch and the remote tracking branch for `master` both remain as they were before we made the merge.

If you compare our new local commit graph with the graph at the remote (still as it was at the start of the scenario) shown in figure 20.7, you'll see that Git can now perform a fast-forward merge of the new commits on our local `master` branch with `master` branch at the remote.

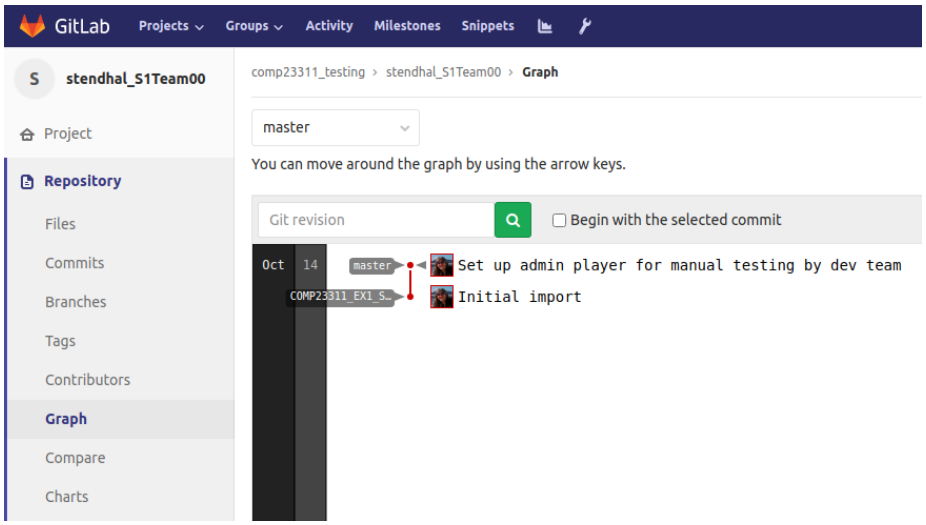

Figure 20.7: New local commit graph with the graph at the remote, still as it was at the start of the scenario

So now, we can push our work, and the push should succeed.

The remote repository now contains our commits, integrated with our team mates' commits, on `master`.

(Note that we didn't push our feature branch, so it is not visible on GitLab ((gitlab.cs.man.ac.uk)[https://gitlab.cs.man.ac.uk/], even though the commits that were made on it are now present in the team remote. Git keeps a clear distinction between commits and branches. If you push a branch, you also push the commits it is pointing to, but you don't automatically push all branches or tags that are pointing to those commits.)

In our local repository, Git has updated the position of the remote tracking branch, `origin/master`, to reflect the change in position of the branch in the remote.

Merge in Git is a great tool when we want to integrate feature branches into our

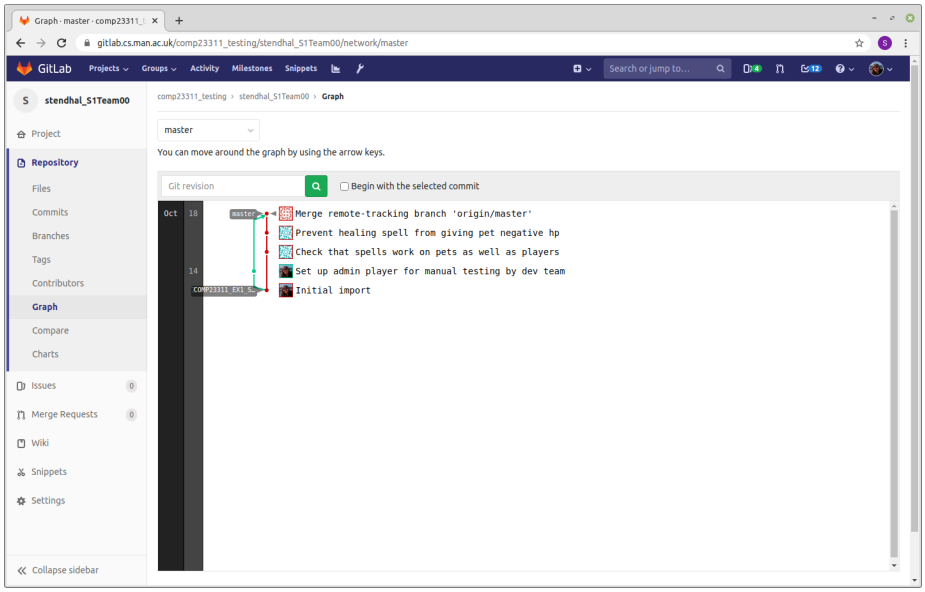

Figure 20.8: The remote repository now contains our commits

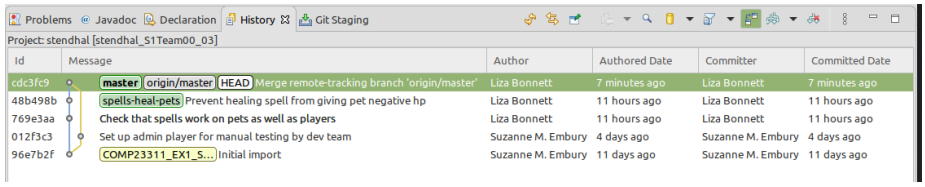

Figure 20.9: Git has updated the position of the remote tracking branch

development branch, or hotfix branches into our release branch, or any situation where a line of development has reached a stable state and needs to be integrated with the next line of development in the pipeline leading to released code.

But, it is not a great tool to use for synchronising our work with our team's. This is because it creates unnecessary merge commits and branching structures in our code history that don't reflect key points of code integration, but simply reflect that we decided to synchronise our code at that time. This isn't information that we need to keep for future readers of the code base. It complicates our code history, making it look less linear and clean than it should. And anything that makes our code base harder to read is an added cost that we should avoid if we can.

Using the second technique for code integration, Git rebase, avoids this problematic cluttering of the code history. It requires a little more effort on the developer's part, but soon becomes second nature. We'll now explain how to carry out the same synchronisation task just discussed, but using rebase instead of merge.

The Git Pull Command You may be wondering why we have not so far mentioned the Git `pull` command, which is commonly talked about as being the way to quickly sync up with your remote repository. Git pull (in its vanilla form) is just syntactic sugar for the execution of two other Git commands: `git fetch` followed by `git merge`. It's a handy shortcut for whenever you want to use the merge approach to synchronising your code, but also therefore shares all the same limitations of using merge for synchronisation. Whenever you execute `git pull` you may be creating a new merge commit in your code history.

It is possible to configure the pull command to work differently (using rebase instead of merge, for example). For this course unit, and especially for students who are new to collaborative coding using Git, we recommend that you avoid the use of the pull command, and instead carry out the two steps separately for yourself. This does not require many more mouse clicks, and allows you to see what is happening at each stage, and adjust your next steps accordingly. It also means you gain a much more solid understanding of what Git is actually doing, rather than just issuing a pull command and hoping for the best.

20.4 Integrating the New Commits using Rebase

Let's go back in time, to the point at which we started to integrate the commits we had fetched down from the remote. (Luckily, we are using Git, which is a tool for going back in time very easily.) Our local commit graph looks like figure 20.10

And our team remote looks like figure 20.11.

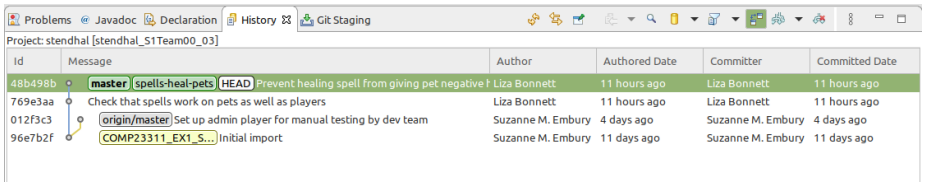

Figure 20.10: Our local commit graph

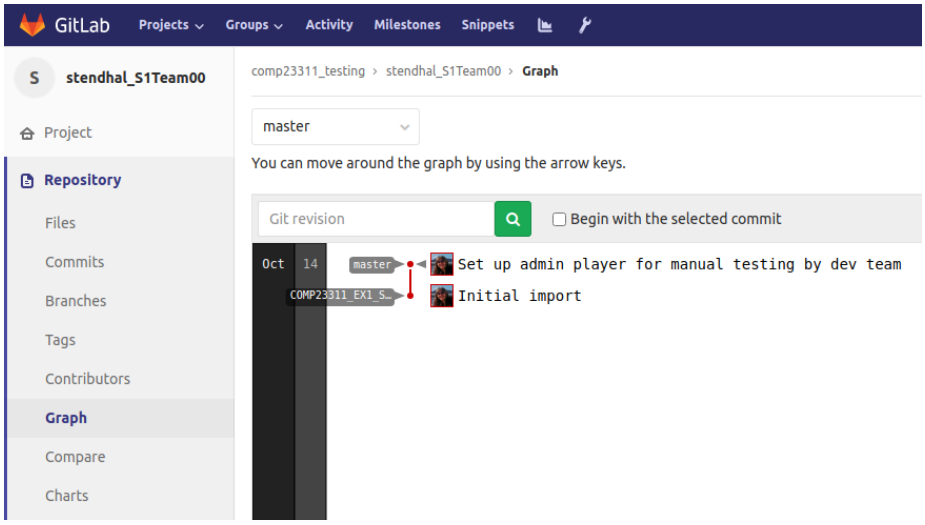

Figure 20.11: Our team remote graph

This time, we're going to use rebase to perform the synchronisation. When we rebase branch A onto branch B, we are asking Git to replay the commits that are unique to branch A on the top of branch B, as though we had made the same changes with B checked out as we did formerly with branch A checked out.

For example, let's look at a simple commit made by the Stendhal development team to prepare one of the 2019 releases shown in figure 20.12

Figure 20.12: A simple commit made by the Stendhal development team

This shows the change introduced by the commit. The red coloured lines (starting with `-`) are lines that have been deleted, and the green coloured lines (starting with `+`) are lines that have been added in the commit. Git stores the 3 lines before the change and the three lines after the change, to allow it to work out where the change should be applied. (Line numbers are also shown, but can't be relied upon entirely - if lines are added or removed to the part of the file before this, then the line numbers will be completely inaccurate.)

Although this change was made to a specific version of the code, the same change can be made to *any* version that includes this file, in a version that contains the lines that are modified by the commit and the preceding and following lines that identify them. In other words, the commit can be taken and replayed in some other part of the commit graph. Provided the affected parts of the files are the same, the commit makes just as much sense when replayed as it does when originally applied. This is rebasing.

We can use this notion to integrate our work, with the work of the team. Before we show how to use rebase in the case of our merged feature branch, we'll first illustrate the idea through some simpler scenarios.

20.4.1 Rebasing Commits Made on the Master Branch

Imagine we have made some commits directly to our `master` branch and need to synchronise these with a commit to `master` made by a team mate and pushed to the team remote. Our local commit graph looks like figure 20.13.

We can create a linear code history suitable for fast-forward merging by rebasing our two commits on top of the commit from our team mate. To do this, we

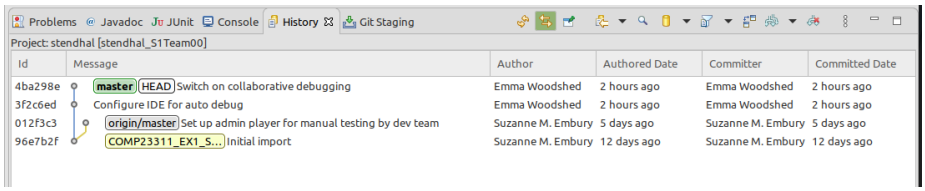

Figure 20.13: Our local commit graph

make sure we have our local `master` branch checked out. Then we right click on the commit we want to rebase onto (commit `012f3c3`) and select `Rebase HEAD on` from the menu that appears.

This produces a commit graph shown in figure 20.14

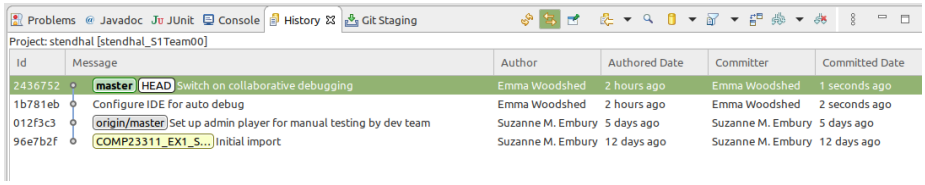

Figure 20.14: A commit graph, note the changes carefully

Look carefully at what happened here. It almost looks as though the two commits on `master` were moved across to the `origin/master` branch. In fact, they are brand new commits; you will see that they have different SHA identifiers. The change made to the code is the same, as are the commit message and the author details. But the committer details have changed to show a different committed date. And the committer could potentially have changed, if the person doing the rebase wasn't the same as the person making the commits in the first place.

The local `master` branch has now moved to point to the latest of the new (rebased) commits. The old commits are still present in the repository, but since they are now not reachable from any branch or tag, Git does not show them.

Since `master` is now ahead of its position in the remote repository, we'll be able to push it to the team repository. The `origin/master` branch will be moved forward to the tip of `master` as a result of the push. When a local branch is at the same location as its remote tracking branch, then we know that the repositories are in sync (at least as far as those branches go).

20.4.2 Rebasing to Sync Repositories When Working on a Feature Branch

Let's now consider a scenario where we are working on a feature branch. We created the feature branch at the initial commit, and have made a couple of commits on it but have not yet finished working on it. At the start of our working day, we want to synchonise our repository with the team's, so that we don't diverge too far away from the work the rest of our team is doing. After we've fetched in the commits from the remote, our local graph looks like figure 20.15.

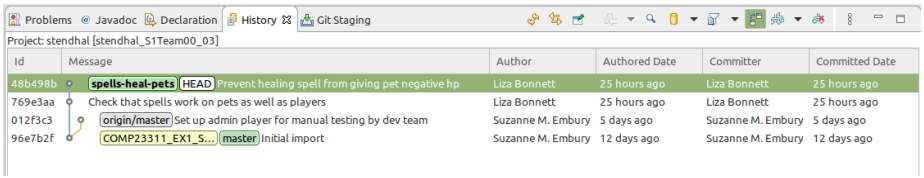

Figure 20.15: Our local graph after we've fetched in the commits from the remote

If we'd started work on this feature branch after our team mate had pushed their commit to the remote, we'd be able to synchronise easily. Our feature branch would have begun at commit `012f3c3`, where our local `master` branch would be. We'd be able to push the code to the remote straightaway, as all integration would be with fast forward merges.

We can use rebase to get our code base into this state.

First, we synchronise the position of our local `master` branch with the `origin/master` branch. To do this, we check out `master` then right click on the commit that `origin/master` is pointing to (commit `012f3c3`) and select `Rebase HEAD on`. The result looks like figure 20.16.

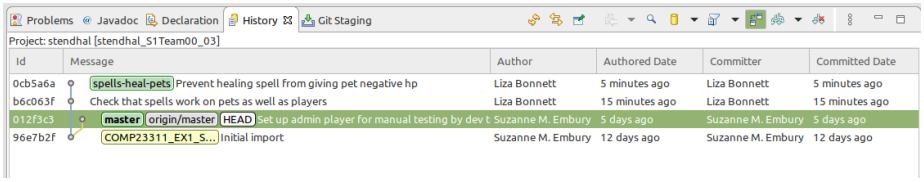

Figure 20.16: updated commit graph

Notice how we appeared to have performed a fast-forward merge of `master` into `origin/master`? There is no difference between the commit graphs that result from a fast-forward merge and from a rebase, in this case.

Next, we need to replay our feature branch commits on top of `master`. Check out the feature branch, then right click on the commit where the two `master`

branches are located and select `Rebase on HEAD`. The commit graph will now look like figure 20.17:

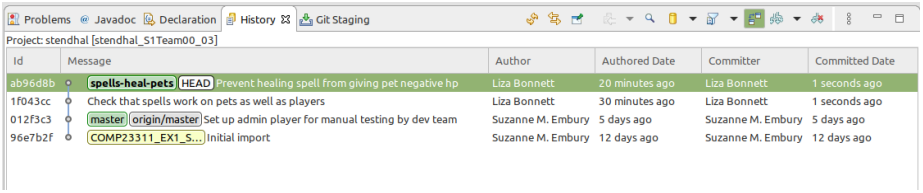

Figure 20.17: New commit graph

As can be seen, we now have a synchronised (and tidy!) commit history. The local `master` branch is at the same location as its remote tracking branch (for now) and the feature branch is based on the latest version of the code. We can now push either `master` or the feature branch successfully, and can continue work on the feature branch knowing we are building on a recent version of the code.

Only Rebase Local History, Not Shared Public History

Rebase is a little riskier than using merge. Merge leaves the whole history intact, only adding a merge commit to it, so there is no risk of losing any commits. But rebase rewrites the history of the code changes, potentially in quite radical ways. Information, and in some cases even commits, can be lost.

While it is no problem to rewrite the code history in your local repository, it's vitally important that you don't attempt to change the history of your team's remote. That is confusing and disruptive for everyone on the team, and runs a significant risk of losing commits—not your own commits, but the commits of your colleagues. They will not be pleased… So when using rebase it's important to keep in mind which commits in your local commit graph also exist in your team's remote and which commits are just local to your own code history. Commits in the former category should not be rebased. Commits in the latter category can be.

20.4.3 Rebasing to Sync Repositories After a Fast-Forward Merge

We can now finally return to the scenario we started this document with. As a reminder, in our local repository, we had just merged a feature branch with our local `master` branch, and then discovered that new commits had appeared on the remote master when we synchronised before attempting to push:

This scenario gives an example of how rebase requires a little extra thought before using it. We need first to decide what we want the code history to look like, and then we can use rebase to make that happen.

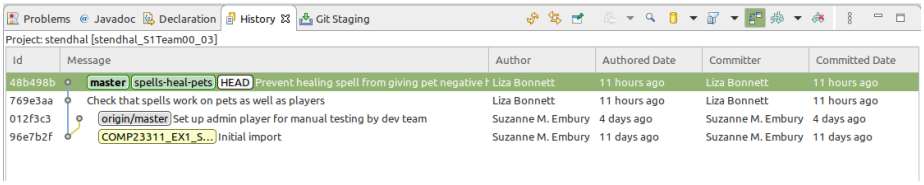

Figure 20.18: A reminder of the commit graph

In this case, we need to decide what we want the history of the merged feature branch to look like. If we had remembered to fetch and sync before starting work on the feature branch, we would have ended up with a fast-forward merge of the branch on `master`, with its parent at `origin/master`. So, let's make the history look like that's what happened.

To achieve this history, we first reset the local `master` branch to point to `origin/master`—the commit it would have been on, if we had remembered to sync before creating the feature branch. Check out `master`, then right click on the `origin/master` commit and select `Reset` > `Hard`.

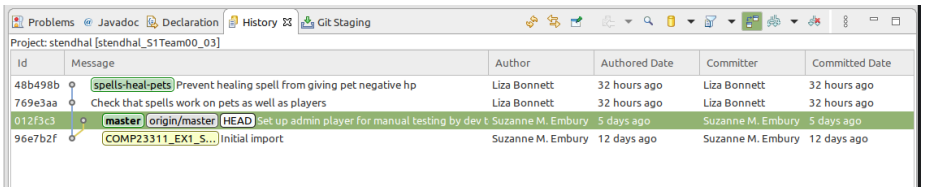

Figure 20.19: local commit graph after reset

Then we rebase the feature branch on top of `master`. We first check out the feature branch, and then right click on the `master` branch commit, and select `Rebase HEAD on`. This results in the graph shown in figure 20.20

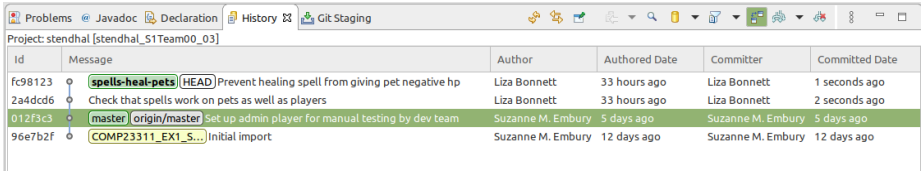

Figure 20.20: Resulting commit graph after feature branch rebasing

Now we have reorganised the history to the point where we are ready to merge the feature branch into our development branch. This is a feature branch integration step, so we should strictly speaking use Git merge. But since it will be a fast forward merge, there is no difference in terms of the commit graph outcome between using merge and rebase in this case.

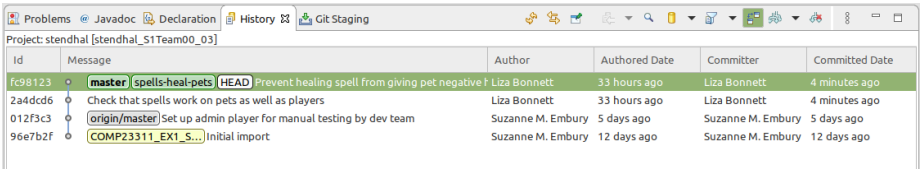

Figure 20.21: Reorganised history

The repository is now ready to push. Since `master` is ahead of `origin/master` on the same line of development, Git will have no problem in making the fast-forward merge needed to allow it to accept the push shown in figure 20.22

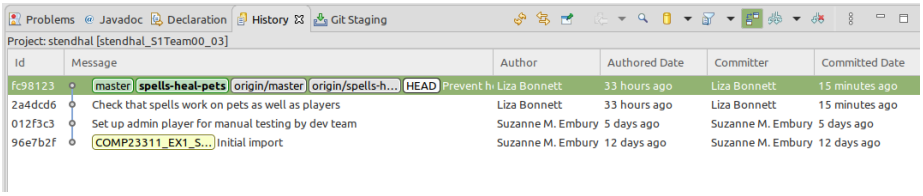

Figure 20.22: local commit graph after push

And our code history on GitLab looks pleasantly clear and linear as in figure 20.23.

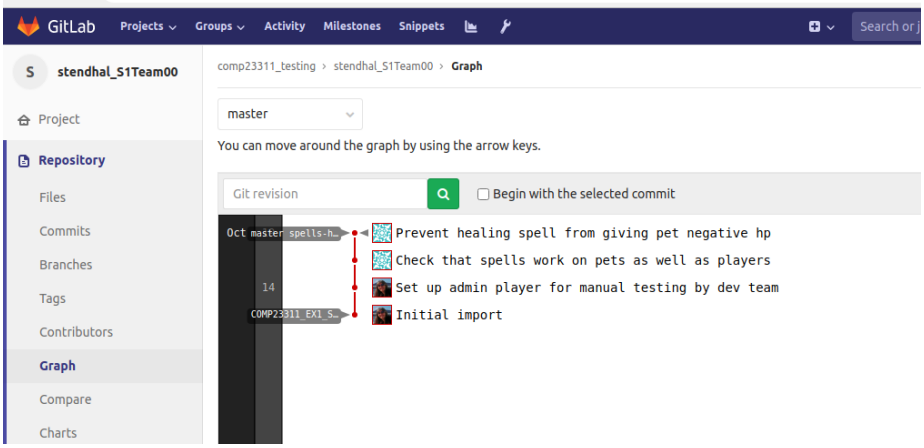

Figure 20.23: Our code history on GitLab is pleasantly clear and linear

Force Push Because rebase changes the code history, it is easy to get into a situation where your local history differs from the remote history in ways that mean Git refuses to push your work. This can be stressful (especially if close to a looming deadline) but you need to resist the temptation to use force push.

This almost always leads to loss of your colleague's commits and can be very disruptive for your team. Instead, you need to get your repository into a state where it can be pushed, without requiring the `--force` flag. As long as you stick to the rule of not changing commits that also exist in your team remote, all should be well. Though it is always worth leaving yourself enough time before the deadline to get help from your team mates or from the course team if things do go wrong.

20.5 A Final Word

In this document, we took you through several different approaches to synchronising your local repository with your team remote, after your team mates have made changes. In this course unit, we ask you to use:

- Git Merge for integrating finished feature branches into your development branch, and
- Git Rebase for synchronising your repository with your team remote.

This allows you to practice both styles of code integration in the single project, while also keep to a fairly simple and well-used workflow (based on feature branches).

This is by no means the only workflow you could use, and we certainly don't claim it as the best in all circumstances. But it is an approach that allows you to gain the core Git skills that will enable you to use any workflow that you may be called up on to comply with in future projects.

One major omission from this document is any discussion of what happens when we encounter conflicts while integrating code or synchronising our repository. We've chosen the issues for exercise 1 to try to avoid conflicts, but you will definitely encounter them in the second team coursework exercise, when you start to work on larger features in sub-teams. We'll cover handling conflicts in another learning resource for the unit.

Good luck!

Chapter 21

Continuous integration

Continuous integration (CI) is the practice of automating the integration of code changes from multiple contributors into a single software project.

21.1 Introduction to Continuous Integration

In COMP23311 so far, you have begun to use a number of modern software development tools that help us keep our code quality high, even when multiple developers are working on the same code base. We're now going to introduce another class of tool: the *continuous integration server*.

As the name indicates, these tools have the job of continually putting a software system through the processes that lead to releasable software. A typical continuous integration server will:

- Integrate the various components of the system.
- Compile the code and create an executable version of the system.
- Run the unit tests, integration tests and acceptance tests.
- Generate reports on the quality of the system (for example, test coverage reports).
- Inform the development team when some part of the above process fails.

These steps will be undertaken at regular intervals, usually based on some automated trigger. Developers do not need to remember to request that it happens; the CI server takes care of it for us.

The aim of repeatedly going through this process is to discover problems early, well before the release deadline, so that there is plenty of time to correct them.

Before such tools came into common use, developers concentrated on their own parts of the system, building and testing them in isolation, until the release deadline approached. Only then would the whole system be compiled and integrated, and the tests run on the full code base. Inevitably, some of the tests would fail or, in the worst cases, some of the code would fail even to compile. With the deadline looming, the panicked team would have to find and fix the problems under pressure, often leading to more bugs being introduced and poor decisions being taken that could have been avoided if only the team had thought to integrate, build and test their code as they went along instead of at the last minute.

There is a lot to be said for building and testing code frequently. When we go through this process after just a few small changes have been made to the code base, it's a fairly safe bet that the cause of any failing tests will be one of the changes we've just made. Even if our changes did not cause the error, something we have done has caused the error to become visible. In most cases, we'll only need to examine these changes to be able to track down the root cause of the error. The problem can then be fixed fairly quickly, before other code changes in that area of the code start to happen and before other developers have made use of the same buggy code.

Contrast this with having to fix multiple interacting bugs at the same time, with the deadline looming, while the code gets messier and messier and we all get more and more frustrated and tired.

So, building and testing the whole system frequently, from the beginning of the project, is a good idea. The problem is that most programmers (being only human) don't remember to do it. We build and test our own local version of the system easily enough: our IDE will continually check for compile errors, as we type, and it is fairly easy to get into the habit of building and running unit tests locally as we code. But how many of us will be willing to break off our chain of thought on a piece of code in order to integrate our work with that of other developers and to wait while lengthy acceptance test suites are run? Instead, the CI server takes on this task for us, only interrupting us when something has gone wrong.

Continuous integration servers typically run on separate shared machines that are set up to build and test the code as frequently as we like. We can set a timed schedule (e.g., to build every morning before our developers come in to start work) or we can ask for the build and test process to be carried out every time someone on the team pushes changes to the shared repository. This latter option is good if we can manage it, because if we have pushed code that breaks some aspect of the build and test pipeline, then we'll get an almost immediate warning of that from our continuous integration server. We'll receive an e-mail from the server, or else the server might be set up to push a notification to our team's chat channel on Slack or Discord or MS-Teams, or to some other communication system.

That is what the CI server is doing for us on COMP23311. One of the best practices we try to follow in this course unit is that of maintaining clean releases. (We obviously don't want to deploy code to the customer that has known faults.) The CI server helps with that by telling you about the problems with the code you push to GitLab within minutes of it appearing there. That means you can fix problems while they are still on your feature branches, and you can make sure that only clean code makes it through to the development branch or (most importantly) to the release tags.

For this course unit, we will use Jenkins (jenkins.io), a well-established CI server. We have set up a number of build jobs for you, that will show you the health of your team's submission for the coursework exercises so far. This brief guide will show you how to use Jenkins to access information about the build health of your team, and how to interpret the reports it provides.

21.2 Logging onto Jenkins CI Server:

You can login to the Departments Jenkins continuous integration (CI) server at ci.cs.manchester.ac.uk/jenkins. This will take you to a login screen. Enter your University username and password into the form and press `Sign in`.

You should now be taken to the main Jenkins "dashboard", showing the folder we have created to contain your build jobs for the COMP23311 team coursework this semester, which should look something like figure 21.1

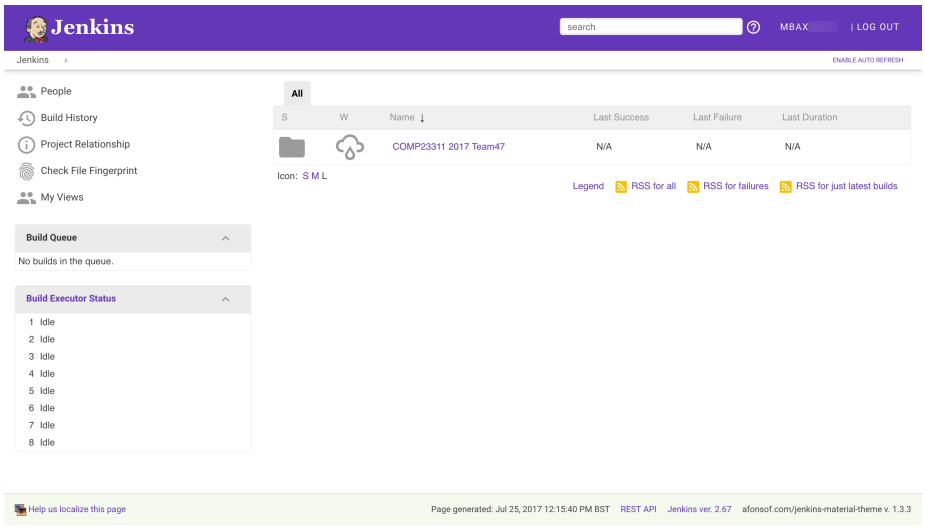

Figure 21.1: The main Jenkins dashboard

There are various features to note about this screen. On the left is a menu of

basic commands, and below this is some information about what builds Jenkins is currently running. We have set the Jenkins server up so it can run a number of builds in parallel (exactly how many varies from year to year, with the resources we have available). If more build jobs than this are requested at any time, some of the jobs will be show in the Build Queue, waiting for a build executor to become available. You will only be able to see build information for your own team's jobs, and often Jenkins will be occupied building jobs from other teams, so at busy times (for example, in the days before a coursework deadline) your team's build jobs may be stuck in the queue for anything from a few minutes up to several hours.

On the right of the dashboard, you can see a list of the top-level jobs and folders you have access to. At the moment, the only thing you can see is the folder for your work on COMP23311. The grey folder icon indicates that this is a folder and not a build job. Next to it, you can see an icon indicating the "weather" for this set of builds. That is, it shows the status of the builds within the folder over recent builds. A stormy weather icon indicates that the builds in this folder are failing, and have been for some time. In contrast, a sun icon would indicate that the builds in the folder are all succeeding. Figure 21.2 shows some of the build health icons, and their meanings.

Figure 21.2: A range of icons indicate the health of a build. From left to right: A rain icon is used to indicate that between 20% to 39% of builds were successful. A cloud icon: 40% to 59%. An overcast icon: 60% and 79% of builds were successful. A sun icon indicates that greater than 80% of builds were successful.

You can get more information about the health of an item by hovering over the weather icon in Jenkins shown in figure 21.3.

As you can see, the tool tip gives more information about the build health, and also indicates the worst performing build within the folder. This is really useful for diagnosing problems, and finding branches that need extra attention.

Team 47 doesn't have a great build health at the moment, but they still have time to improve their build "weather" before the coursework deadline. Take a look at the weather status of your own team's build folder. As you complete the coursework and your builds move from red to green, your build health icon should become sunnier.

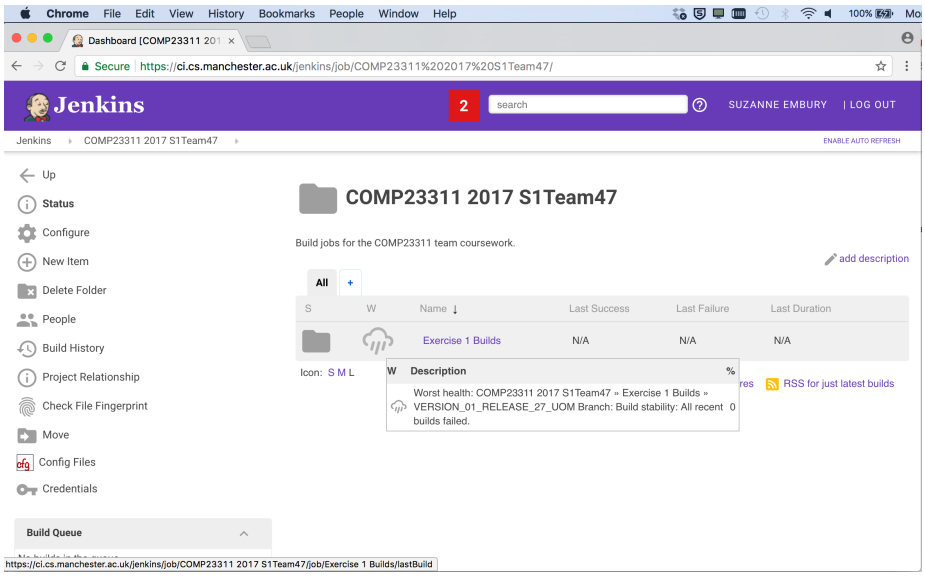

Figure 21.3: Hovering over an icon will display a tool tip that gives more information about what the icon means in terms of the build.

21.3 Accessing your Coursework Builds

Next, we're going to look at the builds that have been set up for your exercise 1 coursework.

Click on the link for your team's COMP23311 folder. This will show you a list of the builds and folders that have been created for your team for this course unit shown in figure 21.4.

At this stage in the course unit, only builds for exercise 1 have been set up. Later, more folders and jobs will appear here, as the coursework progresses.

This year, you will see an additional project in your team's folder. This job creates and stores the reports on your work created by the RoboTA automated feedback and marking system. To see how your team is doing on the coursework so far, click on this job and then selected the *Team Marks and Feedback* report.

We'll now take you through the builds we have set up for your team for team coursework exercise 1. Click on the *Exercise 1 Builds* folder link. This will take you to a Web page showing the following builds as in figure 21.5

On this page, you can see the summaries of some actual builds, instead of just folders. We have set up Jenkins projects to build the important code versions for this coursework: the development branch (`master`) and the release tag (`VERSION_01_RELEASE_27_UOM`) for the team in this screenshot—your team will use a different release tag).

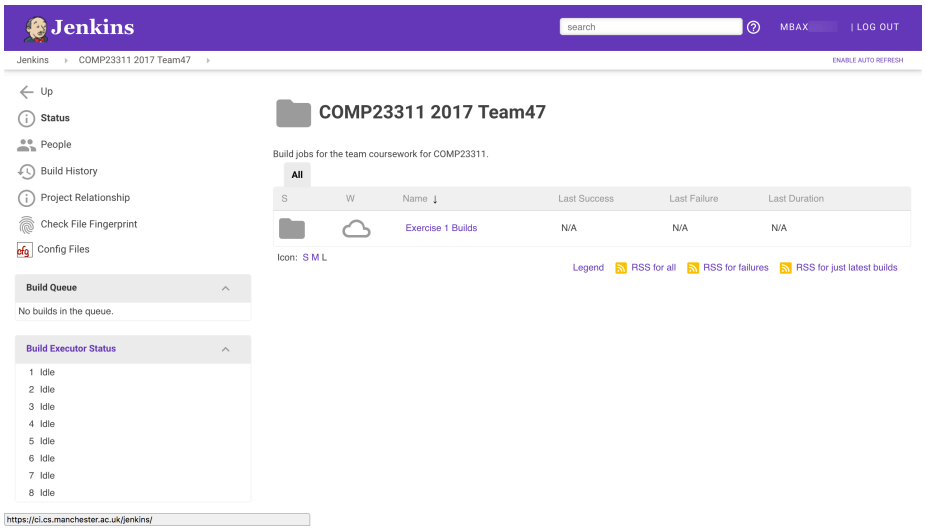

Figure 21.4: An example of how a teams coursework builds look in Jenkins

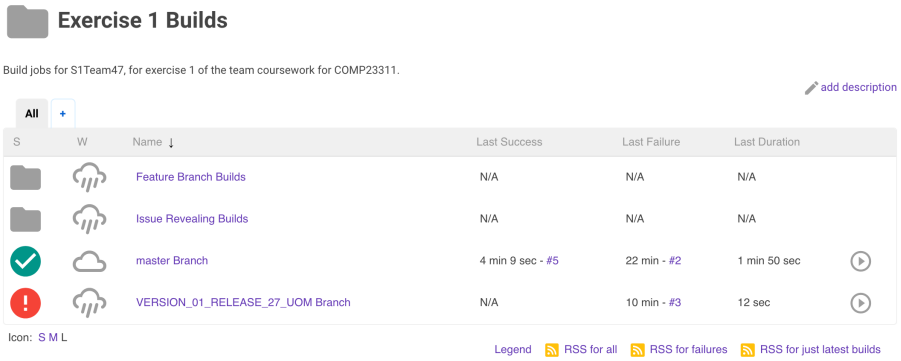

Figure 21.5: An example of the contents of the Exercise 1 Builds folder displayed in Jenkins.

Your team will also have an additional build, not shown in this screen shot, for a tag called `COMP23311_EX1_STARTING_POINT`. This shows you the build health at the start of the coursework.

Instead of a folder icon, build jobs have coloured icons indicating whether the most recent build was successful or not. The red exclamation mark beside the release branch project shows that the latest build for this tag was unsuccessful. Some error occurred during the build process that meant that executable code could not be produced. This could be caused by a compilation error in the code, but it could also mean that the tag or branch to be built doesn't yet exist in the team's GitLab repository. Builds of this kind are called *failed* builds.

The yellow icon beside the development branch build in the figure shows that the build process itself succeeded but that some of the unit tests failed. A build like this, where executable code was created and could be run, but some other later check failed, is called an *unstable* build.

In total, there are four build status icons that you might see in Jenkins, shown in figure 21.6. The goal for most of our builds is to get a green build icon beside them. This indicates a successful build: there were no compilation errors, the Ant build file was executed successfully and the unit tests all passed. This kind of build is called a *stable* build.

The final type of build icon is grey, and this simply indicates either that no builds have yet been attempted for a particular job or that the last build was aborted before it could complete. If no one in your team has pushed code to your repository since the Jenkins builds were created, then all your exercise 1 builds will be grey. This will change as soon as some code changes are pushed.

Figure 21.6: Four build status icons you might see. From left to right: 1. Grey no entry sign (**Not built**) build not yet attempted or build aborted. 2. Red exclamation mark (**Failed build**) the build process could not be completed. 3. Amber exclamation mark (**Unstable**) build succeeded but some tests failed. 4. Green tick (**Stable build**) build process succeeded and all tests passed.

To the right of the build job names, Jenkins gives some information about the previous builds. It tells us how long ago the most recent successful build was (either stable or unstable) and how long ago the most recent failed build was. It also gives a link to information about that build. Here, five builds have been

carried out for the development branch, so the link to the most recent successful build is to build `#5`, while the most recent unsuccessful build was build `#2`.

Jenkins also gives the duration of the last build. This is important on real systems, as if builds are taking too long then it could indicate a problem with the system. In addition, the length of a typical build affects how long programmers need to wait to get feedback on their pushes. Experience has shown that programmers will wait for a few minutes for information on the status of their build, but if they have to wait much longer than 10 minutes, then they will change their focus to another task, and lose the context needed to efficiently fix problems that may be revealed by the build. So it is important that build times are kept under control.

Manually Requesting a Build

The buttons in the final column allow you to manually request that a build of the job be scheduled (as opposed to waiting until someone pushes new code to your GitLab repository). Try pressing the button next to any of the build jobs now. You should see the text `Build Scheduled` appear over the button, and (shortly) the job will appear on one of the Build Executors (or in the queue, if you request the build at a busy period).

When the build completes, the summary of builds for this job should be updated. You may need to refresh your browser window to see the new information.

21.4 Drilling Down on the Development Branch Build

Next, we're going to drill down into the information Jenkins provides about individual build jobs. We'll look at the development branch build first, and will come back to look at the contents of the `Feature Branch Builds` folder later.

Click on the build job link for the `master` branch. You should be taken to a page that looks similar to figure 21.7.

Figure 21.7 is the main page for the build job we have set up for the development branch of your team's repository. It gives a summary of the most recent build status for the job, and also information about the build history. Let's take a look at the information and options provided by it.

At the top, below the purple Jenkins banner, you can see some breadcrumbs showing the parent folders for this particular job. The names are all clickable links.

Just below this, on the left, there is a menu showing the things you can do with your build job. Some of the buttons are links to further information about the build, and some are commands that you can issue. Note, for example, the

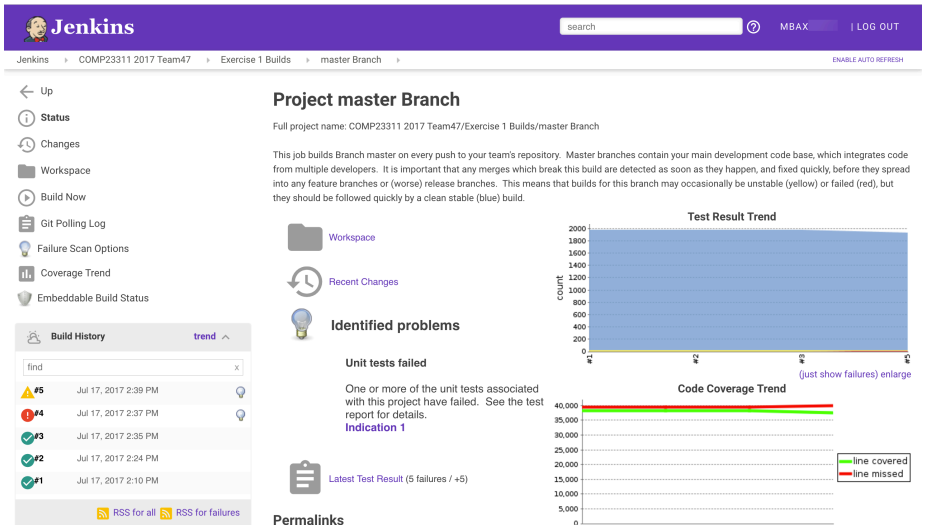

Figure 21.7: Jenkins view of the master branch

presence of the `Build Now` option, which gives another way to manually request that a build of this job be scheduled.

On the bottom left, you will see a table showing the build history for the job. Each row summarises a past build attempt. The status of the build is shown (by the coloured ball icon), followed by a link to the detailed build report and then the date and time at which the build took place. Finally, all non-stable builds should end with a light bulb icon, indicating that the cause of the build failure has been analysed, and information about it has been added to the build report. Hovering your mouse over the light bulb will give an indication of what went wrong.

On the remainder of the page is information about the status of the most recent build, including links to various reports produced during the build. The first is a link to the workspace (i.e., filespace) used for the most recent build, and the second to information about the code changes that triggered the build.

Under the section headed `Identified Problems` (with the light bulb icon) you will see a description of what caused the build to fail or to become unstable. In this case, the code was able to compile successfully. The problem is that some of the test cases have failed.

Next comes a link to a build report (indicated by the grey clipboard icon). In this case, it is a report produced by JUnit when the build was last run. The summary tells us how many tests failed in the build and (importantly) gives information about the change in test failures that occurred in the most recent build. In this case, the summary `5 Failures / +5` indicates that five tests are failing in total, and that all five failures appeared in this recent build (this build

added five failures). This is useful information as test failures that appeared in a build are likely to be caused by the changes that were pushed to the GitLab repository and that triggered the build. Test failures that first appeared in older builds will not have been caused by anything we have just done to the code.

You can click on the `Latest test result` link for information about which tests failed and why.

Two other build reports are shown on this page: the two graphs on the right hand side of the page. The one at the top shows the test results trend over the course of recent builds. The blue line shows the number of tests that pass, the yellow line shows the number of tests that have been skipped (using the `@Ignore` annotation) and the red line (the important one) shows how many tests have been failing. We can just see a slight increase in the number of failing tests for the latest build (plus a corresponding decrease in the number of passing tests).

The graph below shows the code coverage trend, produced from the JaCoCo coverage reports from recent builds. Coverage for this team has been fairly stable, with a slight drop in coverage in the most recent build, caused by the failing tests. (It is normal for coverage to drop a little when tests fail, since not all the test assertions will be executed and therefore some lines will not be covered. A significant decrease, however, could indicate that your tests are too bulky, and should be separated out into a larger number of smaller, more independent unit tests.)

You can click on any of these build reports to get more detailed information about the results of the most recent build. You can also click on individual builds in the Build History, and see copies of the reports as created at the time of the build. You can even see a copy of the console output that was produced when the build was run.

21.5 What Happens When Jenkins Builds a Job

It is all very well to have all this information at our fingertips, but it is only useful if we understand where it comes from. So, now we are going to look at what happens when Jenkins performs a build of some job.

21.5.1 What Triggers the Build?

The first question to answer is what causes Jenkins to start to perform a build for some job. We have already seen that we can manually schedule a build, using the *Build Now* command or button. But that relies on us, the developers, remembering to request that a build be done. As we saw at the beginning of this exercise, the whole point of continuous integration servers is that we don't have to remember to request a build. It should be done automatically, whether we remember or not.

Jenkins builds can be triggered automatically in a number of ways. For example, we can put a schedule in place that causes the job to be built every hour, or every 6 hours, or every 15 minutes, as best suits our needs at the time.

But probably the most popular way of triggering a build is to set up a web hook that causes Jenkins to build a job whenever someone pushes new code (or changed code) to the team repository. This is how we have set up your team repository. Whenever you push code changes to your repository, GitLab sends a notification to Jenkins about the change. Jenkins then uses this information to trigger any jobs that use the repository that was changed.

Keep an eye out for this the next time you or anyone on your team pushes code changes to your team repository. As soon as the push is made, take a look at the Jenkins dashboard and wait to see if a build is triggered.

Triggering Builds Without a Code Change

Jenkins can be set up to trigger on all kinds of GitLab actions, including commenting on an issue and creating a tag. However, when Jenkins detects that one of the triggering events has happened, the first thing it does is to check that something has changed in the code base. If there are no changes to the code, there cannot be a different build outcome, so Jenkins saves time by ignoring the triggering event.

This is sensible behaviour most of the time, but is sometimes a bit of a nuisance. For example, if we have a build job set up for a tag, we might expect Jenkins to notice when the tag is created and to rebuild the job. Jenkins can indeed notice the tag appearing, but since such a change involves no new commits, it will not trigger a rebuild. In these cases, it will be necessary to request a build manually.

You'll encounter this issue with the builds for your release tags. When you've prepared your team's release, you'll add the required release tag to the commit. But Jenkins won't build your release tag project at this stage. You'll need to manually request the build to check whether your release build has good health.

21.5.2 The Jenkins Workspace

When a job is triggered (and code changes have occurred), Jenkins uses a GitLab plugin to connect to your team's repository and to create a clone of the whole repository in the workspace set aside for the job.

It then checks out the branch that has been specified to be the focus of the build.

At this point, Jenkins has created a directory folder in its own internal space, with a copy of your team's code base that mirrors the one you have on your machine. (It obviously doesn't have any files or changes that you have not yet pushed to your team's repository.)

21.5.3 Jenkins Runs the Ant Build

Once we have a copy of the code in the workspace, Jenkins attempts to build an executable from it. It does this in exactly the same way we have practised in the workshops: it calls Ant and invokes the *dist* target in the `build.xml` file within the workspace.

This is done within a shell. Jenkins records all the outputs from running the build script in the shell, and stores this with each build, for later diagnostic purposes.

We have also set the Jenkins jobs up to run the unit tests. Once the build is complete, Jenkins will invoke the *test* Ant target, in order to cause the unit tests to be run, and the coverage information to be gathered — in exactly the same way that these things happen when you run the Ant script from within your IDE.

Digression (Not Examinable)

There is one difference in the way Jenkins runs the unit tests. The machine on which Jenkins is running is a headless server. There is no graphics console attached to it. It sits in a machine room, rather on someone's desk and is operated remotely through the terminal rather than through a graphical interface with a mouse or a trackpad. This means that any test cases that exercise the GUI of the Stendhal game system will fail, because no graphical display is configured for the machine. We get round this by running Xvfb, the X Virtual Framebuffer, before running the unit tests. This creates a virtual graphical display — or just enough of one — to allow the unit tests that exercise GUI elements to function as they would on a normal desktop machine.

21.5.4 The Results are Published

The final step in the Jenkins build process is to publish any reports created during the build process so that they can be accessed from the build web pages. This happens for the JUnit test results and the JaCoCo coverage results. If we had not configured your builds with publishing steps for both of these, then the graphs and web page test results would not appear on the build web page.

21.6 A Look at Feature Branch Builds

This almost concludes our tour around the Jenkins builds we have created for you to use with your work in exercise 1. We'll finish with a look at the contents of the `Feature Branch Builds` and the `Issue Revealing Builds` folders.

In your browser, click on the breadcrumb for the Exercise 1 Builds. Then click on the link for the `Feature Branch Builds` folder.

You should be taken to a web page containing something like figure 21.8.

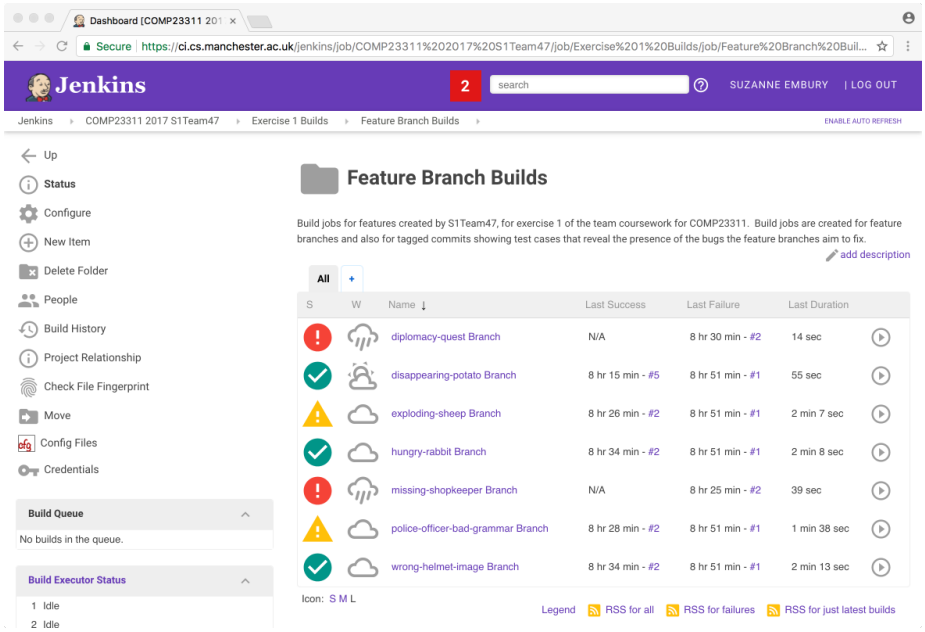

Figure 21.8: Jenkins view of the feature branch builds

As you can probably work out for yourself by now, this folder contains a whole collection of different builds. Team 47 is working on a different set of issues from your team, and therefore is working with different branch names from the ones in your team's builds. But the overall organisation should be similar for your own team. There should be one build job set up for each of the feature branches we have asked you to create. The feature branch builds should be stable because they should compile and (since by the deadline they need to contain code that fixes the bug) they should pass all the tests.

You can drill down into any of these builds to get more information about the build, and information about what caused any failed or unstable builds.

The second set of builds are in the *Issue Revealing Builds* folder. If you navigate down into that folder, you will see a set of builds looking something as in figure 21.9.

These builds are a little unusual. Their job is to run the tests on a code base consisting of the production code from the start of the exercise (i.e., the code that does not contain your fixes) *and* the test code from your feature branch tip (i.e., the code that contains the tests you wrote to reveal the issue). This means that we expect these builds to be unstable. We want to see the tests failing, so that we can see that the issue is properly exposed by them. To be fully sure you have revealed the issue correctly, you'll need to drill down into the build

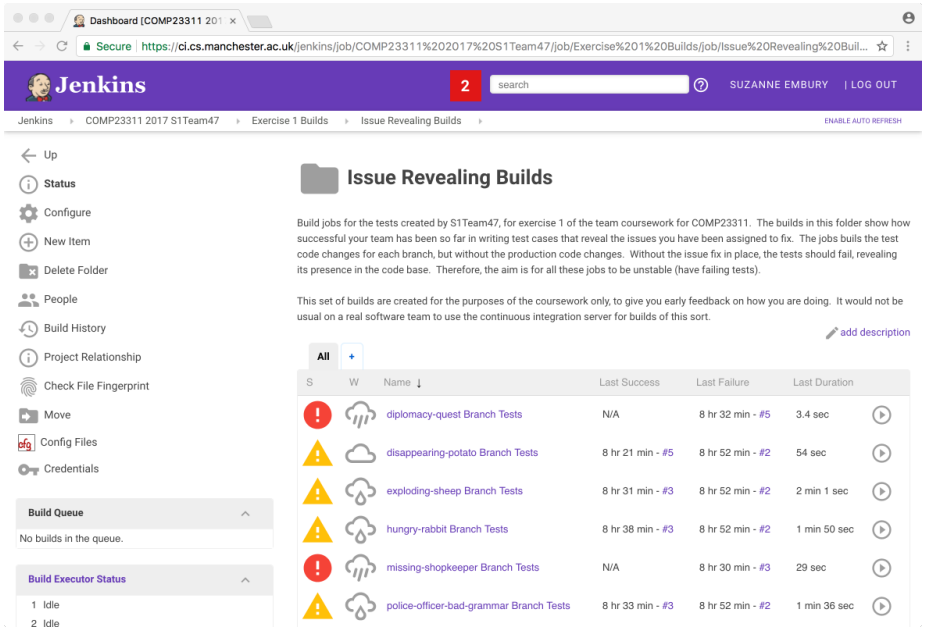

Figure 21.9: The Issue Revealing Builds folder in Jenkins

and take a look at the tests that are failing. Ideally, these should just be the tests you have written to reveal the issue. If some other test cases are failing as well (or, worse, instead of) your new test cases, then your final mark will be reduced.

Builds of this kind would not normally be created for a real software team. We have provided them to help you track your own progress towards the goals of the coursework, and to help us in marking your tests fairly and accurately.

Note that all the builds in these two folders should be either stable or unstable by the deadline. There is only one case in which a failed (red) build will be acceptable in the released repository: when you have fewer people in your team than the number of bugs assigned. Each team is expected to fix one issue per person. For example, a 5 person team will be expected to fix only 5 of the issues provided. In that case, the builds for any issue left unfixed can fail without any marks penalty (since they will fail because the required branch and tag will not be present in your team's repository).

21.7 Confused? Stuck? Need Help?

If any aspect of the Jenkins builds we have created for you are unclear, please come and get help from a TA or a member of staff during the team study

sessions. If you meet the workshop attendance requirements, you are also free to e-mail Suzanne with questions outside the team study sessions.

The School's CI server is administered by Chris Page. Technical faults and outages should be reported to him through support.cs.manchester.ac.uk.

Bugs or feature requests for the RoboTA system should be reported through the project issue tracker at gitlab.cs.man.ac.uk/institute-of-coding/robota-issue-tracker

Chapter 22

Code review

In addition to the time spent debugging code (see chapter 3), software engineers spend lots of time reviewing other people's code. Its a crucial part of building better quality software, see figure 22.1. Code review is a crucial skill to learn, both as a reviewer and a reviewee. This chapter introduces key concepts in code review that you can use on this course.

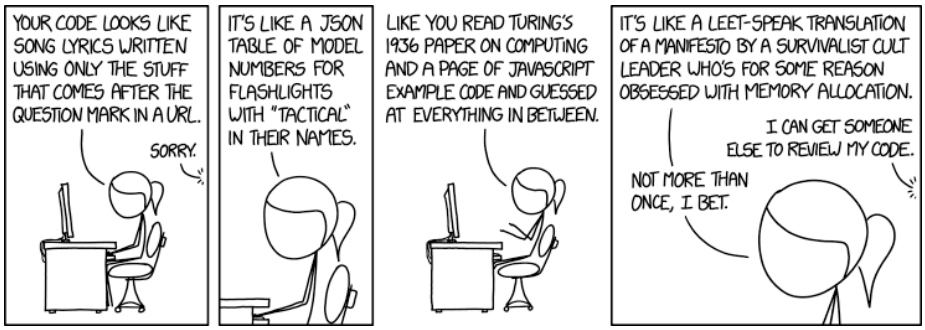

Figure 22.1: Does your code look like song lyrics written using only the stuff that comes after the question mark in a URL? Reviewing other people's code can be hard work but probably not as hard as having other people review *your* code. However, code review is fundamental to building quality software. Code Quality (xkcd.com/1833) by Randall Munroe is licensed under CC BY-NC 2.5

22.1 Why Review Code?

Code review is the process by which program code written by one person (or group of people) is inspected by another person (or group of people), to find

errors and infelicities. It is one of the primary tools used today to manage the quality of an organisation's code base. Code review comes in lots of different flavours (we will list the main ones later in the document) but the core idea, common to them all, is a very basic one: it is easier to spot problems in code written by someone else than in your own code.

It is easy to see why this might be the case. When we have just written some code, the idea of what the code should be doing is fresh in our minds. It is hard to see the discrepancies between our mental model of what the code should be doing and what the code we have written is actually doing. But when we read code written by someone else, we do not have so many preconditions and assumptions that get in the way of understanding what has been written, as opposed to what was intended.

This applies to general code quality issues as well as bugs. A code reviewer can spot when we have failed to follow the naming and layout standards in use within the code, or when we have used comments in a way that does not follow the conventions of the rest of the code base. It is hard to keep track of all these things, especially when new to a team, as well as making the code do what it should. A code review can point out problems before they leave our feature branch, so that only good quality code reaches our development branch.

We saw earlier in this course unit that there are major advantages to finding bugs earlier after they are introduced rather than later, with costs rising especially dramatically when bugs make it through to code that is used by the customer[1]. While automated testing (unit testing, etc.) can go a long way towards finding defects before they reach the customer, it is not by itself a complete solution to the problem. Testing can only find defects that we have thought to check for. And testing cannot help us weed out poor quality code that makes future bugs harder to find and fix.

It turns out that code review is an excellent complement to testing. Studies have shown that code review can find up to 60% of defects (McConnell, 2014), with unit testing only finding 25%. The two techniques work well in partnership: automated testing is relatively cheap to run, and can be run repeatedly without needing (much) human intervention. Code review is more expensive, and requires human effort for each time it is performed, but can find a wider range of defects. A good workflow therefore is to make sure that the bugs that are detected by code review are converted into test cases, so that they can be detected cheaply in future versions of the code, and the valuable code review effort can be put towards finding new defects not currently covered by the code base. Of course, for maximum efficiency, this requires that code review is only ever performed on code that already passes the test suite.

Code review has other advantages as well, in helping to homogenise and improve coding styles across teams, and to spread knowledge of the code base more evenly

[1] Refer back to Boehm's Cost of Quality model, presented in lectures in week 1 and chapter 4 details.

throughout the team. If all code is reviewed by at least one person, then the days when parts of the code are untouchable by anyone except the lone expert who created them are gone and the team's truck factor is increased[2]. And it is human nature to code more carefully and correctly when we know that one of our colleagues will be looking over any code that we push to the team repository.

It is important to be aware of the costs of code review, as well, however. Code review takes time. Typical code review rates are between 100 and 200 lines of code per hour, for experienced professionals (Kemerer and Paulk, 2009; Bisant and Lyle, 1989). New team members will be slower than this. In a typical team, one can expect to spend between 1 and 5 hours per week reviewing code (more, perhaps, as a release deadline approaches). Time spent reviewing is time not spent coding, and it can sometimes be hard to justify spending the time when deadlines are looming. But, it is exactly when the team is under pressure that code reviews are most needed. Any errors that slip through at this stage will only come back in a more expensive form, when the customer feels their bite.

22.2 Types of Code Review

There have been many proposals for different ways of doing code reviews, ranging from the very simple and informal to heavyweight and expensive monitoring programmes. Here, we mention a few of the key types, to give you a feel for the different ways in which code review has been implemented in practice.

22.2.1 Buddy Review

Starting with small and simple, there are several informal kinds of code review. These normally come under the heading of "buddy review". As the name suggests, this kind of review is done informally, between coders with more or less equal status within the team, on an as-needed basis. This could be as simple as asking someone else in your team to look over a particularly tricky piece of code before you commit it ("over the shoulder" review). Or, in some teams, developers have an assigned "buddy" who they talk code through with, when it is ready to be pushed.

In teams that use the agile practice of pair programming, code review will be happening all the time, as the pair takes it in turns to act in the "driver" and "navigator" roles. The navigator looks over the code written by the driver,

[2]The *truck/bus factor* of a team is the number of its members who would need to be run over by a truck/bus for the team to be unable to fulfil its function in some significant way. A team where only one person understands and can safely change the code that interfaces with the database, for example, has a truck factor of 1, and the team should consider itself to be at risk. What do you think the truck factor of your COMP23311 team is? If it is low, what steps might you take to increase the truck factor for later exercises?

performing what is essentially a code review function, on a continuous basis with a very short feedback time.

22.2.2 Team-Based Review

Many teams have more formal structures for reviewing code, to ensure that quality is managed evenly across the team, regardless of individual team members' preferences for or against code review.

The rise of source code repositories like GitHub and GitLab, and the coding workflows that have developed around them give a perfect framework into which to insert code review into the normal day-to-day work of the team. For example, many teams will require the code in a feature branch to be reviewed by another team member before it can be merged with the development branch. This can even be enforced by the use of tools such as Gerrit, which allows code to be held in a "staging area" for code review, and which blocks the integration of code into the main development branch until an authorised user has agreed that it meets the team's quality standards.

However, other forms of team-based code review are possible, and have been in use for many years. One common team-based review technique is the "walkthrough". This involves a meeting of affected team members (typically more than 2) in which one team member gives a verbal presentation of some artefact, taking the rest of the team through the way it works and what it is intended to achieve. Walkthroughs can take place early in a development, to sanity check a planned design, for example, or later in a release cycle, to check that correctness of the implementation of some key algorithm or section of the code, by working through it together line-by-line.

22.2.3 Formal Review

The most formal type of code review involves the work of a team being inspected by an external team. This kind of review is usually only performed in large organisations with very tightly-defined processes for managing software quality across the organisation. They sometimes go by the name of "inspections", "formal technical reviews" and "formal management reviews". A formal technical review will involve the work of the team being assessed by an external team of technical experts. Formal management review, on the other hand, is an assessment of the quality of a team's processes, and will normally involve the work of the team being inspected by more senior (possibly non-technical) staff within the organisation.

These reviews are often linked to the long term future of a team or project. Continued funding may be dependent on successfully passing through a series of formal review processes.

Unsurprisingly, the most formal type of code review is also the most expensive, with extensive documentation and presentations having to be prepared in advance, as well as hotel and meeting rooms needing to be booked for the participants, sometimes for 2 or 3 days.

Now that teams are taking on responsibility for their own code reviews, using more informal techniques, these large scale formal reviews are less common. (There were always doubts about their cost-effectiveness, and the effects on staff of these stressful reviews was often seen as being counter-productive in the long term.) While funding reviews and presentations to senior management or customers are still a normal part of life as a software engineer, code quality is typically managed through more informal, team-based routes in modern organisations.

22.3 Good Practice for Code Reviewers

If code review is to be an effective means of discovering and removing defects from software, then it is important that everyone involved (reviewers and reviewees) see the process as a positive one, rather than as a chore to be endured. It is important that criticisms are worded constructively, and that questions of blame are regarded as being of less importance than finding and fixing the errors. Code reviewers should be seen as allies against a common enemy (bugs that reach the customer), but at the same time few people enjoy having their flaws publicly pointed out.

It is therefore important that code reviewers adopt a neutral tone, and balance negative points out with positive comments about aspects of the code that are done well. This is particularly important when performing code review in a new team. Later, when trust and mutual respect have been built up, code reviews can be less carefully worded. But in the early days, it is important to think about how comments can be worded, to ensure that they will be received in a positive way. But regardless of the diplomacy with which the reviewer carries out their work, for the reviewee, code reviews are a powerful mechanism for learning to be challenged, to cope with constructive criticism and to defend your own ideas where you feel the reviewer has overlooked something of importance.

The learning effect of code reviews can be helped by making all reviews accessible to the whole team (as they are in a GitLab repository). That way, the code reviewer is held accountable for their reviews, just as the coder is being held accountable for their code.

Another technique that can help is the practice of “promiscuous reviewing”, in which all team members review the work of all other team members, over time, rather than sticking with rigid and unchanging buddy review pairings. This avoids the risk of “revenge reviews”, and helps to foster an “all-in-it-together” team mentality. The person reviewing your code will be conscious that, not too

far in the future, the roles will be reversed. This can help to keep the review process constructive and considerate.

When performing a code review, the following aspects of the code can be commented on:

- Possible defects: for example, pointing out an off-by-one error in a loop, or improper handling of negative numbers.
- Possible flaws in testing: for example, a missing case that is not covered by tests, or pointing out when two tests seem to be testing the same thing (i.e., redundant tests).
- Design issues: for example, pointing out that a class should be split into two classes, or recommending that a method be moved to a different class.
- Naming issues: for example, noting where a class name is drifting out of line with the changing function of the class, or where legacy variable names need to be updated to follow the team's current practice.
- Coding style issues: for example, pointing out inconsistent code layout, or where naming of constants is inconsistent with constants elsewhere in the code.
- Presence of code or test smells: for example, methods that are too long, or test methods containing two separate tests bundled together.
- Use of Git: for example, noting when a commit message is insufficiently clear or when commits should be squashed together to make a more meaningful representation of the change being made.
- Comments on the Build: for example, noting which platforms a bug-fix is suitable for, or pointing out where a change will require changes in the build process or other configuration elements.
- Effect on Team Metrics: for example, noting where code coverage is negatively or positively affected by a change, or any additional technical debt that the change incurs.
- Examples of good practice: for example, where an elegant solution has been found to a problem, or where code is clear and readable without requiring any comments.

We can see from this list that code review can address more than just defects and errors, covering also code quality, enforcement of conventions, performance issues and code readability aspects. Code reviewers should try not to comment on matters of preference. For example, if the team has not agreed to on a convention for white space in code, reviewers should not criticise team members for using spaces when their own preference is for using tabs, provided the look of the code is the same for both options. The line between preference and best practice is not always clearly defined, so some degree of trial and error is in order. If you find you are continually pointing out that someone is placing their curly brackets differently from the rest of the team, and that person continually ignores your advice, it is probably time to move on to find bigger fish to fry.

To give an indication of the kinds of conversation that can productively take place using code review, figures 22.2, 22.3, 22.4 and 22.5 show examples of commit comments from the NodeJS project from github.com/nodejs/node.

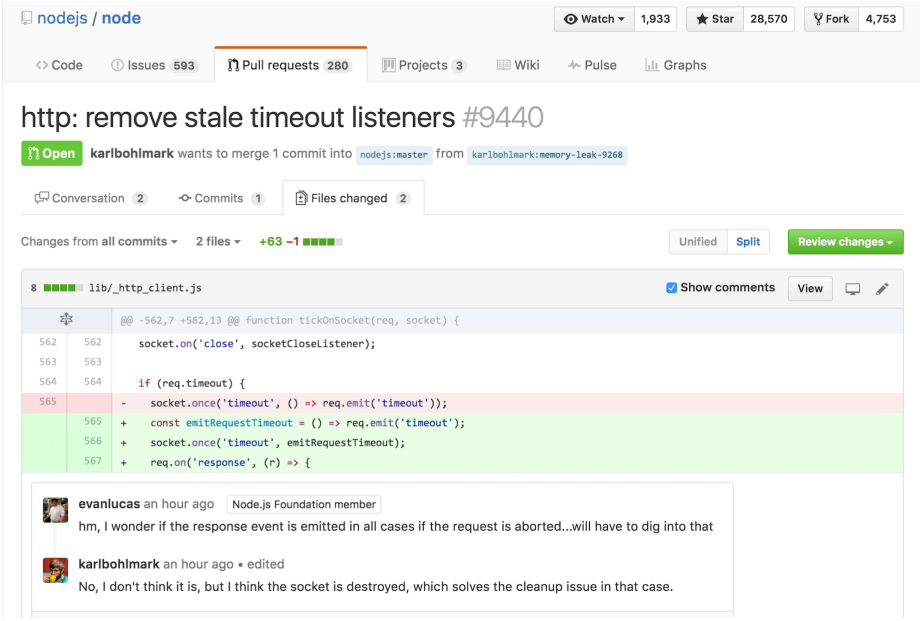

Figure 22.2: Sample conversation from nodejs

22.4 Code Review Facilities in GitLab

GitLab, like GitHub, provides facilities for commenting on merge/pull requests, and on individual commits.

When examining a merge request, GitLab allows comments to be placed on the code changes included in the proposed merge. The merge request is assigned to the person who will review the code. After examining the changes, the reviewer can decide to accept the merge request, or to request further changes. Once the changes are made, the developer can request a second round of review. In this way, the code reviewer acts as a gate keeper, preventing poor quality code from reaching the development branch.

Or, comments can be added directly to individual commits. Comments on the commit as a whole can be added through the dialogue box at the end of the commit. Or, comments can be added at specific lines. To do this, hover your mouse over the line in the commit where you wish to add the comment. A small speech bubble will appear to the left hand side of the line. Click on this to bring

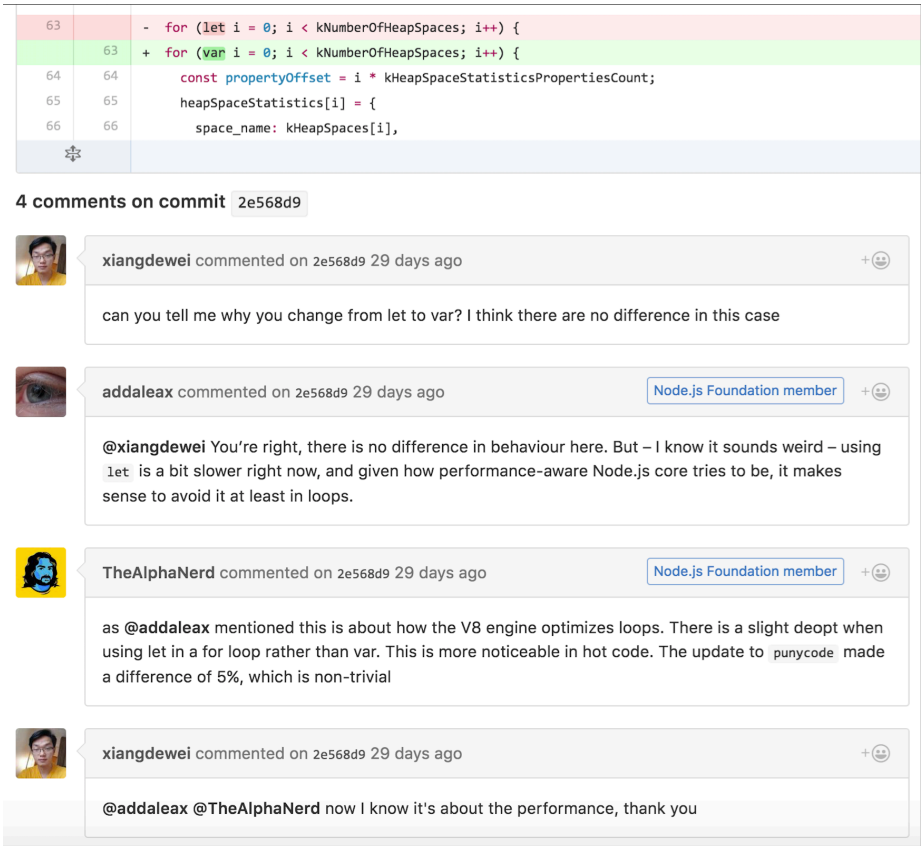

Figure 22.3: Sample conversation from nodejs

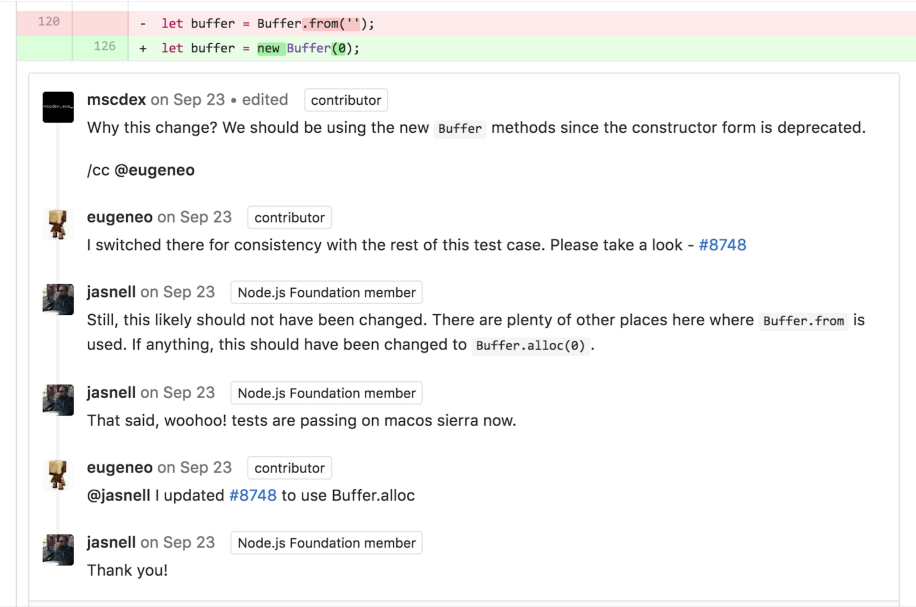

Figure 22.4: Sample conversation from nodejs

up a comment box, shown in figure 22.7 similar to those shown in the NodeJS examples given in the previous section.

22.5 Code Review in COMP23311

For exercises 2 and 3 of the COMP23311 team coursework, you are asked to practice an informal team-based code review system. You should ensure that the code for every feature you add to the Stendhal code base, or change, is reviewed by a separate team member. This includes both test code and production code changes.

You should use GitLab's comment facility to review your team's work. If a feature is short enough to review in one go, and is produced in plenty of time, you can add your code review to the merge request for the feature branch.

If you are attempting a larger feature, or a smaller feature is taking a long time to implement, you may want to give interim code reviews, on partial versions of the feature or its test code. For this, you can use GitLab's comment facility on individual commits. This means you can review any commits that have been pushed to your team's repository, even if the branch they are on has not been merged into the main development branch yet. In this case, you should check with the coder whether they are happy to have the commits reviewed, as it can

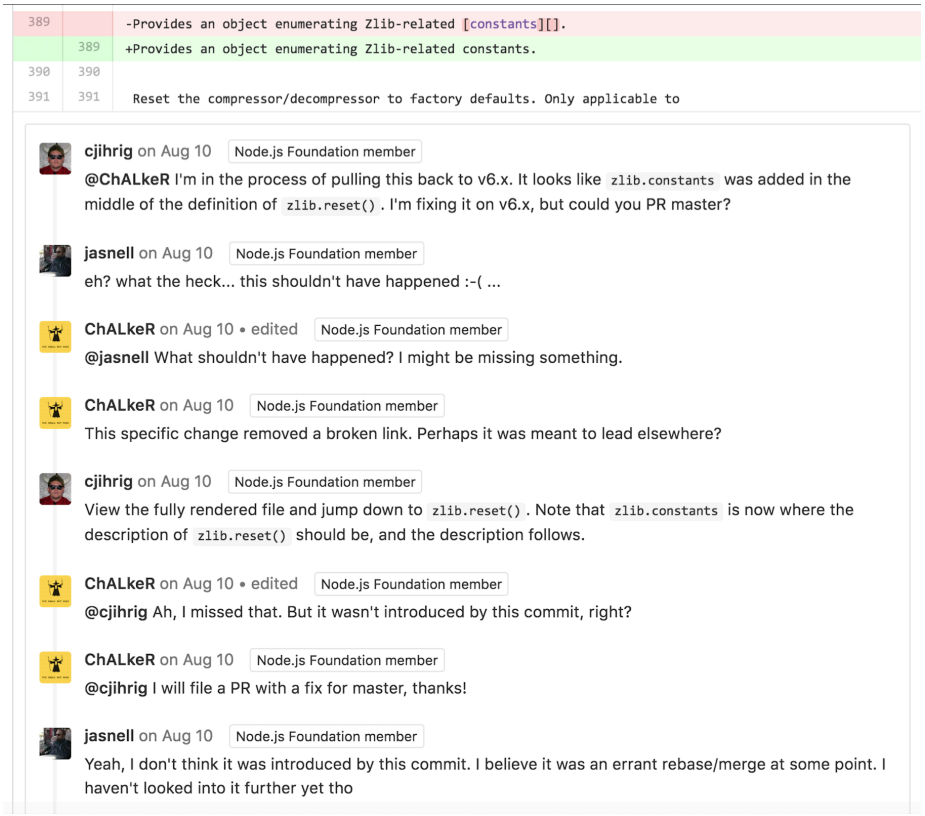

Figure 22.5: Sample conversation from nodejs

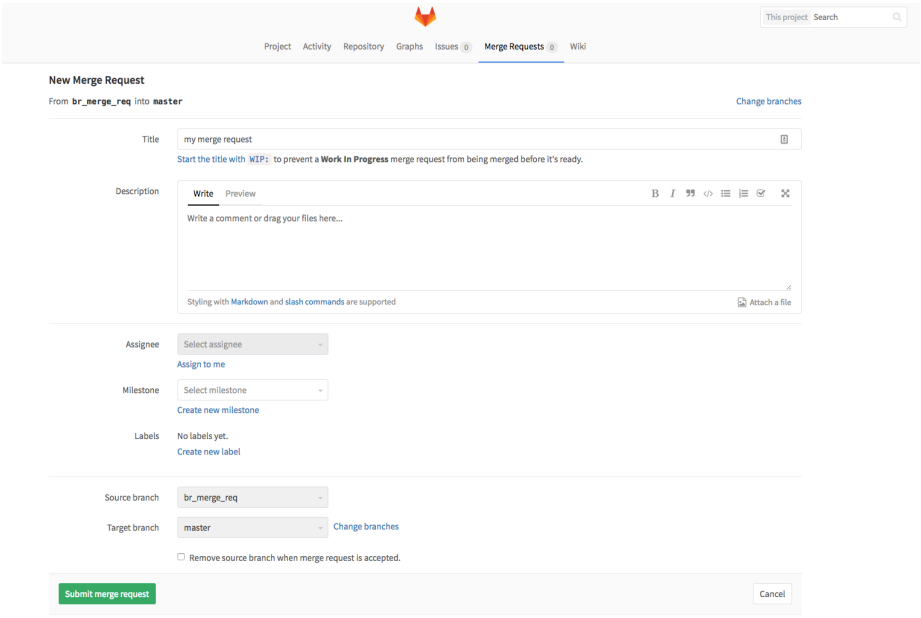

Figure 22.6: Sample conversation from nodejs

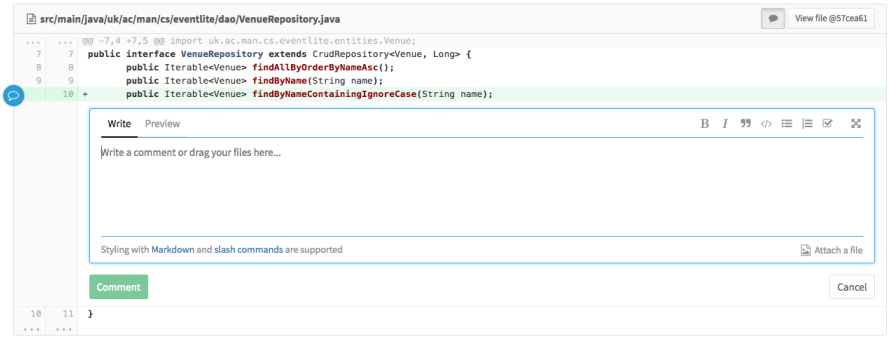

Figure 22.7: Sample conversation from nodejs

be quite annoying to be told about problems you are aware of, and are in the middle of fixing.

It is up to you how you organise your team to complete the reviews provided that:

- All code changes made for your new features are reviewed.
- All contributing team members carry out at least one code review.

Code review is very widely used in the software development industry today. You can expect to be subjected to code reviews as soon as you start to write code that becomes visible in your team's repository. Many companies also expect all software engineers, no matter how junior, to review the code of others. As well as helping to keep the quality of your team's code base high, code review is also a great way to learn the conventions and standards used by your team. Look at code reviews provided by your colleagues to learn what is considered good and bad practice, and to get a feel for the degree and style of review comments your team members expect.

In COMP23311, we ask you to carry out some basic buddy reviewing. Through the coursework, you can practice commenting on others code, and responding to comments on your own. This will be useful interviewing material, and will also prepare you to join a software team using modern software quality management techniques.

You can also ask your industry mentor about code reviewing practices used in their organisation, and how they help the work of the team.

Good luck!

Chapter 23

Unit testing

> "Never in the field of software development was so much owed by so many to so few lines of code"
>
> — Martin Fowler on JUnit.org

23.1 Introduction

In COMP23311, we are going to make use of an industry strength toolkit for software engineering. This document introduces you to a part of that toolkit that we'll be making use of right from the beginning of the course unit: JUnit. JUnit is a testing harness for Java that allows us to write concise and readable automated tests for Java code. It also provides facilities for executing tests and reporting on the results.

Those of you who took COMP16412 (Programming 2) last year will have encountered JUnit while learning to code in Java. For that course unit, JUnit test suites were provided for you to use, but not much was said about how to interpret them or how to write them. For others, JUnit will be completely new.

Either way, by the end of COMP23311, you will have written your own JUnit tests—possible quite a lot of them—and you will have experienced the benefits of coding with the support of a large (ish) automated test suite. We'll be spending quite a lot of the workshops and the coursework talking about and developing these ideas. For now, this short document introduces you to the basic concepts you need to get started.

23.2 What is Automated Testing and Why Do We Need It?

Testing a piece of software is the process of running it to determine how closely its actual behaviour matches the requirements set for it. This means deciding on a selection of *input values* the code will be run with, and working out in advance of running (and sometimes even writing) the code what the *expected output* should be for each input if the code is behaving as we wish it to. When we run the code with the selected inputs, we check the *actual output* produced by the code, and compare it to the expected output. If the actual output matches the expected output, then we say that the test *passes*. If it differs in some way, then we say that the test *fails*[1].

A failing test is evidence that the software we are building does not correctly implement the behaviour we require of it. It tells us that we have more coding work to do before we are done, and gives us some information about what that work is.

By contrast, we can't learn much from the fact that an individual test passes, since bugs may still exist in parts of the code not covered by the test. But if we have a comprehensive test suite, covering all the key cases, then we can start to have some confidence that we might have implemented it correctly once all the tests in the suite pass.

The most basic approach to testing software is manual testing. In manual testing, a person operates the software, entering the selected input values and painstakingly observed the results the software outputs, to check whether it was what was expected or not. In the early days of software engineering, all testing was done like this. Humans are flexible and creative, but they are also slow and unreliable. But thorough manual testing requires a lot of effort and is very boring and repetitive. It is easy for a human tester to miss out key cases, to mistype a selected input or to misread an output.

Computers, on the other hand, are excellent at repeating the same action over and over again, and they can do this very quickly and with perfect reliability. In theory, they ought to be much better at systematic testing than humans, and in fact this turns out to be largely (though not completely) the case. Software testers started to write scripts to automate the process of running the software with the selected inputs, so the human tester only had to eyeball the output and see whether it matched what was expected. These scripts save a lot of time, and help manual testers to be more consistent and thorough in the test cases they check. But, they are not fully automated tests, since they do not check for themselves whether the actual output matches the expected output. The human testers still had to do this work for themselves.

[1] Note the binary outcome here. A test either passes or fails. It is important to stick to this, and not to allow yourself to think of tests as "partially passing" or "nearly passing". These halfway house concepts aren't helpful to us in achieving high code quality.

It turns out that computers can do this part of the testing process for us too, and can do it much faster and more precisely than manual testers could hope to. By fully automating our testing, we get a test suite that takes a little bit more effort to set up in the first place, but which we can run many times over, very cheaply. This simple idea has revolutionised the way we develop software over the course of the last two decades.

Let's look at one way in which an automated test suite can save us time when coding. Suppose we have a comprehensive, semi-automated test suite for some code we are about to write, perhaps in the form of a collection of scripts. Running the scripts and checking the results takes a good 15 minutes of concentrated effort, and so we normally only run it a couple of times a day, sometimes only running the suite at the very end of a day of coding. One afternoon, when we run the tests, we notice that some of the tests that used to pass now fail. Something we have added to the code that day has broken functionality that we thought was working.

This is called "regression", since the behaviour of the software system has "regressed" from the requirements which were previously met. When we cause a regression, it should (ideally) be fixed before we try to add more functionality or fix other bugs. Often, regressions are caused by changes we have made recently, so the starting point is to look through the 50 or so new lines we added that day and the 100 lines of code we changed to find the source of the regression, and fix it. We might also need to look through all the lines of code that the lines we have changed or added interact with. That is going to take some time!

Imagine instead that we have a fully automated test suite that takes just seconds to run, and which works out which tests have failed for us. Instead of running this suite just once or twice each day, we run the test suite after making every small code change. Now, when we notice a newly failing test, we only have to look at the last 5 or 6 lines of code that we changed since we last ran the tests (and the related lines of code) to find the source of the problem. This reduces the scope of the debugging task, and makes bugs much cheaper and simpler to find. We also find bugs earlier, when they are easier to correct because our attention is already focused on the area of code they are hidden in. We never end up in the situation where we have a large body of code with several (many!) bugs all mixed up together, requiring a marathon debugging session of goodness-knows-how-long to fix.

The cost savings from the frequent use of a comprehensive automated test suite can be significant—so much so that many organisations now make use of continuous integration and test systems, which automatically build and test each new piece of code that is checked into the version control system, reporting back to the developers if and when problems are discovered. You'll get a chance to work with such a system later in this course unit.

23.3 Automated Testing in JUnit: a Simple Example

The most commonly used testing harness for Java code is JUnit and that is the main testing tool you will learn to use in this course unit. The principle behind JUnit is very simple: an automated test case in JUnit is merely a Java method (the test method) that invokes another Java method (the code under test) with some selected inputs, and compares the output against the expected result. If the actual output matches what is expected, then the test has passed and JUnit exits quietly. But if there is a discrepancy, then JUnit reports the test as having failed, and gives the programmer some information about the differences in output it observed.

We'll introduce these ideas by looking at how we would use JUnit to test a very simple Java method. In the code listing below, you'll find the code for some JUnit tests for a method that calculates the largest square that is less than or equal to its parameter. We'll go through this test class line by line.

```
import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.core.Is.is;

import org.junit.Test;

public class LargestSquareTest {

    @Test
    public void shouldReturn0AsLargestSquareLessThanOrEqualTo0() {
        assertThat(LargestSquare.lessThanOrEq(0), is(0));
    }

    @Test
    public void shouldReturn1AsLargestSquareLessThanOrEqualTo1() {
        assertThat(LargestSquare.lessThanOrEq(1), is(1));
    }

    @Test
    public void shouldReturn1AsLargestSquareLessThanOrEqualTo3() {
        assertThat(LargestSquare.lessThanOrEq(3), is(1));
    }

    @Test
    public void shouldReturn4AsLargestSquareLessThanOrEqualTo4() {
        assertThat(LargestSquare.lessThanOrEq(4), is(4));
    }
}
```

After some import statements to pull in the JUnit classes and static methods we need, you'll see what should be a definition for a class called `LargestSquareTest` with `public class LargestSquareTest`. JUnit classes are ordinary Java classes, defined in the usual way. This JUnit class is going to contain tests for a solution class called `LargestSquare`, so we will call it `LargestSquareTest`. In fact, the class could have any legal name. But, here, we're following a common convention that JUnit test classes are named after the class they test, with the word `Test` stuck on the end. This is useful because, as a reader of code, we can see instantly which classes are test classes and which not, and also which class is being tested by which test class.

Inside the class, there are four method definitions. Each of these methods describes a separate test case for the class under test. These too are ordinary Java instance methods, of the kind you have met before, with the exception of the fact that they are each annotated with `@Test`. You have not encountered annotations in COMP16121/212, but they are very simple to understand. They allow us to annotate code with information that is useful to the compiler and other language processors, but which will be ignored during ordinary execution. In this case, the purpose of the annotations is purely to tell the JUnit test runner which of the methods defined on the class should be executed as test cases, and which should not.

There are a couple of other JUnit annotations that we'll encounter later in the course. For now, the important thing to note is that we must put the `@Test` annotation at the start of every test case method we write. If we don't, the test case described by the un-annotated method won't get executed when we run the test suite.

Now let's look at the test methods themselves. Every JUnit test method should be public (to allow the JUnit runner to call it) and should have a void return type. JUnit test methods must have no input parameters. They can be called any legal Java method name, but (just as with JUnit class names) it is usual to follow some naming conventions. Some people (for historical reasons) begin every test method with the word "test". I prefer to follow the convention of beginning the test names with the word "should", and making the name describe the behaviour that the test is testing. The idea is that the names of the tests, when viewed in isolation (as is possible in some IDEs) should read like a specification for the code under test.

You might be a bit surprised by how long these method names are. You may even be wondering if such long method names can possibly be best practice. It's true that these names would be too long and cumbersome for ordinary code. But we are not writing ordinary code here. JUnit methods are called by the JUnit runner, which uses reflection to identify the methods tagged with the `@Test` annotation, and then run them. No human will ever have to write code which calls these JUnit methods. No human will ever have to type these long names in. So, the only role they play is one of documentation; the method names should tell us what the intent of the test case is. That usually means

writing quite a long method name, but it is also useful, as it forces us to think what the test we are going to write is for, before we get into the nitty gritty of coding it up.

Next, we'll look at what is happening inside the test case methods themselves. Let's focus on the example `assertThat(LargestSquare.lessThanOrEq(0), is(0))`. Here, we are calling the code under test (a static method on the `LargestSquare` class called `lessThanOrEq()` with a specific input value (in this case, `0`). Then, we are using a method provided by the test harness `assertThat()` to state what we expect the result of this to be, when the method is called with the specified input value. In this case, we expect the output to be `0`.

`assertThat()` is a matcher method that is provided as part of the Hamcrest matching library. It comes in several forms (some of which are quite complicated), but all you need to understand at the moment is that it takes the value given as its first parameter, and matches it with the expression in its second. If the values match, then the assertion exits quietly. If there is a mismatch then the assertion flags this up as a failing test, along with some information about the exact form of the mismatch detected. The assertions in the test class in the code listing above all use the pre-provided matcher `is(value)`, that checks whether the provided value is equal to `value` or not. So, the `assertThat()` statement in the code listing checks that `LargestSquare.lessThanOrEq(0)` returns a value that is equal to 0, and lets the programmer know about it if it doesn't.

Notice how the whole test case reads quite like an English sentence describing how we want the code to behave. We want to "assert that the largest square less than or equal to 0 is 0". Well-written test cases should have this property. They should (as far as practicable) read like natural, readable statements of the behaviour we are trying to implement.

There is a lot more to learn about JUnit than the very simple tests we have described so far. But, this brief introduction should be enough to help you get started on working with tests in the workshops and coursework over the next few weeks.

Chapter 24

References

As well as the recommended reading from section 0.3, this section lists everything we've cited. Links to `librarysearch.manchester.ac.uk` take you to electronic copies (login required).

Since you are reading the pdf version, all the references can be found in the bibliography section following this page.

Bibliography

Bisant, D. and Lyle, J. (1989). A two-person inspection method to improve prog ramming productivity. *IEEE Transactions on Software Engineering*, 15(10):1294–1304.

Chacon, S. and Straub, B. (2014). *Pro Git*. APress.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Hull, D. (2021). Experiencing your future: open source experience. In *Coding Your Future: A Guidebook for Students*. University of Manchester, github.com.

Hunt, A. and Thomas, D. (2004). *The pragmatic programmer : from journeyman to master*. Addison-Wesley.

Kemerer, C. and Paulk, M. (2009). The impact of design and code reviews on software quality: An empirical study based on PSP data. *IEEE Transactions on Software Engineering*, 35(4):534–550.

Koskela, L. (2013). *Effective unit testing : a guide for Java developers*. Manning.

Martin, R. C. (2011). *The Clean Coder: A Code of Conduct for Professional Programmers*. Prentice Hall.

Martin, R. C. and Feathers, M. C. (2009). *Clean code : a handbook of agile software craftsmanship*. Prentice Hall.

McConnell, S. (2014). *Code Complete*. Microsoft Press.

Spinellis, D. (2021). Why computing students should contribute to open source software projects. *Communications of the ACM*, 64(7):36–38.

Stephens, R. (2015). *Beginning Software Engineering*. Wrox programmer to programmer. Wrox, 1st edition.