Workshop

Building and Testing Marauroa

COMP23311- Software Engineering I

University of Manchester

School of Computer Science

2018/2019

written by

Suzanne M. Embury

Contents

1 Introduction

In this workshop, 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.**

Acknowledgements

We are grateful for the help of Martina Catizone and Liam Pringle, who helped us develop the material for this workshop.

2 Acquiring a Local Copy of the Marauroa Project

2.1 Run the IDE You Will Use for this Activity

The School 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 Oxygen*. 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 Eclipse Oxygen.

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

```
eclipse-oxygen &
```

Note that running just the command ”eclipse” will cause Eclipse Kepler to be run. This is an old version that includes a number of large plug-ins needed by another course unit in the past. It is extremely unstable, and will not work for this exercise.

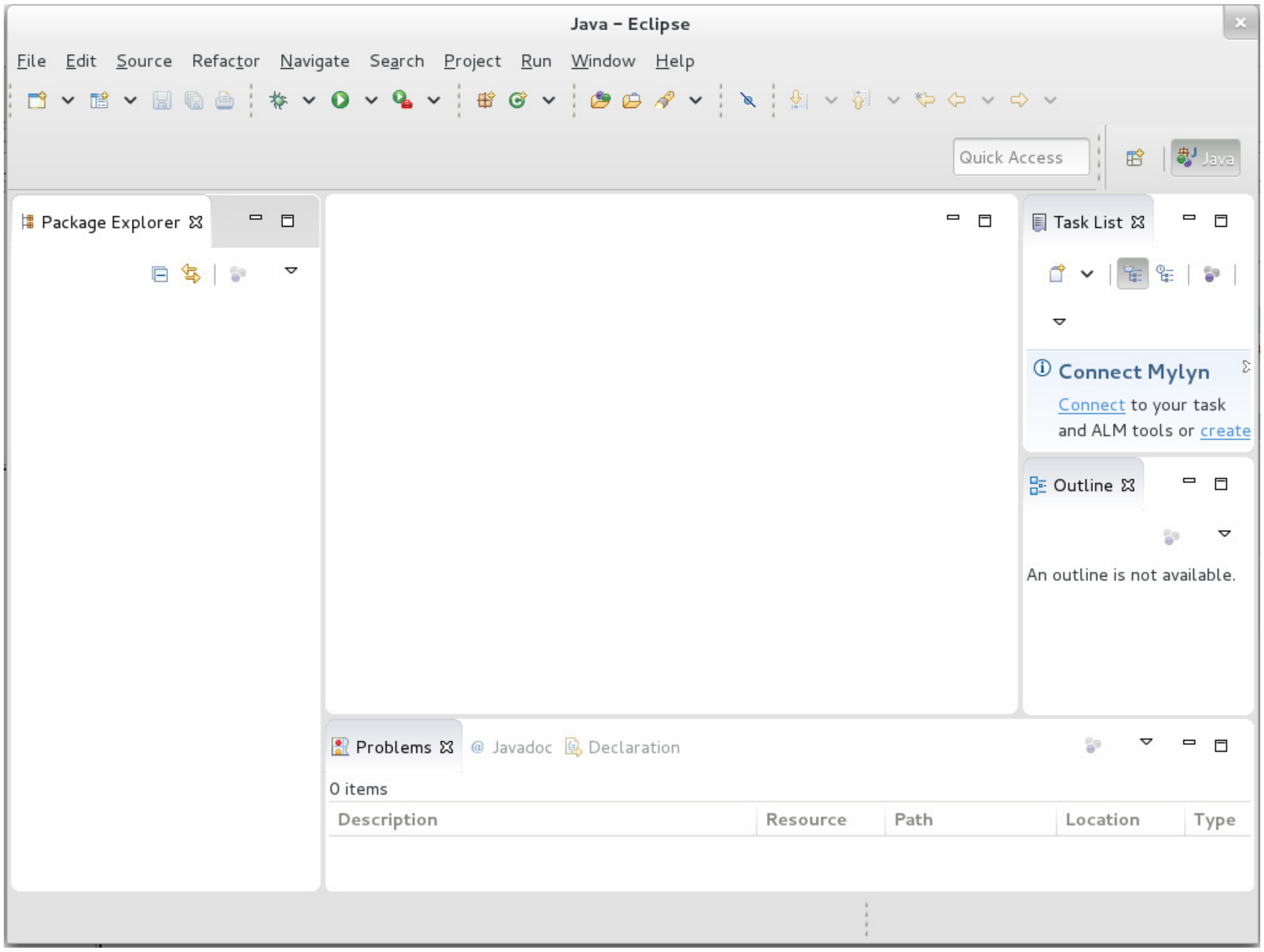
2.2 Select the Workspace You Will Use

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.)

2.3 Organise Your Main Eclipse Workspace Window

You should now see a window that looks something like this:



If you used the same workspace you created for the GitLab Access Check activity, 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.

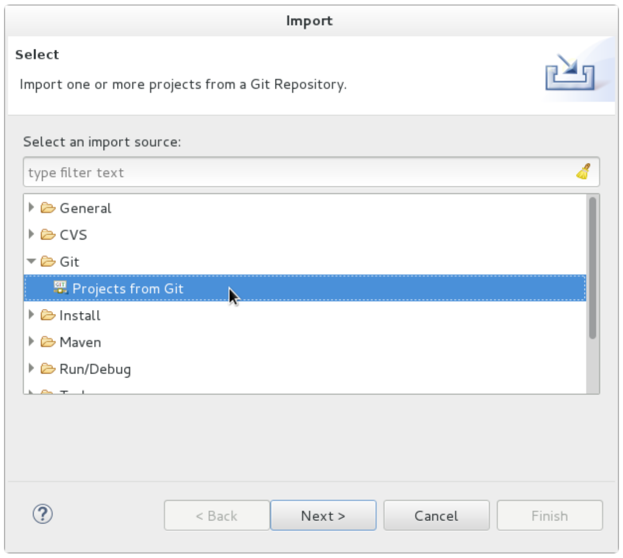
Note that you can always get any views you delete back again, using the `Window→Show View` menu option.

2.4 Create a New Project by Cloning the Public Marauroa Git Repository

Next, we're going to pull down the 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.

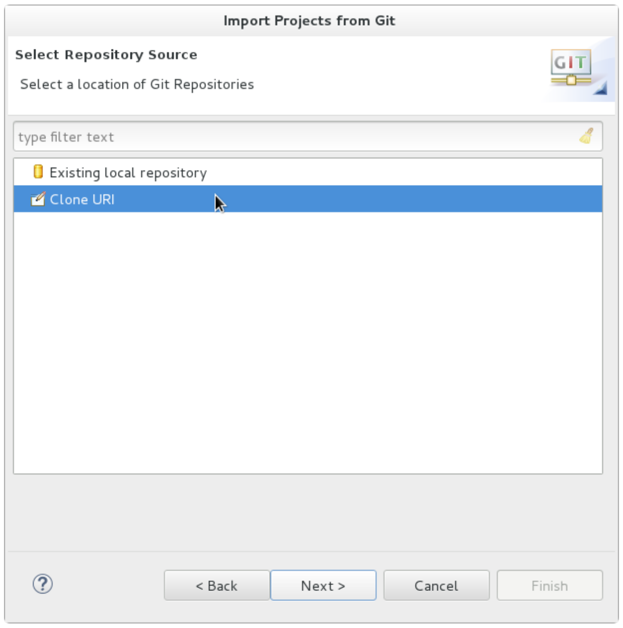
The first step is to ask Eclipse to import the Marauroa project for us, from a public Git repository.

Select the `File→Import` menu option. Then choose `Git→ImportFromGit`.



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

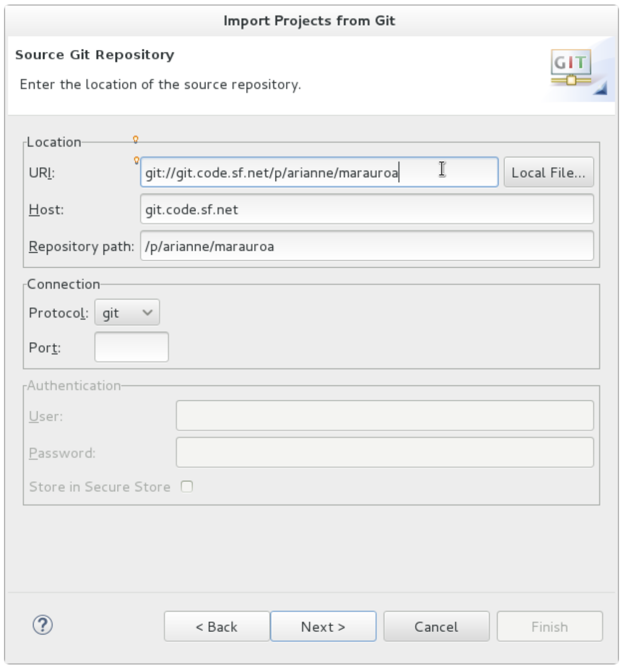
A dialogue box appears showing the two ways in which you can import a project from Git. We're going to **clone a project from a URI**, so select that option:



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 this before proceeding:



URI: git://git.code.sf.net/p/arianne/marauroa
Host: git.code.sf.net
Repository path: /p/arianne/marauroa
Protocol: git
Port:

If everything looks okay then select `Next`.

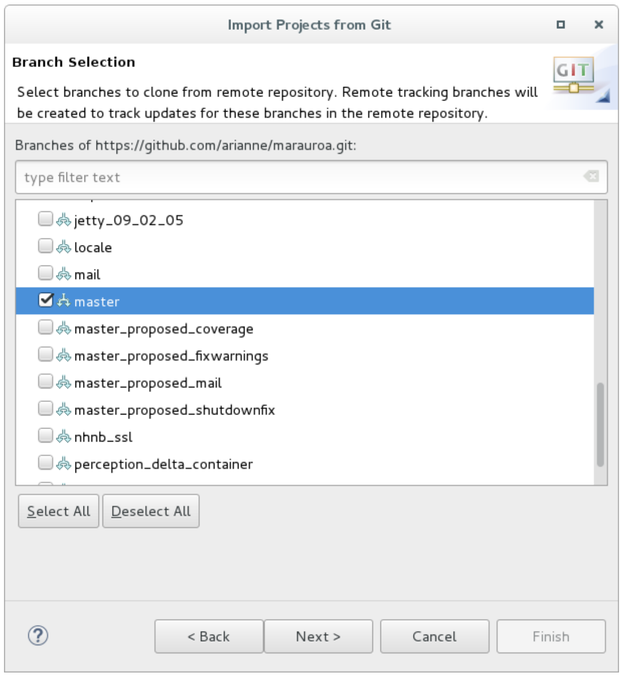
Clone Attempt Fails With An Error?

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.



Select branches to clone from remote repository. Remote tracking branches will be created to track updates for these branches in the remote repository.

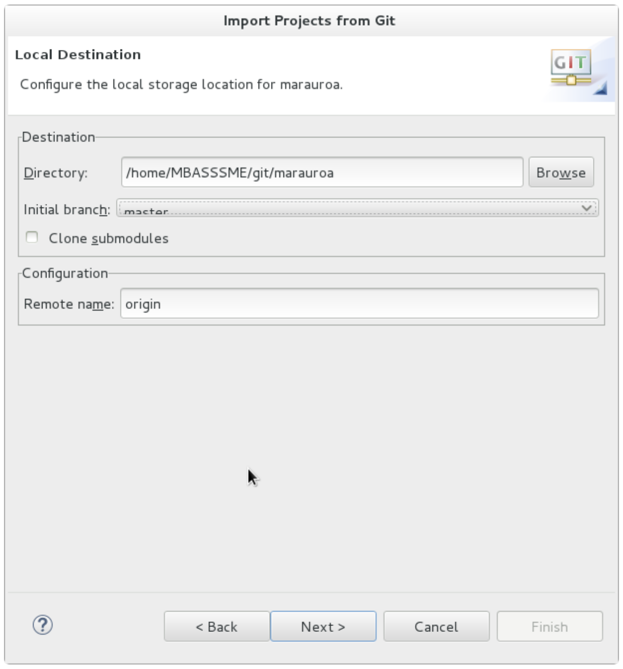
Branches of https://github.com/arianne/marauroa.git:

- [ ] jetty_09_02_05
- [ ] locale
- [ ] mail
- [✓] master
- [ ] master_proposed_coverage
- [ ] master_proposed_fixwarnings
- [ ] master_proposed_mail
- [ ] master_proposed_shutdownfix
- [ ] nhnb_ssl
- [ ] perception_delta_container

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.)

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.


Import Projects from Git

Local Destination

Configure the local storage location for marauroa.

Destination

Directory: /home/MBASSSME/git/marauroa

Initial branch: master

Clone submodules

Configuration

Remote name: origin

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.

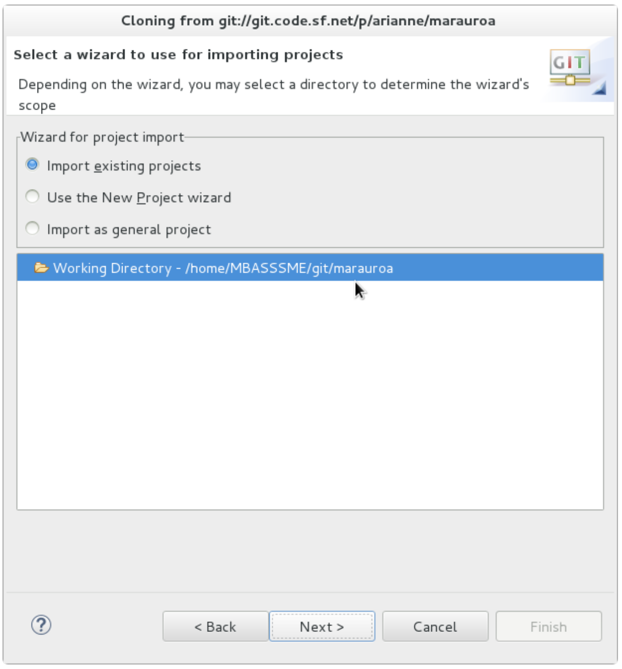
The next step is to import the Marauroa project from your local Git repository into Eclipse, so you can start to work on it.

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).

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. (If we were loading a project built in another IDE, we would need to use one of the other wizards).


Cloning from git://git.code.sf.net/p/arianne/marauroa

Select a wizard to use for importing projects

Depending on the wizard, you may select a directory to determine the wizard's scope

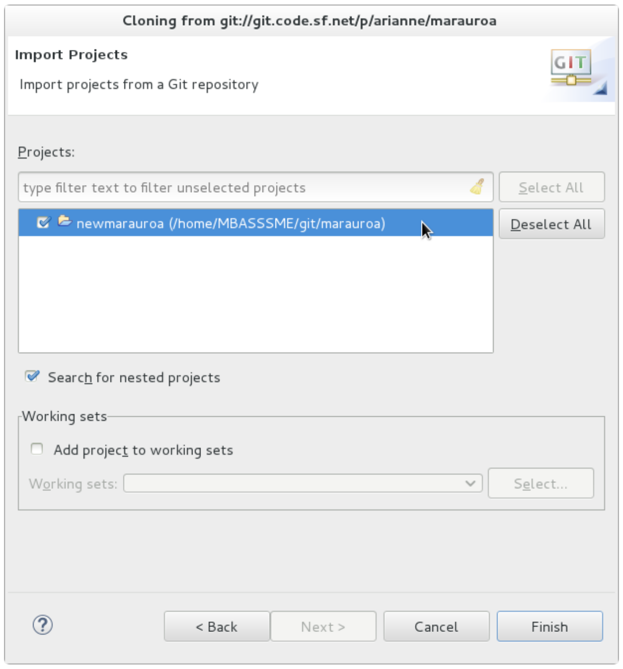
Wizard for project import

(●) Import existing projects
( ) Use the New Project wizard
( ) Import as general project

Working Directory - /home/MBASSSME/git/marauroa

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`):


Cloning from git://git.code.sf.net/p/arianne/marauroa

Import Projects

Import projects from a Git repository

Projects:

type filter text to filter unselected projects

✓ newmarauroa (/home/MBASSSME/git/marauroa)

✓ Search for nested projects

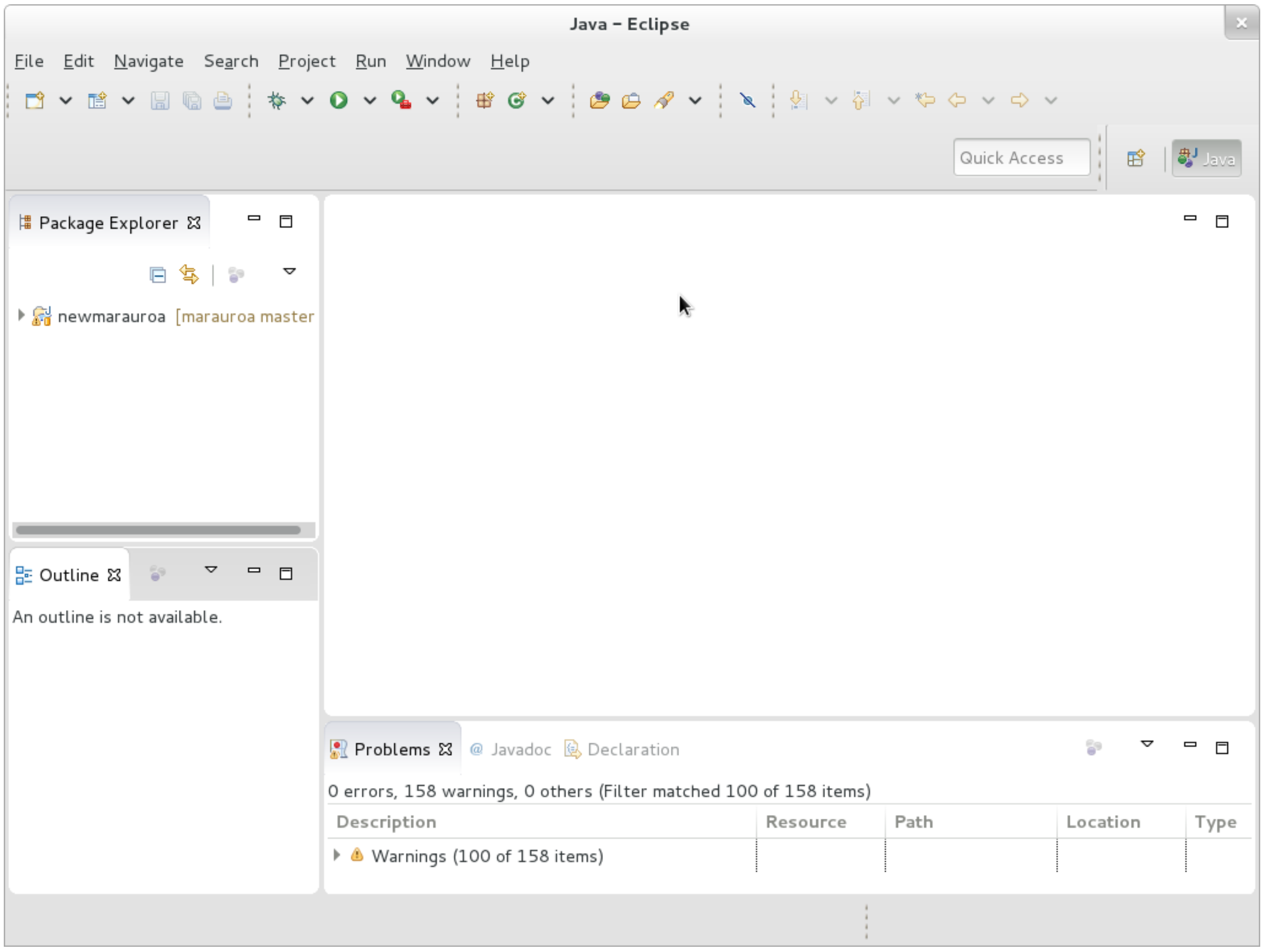
Working sets

Add project to working sets

Working sets:

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:


Java - Eclipse
File Edit Navigate Search Project Run Window Help
Quick Access
Java
Package Explorer
newmarauroa [marauroa master
Outline
An outline is not available.
Problems
Javadoc
Declaration
Description
Resource
Path
Location
Type
Warnings (100 of 158 items)

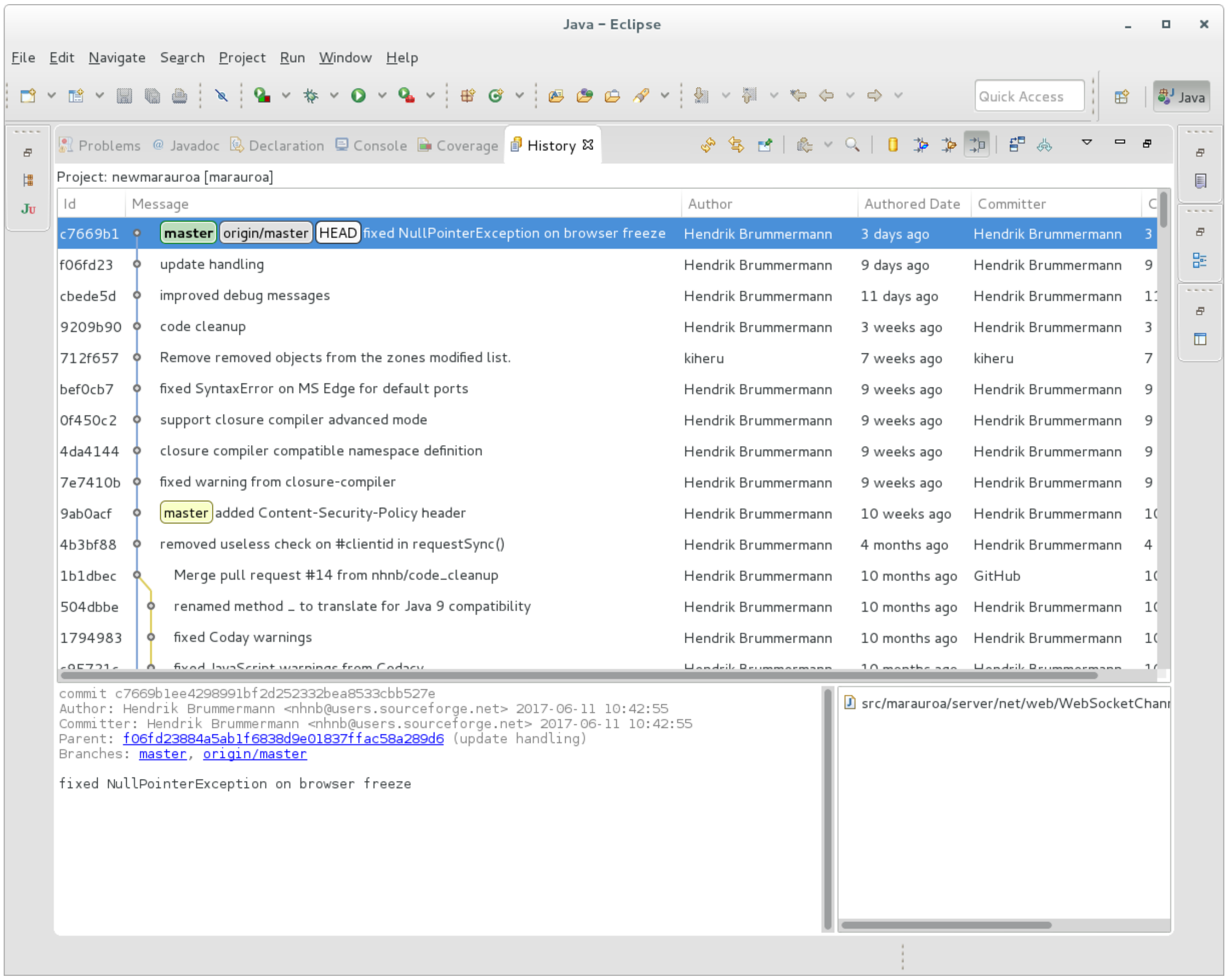
0 errors, 158 warnings, 0 others (Filter matched 100 of 158 items)

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: `f06fd23`.

The easiest way to check out a commit, branch or tag from with Eclipse is to use the History View. To open it, right click on the `newmarauroa` project name in the Package Explorer view. Select `Team→Show in History` from the menu that appears. The History View 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.



Project: newmarauroa [marauroa]

Id	Message	Author	Authored Date	Committer	C
c7669b1	master origin/master HEAD fixed NullPointerException on browser freeze	Hendrik Brummermann	3 days ago	Hendrik Brummermann	3
f06fd23	update handling	Hendrik Brummermann	9 days ago	Hendrik Brummermann	9
cbede5d	improved debug messages	Hendrik Brummermann	11 days ago	Hendrik Brummermann	1
9209b90	code cleanup	Hendrik Brummermann	3 weeks ago	Hendrik Brummermann	3
712f657	Remove removed objects from the zones modified list.	kiheru	7 weeks ago	kiheru	7
bef0cb7	fixed SyntaxError on MS Edge for default ports	Hendrik Brummermann	9 weeks ago	Hendrik Brummermann	9
0f450c2	support closure compiler advanced mode	Hendrik Brummermann	9 weeks ago	Hendrik Brummermann	9
4da4144	closure compiler compatible namespace definition	Hendrik Brummermann	9 weeks ago	Hendrik Brummermann	9
7e7410b	fixed warning from closure-compiler	Hendrik Brummermann	9 weeks ago	Hendrik Brummermann	9
9ab0acf	master added Content-Security-Policy header	Hendrik Brummermann	10 weeks ago	Hendrik Brummermann	1
4b3bf88	removed useless check on #clientid in requestSync()	Hendrik Brummermann	4 months ago	Hendrik Brummermann	4
1b1dbec	Merge pull request #14 from nhnb/code_cleanup	Hendrik Brummermann	10 months ago	GitHub	1
504dbbe	renamed method _ to translate for Java 9 compatibility	Hendrik Brummermann	10 months ago	Hendrik Brummermann	1
1794983	fixed Coday warnings	Hendrik Brummermann	10 months ago	Hendrik Brummermann	1
c95721c	fixed JavaScript warnings from Codacy	Hendrik Brummermann	10 months ago	Hendrik Brummermann	1

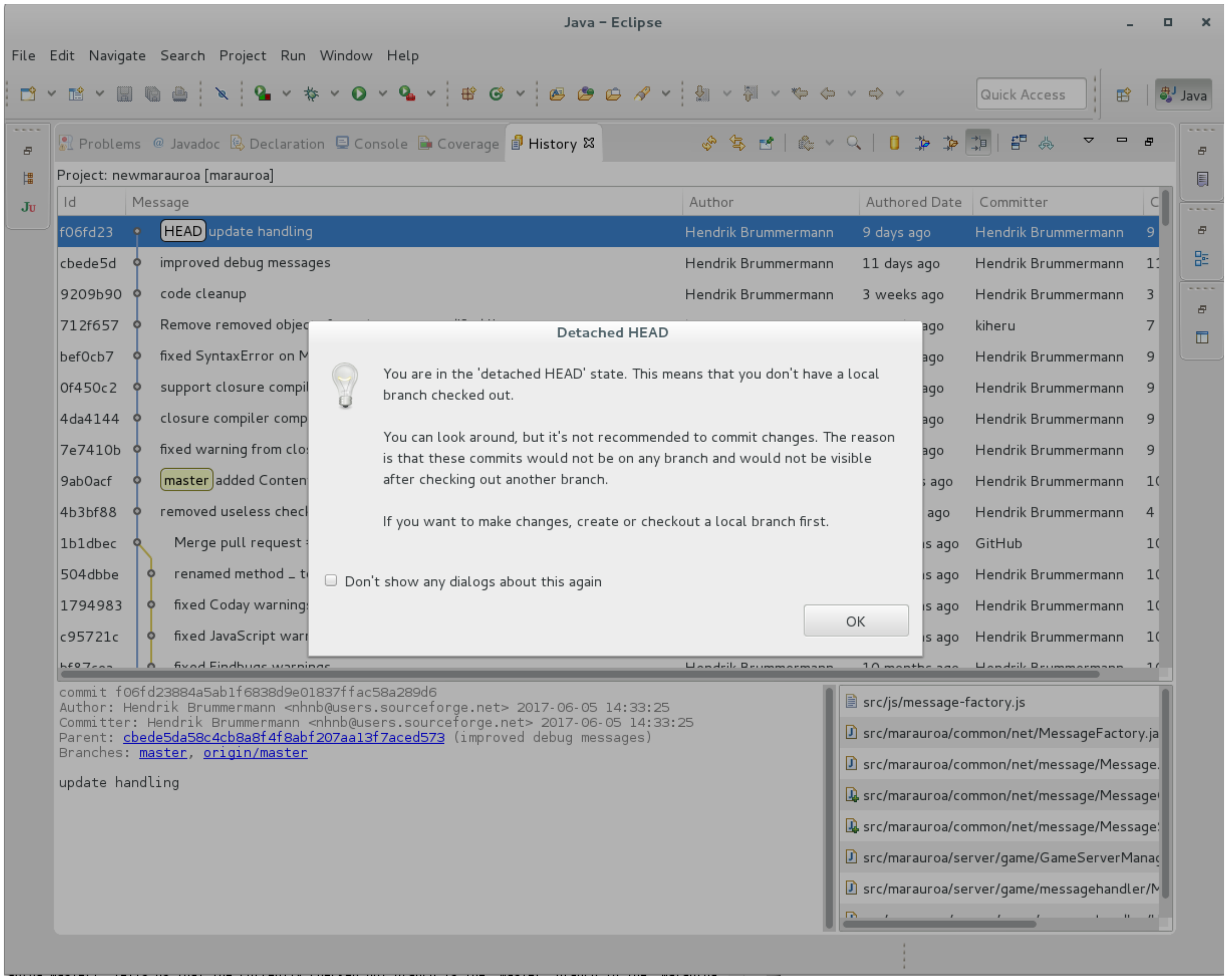
```
commit c7669b1ee4298991bf2d252332bea8533cbb527e
Author: Hendrik Brummermann <nhnb@users.sourceforge.net> 2017-06-11 10:42:55
Committer: Hendrik Brummermann <nhnb@users.sourceforge.net> 2017-06-11 10:42:55
Parent: f06fd23884a5ab1f6838d9e01837ffac58a289d6 (update handling)
Branches: master, origin/master

fixed NullPointerException on browser freeze
```

src/marauroa/server/net/web/WebSocketChan

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 `f06fd23`. It should have the (not terribly helpful) commit message "`update handling`". 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.


Java - Eclipse
File Edit Navigate Search Project Run Window Help
Quick Access
Java
Problems Javadoc Declaration Console Coverage History

Project: newmarauroa [marauroa]

Id	Message	Author	Authored Date	Committer	C
f06fd23	HEAD update handling	Hendrik Brummermann	9 days ago	Hendrik Brummermann	9
cbede5d	improved debug messages	Hendrik Brummermann	11 days ago	Hendrik Brummermann	1
9209b90	code cleanup	Hendrik Brummermann	3 weeks ago	Hendrik Brummermann	3
712f657	Remove removed objec		ago	kiheru	7
bef0cb7	fixed SyntaxError on M		ago	Hendrik Brummermann	9
0f450c2	support closure compil		ago	Hendrik Brummermann	9
4da4144	closure compiler comp		ago	Hendrik Brummermann	9
7e7410b	fixed warning from clo		ago	Hendrik Brummermann	9
9ab0acf	master added Conten		ago	Hendrik Brummermann	1
4b3bf88	removed useless checl		ago	Hendrik Brummermann	4
1b1dbec	Merge pull request		s ago	GitHub	1
504dbbe	renamed method _ t		s ago	Hendrik Brummermann	1
1794983	fixed Coday warning		s ago	Hendrik Brummermann	1
c95721c	fixed JavaScript warr		s ago	Hendrik Brummermann	1

Detached HEAD

You are in the 'detached HEAD' state. This means that 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.

If you want to make changes, create or checkout a local branch first.

☐ Don't show any dialogs about this again

OK

```
commit f06fd23884a5ab1f6838d9e01837ffac58a289d6
Author: Hendrik Brummermann <nhnb@users.sourceforge.net> 2017-06-05 14:33:25
Committer: Hendrik Brummermann <nhnb@users.sourceforge.net> 2017-06-05 14:33:25
Parent: cbede5da58c4cb8a8f4f8abf207aa13f7aced573 (improved debug messages)
Branches: master, origin/master

update handling
```

src/js/message-factory.js
src/marauroa/common/net/MessageFactory.ja
src/marauroa/common/net/message/Message.
src/marauroa/common/net/message/Message
src/marauroa/common/net/message/Message
src/marauroa/server/game/GameServerManag
src/marauroa/server/game/messagehandler/M

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. 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 ignore the warning.

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 Git?*

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.

2.6 Explore the Project and Look Around a Little

You now have your own copy of the Marauroa project source to play with. 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 f06fd23]”.) 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 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 auto-**

mated build scripts provided by the Marauroa team to build an executable version of the Marauroa engine.

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.

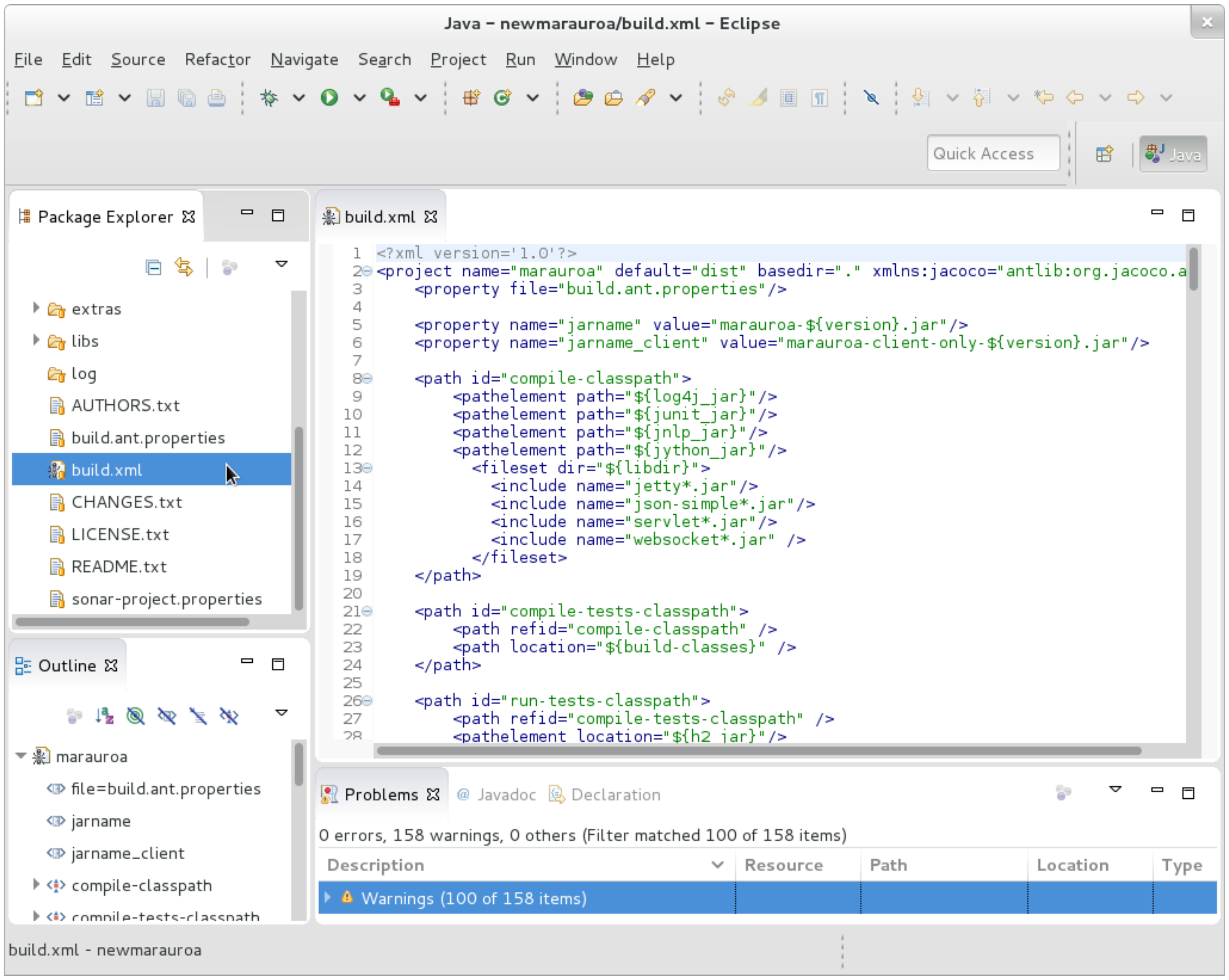
In this step, you're going to be introduced to the Apache Ant automated build tool, which is the tool chosen by the Marauroa team for use on their project. You'll learn how to use it to create executable code from the source we've just downloaded.

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!)

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.



```
<?xml version='1.0'?>
<project name="marauroa" default="dist" basedir="." xmlns:jacoco="antlib:org.jacoco.a
    <property file="build.ant.properties"/>

    <property name="jarname" value="marauroa-${version}.jar"/>
    <property name="jarname_client" value="marauroa-client-only-${version}.jar"/>

    <path id="compile-classpath">
        <pathelement path="${log4j_jar}"/>
        <pathelement path="${junit_jar}"/>
        <pathelement path="${jnlp_jar}"/>
        <pathelement path="${jython_jar}"/>
          <fileset dir="${libdir}">
            <include name="jetty*.jar"/>
            <include name="json-simple*.jar"/>
            <include name="servlet*.jar"/>
            <include name="websocket*.jar" />
          </fileset>
    </path>

    <path id="compile-tests-classpath">
        <path refid="compile-classpath" />
        <path location="${build-classes}" />
    </path>

    <path id="run-tests-classpath">
        <path refid="compile-tests-classpath" />
        <pathelement location="${h2_jar}"/>
```

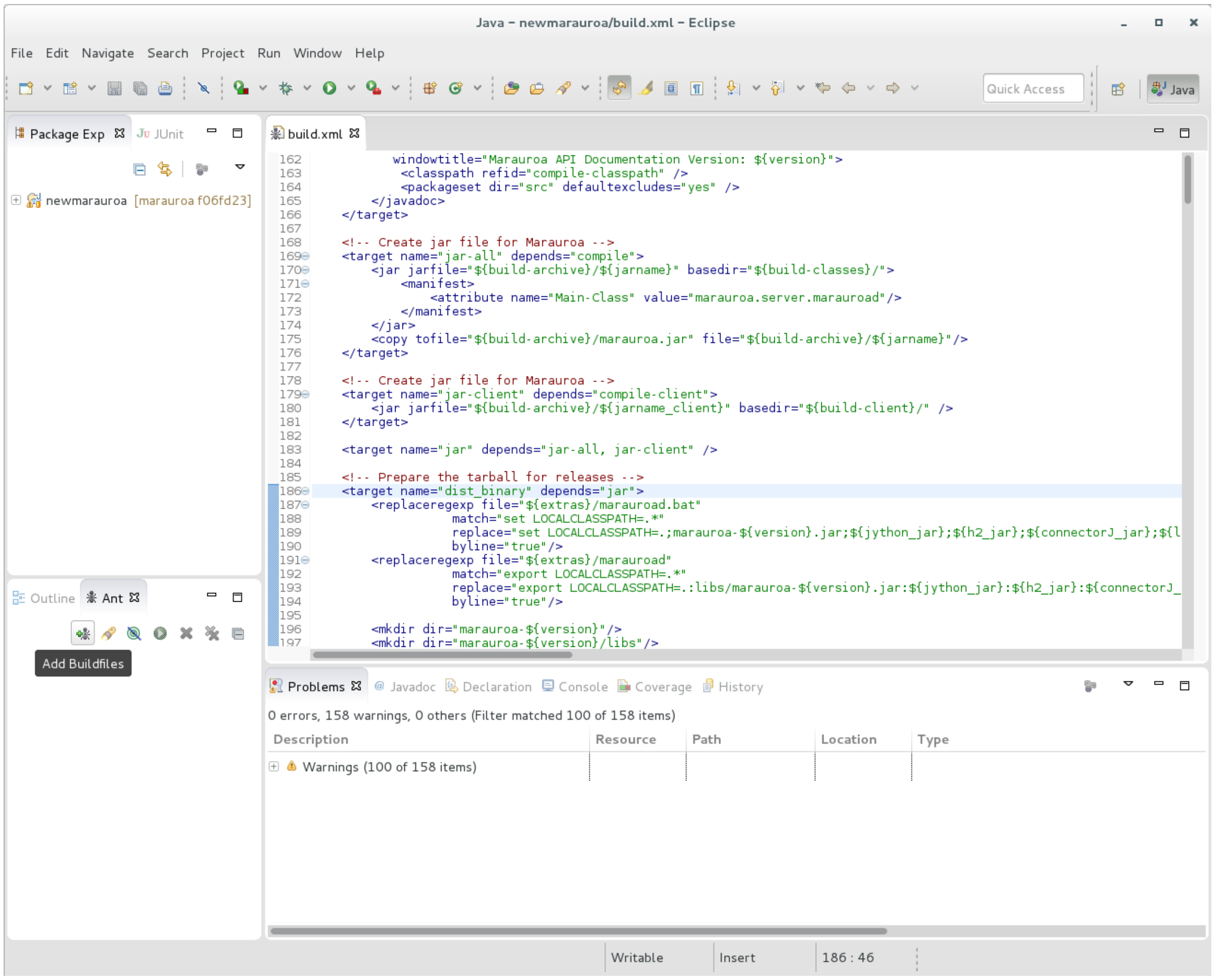
0 errors, 158 warnings, 0 others (Filter matched 100 of 158 items)

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.


Java – newmarauroa/build.xml – Eclipse
File Edit Navigate Search Project Run Window Help
Quick Access
Java
Package Exp
JUnit
newmarauroa [marauroa f06fd23]
build.xml
Outline
Ant
Add Buildfiles
Problems
Javadoc
Declaration
Console
Coverage
History

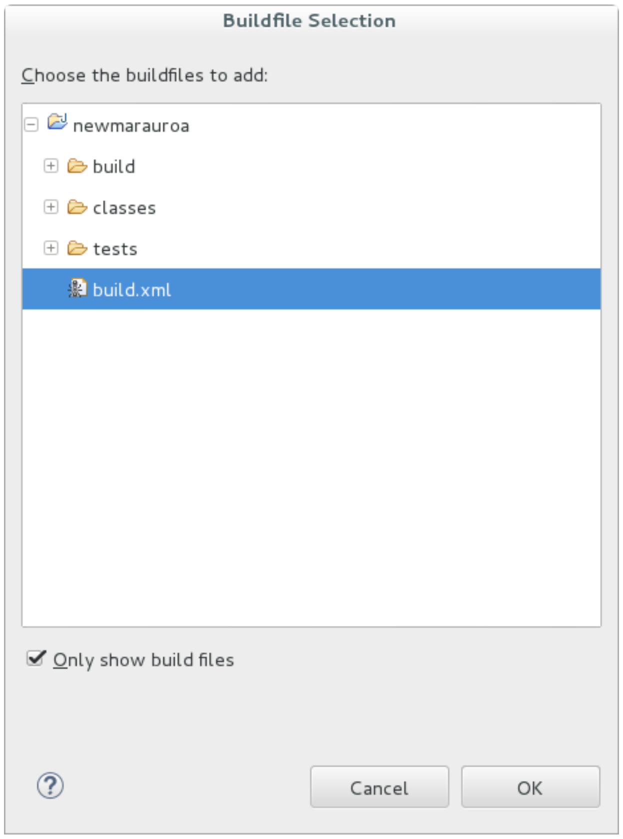
```
            windowtitle="Marauroa API Documentation Version: ${version}">
             <classpath refid="compile-classpath" />
             <packageset dir="src" defaultexcludes="yes" />
        </javadoc>
    </target>

    <!-- Create jar file for Marauroa -->
    <target name="jar-all" depends="compile">
        <jar jarfile="${build-archive}/${jarname}" basedir="${build-classes}/">
            <manifest>
                <attribute name="Main-Class" value="marauroa.server.marauroad"/>
            </manifest>
        </jar>
        <copy tofile="${build-archive}/marauroa.jar" file="${build-archive}/${jarname}"/>
    </target>

    <!-- Create jar file for Marauroa -->
    <target name="jar-client" depends="compile-client">
        <jar jarfile="${build-archive}/${jarname_client}" basedir="${build-client}/" />
    </target>

    <target name="jar" depends="jar-all, jar-client" />

    <!-- Prepare the tarball for releases -->
    <target name="dist_binary" depends="jar">
        <replaceregexp file="${extras}/marauroad.bat"
                   match="set LOCALCLASSPATH=.*"
                   replace="set LOCALCLASSPATH=.;marauroa-${version}.jar;${jython_jar};${h2_jar};${connectorJ_jar};${l
                   byline="true"/>
        <replaceregexp file="${extras}/marauroad"
                   match="export LOCALCLASSPATH=.*"
                   replace="export LOCALCLASSPATH=.:libs/marauroa-${version}.jar:${jython_jar}:${h2_jar}:${connectorJ_
                   byline="true"/>

        <mkdir dir="marauroa-${version}"/>
        <mkdir dir="marauroa-${version}/libs"/>
```

0 errors, 158 warnings, 0 others (Filter matched 100 of 158 items)

Description	Resource	Path	Location	Type
Warnings (100 of 158 items)				

Writable | Insert | 186 : 46

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



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

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.

3.2 Build the System Using the Build Script

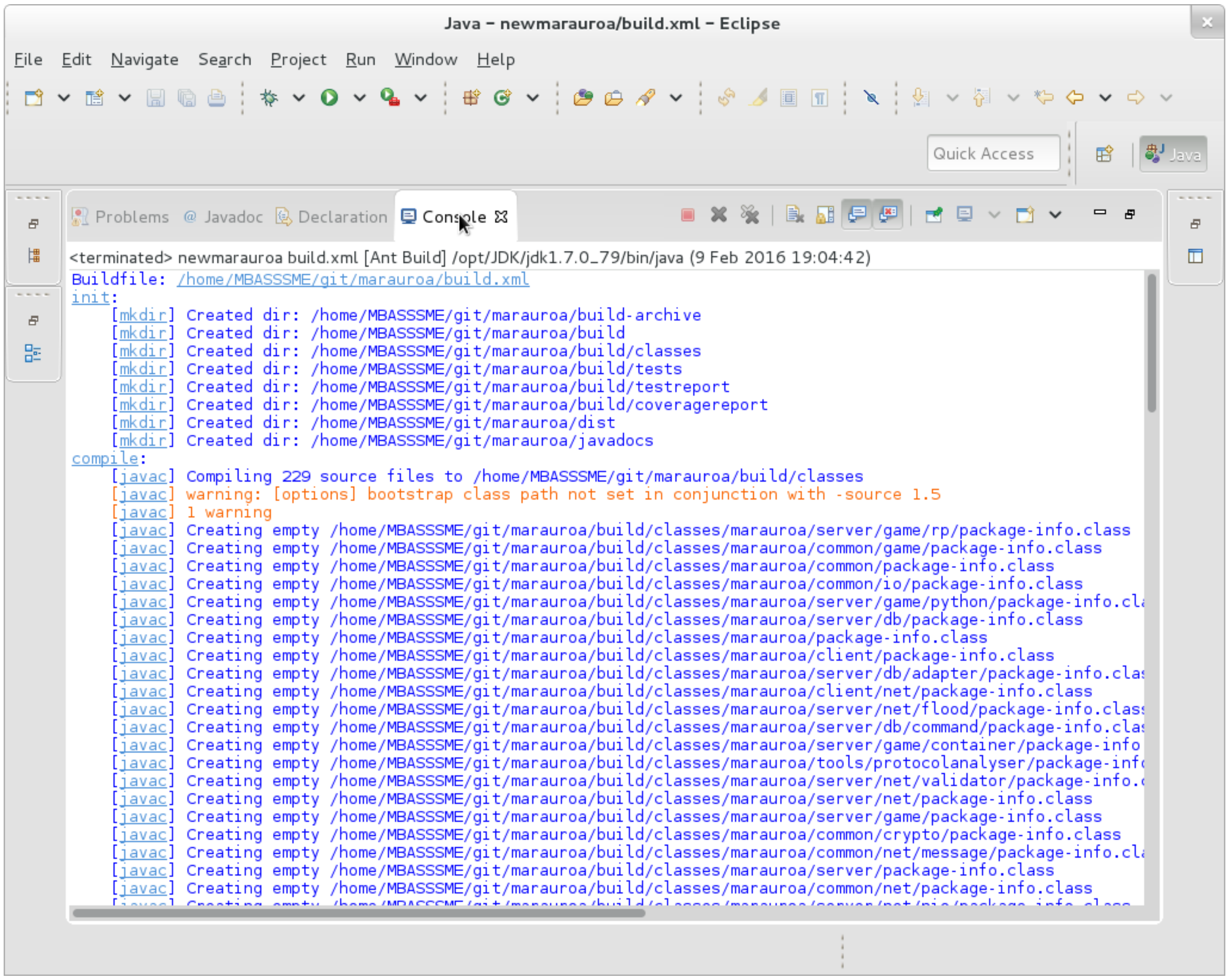
Now that we have seen something of the build script, we are going to use it to build the whole Marauroa distribution. That is, **we are going to ask Ant to build the "dist" target**.

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 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.

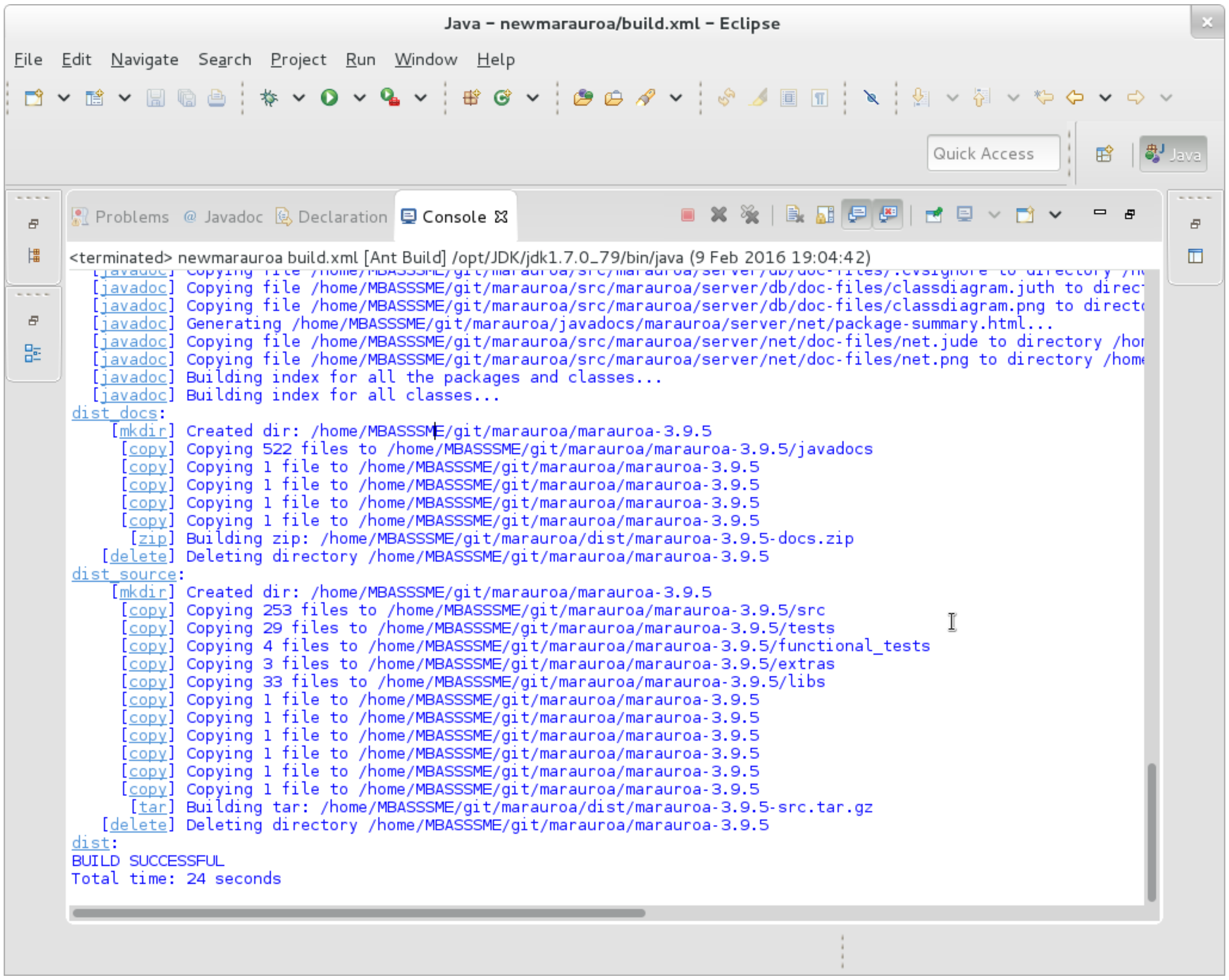


```
<terminated> newmarauroa build.xml [Ant Build] /opt/JDK/jdk1.7.0_79/bin/java (9 Feb 2016 19:04:42)
Buildfile: /home/MBASSSME/git/marauroa/build.xml
init:
    [mkdir] Created dir: /home/MBASSSME/git/marauroa/build-archive
    [mkdir] Created dir: /home/MBASSSME/git/marauroa/build
    [mkdir] Created dir: /home/MBASSSME/git/marauroa/build/classes
    [mkdir] Created dir: /home/MBASSSME/git/marauroa/build/tests
    [mkdir] Created dir: /home/MBASSSME/git/marauroa/build/testreport
    [mkdir] Created dir: /home/MBASSSME/git/marauroa/build/coveragereport
    [mkdir] Created dir: /home/MBASSSME/git/marauroa/dist
    [mkdir] Created dir: /home/MBASSSME/git/marauroa/javadocs
compile:
    [javac] Compiling 229 source files to /home/MBASSSME/git/marauroa/build/classes
    [javac] warning: [options] bootstrap class path not set in conjunction with -source 1.5
    [javac] 1 warning
    [javac] Creating empty /home/MBASSSME/git/marauroa/build/classes/marauroa/server/game/rp/package-info.class
    [javac] Creating empty /home/MBASSSME/git/marauroa/build/classes/marauroa/common/game/package-info.class
    [javac] Creating empty /home/MBASSSME/git/marauroa/build/classes/marauroa/common/package-info.class
    [javac] Creating empty /home/MBASSSME/git/marauroa/build/classes/marauroa/common/io/package-info.class
    [javac] Creating empty /home/MBASSSME/git/marauroa/build/classes/marauroa/server/game/python/package-info.cla
    [javac] Creating empty /home/MBASSSME/git/marauroa/build/classes/marauroa/server/db/package-info.class
    [javac] Creating empty /home/MBASSSME/git/marauroa/build/classes/marauroa/package-info.class
    [javac] Creating empty /home/MBASSSME/git/marauroa/build/classes/marauroa/client/package-info.class
    [javac] Creating empty /home/MBASSSME/git/marauroa/build/classes/marauroa/server/db/adapter/package-info.clas
    [javac] Creating empty /home/MBASSSME/git/marauroa/build/classes/marauroa/client/net/package-info.class
    [javac] Creating empty /home/MBASSSME/git/marauroa/build/classes/marauroa/server/net/flood/package-info.class
    [javac] Creating empty /home/MBASSSME/git/marauroa/build/classes/marauroa/server/db/command/package-info.clas
    [javac] Creating empty /home/MBASSSME/git/marauroa/build/classes/marauroa/server/game/container/package-info
    [javac] Creating empty /home/MBASSSME/git/marauroa/build/classes/marauroa/tools/protocolanalyser/package-inf
    [javac] Creating empty /home/MBASSSME/git/marauroa/build/classes/marauroa/server/net/validator/package-info.
    [javac] Creating empty /home/MBASSSME/git/marauroa/build/classes/marauroa/server/net/package-info.class
    [javac] Creating empty /home/MBASSSME/git/marauroa/build/classes/marauroa/server/game/package-info.class
    [javac] Creating empty /home/MBASSSME/git/marauroa/build/classes/marauroa/common/crypto/package-info.class
    [javac] Creating empty /home/MBASSSME/git/marauroa/build/classes/marauroa/common/net/message/package-info.cla
    [javac] Creating empty /home/MBASSSME/git/marauroa/build/classes/marauroa/server/package-info.class
    [javac] Creating empty /home/MBASSSME/git/marauroa/build/classes/marauroa/common/net/package-info.class
```

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:


Java - newmarauroa/build.xml - Eclipse
File Edit Navigate Search Project Run Window Help
Quick Access
Java
Problems @ Javadoc Declaration Console

```
<terminated> newmarauroa build.xml [Ant Build] /opt/JDK/jdk1.7.0_79/bin/java (9 Feb 2016 19:04:42)
  [javadoc] Copying file /home/MBASSSME/git/marauroa/src/marauroa/server/db/doc-files/.cvsignore to directory /ho
  [javadoc] Copying file /home/MBASSSME/git/marauroa/src/marauroa/server/db/doc-files/classdiagram.juth to direc
  [javadoc] Copying file /home/MBASSSME/git/marauroa/src/marauroa/server/db/doc-files/classdiagram.png to directo
  [javadoc] Generating /home/MBASSSME/git/marauroa/javadocs/marauroa/server/net/package-summary.html...
  [javadoc] Copying file /home/MBASSSME/git/marauroa/src/marauroa/server/net/doc-files/net.jude to directory /hor
  [javadoc] Copying file /home/MBASSSME/git/marauroa/src/marauroa/server/net/doc-files/net.png to directory /home
  [javadoc] Building index for all the packages and classes...
  [javadoc] Building index for all classes...
dist_docs:
    [mkdir] Created dir: /home/MBASSSME/git/marauroa/marauroa-3.9.5
     [copy] Copying 522 files to /home/MBASSSME/git/marauroa/marauroa-3.9.5/javadocs
     [copy] Copying 1 file to /home/MBASSSME/git/marauroa/marauroa-3.9.5
     [copy] Copying 1 file to /home/MBASSSME/git/marauroa/marauroa-3.9.5
     [copy] Copying 1 file to /home/MBASSSME/git/marauroa/marauroa-3.9.5
     [copy] Copying 1 file to /home/MBASSSME/git/marauroa/marauroa-3.9.5
      [zip] Building zip: /home/MBASSSME/git/marauroa/dist/marauroa-3.9.5-docs.zip
   [delete] Deleting directory /home/MBASSSME/git/marauroa/marauroa-3.9.5
dist_source:
    [mkdir] Created dir: /home/MBASSSME/git/marauroa/marauroa-3.9.5
     [copy] Copying 253 files to /home/MBASSSME/git/marauroa/marauroa-3.9.5/src
     [copy] Copying 29 files to /home/MBASSSME/git/marauroa/marauroa-3.9.5/tests
     [copy] Copying 4 files to /home/MBASSSME/git/marauroa/marauroa-3.9.5/functional_tests
     [copy] Copying 3 files to /home/MBASSSME/git/marauroa/marauroa-3.9.5/extras
     [copy] Copying 33 files to /home/MBASSSME/git/marauroa/marauroa-3.9.5/libs
     [copy] Copying 1 file to /home/MBASSSME/git/marauroa/marauroa-3.9.5
     [copy] Copying 1 file to /home/MBASSSME/git/marauroa/marauroa-3.9.5
     [copy] Copying 1 file to /home/MBASSSME/git/marauroa/marauroa-3.9.5
     [copy] Copying 1 file to /home/MBASSSME/git/marauroa/marauroa-3.9.5
     [copy] Copying 1 file to /home/MBASSSME/git/marauroa/marauroa-3.9.5
     [copy] Copying 1 file to /home/MBASSSME/git/marauroa/marauroa-3.9.5
      [tar] Building tar: /home/MBASSSME/git/marauroa/dist/marauroa-3.9.5-src.tar.gz
   [delete] Deleting directory /home/MBASSSME/git/marauroa/marauroa-3.9.5
dist:
BUILD SUCCESSFUL
Total time: 24 seconds
```

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, 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.

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 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.

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.

4.1 Finding Out What Tests There Are to Run

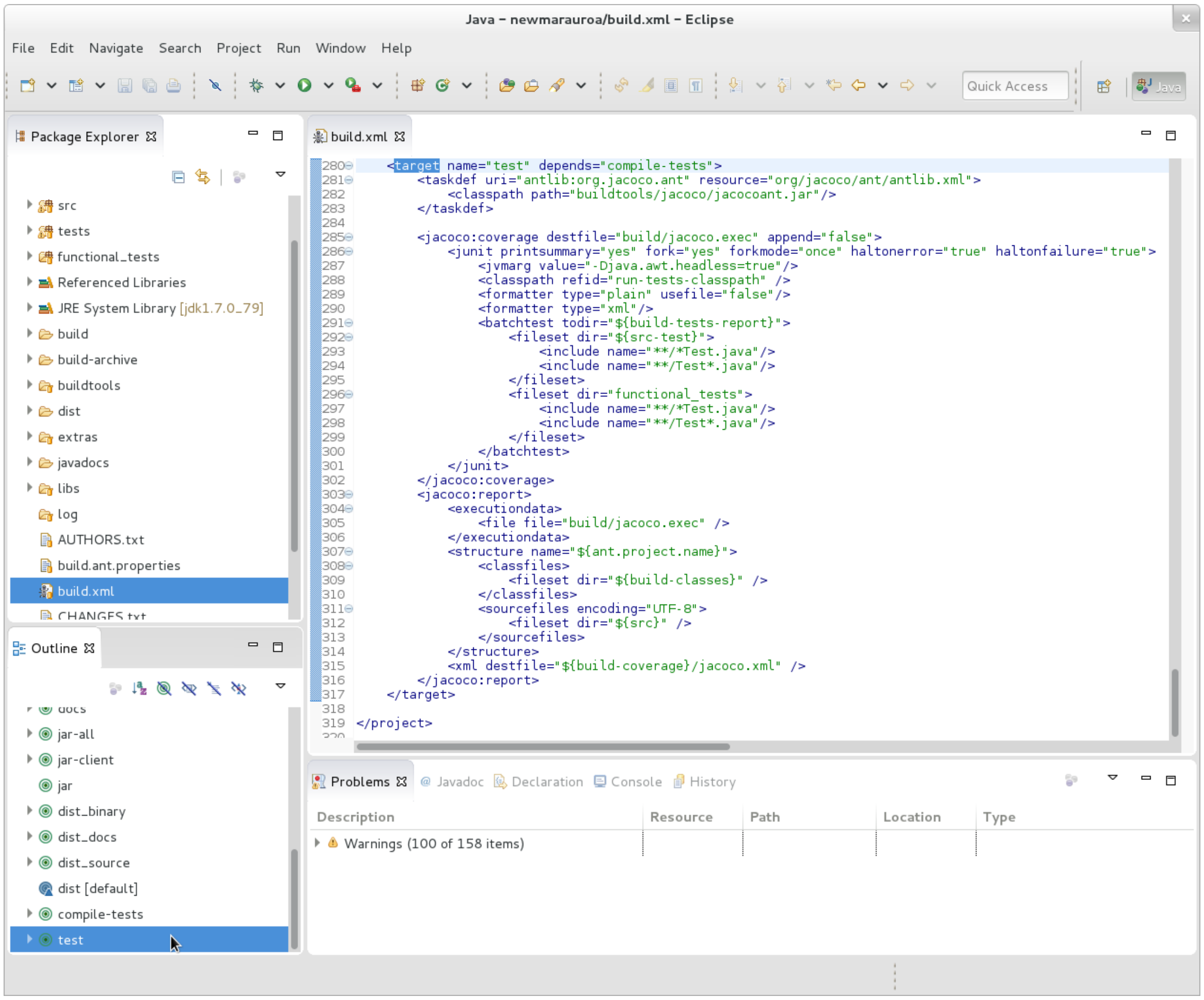
Before we run the tests, it is helpful first to take a high level look at the test suites provided by the developers of the system we are working with. One way to do this is to look at the source folders in the project. The source code for any large project, nowadays, is typically split into two halves: 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.

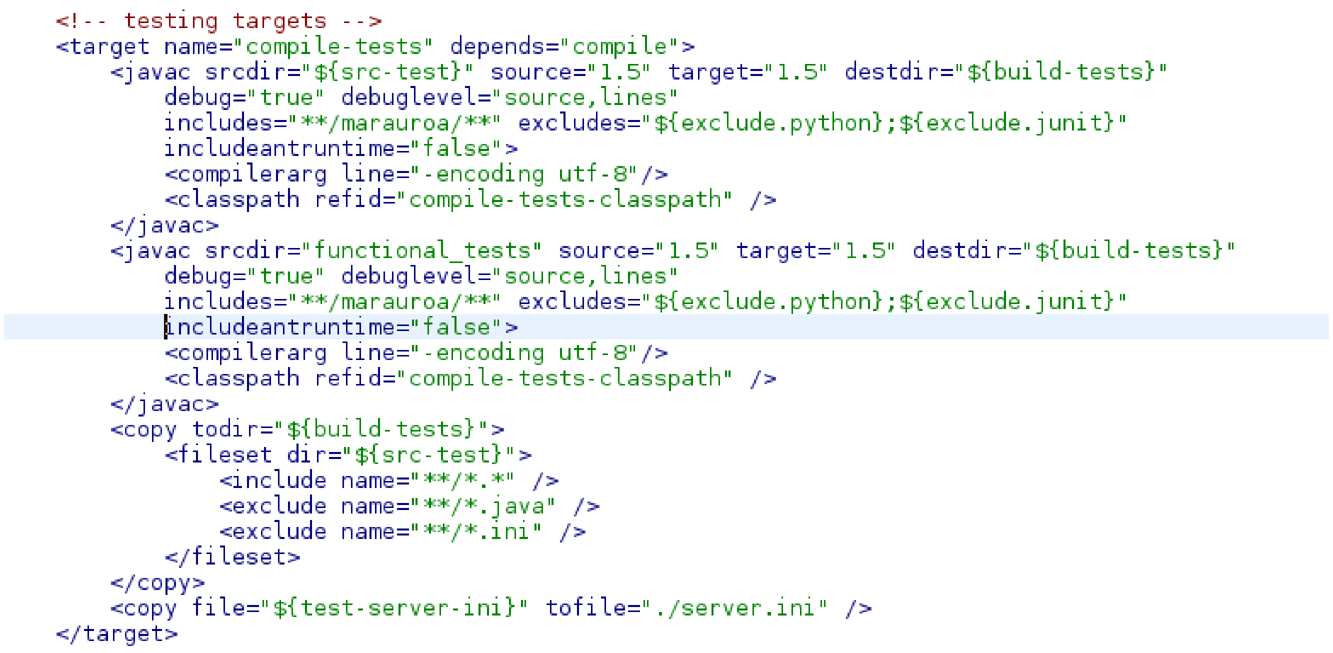
Take a look at the targets in the build script. We can see that there is one called `test`. That sounds promising. Let's take a look.



```
    <target name="test" depends="compile-tests">
        <taskdef uri="antlib:org.jacoco.ant" resource="org/jacoco/ant/antlib.xml">
            <classpath path="buildtools/jacoco/jacocoant.jar"/>
        </taskdef>

        <jacoco:coverage destfile="build/jacoco.exec" append="false">
            <junit printsummary="yes" fork="yes" forkmode="once" haltonerror="true" haltonfailure="true">
                <jvmarg value="-Djava.awt.headless=true"/>
                <classpath refid="run-tests-classpath" />
                <formatter type="plain" usefile="false"/>
                <formatter type="xml"/>
                <batchtest todir="${build-tests-report}">
                    <fileset dir="${src-test}">
                        <include name="**/*Test.java"/>
                        <include name="**/Test*.java"/>
                    </fileset>
                    <fileset dir="functional_tests">
                        <include name="**/*Test.java"/>
                        <include name="**/Test*.java"/>
                    </fileset>
                </batchtest>
            </junit>
        </jacoco:coverage>
        <jacoco:report>
            <executiondata>
                <file file="build/jacoco.exec" />
            </executiondata>
            <structure name="${ant.project.name}">
                <classfiles>
                    <fileset dir="${build-classes}" />
                </classfiles>
                <sourcefiles encoding="UTF-8">
                    <fileset dir="${src}" />
                </sourcefiles>
            </structure>
            <xml destfile="${build-coverage}/jacoco.xml" />
        </jacoco:report>
    </target>

</project>
```

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.

Let's take a quick look at that target before we look at the rest of the `test` target.



```
<!-- 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>
```

This target 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).

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 COMP16212. 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.

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.)

This time, you should see a failed build. This is because one of the tests has failed. A little later, we'll look at some tools for working with test results, so we can examine just why the build failed. For now, we note only that the failing test caused the process of executing the test suite to come to a full stop. This is because the Marauroa build script tells JUnit to stop as soon as a failing test is encountered. 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.

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.

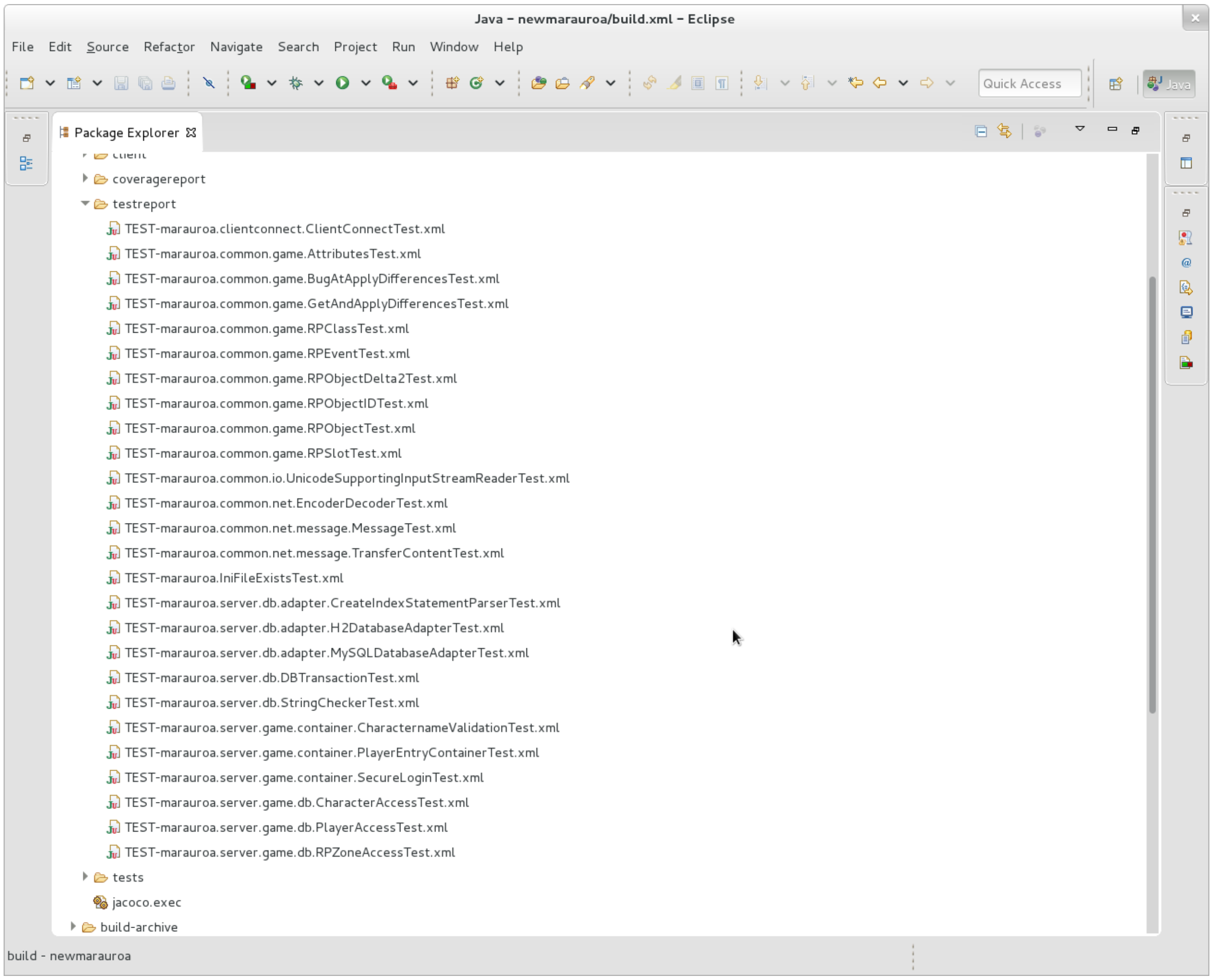
4.3 Examining the Test Results

It's useful to know that some of the tests failed, but the summary output we get on the console from the build is not very helpful when we want to know which tests exactly failed, and why. We need some way to get more detail on 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:


Java – newmarauroa/build.xml – Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Java
Package Explorer

```
client
coveragereport
testreport
  TEST-marauroa.clientconnect.ClientConnectTest.xml
  TEST-marauroa.common.game.AttributesTest.xml
  TEST-marauroa.common.game.BugAtApplyDifferencesTest.xml
  TEST-marauroa.common.game.GetAndApplyDifferencesTest.xml
  TEST-marauroa.common.game.RPClassTest.xml
  TEST-marauroa.common.game.RPEventTest.xml
  TEST-marauroa.common.game.RPObjectDelta2Test.xml
  TEST-marauroa.common.game.RPObjectIDTest.xml
  TEST-marauroa.common.game.RPObjectTest.xml
  TEST-marauroa.common.game.RPSlotTest.xml
  TEST-marauroa.common.io.UnicodeSupportingInputStreamReaderTest.xml
  TEST-marauroa.common.net.EncoderDecoderTest.xml
  TEST-marauroa.common.net.message.MessageTest.xml
  TEST-marauroa.common.net.message.TransferContentTest.xml
  TEST-marauroa.IniFileExistsTest.xml
  TEST-marauroa.server.db.adapter.CreateIndexStatementParserTest.xml
  TEST-marauroa.server.db.adapter.H2DatabaseAdapterTest.xml
  TEST-marauroa.server.db.adapter.MySQLDatabaseAdapterTest.xml
  TEST-marauroa.server.db.DBTransactionTest.xml
  TEST-marauroa.server.db.StringCheckerTest.xml
  TEST-marauroa.server.game.container.CharacternameValidationTest.xml
  TEST-marauroa.server.game.container.PlayerEntryContainerTest.xml
  TEST-marauroa.server.game.container.SecureLoginTest.xml
  TEST-marauroa.server.game.db.CharacterAccessTest.xml
  TEST-marauroa.server.game.db.PlayerAccessTest.xml
  TEST-marauroa.server.game.db.RPZoneAccessTest.xml
tests
jacoco.exec
build-archive
```

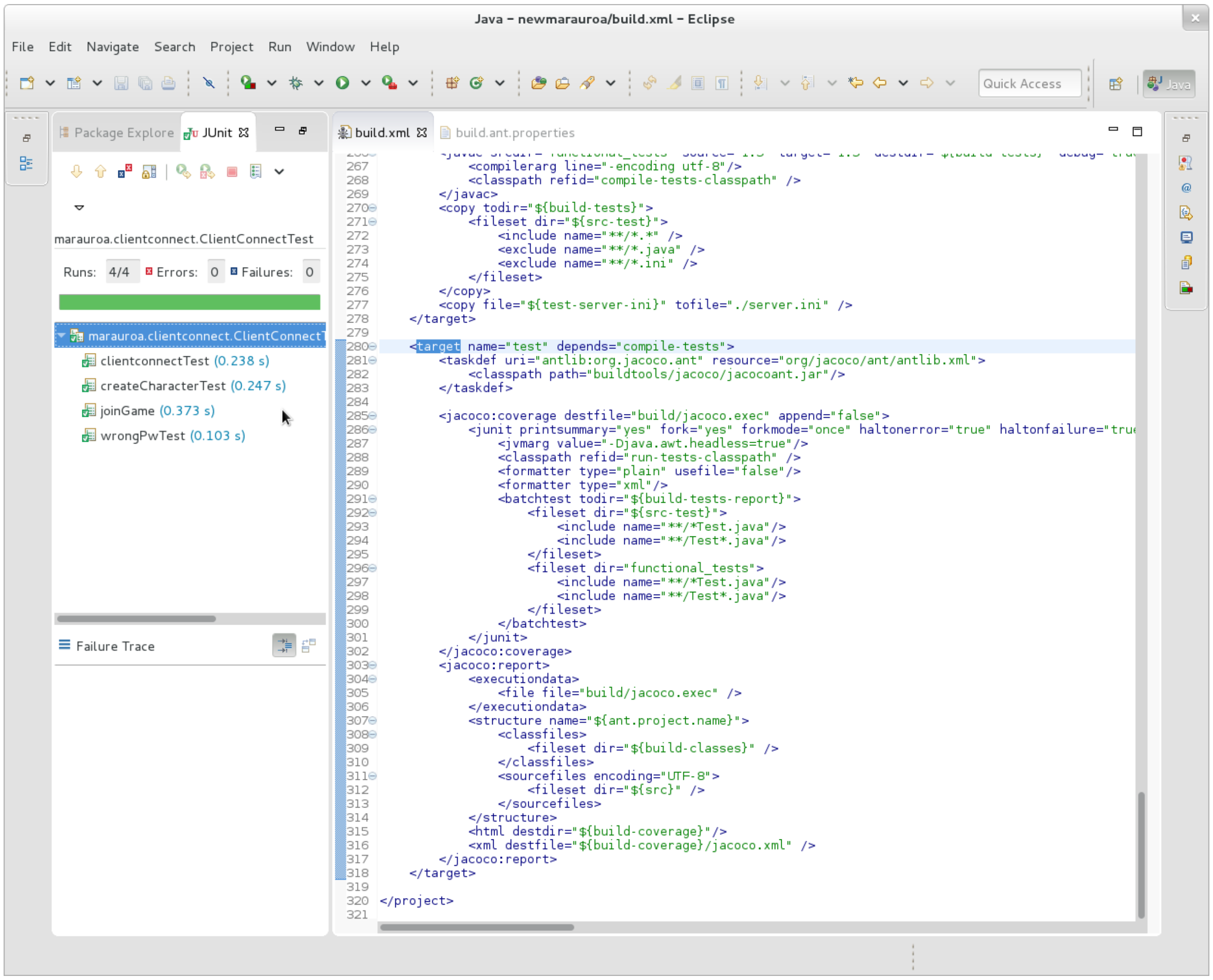
build - newmarauroa

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 following JUnit view:


Java – newmarauroa/build.xml – Eclipse
File Edit Navigate Search Project Run Window Help
Quick Access
Java
Package Explore
JUnit
build.xml
build.ant.properties
marauroa.clientconnect.ClientConnectTest
Runs: 4/4
Errors: 0
Failures: 0
marauroa.clientconnect.ClientConnect
clientconnectTest (0.238 s)
createCharacterTest (0.247 s)
joinGame (0.373 s)
wrongPwTest (0.103 s)
Failure Trace

```
            <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>

    <target name="test" depends="compile-tests">
        <taskdef uri="antlib:org.jacoco.ant" resource="org/jacoco/ant/antlib.xml">
            <classpath path="buildtools/jacoco/jacocoant.jar"/>
        </taskdef>

        <jacoco:coverage destfile="build/jacoco.exec" append="false">
            <junit printsummary="yes" fork="yes" forkmode="once" haltonerror="true" haltonfailure="true
                <jvmarg value="-Djava.awt.headless=true"/>
                <classpath refid="run-tests-classpath" />
                <formatter type="plain" usefile="false"/>
                <formatter type="xml"/>
                <batchtest todir="${build-tests-report}">
                    <fileset dir="${src-test}">
                        <include name="**/*Test.java"/>
                        <include name="**/Test*.java"/>
                    </fileset>
                    <fileset dir="functional_tests">
                        <include name="**/*Test.java"/>
                        <include name="**/Test*.java"/>
                    </fileset>
                </batchtest>
            </junit>
        </jacoco:coverage>
        <jacoco:report>
            <executiondata>
                <file file="build/jacoco.exec" />
            </executiondata>
            <structure name="${ant.project.name}">
                <classfiles>
                    <fileset dir="${build-classes}" />
                </classfiles>
                <sourcefiles encoding="UTF-8">
                    <fileset dir="${src}" />
                </sourcefiles>
            </structure>
            <html destdir="${build-coverage}"/>
            <xml destfile="${build-coverage}/jacoco.xml" />
        </jacoco:report>
    </target>

</project>
```

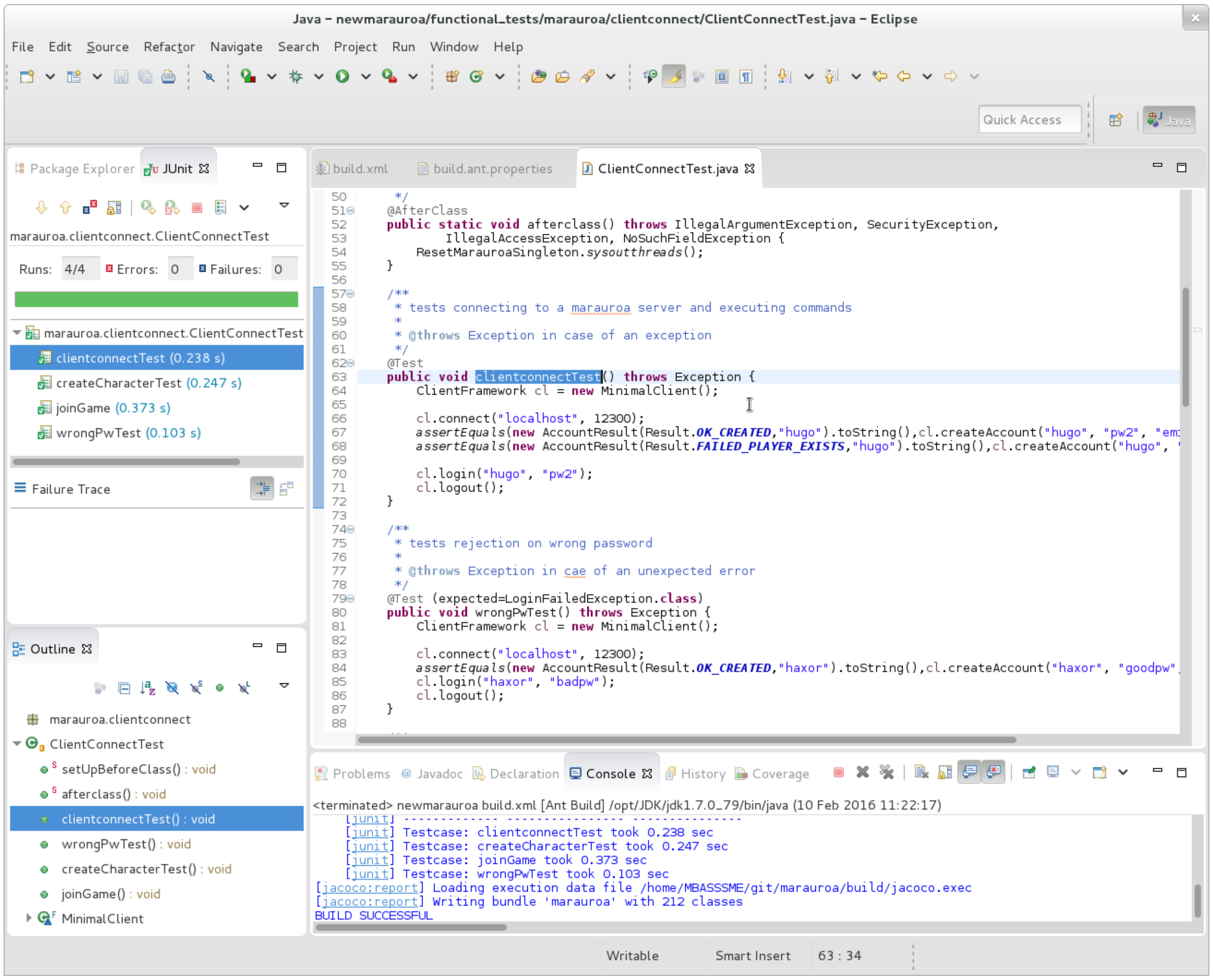
We can see that there were four test cases in this class:

`clientconnectTest`, `createCharacterTest`, `joinGame` and `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 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:


Java - newmarauroa/functional_tests/marauroa/clientconnect/ClientConnectTest.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Package Explorer
JUnit
marauroa.clientconnect.ClientConnectTest
Runs: 4/4 Errors: 0 Failures: 0
marauroa.clientconnect.ClientConnectTest
clientconnectTest (0.238 s)
createCharacterTest (0.247 s)
joinGame (0.373 s)
wrongPwTest (0.103 s)
Failure Trace
Outline
marauroa.clientconnect
ClientConnectTest
setUpBeforeClass() : void
afterclass() : void
clientconnectTest() : void
wrongPwTest() : void
createCharacterTest() : void
joinGame() : void
MinimalClient
build.xml
build.ant.properties
ClientConnectTest.java
Problems
Javadoc
Declaration
Console
History
Coverage
Writable
Smart Insert
63 : 34

```
     */
    @AfterClass
    public static void afterclass() throws IllegalArgumentException, SecurityException,
            IllegalAccessException, NoSuchFieldException {
        ResetMarauroaSingleton.sysoutthreads();
    }

    /**
     * tests connecting to a marauroa server and executing commands
     *
     * @throws Exception in case of an exception
     */
    @Test
    public void clientconnectTest() throws Exception {
        ClientFramework cl = new MinimalClient();

        cl.connect("localhost", 12300);
        assertEquals(new AccountResult(Result.OK_CREATED,"hugo").toString(),cl.createAccount("hugo", "pw2", "em
        assertEquals(new AccountResult(Result.FAILED_PLAYER_EXISTS,"hugo").toString(),cl.createAccount("hugo", '

        cl.login("hugo", "pw2");
        cl.logout();
    }

    /**
     * tests rejection on wrong password
     *
     * @throws Exception in cae of an unexpected error
     */
    @Test (expected=LoginFailedException.class)
    public void wrongPwTest() throws Exception {
        ClientFramework cl = new MinimalClient();

        cl.connect("localhost", 12300);
        assertEquals(new AccountResult(Result.OK_CREATED,"haxor").toString(),cl.createAccount("haxor", "goodpw",
        cl.login("haxor", "badpw");
        cl.logout();
    }
```

```
<terminated> newmarauroa build.xml [Ant Build] /opt/JDK/jdk1.7.0_79/bin/java (10 Feb 2016 11:22:17)
    [junit] ------------- ---------------- ---------------
    [junit] Testcase: clientconnectTest took 0.238 sec
    [junit] Testcase: createCharacterTest took 0.247 sec
    [junit] Testcase: joinGame took 0.373 sec
    [junit] Testcase: wrongPwTest took 0.103 sec
[jacoco:report] Loading execution data file /home/MBASSSME/git/marauroa/build/jacoco.exec
[jacoco:report] Writing bundle 'marauroa' with 212 classes
BUILD SUCCESSFUL
```

You can see that the test is short, and quite descriptive. This is typical of 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.

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`.

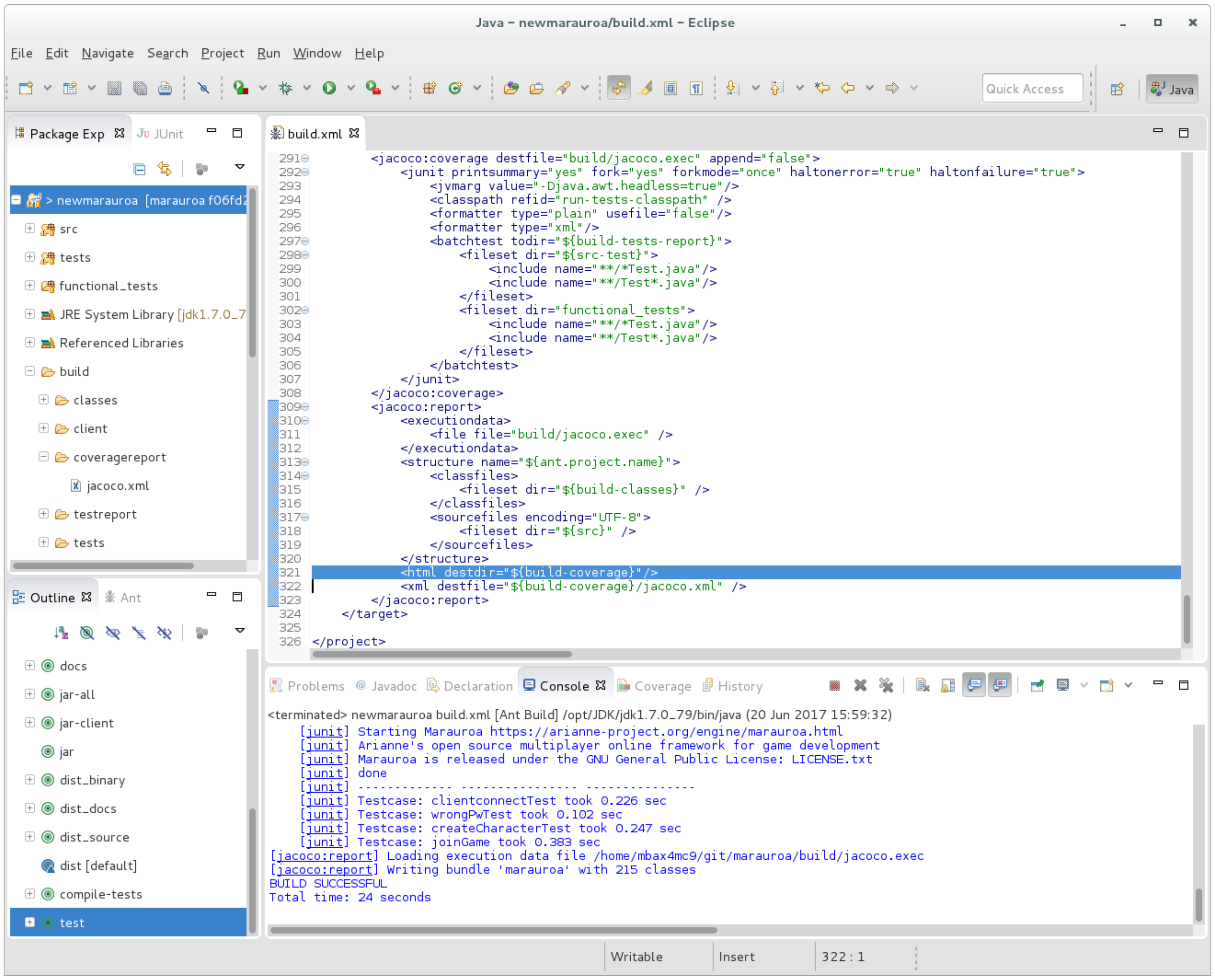
Important Note: Don't try to open the jacoco.xml file in Eclipse! It is huge and Eclipse will spend a lot of time trying (and probably failing) to grab enough memory for it. If you want to see what it contains, then use a lightweight text editor or a command like "head -c" from the command line.

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. The screenshot below shows how the edited build file should look, with the new line highlighted.



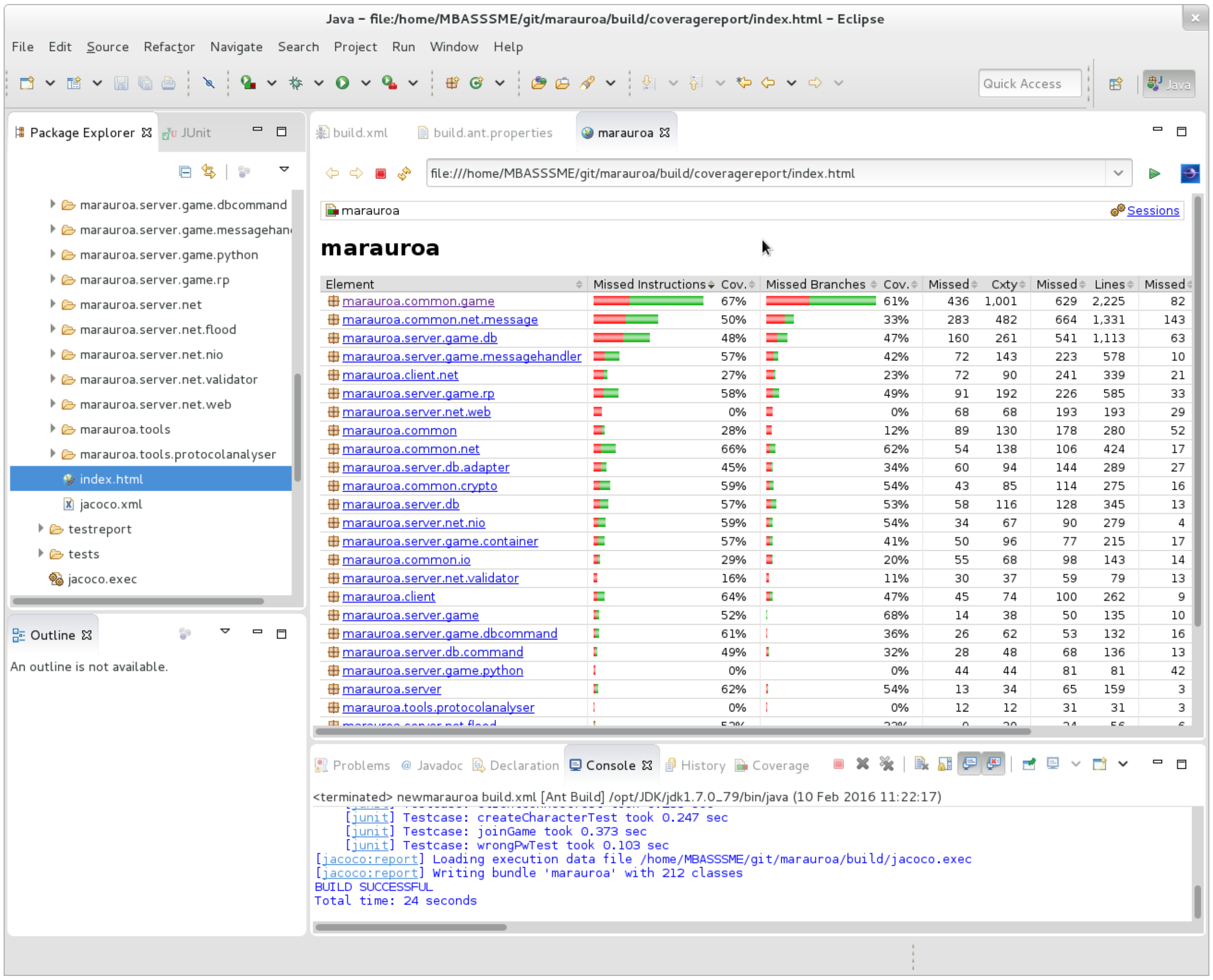
```
        <jacoco:coverage destfile="build/jacoco.exec" append="false">
            <junit printsummary="yes" fork="yes" forkmode="once" haltonerror="true" haltonfailure="true">
                <jvmarg value="-Djava.awt.headless=true"/>
                <classpath refid="run-tests-classpath" />
                <formatter type="plain" usefile="false"/>
                <formatter type="xml"/>
                <batchtest todir="${build-tests-report}">
                    <fileset dir="${src-test}">
                        <include name="**/*Test.java"/>
                        <include name="**/Test*.java"/>
                    </fileset>
                    <fileset dir="functional_tests">
                        <include name="**/*Test.java"/>
                        <include name="**/Test*.java"/>
                    </fileset>
                </batchtest>
            </junit>
        </jacoco:coverage>
        <jacoco:report>
            <executiondata>
                <file file="build/jacoco.exec" />
            </executiondata>
            <structure name="${ant.project.name}">
                <classfiles>
                    <fileset dir="${build-classes}" />
                </classfiles>
                <sourcefiles encoding="UTF-8">
                    <fileset dir="${src}" />
                </sourcefiles>
            </structure>
            <html destdir="${build-coverage}"/>
            <xml destfile="${build-coverage}/jacoco.xml" />
        </jacoco:report>
    </target>

</project>
```

```
<terminated> newmarauroa build.xml [Ant Build] /opt/JDK/jdk1.7.0_79/bin/java (20 Jun 2017 15:59:32)
    [junit] Starting Marauroa https://arianne-project.org/engine/marauroa.html
    [junit] Arianne's open source multiplayer online framework for game development
    [junit] Marauroa is released under the GNU General Public License: LICENSE.txt
    [junit] done
    [junit] ------------- ---------------- ---------------
    [junit] Testcase: clientconnectTest took 0.226 sec
    [junit] Testcase: wrongPwTest took 0.102 sec
    [junit] Testcase: createCharacterTest took 0.247 sec
    [junit] Testcase: joinGame took 0.383 sec
[jacoco:report] Loading execution data file /home/mbax4mc9/git/marauroa/build/jacoco.exec
[jacoco:report] Writing bundle 'marauroa' with 215 classes
BUILD SUCCESSFUL
Total time: 24 seconds
```

The line to add has been highlighted in the editor window here. 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 Oxygen 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.)



marauroa

marauroa

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed
marauroa.common.game		67%		61%	436	1,001	629	2,225	82
marauroa.common.net.message		50%		33%	283	482	664	1,331	143
marauroa.server.game.db		48%		47%	160	261	541	1,113	63
marauroa.server.game.messagehandler		57%		42%	72	143	223	578	10
marauroa.client.net		27%		23%	72	90	241	339	21
marauroa.server.game.rp		58%		49%	91	192	226	585	33
marauroa.server.net.web		0%		0%	68	68	193	193	29
marauroa.common		28%		12%	89	130	178	280	52
marauroa.common.net		66%		62%	54	138	106	424	17
marauroa.server.db.adapter		45%		34%	60	94	144	289	27
marauroa.common.crypto		59%		54%	43	85	114	275	16
marauroa.server.db		57%		53%	58	116	128	345	13
marauroa.server.net.nio		59%		54%	34	67	90	279	4
marauroa.server.game.container		57%		41%	50	96	77	215	17
marauroa.common.io		29%		20%	55	68	98	143	14
marauroa.server.net.validator		16%		11%	30	37	59	79	13
marauroa.client		64%		47%	45	74	100	262	9
marauroa.server.game		52%		68%	14	38	50	135	10
marauroa.server.game.dbcommand		61%		36%	26	62	53	132	16
marauroa.server.db.command		49%		32%	28	48	68	136	13
marauroa.server.game.python		0%		0%	44	44	81	81	42
marauroa.server		62%		54%	13	34	65	159	3
marauroa.tools.protocolanalyser		0%		0%	12	12	31	31	3

```
<terminated> newmarauroa build.xml [Ant Build] /opt/JDK/jdk1.7.0_79/bin/java (10 Feb 2016 11:22:17)
    [junit] Testcase: createCharacterTest took 0.247 sec
    [junit] Testcase: joinGame took 0.373 sec
    [junit] Testcase: wrongPwTest took 0.103 sec
[jacoco:report] Loading execution data file /home/MBASSSME/git/marauroa/build/jacoco.exec
[jacoco:report] Writing bundle 'marauroa' with 212 classes
BUILD SUCCESSFUL
Total time: 24 seconds
```

This report 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 first package, `marauroa.common.game`, we can see from the report that the test suite executed 67% of the instructions in the package. (An instruction, here, is a single Java byte code instruction.) But, 33% (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 61% 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:

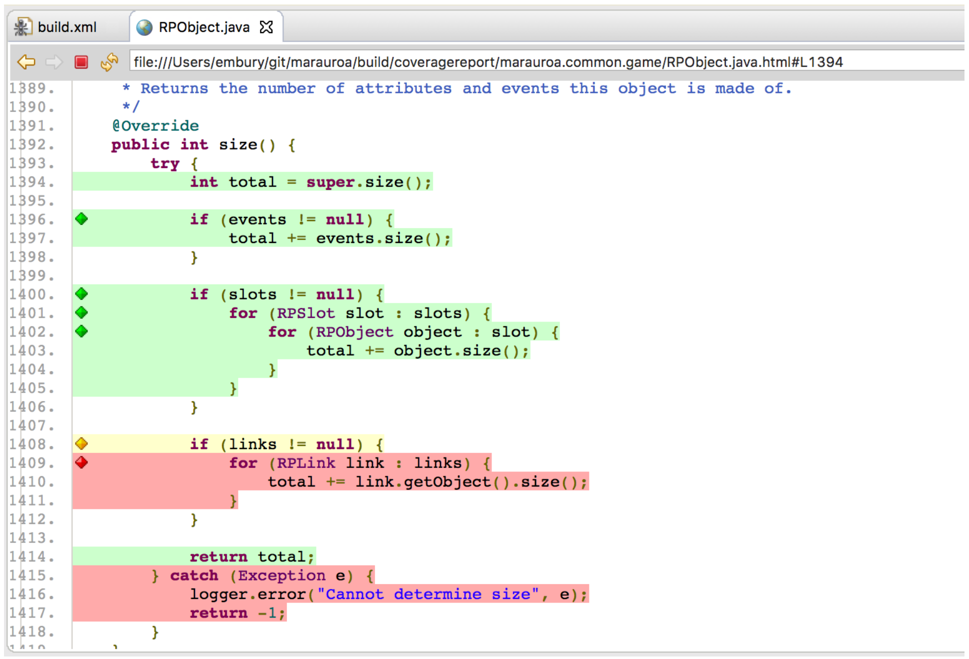
```
int x = Math.random();
System.out.println(x);
```

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, here is the detailed coverage report for the method `marauroa.common.game.RPObject.size()`:


build.xml
RPObject.java
file:///Users/embury/git/marauroa/build/coveragereport/marauroa.common.game/RPObject.java.html#L1394

```
      * Returns the number of attributes and events this object is made of.
      */
    @Override
    public int size() {
        try {
            int total = super.size();

            if (events != null) {
                total += events.size();
            }

            if (slots != null) {
                for (RPSlot slot : slots) {
                    for (RPObject object : slot) {
                        total += object.size();
                    }
                }
            }

            if (links != null) {
                for (RPLink link : links) {
                    total += link.getObject().size();
                }
            }

            return total;
        } catch (Exception e) {
            logger.error("Cannot determine size", e);
            return -1;
        }
```

Here, 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 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.

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.

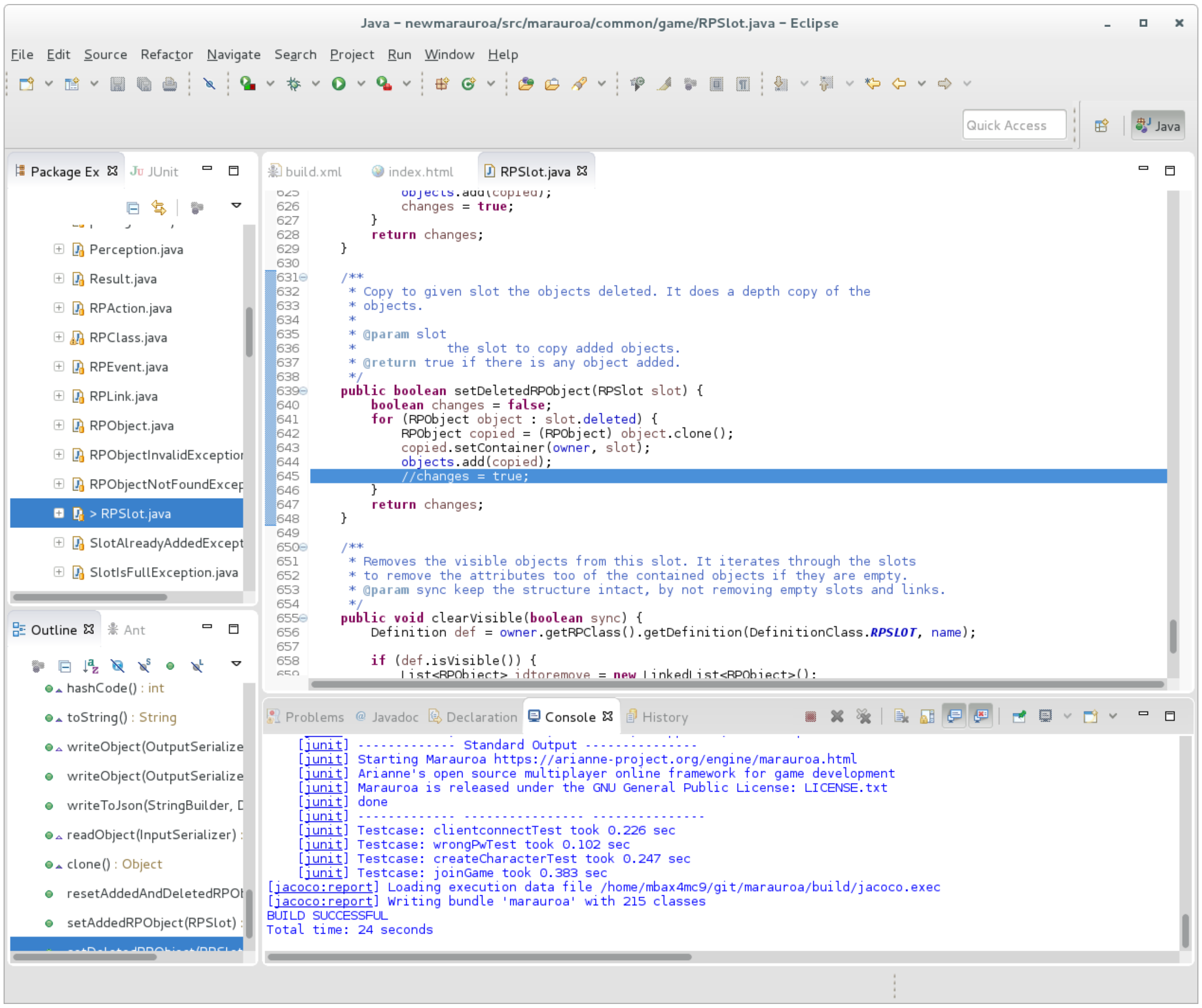
For example, let's make a small change to the method:

```
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 645, as shown below:

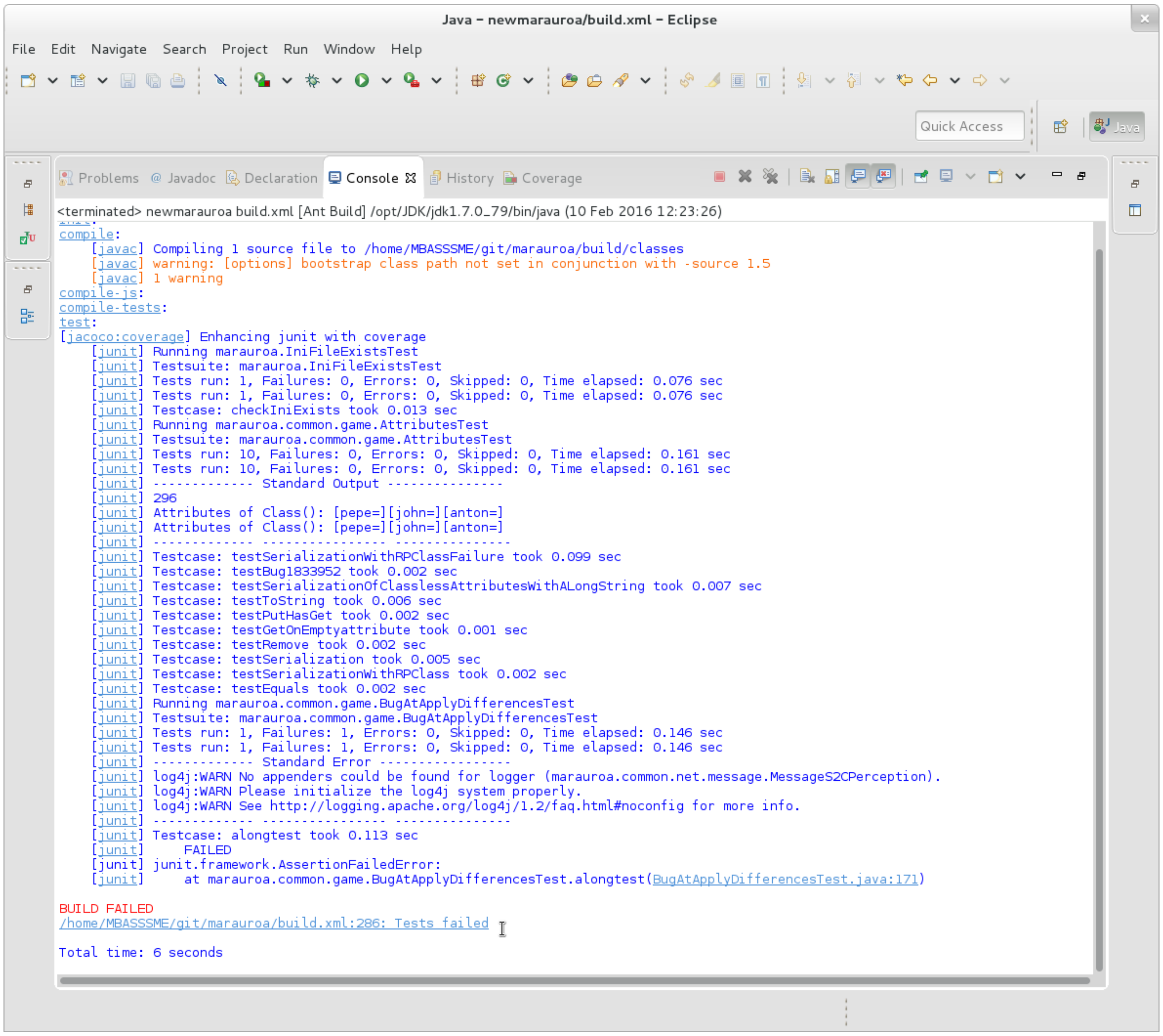


```
            objects.add(copied);
            changes = true;
        }
        return changes;
    }

    /**
     * Copy to given slot the objects deleted. It does a depth copy of the
     * objects.
     *
     * @param slot
     *            the slot to copy added objects.
     * @return true if there is any object added.
     */
    public boolean setDeletedRPObject(RPSlot slot) {
        boolean changes = false;
        for (RPObject object : slot.deleted) {
            RPObject copied = (RPObject) object.clone();
            copied.setContainer(owner, slot);
            objects.add(copied);
            //changes = true;
        }
        return changes;
    }

    /**
     * Removes the visible objects from this slot. It iterates through the slots
     * to remove the attributes too of the contained objects if they are empty.
     * @param sync keep the structure intact, by not removing empty slots and links.
     */
    public void clearVisible(boolean sync) {
        Definition def = owner.getRPClass().getDefinition(DefinitionClass.RPSLOT, name);

        if (def.isVisible()) {
            List<RPObject> idtoremove = new LinkedList<RPObject>();
```

```
    [junit] ------------- Standard Output ---------------
    [junit] Starting Marauroa https://arianne-project.org/engine/marauroa.html
    [junit] Arianne's open source multiplayer online framework for game development
    [junit] Marauroa is released under the GNU General Public License: LICENSE.txt
    [junit] done
    [junit] ------------- ---------------- ---------------
    [junit] Testcase: clientconnectTest took 0.226 sec
    [junit] Testcase: wrongPwTest took 0.102 sec
    [junit] Testcase: createCharacterTest took 0.247 sec
    [junit] Testcase: joinGame took 0.383 sec
[jacoco:report] Loading execution data file /home/mbax4mc9/git/marauroa/build/jacoco.exec
[jacoco:report] Writing bundle 'marauroa' with 215 classes
BUILD SUCCESSFUL
Total time: 24 seconds
```

Save the file.

Then run the test build target again. You should see output similar to this.


Java - newmarauroa/build.xml - Eclipse
File Edit Navigate Search Project Run Window Help
Quick Access
Java
Problems @ Javadoc Declaration Console History Coverage

```
<terminated> newmarauroa build.xml [Ant Build] /opt/JDK/jdk1.7.0_79/bin/java (10 Feb 2016 12:23:26)
compile:
    [javac] Compiling 1 source file to /home/MBASSSME/git/marauroa/build/classes
    [javac] warning: [options] bootstrap class path not set in conjunction with -source 1.5
    [javac] 1 warning
compile-js:
compile-tests:
test:
[jacoco:coverage] Enhancing junit with coverage
    [junit] Running marauroa.IniFileExistsTest
    [junit] Testsuite: marauroa.IniFileExistsTest
    [junit] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.076 sec
    [junit] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.076 sec
    [junit] Testcase: checkIniExists took 0.013 sec
    [junit] Running marauroa.common.game.AttributesTest
    [junit] Testsuite: marauroa.common.game.AttributesTest
    [junit] Tests run: 10, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.161 sec
    [junit] Tests run: 10, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.161 sec
    [junit] ------------- Standard Output ---------------
    [junit] 296
    [junit] Attributes of Class(): [pepe=][john=][anton=]
    [junit] Attributes of Class(): [pepe=][john=][anton=]
    [junit] ------------- ---------------- ---------------
    [junit] Testcase: testSerializationWithRPClassFailure took 0.099 sec
    [junit] Testcase: testBug1833952 took 0.002 sec
    [junit] Testcase: testSerializationOfClasslessAttributesWithALongString took 0.007 sec
    [junit] Testcase: testToString took 0.006 sec
    [junit] Testcase: testPutHasGet took 0.002 sec
    [junit] Testcase: testGetOnEmptyattribute took 0.001 sec
    [junit] Testcase: testRemove took 0.002 sec
    [junit] Testcase: testSerialization took 0.005 sec
    [junit] Testcase: testSerializationWithRPClass took 0.002 sec
    [junit] Testcase: testEquals took 0.002 sec
    [junit] Running marauroa.common.game.BugAtApplyDifferencesTest
    [junit] Testsuite: marauroa.common.game.BugAtApplyDifferencesTest
    [junit] Tests run: 1, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.146 sec
    [junit] Tests run: 1, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.146 sec
    [junit] ------------- Standard Error -----------------
    [junit] log4j:WARN No appenders could be found for logger (marauroa.common.net.message.MessageS2CPerception).
    [junit] log4j:WARN Please initialize the log4j system properly.
    [junit] log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
    [junit] ------------- ---------------- ---------------
    [junit] Testcase: alongtest took 0.113 sec
    [junit]     FAILED
    [junit] junit.framework.AssertionFailedError:
    [junit]     at marauroa.common.game.BugAtApplyDifferencesTest.alongtest(BugAtApplyDifferencesTest.java:171)

BUILD FAILED
/home/MBASSSME/git/marauroa/build.xml:286: Tests failed

Total time: 6 seconds
```

Notice the important line at the bottom in red. The build failed, because one or more of the tests failed. If you scroll through the build output, you'll see that this failure is not just caused by the failing test we saw at the beginning. Three new test failures are being reported. To see the details, open up the JUnit results file:

```
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 neighbour 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 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.

Ask Suzanne if you are stuck!