Test Driven Development with Java

Lab Exercises

Table of Contents

Introduction

This workbook is designed to accompany Wakaleo Consulting's Test Driven Development for Java training course. It consists of a sequence of lab exercises which will introduce the concepts of Test Driven Development for Java developers. These lab exercises should be completed in sequence in the presence of an instructor. The instructor will distribute the software and code necessary to complete these exercises. If, at any time during the lab exercise, you encounter problems with your software installation or if you do not understand any of the instructions, please ask your instructor for help.

This lab manual and training materials assume that you have been supplied with the necessary software prerequisites. Your instructor will distribute installation media or provide instructions for downloading this material from the Internet. The supporting software used in this course are:

- Java Development Kit version 1.6.0_15 (JDK)
- Git
- Intellij Community Edition 11.0.x
- Apache Maven 3.0.3

The labs assume that a recent version of Java (JDK 1.6.0 or higher), has been installed on the workstations. Git is not strictly necessary, but you will need it if you want to install the source code for the lab projects on your own machine.

Lab 1. Setting up your environment

Goal

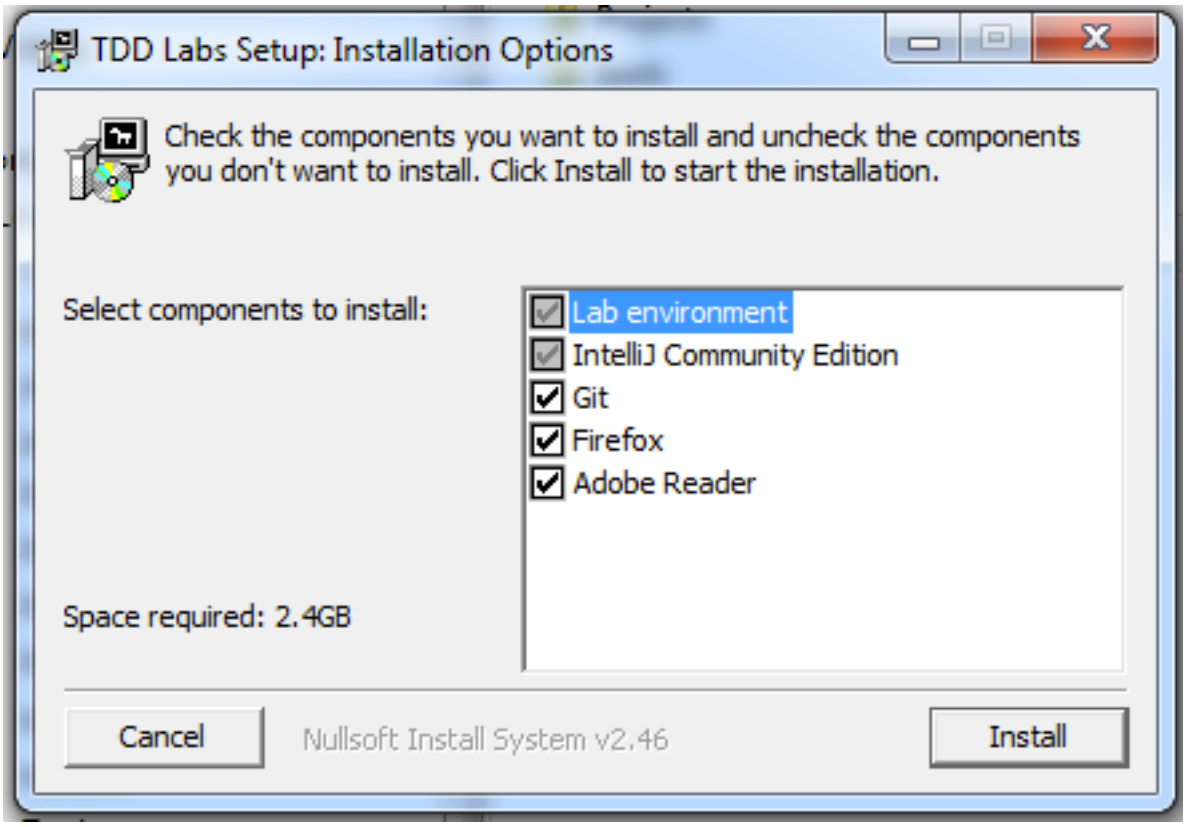
The aim of this lab exercise is to set up your workstation with the tools required for this workshop. After completing this exercise, you should be able to set up and configure these tools on other workstations.

Agenda

- Install the lab tools and environment
- Open the gameoflife lab project in IntelliJ
- Verify that the gameoflife application builds and runs correctly

Lab Exercises

In this exercise, you are going to install the lab exercises onto your workstation. The lab exercises are bundled into a Windows installer, which includes Java, IntelliJ community edition, Maven, as well as Firefox, Git and Adobe Reader. If you already have Firefox and Adobe Reader installed, you can choose not to install these packages:

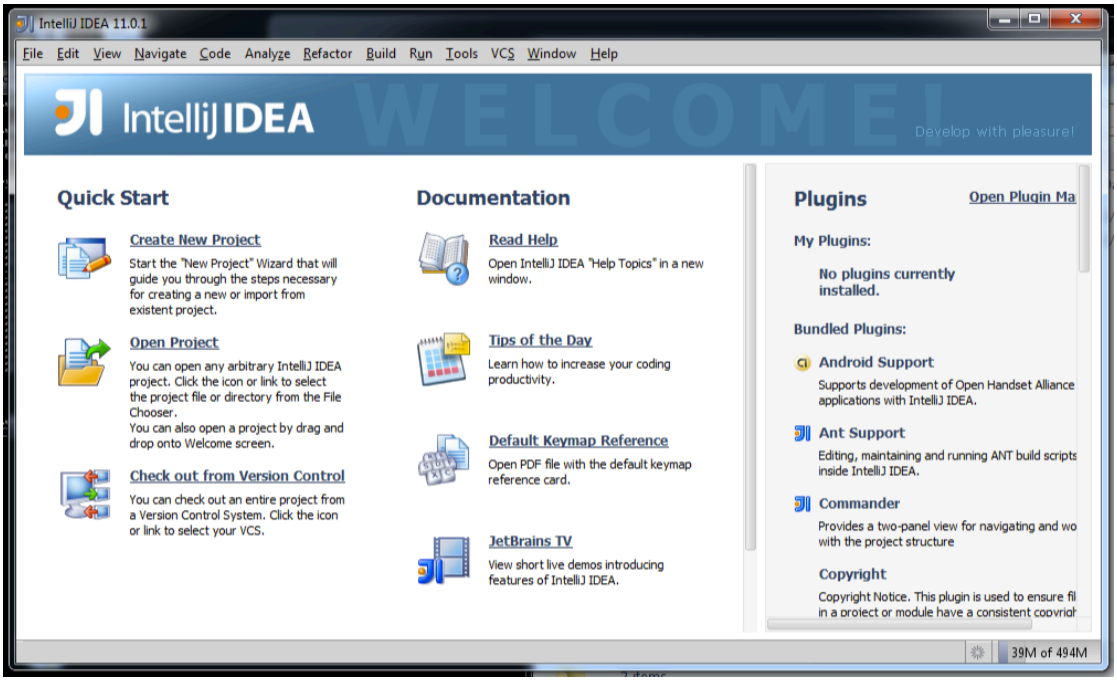


Step 6: Installing the lab project

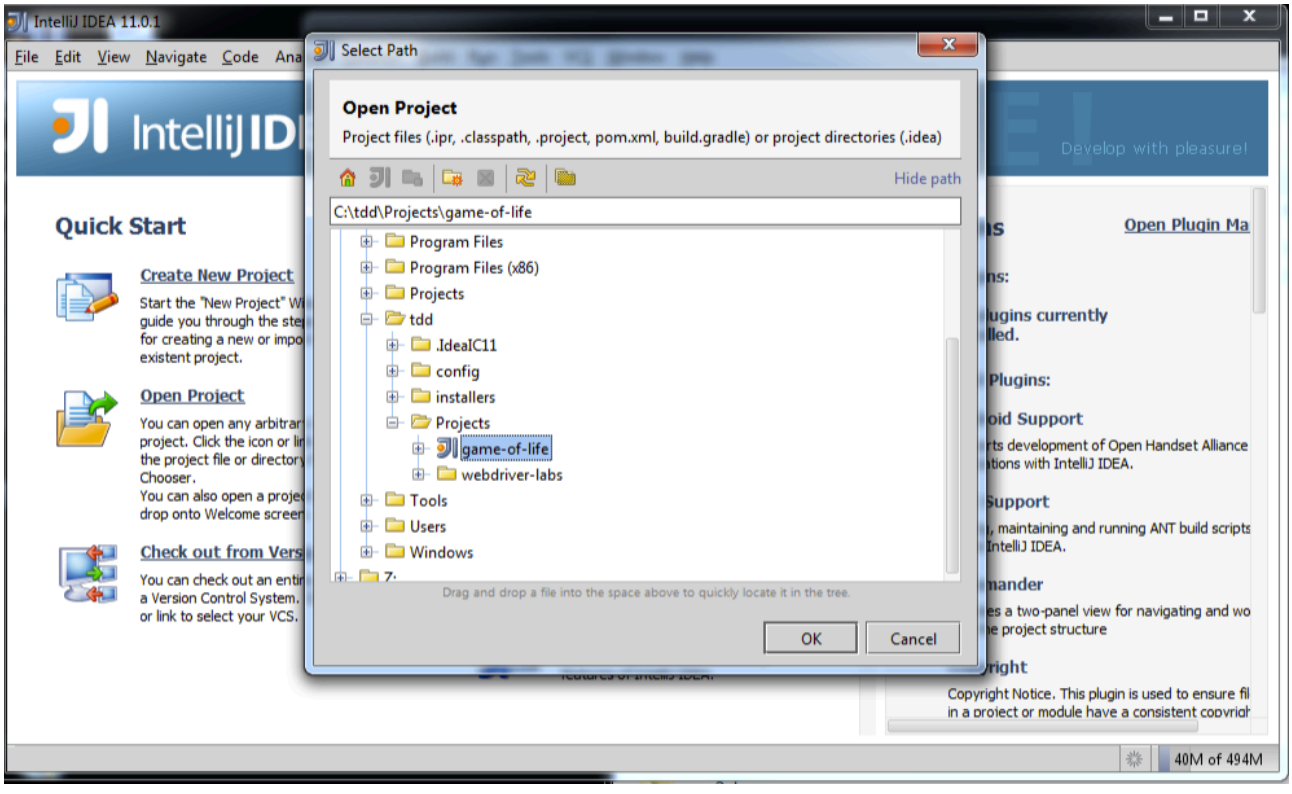
During this workshop, we will be working on a sample application called Game Of Life. The initial source code for this project will be provided.

Step 7: Open the game-of-life project in IntelliJ

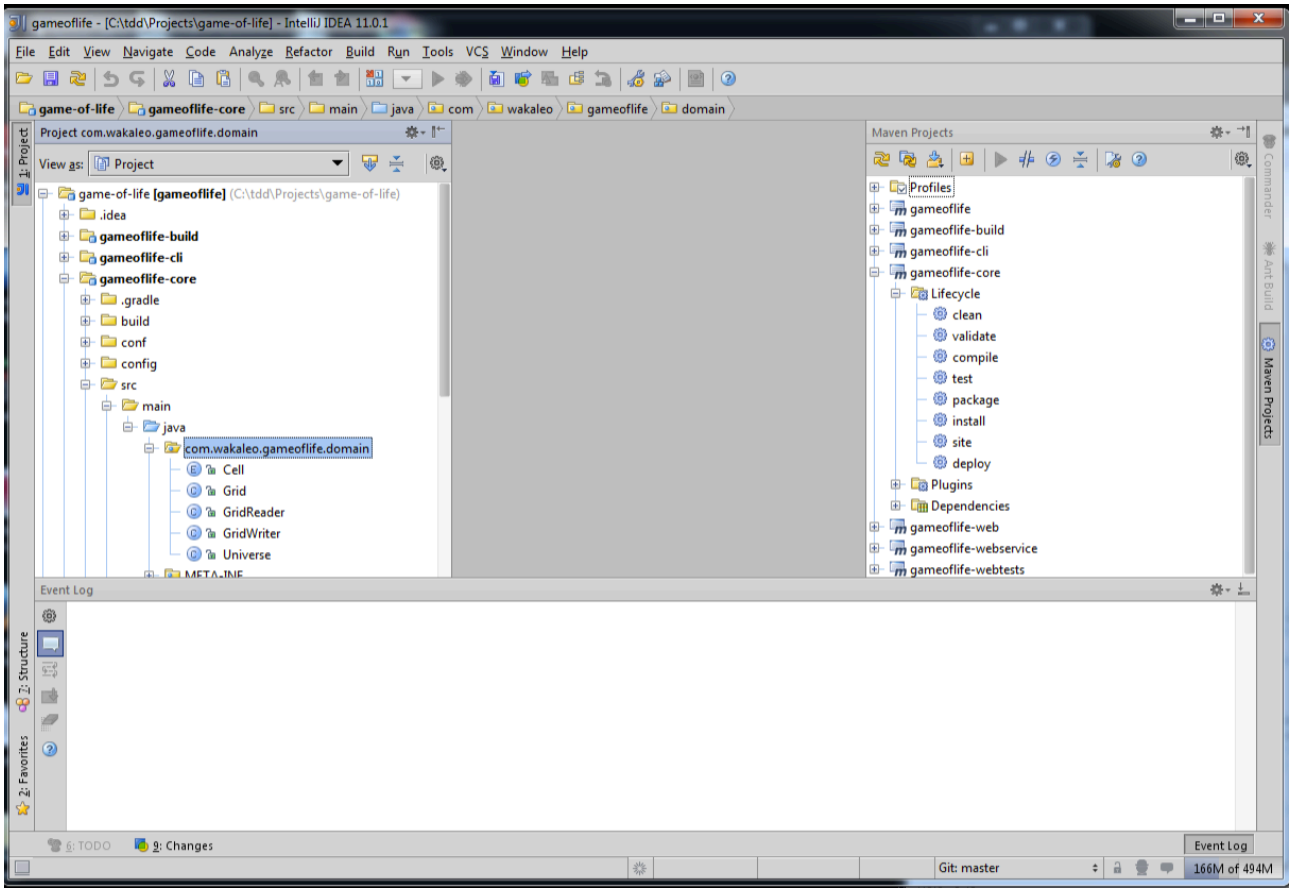
Start up IntelliJ and choose 'Open Project'



The Game of Life project should open automatically. If not, just choose the 'game-of-life' project in `C:\tdd\Projects\game-of-life`:



You should now see the Game Of Life project in IntelliJ



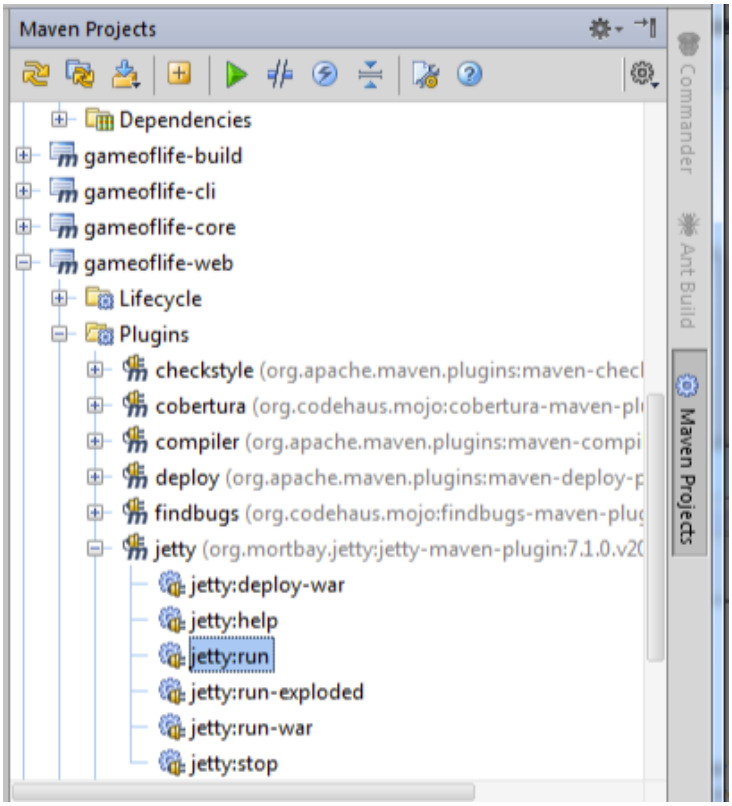
Step 7: Verify that the application builds and runs correctly

Now we will check that the application builds and runs correctly. The easiest way to do this using the 'Maven Projects' view in Intellij, clicking on the 'install' entry in the 'Lifecycle' section of the 'gameoflife' project. Alternatively, you can do this from the command line in the gameoflife root directory:

```
C:\TDD\Projects\game-of-life> mvn install
```

This should build and test the application. To see the application running, you can run it using Jetty either from IntelliJ or on the command line.

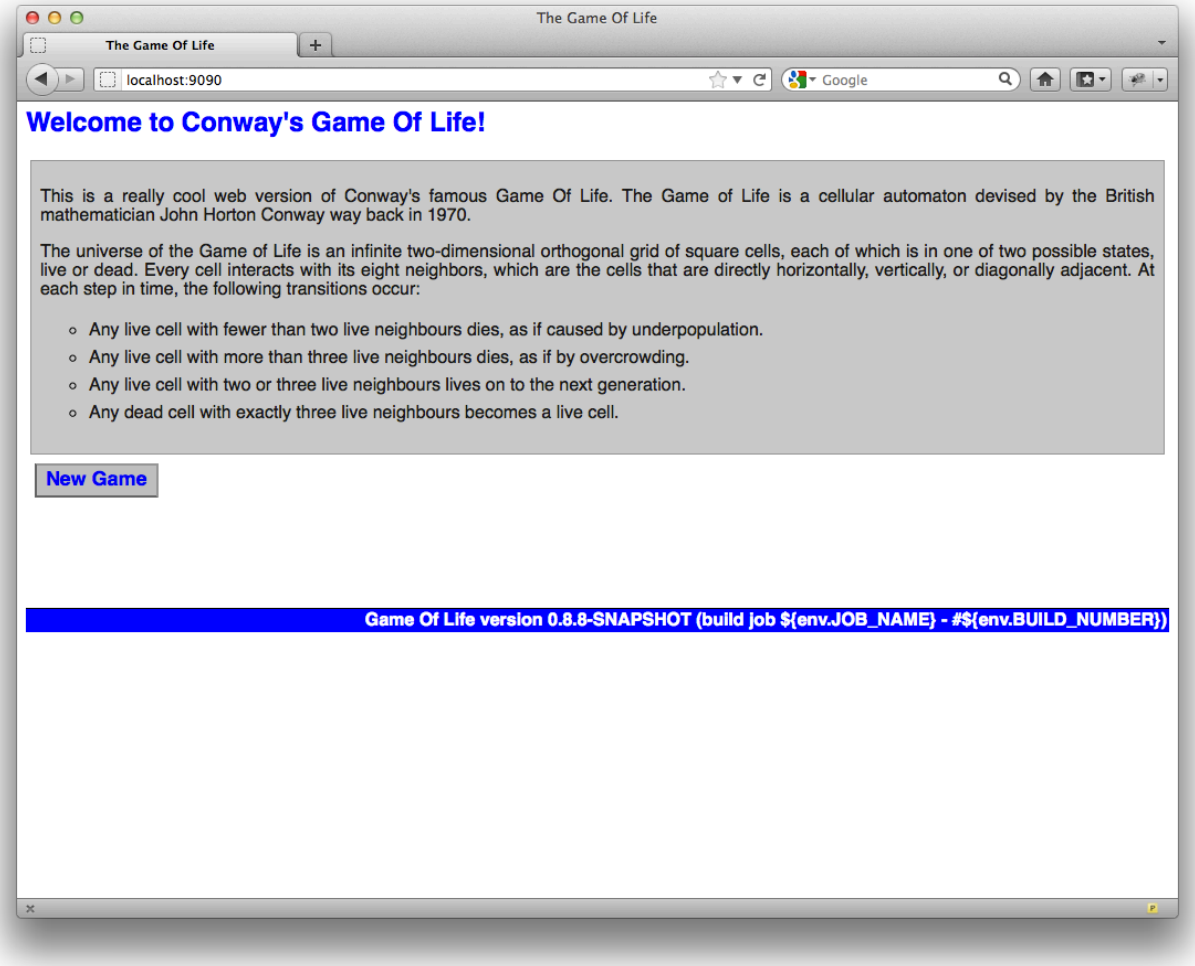
In the 'Maven Projects' view in IntelliJ, open the 'Plugins' section in the 'gameoflife-web' project and click on 'jetty:run':



Or, from the command line, go to the gameoflife-web directory and run the Maven Jetty plugin as follows:

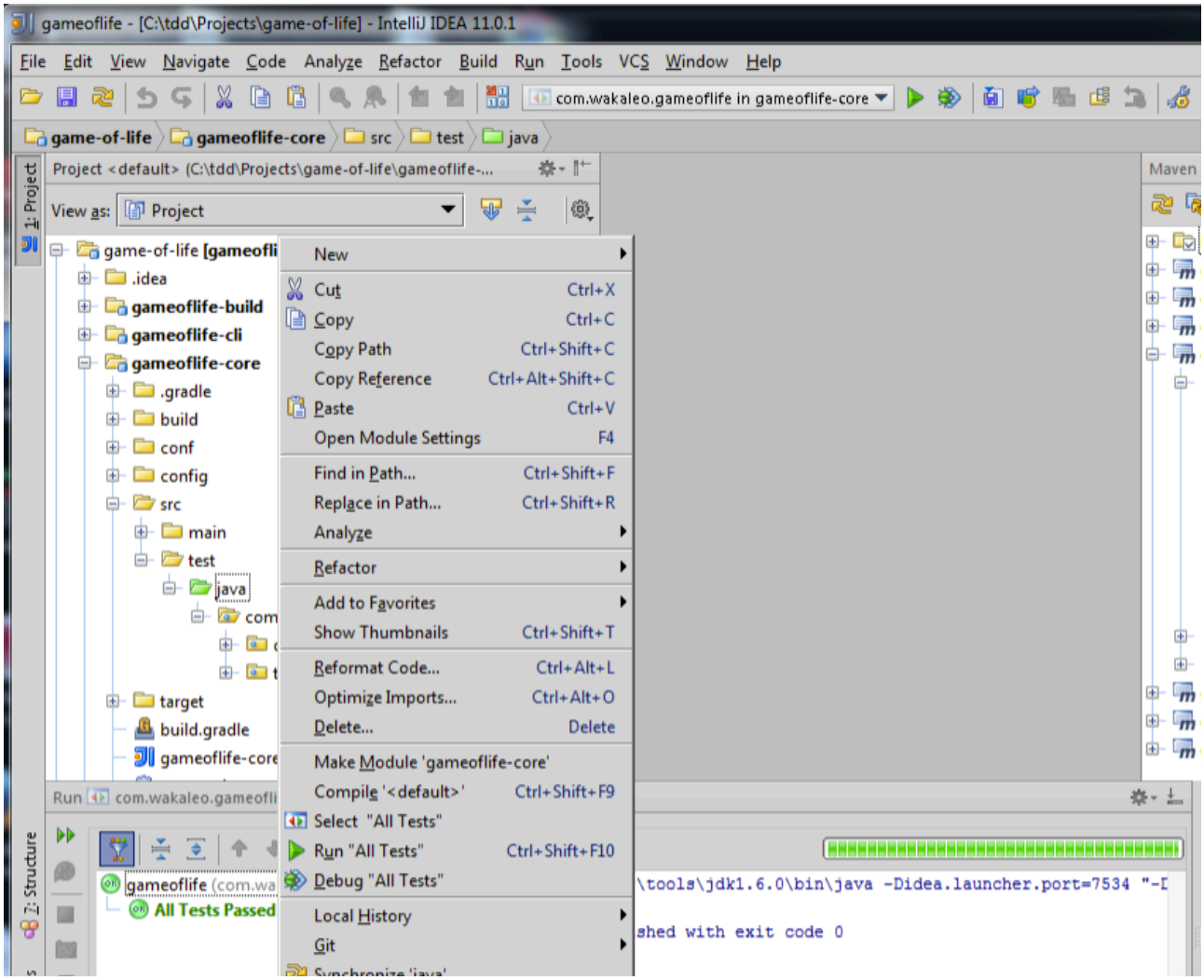
```
C:\TDD\Projects\game-of-life> cd gameoflife-web
C:\TDD\Projects\game-of-life\gameoflife-web> mvn jetty:run
```

The application will now be running on port 9090:



Now experiment with the application to see how it works.

You should check that the tests run correctly. We will be initially working on the core (domain) module. To run all of the unit tests in the core module, go to the gameoflife-core module and select “test” in the Maven Lifecycles, or by selecting the java package in the src/test directory of ‘gameoflife-core’ and then selecting ‘Run All Tests’ in the contextual menu (see below).



Now that you have seen the application build and run successfully, we will concentrate on the gameoflife-core module for the remainder of this lab.

Lab 2 - Getting Started with TDD

Goal

During the following few labs we will be working on adding a 'History' feature to the Game of Life application. The idea is to allow users to record their previous games and to be able to retrieve and replay them.

User Requirement

"As a 'Game Of Life' researcher
I want to be able to store my previous simulations
So that I can replay and compare them"

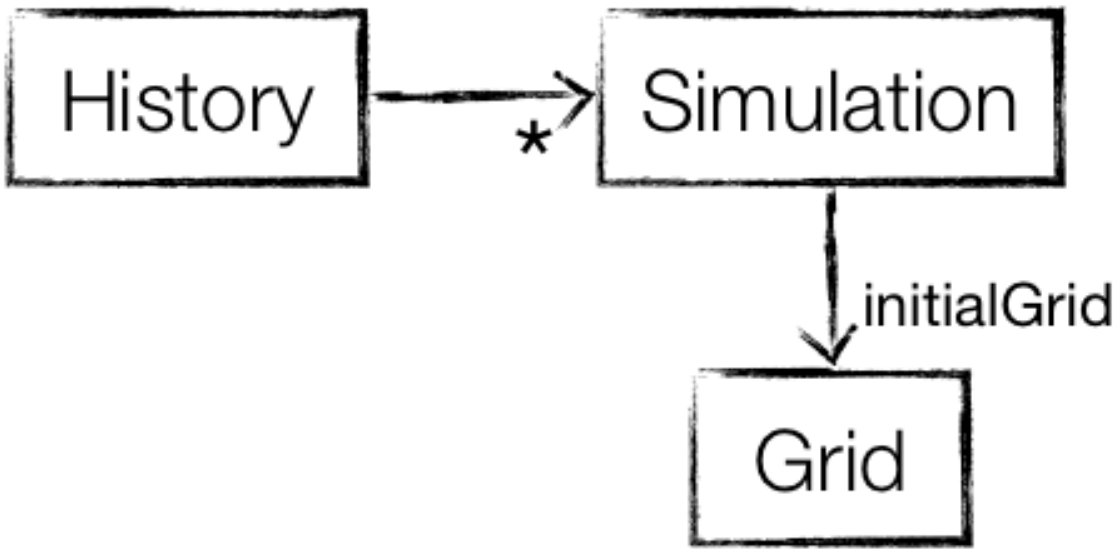
The aim of this first lab exercise will be to work on the domain layer for this feature.

Before starting, as a group (or several groups) draw up a list of high-level acceptance criteria for this requirement. Some examples might be:

- A researcher should be able to see a list of previously run simulations
- A researcher should be able to replay a past simulation
- Simulations should be listed in chronological order

and so forth.

Once this is done, we will move to working on the domain level (for simplicity, we will bypass the acceptance tests at this stage). Suppose that we have run a workshop and come up with the following domain model:



In the current application, a game of life takes the form of a sequence of grids. Since the life of a grid is deterministic and depends on the initial state of the grid, all we really need to store is the state of the universe at the start of the simulation (i.e. the initial grid), and the date that the simulation was executed.

In this lab, we will use TDD to implement this domain model.

Lab Exercises

Step 1: Introducing the concept of the Simulation

In the current implementation, an instance of the Universe class is used to keep track of the state of the cells over the life of the grid. However, for our History feature, we have decided to extend this concept and introduce the idea of a Simulation. A Simulation represents a particular run of the Game of Life, starting with a given grid.

Note that, in the interests of simplicity, we will be using a Bottom-Up approach here, based on the domain model we elaborated earlier. A more typical real-world approach would be to write the tests for the History class first, and to determine the requirements of the Simulation class once the exact behaviour of the History class has been established.

For now, draw up a list of tests that can be used to design the behaviour of this Simulation domain object. Some possible test candidates might be:

- A simulation should record the initial state of the grid

- A simulation should record the date of execution

and so on.

You may also include some more technical requirements, such as:

- A simulation should be immutable

and so on.

Now implement your tests, using the TDD process discussed in the course:

Step 1.a: Write the test

Create a new test class with a narrative-style name (for example, something like `WhenWeRecordASimulatedGameOfLife`). Next, start off with the first test on your list (or add them all, but as @Ignored tests), and implement the test. When you save the completed test, you will need to add the new domain class and the empty methods you have just dreamt up in the test case. Then run the JUnit tests to ensure that this new test is executed, and fails.

Step 1.b: Write the code

Now implement the code and rerun the tests.

Step 1.c: Refactor

Always check your code after you have written it to see if there are ways you might improve it and make it more readable and maintainable. This will of course depend on your code, and there may be nothing to do at certain stages (particularly at the start)

Step 1.d: Check Coverage

Run your tests through EclEmma to verify the test coverage. It should not be lower than the previous value.

Step 2 (optional): Implement the History class

Apply the same approach to implement the History class. Define a list of tests that can be used to design the behaviour of this History domain object. The idea of the History class will be to store and retrieve played simulations. To develop this idea, you may want to investigate ideas such as:

- What services is the history object expected to provide?
- What tests could verify that these services are provided?
- How will the other concepts discussed above interact with the History object?

Next, define a list of tests that can be used to design the behaviour of this History domain object. This might include:

- Should save a simulation for future reference

- Should be able to list all recorded simulations

Step 2.a: Write the test

Create a new test class with a narrative-style name (for example, something like `WhenYouRecordASimulationInTheSimulationHistory`). Next, start off with the first test on your list (or add them all, but as @Ignored tests), and implement the test. When you save the completed test, you will need to add the new domain class and the empty methods you have just dreamt up in the test case. Then run the JUnit tests to ensure that this new test is executed, and fails.

Step 2.b: Write the code

Now implement the code and rerun the tests.

Step 2.c: Refactor

Always check your code after you have written it to see if there are ways you might improve it and make it more readable and maintainable. This will of course depend on your code, and there may be nothing to do at certain stages (particularly at the start)

Step 2.d: Check Coverage

Run your tests through EclEmma to verify the test coverage. It should not be lower than the previous value.

Lab 3 - Working with fixtures

Goal

In this exercise we will use TDD to add the following feature to our Simulation class:

- Simulations should be ordered by date of execution

Lab Exercises

Step 1: Define the tests

Define some technical acceptance criteria that would demonstrate that your Simulation class is ordered by the date of execution.

Step 2: Implement the tests

Create a new test class (called, for example, `WhenOrderingSimulations`) and implement the acceptance criteria as unit tests. Use fixture methods to define the Simulation objects to use in the tests in one place.

Step 3: Refactor

Now review your code to see if anything can be expressed more clearly.

Lab 4 - An introduction to Hamcrest asserts

We will be using Hamcrest asserts throughout the rest of this course. To get you started, here are a few basic exercises.

Step 1) Create a new test class called WhenUsingHamcrestAsserts, as follows:

```
import static org.junit.Assert.assertEquals;
import static org.hamcrest.Matchers.*;
import static org.hamcrest.MatcherAssert.assertThat;

import org.junit.Test;

public class WhenUsingHamcrestAsserts {
	@Test
	public void testAssertThatIs(){
		String color = "red";

	}
}
```

Complete the `testAssertThatIs()` method with a Hamcrest assert that checks that color is equal to “red”. Hint: use *assertThat()* method with the *is()* matcher.

Change the value of the color variable. What is the error message produced?

Step 2) Add and complete the following method using the *is* matcher. What advantages does it have over the equivalent assertEquals() expression?

```
	@Test
	public void testAssertThatIsForDoubles(){
		double expectedResult = 100.0;
		double calculatedResult = 10.0 * 10.0;
		assertThat(...)
	}
```

Step 3) Add and complete the following methods using the *isIn* matcher.

```
	@Test
	public void testAssertThatIsIn (){
		String[] colors = new String[] {"red","green","blue"};
		String color = "blue";
		assertThat(..);
	}
```

```
@Test
public void testAssertThatIsIn(){
	List<String> colors = Arrays.asList("red","green","blue");
	String color = "blue";
	assertThat(..);
}
```

Step 4) Add and complete the following method using the *not* and *isIn* matcher.

```
@Test
public void testAssertThatIsNot(){
	List<String> colors = Arrays.asList("red","green","blue");
	String color = "yellow";
	assertThat(...);
}
```

Step 5) Add and complete the following method using the *isOneOf* matcher. The test should verify that color is either “red”, “green” or “blue.

```
@Test
public void testAssertThatIsOneOfRedGreenBlue(){
	String color = "blue";
	assertThat(...);
}
```

Step 6) Add and complete the following method using the `notNullValue` matcher. The test should verify that color is not null

```
@Test
public void testAssertThatIsNotNull(){
	String color = "blue";
	assertThat(...);
}
```

Step 7) Add and complete the following method using the *not*, *hasItem* and *lessThan* matchers. The test should verify that the ages list contains no value that is less than 18.

```
@Test
public void testAssertThatNoneLessThan18(){
	List<Integer> ages = new ArrayList<Integer>();
	ages.add(20);
	ages.add(30);
	ages.add(40);
	assertThat(...);
}
```

Step 8) (optional) Write a custom matcher called hasSize(), that will work as illustrated in the following tests:

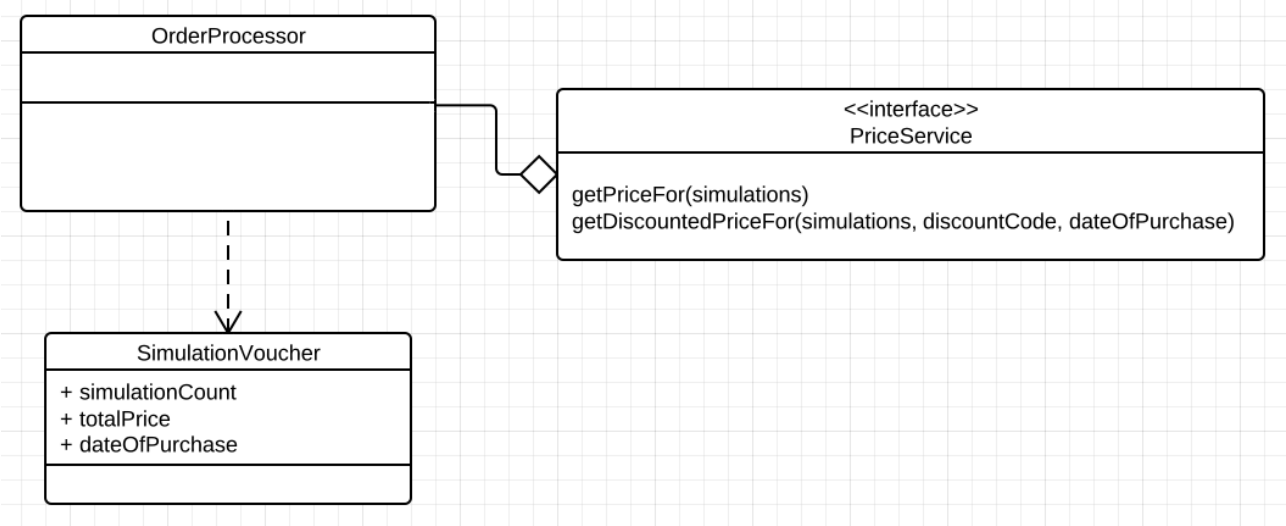
```
@Test
public void thehasSizeMatcherShouldMatchACollectionWithExpectedSize() {
      List<String> items = new ArrayList<String>();
      items.add("java");
      assertThat(items, hasSize(1));
}

@Test
public void weCanUseCustomMatchersWithOtherMatchers() {
      List<String> items = new ArrayList<String>();
      items.add("java");
      assertThat(items, allOf(hasSize(1), hasItem("java")));
}
```

Lab 5 - Stubs and Fakes with Mockito

In this exercise we will use Mockito to experiment with writing Stubs and Fakes.

The next feature we need to implement is the ability to sell batches of simulations. The idea is that users will by vouchers for a certain number of simulations, and these vouchers will be used up as they run simulations. The price per simulation may vary over time depending on decisions from the sales team, and customers will also be able to use discount codes to obtain discounts.

We will use a similar domain model to the one discussed in the slides:



The PriceService is a complex domain object that will take into account both the discount code and the date of purchase. This module is being developed by another team, so we will need to stub it out to write the OrderProcessor. To make it easier to coordinate work between the two teams, an interface has been defined for the PriceService.

The OrderProcessor will process the requested number of simulations, along with an optional discount code, and produce a SimulationVoucher containing the total price, the number of simulations purchased, and the date of purchase.

Some examples of the requirements we need to implement are the following:

- Should be able to process an order of simulations with no discount

- Should be able to process an order of simulations with a valid discount code

- Should be able to process an order of simulations with an invalid discount code

Now implement this feature using a TDD approach. Make sure you do not write a real implementation for the the PriceService class - rather, create stubs and fakes in order to isolate the OrderProcessor from the PriceService.

Lab 6 - Handling Exceptions

The specifications for the PriceService interface specify that the getDiscountedPriceFor() method should throw two exceptions:

- *InvalidDiscountCodeException* if the code does not exist, and
- *ExpiredDiscountCodeException* if the code does exist, but has expired.

Using TDD, modify the *purchaseDiscountedSimulations()* method in the *OrderProcessor* class to handle these exceptions by throwing a single exception, *DiscountNotAppliedException*, but with a different error message for each type of error.

Typical behaviour you may need to test could be:

- When an unknown discount code is used a DiscountNotApplication exception should be raised

- When an unknown discount code is used the error message should mention the invalid discount code

- etc.

Lab 7 - Parameterized tests

The product owner wants to add a premium service with a dynamic pricing system. Users will pay for the number of simulations they run:

- 1-10 simulations: 10c / simulation
- 11-99 simulations: 9c / simulation
- 100+ simulations: 8c / simulation

We will implement this by writing a class that implements the PriceService interface.

Use a parameterized test to create a test for this class to be run against a significant selection of data. Remember, write the test case first. Once the tests are written, create an implementation of the PriceService interface (say, BulkDiscountPriceService).

Now implement the method.

Optional Step: Now apply the same technique to ensure that the pricing algorithm also works with discount codes.

Discuss how you might import test data from an Excel spreadsheet.

Lab 8 - Interaction-based testing

Following the success of the error handling feature of Lab 6, a new requirement has emerged: to avoid cheating, whenever an invalid discount code is used, the error must be registered with a DiscountFraudService. The interface for the service looks like this:

```
public interface DiscountFraudService {
   void registerInvalidDiscount(String discountCode,
                                DateTime date,
                                int numberOfSimulations);
}
```

Write tests to verify that when any invalid discount code is used, this method is called with the correct parameters.

Lab 9 - Refactoring

In this exercise we will practice some of the refactoring techniques discussed during the course.

Step 1) A new requirement is that a user can view the discount code used for a simulation voucher. First, write a test to express this requirement, e.g.

- When you purchase discounted simulations the voucher should store the discount code

Now implement this feature.

In this code, you will typically need either two constructors, or a single constructor where an empty or null discountCode parameter is sometimes used. Refactor this using the “Replace Constructors with Creation Methods” pattern.

Step 2) Refactor the Simulation class to use a more fluent approach using the builder pattern, e.g:

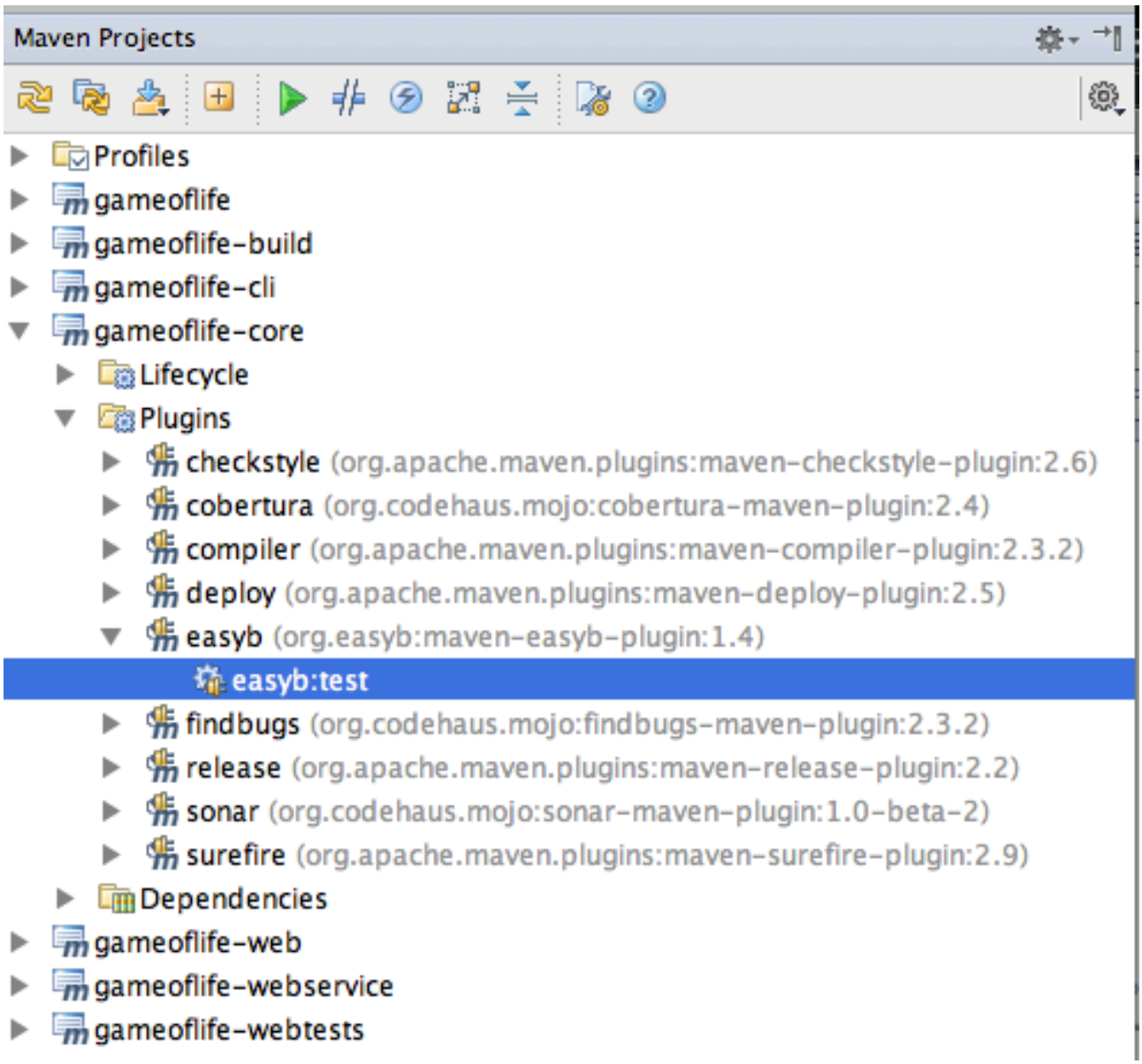
```
Simulation simulation = Simulation.startingWith(someGrid).andRunOn(runDate);
```

Step 3) (Optional) See if you can find any other code that could benefit from refactoring using any of the patterns discussed in class.

Lab 10 - Using easyb

The aim of this exercise is to experiment with writing BDD-style tests using easyb. The sample code is configured to run easyb stories in the gameoflife-core module, placed n the *src/test/stories* directory.

Step 1) Run the existing easyb stories. The easiest way to do this is to run the **easyb:test** maven goal in the *gameoflife-core* project, as shown below:



You should see the following output:

```
[java] Scenarios run: 1, Failures: 0, Pending: 1, Time elapsed: 0.04 sec
[java] 2 total behaviors ran with no failures
```

Now study the implemented story (*CountingThings.story*) to see how it works, and then implement the *MultiplyingThings.story*. The output should now have zero pending tests:

```
[java] Scenarios run: 1, Failures: 0, Pending: 0, Time elapsed: 0.04 sec
[java] 2 total behaviors ran with no failures
```

Step 2) Add an easyb story to test the price calculator we implemented in lab 7. A sample story could be the following:

```
import com.wakaleo.gameoflife.purchases.BulkDiscountPriceService

scenario "Bulk pricing simulations with a 10% discount", {
    given 'we are offering bulk discounts on simulations', {
        priceService = new BulkDiscountPriceService()
    }

    when 'a client purchases 25 simulations', {
        numberOfSimulations = 25
        totalPrice = priceService.getPriceFor(numberOfSimulations)
    }

    then 'the total price should be $2.25', {
        totalPrice.shouldBe 2.25
    }
}
```

Implement several scenarios along these lines.

Step 3) Implement a data-driven test using easyb, e.g:

```
import com.wakaleo.gameoflife.purchases.BulkDiscountPriceService

examples "When #{numberOfSimulations} simulations are purchased, the
total price should be #{totalPrice}", {
    numberOfSimulations = [1,    2,    9,   10,  11,   25]
    totalPrice =          [0.10, 0.20, 0.9, 1.0, 0.99, 2.25]
}

scenario "Purchasing #numberOfSimulations for #totalPrice", {
    given 'we are offering bulk discounts on simulations', {
        priceService = new BulkDiscountPriceService()
    }

    when "we are purchasing #numberOfSimulations", {
        calculatedTotalPrice =
priceService.getPriceFor(numberOfSimulations)
    }

    then "the total price should be #totalPrice", {
        calculatedTotalPrice.shouldBe totalPrice
    }
}
```