

(11224) INTRODUCTION TO SOFTWARE ENGINEERING

Lecture 4: Parameterised Tests

Test Driven Development

Mock Objects

Shereen Fouad

Announcement

- Starting from next week (26 Jan 2015) Monday lectures will be split into two groups as follows:
- **Morning session (11:00-12:00hrs)** LTA, Watson
- **Afternoon session (15:00-16:00hrs)** LT2, Law
- Students First name starting with A---L attend the morning session
- Students First name starting with M---Z attend in the afternoon session
- Any students doing joint degree programmes will need to be in the morning session

Reminder of Previous Lecture

- Testing is a key activity in software development process
- Testing can reveal the presence of errors NOT their absence
- Testing may be done in different levels:
 - Unit Testing
 - Integration Testing
 - End to End Testing
- TestNg is an open source automated testing framework

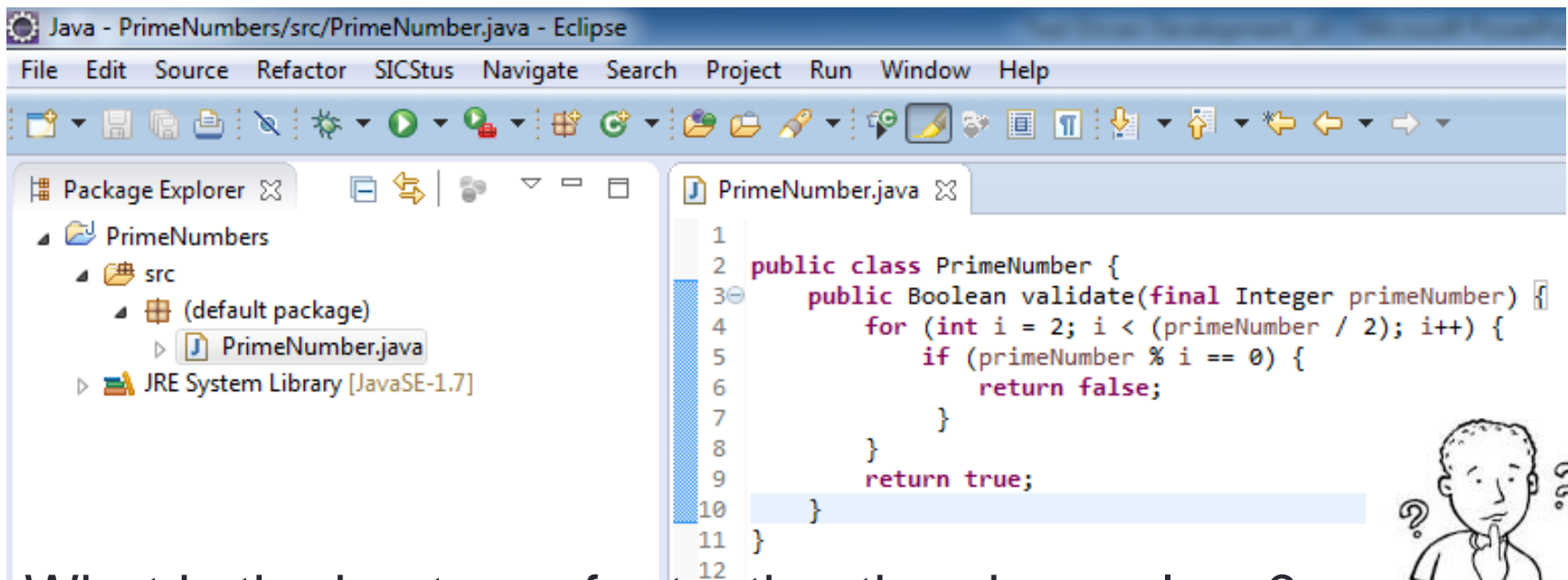
Overview

- Parameterised Tests in TestNG
 - Example
- Test Driven Development (TDD)
 - TDD Cycle
 - Why TDD
- Test with Mock Objects
 - Mockito Framework

Class PrimeNumberChecker.java

- This class checks if the number is prime.
- *“Prime number is a positive natural number that has only two positive natural number divisors - one and itself.”*

http://en.wikipedia.org/wiki/Prime_number



```
1
2 public class PrimeNumber {
3     public Boolean validate(final Integer primeNumber) {
4         for (int i = 2; i < (primeNumber / 2); i++) {
5             if (primeNumber % i == 0) {
6                 return false;
7             }
8         }
9         return true;
10    }
11 }
12
```



What is the best way for testing the above class?

http://www.tutorialspoint.com/testng/testng_parameterized_test.htm

Parameterised Tests !



- The below business logic requires a hugely varying number of tests.
- Coding the below test using the simple test methods we have seen so far is somewhat tedious and repetitive.
- TestNG provides Parameterised tests to more conveniently handle such sets of tests.

A screenshot of the Eclipse IDE interface. The title bar reads 'Java - PrimeNumbers/src/PrimeNumber.java - Eclipse'. The menu bar includes File, Edit, Source, Refactor, SICStus, Navigate, Search, Project, Run, Window, and Help. The Package Explorer on the left shows a project named 'PrimeNumbers' with a sub-package 'src' containing a file 'PrimeNumber.java'. The main editor window displays the code for 'PrimeNumber.java'.

```
1
2 public class PrimeNumber {
3     public Boolean validate(final Integer primeNumber) {
4         for (int i = 2; i < (primeNumber / 2); i++) {
5             if (primeNumber % i == 0) {
6                 return false;
7             }
8         }
9         return true;
10    }
```

Parameterised Tests

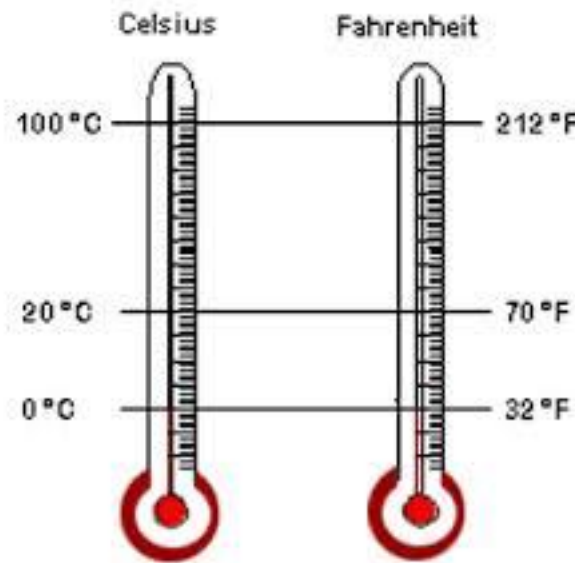
- A normal test method which has a mechanism associated with it for running the same test multiple times with different parameters.
- TestNG lets you pass parameters directly to your test methods With Data Providers
- An interesting feature available in TestNG.

Passing Parameters with *@DataProvider*

- Dataproviders are used to pass complex parameters or parameters that need to be created from Java.
- A Data Provider is a method annotated with *@DataProvider*.
- A Data Provider returns an array of objects.

Example of using @DataProvider in Test Class

If we want to test a method to translate Fahrenheit to Celsius, then you need to pass some values which covers all the boundary cases, i.e. cases which separate different situations for the method and are common sources of bugs.



Step one:

- Define the method `getFahrenheitToCelsiusData ()` which is defined as a `Dataprovider` using the annotation.
- This method returns array of object array.
- Each row in the array provides one set of parameters to the test method.

```
1.  @DataProvider
2.  private static final Object [][] getFahrenheitToCelsiusData ()
3.  {
4.      return new Object [][]
5.          {
6.              { -459.67, -273.15 },
7.              { -4.0, -20.0 },
8.              { 0.0, -17.77777778 },
9.              { 32.0, 0.0 },
10.             { 77.0, 25.0 },
11.             { 212.0, 100.0 },
12.             { 400.0, 204.44444444 }
13.         };
14. }
```

Step Two:

- Define the test method `fahrenheitConversionTest ()` with a test annotation that
 1. identifies the `getFahrenheitToCelsiusData` method that provides the sets of parameters to use
 2. This method takes two input parameters.
 3. This method validates the parameter passed

```
1.  @Test{DataProvider="getFahrenheitToCelsiusData"}
2.  public void fahrenheitConversionTest(float fahrenheit, float celsius)
3.  {
4.      assertEquals(fahrenheitToCelsius(fahrenheit, celsius, 1.0e-6);
5.  }
```

Test Driven Development (TDD)

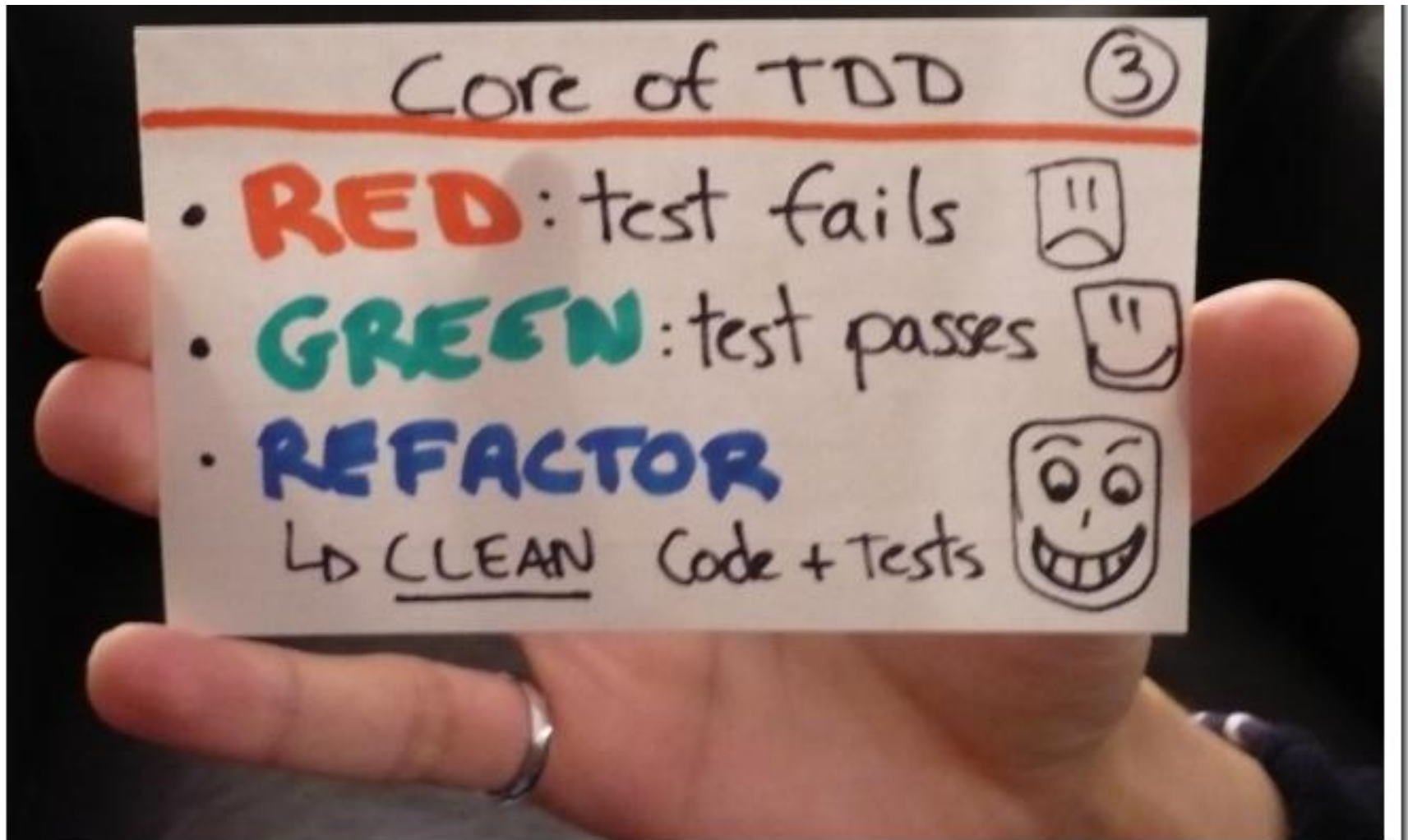
Can you tell what is TDD?



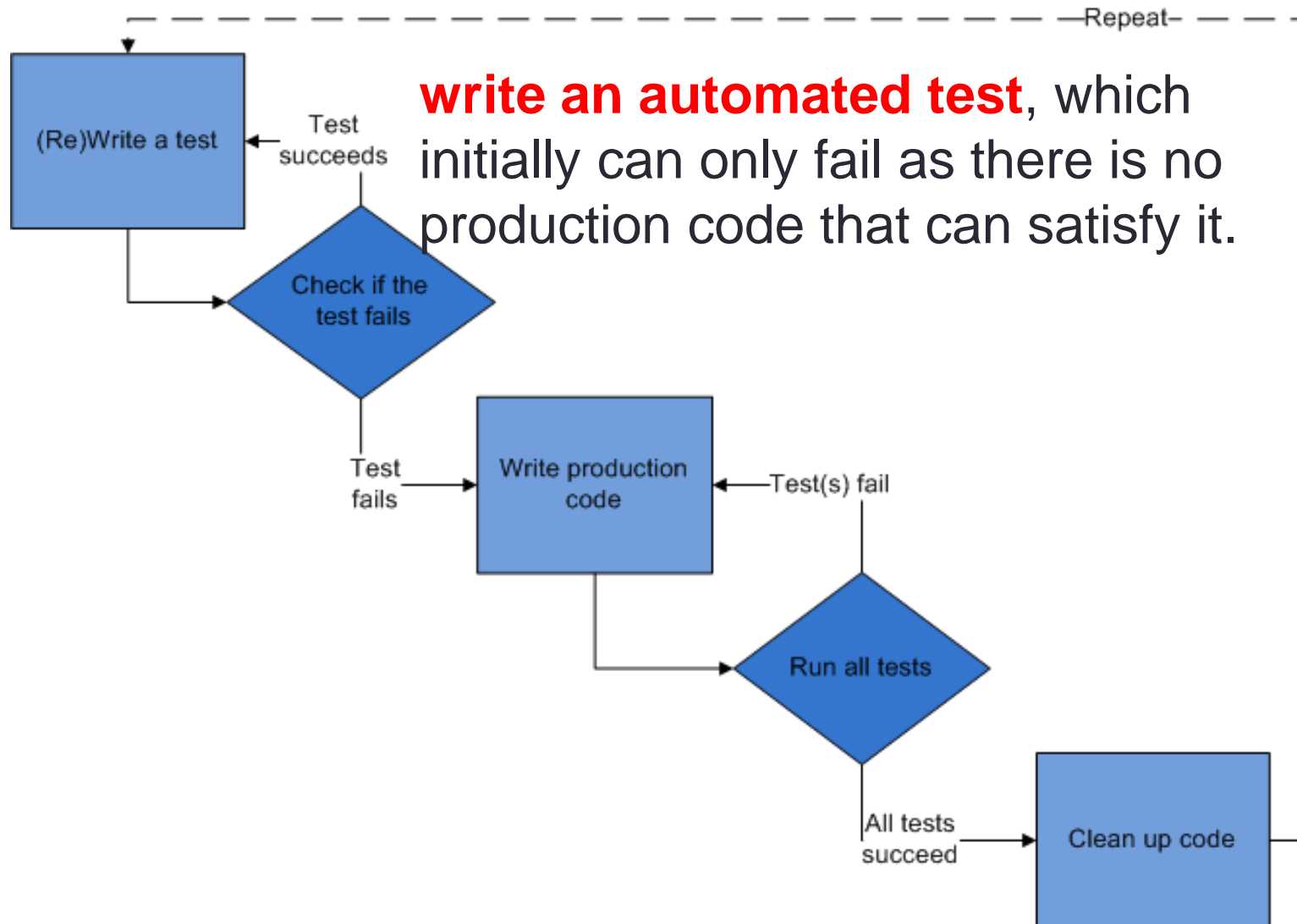
Test Driven Development (TDD)

- Traditionally programmers develop their production program code and then write test code to test it.
- Since 1999, a movement in software engineering, called **Extreme Programming**, has led to considering reversing this natural order and proposes writing the tests before writing the production code to test.
- This process is called **Test Driven Development, or TDD**.
- TDD is now seen as applicable in many software engineering and programming styles.

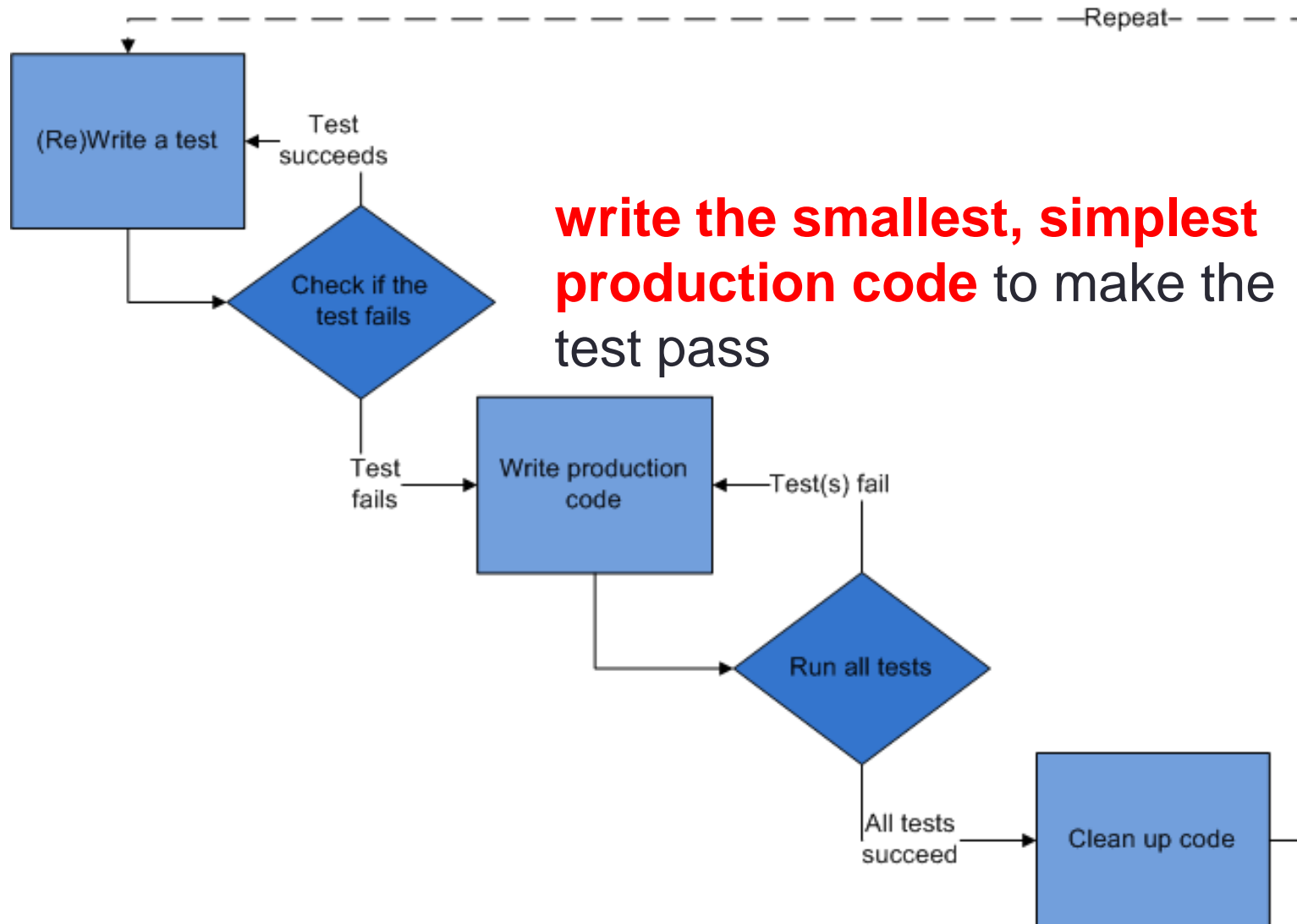
Standard Motto of TDD



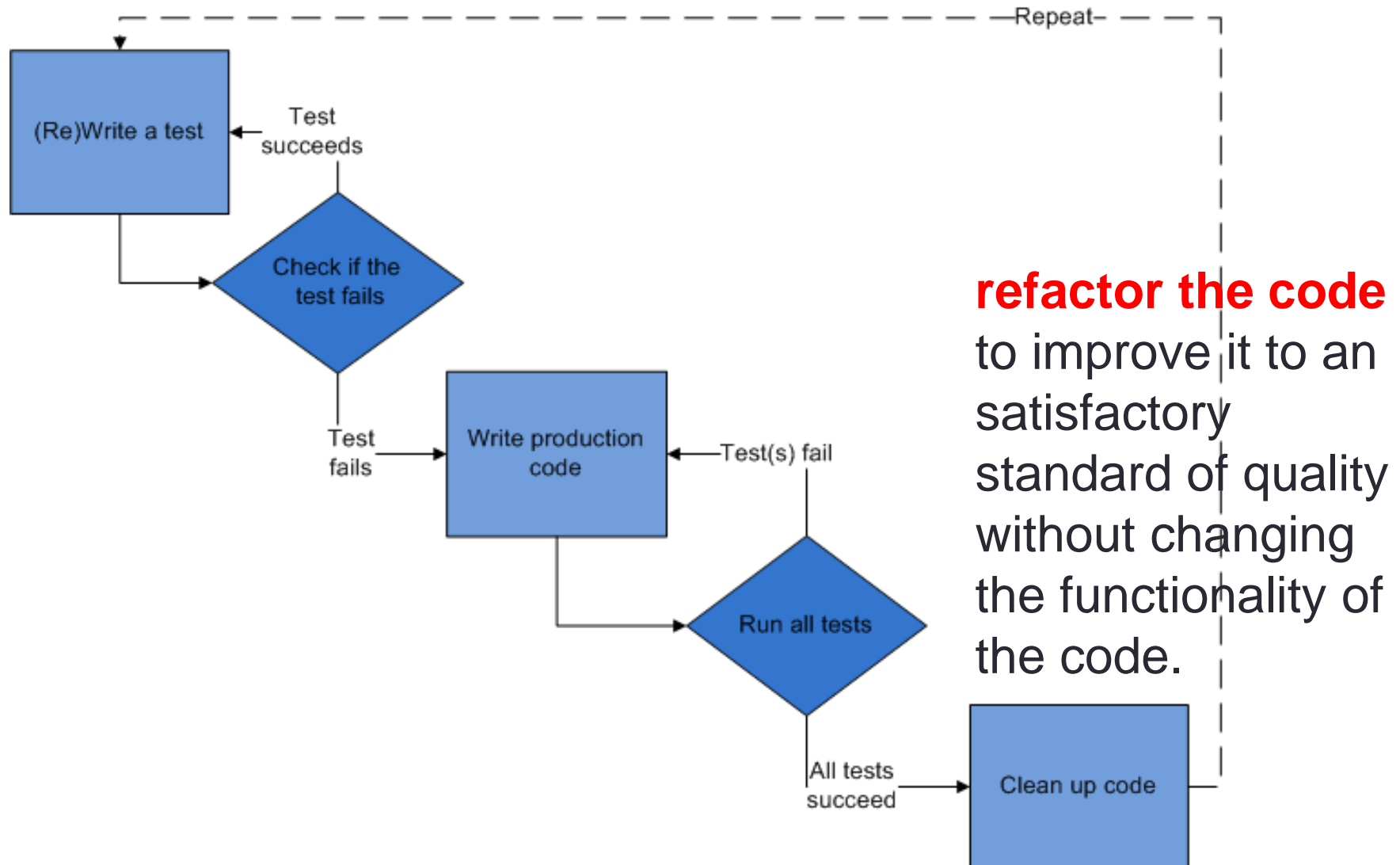
TDD Cycle



TDD Cycle



TDD Cycle



Rinse and Repeat

- The cycle is then repeated, starting with another new test to push forward the functionality.
- The size of the steps can be as small as the developer likes, or get larger if s/he feels more confident.

What if the code written to satisfy a test does not do so?



Rinse and Repeat

- The cycle is then repeated, starting with another new test to push forward the functionality.
- The size of the steps can be as small as the developer likes, or get larger if s/he feels more confident.

What if the code written to satisfy a test does not do so?

- then the step-size may have been too big, and maybe the increment should be split into smaller testable steps.

Why TDD??



Why TDD??

- Writing clear requirements
- Development in small steps makes an easier debugging
- Tests reduces bugs in systems
- Better designs of the interfaces
- Generate skeleton code for classes and methods
- Tests Reduce the Cost of Change
- Testing Forces You to Slow Down and Think
- Testing Makes Development Faster
- Testing Is Fun!!

Exercise !

- Take one of the early exercises in your Autumn Semester Software Workshop module and re- implement it from scratch using a test driven development approach. Compare your two different development experiences.

Quiz !!

- A Data Provider returns
 - a. an array of objects
 - b. Boolean
 - c. double
- The following process improves the code to a satisfactory standard of quality without changing the functionality of it
 - a. Testing
 - b. Debugging
 - c. Refactoring

Challenge of Unit Testing

- A unit test should test a class in isolation.
- In unit testing, the SUT is an object and the DOCs are other objects that the SUT interacts with.
- What are the possible forms of interaction?
- If we allow the true DOC objects to be used, then when we run a test on the SUT object, we are really testing not just the SUT object itself, but the whole graph of objects that are used by SUT.

Problem!

- This then is no longer a short and simple unit test, but a much larger and more complex mix of a unit test and an integration test.
- What about in the TDD?



Can you suggest something?

Solution: use Mock Objects (also known as Test Doubles)



- Replace our DOC objects with some form of substitute objects (mock object) that can support the interactions required by the SUT object but are simple objects that are part of the test code and don't add extra complexity to our unit testing.
- Can simulate behaviour of complex, real (non-mock) objects
- Very useful when a real object is impractical or impossible to incorporate into a unit test.

Test Doubles

- **Dummy Object** A substitute for a DOC object that can be passed to the SUT but is never used
- **Test Stub** A substitute for a DOC object that the SUT can invoke methods on, i.e. it provides indirect input to the SUT. A *stub* can be told to return a specified fake value when a given method is called.
- **Test Spy** A substitute for a DOC object that allows verification that the SUT calls the correct methods of the DOC with the correct parameters, i.e. that the SUT performed the correct indirect output.
 - *set expectations on its interactions with another object.*

Test Doubles

- **Mock Object** A substitute for a DOC object that can act both as a Test Stub and as a Test Spy.
- **Fake Object** A full substitute for a DOC object that provides all the functionality of the DOC object with an alternate implementation of the same functionality.
 - *(simpler or less resource consuming way)*
 - e.g. a fake distance finder object in a Satellite Navigation system might calculate the distance between two points as the straight distance rather than by finding the shortest route by the road network and calculating the distance from that.



- A mocking framework for Java that provides, in conjunction with testing frameworks like TestNG or JUnit, support for creating and using various kinds of mock objects in tests.
- Mockito does not provide support for Mock objects directly, but it does provide the same functionality as Mock objects through its mechanisms for stubbing and spies.
- Fake objects, as they implement true application specific logic, must be written by the programmer directly.

Adding the Mockito library in Eclipse

- Download the Mockito library jar file from the Mockito web site to your filesystem.
- Select the project from the Package Explorer panel,
- Then select menu item Project|Properties.
- In the dialog box that opens, select Java Build Path in the left panel and then the Libraries tab.
- You should have the JRE System Library and TestNG there.
- Click the Add External Jar... button and select the downloaded Mockito jar file.
- Click OK to accept the selection and then OK to close the properties dialog.

Package Explorer

Interest.java

EmployeeDetails

New

Go Into

Open in New Window

Open Type Hierarchy

F4

Show In

⌘W

▶

Copy

⌘C

Copy Qualified Name

Paste

⌘V

Delete

⌘X

Build Path

Source

⌘S

Refactor

⌘T

▶

Import...

Export...

Refresh

F5

Close Project

Close Unrelated Projects

Assign Working Sets...

Debug As

Run As

Team

Compare With

Restore from Local History...

Configure

TestNG

Properties

⌘I

package test;

import org.testng.Assert;

public class TestEmployeeDetails{

EmplBussinessLogic empBusin

EmployeeDetails employee = r

@Test

public void testCalculateApp

employee.setName("Rajeev

employee.setAge(25);

employee.setMonthlySalar

double appraisal = empBu

.calculateAppraisal()

Assert.assertEquals(500,

}

// test to check yearly sal

blems

@ Javadoc

Declaration

☰

ch:

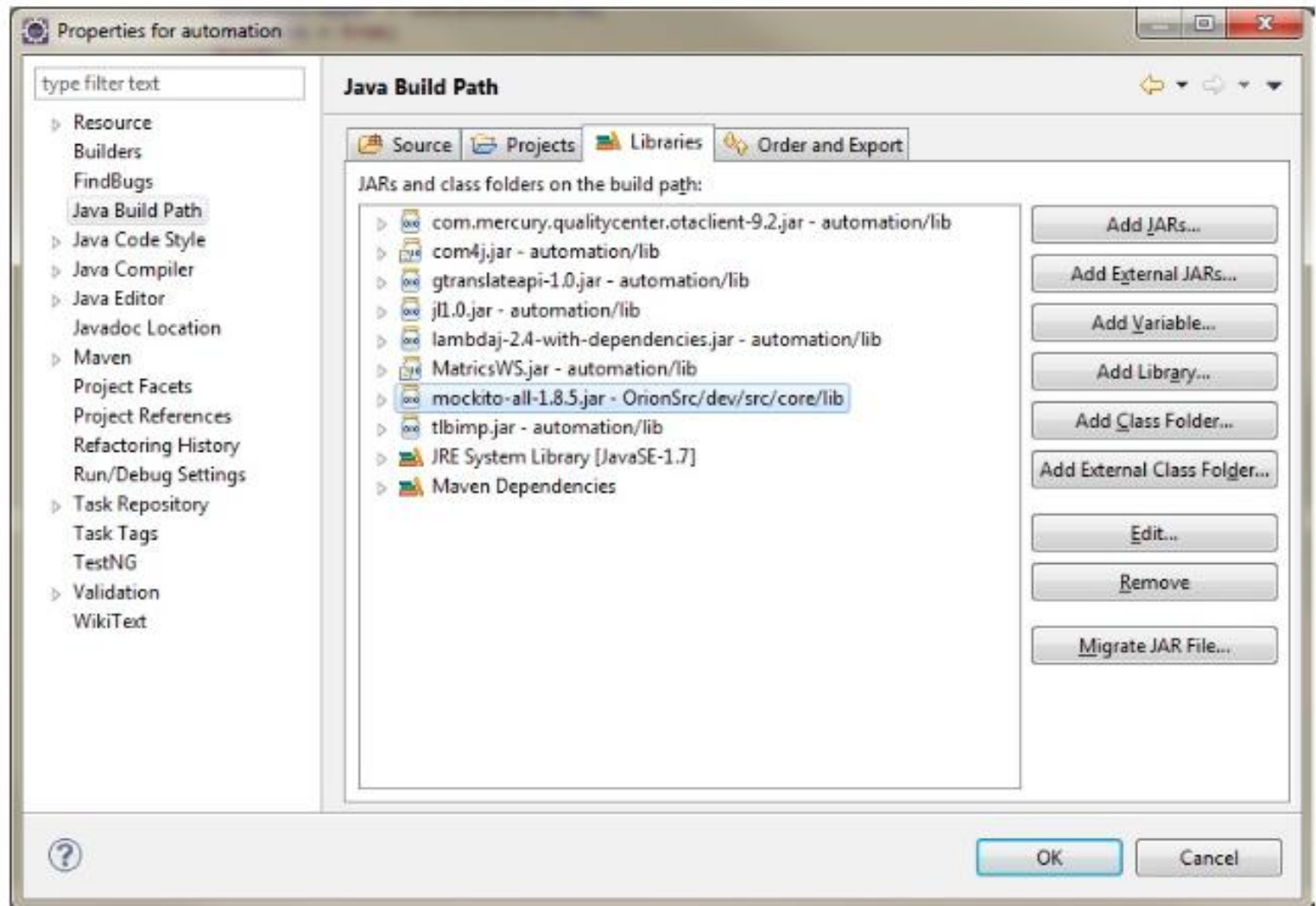
All Tests

Failed Tests

Summary

Emp

Adding the Mockito library in Eclipse



Example

- Assume that we are testing a Satellite Navigation class, SatNav which uses a DOC called RouteFinder to calculate routes.
- We thus do not want to create a real RouteFinder object when we are unit testing SatNav.



Using a Mockito Dummy

- Any test file that uses Mockito should do a static import as follows:

```
import static org.mockito.Mockito.*;
```

- To test the SatNav constructor, we can then do the following:

```
1.  ...
2.  @Test
3.  public class SatNavTest
4.  {
5.      private RouteFinder mockRouteFinder = mock(RouteFinder.class);
6.
7.      public void satNavConstructorTest()
8.      {
9.          SatNav satNav = new SatNav(mockRouteFinder);
10.         ///... some TestNG assertions about satNav ...
11.     }
12. }
```

- As far as the SatNav constructor is concerned, it was given a valid RouteFinder object which it does not actually interact with in its constructor so the constructor should be able to pass its tests.

Using a Mockito Stub

- If methods of the mock object must return certain values when invoked by the SUT object, then we use a `when (...).thenReturn(...)` construct,
- It requests that when the specified method is called, it returns the specified value.

```
1.  @Test
2.  public class SatNavTest
3.  {
4.      private RouteFinder mockRouteFinder = mock(RouteFinder.class);
5.
6.      public void satNavDistanceTest()
7.      {
8.          when(mockRouteFinder.getDistance()).thenReturn(25.0);
9.
10.         SatNav satNav = new SatNav(mockRouteFinder);
11.         // This assumes that satNav.getDistance() calls getDistance() on its
12.         // RouteFinder to get the result
13.         assertEquals(satNav.getDistance(), 25.0, 1e-7);
14.     }
15. }
```

References

- A number of slides in this talk is based on:
 - Alan P. Sexton hand-outs (Introduction to Software Engineering. The University of Birmingham. Spring Semester 2014)
 - SOFTWARE ENGINEERING 9 Ed. by Ian Sommerville