"We all know that testing is an important part of software development, but delays in projects and budget pressures can mean that time available for testing is less than ideal."

Text from advert for this workshop

Workshop Structure

- Automated Testing in Context
- Part 1: Calculator
- Some terminology
- xUnit
- Test-Driven Development
- Readable Tests
- Unit, Integration and Acceptance Tests
- Summary

Software for this workshop

- Java (6 or later, 8 is most current)
- A Java IDE, e.g. Eclipse, Netbeans, IntelliJ

  Or

- Your favourite text editor, ideally with syntax highlighting
- The following tools
  - Apache Ant
  - Junit (including related hamcrest files)

Automated Testing in Context

Some testing strategies

- There are different approaches to testing a system. Examples:
  - Unit testing
  - Subsystem testing
  - System testing (integration tests)
  - Acceptance testing
  - Regression testing
  - Performance testing
  - Stress testing
  - ...

Specifying and Documenting Tests

- One approach is to use Test Tables.
- What are the advantages and disadvantages?

Example for an E-Bay style system:

ID	Requirement	Description	Inputs	Expected outputs	Pass/ Fail	Comments
A1.1	FR12	Enter bid value: Item ID = 1234 (see Table 1), current value = £1, bid increment = 10p	Enter £1.10	Screen shot SS1 is displayed	P	
			Enter £1.09	Screen shot SS2 is displayed	P	
			Enter minus £1.10	Screen shot SS3 is displayed	F	Treats the input as if it were positive
A1.2	FR13	Select category: See Table 2 for categories test data.	Click "All categories" link.	Screen shot SS4 is displayed	F	A NullPointer Exception was thrown
			Click "Cars" link	Screen shot SS5 is displayed	P	

Motivation for Automated Testing

- For testing to be effective, we should automate it as much as possible.
- Benefits:
  - **Speed** – ideally, issue one command to start a full run of tests. Quicker than a human following a test list.
  - **Frequency** – regular execution of the tests.
  - **Repeatability** – run the same tests multiple times.
  - **Accuracy** – fewer mistakes made during testing
  - **Documentation** – such tests are also a form of documentation, listing what needs to be tested

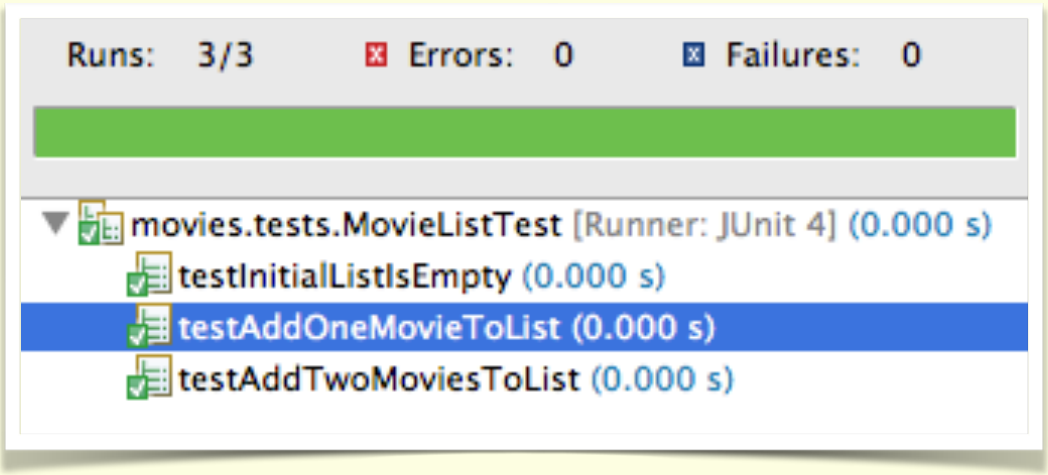
Unit Testing

- Focus on small parts of the system
- Automate the process - write the tests as code

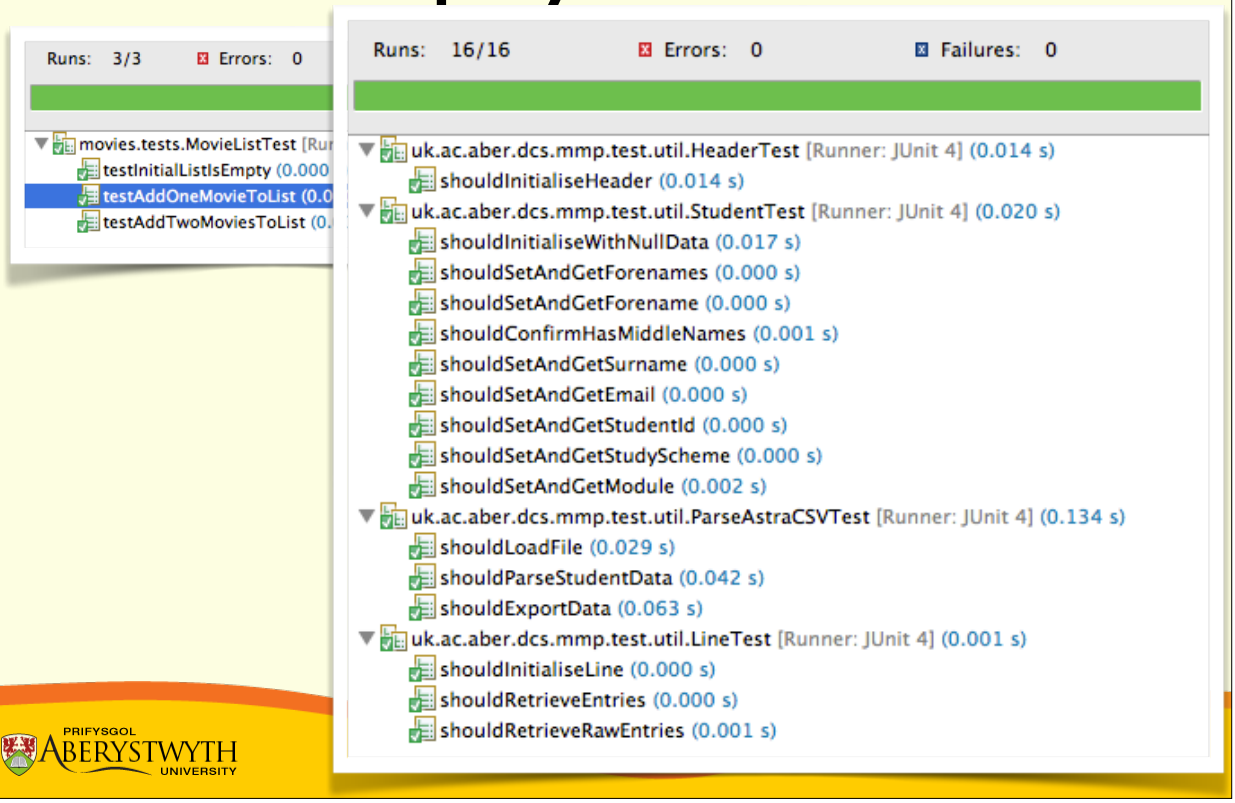
```
@Test
public void testAddTwoMoviesToList() {
    MovieList list = new MovieList();
    list.addMovie(new Movie());
    list.addMovie(new Movie());
    assertEquals("List should contain 2 entries", 2, list.size());

}
```


Visual display of the results



Visual display of the results



Automated Testing Toolbox

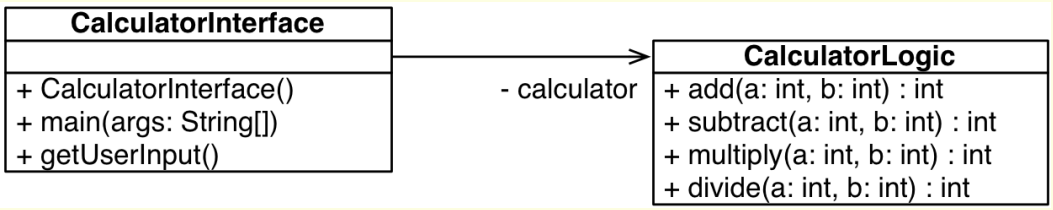
- Version control system:
  - e.g. git, svn, mecurial, cvs, …
- Build system:
  - e.g. ant, maven, gradle, …
- Testing Frameworks
  - e.g. JUnit, JMock, Cucumber, …
- Continuous Integration,
  - e.g. Jenkins (Hudson), Cruise Control, Bamboo, …

Anything new?

- The industry has worked to create tools to help automate tests for many years
  - Some of the tools would record actions performed and create tests scripts that can be run multiple times
  - Custom tools to provide test harnesses
- The more recent language facilities and tools offer a change of focus, encouraging earlier use of automated testing

Part 1: Calculator

- In this section, we will test the Calculator
  - If you were in the One Click Deployment workshop, the code is very similar



Project Files

- Ant Build Script: **build.xml**
  - This will create **bin** directory for compiled code and **dist** directory for the jar file.
  - Key tasks: **dist** (compile) and **test**
- Source Directory: **src**
  - uk.ac.aber.dcs.CommandLineCalculator
    - CalculatorInterface.java
    - CalculatorLogic.java
  - uk.ac.aber.dcs.CommandLineCalculator.tests
    - CalculatorLogicTest.java

Running the tests

- At the command line, type:

ant test ← Change to: ant -Djunit.show test

- Example output:

```
ant test
Buildfile: /Users/neiltaylor/…/AutomatedTesting/CalculatorProject/build.xml

compile:

dist:

test:
    [junit] Running
            uk.ac.aber.dcs.CommandLineCalculator.tests.CalculatorLogicTest
    [junit] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.063
sec

BUILD SUCCESSFUL
Total time: 0 seconds
```

Look at the code

- Look at the three classes in a text editor

```
package uk.ac.aber.dcs.CommandLineCalculator.tests;

import uk.ac.aber.dcs.CommandLineCalculator.*;
import static org.junit.Assert.*;

import org.junit.Test;

public class CalculatorLogicTest {

   @Test
   public void testAdd() {
      CalculatorLogic calculator = new CalculatorLogic();
      assertEquals(3, calculator.add(2,1));
   }

}
```

Test Case

Exercise: Add another test

- Edit the CalculatorLogicTest.java class
- Add a new test case to confirm that the subtract operator works
  - This is very similar to the testAdd() method on the previous slide.
  - Run the tests using the ant script. Does it pass?

Tidying Up The Test Cases

- Several repeated lines to create a CalculatorLogic object for each test case
- Could simplify this by using a special method to be run before each test case
- Use the @Before annotation (from org.junit package)


```
package uk.ac.aber.dcs.CommandLineCalculator.tests;

import uk.ac.aber.dcs.CommandLineCalculator.*;
import static org.junit.Assert.*;

import org.junit.Before;
import org.junit.Test;

public class CalculatorLogicTestWithBeforeMethod {

   private CalculatorLogic calculator;

   @Before
   public void setup() {
      calculator = new CalculatorLogic();
   }

   @Test
   public void testAdd() {
      assertEquals(3, calculator.add(2,1));
   }
```

What else can we test?

- Not much code in the example
- Could we use JUnit to test the CalculatorInterface class?

Problems with CalculatorInterface

- There is one useful method, which reads characters from the command line
- It writes output to the command line
- What would be the setup for the test?
- How would we check if the method worked or not? Can we get the output and check it?

```
public void getUserInput()
{
    Scanner keyboard = new Scanner(System.in);
    System.out.println("enter an integer");
    int a = keyboard.nextInt();

    System.out.println("enter an operation (+,-,* or /)");
    String operator = keyboard.next();

    System.out.println("enter another integer");
    int b = keyboard.nextInt();
    keyboard.close();

    int result=0;

    switch(operator) {
        case "+":
            result=calculator.add(a,b);
            break;
        case "-":
            result=calculator.subtract(a,b);
            break;
        case "*":
            result=calculator.multiply(a,b);
            break;
        case "/":
            result=calculator.divide(a,b);
            break;
    }

    System.out.println(a + " " + operator + " " + b + " = " + result);
}
```


Possible Test

```
package uk.ac.aber.dcs.CommandLineCalculator.tests;

import static org.junit.Assert.*;

import org.junit.Test;

import uk.ac.aber.dcs.CommandLineCalculator.CalculatorInterface;

public class CalculatorInterfaceTestBeforeChanges {

   @Test
   public void testGetUserInput() {
      CalculatorInterface calculator = new CalculatorInterface();
      calculator.getUserInput();
   }

}
```

Problems:

1. Test will pass (assuming there is a 'command line')
    1. There isn't an assert statement
2. Different behaviour if there isn't a 'command line'

If we want to automate our tests, we may need to rethink our software designs

Can we make our code more 'testable'?


```
public void getUserInput()
{
    Scanner keyboard = new Scanner(System.in);
    System.out.println("enter an integer");
    int a = keyboard.nextInt();

    System.out.println("enter an operation (+,-,* or /)");
    String operator = keyboard.next();

    System.out.println("enter another integer");
    int b = keyboard.nextInt();
    keyboard.close();

    int result=0;

    switch(operator) {
        case "+":
            result=calculator.add(a,b);
            break;
        case "-":
            result=calculator.subtract(a,b);
            break;
        case "*":
            result=calculator.multiply(a,b);
            break;
        case "/":
            result=calculator.divide(a,b);
            break;
    }

    System.out.println(a + " " + operator + " " + b + " = " + result);
}
```

Separate into another method, which we can test

Exercise: Testing CalculatorInterface

- Modify the method getUserInput() in the CalculatorInterface class
  - Create a new method which you can test
  - Call this from the getUserInput() method
- Create a new Test Class in the tests package
  - A a new test method, which checks the output of your code.

Recap

- Looked at the Calculator Project
  - Ant build script (easier to automate, repeatable)
    - Output options, configured in the build.xml file
  - JUnit tests
    - @Test for each test method
    - Each test method should have public accessibility
    - Used @Before to have some common setup
    - Different test classes
    - Use of assertEquals method
  - Might re-assess design to make it more testable

Some Terminology

- SUT
- Test Case
- Test Driver
- Test Fixture
- Automated Test Phases

- Related to definitions in "xUnit Test Patterns, Refactoring Test Code", Meszaros, 2007, Pearson Education, USA.

SUT

- System Under Test – the item that we are setting up tests for.
- This is not necessarily a full system – it could just be a small part of a system, e.g. a class

Test Case

- An individual test
  - An identifier (name)
  - Specifies what will be tested
  - What setup is required?
  - Specifies test data to be used
    - boundary data, equivalence partitions…
    - Normal data
  - Abnormal data
  - Specifies expected results

Test Driver

- A script or program that takes a set of test cases, runs each case and records the results.
- JUnit is an example of a Test Driver, although we more commonly call it a Test Automation Framework (term used in the xUnit book)
- Each test case is described as a test method, annotated with @Test

Test Fixture

- The test context used when running a test case.
  - Setting up common data
  - Creating objects
  - Opening files, or other sources of data
- Preparing to run a specific item to test, e.g. calling the add method on the calculator, before verifying the expected results.

Automated Test Phases

- Each test case typically does three things:
  - **Setup** the initial state (test fixture…)
  - **Exercise** the class under test
  - **Verify** expected output data against actual output data...
- General Setup?
- **Teardown?**

xUnit Testing

JUnit Overview

- Looking at unit testing since xUnit/*Unit is used very widely in SE community...
- Open source (Common Public Licence)
  - Latest version: 4.11 from www.junit.org
  - Installed version of Eclipse has built-in support for an earlier version of 4.x
- Testing frameworks are available for other languages

Example assert statements

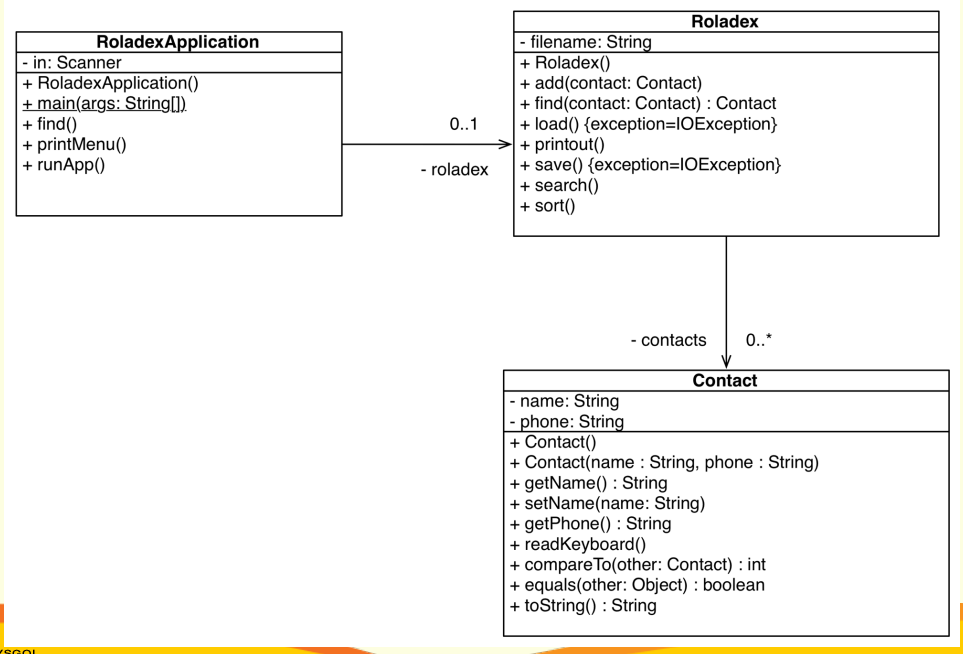
- assertEquals(expected, actual)
- assertFalse(condition)
- assertNotNull(object)
- assertNotSame(expected, actual)
- assertNull(object)
- assertSame(expected, actual)
- assertTrue(condition)
- fail(message)

Examples (with messages)

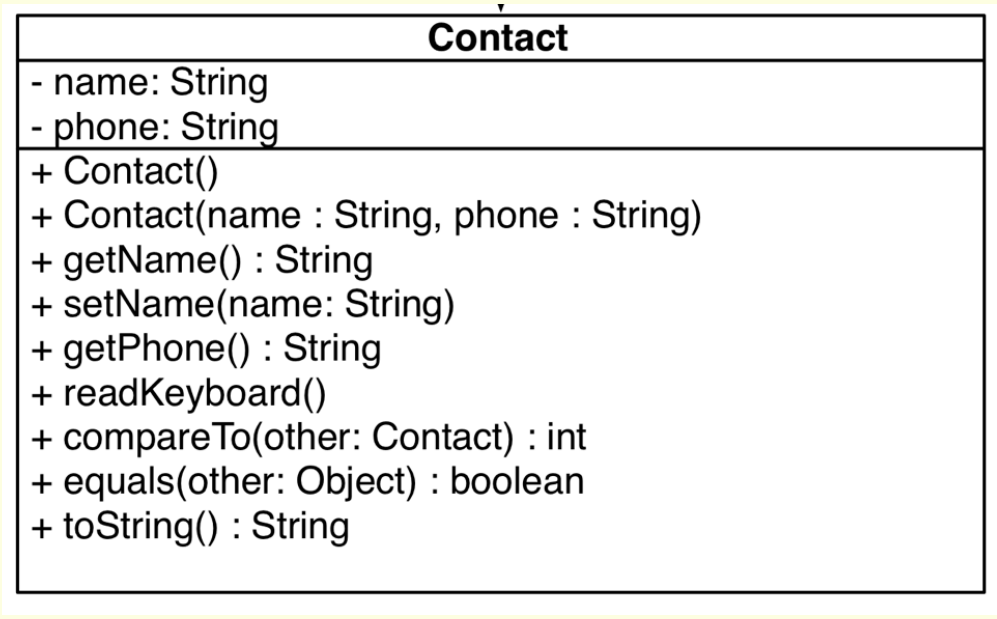
- assertEquals(**message,** expected, actual)
- assertFalse(**message,** condition)
- assertNotNull(**message,** object)
- assertNotSame(**message,** expected, actual)
- assertNull(**message,** object)
- assertSame(**message,** expected, actual)
- assertTrue(**message,** condition)
- fail(message)

Roladex Application

- Computer version of a Rolodex



Contact Class



Exercise

- Look at the Address, Contact, Roladex and RoladexApplication classes
- Which of these classes are suitable for unit testing?
- What changes might you propose to make it easier to test this code?

Working with code

- We will look at the code and talk about some issues, including:
  - Setup, Exercise, Verify for tests with Contact
  - Choice of test case names
  - Multiple Assertions in a single Test Case
  - Keep the test methods simple and easier to read
  - Testing get and set methods
  - Using utility methods
  - Working with exceptions

Handling Exceptions

- We could run the code. If we get to the line following one that should have generated an exception, then we could use the fail() statement to indicate an error

- In JUnit 4, we can use:

  @Test(expected=NotFoundException.class)

Test Driven-Development

- Use automated tests to help design and deliver your project
- Write a test, before you write the code that will pass the test.
- Use the same tools that we have discussed so far, but change the emphasis to thinking about the tests and making sure that they are in place before we have written the code.
- This approach has grown out of Agile approaches to development

Red

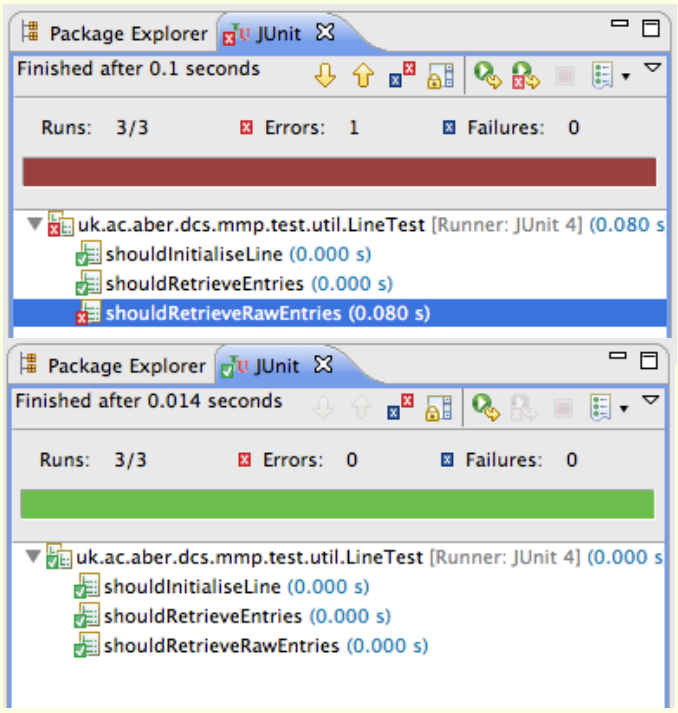
Green

Refactor

1

2

3



Refactor

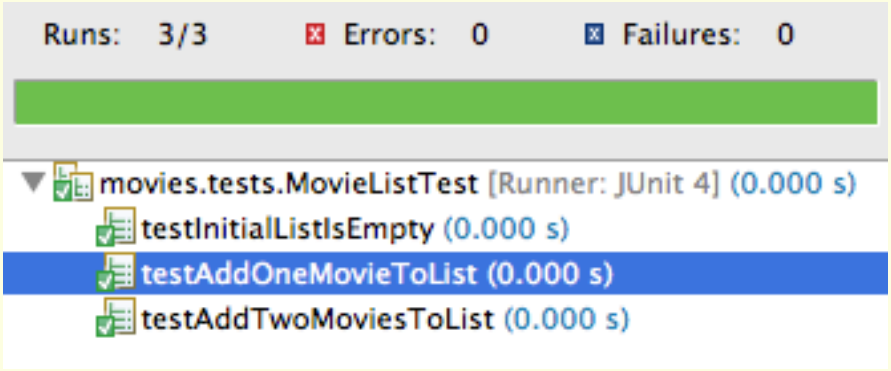
TDD Example: Movie List

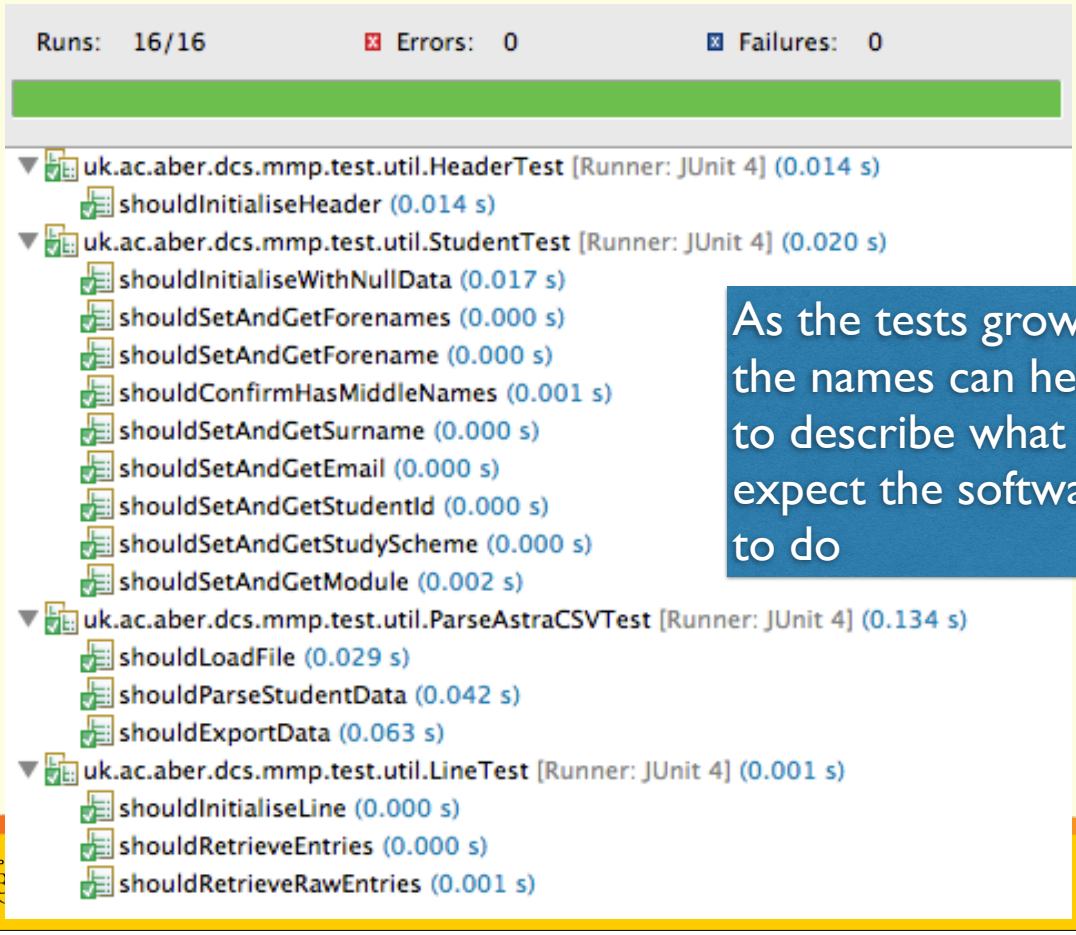
- Develop part of a new system to hold details about Movies. Based on example from early book on TDD, by David Astels.
- Look at tasks on the hand-out.
- Based on the given task, we select a set of tests. We then write a test and part of the code, following Red-Green-Refactor. We then move on to the next test, and so-on.

Exercise: Shopping Cart

- Have a look at the hand-out sheet with stories for building a shopping cart.
- Use TDD to develop tests and code for the first story.
- Remember - tests first.
- Only have one failing test at a time.

Tests as documentation…?





As the tests grow, the names can help to describe what we expect the software to do

Readable Tests

- When you add JUnit to your project in an IDE, you may see Hamcrest linked to your project too.
- Hamcrest has developed a set of matchers, which can be used in unit tests to help improve the readability of your tests.
- Originally developed in Java, but ported to other languages
- Some documentation at:
  - https://code.google.com/p/hamcrest/wiki/Tutorial


```
package uk.ac.aber.dcs.CommandLineCalculator.tests;

import uk.ac.aber.dcs.CommandLineCalculator.*;
import static org.junit.Assert.*;
import static org.hamcrest.Matchers.*;

import org.junit.Test;

public class CalculatorLogicTestWithHamcrest {

   @Test
   public void testAdd() {
      CalculatorLogic calculator = new CalculatorLogic();
      int result = calculator.add(2,1);
      assertThat(result, equalTo(3));
      // the following is exactly the same
      // assertThat(result, is(equalTo(3)));
   }

}
```

Unit, Integration and Acceptance Tests

- xUnit tools provide a useful way for developers to create an automated suite of tests
- Useful at higher levels to test integration of different parts of a system
- This is too detailed for most customers. There has also been work to help create tools for customers to be part of the agile process and create tests:
  - Cucumber - example of tool to support Behaviour Driven-Develoment
  - Fitnesse - a wiki driven approach to help customers create test cases

Automated Testing Summary

- Test Framework provides the basis
- Part of a wider tool setup
- Collection of readable tests that should describe the expected behaviour of a system
- Not suitable for all tests, but prompts us to think about design changes to aid testing
- Test Driven-Development uses tests to steer the development