Object Oriented Programming

Testing

Dr. Seán Russell
sean.russell@ucd.ie

School of Computer Science,
University College Dublin

September XX, 2019

Learning outcomes

After this lecture and the related practical students should...

- understand the purpose of testing
- be able to describe the difference between traditional testing and unit testing
- understand the idea of Test Driven Development
- be able to design and implement basic state tests using JUnit
- understand the application of behaviour testing in unit tests

Table of Contents

Testing

- Testing is an important part of programming and software engineering
- Described simply, testing is making sure that the program you have written works correctly
- In professional software engineering, testing is evaluation of the software against the requirements
- There are two primary methods of testing,
  - **Traditional Testing**
  - **Unit Testing**

Table of Contents

Traditional Testing

- Traditional testing usually involves running the whole program and seeing the result while unit testing tests each component of the system individually
- Without testing, it is hard to know if a program is correct
- We may also miss problems in our tests

Traditional Testing

- It can be very difficult to test every part of the system
- We would need to consider all possible situations that may happen
- It can be difficult to know where an error actually is when it happens

Testing strategies

There are a number of different ways that traditional testing is done

- Adding print statements
- Using a debugger
- Testing scripts

Print statements

Printing information is the easiest form of testing

- We add print statements to the code to print out variables at different times
- Only we know the meaning and expected output
- This creates a lot of output
- Must be removed in production and can't be used

Debuggers

- A debugger is a special piece of software that can be used to execute a program
- Debuggers allow the execution to be paused at locations and the user to view the values of variables
- Many allow the user to step through code one statement at a time
- Debuggers can be difficult to learn how to use well
- Finding errors using a debugger requires knowledge of an entire system as we need to know how the pieces of the application work together

Testing scripts

- A testing script is automated script that will execute a program and provide it with the required input
- The output of the program is then recorded and compared against the output that was expected
- We need to already know exactly what output is expected
- If the output is different from the expected output by even a single character it may be viewed as a fail
- This means that when we change the program we also need to change the expected output

Know what Should Happen

- All traditional testing methods have the same requirement, we need to know what the output should be before the program can be tested
- If we do not know what the result should be, how would we know if it was correct or not
- This is possibly the biggest difficulty when it comes to testing

Table of Contents

Unit testing

- The best alternative to traditional testing is **unit testing**
- The idea is that we test each component of the system **individually**
- Errors are found earlier because tests fail as soon as an individual component is completed

Table of Contents

JUnit

- JUnit is a library for Java that makes it easy to write and execute unit tests
- JUnit is the most commonly used unit test library for Java
- This is because with JUnit, it is easy to use, tests can be written easily and tests can be automated
- More information can be found at `http://www.junit.org`

JUnit is not perfect

JUnit is a very useful framework, but it cannot be used for everything.

- JUnit is not good for testing GUIs (Graphical user interfaces)
- JUnit does not compile good reports for large projects
- JUnit takes time to set up
- It is difficult to test non Java elements of programs

Testing Classes

- The main components of JUnit are test classes
- These are separate classes used only for testing that are not included in the final program
- Within test classes, there can be many test methods
- Each of these test methods can be executed individually or all together as a group

Table of Contents

Assertions

- Testing in JUnit is done by adding declarative statements called assertions
- These statements will state that some particular property will be true at the time they are executed
- If any of the statements within a test are not true, then the entire test has failed

Common Assertions

Here is a list of the most commonly used declarative statements;

- **assertEquals**
- **assertTrue**
- **assertFalse**
- **fail**
- **assertSame**

assertEquals

- The `assertEquals` declaration takes two parameters of the same type and fails only if the values are not equal
- Typically, we would use this to test if a variable has the value we expect it to have at a give time
- For example, `assertEquals(10, x);` will fail if the value of x is not 10.

assertTrue

- The assertTrue declaration takes a single parameter that must be a boolean expression and fails if the result of the expression is not true
- This allows a more varied type of check, where we can be sure that some property is true
- For example, `assertTrue(x <= 100);` only fails if the value of x is greater than 100 but will allow many different values for x

assertFalse

The assertFalse declaration is basically the reverse of the assertTrue, this will fail if the result of the expression is not false

fail

- The fail declaration takes no parameters and always causes a test to fail
- This might seem like it is not very useful, but it is
- Typically, this declaration would be added to a part of the code that should never be executed
- For example, if we have a large series of if else statements, we could add the fail declaration so that we know when none of the if statements matched

assertSame

- The assertSame declaration takes two objects as parameters and checks to see if in fact they are two references to the same object in memory

- This is is similar to assertEquals, but even if two objects are equal (contain the same data) they may be separate in memory

Table of Contents

Setting up unit tests

- Test classes usually have the same name as the class they are testing, but we add the word "Test" to the end
- For example, if we are testing the `Calculator` class, we would create a class called `CalculatorTest`
- The steps for this in eclipse are
  1. Select the class we want to test and right-click in the package explorer
  2. From the menu select `New - JUnit Test Case`
- The first time this is done for a project, eclipse will ask you if you want to automatically include the JUnit library in the project, click Yes or OK.

Creating test classes

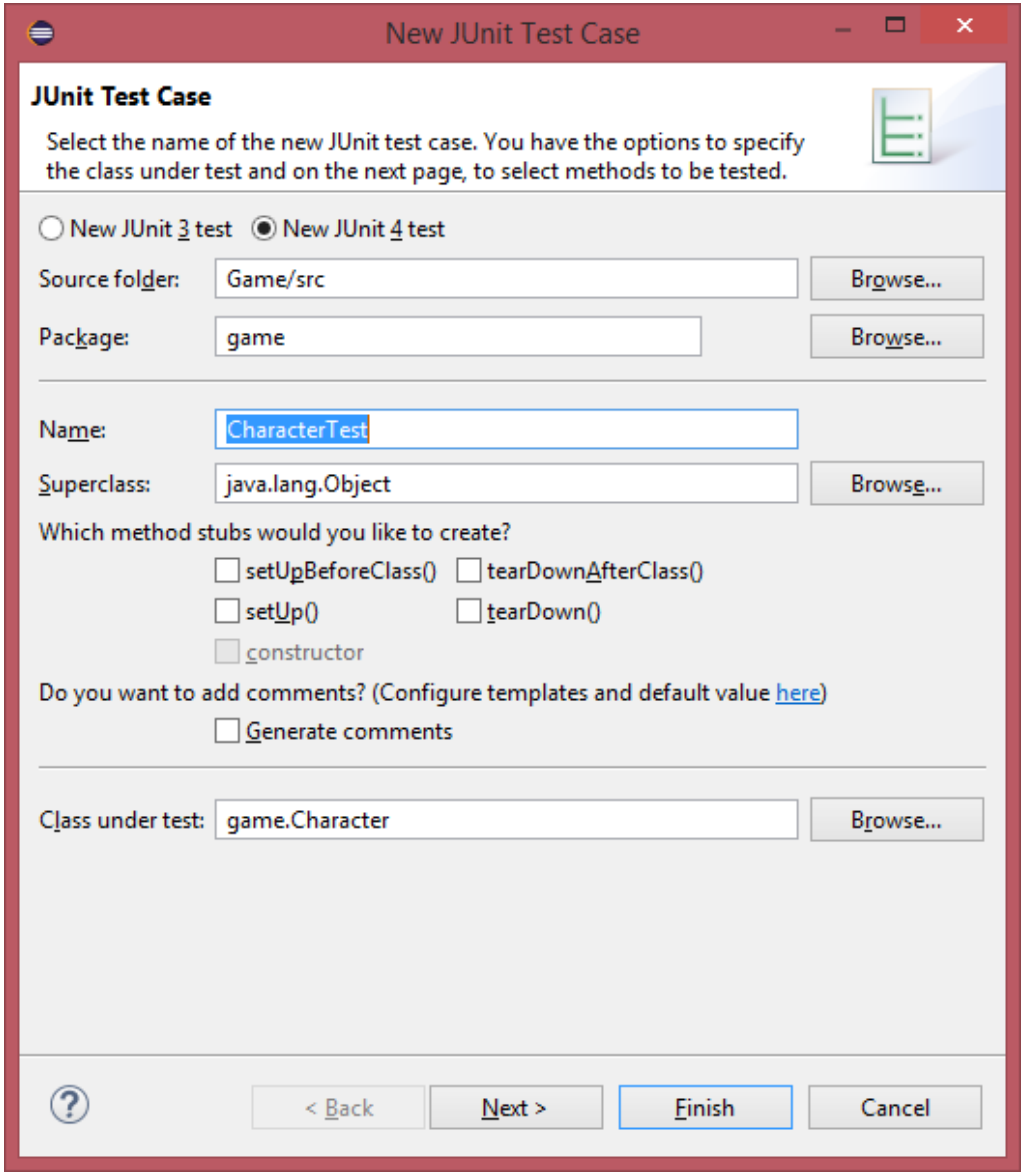


Table of Contents

Annotations

- Tests in JUnit can be given any name, but in order for the system to be able to know which methods are tests we need to add a piece of information to the declaration
- This piece of information is known as an **annotation**
- There are many different uses for annotations and you can study them in your own time

Annotations

- Annotations are used to put a note on a method, class or variable
- Annotations have no effect on the execution of the code in Java
- Annotations are written using the 'at' symbol followed by a name, e.g. `@Test` or `@Override`
- There are different annotations used in JUnit:
  - `@Test` - When this annotation is placed before a method, it tells JUnit that the following method is a test that should be executed when the class is being tested.
  - `@Before` - When this annotation is placed before a method it tells JUnit that the method is used to perform set up before any of the tests should be executed.

Annotation Example

- In order for the system to know that a method is a test, we just have to add `@Test` before it

Empty Test Method

```
1 @Test
2 public void testSomething(){
3   // do some testing
4 }
```

Table of Contents

Test-Driven Development (TDD)

- TDD is a development technique where you must first write a test that fails before you write new functional code
- The idea is that you must fully understand what the code should do before you write it

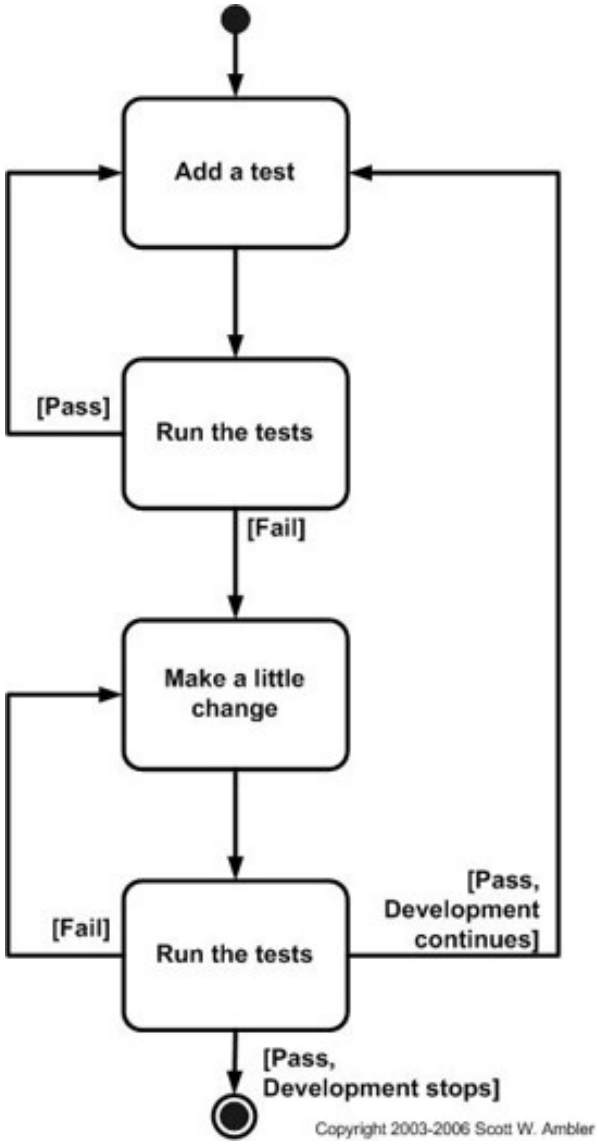
Why TDD?

- Develop better quality software
  - Write code that contains less bugs
  - Prevent changes from breaking older code
  - Guarantee software works as intended
- Quicker software development
  - Less bugs means less time spend bug fixing
  - Developing test cases forces better understanding of the problem you are trying to solve

Implementing TDD

- TDD is about testing the components, modules, objects, methods, . . . of a system
- Unit Tests are written by the developers themselves
- The quality of a system is a product of the quality of its constituent components as assessed by the tests

The TDD Cycle



Types of Tests

- Unit testing can validate **state** or **behaviour**
  - State Tests: does the unit generate the correct outputs for a given set of inputs?
  - Behaviour Tests: is the correct sequence of sub-behaviours executed?

Table of Contents

State-Based Testing

- State-based Unit Tests are simple:
  - Create the object(s) you want to test
  - Invoke the relevant method(s)
  - Evaluate the output of the method(s)
- Lets go through an example of developing a string based calculator

Example

```
1 public interface StringCalculator {
2   public String add(String a, String b, int
     base);
3   public String subtract(String a, String b,
     int base);
4   public String multiply(String a, String b,
     int base);
5   public String divide(String a, String b,
     int base);
6 }
```

```
1 public class StringCalc implements
    StringCalculator {
```

Example

```
1  public class StringCalcTest {
2    private StringCalculator calc;
3
4    @Before
5    public void setupCalculator() {
6      calc = new StringCalc(new Converter());
7    }
8
9    @Test
10   public void testAddDec() {
11     assertEquals("5555", calc.add("1234",
       "4321", 10));
12   }
```

Requirements

- This would seem like a satisfactory test
- But without knowing the requirements we don't know if it is sufficient

String Calculator Requirements

Lets look at the overall requirements for the class (focusing on add)

- Should work for decimal and hexadecimal (base is final parameter)
- The add method should add the two numbers together and return the result as a string
- It should work with negative numbers (always preceded by - not two's complement)
- Invalid numbers should cause an exception (e.g. A in an decimal number or G in hexadecimal)

TTD Steps

1. Identify the Test Cases
2. Create Unit Tests
3. Implement Code
4. Run Tests

1 - Identify Test Cases add

- Positive Cases:
  - Add decimal: "1234", "4321", 10 (returns "5555")
  - Add hex: "AA33", "11FE", 16 (returns "BC31")
  - Add decimal negative: "-1234", "-4321", 10 (returns "-5555")
  - Add hex negative: "-AA33", "-11FE", 16 (returns "-BC31")
- Negative Cases:
  - Invalid number decimal: "0101", "AA78", 10 (throws exception)
  - Invalid number hex: "0101", "GA78", 16 (throws exception)

2 - Unit Tests - Setup

```
1 public class StringCalcTest {
2   private StringCalculator calc;
3
4   @Before
5   public void setupCalculator() {
6     calc = new StringCalc(new Converter());
7   }
```

2 - Unit Tests - add Decimal and Hex

```
9   @Test
10  public void testAddDec() {
11    assertEquals("5555", calc.add("1234",
     "4321", 10));
12  }
13
14  @Test
15  public void testAddHex() {
16    assertEquals("BC31", calc.add("AA33",
     "11FE", 16));
17  }
```

2 - Unit Tests - Negative

```
19    @Test
20    public void testNegativeHex() {
21      assertEquals("-5555", calc.add("-1234",
       "-4321", 16));
22    }
23
24    @Test
25    public void testNegativeDec() {
26      assertEquals("-5555", calc.add("-1234",
       "-4321", 10));
27    }
```

2 - Unit Tests - Invalid Number

```
29    @Test(expected =
       InvalidNumberForBaseException.class)
30    public final void InvalidNumberUsedHex() {
31      calc.add("0101", "12G4", 16);
32    }
33
34    @Test(expected =
       InvalidNumberForBaseException.class)
35    public final void
       InvalidNumberUsedDecimal() {
36      calc.add("0101", "AA78", 10);
37    }
```

3 - Write Class

```
1  public class StringCalc implements StringCalculator {
2    private Converter conv;
3
4    public StringCalc(Converter c) {
5      conv = c;
6    }
7
8    public String add(String a, String b, int base) {
9      long v1 = conv.convertToNum(a, base);
10     long v2 = conv.convertToNum(b, base);
11     long result = v1 + v2;
12     String r = conv.convertToString(result, base);
13     return r;
14   }
```

4 - Run Tests

- When we run the test we expect them to pass
- For positive cases this means the values were as expected
- For negative cases, this means the exception was generated
- We iteratively do steps 3 and 4 until all tests pass

Running Unit Tests

Executing Unit tests in eclipse is very easy, in the package explorer you right-click on the test class and select **Run As->JUnit Test**

- Eclipse will show a new tab on the left
- This tab will show the details of the test run
- It shows all the classes tested and all of the tests run

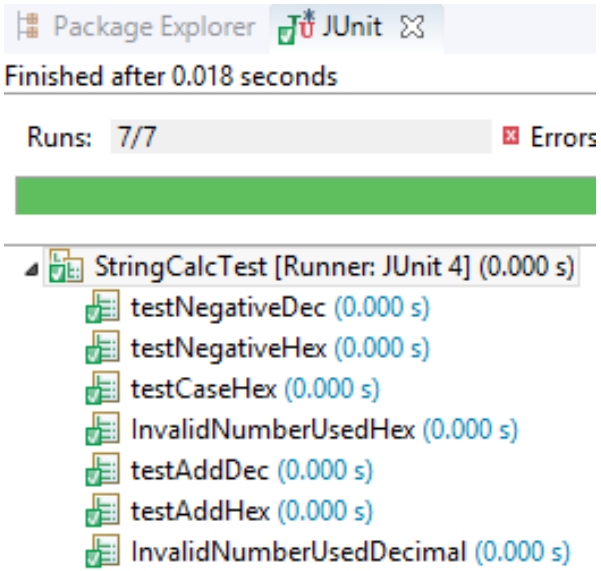


Table of Contents

Behaviour Testing

- Testing behaviour is more complicated than testing state
  - We want to know what other sub-behaviours (methods/functions) were executed; in what order; and with what parameters
  - This is not a normal feature of a language
  - Need a mechanism to intercept sub-behaviour calls

Behaviour Testing Approach

- The approach is very similar to state tests:
  - Create the object(s) you want to test (the mock object)
  - Invoke (call) the relevant method(s) – log all method calls to the mocked object (trace)
  - Inspect the method call trace

Reflection API

- We can trace in Java by creating mock objects
- We can do this with the **Reflection API**
  - This is a set of classes for representing class, interfaces, methods and so on
  - The most common use of the API is to create an instance of a class based on a string representation of its canonical name
- ```
  Object obj =
  Class.forName("java.lang.Object").newInstance();
  ```
- Implementing this is extremely complicated and difficult to get right
  ```

Better Approach

- A better approach to behaviour testing is to use a framework like **Mockito**
- Mockito allows us to use mock objects by adding some annotations to our code
- We can even specify what answers the mock object should give when called

Add Behaviour

- We expect that the behaviour of the add method will be to convert both strings to numbers perform the addition and then convert the result back to a string
- These conversions are done using the Converter object
- We can test that these interactions happen the way they are supposed to, without having to write the methods

Mocking Objects

- We can create a mock of an object by declaring it with the `@Mock` annotation e.g. `@Mock Converter conv;`
- We can tell Mockito what the response should be using the `when().thenReturn()` syntax
- E.g. `when(conv.convertToNum("321", 10)).thenReturn(321L);`

Testing Mocked Objects

- We test that the method was actually called using the verify function
- This allows us to specify the parameters that were used when a method was called
- E.g. `verify(conv).convertToNum("321", 10);`
- If the method was not called with these parameters, the test will fail

Testing add Behaviour

```
1  @Mock Converter conv;
2  @Rule public MockitoRule mockitoRule =
       MockitoJUnit.rule();
3  @Test
4  public void testQuery() {
5    when(conv.convertToNum("321", 10)).thenReturn(321L);
6    when(conv.convertToNum("123", 10)).thenReturn(123L);
7    when(conv.convertToString(444,
       10)).thenReturn("444");
8    StringCalc calc = new StringCalc(conv);
9    assertEquals("444", calc.add("123", "321", 10));
10   verify(conv).convertToNum("123", 10);
11   verify(conv).convertToNum("321", 10);
12   verify(conv).convertToString(444, 10);
13 }
```

Table of Contents

Code Coverage

- How well our code is tested?
- Code coverage is a measurement of the code that has been tested
- This is generally measured by tools and expressed as a percentage of the total code

Code Coverage

- There are a number of coverage criteria, the main ones being:
  - Function coverage - Has each function in the application been called by the test code?
  - Statement coverage - Has each statement in the application been called by the test code?
  - Branch coverage - Has each branch of each control structure (such as in if and case statements) been executed by the test code?
  - Condition coverage - Has each Boolean sub-expression evaluated both to true and false?

Coverage Example

```
1 public int foo (int x, int y) {
2     int z = 0;
3     if ((x>0) && (y>0)) {
4         z = x;
5     }
6     return z;
7 }
```

Coverage Example

- If during the execution of the tests the method 'foo' was called at least once, then function coverage for this function is satisfied
- If during the execution of the tests every statement was executed, then statement coverage is satisfied. E.g. `foo(1,1)`
- If during the execution of the tests every possibility for each if statement is executed (true and false), then branch coverage is satisfied. E.g. `foo(1,1)` and `foo(0,1)`

Condition Coverage Example

- If during the execution of the tests every possibility for each part of the condition in an if statement is executed (true and false), then condition coverage is satisfied. To achieve this we need to execute the tests calling `foo(1,1)`, `foo(1,0)` and `foo(0,0)`.
  - In the first case both parts of the condition `(x>0)` and `(y>0)` are evaluated as true
  - In the second case, the first part is evaluated as true and the second as false
  - In the third case, the first part is evaluated as false and the second is not evaluated.

Measuring Code Coverage

- Code Coverage is measured as a percentage
- This leads to the question of how much should I test?
- The answer is "It Depends"
  - Some say we should focus on the logic and functionality
  - It is quite reasonable to have a, say, 50% coverage rate if only because only 50% of the code contains logic that can be tested, and the other 50% happens to be simple objects or things that are handled by a framework
  - TDD recommends 100% code coverage