Object-Oriented Programming

Testing

Dr. Seán Russell
sean.russell@ucd.ie

School of Computer Science,
University College Dublin

November 13, 2018

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
- be able to create test classes using JUnit
- be able to design and implement basic tests using JUnit
- be familiar with the use of tests to find errors in code

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
- When talking about professional software engineering, testing is evaluation of the software against requirements gathered from users and system specifications
- 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 our programs in some way, it is very difficult to know if they are correct or not
- Even when some testing is done, if it is not very thorough there may be mistakes that we have not found

Traditional Testing

- When testing the entire program, it can be very difficult to test every part of the system
- E.g. a program that reads some coordinates from the user and tells prints a message telling the user if they are inside a rectangle or not
- We would need to test the program in both situations (inside and outside) to see if it is correct
- In more complicated programs where there are many situations like this, and testing each one of them would be very difficult
- Additionally, as we are testing the whole program when we find that something is incorrect, it can be difficult to know where this error actually is

Testing strategies

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

- Adding print statements
- Using a debugger
- Testing scripts

Print statements

Adding print statements to a program is the easiest form of testing

- The programmer adds print statements to their code that print out the values of variables at different times
- Only the programmer knows the meaning and expected output of the print statements
- This creates a lot of output and can be hard to follow in complex applications
- Testing has to be removed when software is finished (Users should not see this information) - Cannot be used for further testing

Print Statements Example

- Take a very simple example of adding two numbers together, to make sure it was operating correctly we would add a print statement afterwards that prints the value of both variables and the result
- We can then verify that the result was correct

```java
int result = x + y;
System.out.println(result + " = " + x + " + " + y);
```

Problems with this Type of Testing

- Only the original programmer understands meaning and expected output of the print statements
- Another programmer might not recognise when the output was incorrect
- Additionally, this type of testing adds a lot of output and can be difficult to understand in complex applications (particularly if there are multiple threads)
- Finally, this test code has to be removed when the software is completed
- If any further problems are identified, we can have to add new code for testing to find the source of the problem

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 behind unit testing is that we test each component of the system individually
- If we are writing a method that finds the largest value in an array, we would write test for just this method rather that the entire program
- We would write tests for all of the components
- We find errors earlier because they can be found as soon as an individual component is completed

Unit teating ideals

The idea behind unit testing is that all tests should be...

- **Isolatable**
  - We should be able to run the test without the rest of the program being executed
- **Repeatable**
- **Automatable**
  - We should be able to have the tests run automatically
- **Easy to write**

Benefits of Unit Testing

There are a lot of benefits to using unit testing in the development in large programs

- Faster debugging
  - Unit tests cover a smaller amount of code, when there is an error it is easier to locate it
- Faster development
  - Less time is spent debugging
  - New code can be added knowing that the tested code work correctly
- Better design
  - Unit tests make developers focus more on what classes should do
- Reduces future costs
  - Unit testing takes time at the beginning but saves time over the long term

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(x, 10);` 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

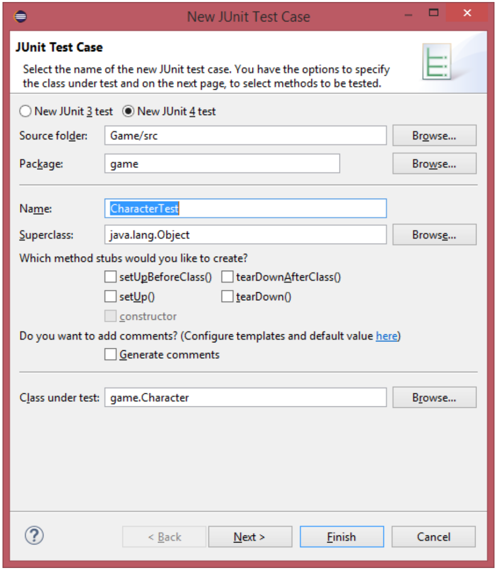
- The assertSame declaration takes two objects as parameters and checks to see if in fact they are two references to the same object in memory
- This is is similar to assertEquals, but even if two objects are equal (contain the same data) they may be separate in memory

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 `Character` class, we would create a class called `CharacterTest`
- 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


New JUnit Test Case

JUnit Test Case

Select the name of the new JUnit test case. You have the options to specify the class under test and on the next page, to select methods to be tested.

( ) New JUnit 3 test [✓ New JUnit 4 test]

Source folder: Game/src

Package: game

Name: CharacterTest

Superclass: java.lang.Object

Which method stubs would you like to create?

- [ ] setUpBeforeClass()
- [ ] tearDownAfterClass()
- [ ] setUp()
- [ ] tearDown()
- [ ] constructor

Do you want to add comments? (Configure templates and default value here)

- [ ] Generate comments

Class under test: game.Character

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

```java
@Test
public void testSomething(){
  // do some testing
}
```

Table of Contents

Knowing What to Test

- Before we can write test code we need to understand exactly what the result of the code we are testing should be
- We will use examples of a `StringCalculator` class that works with strings, it contains the following methods
  - `public int convertToInt(String number)`
  - `public String convertToString(int i)`
  - `public String add(String n1, String n2)`
  - `public String divide(String n1, String n2)`
  - `public String subtract(String n1, String n2)`
  - `public String multiply(String n1, String n2)`

Testing conversion to Integer

- The test case should be given a descriptive name such as `testConvertToInt`
- The first step is that we need to create a calculator object to test
- Then we call the method and check the answer is what we expected
- The String calculator is designed to use different bases, so we should test with different bases

Testing conversion to Integer

```
@Test
void testConvertToInt() {
  StringCalculator sc = new StringCalculator(10);
  int ans = sc.convertToInt("1234");
  assertEquals(1234, ans);

  sc = new StringCalculator(16);
  ans = sc.convertToInt("FF");
  assertEquals(0xFF, ans);
}
```

Testing conversion to String

- The test case should be given a descriptive name such as `testConvertToString`
- The first step is that we need to create a calculator object to test
- Then we call the method and check the answer is what we expected
- The String calculator is designed to use different bases, so we should test with different bases

Testing conversion to String

```java
@Test
void testConvertToString() {
  StringCalculator sc = new StringCalculator(10);
  String ans = sc.convertToString(1234);
  assertEquals("1234", ans);

  sc = new StringCalculator(16);
  ans = sc.convertToString(0xFE);
  assertEquals("FE", ans);
}
```

Testing Addition

- The test case should be given a descriptive name such as `testAdd`
- The first step is that we need to create a calculator object to test
- Then we call the method and check the answer is what we expected
- The String calculator is designed to use different bases, so we should test with different bases

Testing conversion to String

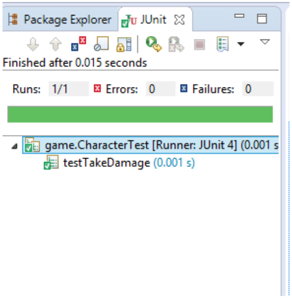
```
@Test
void testAdd() {
  StringCalculator sc = new StringCalculator(10);
  String ans = sc.add("1234", "5678");
  assertEquals("6912", ans);

  sc = new StringCalculator(16);
  ans = sc.add("AAAA", "BBBB");
  assertEquals("16665", ans);
}
```

Table of Contents

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



Finished after 0.015 seconds

Runs: 1/1 Errors: 0 Failures: 0

game.CharacterTest [Runner: JUnit 4] (0.001 s
testTakeDamage (0.001 s)

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

```java
public int foo (int x, int y) {
    int z = 0;
    if ((x>0) && (y>0)) {
        z = x;
    }
    return z;
}
```

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”
  - We should focus on the logic and functionality rather than testing everything
  - Different applications will have different percentages of code that should be tested
  - 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