

Test Driven Development

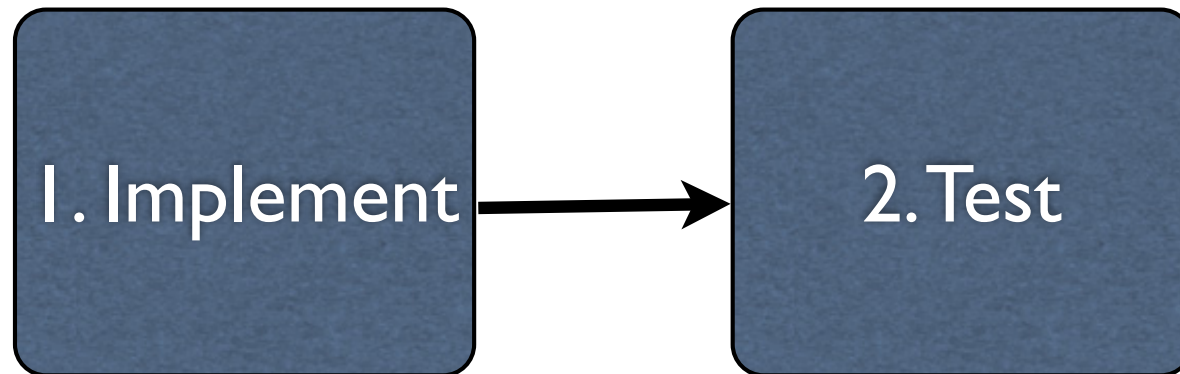
Mel Ó Cinnéide
School of Computer Science
University College Dublin

Test Driven Development (TDD)

The notion of developing automated tests has led to the idea of making testing more central to the development process.

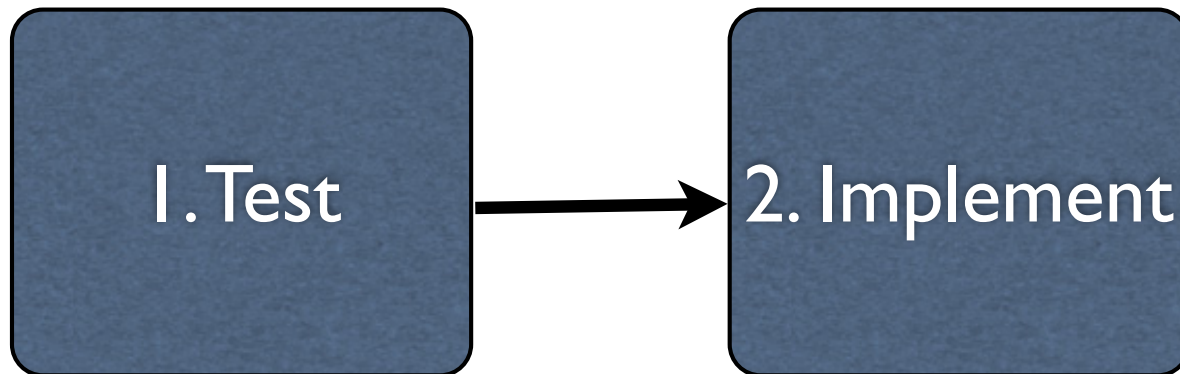
We'll look at this idea in the coming slides...

“Traditional” Order of Implementation and Testing

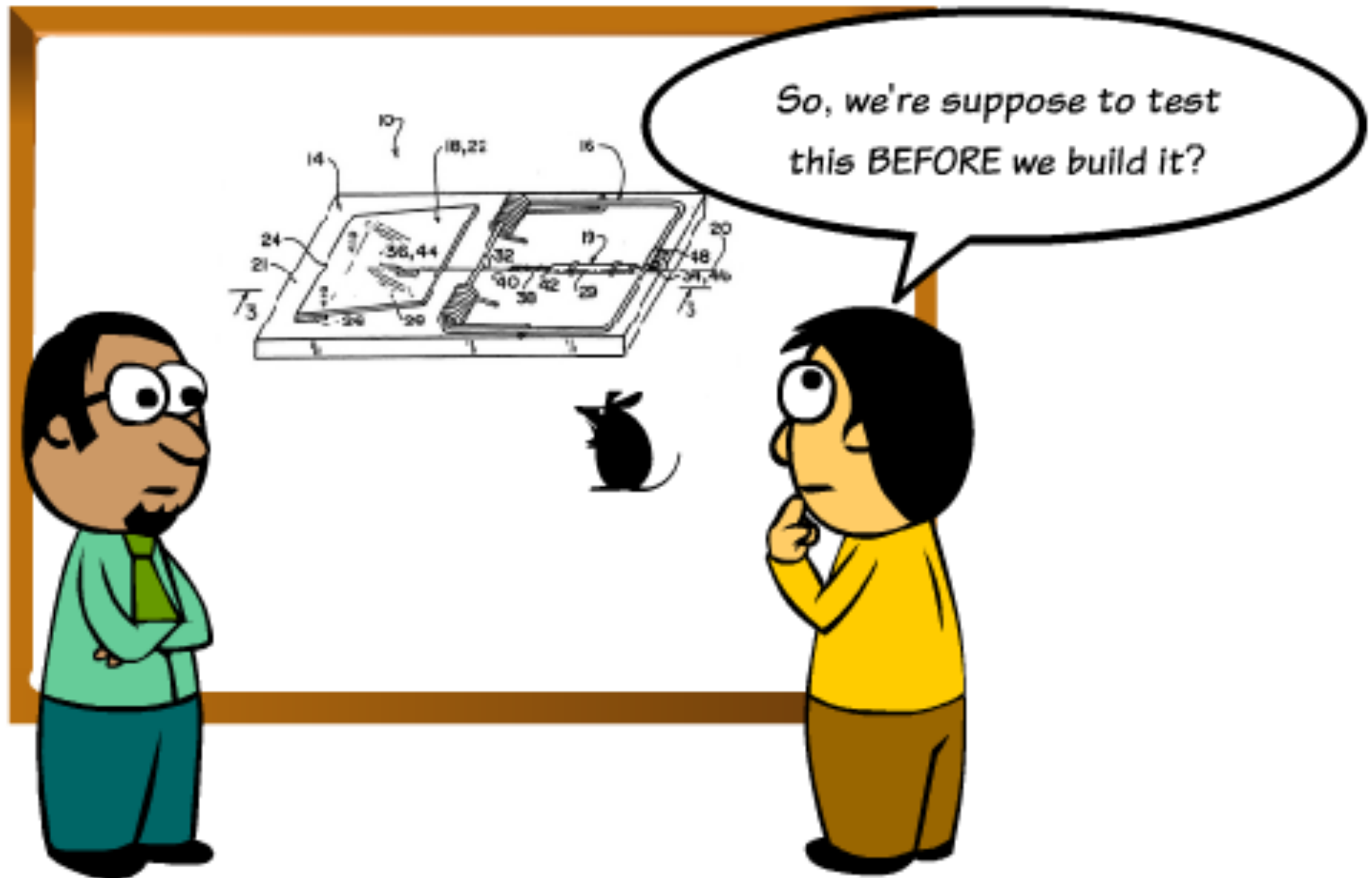


Test-Driven Development (**TDD**)

Aka **Test-First** Development



How can you test before you implement?



A Specification is like a Test

Any specification implies certain tests that the program must pass, e.g.

Write a method that accepts a array of integers and returns their average value, e.g. for the input

[1,2,3,4,5]

the value 3 will be returned.

The specification defines the whole range of input/output pairs, and in this case even provides a concrete example.

The difference with test-first is that we use the unit tests to *drive* the development of the software.

Test-First isn't that unusual

Although test-first may seem odd at first, it's not that uncommon in the real world.

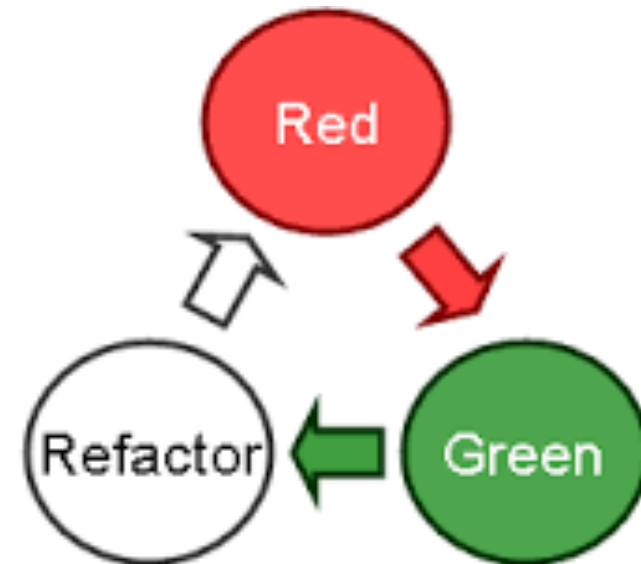
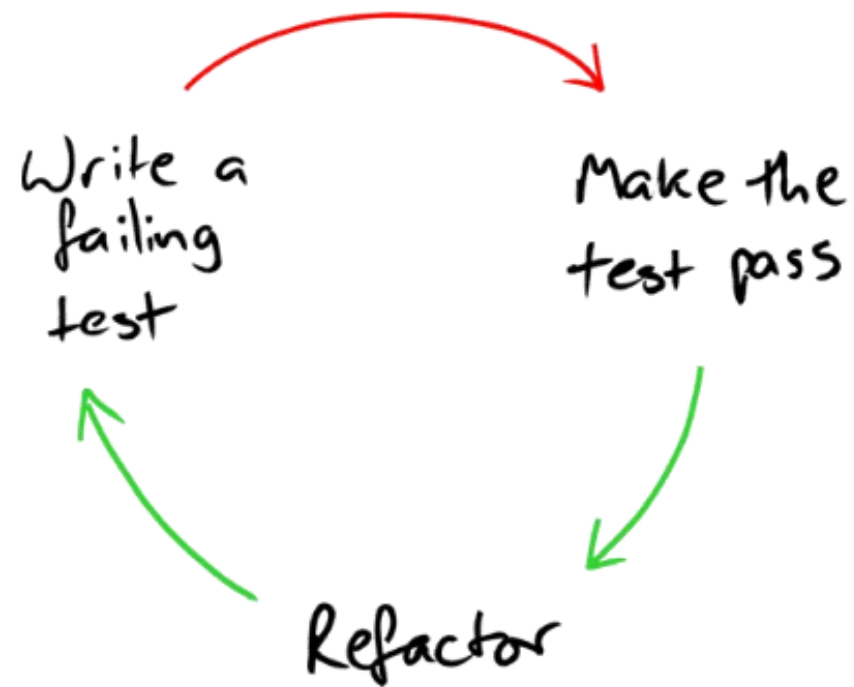


E.g., the Citroen Dyane was originally specified as follows:

- enable four farmers to drive 50 kg of farm goods at 50 km/h across muddy unpaved roads.
- Use no more than 3 litres of petrol to travel 100 km
- Drive across a ploughed field while carrying eggs without breaking them.

In a sense, any specification describes a set of tests that determine if an artefact is a correct implementation.

Process of TDD



Red	Write a test that fails.
Green	Write code to pass the test.
Refactor	Clean up your code and the test.

JUnit 4

JUnit 4 is a industrial-strength unit testing framework.

It's a large, quite complex framework that supports different styles of unit testing.

We're only using the basics of it, so don't get led astray when reading more about JUnit online.

Anatomy of JUnit 4 unit tests

```
import static org.junit.Assert.*;  
import org.junit.Test;
```

use the **JUnit**
testing framework

```
public class FooBarTest {
```

Convention: FooBarTest has
test cases for the FooBar class.

```
@Test    tells JUnit that this is a unit test
```

```
public void testOne() {
```

```
    ...
```

```
    assertTrue(...);
```

```
}
```

Convention: unit tests
are named **testXxx**.

```
@Test
```

```
public void testTwo() {
```

```
    ...
```

```
    assertEquals(...);
```

```
}
```

```
}
```

Test methods will always
contain **assertions**.

Types of Assertion: `assertTrue`

The most basic assertion you use in testing is `assertTrue`.

```
assertTrue(errorMessage, booleanExpression);
```

Example:

```
assertTrue("day not in [1..31]", day >= 1 && day <= 31);
```

So `assertTrue` an expression that should be true.

If it is true, nothing happens (the unit test passes).

If it's false, `<errorMessage>` is output.

Summary of basic assertions

<code>assertTrue([message,] boolean condition)</code>	Checks that the boolean condition is true.
<code>assertFalse([message,] boolean condition)</code>	Checks that the boolean condition is false.
<code>assertEquals([message,] expected, actual)</code>	Tests that two values are the same. Note: for arrays the reference is checked not the content of the arrays.
<code>assertEquals([message,] expected, actual, tolerance)</code>	Test that float or double values match. The tolerance is the number of decimals which must be the same.
<code>assertNull([message,] object)</code>	Checks that the object is null.
<code>assertNotNull([message,] object)</code>	Checks that the object is not null.
<code>assertSame([message,] expected, actual)</code>	Checks that both variables refer to the same object.
<code>assertNotSame([message,] expected, actual)</code>	Checks that both variables refer to different objects.

@Before and @After annotations

There may be some setup tasks that all test methods have in common; there may similarly be teardown tasks that they all require.

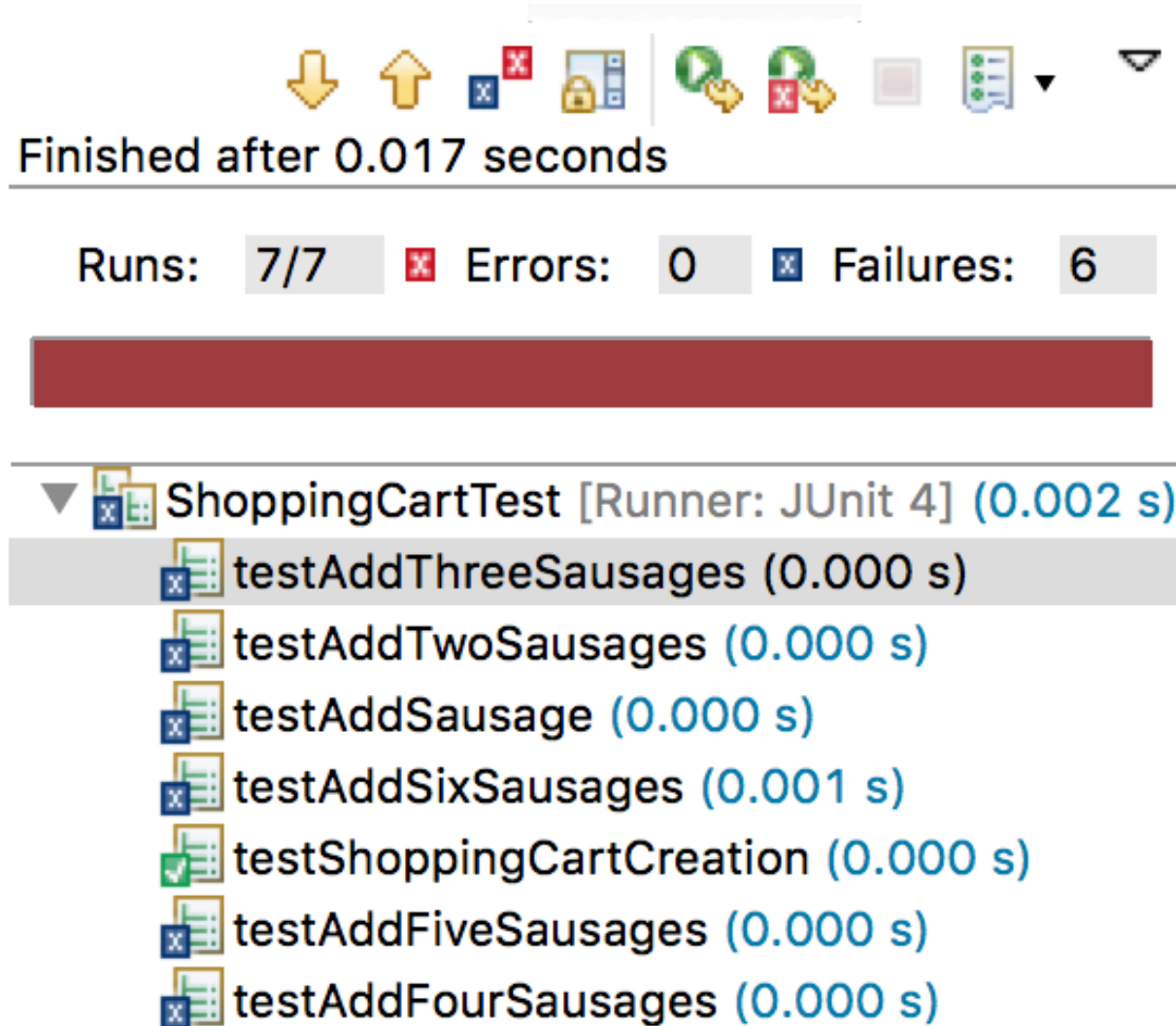
Methods with the **@Before** annotation are executed **before** each unit test:

```
@Before
public void setup(){
    myCart = new Cart();
}
```

Methods with the **@After** annotation are executed **after** each unit test:

```
@After
public void teardown(){
    myCart = null;
}
```



Write a Unit Tests that fails -- **RED**












Unit Tests should fail at first...

This provokes you to write code to make them pass.

Write code to make the Unit Test pass -- **GREEN**

Runs: 7/7  Errors: 0  Failures: 0

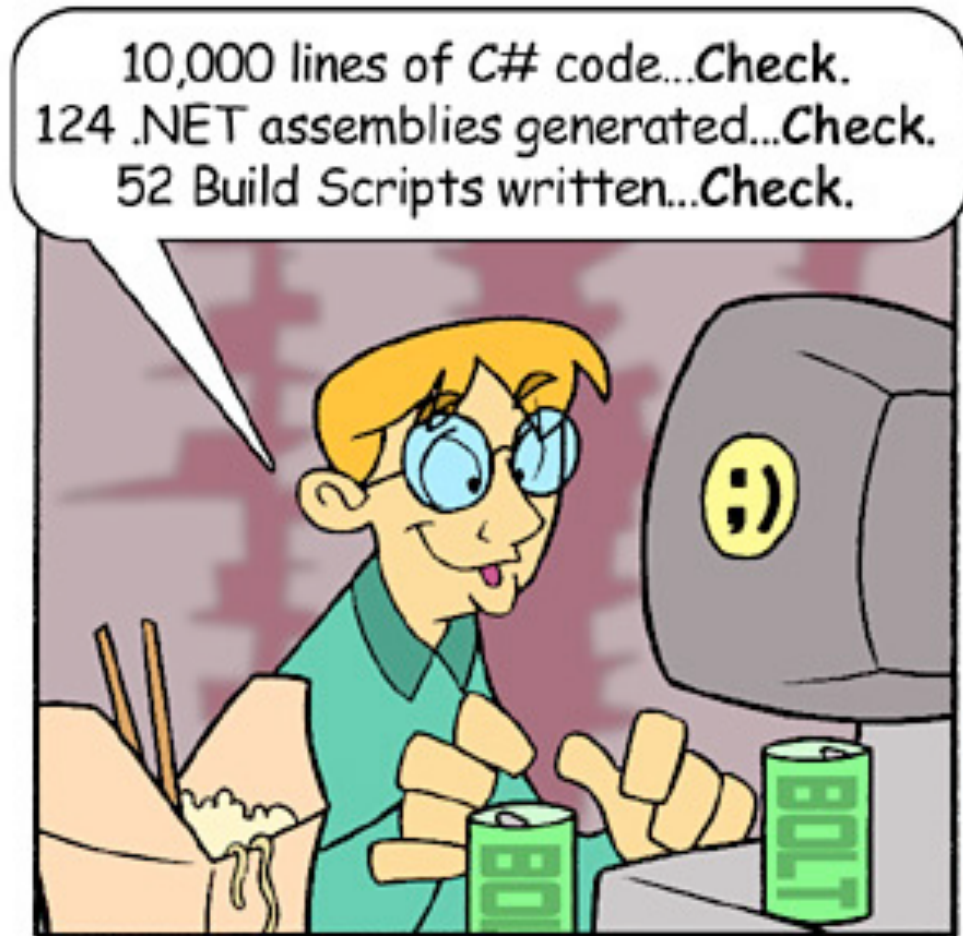
- 
-
- ▼  ShoppingCartTest [Runner: JUnit 4] (0.001 s)
-  testAddThreeSausages (0.000 s)
 -  testAddTwoSausages (0.000 s)
 -  testAddSausage (0.000 s)
 -  testAddSixSausages (0.000 s)
 -  testShoppingCartCreation (0.000 s)
 -  testAddFiveSausages (0.001 s)
 -  testAddFourSausages (0.000 s)

Now tidy up your code -- **REFACTOR**



Tidy up your code, make sure there is no duplicated code or ugly bits. (We return to **refactoring** later in the module.)

Test cases can take up a lot of code



... but this is not a bad thing.

Key Points

Testing is a vital part of software development.

Unit testing involves writing test cases for individual classes. A **unit testing framework** like JUnit can manage these test cases and run them on command.

Test-Driven Development (TDD) is where test cases are written first, as a way of driving the development of software. This is a key part of the **Agile** approach to software development.

In the lab today you'll be using TDD for a very simple example, and then for a more challenging one.

TDD Demo

Here we work through an example of using TDD to develop a simple class. Some points to note about the demo:

- The example is trivial of course; the purpose is to illustrate TDD
- We take very small steps, which is what you do when you're unsure of the problem. In a simple case like this we could have made larger steps.
- At every stage we have working code, and can confirm what it works by rerunning the unit tests

Developing a Shopping Cart using TDD

Problem: Create a simple shopping cart using TDD

The shopping cart may contain two types of product:

- * packets of sausages at €5 each
- * packets of bacon at €6 each

There's a “two for the price of one offer” on sausages

We'll apply TDD rigorously.



Example based on <http://www.basicsbehind.com/tdd-by-example/>

Test Case #1: New cart is empty

A newly-created cart should have no products in it.

As all our testcases rely on a cart object, so we use before and after methods to create/delete the cart:

```
@Before
public void setup(){
    cart = new ShoppingCart();
}
```

```
@After
public void teardown(){
    cart = null;
}
```

Then our test case is:

```
@Test
public void testShoppingCartCreation() {
    assertEquals("product count not zero", 0, cart.noOfSausages());
}
```

Test Case #2: Add Sausages

Add one unit of Irish Sausage, unit price €5

Then:

- product count should be 1
- the total value of cart should be €5



Test Case #3: Add More Sausages

Add two unit of Irish Sausage, unit price €5

Then:

- product count should be 2
- the total value of cart should be €10



Test Case #4: Add More Sausages again

Add three units of Irish Sausage, unit price €5, taking 2-for-1 offer into account.

Then:

- product count should be 3
- the total value of cart should be €10

etc.

etc.

