Ollscoil na Gaillimhe
University of Galway

CT326 Programming III

Lecture 3
Unit Testing in Java & Test Driven Development Part 1

- Dr. Adrian Clear -
School of Computer Science

Objectives…

- By the end of this lecture, you will:

1. have been introduced to testing terminology and the main kinds of testing
2. understand the practice of Test Driven Development (TDD)
3. have been introduced to unit testing and JUnit, the Java unit testing framework
4. have written some tests using a TDD approach

Testing

- Systematic process of analysing a system or system component to detect the differences between specified (required) and observed (existing) behavior.
- Attempt to show that the implementation of a system is inconsistent with the desired functionality
- Goal is to design tests that exercise defects in the system and to reveal problems
- Can’t test everything in a large system.
  - Tradeoffs required with budget and time constraints.

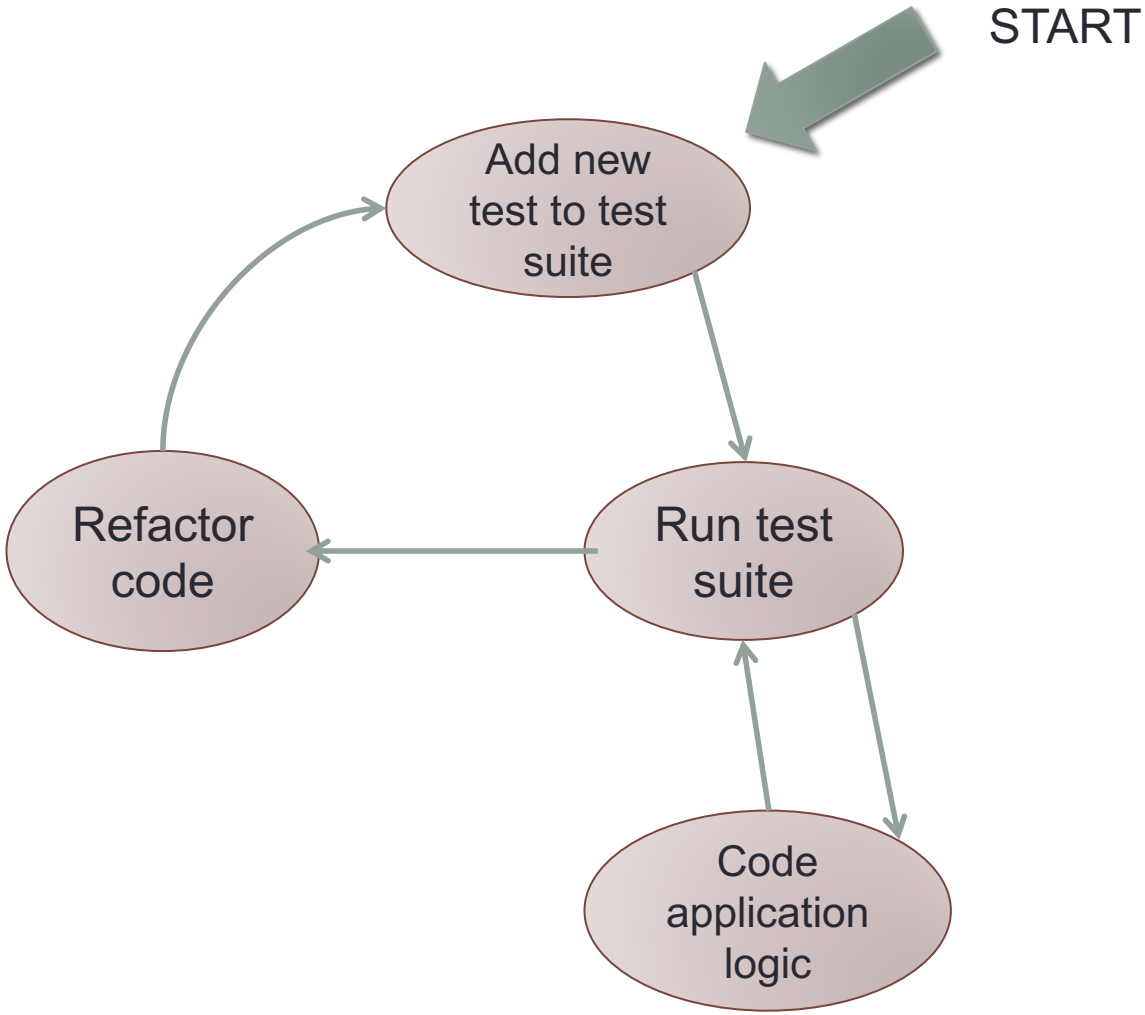
Levels of testing

- **Unit testing** involves testing individual classes and mechanisms.
- **Integration testing** involves testing groups of classes or components and the interfaces between them.
- **System testing** involves integration testing the system as a whole to check it meets the requirements.

Test-Driven Development (TDD)

- A software development process that relies on the repetition of a very short development cycle
- The general rationale behind this methodology is "*first write the test, then the code*" such that the tests drive the development of your code
- Tries to find faults in participating objects and/or subsystems with respect to the use cases from the use case model.

The TDD cycle



Red, Green, Refactor!

- A test will initially fail; we write a minimal amount of code to make a test pass
- Refactor our application and test code before moving on to the next one
- Build a test suite as our implementation progresses

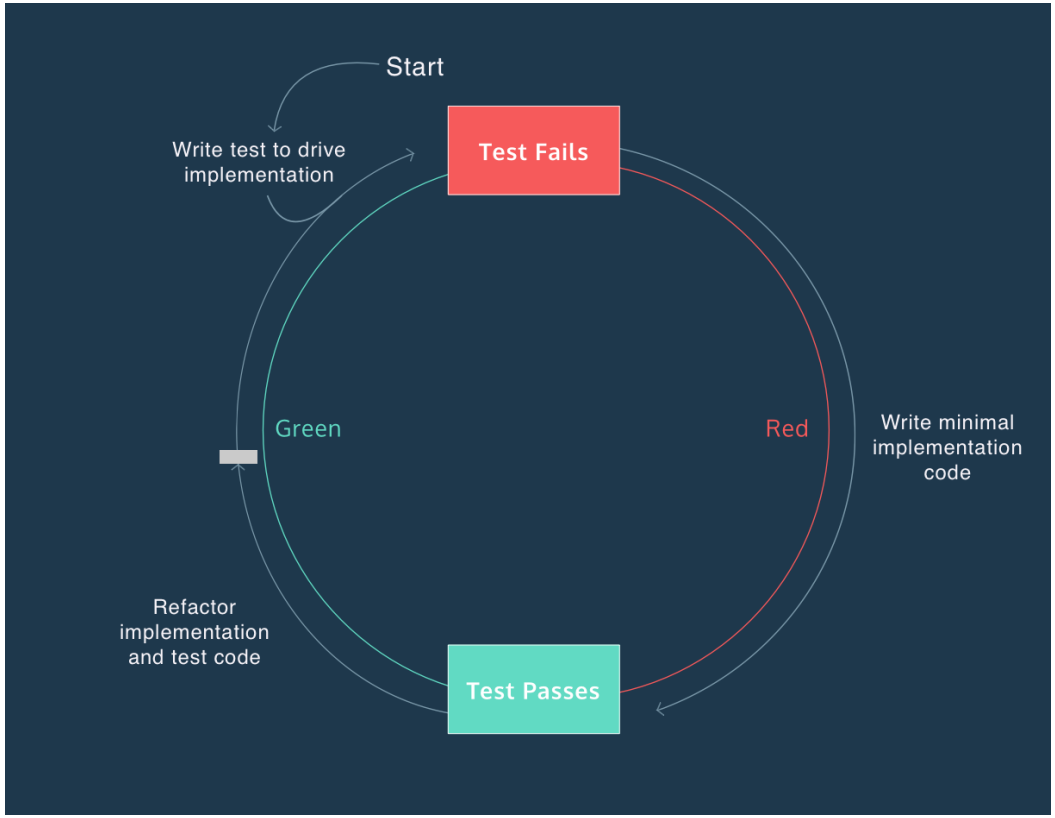
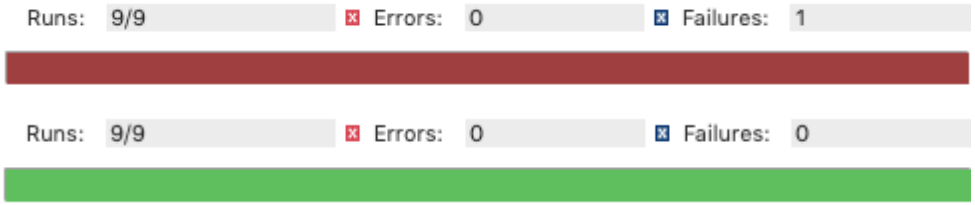


Image from www.codecademy.com



Use case descriptions for a shopping cart

Use case description: **Add item(s) to cart**

1. Customer adds items to their shopping cart by selecting the item to add and indicating the number that they wish to add.
2. System confirms that items have been added successfully.

Use case description: **Remove item(s) from cart**

1. Customer performs “View shopping cart” use case
2. Customer selects an item to remove.
3. System removes the item from the cart such that the quantity of the item in the cart is 0.

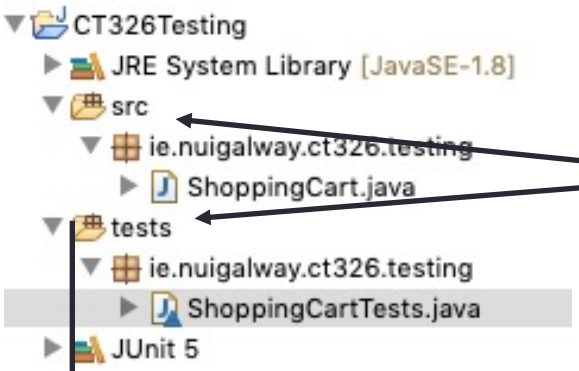
Use case description: **Update number of items in cart**

1. Customer performs “View shopping cart” use case
2. Customer indicates the new number of a particular item that they wish to have in their cart.
3. System changes the number of the item in the cart and updates the display.

Use case description: **View shopping cart**

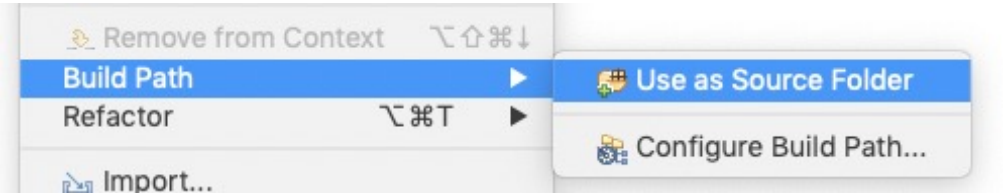
1. Customer indicates that they wish to view all of the items currently in their cart.
2. System displays items currently contained in the shopping cart

TDD example in Eclipse



Good practice to separate tests from production code

Once you create a new folder for your tests, you'll need indicate that it should be used as a source folder



Getting started…

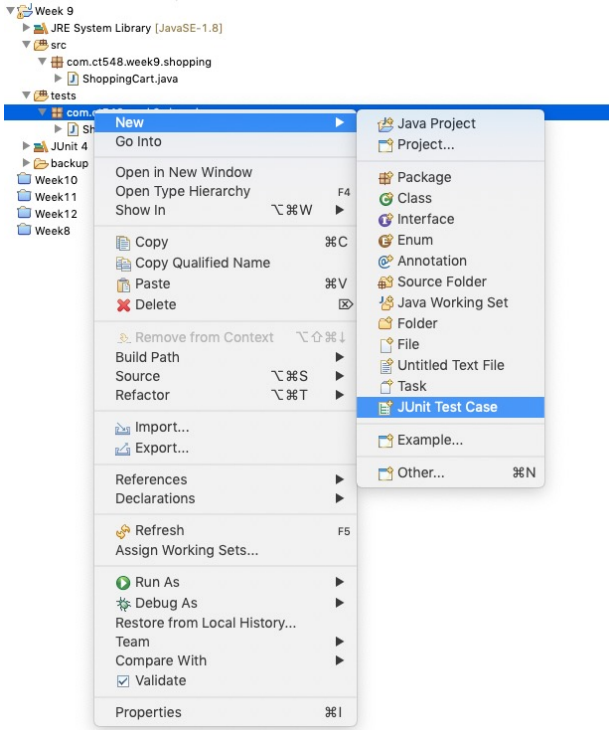
- We start with an empty ShoppingCart class

```java
package ie.nuigalway.ct326.testing;

/**
 * Class representing a shopping cart for an online supermarket.
 * @author Adrian Clear
 *
 */
public class ShoppingCart {

}
```

- Create our first Test Case by right-clicking on the tests package and selecting
  New --> Junit Test Case



- You may be prompted to add JUnit to your project. I'm using JUnit 5 here

Test cases

- A **test component** is a part of the system that can be isolated for testing
  - could be an object, a group of objects, or one or more subsystems.
- **Unit testing** finds differences between a specification of an object and its realisation as a component
- **JUnit** is a unit testing framework for test-driven development in Java
  - available in Eclipse out-of-the-box

Test cases

- A **test case** is a set of inputs and expected results that exercises a test component with the purpose of causing failures and detecting faults.
- **Blackbox tests** focus on the input/output behaviour of the component (i.e., the functionality, not the internal aspects)
- **Whitebox tests** focus on the internal structure and dynamics of the component

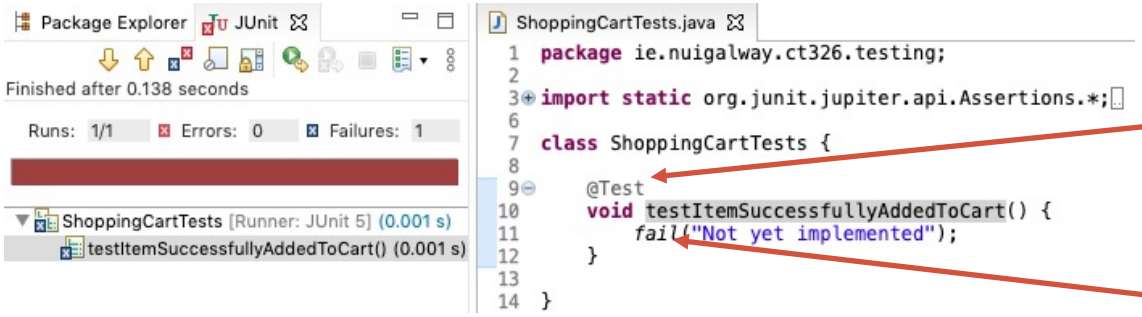
Our first test

- Let's start with the adding and item use case

  Use case description: **Add item(s) to cart**

  1. Customer adds items to their shopping cart by selecting the item to add and indicating the number that they wish to add.
  2. System confirms that items have been added successfully.

- …and a test that's "Red"



- Annotations tell JUnit that this is a test case
- The fail() assertion explicitly causes a test to fail

Let's write some test code

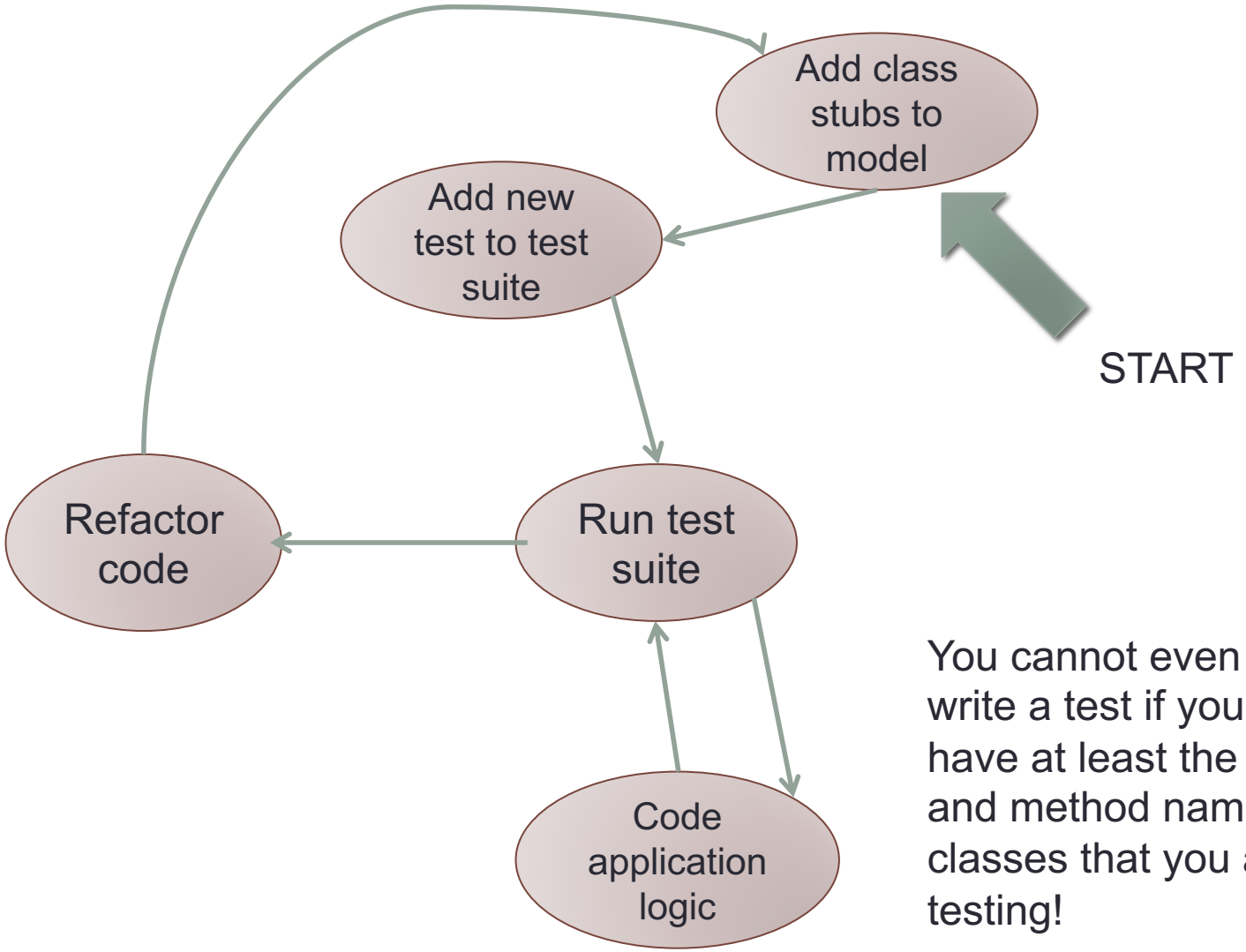
ShoppingCartTests.java | ShoppingCart.java

```
5  import org.junit.Test;
6
7  public class ShoppingCartTests {
8
9      @Test
10     public void testItemSuccessfullyAddedToCart() {
11         ShoppingCart myCart = new ShoppingCart();
12         myCart.add(new Item(new Product(), 1));      //I need to make a design decision here that
13                                                      //an item is represented as a quantity of products
14
15         assertTrue(myCart.getNoOfItems() == 1);      //Again, I need to make a design decision that a
16                                                      //Shopping Cart will contain some sort of list and
17                                                      //that the ShoppingCart class will have a method for
18                                                      //returning the total number of items in it.
19     }
20
21 }
```

Notice the red underline as these haven't been implemented yet

- JUnit uses assertions to indicate assumptions about the outcome of a test
- If the assertion is correct, the test passes; otherwise, it fails
- assertTrue(boolean statement): We assume the statement is true for an implementation that matches the specified requirement

The TDD cycle in OO development



You cannot even begin to write a test if you don't have at least the definitions and method names of the classes that you are testing!

Assert methods

- assertTrue(boolean ***test***)
  assertTrue(String ***message***, boolean ***test***)
- assertFalse(boolean ***test***)
  assertFalse(String ***message***, boolean ***test***)
- assertEquals(Object ***expected***, Object ***actual***)
  assertEquals(String ***message***, Object ***expected***, Object ***actual***)
- assertSame(Object ***expected***, Object ***actual***)
  assertSame(String ***message***, Object ***expected***, Object ***actual***)
- assertNotSame(Object ***expected***, Object ***actual***)
  assertNotSame(String ***message***, Object ***expected***, Object ***actual***)
- assertNull(Object ***object***)
  assertNull(String ***message***, Object ***object***)
- assertNotNull(Object ***object***)
  assertNotNull(String ***message***, Object ***object***)
- fail()
  fail(String ***message***)

Make your test compile

```java
public class ShoppingCart {

    public void add(Item item) {
        // TODO Auto-generated method stub

    }

    public int getNoOfItems() {
        // TODO Auto-generated method stub
        return 0;
    }

}
```

```java
package ie.nuigalway.ct326.testing;

public class Item {

    public Item(Product product, int i) {
        // TODO Auto-generated constructor stub
    }

}
```

```java
package ie.nuigalway.ct326.testing;

public class Product {

}
```

Package Explorer | JUnit

Finished after 0.14 seconds

Runs: 1/1 Errors: 0 Failures: 1

ShoppingCartTests [Runner: JUnit 5] (0.020 s)
testItemSuccessfullyAddedToCart() (0.020 s)

ShoppingCartTests.java | Product.java

```java
package ie.nuigalway.ct326.test

import static org.junit.jupiter

class ShoppingCartTests {

    @Test
    void testItemSuccessfullyAd
        ShoppingCart myCart = n
        myCart.add(new Item(new

        assertTrue(myCart.getNo
    }

}
```

…but it should still fail as we haven't implemented any functionality yet to make it pass.

“Go green”: Write minimum code to pass

- This is obviously not the correct implementation but it’s sufficient (the minimum code) to make our test pass
- Let’s make our test more sophisticated by adding a second item
- We’re back to red, indicating a deficiency with our previous implementation

```
public int getNoOfItems() {
    return 1;
}
```

Runs: 1/1 Errors: 0 Failures: 0

ShoppingCartTests [Runner: JUnit 5] (0.026 s)

Runs: 1/1 Errors: 0 Failures: 1

ShoppingCartTests [Runner: JUnit 5] (0.031 s)
testItemSuccessfullyAddedToCart() (0.031 s)

```
6
7  class ShoppingCartTests {
8
9      @Test
10     void testItemSuccessfullyAddedToCart() {
11         ShoppingCart myCart = new ShoppingCart();
12         myCart.add(new Item(new Product(), 1));
13
14         assertTrue(myCart.getNoOfItems() ==1);
15
16         myCart.add(new Item(new Product(), 3));
17         assertTrue(myCart.getNoOfItems() == 2);
18     }
19
20 }
```

We code again to make our test pass

Package Explorer | JUnit

Finished after 0.123 seconds

Runs: 1/1 Errors: 0 Failures: 0

ShoppingCartTests [Runner: JUnit 5] (0.024 s)

ShoppingCartTests.java | Product.java | Item.java | ShoppingCart.java

```java
package ie.nuigalway.ct326.testing;

import java.util.List;
import java.util.ArrayList;

/**
 * Class representing a shopping cart for an online supermarket.
 * @author Adrian Clear
 *
 */
public class ShoppingCart {

    List<Item> items;

    public ShoppingCart() {
        items = new ArrayList<Item>();          //I need to decide what data structure I'm going
    }                                           //to use for storing cart items.


    public void add(Item item) {
        items.add(item);
    }

    public int getNoOfItems() {
        return items.size();
    }

```

```java
@Test
public void testItemSuccessfullyAddedToCart() {
    ShoppingCart myCart = new ShoppingCart();
    myCart.add(new Item(new Product(), 1));
    assertTrue(myCart.getNoOfItems() == 1);

    myCart.add(new Item(new Product(), 3));
    assertTrue(myCart.getNoOfItems() == 2);
}
```

- Our first meaningful test goes green!

Demo: Account

- It should be possible to
  - Withdraw a non-negative amount from the account that doesn't exceed the balance
  - Deposit a non-negative amount to the account
  - query an account for its balance and account number
- An account must have an account number

- Use TDD to implement the withdraw functionality
  - Write a test for making a valid withdrawal
  - Red, Green, Refactor
- Use TDD to implement the “get account number” functionality