Modern TDD in JAVA

Introduction

Traditionally, testing software has been viewed as a manual task, taking place **after** the code has been written.

Test Driven Development shows us a different way.

We write **automated tests**, and we write them **while building the software**.

Introduction

Correctly applied, TDD doesn't just lead to an automated suite of tests.

It helps:

- To **drive and structure** the development process
- encouraging us to **think carefully about our APIs**
- giving us the confidence we need to **boldly refactor** our way towards better designs.

Straight To The Action: A TDD Spoiler

Introducing our demo domain

To give you a feel for how TDD looks in practice, we shall begin with a live demonstration.

We are building a class to represent a travel card for a city subway.

- Travelers buy a card for a number of journeys.
- On entering a station, they should tap their card on a sensor ("**touch in**").
- On leaving their destination station, they should do a similar ("**touch out**").

Our task

We need to implement and test the following requirements:

- Given the card is personal, we should not allow a traveler to touch in multiple times (they must touch out before touching in again)
- They also must not be allowed to touch out if they did not first touch in to the system.
- There should be a way for train staff to check card is currently in a "touched in" state

Dissecting the demo

Every test has a similar structure, where we do some setup or prepararation, followed by an operation. We then test the effects of the operation by doing **assertions**.

```
@Test
public void touchingInDecrementsBalance() {

    TravelCard sut = new TravelCard(15);

    // Touch in/out a couple of times
    sut.touchIn(); sut.touchOut();
    sut.touchIn(); sut.touchOut();

    Assert.assertEquals("Got decremented balance",
                        13, sut.getTravelBalance());
}
```

For instance, in the test above, we assert that the state has changed as expected after a series of touch in/out calls.

Dissecting the demo

In TDD, we aim to write the tests **before** the implementation

This lets us model the **interaction** with the class from the outside, rather than starting with the implementation details

```
@Test(expected = AlreadyTouchedInException.class)   <-- Test expecting exception
public void multipleTouchInsNotAllowed() {

   TravelCard sut = new TravelCard(5);

    sut.touchIn();
    sut.touchIn();   <-- Should throw exception if you try to touchIn twice
}
```

This test will **fail** since we haven't written the implementation.

Dissecting the demo

Getting the test to pass is simple in this case.

We just add this code to the **TravelCard** class:

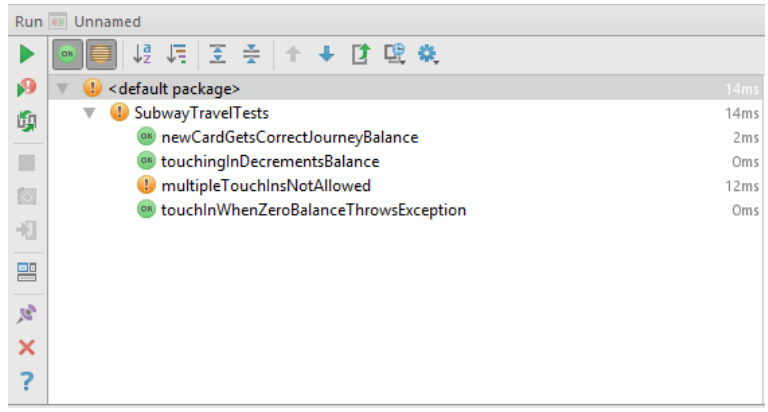
```
private int travelBalance;

public void touchIn() throws ZeroBalanceException {
   if (travelBalance == 0)
       throw new ZeroBalanceException("Not enough travels left on card");

   travelBalance--;
}
```


Dissecting the demo

Adding the implementation might introduce regression, causing one of our old tests to fail.



However, since we know **which** test is failing, the debugging scope gets significantly narrowed down.

Dissecting the demo

TDD in a nutshell:

We write tests to model interaction with our classes by implementing **failing** tests. Only when we have a failing test will we add new functionality.

This flow is usually described as **red/green/refactor**, and we'll talk more about it in the next module.

But first, let us back up a bit and talk about TDD in more general terms.