Introduction to TDD.NET

About This Course

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**.

About This Course

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

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

This course captures our collective experience of TDD in the real world: the theory, the practice, the wins, and the pitfalls.

Straight To The Action: A TDD Spoiler

Introducing our demo domain

EDUMENT

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

EDUMENT

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 preparation, followed by an operation. We then test the effects of the operation by doing **assertions**.

```
[Test]
public void TouchingInDecrementsBalance()
{
    var sut = new TravelCard(15);

    // Touch in/out a couple of times
    sut.TouchIn();
    sut.TouchOut();
    sut.TouchIn();
    sut.TouchOut();

    Assert.AreEqual(13, sut.TravelBalance, "Got decremented balance");
}
```

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

EDUMENT

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]
public void MultipleTouchInsNotAllowed()
{
    var sut = new TravelCard(5);

    Assert.Throws<AlreadyTouchedInException>(() =>
    {
        sut.TouchIn();
        sut.TouchIn();
    });
}
```

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

Dissecting the demo

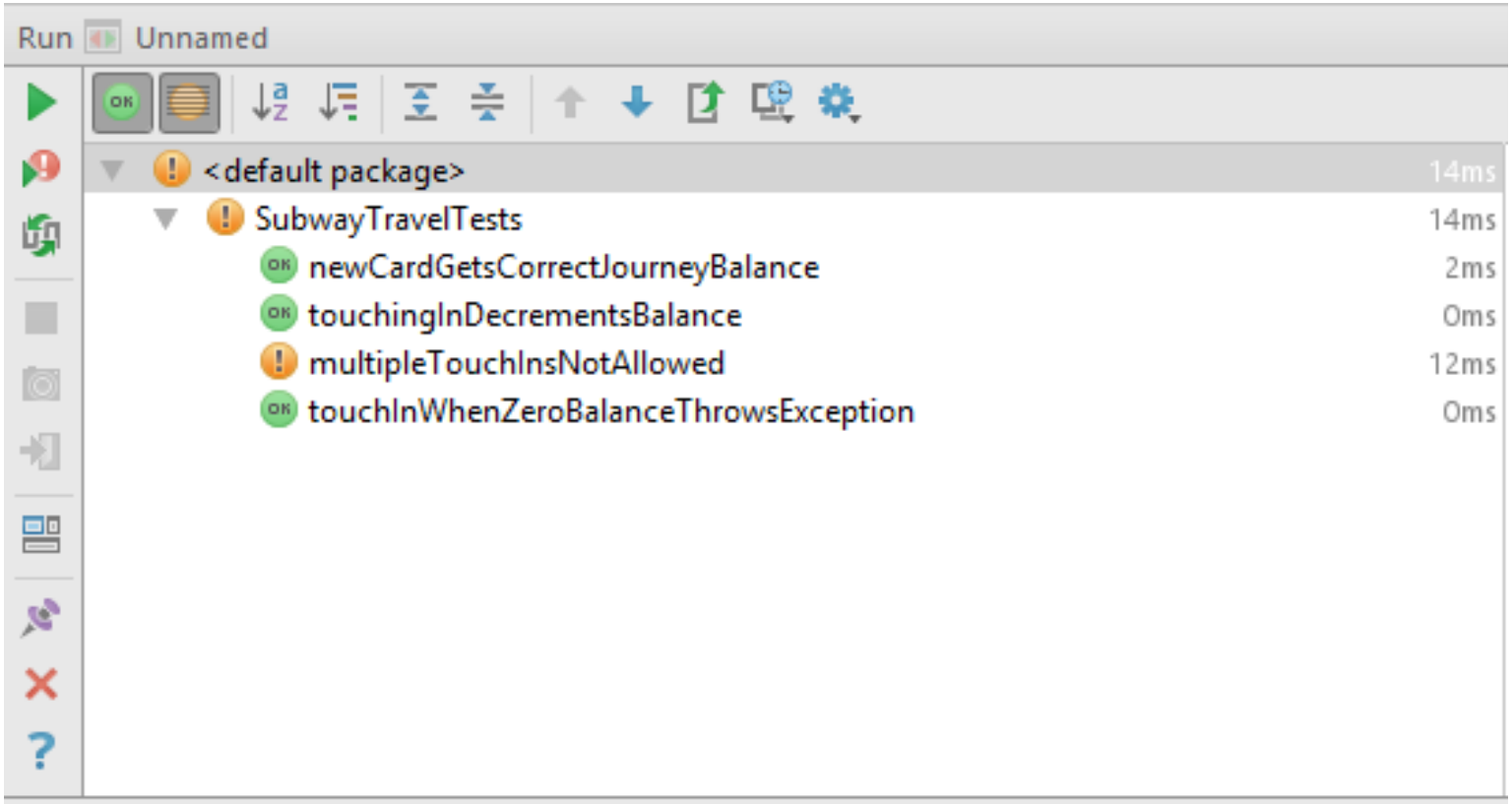
Getting the test to pass is simple in this case.

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

```
private bool touchedIn;

public void TouchIn()
{
    if (touchedIn)
        throw new AlreadyTouchedInException();

    touchedIn = true;

    //...
}
```

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.

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.