

# Test-Driven Development

Mel Ó Cinnéide  
School of Computer Science  
University College Dublin

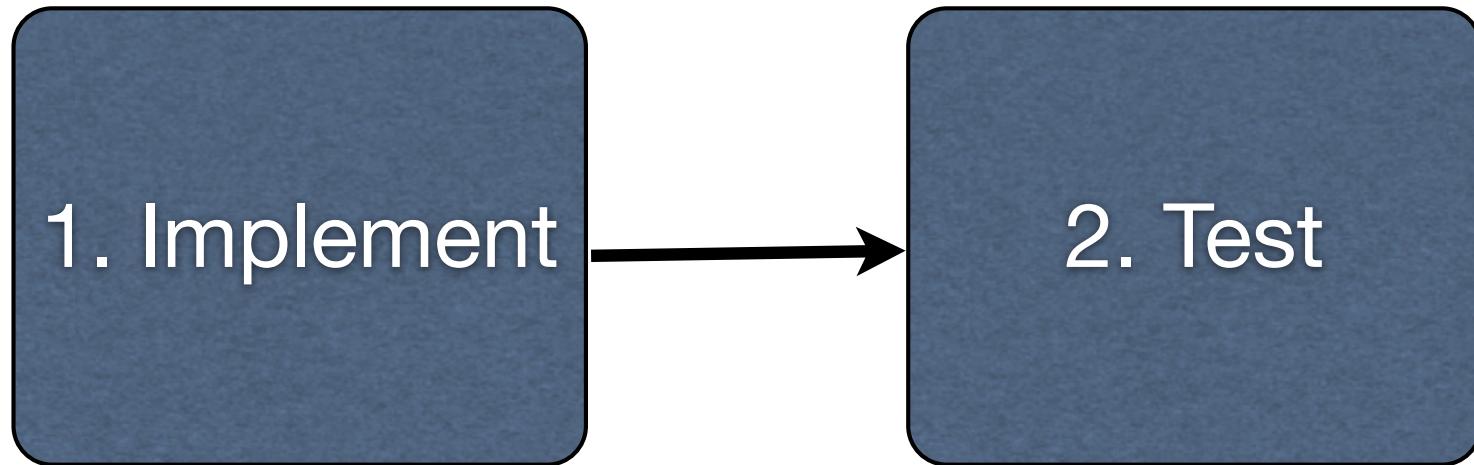
# **Test Driven Development (TDD)**

The idea of TDD was created ("rediscovered" in his words) by Kent Beck and formed a key part of the **Extreme Programming** agile process.

TDD has since become a mainstream way of developing software, not just as part of XP.

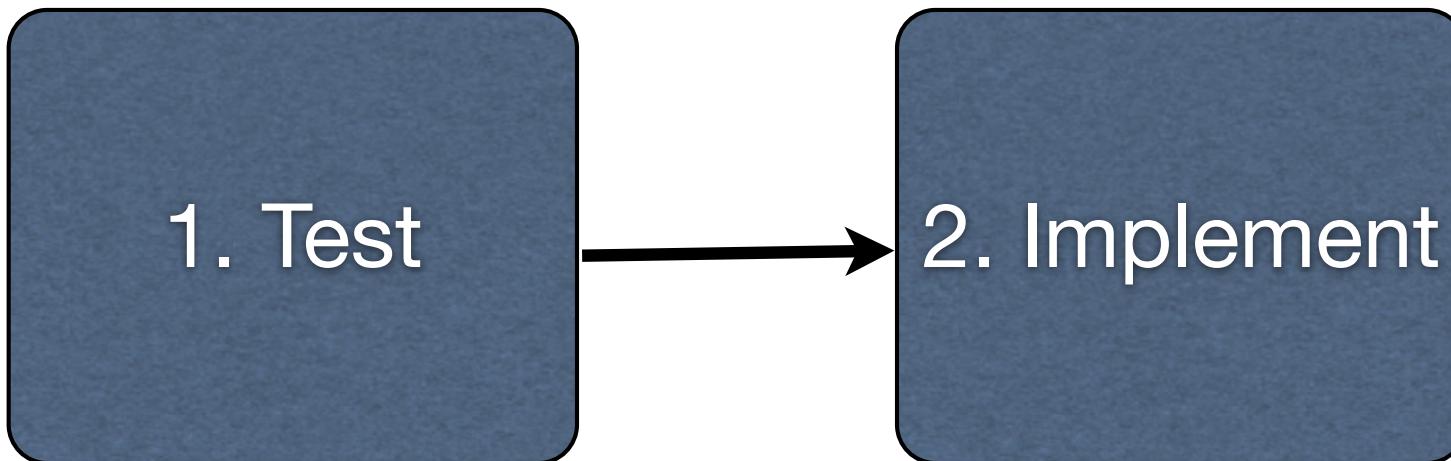
We'll explore at this idea in the coming slides...

# “Traditional” Order of Implementation and Testing



# Test-Driven Development (TDD)

Aka **Test-First** Development



# Why other engineers don't use TDD

# Why test first?

Firstly, note that unit tests are essential, so the TDD-or-not question is purely a matter of order.

A unit test states *what* the program should do. It's useful to think about this before deciding *how* it should do it.

Developers often write methods that they subsequently find hard to unit test. TDD avoids this.

TDD, applied well, guarantees excellent code coverage.

# A User Story is like a test

Any specification or user story implicitly provides 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.*

A user story may describe a whole range of input/output pairs. In this case it even provides a concrete example.

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

# Process of TDD

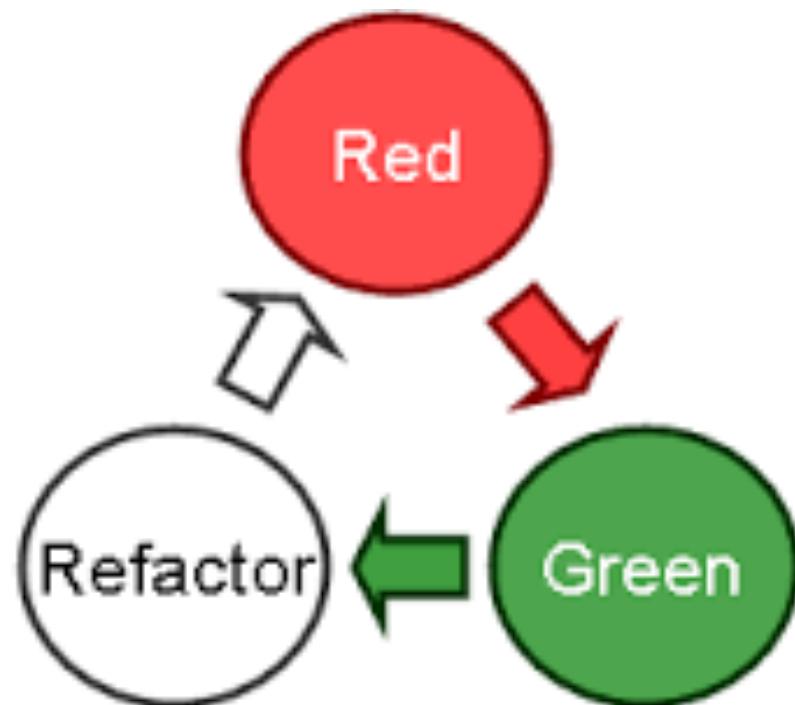


Write a failing test

Make the test pass



Refactor



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

# Unit test structure

This is the typical structure for a unit test:

- 1. Setup:** Put the system into the required state to execute the test.
- 2. Execution:** Execute the method that is being tested and collect the results.
- 3. Validation:** Ensure that the results of the test are as expected.
- 4. Teardown:** Restore the system to its pre-test state.

For simple tests, **setup** and **teardown** may not be necessary.

# JUnit

JUnit is a industrial-strength unit testing framework, originally written by Kent Beck and Erich Gamma.

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

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

(I'm using JUnit 5, but JUnit 4 is fine to use in practicals if that's what you have installed.)

**Erich Gamma:** Best known as first author of the seminal *Design Patterns* text. Co-creator of JUnit, team lead for Eclipse Java Development Tools (JDT).

# Anatomy of a JUnit 5 Unit Test

```
import org.junit.jupiter.api.Test;  
import static org.junit.jupiter.api.Assertions.*;
```

use the **JUnit** testing framework

```
public class FooBarTest {
```

Convention: FooBarTest has unit tests for the FooBar class.

@Test tells JUnit that this is a unit test

```
public void testOne() {  
    // execute unit under test...  
    assertTrue(...);  
}
```

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

@Test

```
public void testTwo() {  
    // execute unit under test...  
    assertEquals(...);  
}
```

Test methods always contain **assertions**.

# Summary of basic JUnit 5 assertions

**assertEquals(expected, actual)**: Asserts that the expected and actual values are equal (also **assertNotEquals**).

**assertTrue(condition)**: Asserts that the given condition is true (also **assertFalse**).

**assertNull(value)**: Asserts that the given value is null (also **assertNotNull**).

**assertSame(expected, actual)**: Asserts that expected and actual refer to the same object (also **assertNotSame**).

**assertThrows(exceptionType, executable)**: Asserts that executable throws an exception of the specified type.

# **@BeforeEach and @AfterEach annotations**

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

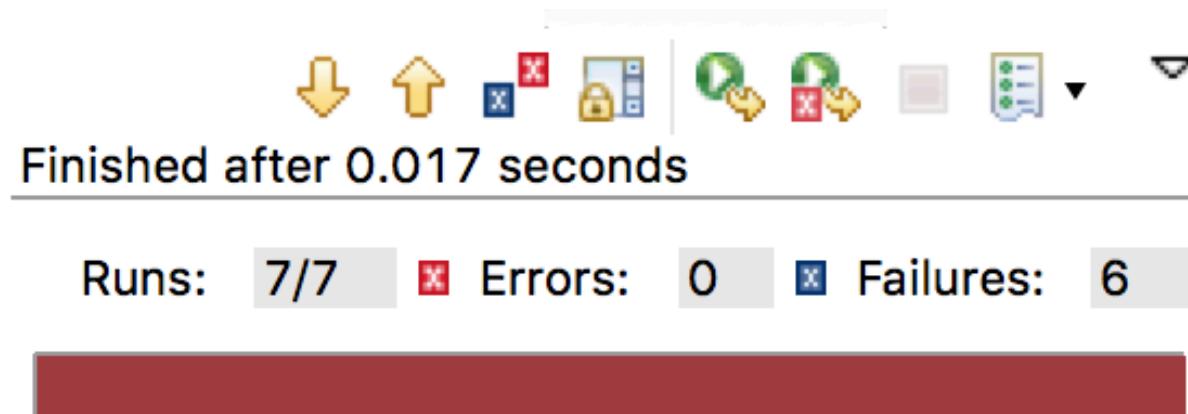
Methods with the **@BeforeEach** annotation are executed **before** each unit test:

```
@BeforeEach
void setUp(){
    myCart = new Cart();
}
```

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

```
@AfterEach
void tearDown(){
    myCart = null;
}
```

# Write a Unit Tests that fails -- RED



Finished after 0.017 seconds

Runs: 7/7 Errors: 0 Failures: 6

ShoppingCartTest [Runner: JUnit 4] (0.002 s)

- testAddThreeSausages (0.000 s)
- testAddTwoSausages (0.000 s)
- testAddSausage (0.000 s)
- testAddSixSausages (0.001 s)
- testShoppingCartCreation (0.000 s)
- testAddFiveSausages (0.000 s)
- testAddFourSausages (0.000 s)

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

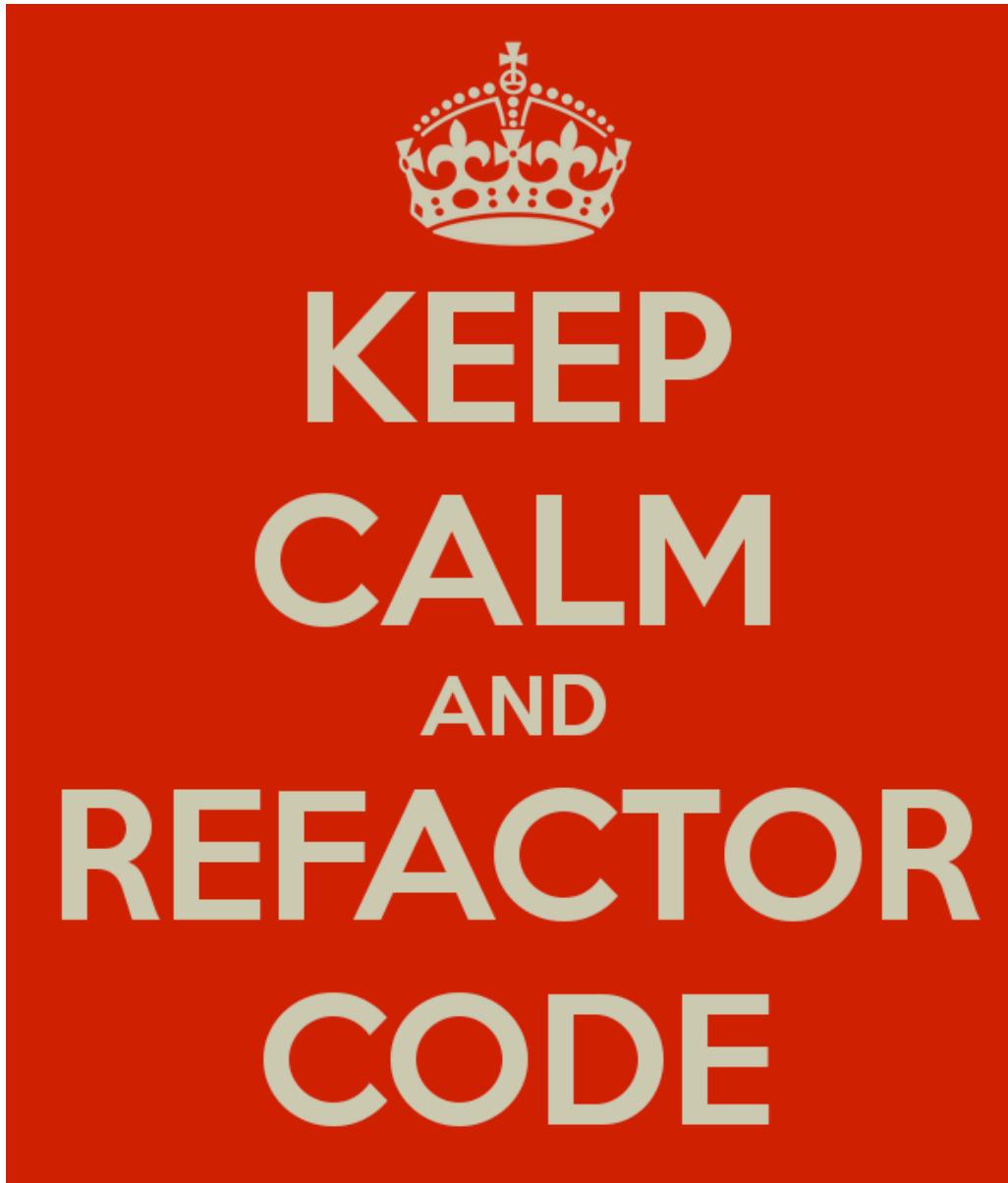


---

▼  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)

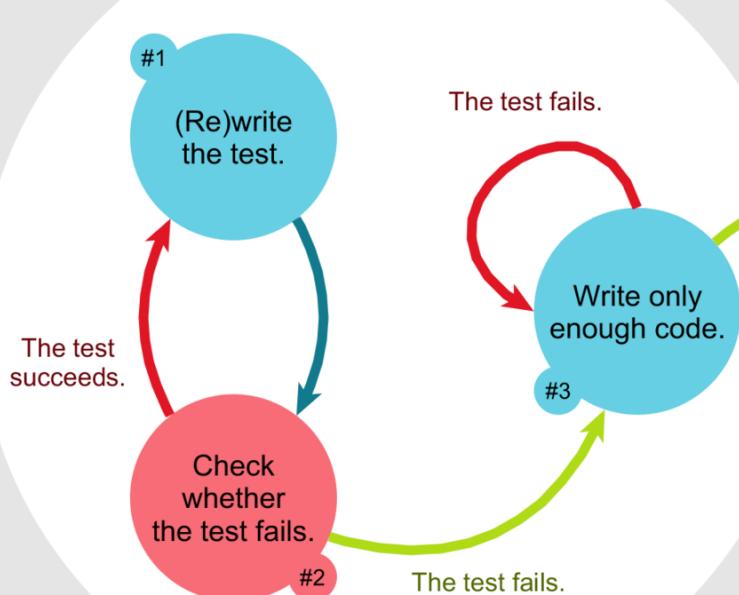
Now tidy up your code -- **REFACTOR**



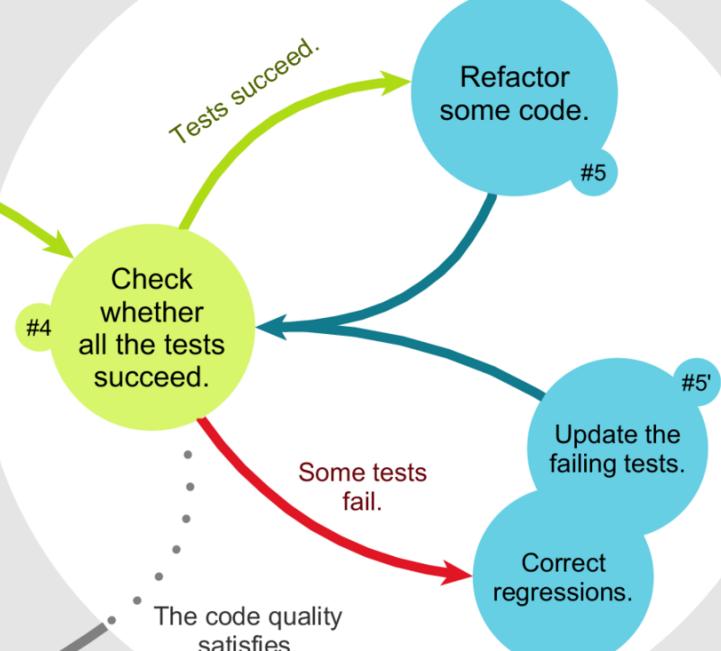
Tidy up your code, make sure there is no duplicated code or ugly bits. (We return to **refactoring** later in the module.)

# An overview of the TDD process

## CODE-DRIVEN TESTING



## REFACTORING



Iterate

focus  
Completion of the contract  
as defined by the test

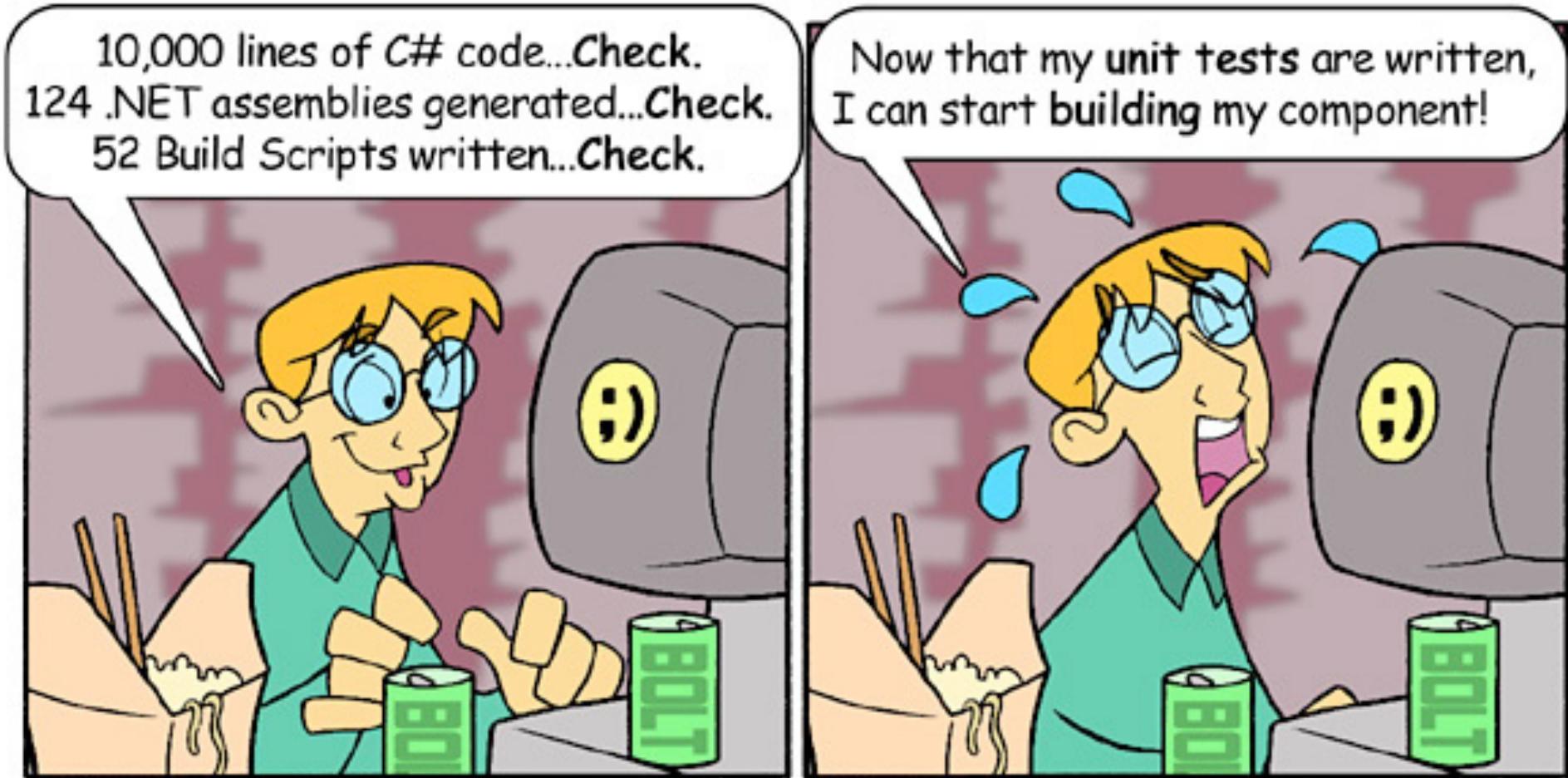
focus  
Alignment of the design  
with known needs

TEST-DRIVEN DEVELOPMENT



Xavier Pige

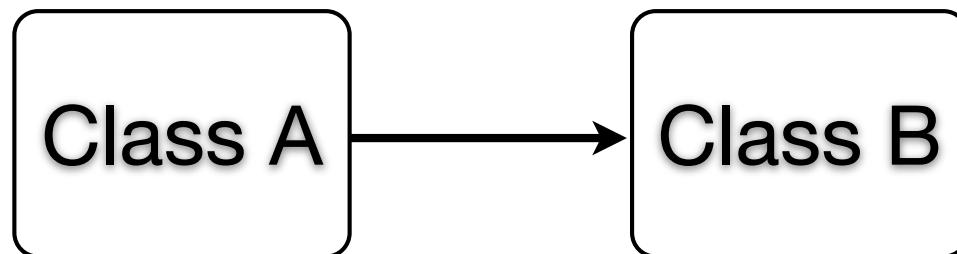
# Test cases can take up a lot of code



... but this is not a bad thing.

# Styles of TDD

Say we have this simple structure, where class A uses class B:



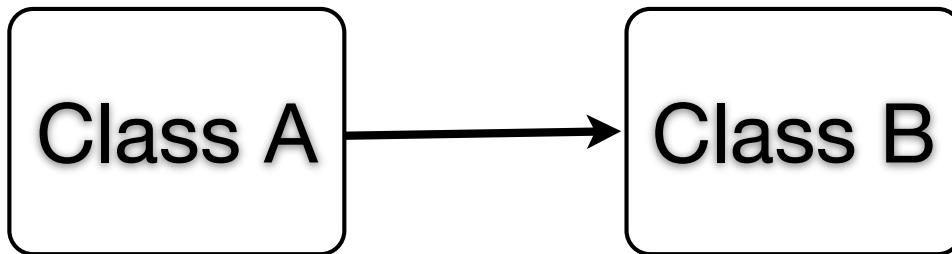
Which class do we test & develop first?

There are two main approaches:

**Classicist** (Detroit style, inside-out)

**Mockist** (London style, outside-in)

# Classicism Approach

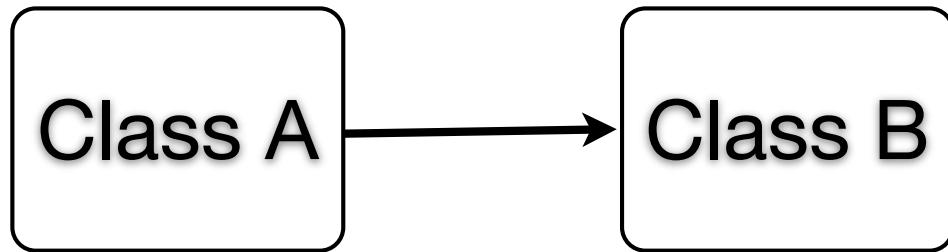


Apply TDD to Class B first.

Then apply TDD to Class A, relying on the correct implementation of B.

We typically use asserts to ensure that the A object is in the correct state at the end of the test.

# Mockist Approach



Apply TDD to Class A first.

How can we test Class A, when we don't have a Class B?

We **mock** B.

A **mock object** mimics the behaviour of a real object in a controlled way.

In the mockist approach, we ensure that objects **interact** together correctly.

# A Taste of Mocking

A mock object pretends to be a real one during a test.

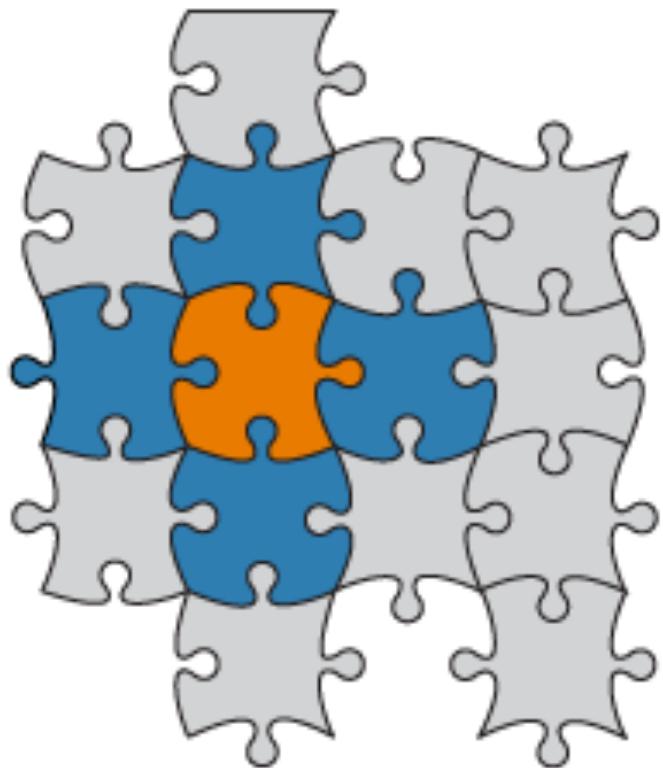
It provides the same interface as the object it's mocking.

It can also check e.g. that it is sent the right messages, in the right order, with the right arguments.

So mocks can test **interaction**, not just state.

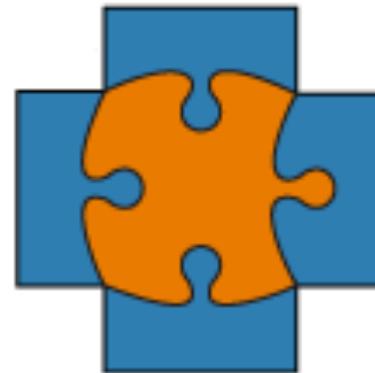
# Mocking isolates the component under test

System in Production



- Component Under Test
- Depended on Components
- Additional Components

System in Unit Test



- Component Under Test
- Mocks for Components

# An Example

```
class Counter {  
    private int count = 0;  
    public void increment() {count++;}  
    public int getCount() {return count;}  
}
```

A simple class for maintaining a count.

```
class MonitorTemp {  
    private Counter counter;  
  
    public MonitorTemp(Counter counter) {  
        this.counter = counter;  
    }  
  
    public void newTemp(int temp) {  
        if (temp < 0) {  
            counter.increment();  
        }  
    }  
  
    public int countFreezing() {  
        return counter.getCount();  
    }  
}
```

A class that uses the Counter class above.

# TDD using the Classicist approach

1. Use TDD to develop the Counter class.
2. Use TDD to develop the MonitorTemp class, using the existing implementation of Counter.

The MonitorTempTest class will contain unit tests that test **state** like this:

```
monitor.newTemp(12);
monitor.newTemp(-1);
monitor.newTemp(1);
assertEquals(1, monitor.countFreezing);
```

Note how this is also testing the Counter class behind the scenes.

# Pros and Cons of the Classicist approach

## Cons

We're testing the Counter class unnecessarily. It already has its own unit tests.

If the Counter class is updated to a buggy implementation, the unit tests for both Counter and MonitorTemp will fail.

(If 100 classes use Counter, the one bug will lead to 101 classes with failing tests.)

## Pros

We can refactor fearlessly.

E.g. we can move the counting functionality into the MonitorTemp class, delete the the Counter class and its tests, and all will be well.

# TDD using the Mocking approach

1. Use TDD to develop the MonitorTemp class, using a mock implementation of Counter.

The real implementation of Counter is not used and can be developed afterwards (the 'outside in' approach).

We'll see how the MonitorTempTest class could be implemented using Mockito on the coming slides.

Mocks can also be developed by hand. A framework like Mockito makes the task easier.

# Mocking the Counter class

The unit testing of MonitorTemp would look like this:

```
Counter counterMock = mock(Counter.class);
MonitorTemp monitor = new MonitorTemp(counterMock);
monitor.newTemp(12);
monitor.newTemp(-1);
monitor.newTemp(1);
verify(counterMock, times(3)).increment();
```

counterMock keeps track of its interactions with other objects.

The verify statement tests if increment has been called on counterMock three times.

If it hasn't, the test fails.

The pros and cons of mocking are the opposite of the classicist approach. Main con: it makes refactoring hard.

# Mocking by hand

Let's see how to do that example using hand mocking (i.e. without Mockito).

First, create an interface that reflects how the **MonitorTemp** class views the **Counter** class:

```
interface CounterInf {  
    public void increment();  
}
```

# Create the Mock

E.g. to enable checking how many times `increment` is called:

```
class CounterMock implements CounterInf {  
    private int numCallsToIncrement;  
  
    public CounterMock() {  
        numCallsToIncrement = 0;  
    }  
  
    public void increment(){  
        numCallsToIncrement++;  
    }  
  
    public int getIncrementNumCalls() {  
        return numCallsToIncrement;  
    }  
}
```

# Using the Mock

Sample unit test:

```
@Test  
void testForThreeReadings() {  
    CounterMock counter = new CounterMock();  
    MonitorTemp monitor = new MonitorTemp(counterMock);  
    monitor.newTemp(12);  
    monitor.newTemp(-1);  
    monitor.newTemp(1);  
    assertEquals(3, counter.getIncrementNumOfCalls());  
}
```

You can add stub methods to CounterMock to return certain values, check that methods are called a certain number of times, check that method arguments have the correct values -- anything you want.

# When to Mock?

Any class that the class under test depends on *could* be mocked.

Any class that performs an external action like database access or sending an email *should* be mocked.

If there's interesting interaction between the class under test and a class it depends on, consider mocking the dependent class.

After that, how much mocking you should do is a matter of judgment.

Remember that the more you mock the interaction between objects, the harder the code will become to refactor.

# Advantages of TDD

Code is written with testability in mind.

Excellent code coverage tends to happens naturally\*.

Thinking about testing first helps the developer to focus on *what* the method does before thinking about *how* to do it.

Reduces the risk of time/customer pressure leading to testing being delayed or even omitted.

\* In SE-Radio Episode 324, the creator of JaCoCo, Marc Hoffmann, strongly promotes TDD as the best way to achieve meaningful test coverage.

# TDD Summary

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.

Two main styles of TDD: Classicist and Mockist.