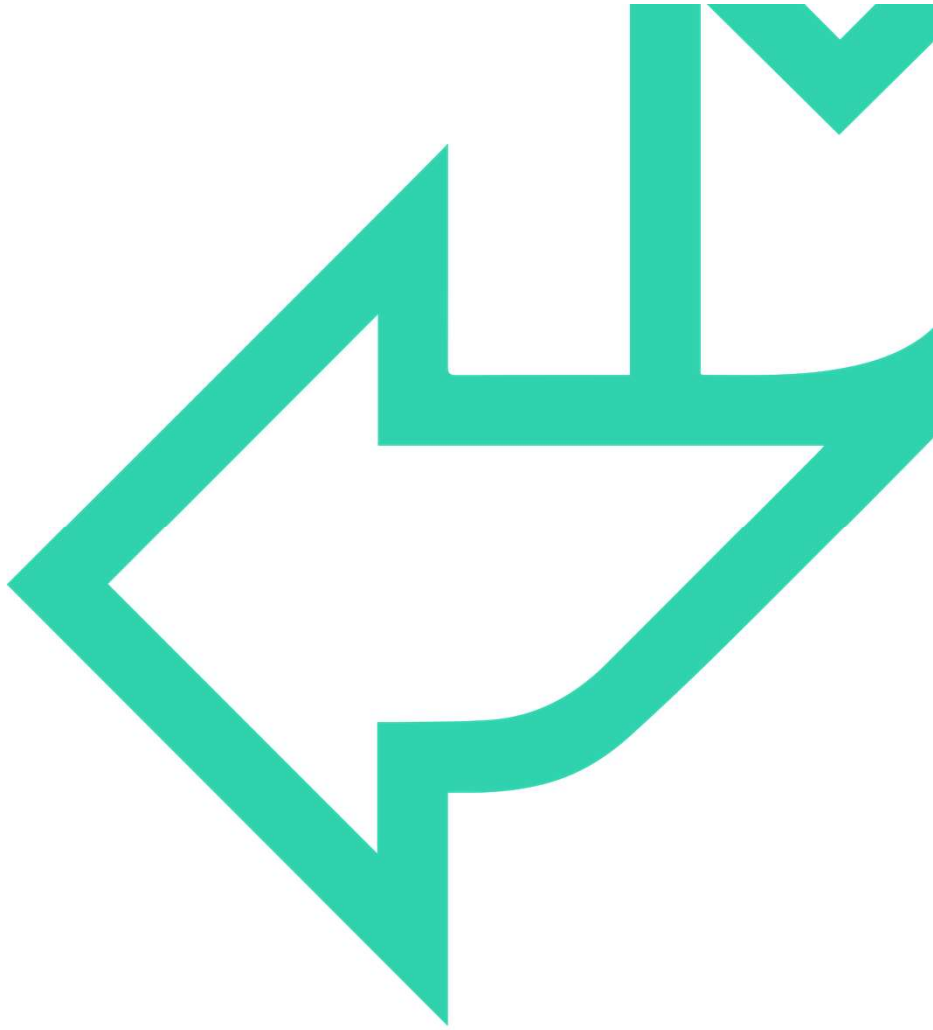
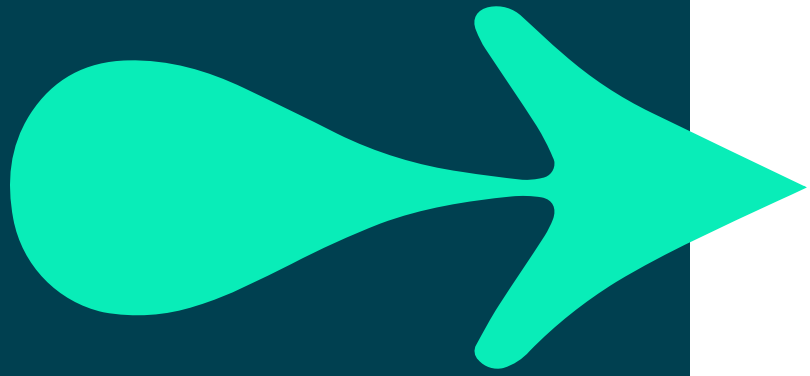
POWERING POTENTIAL

Test-driven Development



Java

OVERVIEW



Objectives

- To explain the aims and objectives of the course

Contents

- Course administration
- Course objectives and assumptions
- Introductions
- Any questions?

Exercises

- Locate the exercises
- Locate the help files

Administration

Front door security
Name card
Chairs

Fire exits
Toilets
Coffee Room

Timing
Breaks
Lunch

Downloads and viruses
Admin. support
Messages

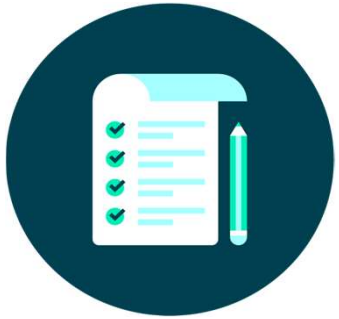
Taxis
Trains / coaches
Hotels

First Aid

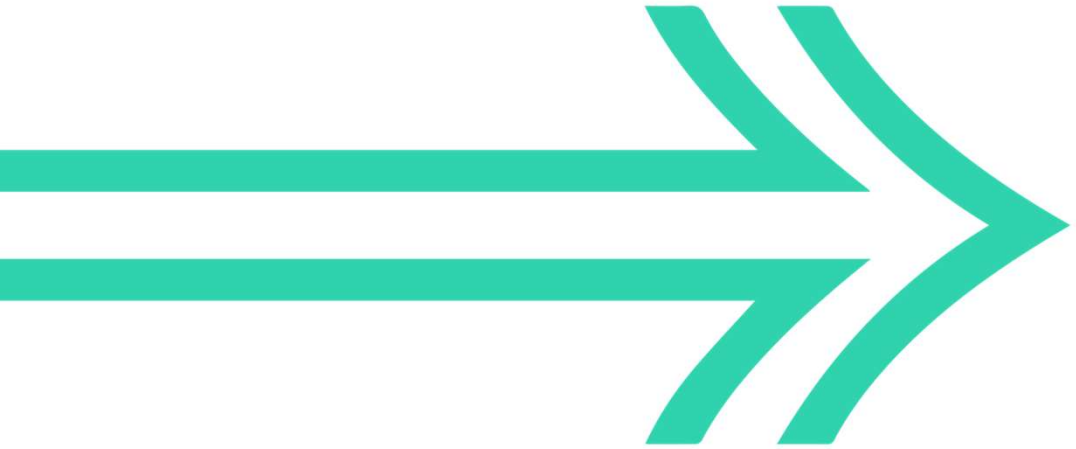
Telephones / mobiles

Course delivery

Hear and forget
See and remember
Do and understand



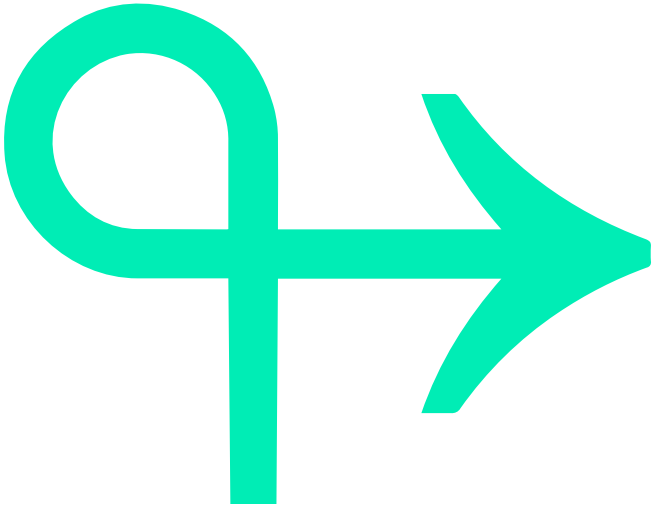
The training experience



A course should be:

- A two-way process
- A group process
- An individual experience

Workshop outcomes



By the end of the course, you will be able to:

- Appreciate the problem TDD is trying to solve
- Appreciate the difference between a test after and test before approach in software development
- Improve the test coverage in your code
- Develop production code following a TDD approach
- Write a Unit Test in Java
- Specify a good and bad unit test
- Understand the relationship between a unit test, the class under test, and the dependants to the class under test
- Work with Stubs
- Work with Mocks
- Use Unit Tests and Test Doubles in a TDD cycle

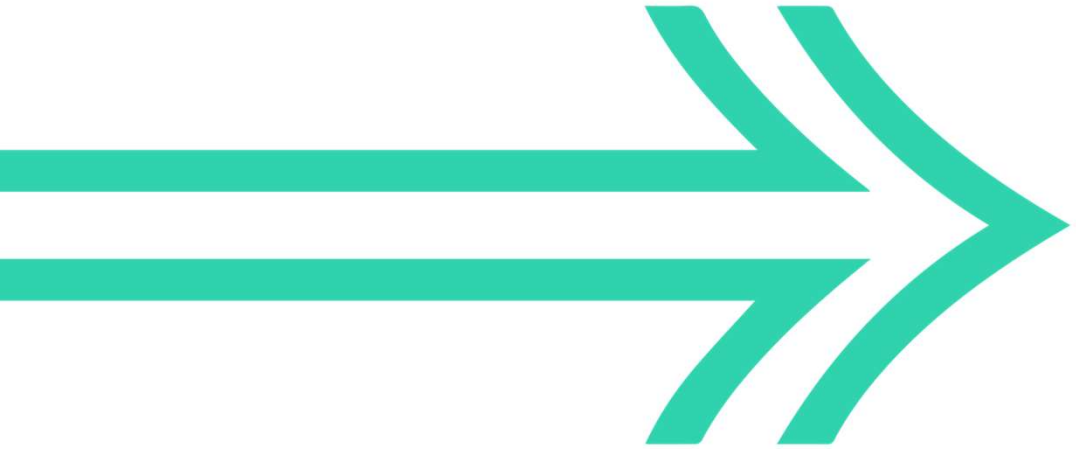
Assumptions

This course assumes the following prerequisites:

- You have a basic knowledge of Java development
- You know how to use maven

If there are any issues, please tell your instructor now

Introductions



Please say a few words about yourself:

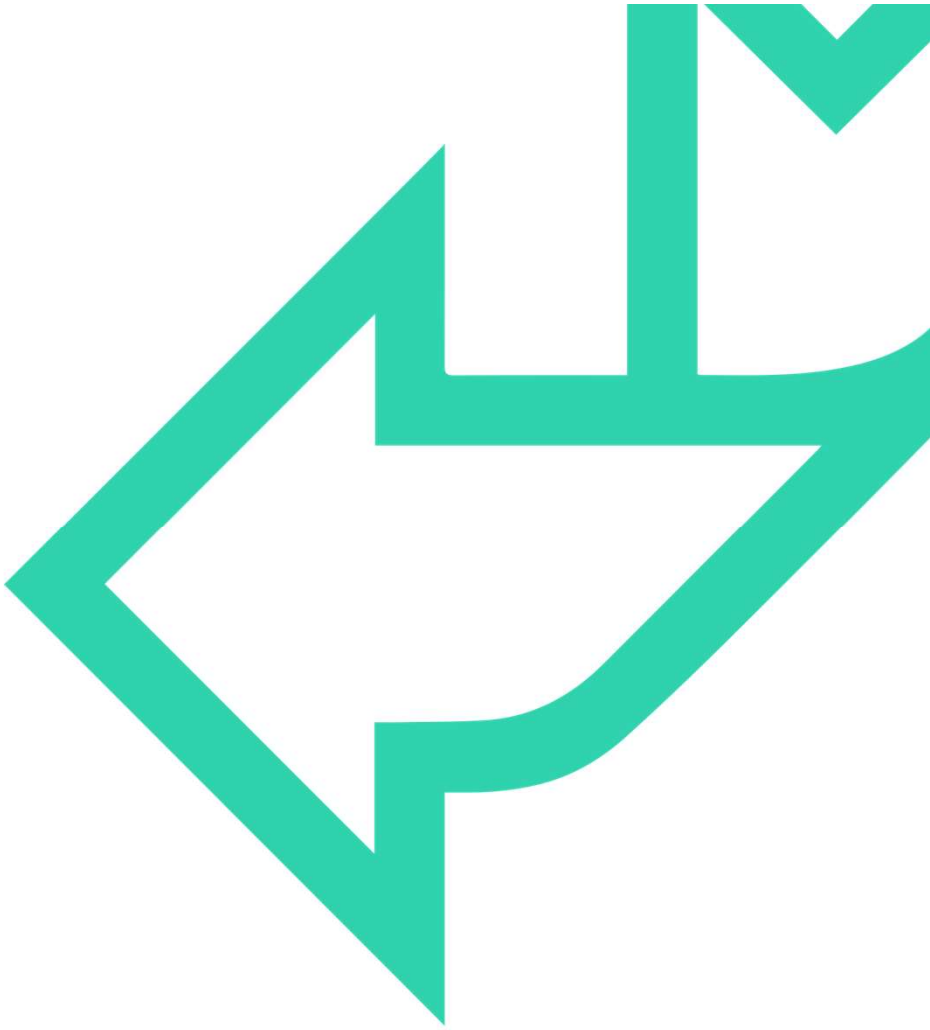
What is your name and job?

What is your current experience of:

- Computing?
- Programming?
- Testing?

What is your main objective for attending the course?

Any questions?

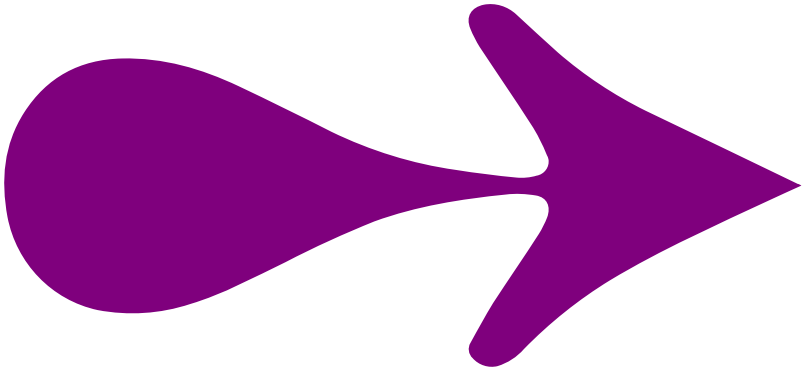


Golden Rule
→ ‘There is no such thing as a stupid question’
First amendment to the Golden Rule
→ ‘… even when asked by an instructor’
Corollary to the Golden Rule
→ ‘A question never resides in a single mind’

TDD

- Test
- Driven
- Development

What's the problem?



Most developers who use a testing framework follow this pattern

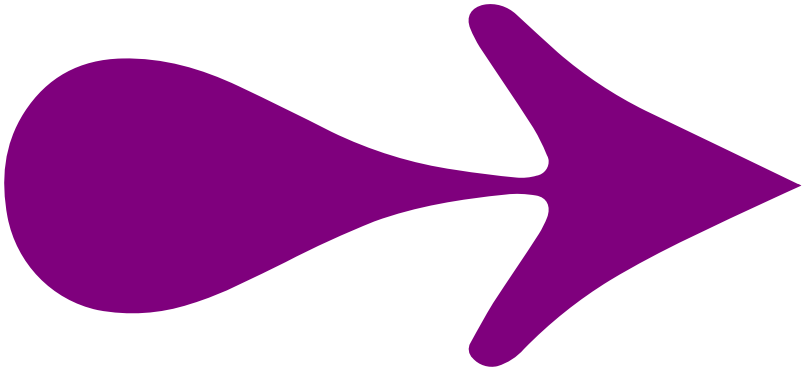
- Write production code (maybe all of the code)
- Think how they are going to test it
- Write tests with no real understanding of how they can achieve full test coverage
  - Tests might simply be print outs to standard out, so no real regression possible
- Test what they think works
- Test areas of the code that they are not sure if it functions as expected

Tests are not seen as a way of documenting their software

Tests are not seen as a tool for instilling confidence in their code

TDD is

- **An approach**
- **A philosophy**



Tests can:

- Act as documentation
- Create a framework for developing new code with confidence
- Create a framework for modifying existing code with confidence
- Create a framework to help improve the design of your code

A **test first driven** approach is all of the above and:

- A tool for increasing test coverage
- A more robust approach to developing new code and modifying existing code

Tests → Development

The software industry didn't invent the idea of tests first, development after (TDD).

Like most ideas in software (patterns, interfaces, etc.) they come from real world systems.

There is a lot that can be learned from how other industries inject quality into their systems, and tests prior to development is one of those good practices.

It may sound like a crazy idea but look at the images on the right. A measure of quality is established before the production of any of those items.

Whether it be food, semiconductor parts, or Parts for cars, etc., quality has to be built into the processes

Tests are about ensuring quality in your code

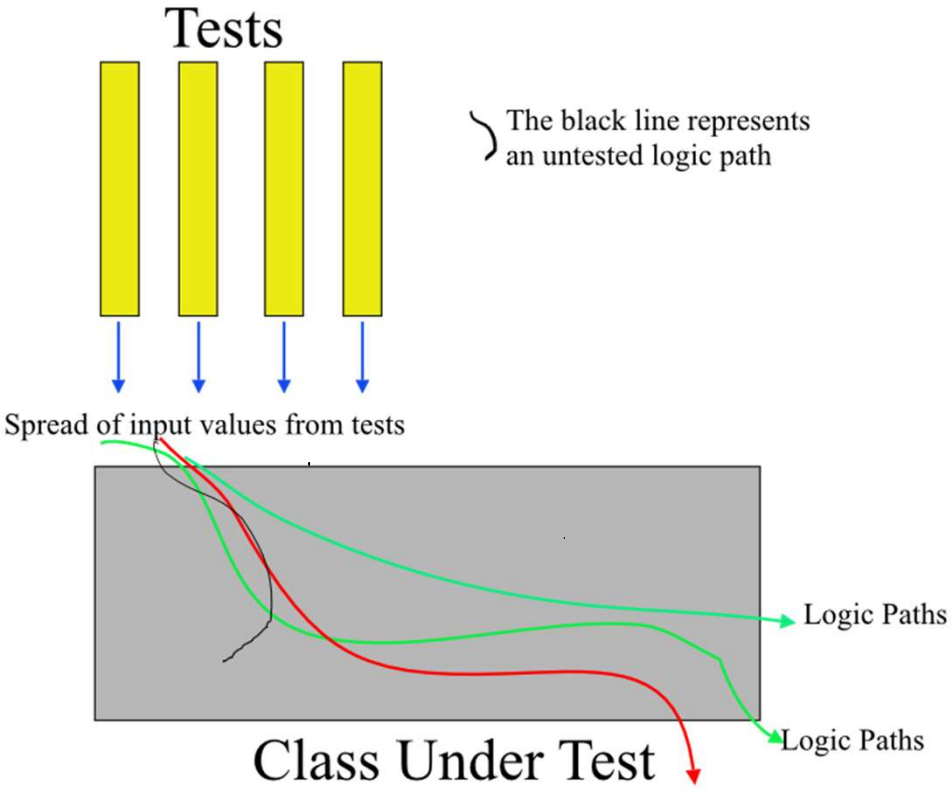
Test coverage

Test after approach

It's common knowledge that's it's very difficult to design all the tests for code that is already written

There's no guarantee that:

- All logic paths are covered
- You achieve your desired test coverage
- Testing the right thing, it should be about the logic paths



Test coverage

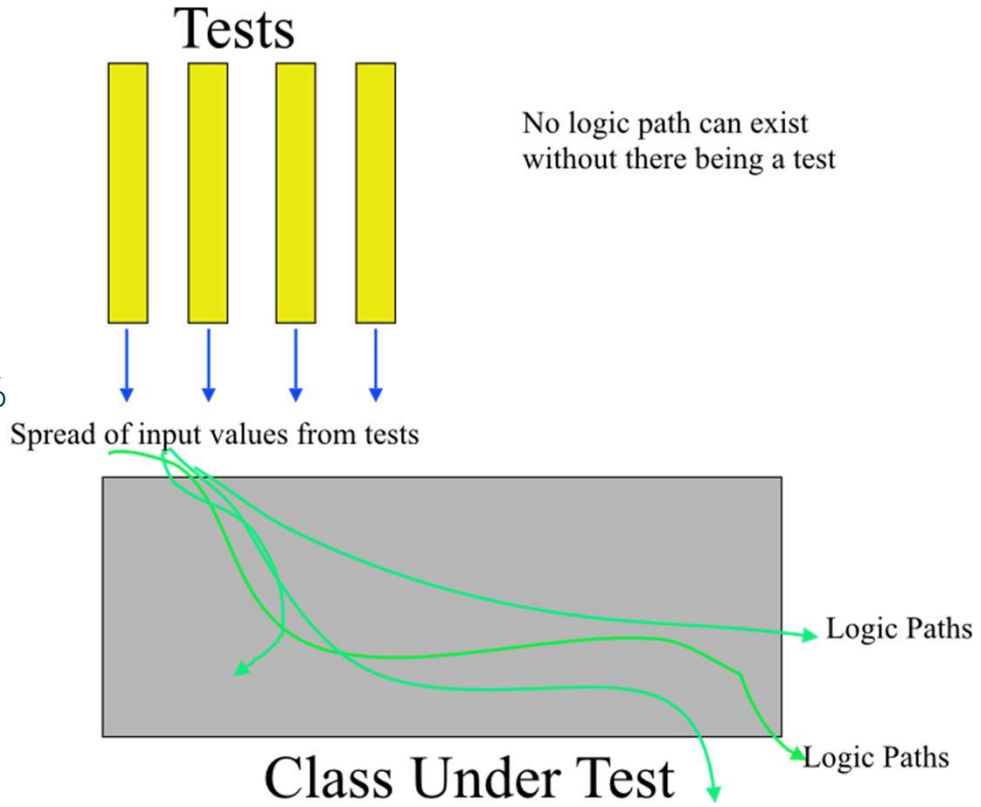
Test before approach

If no production code can exist without there being a test for it, then you should be able to achieve 90-100% coverage.

If you only write enough production code to pass a test, then all logic are covered.

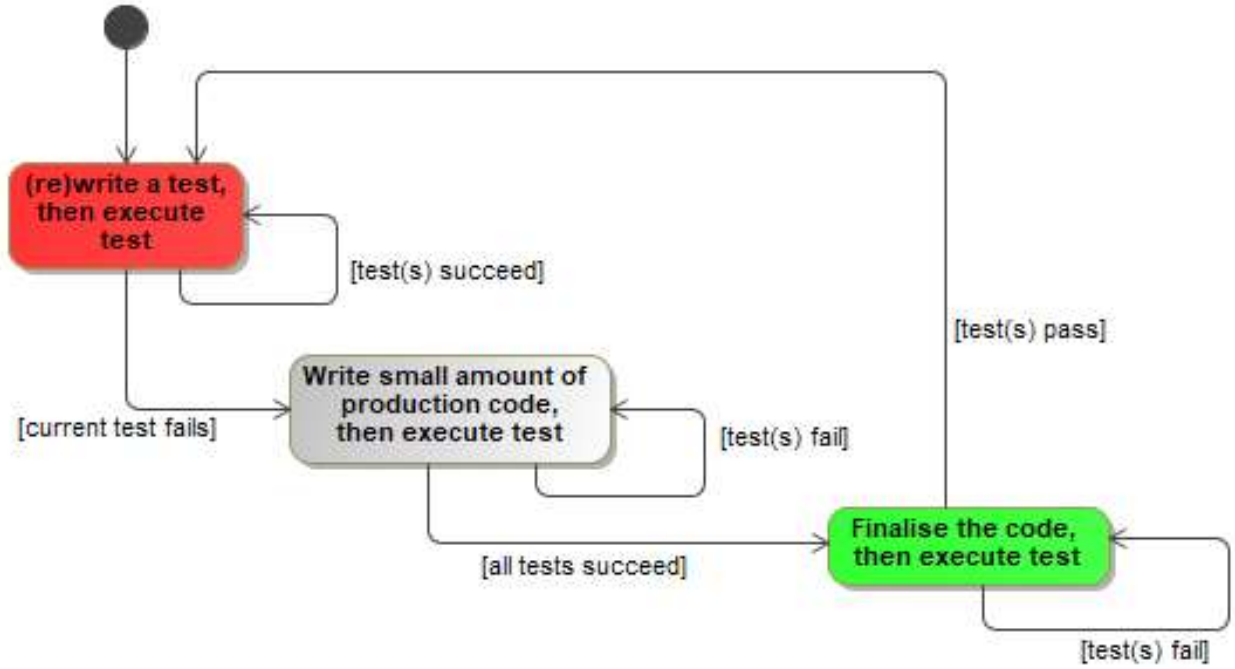
Many organisations have policies of not testing setters and getters.

Leads to an improved quality of tests because you test what is actually there.



TDD lifecycle

- Write a test
- Write just enough code to pass the test
- When test passes, push code to repo
- Refactor code if needed
- Ensure tests are still passing
- Push code to repo



The source of test data

Consider the following scenario:

You are part of a consultancy working with the government (UK), a rail infrastructure management team (NR), and the construction companies that maintain and build the rail infrastructure.

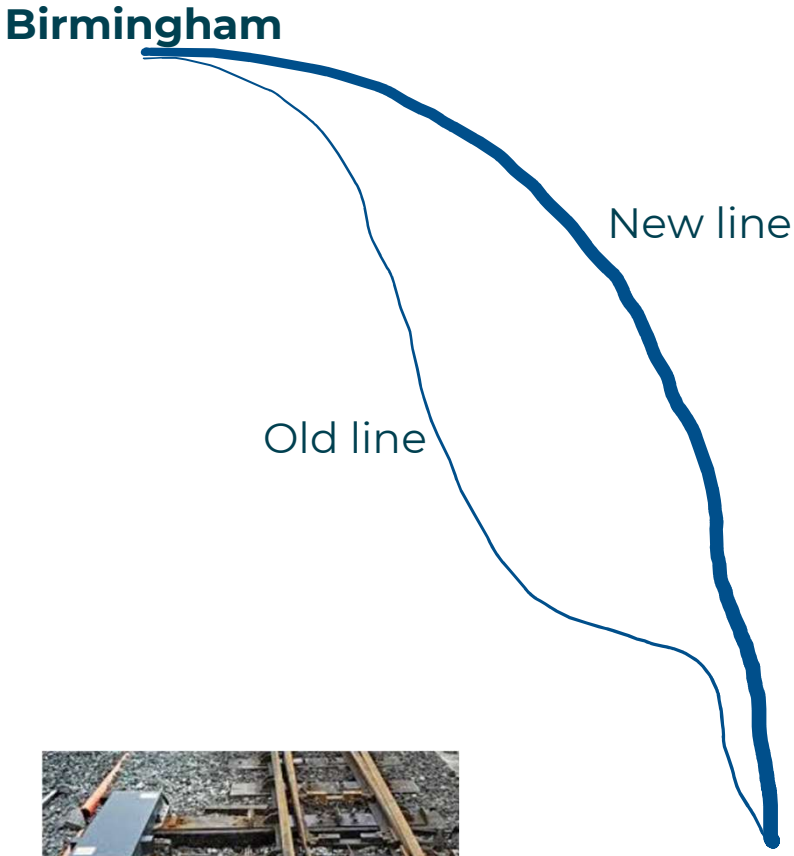
Your company helps to coordinate the three parties above.

NR and the construction companies purchase components from other companies further down the chain.

Upgrade scenario

New track along a new route must be laid to accommodate new highspeed rolling stock (engines and carriages).

Old tracks would buckle under new rolling stock.

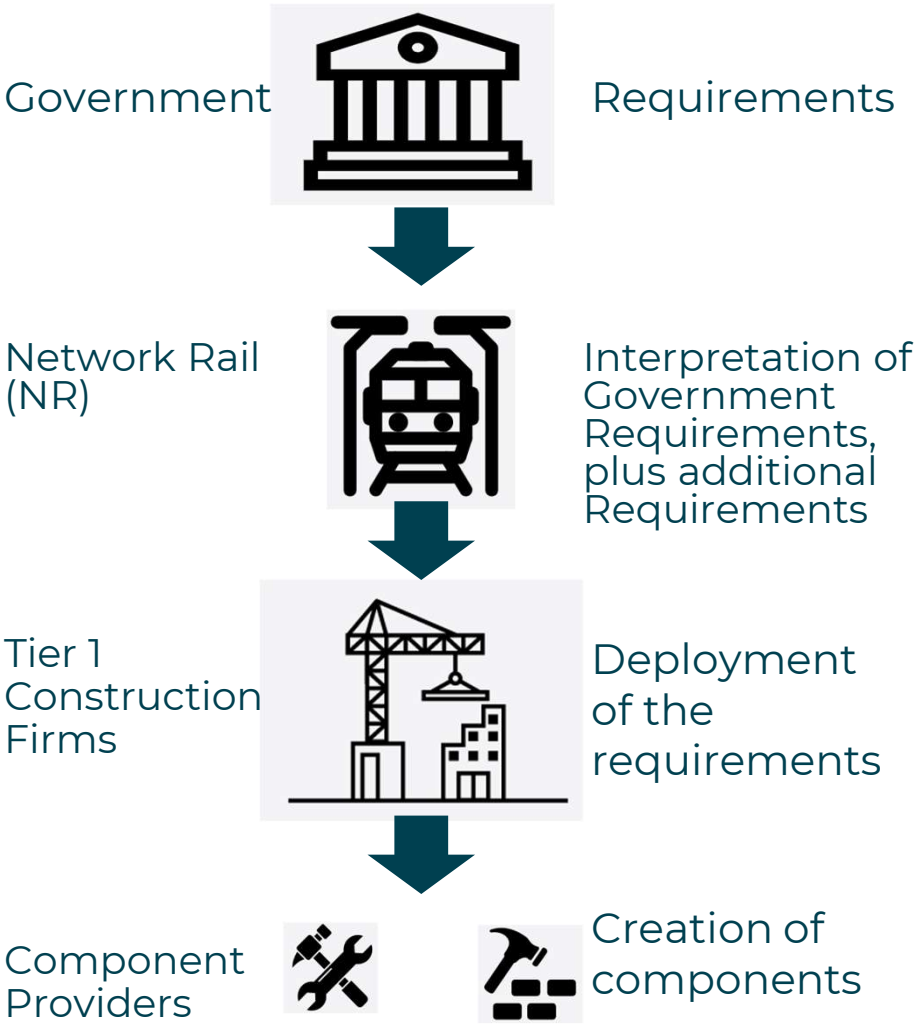


Points

Hierarchy of commands

The government mandates certain requirements onto the rail industry. NR interpret those requirements and ensure that they are delivered by the tier 1 construction firms. NR may also add other requirements to the mix based on their experience and knowledge of the industry.

NR and tier 1 construction companies work with component providers to decide on the components required to meet the requirements.

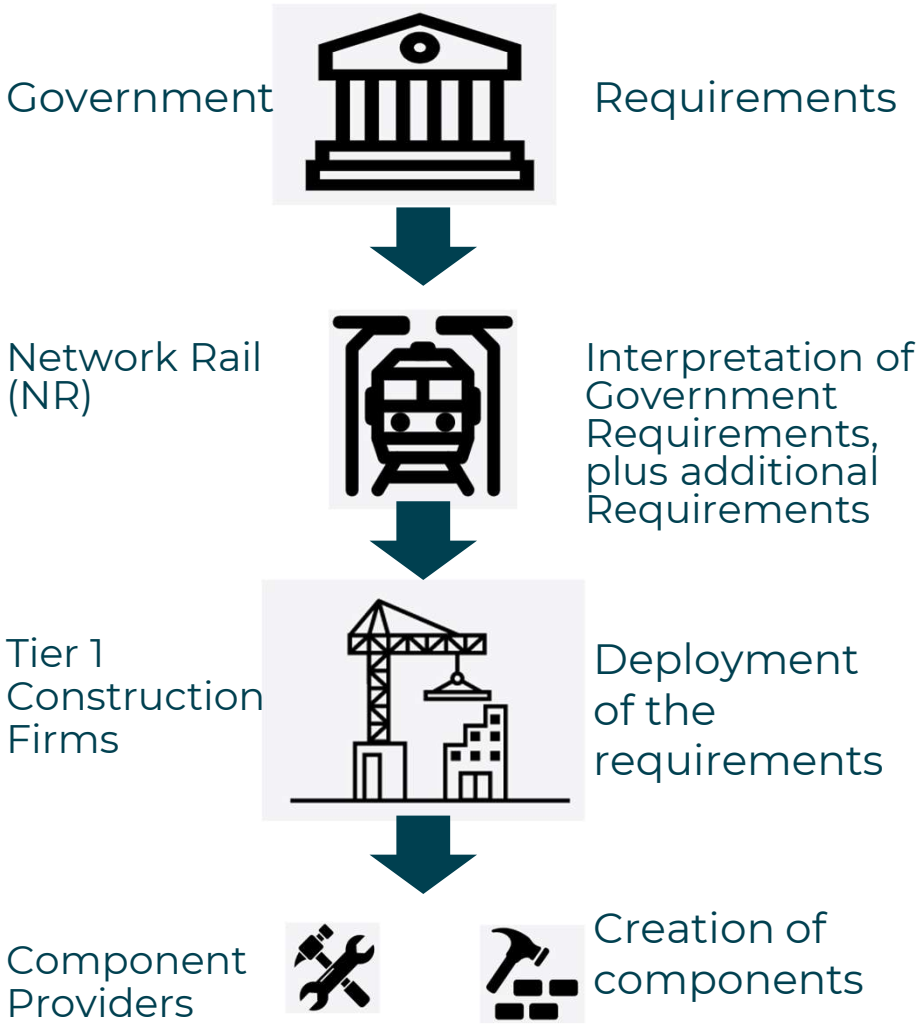


So where is the unit test?

If there is a trackside problem whose responsible for fixing it?

- Government?
- NR?
- Tier 1 companies?
- Component providers?

In software, TDD traditionally sits at the software component level



Structure of tests

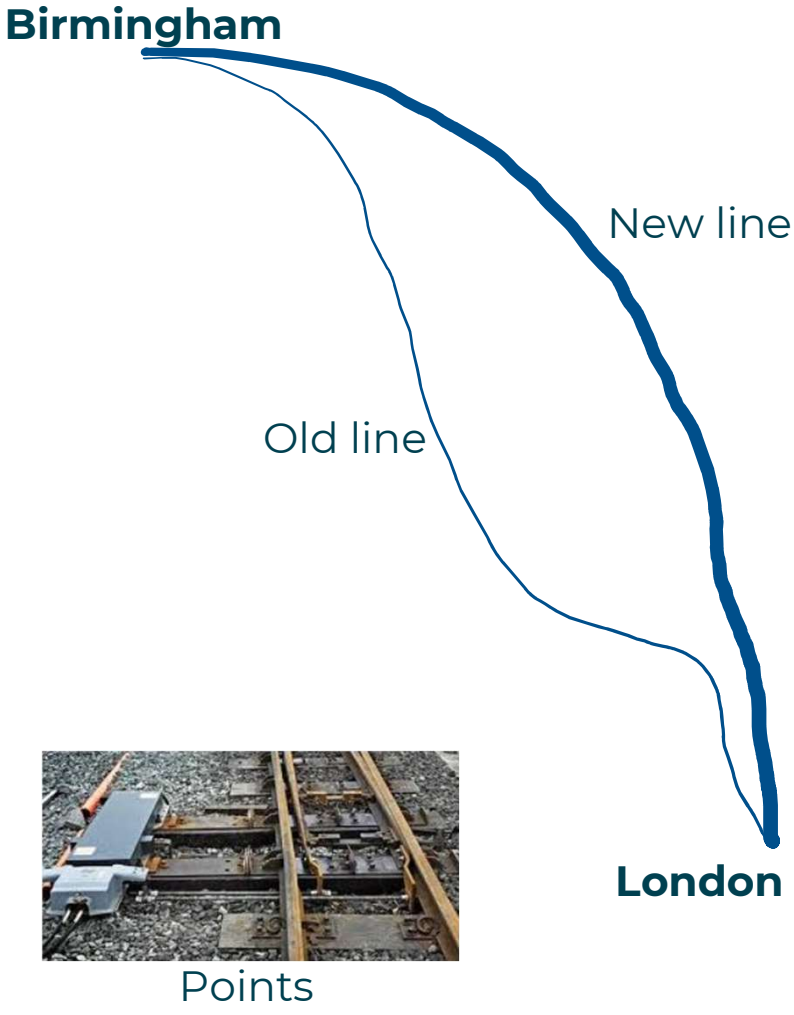
Tests can come in many forms

- Acceptance criteria statements
- Spec by example
- Truth tables

All of the above can be defined a state testing

As a software engineer you have to find a way to get those test values into the tests you writing for your classes / components

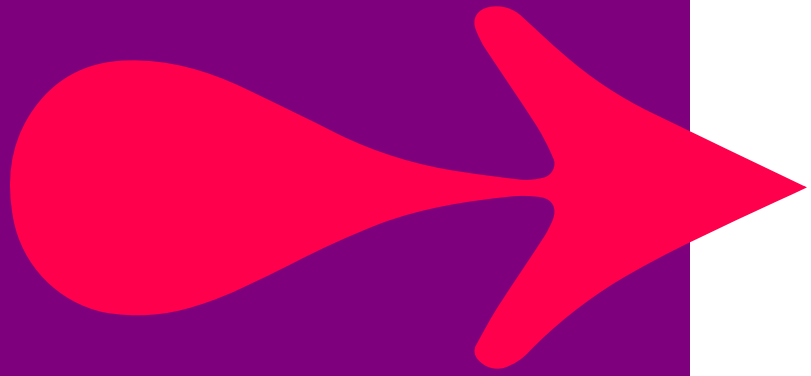
- There is no magic bullet



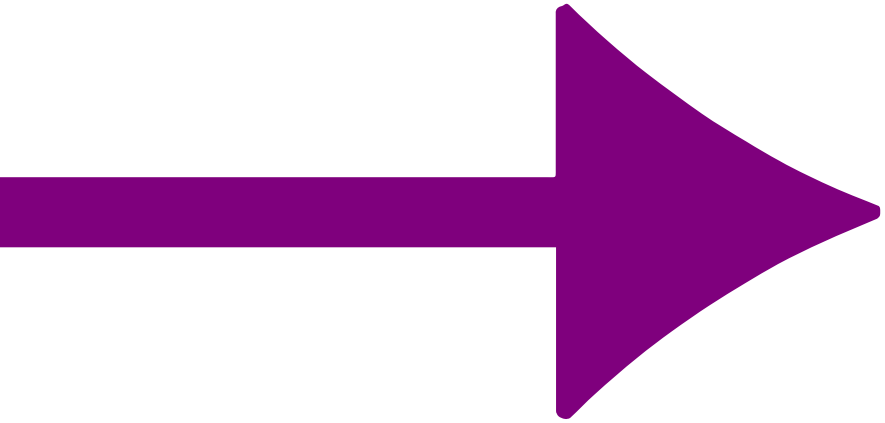
Points

QUICKLABS

TDD LIFECYCLE



Quicklab (QL-1) TDD Walkthrough

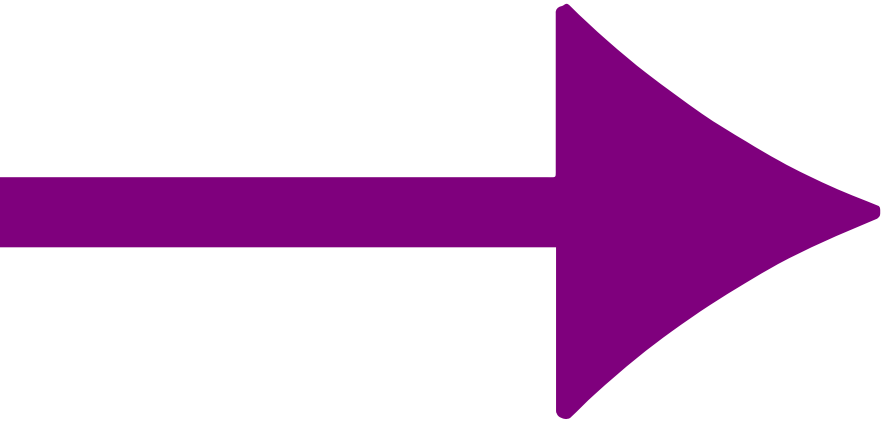


String Tokeniser

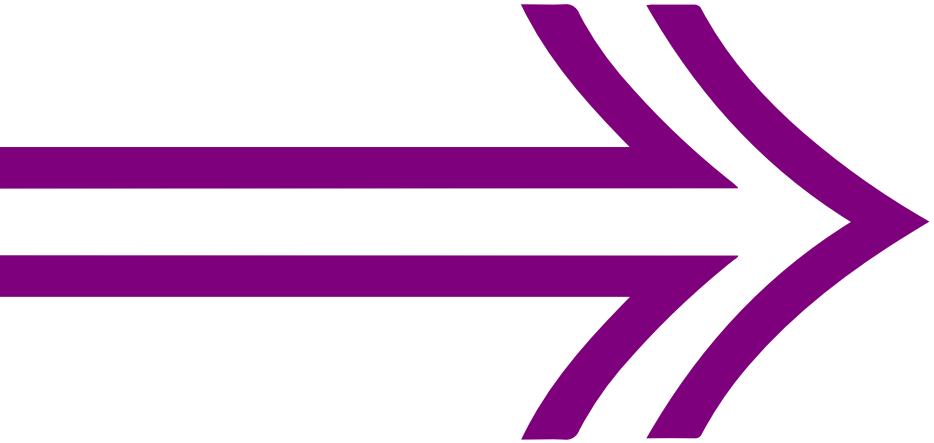
You are going to develop a small piece of code that accepts a comma-delimited list of tags (words/tokens - characters, numbers, some symbols like $£%& but not a comma) and must return an array of tags. The preceding and proceeding commas should be removed, white spaces preceding the first tag must be removed, white spaces succeeding the last tag must be removed, and contiguous series of words separated with spaces and ended with a comma should be treated as a single tag

Quicklab (QL-2) TDD

1. UK Car Registration Plates



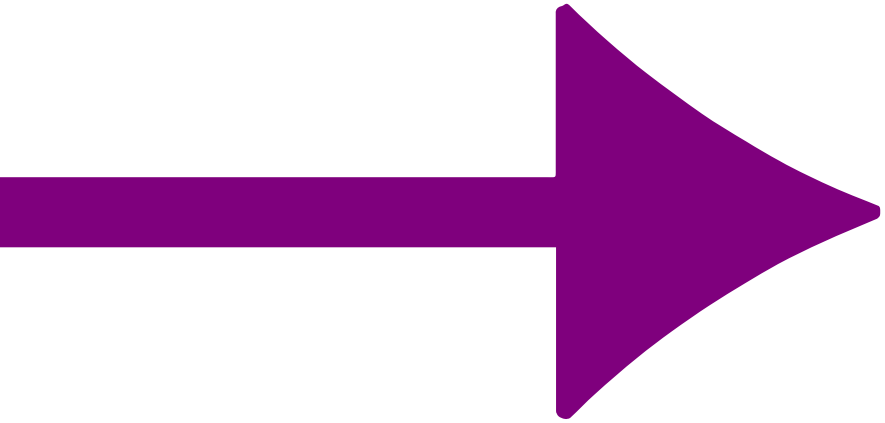
TDD takeaways



- TDD comprises of a series of unit tests
- You need to work with a source control platform
- We follow a RED, GREEN, REFACTOR strategy
  1. Write a test (it's failing – RED)
  2. Write only enough production code to pass the test (it passes – GREEN)
  3. Don't write any more code than is required to pass the test
  4. When a test passing, commit your code to a source control repo
  5. If you need to refactor the code, do so, but ensure tests are still passing, then push your code to a source control repo
  6. Move onto your next test
- Add tests incrementally. If you don't you won't know which piece of production code is causing a test to fail

Quicklab (QLC-1) TDD case study

Highest number finder

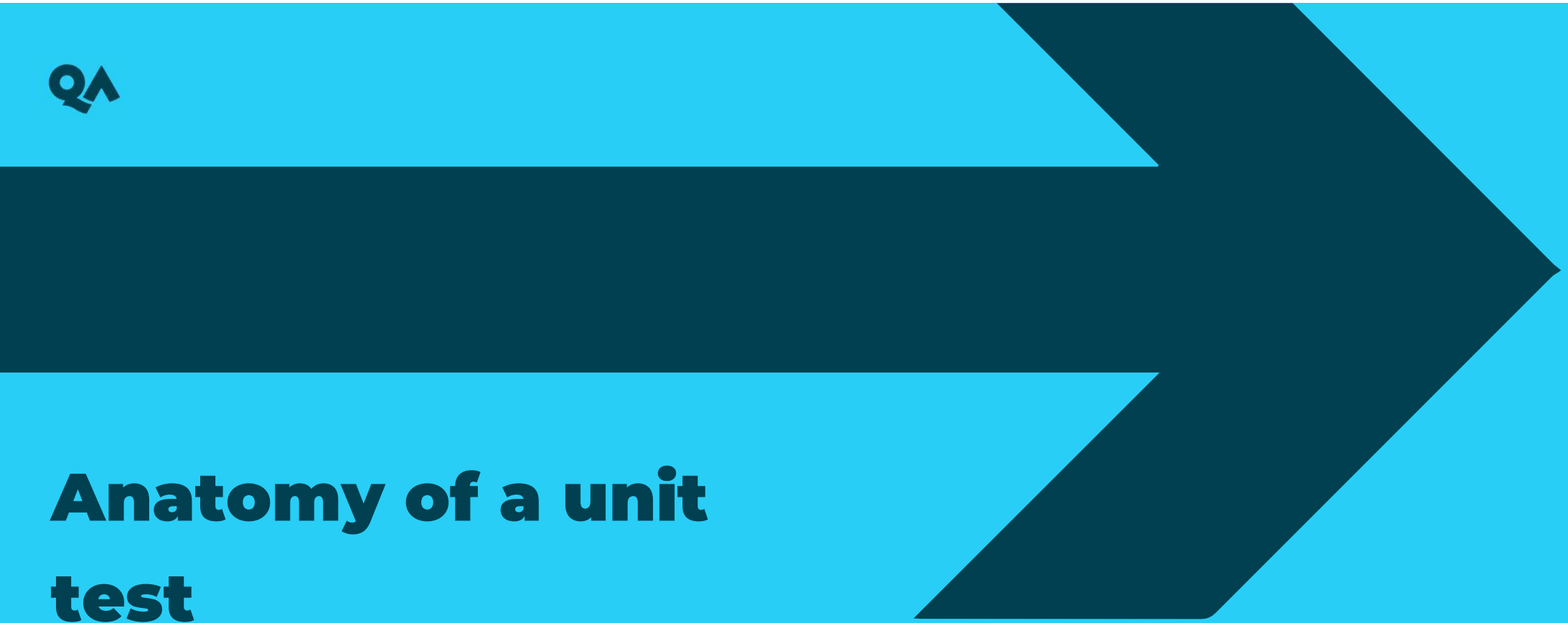


Find the highest number in an array of integers

Given the following specification:

- If the input were {4, 5, -8, 3, 11, -21, 6} the result should be 11
- An empty array should throw an exception
- A single item array should return the single item
- If several numbers are equal and highest, only one should be returned
- If the input were {7, 13} then the result should be 13
- If the input were {13, 4} then the result should be 13

The most challenging part is determining which test to write first. Always start simple and with a test that will not need to handle exceptions.



Anatomy of a unit test

Unit tests

- Structure of a unit test
- Dos and don’ts of good unit tests

Recall our first test method

```
@Test
public void find_highest_in_array_of_one_expect_single_item()
{
    // Arrange
    int array[] =
    {
        10
    };
    HighestNumberFinder cut = new HighestNumberFinder();
    int expectedResult = 10;

    // Act
    int result = cut.findHighestNumber( array );

    // Assert
    assertEquals( expectedResult, result );
}
```

Qualities of a good unit test

Isolated – does not depend on any other unit test.

Comprises of the three A's:

- Arrange
- Act
- Assert

The object being tested is usually named as the CUT

Performs no (all of these must accomplished through test doubles):

- IO
- Network calls
- DB Access

Quick – the tests execute in milliseconds.

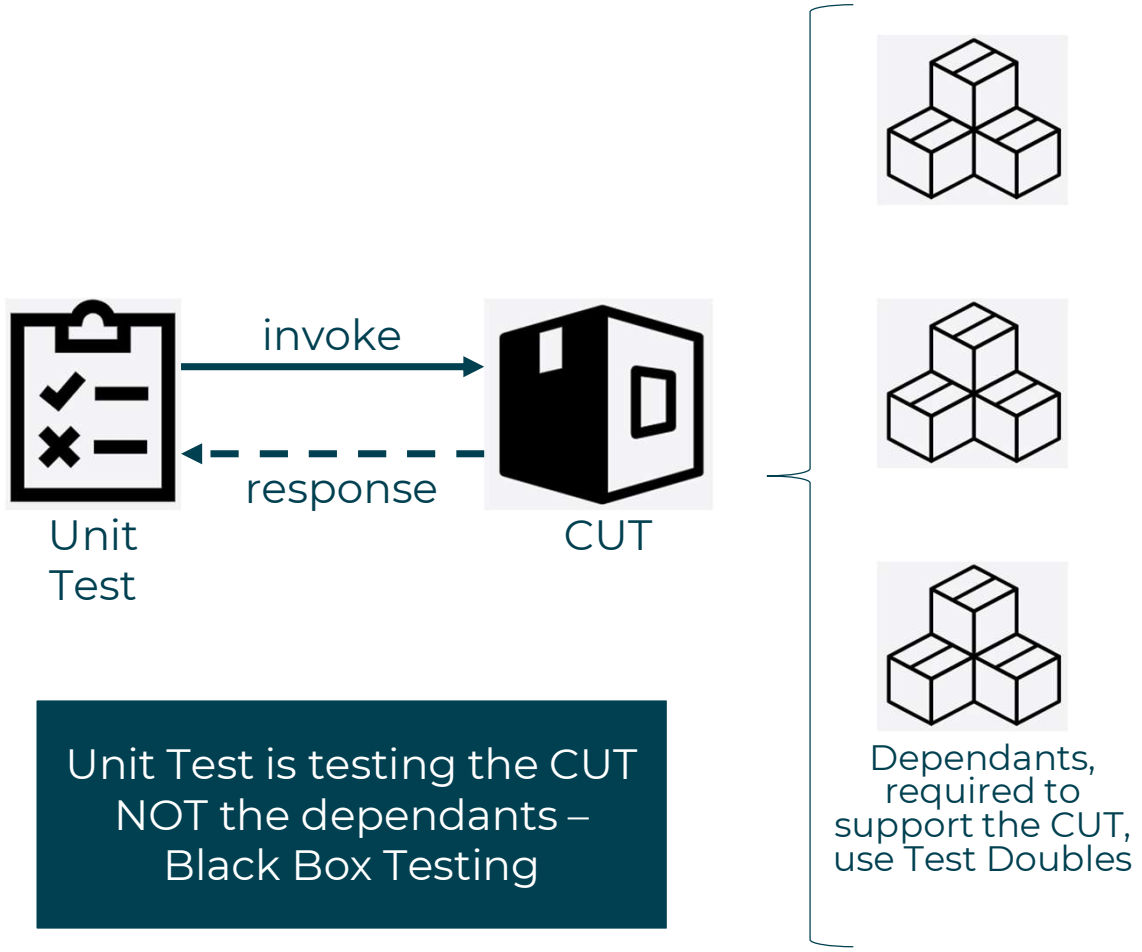
Implementation Class

```
public class HighestNumberFinder
{
  public int findHighestNumber(int[] values)
  {
        return values[0];
  }
}
```

- Only write enough implementation code to pass the test
- Do not attempt to write other pieces of code for tests that you have not implemented yet, this maintains your high-test coverage
- Writing more code than is required to pass the test means you now have untested code, this will reduce your test coverage

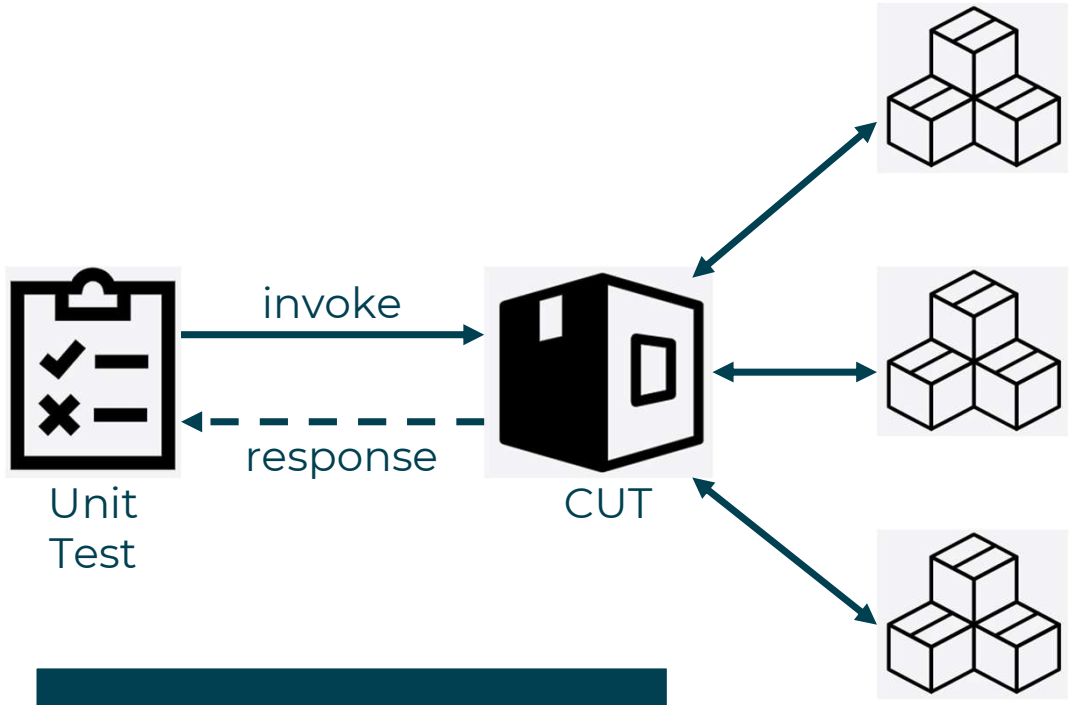
Only write what is needed to maintain good test coverage

Relationship between Unit Test, CUT, and its dependents



Unit Test is testing the CUT NOT the dependants – Black Box Testing

Relationship between Unit Test, CUT, and its dependents



Unit Test is testing the CUT NOT the dependants – White Box Testing

Consider the following extension to Find Highest Number

A requirement has come through that the highest number finder must deal with a finding the highest number for a series subjects being taught.

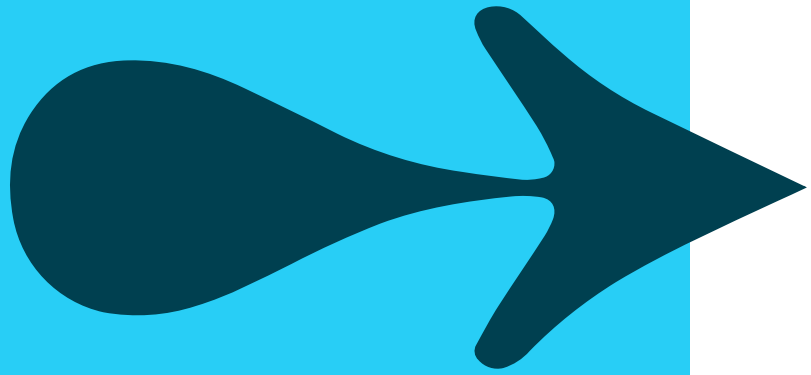
You might be tempted to butcher the HighestNumberFinder class. But this would break the tenets of clean code

- Single responsibility
- Cohesive

It would be better to design the code as follows

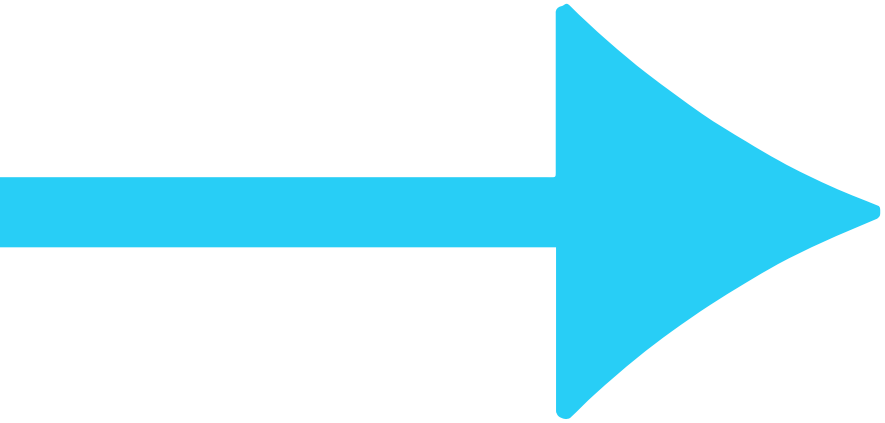
QUICKLAB

WORKING WITH DEPENDENTS



Quicklab (QLC-2.1) TDD casestudy

Topic Manager



Find the highest score for a series of topics

Given the following specification

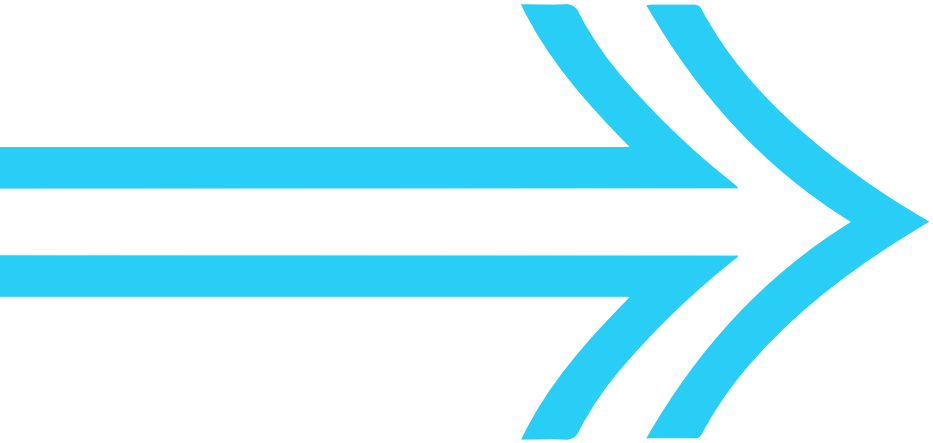
- If the input is [{“Physics”, {56, 67, 45, 89}}], the result should be [{“Physics”, 89}]
- If the input is [] the result should be []
- If the input is [{“Physics”, {56, 67, 45, 89}}, {“Art”, {87, 66, 78}], the result should be [{“Physics”, 89}, {“Art”, 87}]
- If the input is [{“Physics”, {56, 67, 45, 89}}, {“Art”, {87, 66, 78}}, {“Comp Sci”, {45, 88, 97, 56}}], the result should be [{“Physics”, 89}, {“Art”, 87}, {“Comp Sci”, 97}]

The most challenging part is determining which test to write first. Always start simple and with a test that will not need to handle exceptions.

DEPENDENTS TAKEAWAY

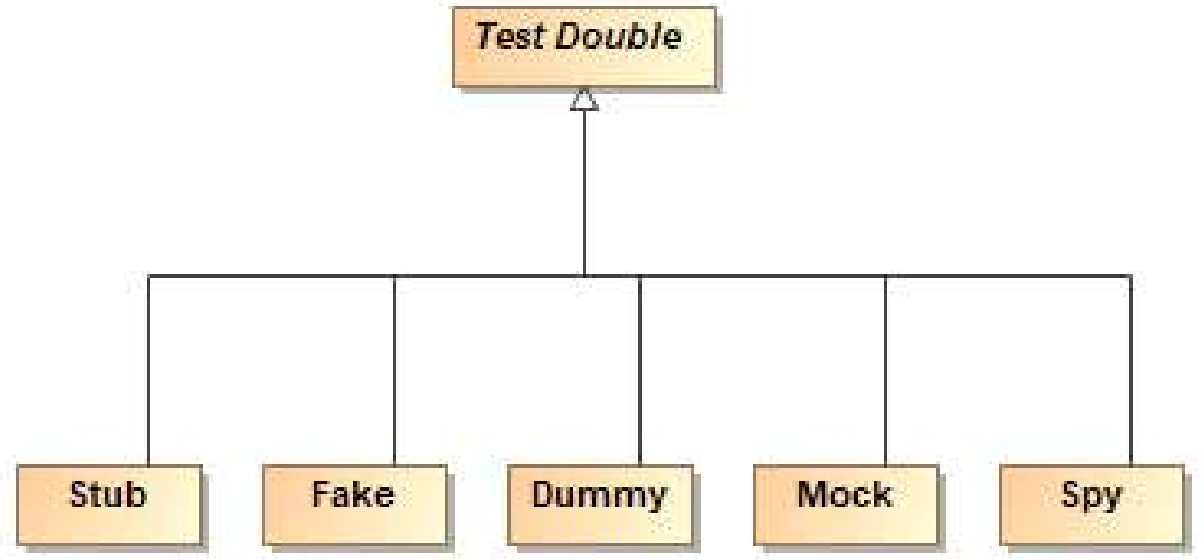
When working with tests watch out for tightly coupled code, it leads to

- Untestable code
- Testing more than the CUT in a single unit test



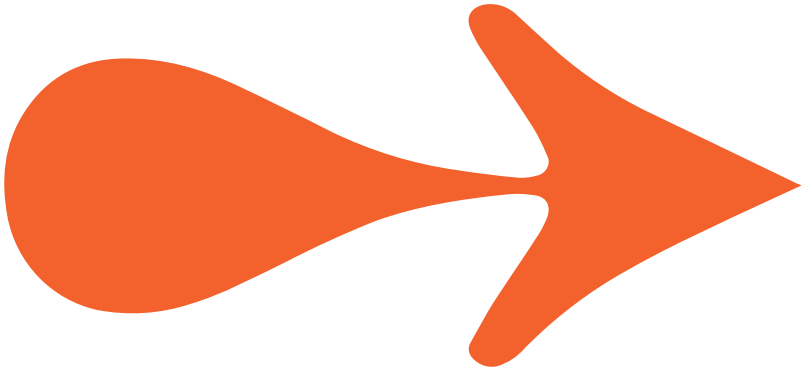
Test doubles

Test doubles types



Test doubles

- **Design technique**
- **Improves code testability**



As well as giving developers the confidence to modify their code base

You may be waiting on code from another developer

Tests are usually executed on build pipelines, they need to be isolated

We want to ensure that our tests are not performing any of the following :

- IO
- Network calls
- DB Access

Tests need to be isolated

Stubs
The problem

An object used to return canned results from methods

If FileLoader is a dependent to a CUT, calculateFileSize() is difficult to regulate

```
int loadFile(String fname){
    try{
        lines = Files.readAllLines(Paths.get(fname),
                                   StandardCharsets.UTF_8);
    }
    catch (IOException e){}

    return calculateFileSize();
}

private int calculateFileSize(){
    IntWrapper result = new IntWrapper();

    lines.forEach(line -> {
            result.value += line.length();
    });

    return result.value;
}
```

Stubs
The solution

The creation of an object predefined results from its methods

```
public class FileLoaderStub
{
   private  String fileData;

   public   int  loadData( String fname )
   {
      fileData = "some random data";

      return calculateFileSize();
   }

   private int calculateFileSize()
   {
      return fileData.length();
   }
}
```

Quicklab (QLC-2.2) TDD casestudy

TopicManager

Test Doubles - Stubs

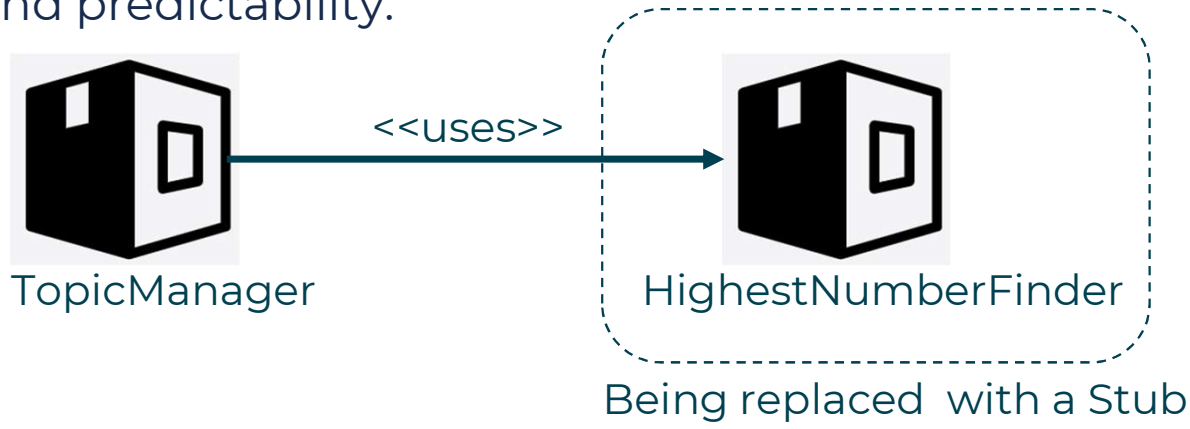
Introduction to Stubs

A Stub is part of the family of Test Doubles.

They are used to ensure that the tests focus on the behaviour of the CUT and not its dependents.

Test environments should be controlled and predictable.

Test Doubles give you that measure of stability and predictability.



TDD takeaway

Having written the tests first we were able to:

- Identify code smells
- Perform regression testing easily
- Think more clearly about the design of the code

A TDD approach has given you the confidence to modify the code

Quicklab (QLC-2.3) TDD case study

TopicManager

Test Doubles - Stubs

Continue working with Stubs

If the input is [{“Physics”, {56, 67, 45, 89}}, {“Art”, {87, 66, 78}], the result should be [{“Physics”, 89}, {“Art”, 87}]

One of the tests will fail. Why?

Stubs takeaway

When you wrote the other tests, they were failing because the stub was returning the same canned result.

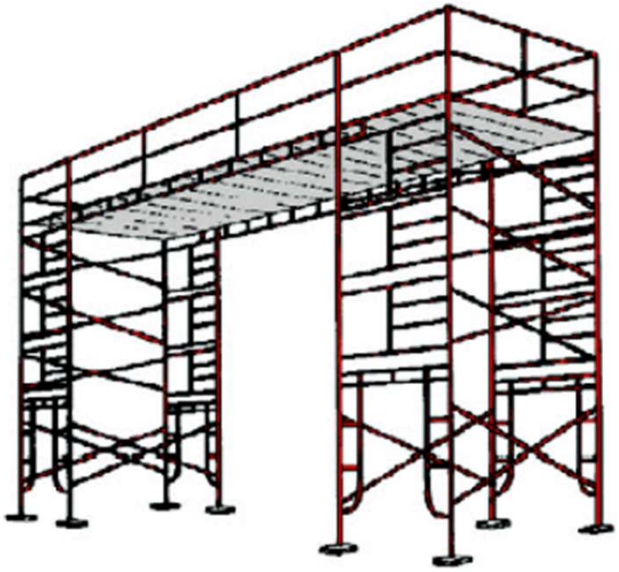
Stubs have their limitations

You could find yourself writing a lot of code to create a stub test double, in doing so you introduce errors into the tests as we just saw


Test doubles and
verification testing

Mock objects

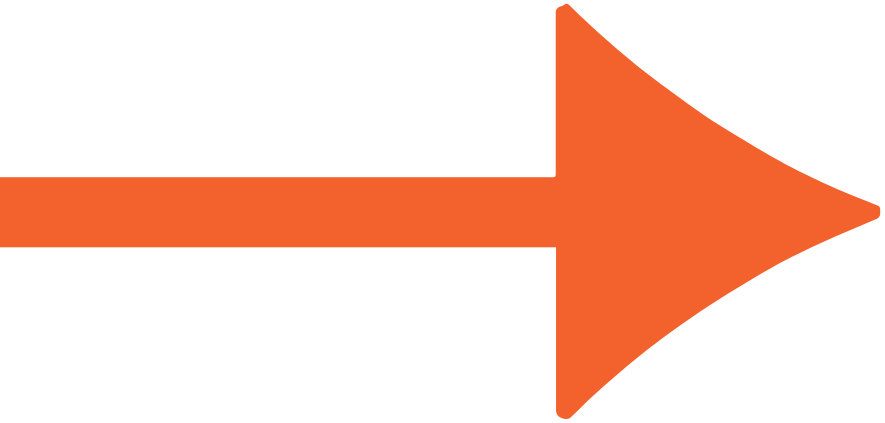
A strongly typed object that when created has methods matching the object it is replacing. These methods have no functionality – they are null, they do nothing and if invoked, will return the default value of the return type of the method unless explicitly told to do otherwise, through what is known as ‘setting the expectations’



Mocks provide the scaffolding but no real structure. Structure is added through the use of expectations

Quicklab (QL-3) code demo

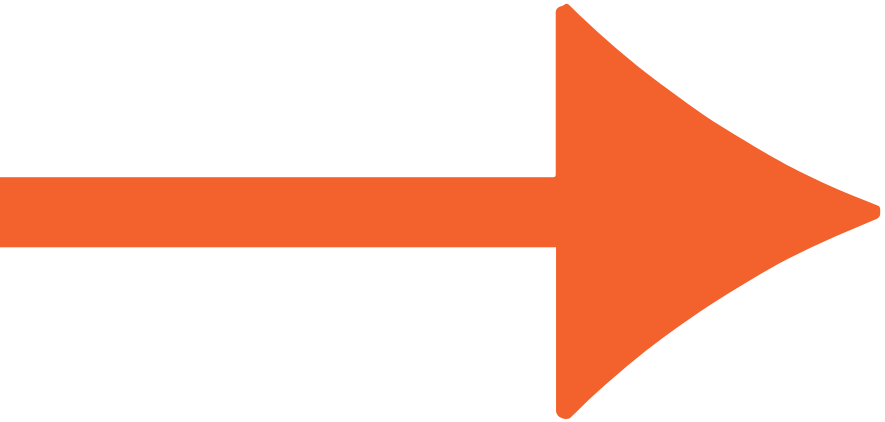
Mocks are **not** Stubs



Quicklab (QLC-2.4)

TopicManager

Test Doubles – Mocks



Working with Mocks

Continuing in a TDD manner complete the last two requirements for the TopicManager class

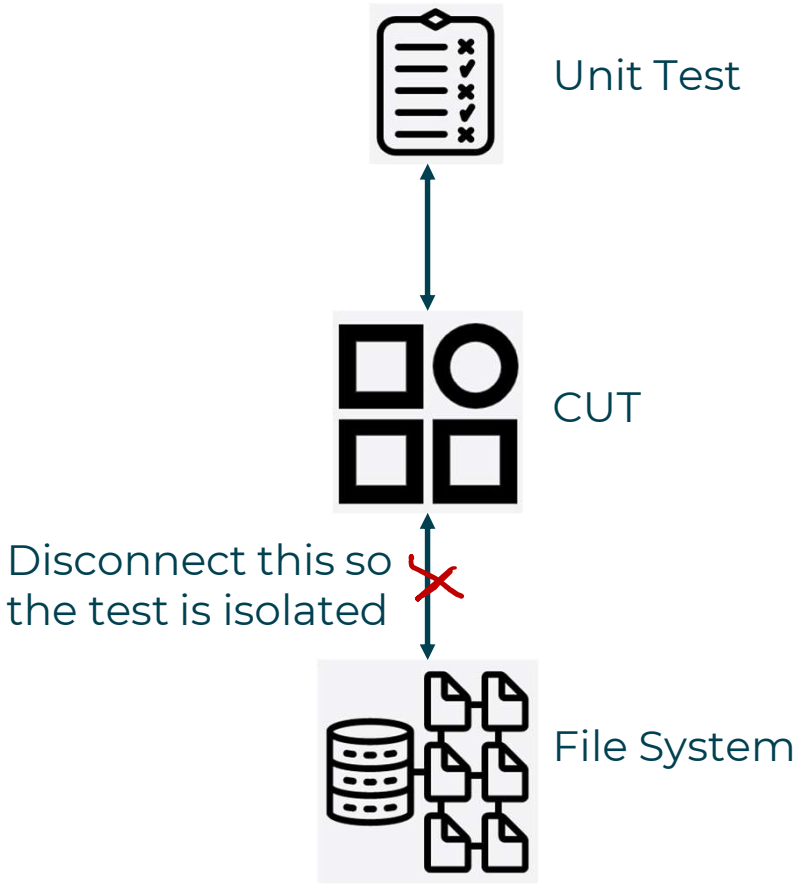
- If the input is [{“Physics”, {56, 67, 45, 89}}, {“Art”, {87, 66, 78}}, {“Comp Sci”, {45, 88, 97, 56}}], the result should be [{“Physics”, 89}, {“Art”, 87}, {“Comp Sci”, 97}]

Use a Mock in place of the production HighestNumberFinder class

Working with Mocks

We are going design and refactor code using a TDD approach and Mocks.

The final CUT will be called FileLoader. It's job is to load a text file from disk, and return the number characters (excluding CR/LF) in the file.



Quicklab (QL-4.1)

The unit test

Write the unit test first for a CUT called FileLoader. It should only have one public method on it. Once the test is written, develop the CUT to pass the test.

```
public int loadFile(string fname)
```

Question: What's the weakness in this design?

Quicklab (QL-4.2)

Using Lambdas

Create a new test that allows a lambda expression to be passed to loadFile() on the CUT.

Add a new loadFile() method to the CUT that takes a lambda expression. A stub can then be passed as the lambda expression.

Hint: use a delegate to create the lambda signature.

Quicklab (QL-4.3)

Using Stubs

Add a new test that makes use of the ability to pass a lambda to loadFile() on CUT. In the lambda substitute, the data read from a file with dummy data in a List. Each item in the list representing one line in a file.

Quicklab (QL-4.4)

Replacing the Stub with a Mock

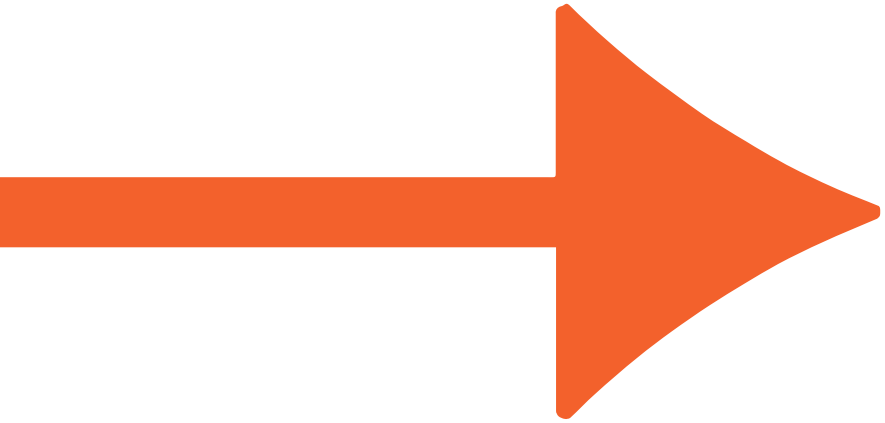
Wrap the C# library called File.ReadLines() in a normal class with a method matching File.ReadLines.

Mock your wrapper class so that your ReadLines() returns dummy data.

Pass a lambda to LoadFile. In the lambda call your mocked File object's ReadLine().

Quicklab (QLC-3)
TDD casestudy

TopicScoreWriter
Test Doubles – Mocks



Mocking objects takeaway

Mocks are **not** stubs.

Contrary to common thought, they are used to verify the interaction between the CUT and its dependents

Stubs are still valid and should be used when you need to mock the data

- Mock objects are a shorthand to creating stubs for mocked data but that's not their purpose

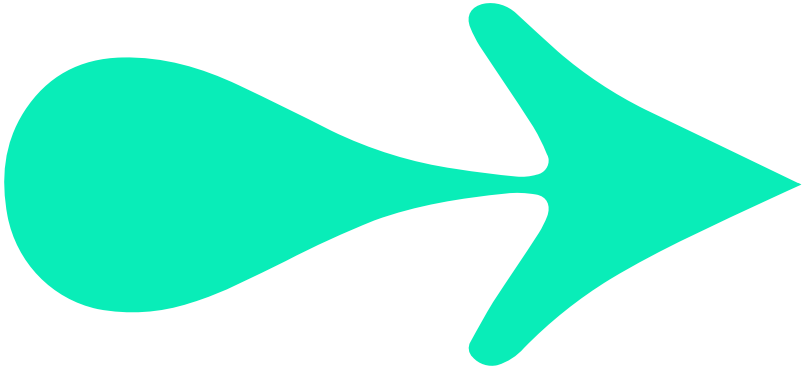
Spy object

A strongly typed object that, when created, has methods matching the object it is replacing. These methods are functionality equivalent to those on the object that is being spied upon. If any of methods are invoked, they will call the real object's method. If the Spy has expectations that override the real object's methods, the overriding method will be called.

Spies provide the scaffolding around an already existing structure. The structure can be modified through the use of expectations

UNTESTABLE CODE

What is untestable code?

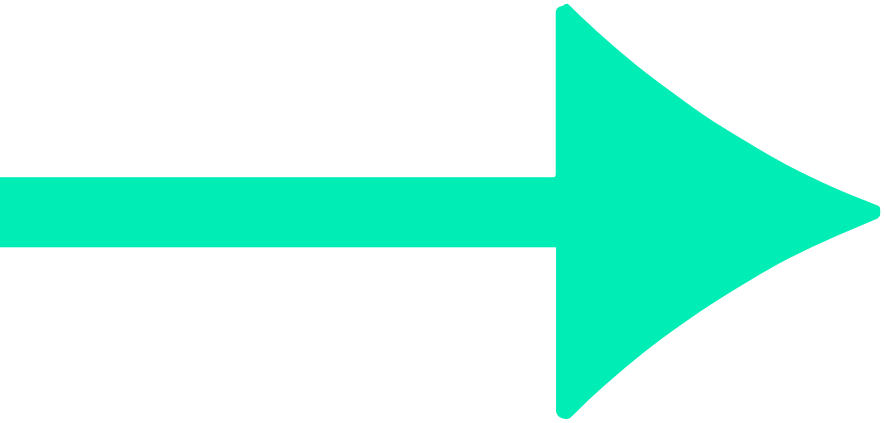


Code for which it is difficult to

- Isolate – makes use of IO, networks, or external DB
- Determine its state after invoking methods on it (even if those methods have parameters)
- Logic paths that are difficult to reach; such as time constraints, e.g., only executes logic path at specific points in time
- Tightly coupled to other pieces of code that require configuring to run
- Legacy code for which you don't have the source code but are dependent upon it

Quicklab (QL-5)

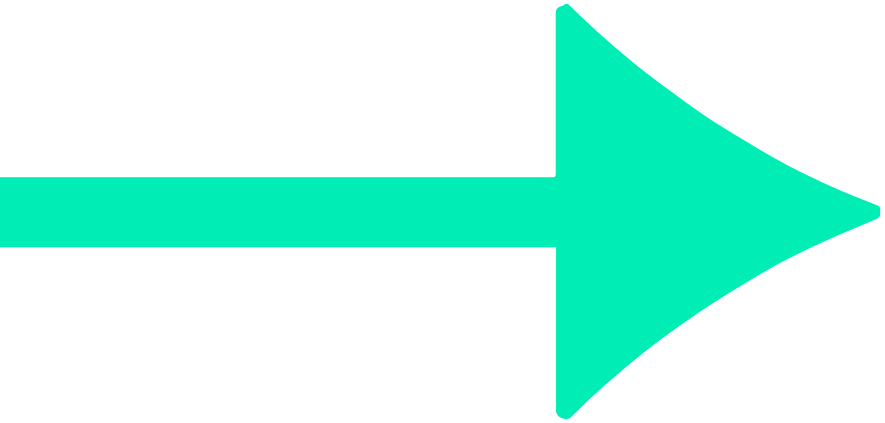
Working with untestable code



Here we provide ideas on how to deal with elements in your software systems

Quicklab (QLC-4)
TDD casestudy

There is no lab



Conclusion

- TDD is not as complex as it may first seem
- Unit tests are ideal for identifying weaknesses in your design
- Unit tests combined with a TDD philosophy can really help you design well structured and designed code