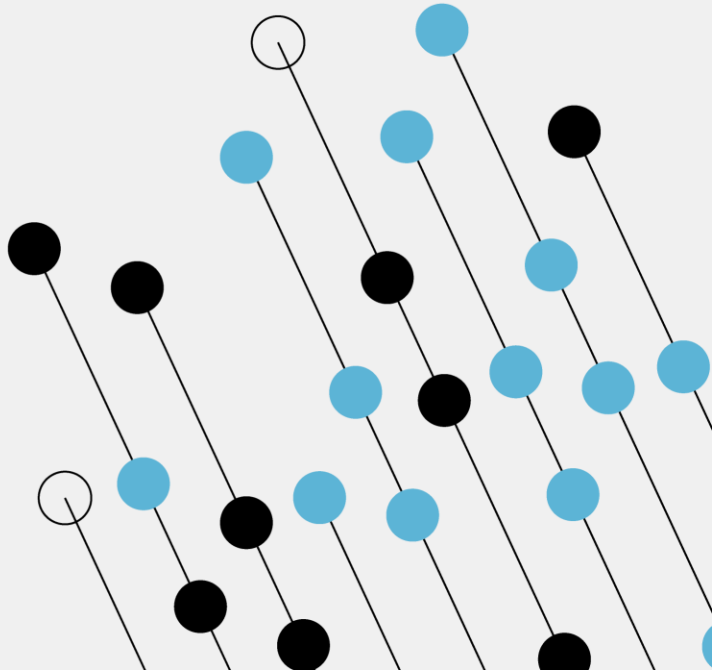
Course Slides

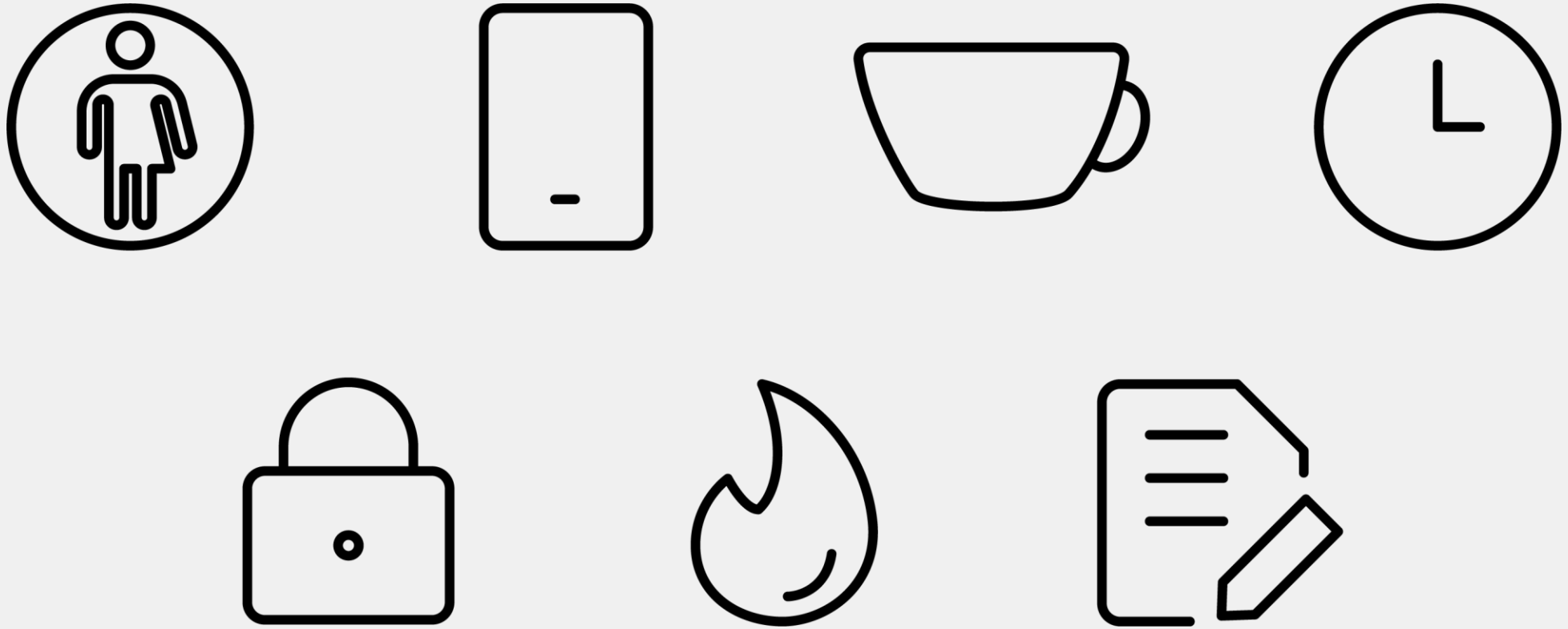
TDD – Test Driven Development

QA

Learn. To Change.



Housekeeping



Course Delivery

Hear and forget

See and remember

Do and understand

Your Training Experience

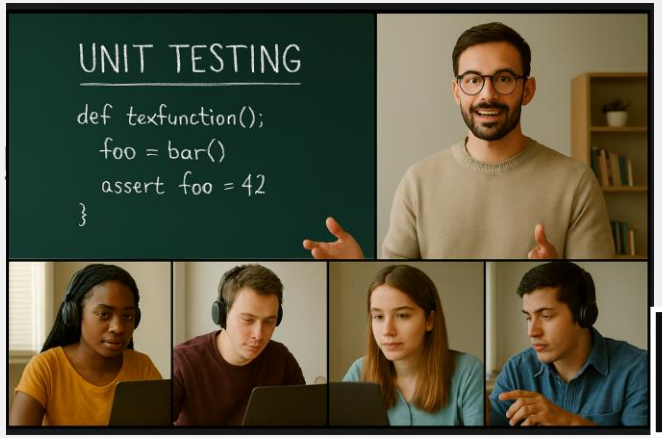
A course should be:

- A two-way process
- A group process
- An individual experience
- Hands-on
- Engaging

There is no such thing as a stupid question!

Even when asked by a trainer!

A Question never resides in a single mind!

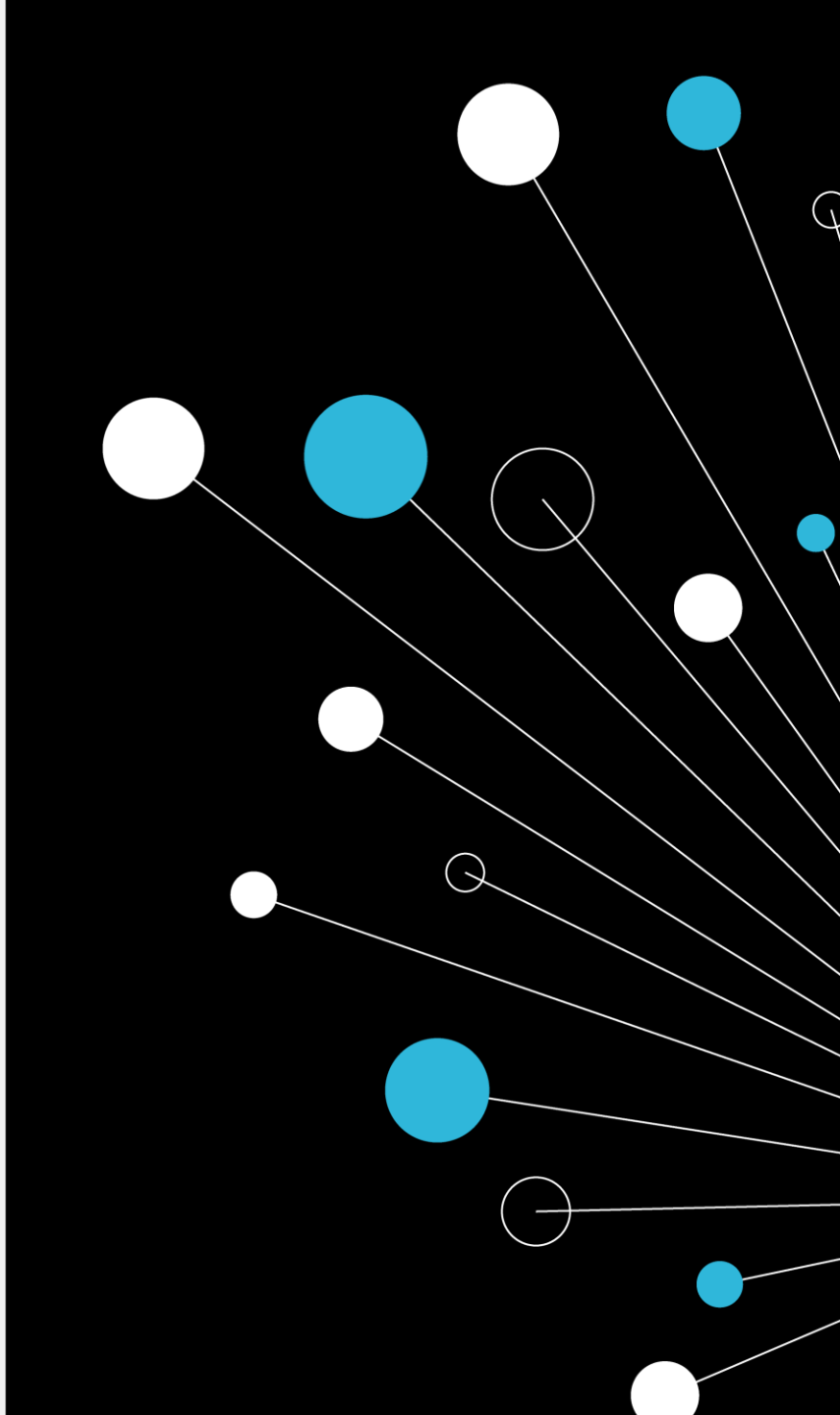


Keep Webcams ON

Learning objectives

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 Python
- 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
- Understand what **Mutation** Testing is



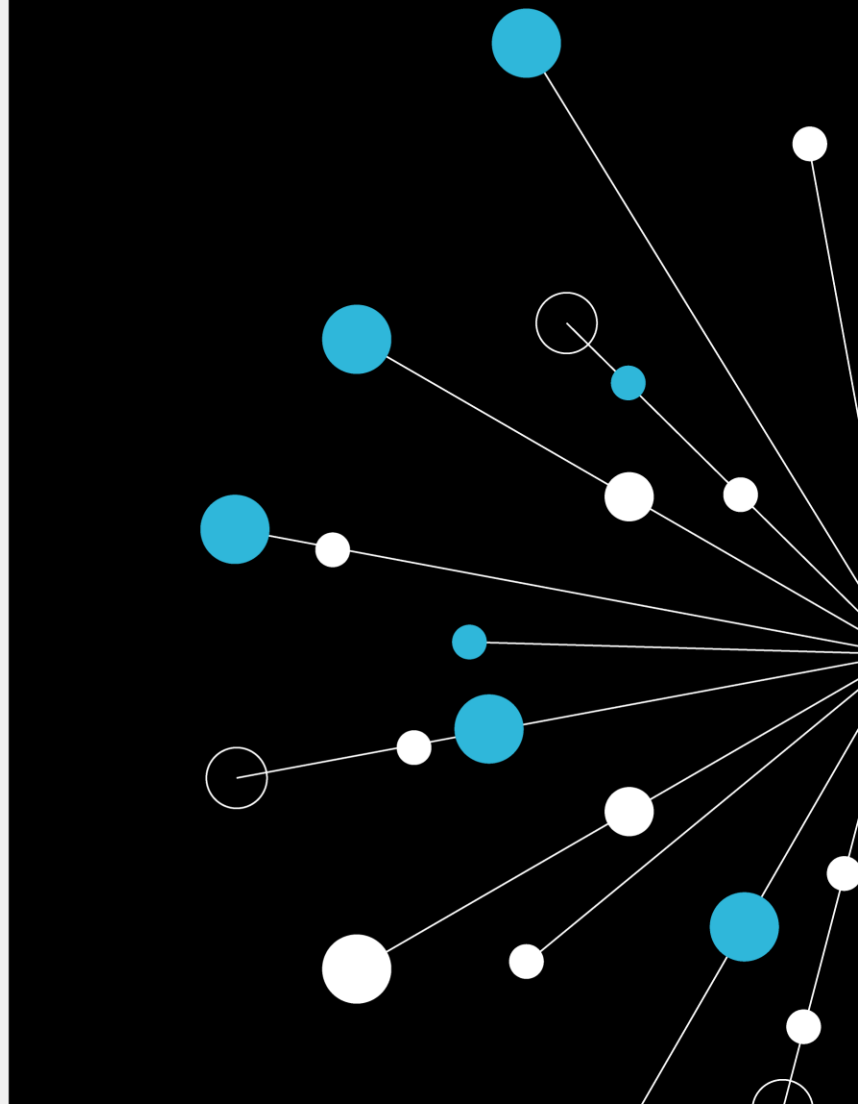
QA

Pre-requisites

This course assumes the following prerequisites:

- You have a basic knowledge of Python development
- You have a basic knowledge of OOP
- You have used an IDE like Pycharm or VSCode

If there are any issues, please tell your instructor now



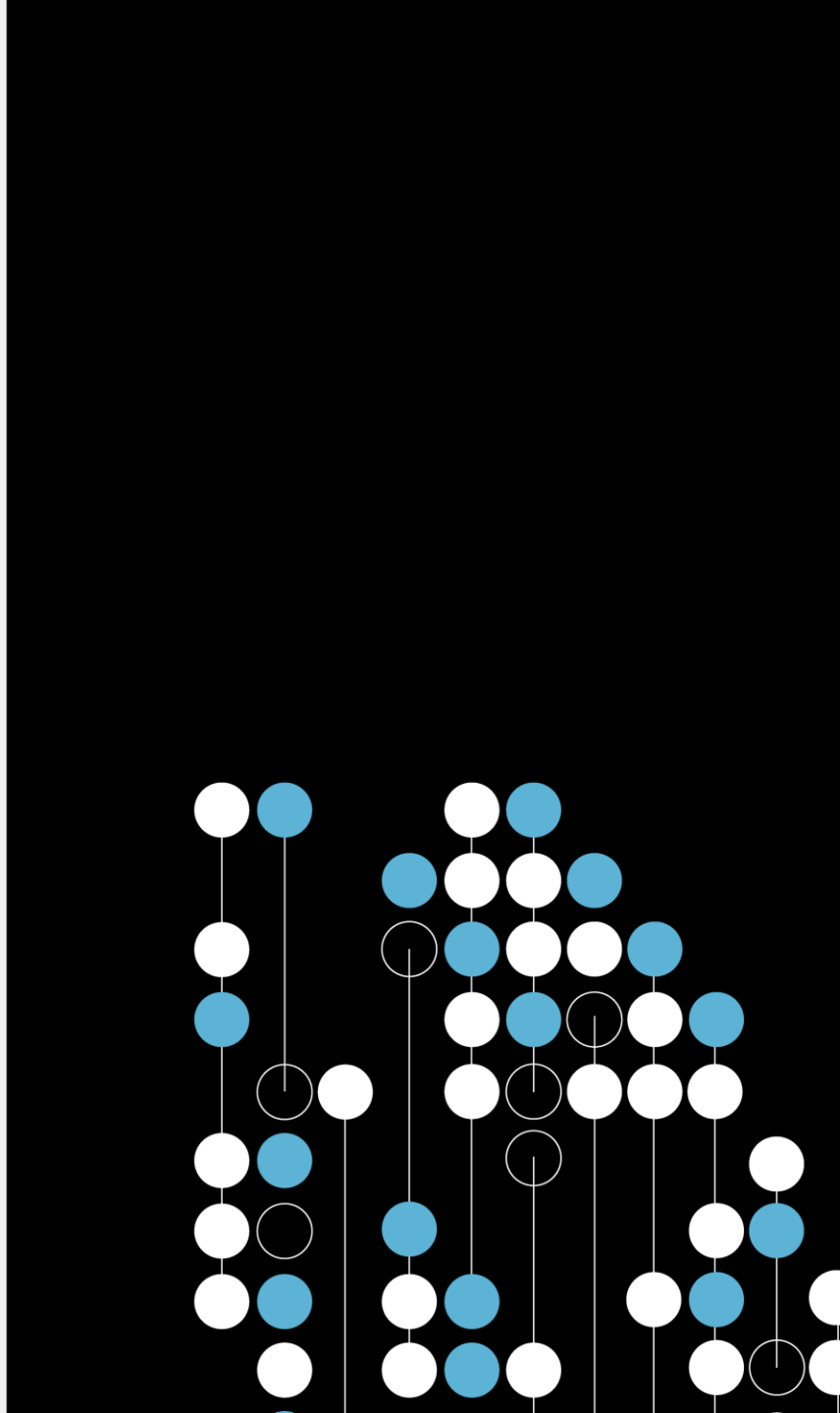
QA

Introductions

Please say a few words about yourself:

- What is your name and job?
- What is your current experience of:
- Python (rate 0-5)?
- Programming (Other Languages)?
- Testing (Have you written a Unit Test)?

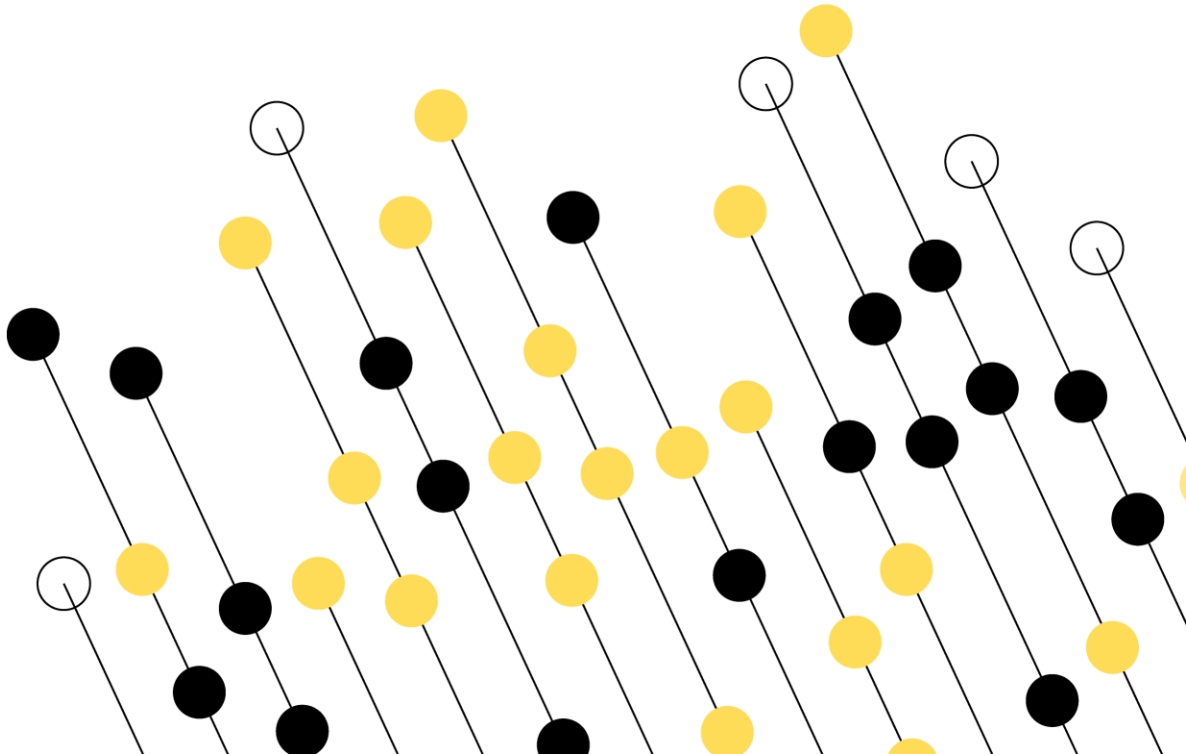
What is your main objective for attending the course?



QA

TDD

Test Driven Development



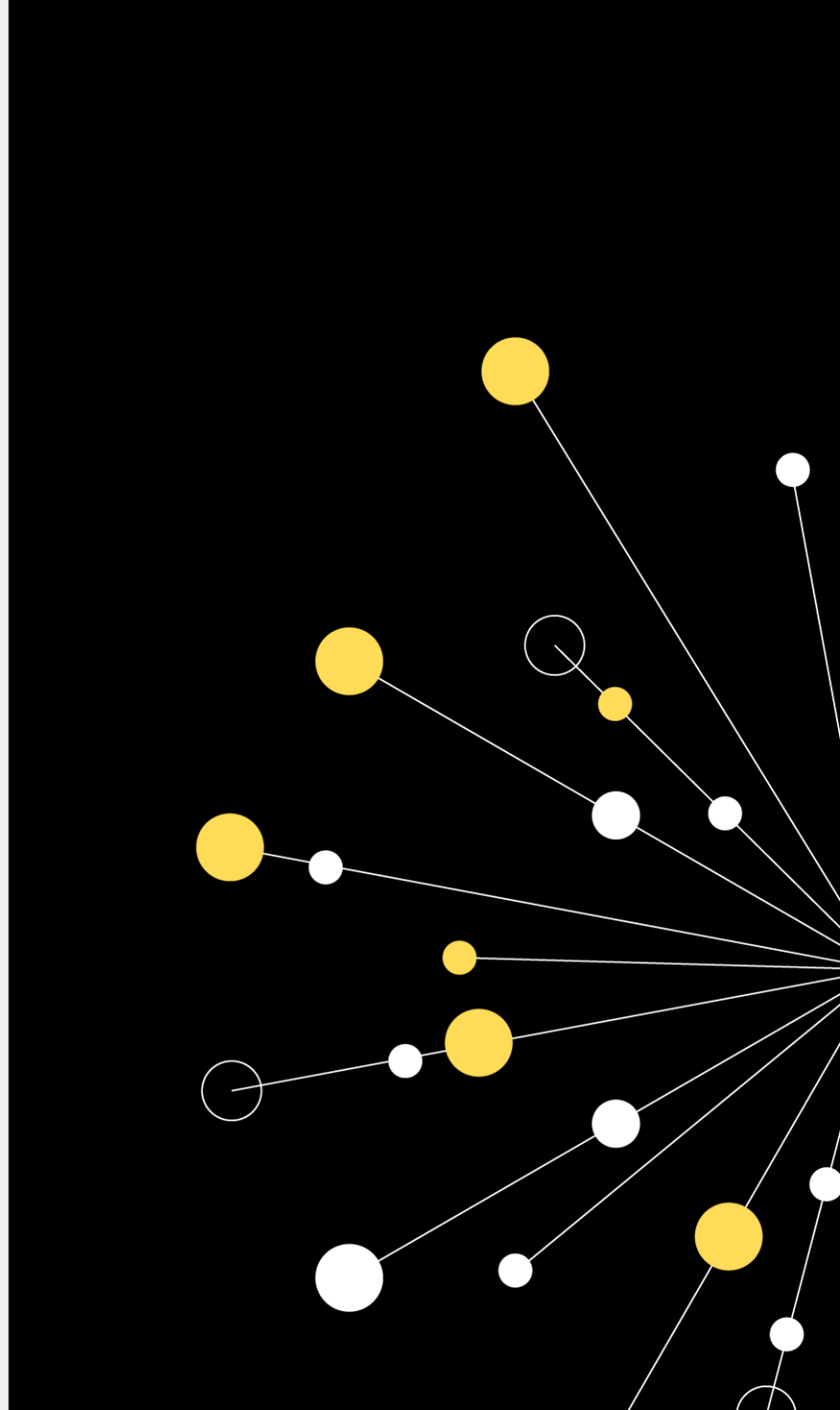
What’s the problem?

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 printouts to standard out, so no real regression is 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



QA

TDD is an Approach and 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 to Development

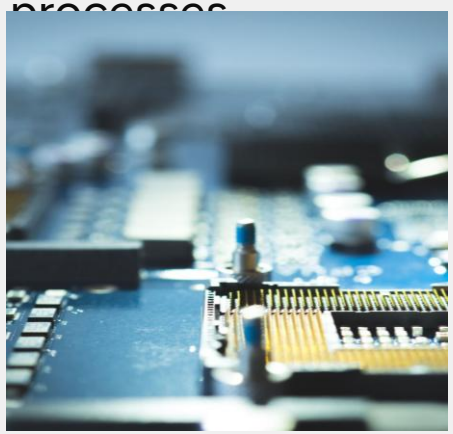
The software industry didn't invent the idea of tests first, development after (TDD).

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 before 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 Coverage

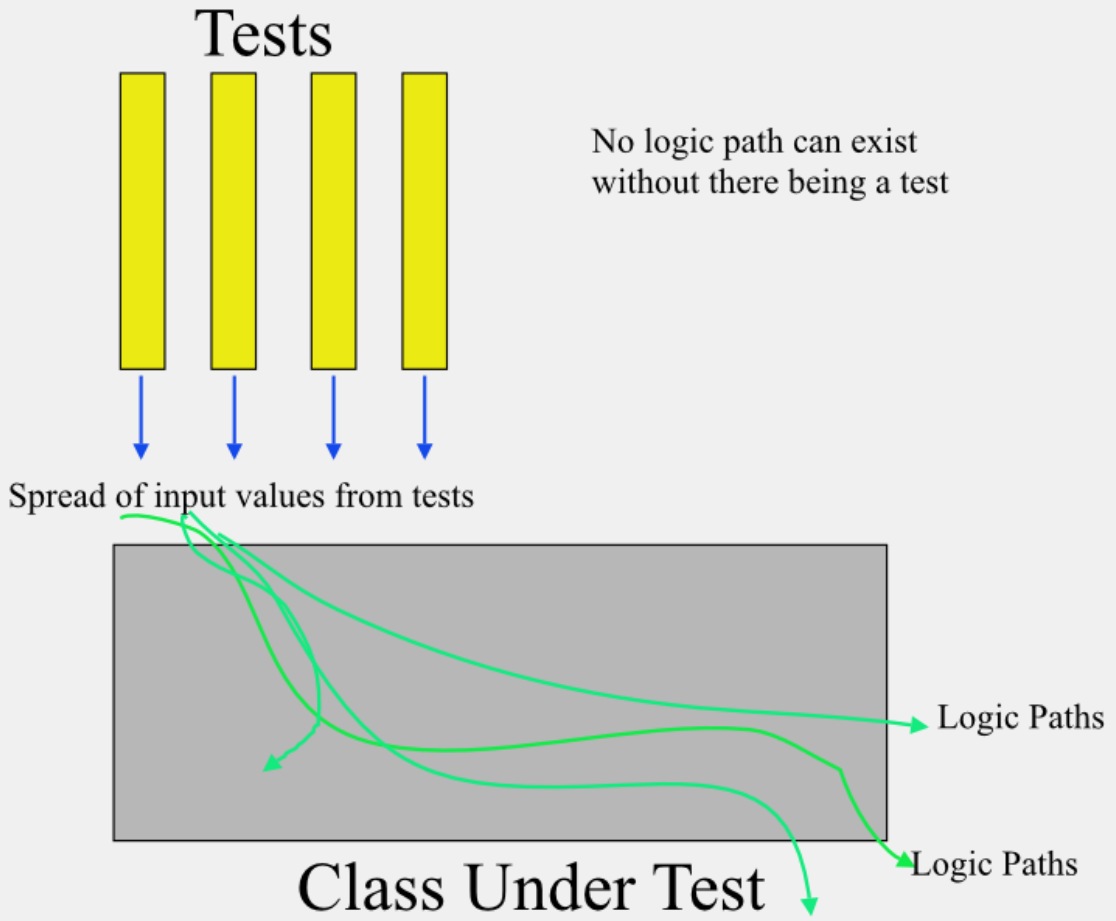
Test before approach

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

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

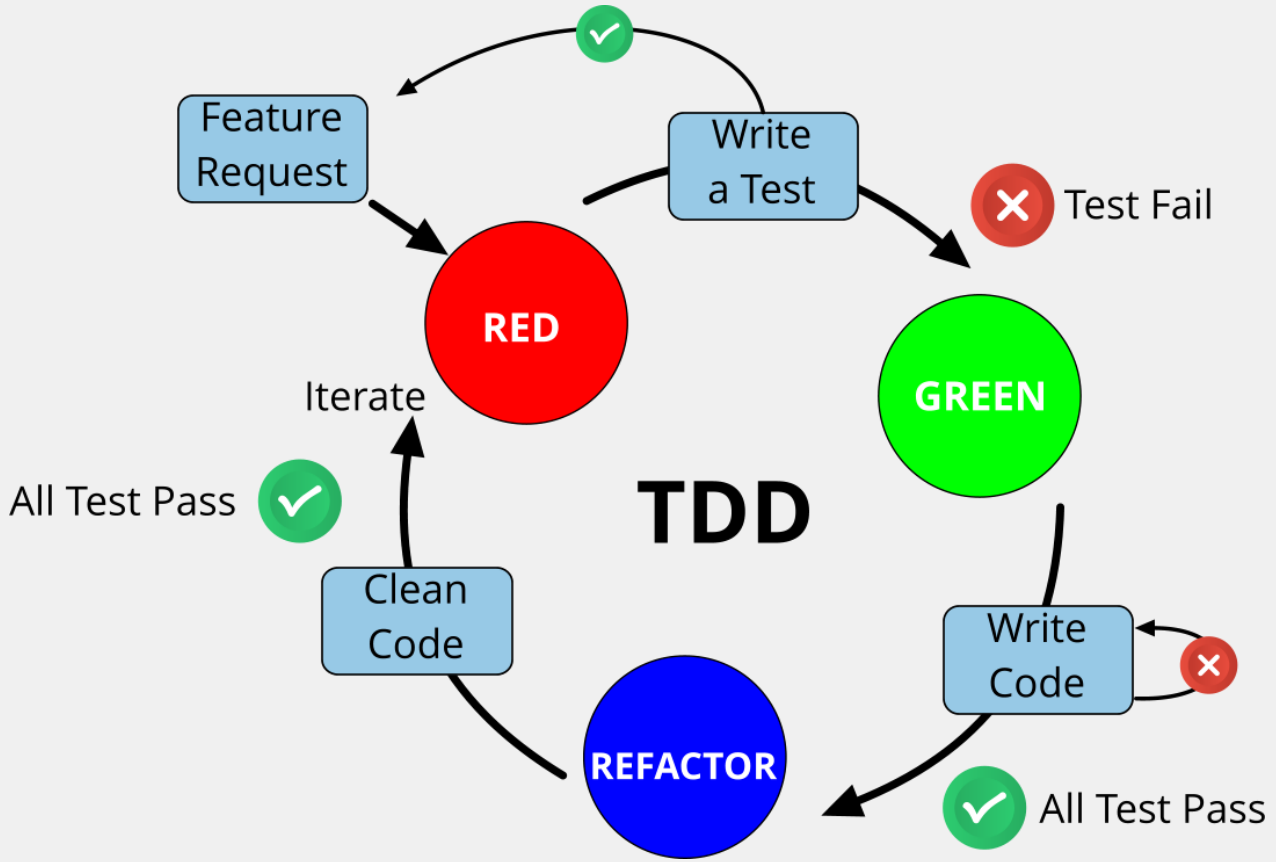
Many organisations have policies of not testing setters and getters.

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



TDD Lifecycle

1. Feature Request
2. **Write a test**
3. **Write just enough code to pass the test**
4. When test passes, push code to repo
5. **Refactor code if needed**
6. Ensure tests are still passing
7. Push code to repo



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.

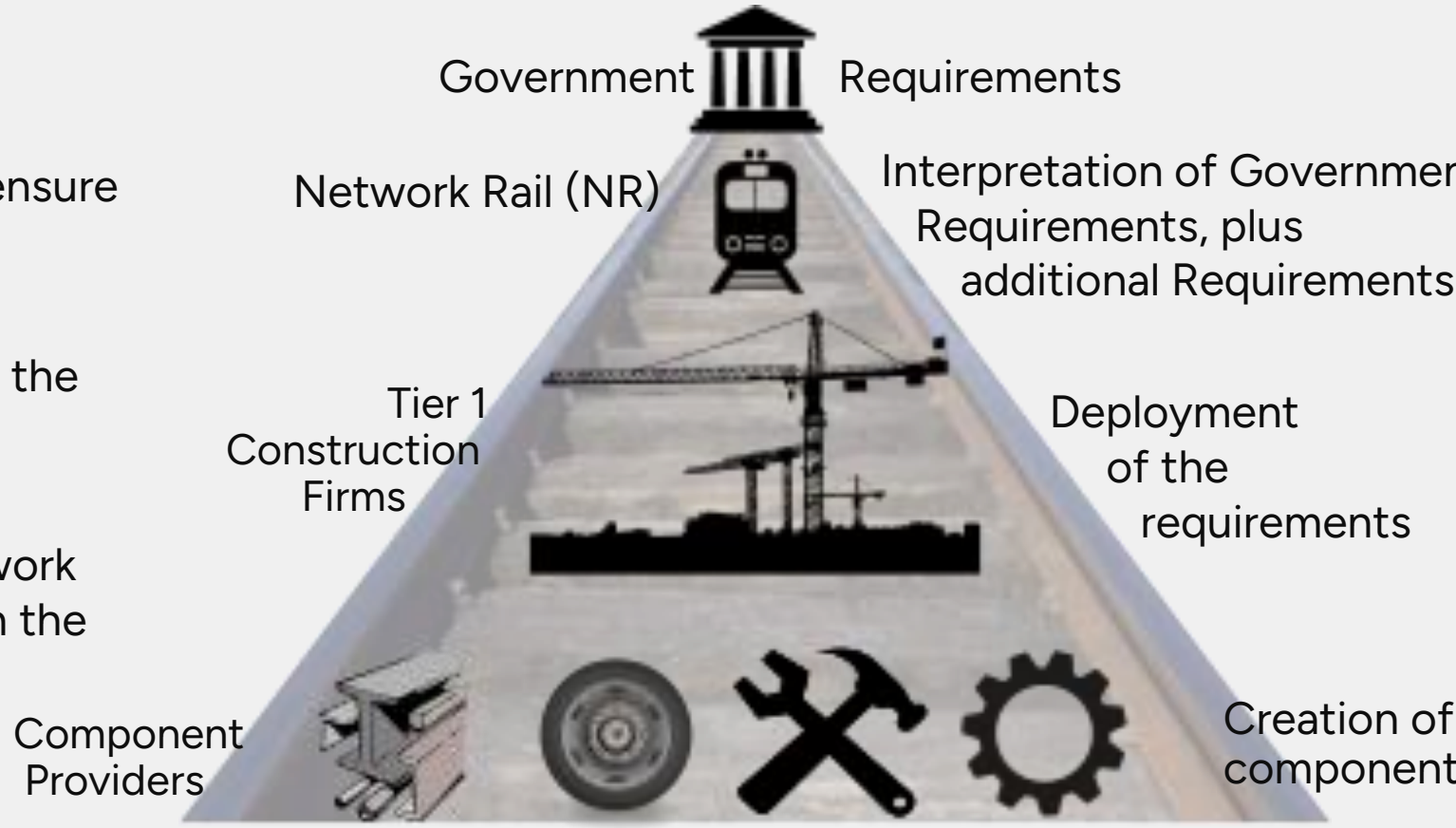
Upgrade Scenario

Birmingham to London

- New track
- New route
- Must accommodate
  - New high-speed rolling stock
  - Engines and carriages
- Old tracks would buckle under new rolling stock.

Hierarchy of Commands

1. The government mandates certain requirements on the rail industry.
2. NR interprets those requirements and ensure that they are delivered by the tier 1 construction firms.
3. NR may also add other requirements to the mix based on their experience and knowledge of the industry.
4. NR and tier 1 construction companies work with component providers to decide on the components required to meet the requirements.

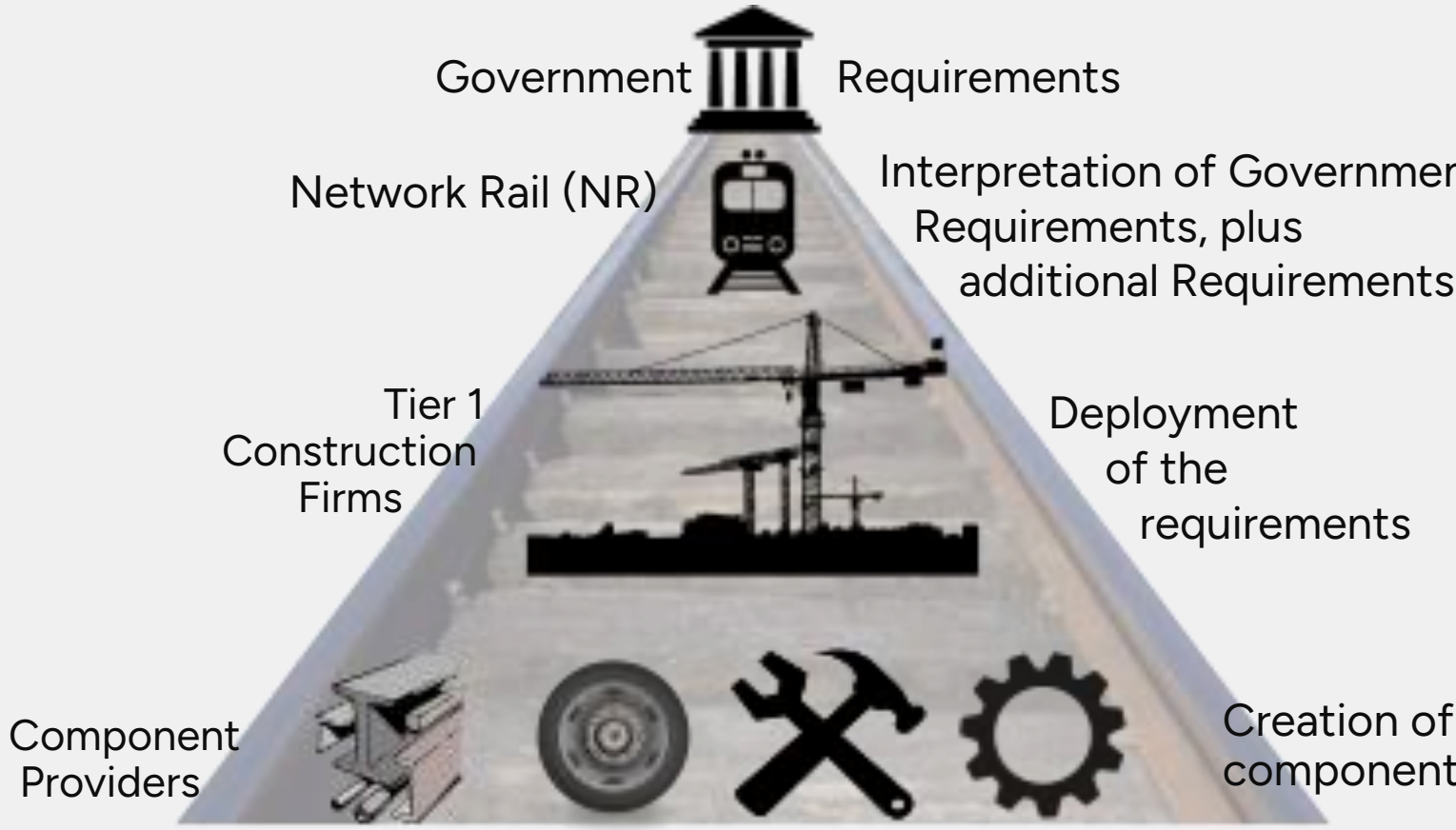


So where is the Unit Test?

If there is a trackside problem who is 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 as state testing

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

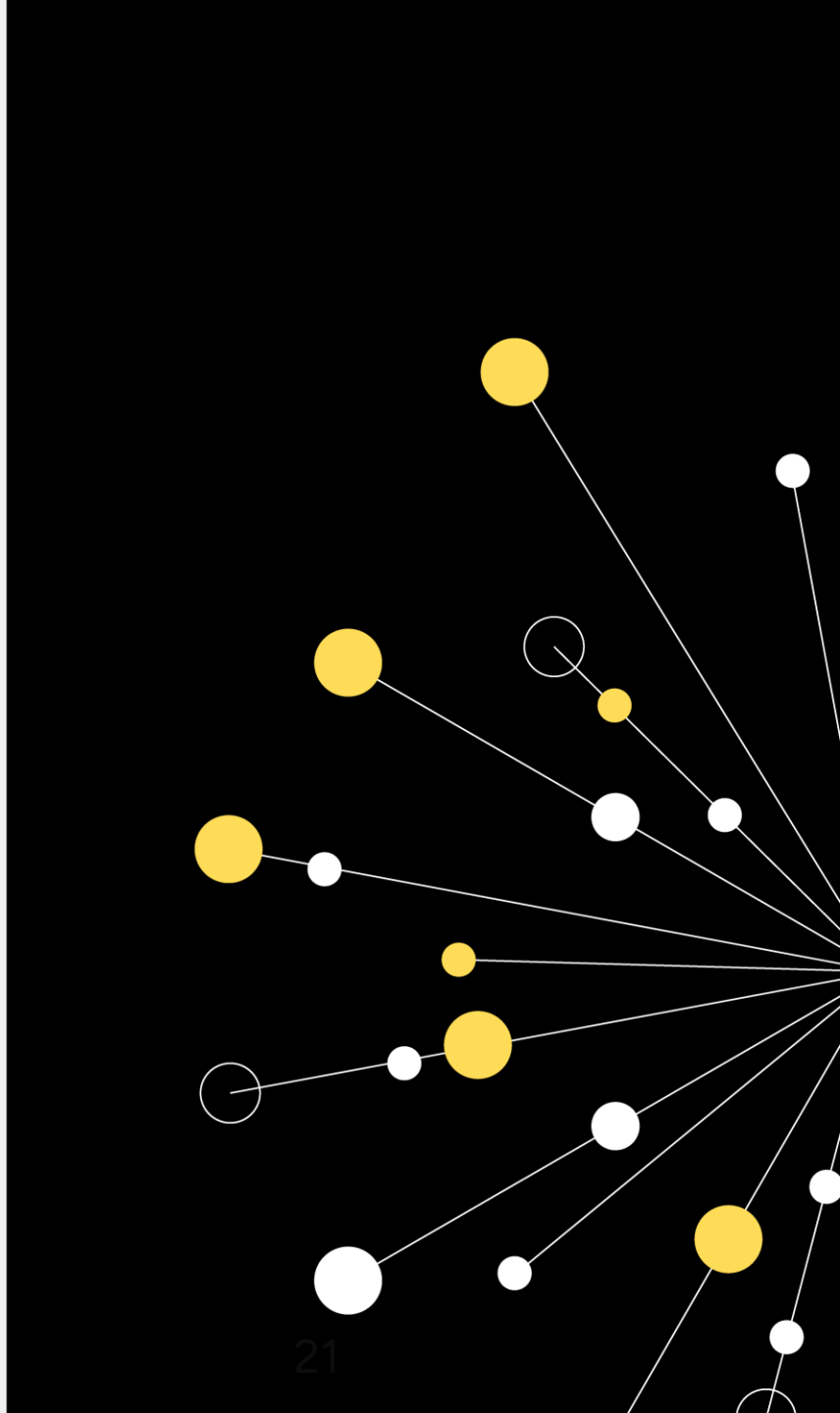
- There is no magic bullet

Quicklabs – TDD Walkthrough

String Tokeniser

You will develop a small program that accepts a comma-delimited string of tags. Each tag may consist of characters, numbers, and certain symbols (e.g., \$£%&), but not commas. The program should return a **list of tags**, adhering to the following rules:

- Leading and trailing commas must be removed.
- Leading whitespace before the first tag and trailing whitespace after the last tag must be trimmed.
- Any contiguous sequence of words separated by spaces and ending with a comma should be treated as a **single tag**.

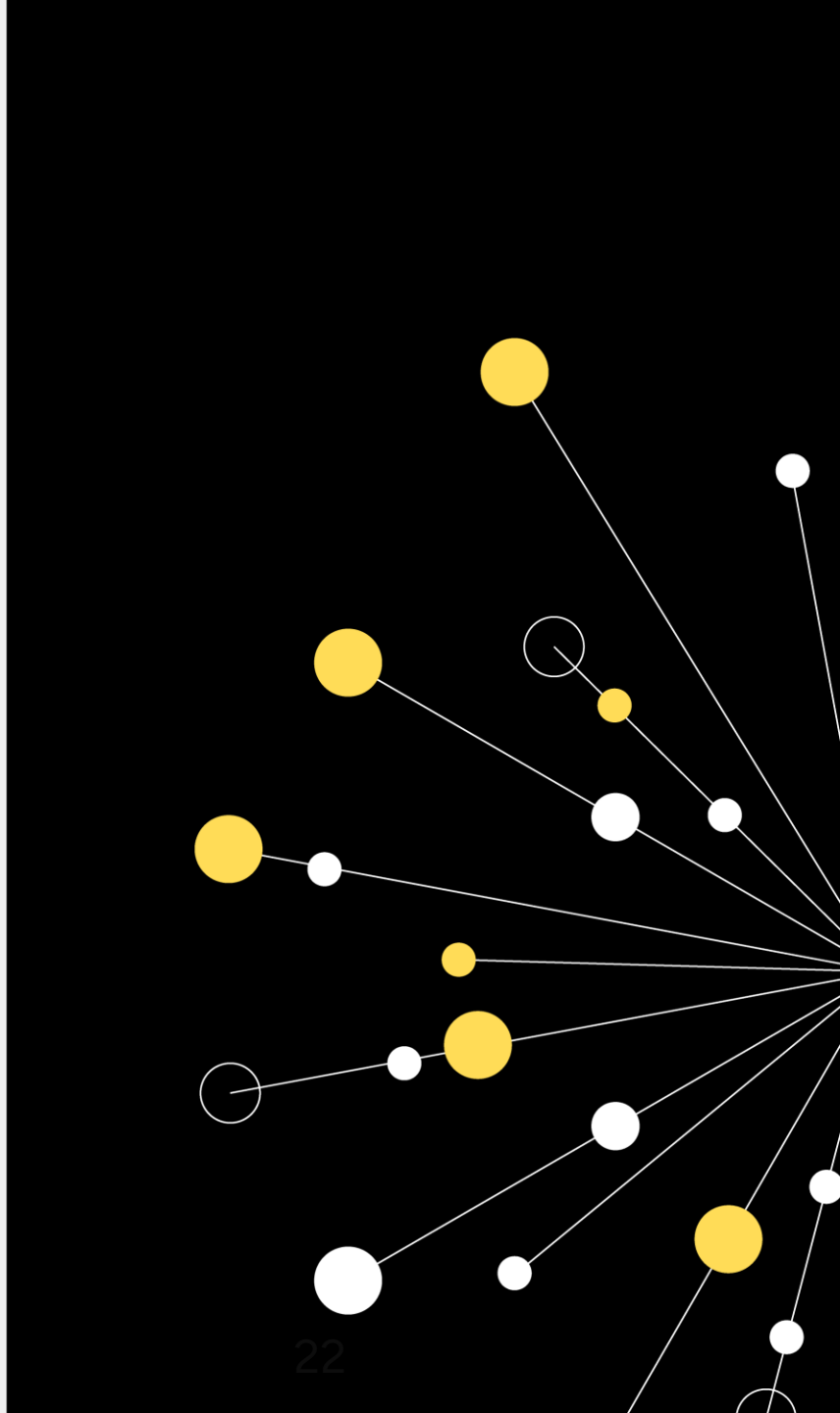


QA

Quicklabs – TDD Lifecycle

QL-2 TDD

UK Car Registration Plates [In Exercise Guide]



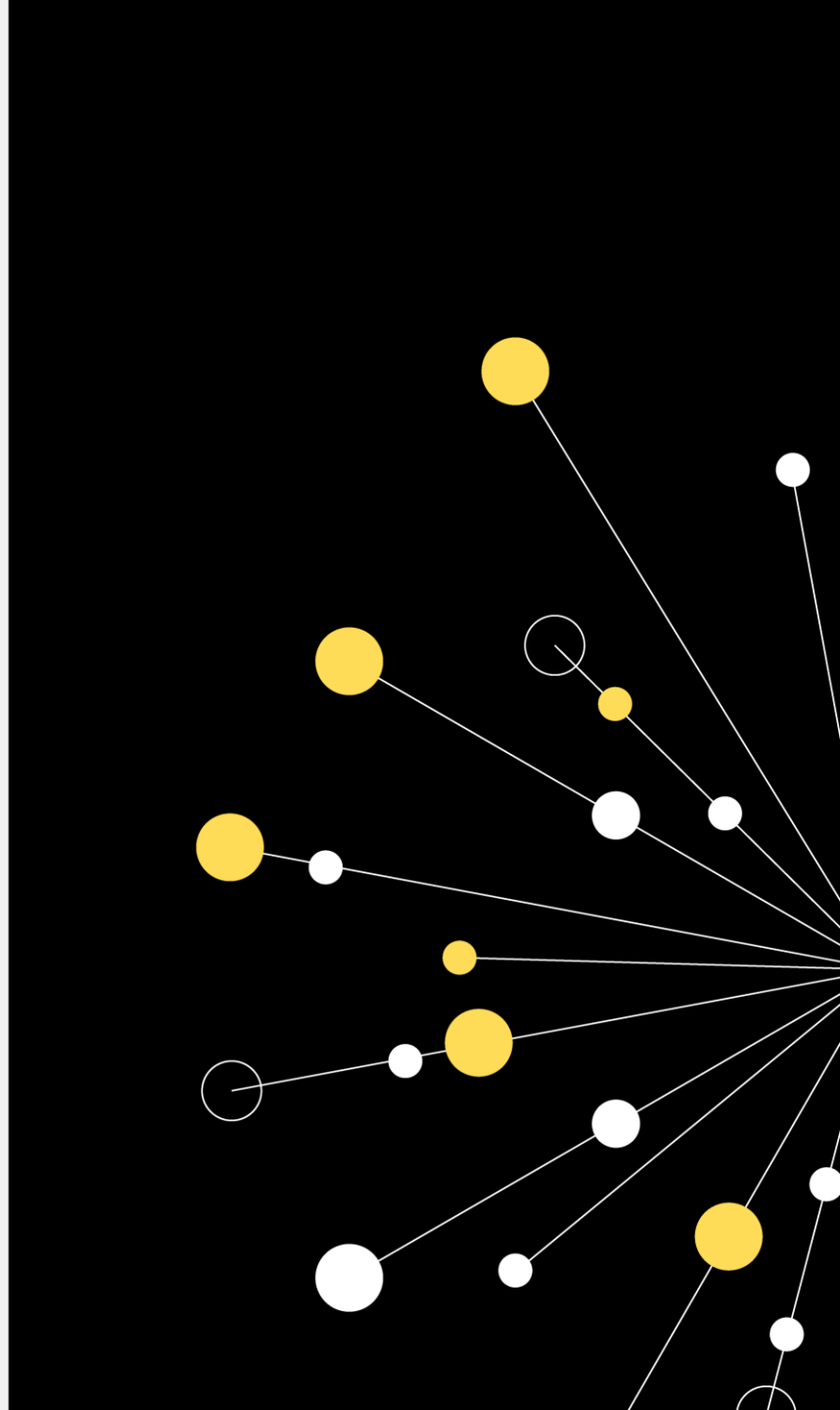
TDD Takeaways

TDD comprises a series of unit tests

You need to work with a source control platform

1. We follow a RED, GREEN, REFACTOR strategy
2. Write a test (it's failing – RED)
3. Write only enough production code to pass the test (it passes – GREEN)
4. Don't write any more code than is required to pass the test
5. When a test passes, commit your code to a source control repo
6. If you need to refactor the code, do so, but ensure tests are still passing, then push your code to a source control repo
7. Move on to 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



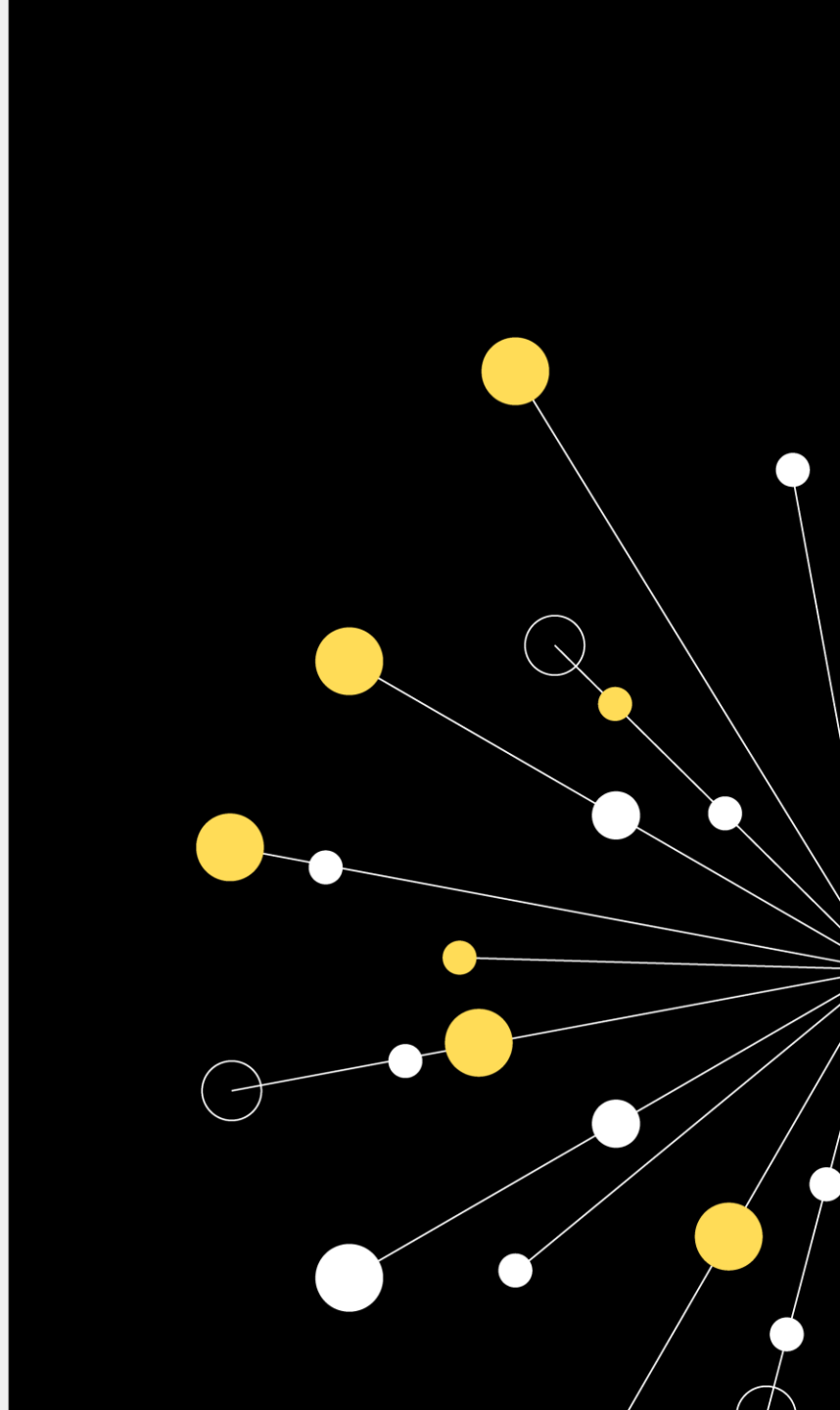
Quicklabs – TDD Case Study

QLC-1 - Highest Number Finder

Given the following specification:

- If the input were [4, 5, -8, 3, 11, -21, 6] the result should be 11
- An empty list should throw an exception
- A single-item list 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

Note: **The most challenging part is determining which test to write first.** Always start simple, with a test that does not need to handle exceptions.

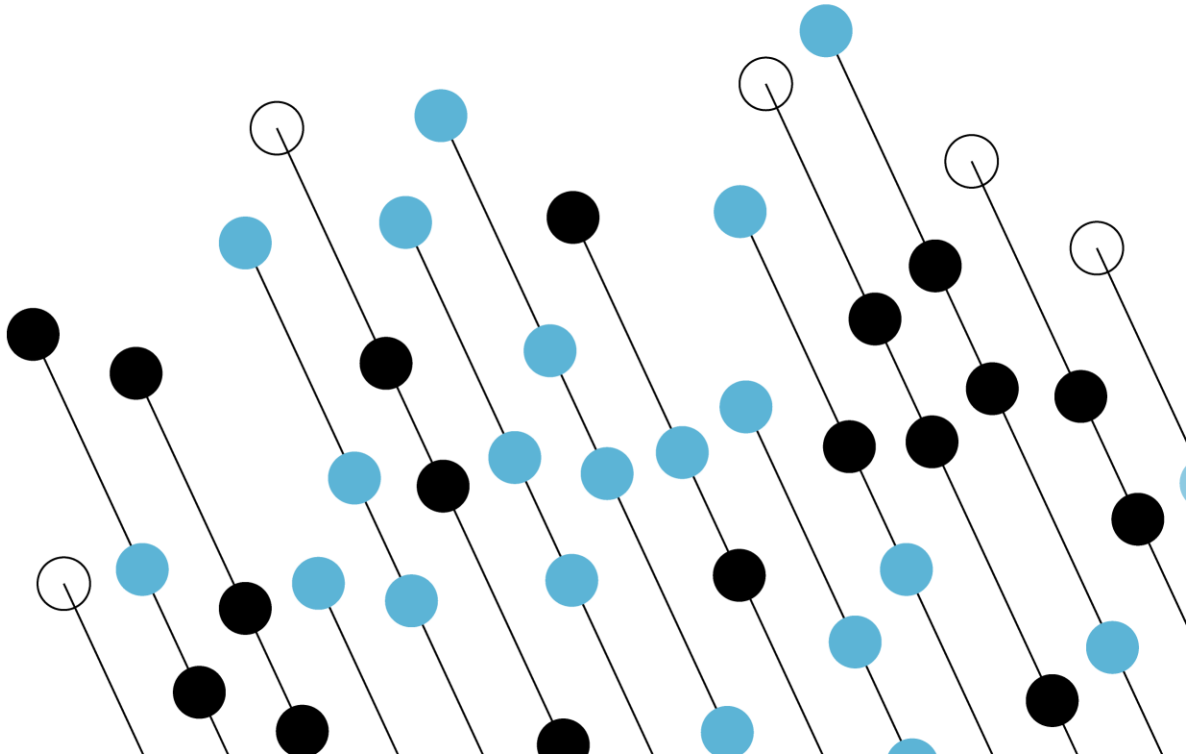


QA

Anatomy of a Unit Test

Structure of a Unit Test

Do’s and Don’ts of good Unit Tests



QA

Recall our 1st Test Method

```
class TestHighestNumberFinder(unitest.Testcase):
    def test_find_highest_in_list_of_one_expect_single_item(self):
        # Arrange
        numbers = [10]
        cut = HighestNumberFinder()
        expected_result = 10

        # Act
        result = cut.find_highest_number(numbers)

        # Assert
        self.assertEqual(expected_result, result)
```


QA

Qualities of a Good Unit Test

Isolated – does not depend on any other unit test.

Comprises of the **three A's**:

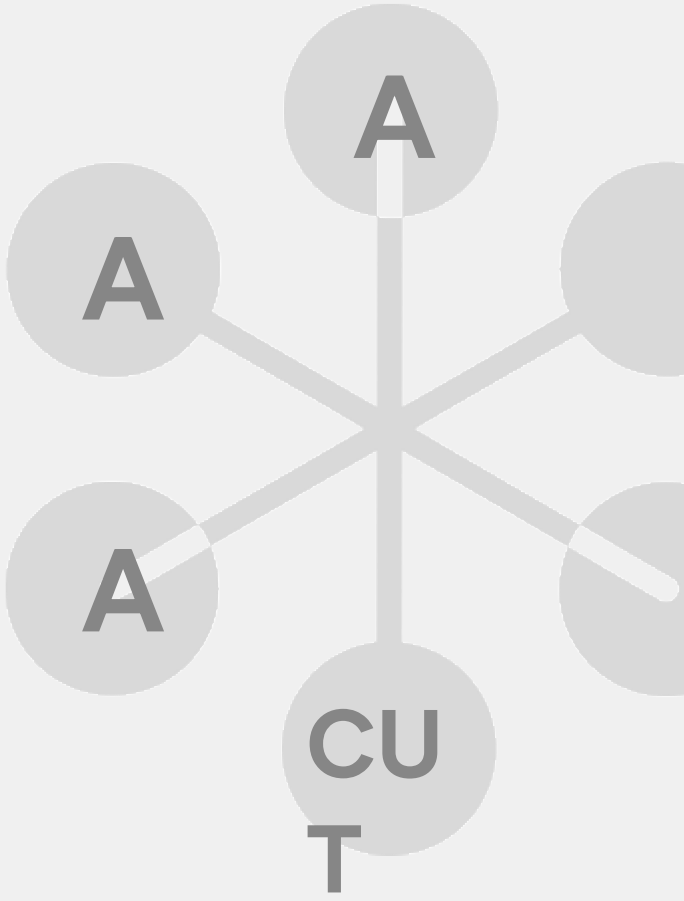
1. **A**rrange
2. **A**ct
3. **A**ssert

The object being tested is usually named as the **CUT**

Does NOT perform (must use test doubles instead!):

- IO
- Network calls
- DB Access

Quick – the tests execute in milliseconds.



Implementation Class

```
class HighestNumberFinder:
    def find_highest_number(self, numbers):
        """
        Returns highest number from the list.
        Assumes the list is not empty and
        Contains only integers.
        """
        return numbers[0]
```

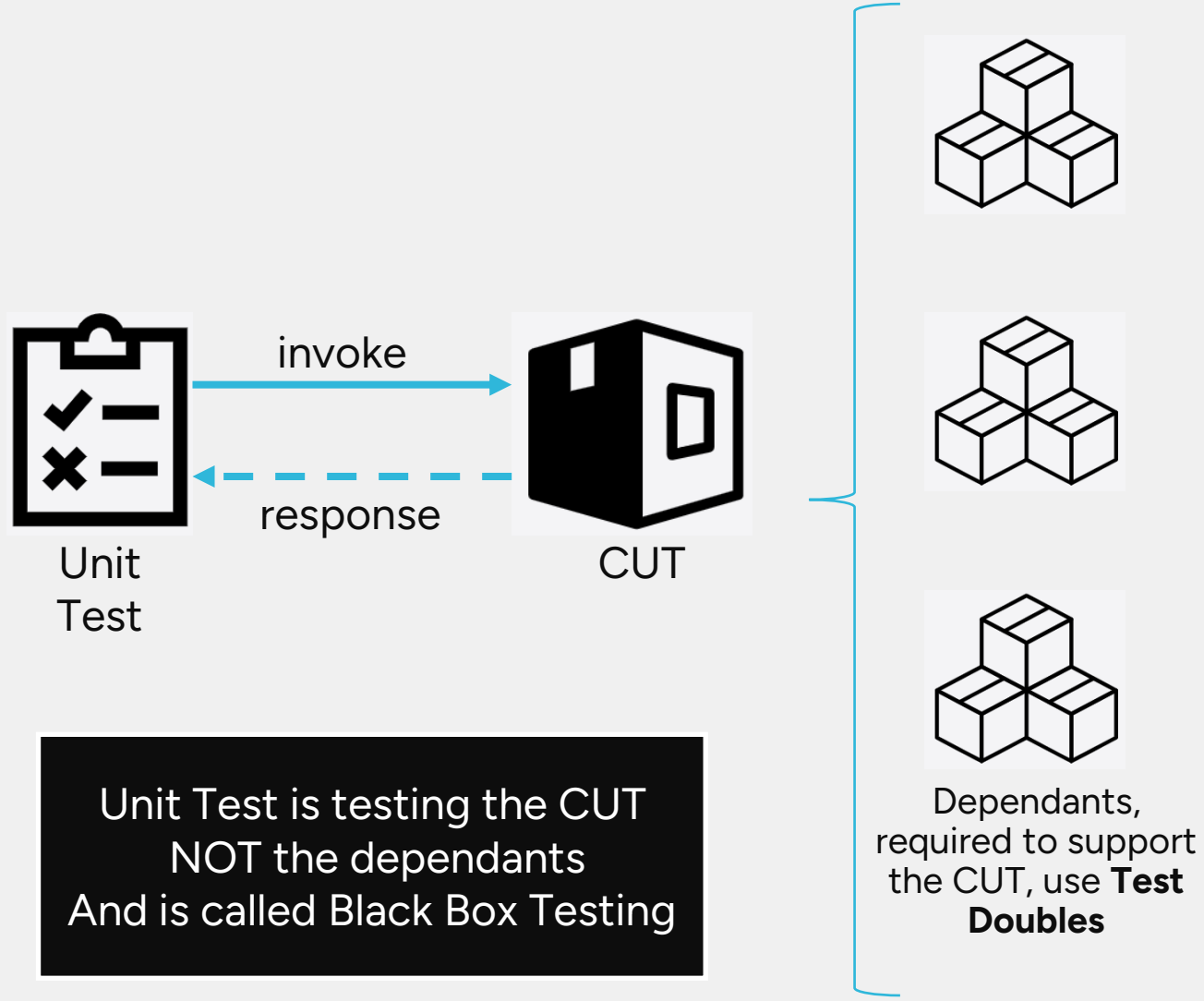
Only write enough implementation code to pass the test

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, which will reduce your test coverage

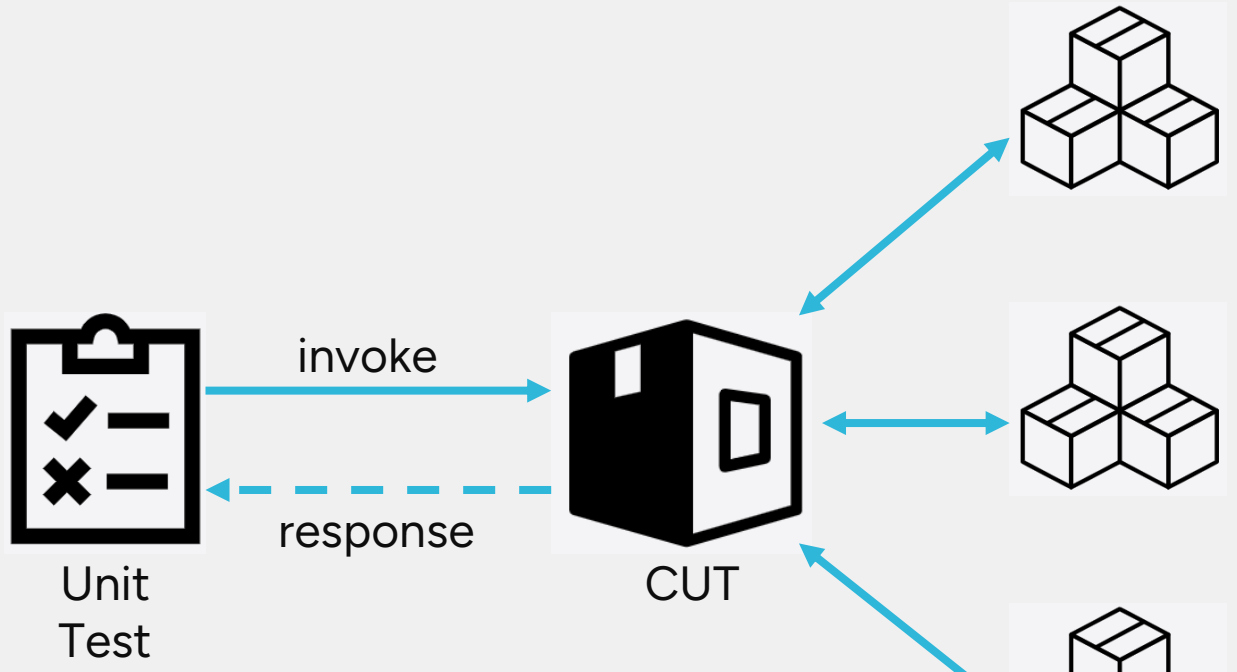
Only write what is needed to maintain good test coverage

Relationship between Unit Test, CUT and its Dependents



Unit Test is testing the CUT
NOT the dependants
And is called Black Box Testing

Relationship between Unit Test, CUT and its Dependents



Unit Test is testing the CUT
NOT the dependants
And it is called White Box
Testing

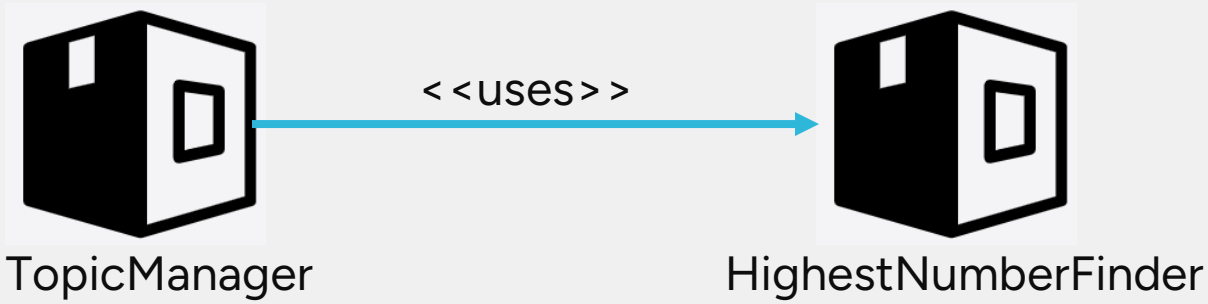
Extended Lab – Find Highest Number

A requirement has come through that the highest number finder must deal with finding the highest number for a series of 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

QLC-2.1 TDD Case Study

Topic Manager

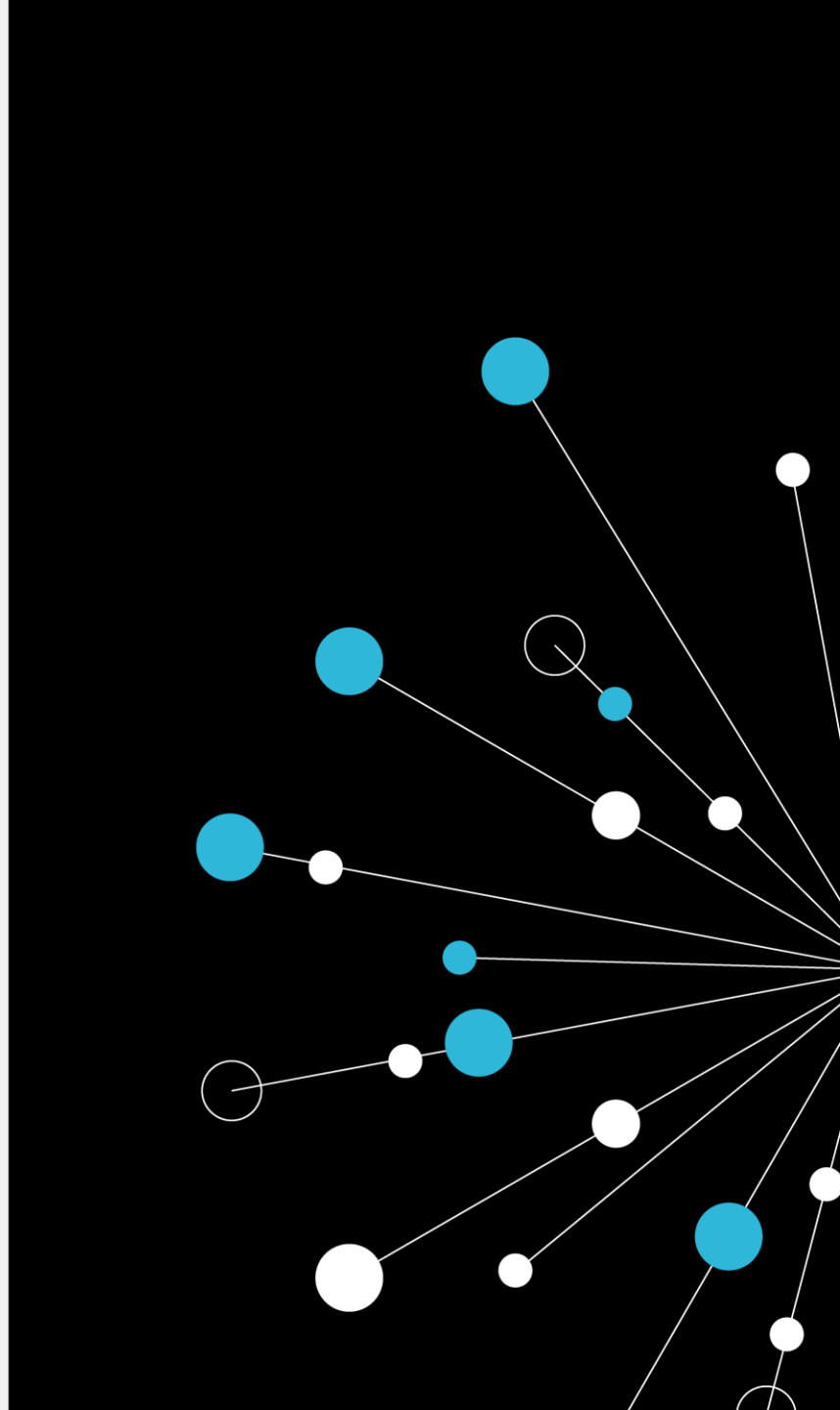
Find the highest score for a series of topics

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.

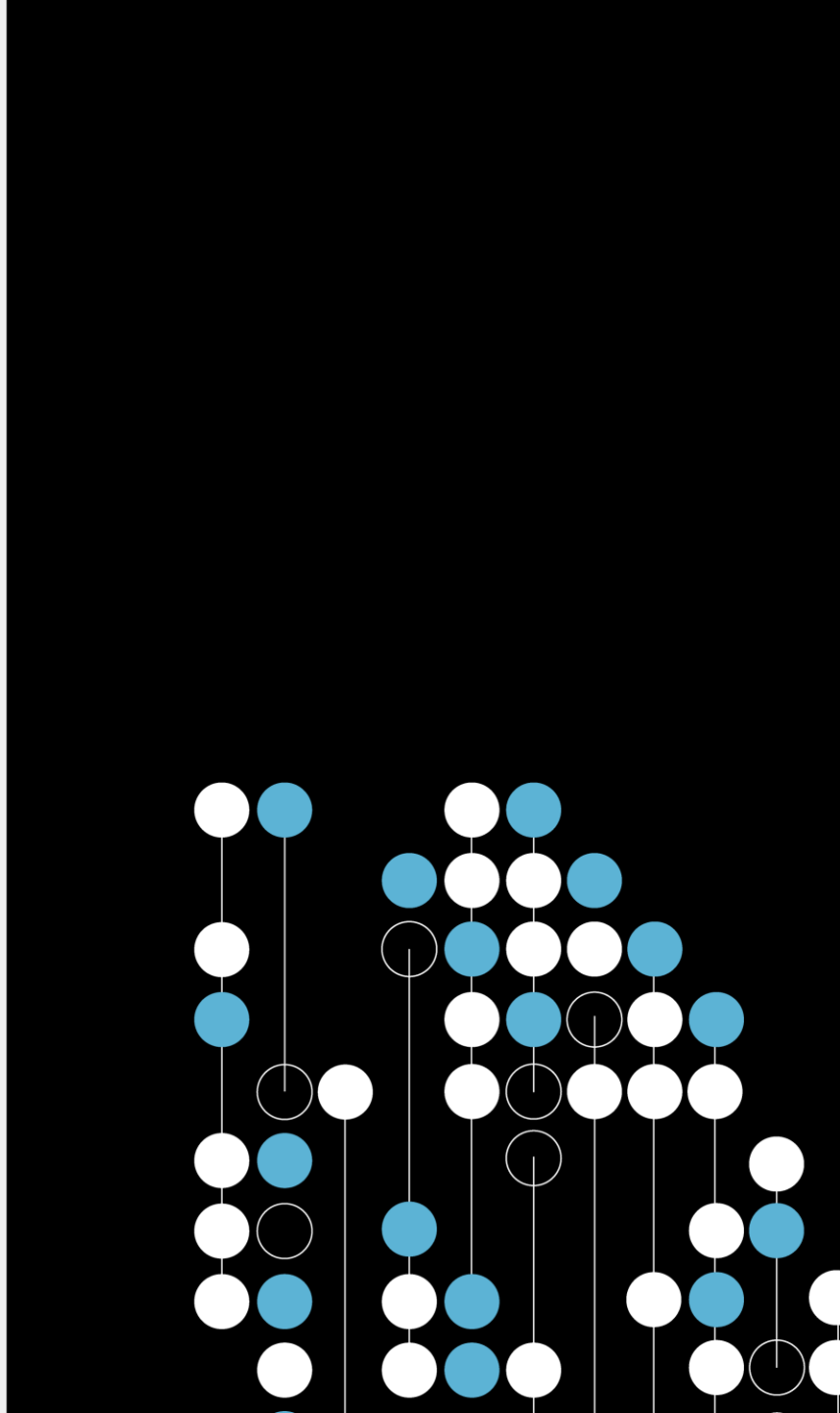
Always start simple, with a test that does not need to handle exceptions.



Dependents Takeaway

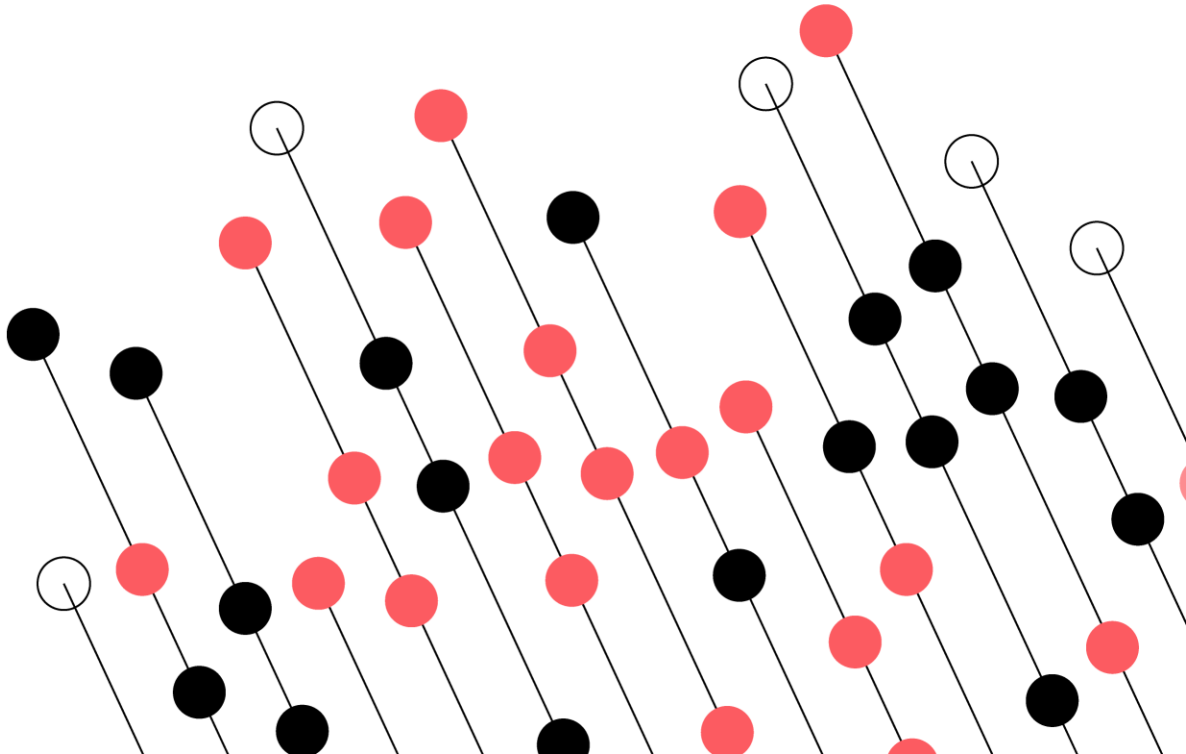
When working with tests, watch out for tightly coupled code, it leads to

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



QA

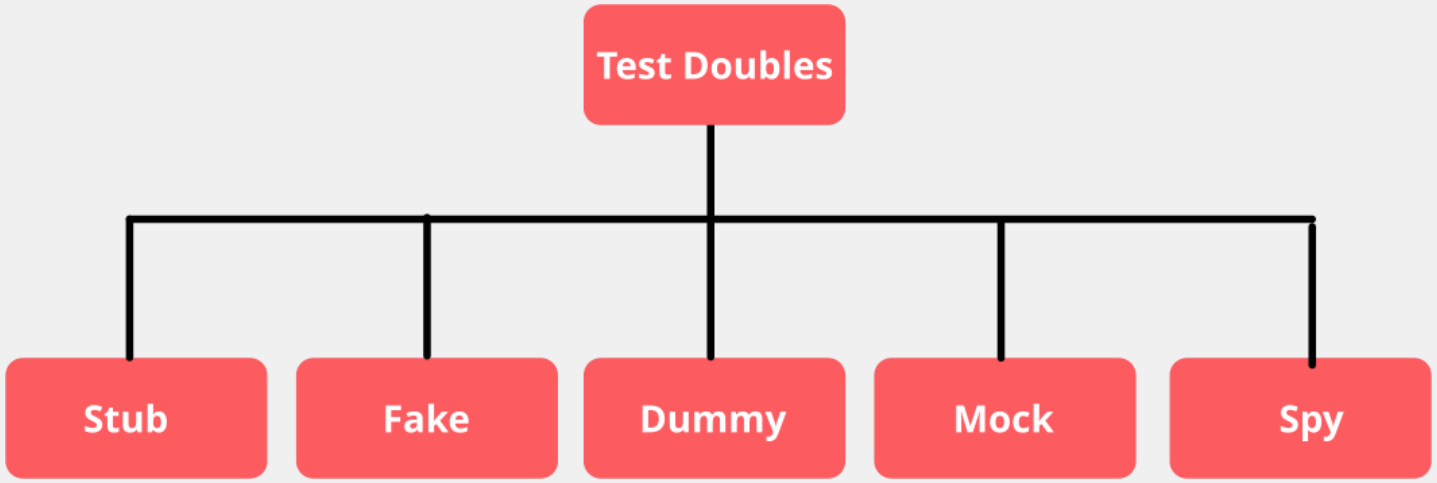
Test Doubles



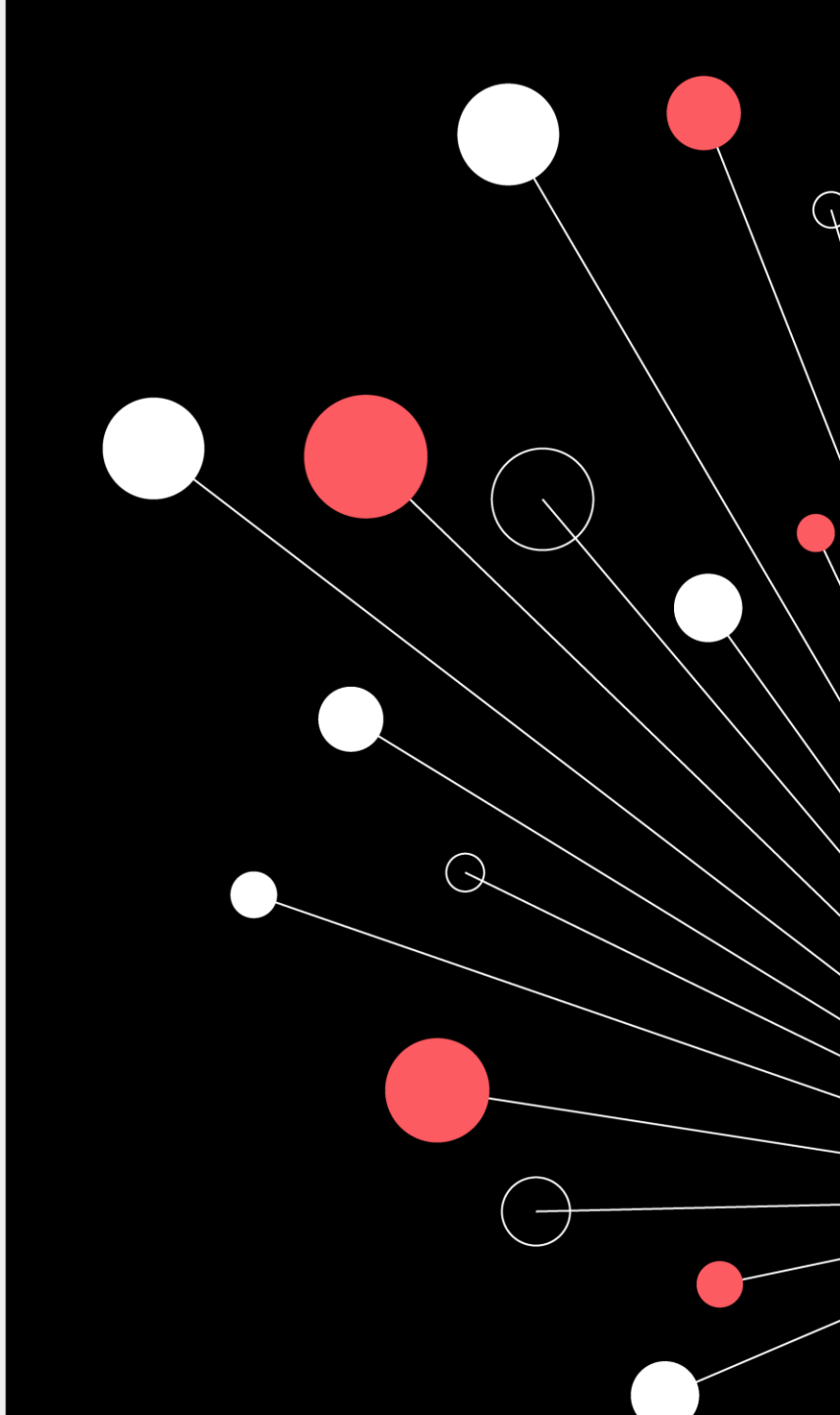
Learning objectives

Test Doubles – replace dependencies

Several Types



Ref:
https://martinfowler.com/bliki/TestDouble.html



QA

Test Doubles – Design Technique

Good 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

QA

Stubs – The Problem

Tight coupling to file I/O

- load_file() reads directly from disk, making tests slow, fragile, and environment-dependent.

Difficult to control test inputs

- lines is populated internally, you can't inject canned data to test calculate_file_size() independently.

Cannot easily isolate logic

- File reading and size calculation are bundled.

Not easily mockable

- No parameters passed to _calculate_file_size() - relies on internal state (self.lines) set by file operation.

Violates Single Responsibility Principle

- One method responsible for IO and computing size

```
def __init__(self, file_to_load):
    self.file_to_load = file_to_load
    self.lines = []

def load_file(self, fname):
    try:
        with open(fname, encoding='utf-8') as f:
    except IOError:
        self.lines = []
    return self._calculate_file_size()

def _calculate_file_size(self):
    total_length = sum(len(line) for line in self.lines
    return total_length
```

Stubs – The Solution

```
class FileLoaderStub:
    def __init__(self):
        self.file_data = “”

    def load_data(self, fname):
        self.file_data = “Some random data”
        return self._calculate_file_size()

    def _calculate_file_size(self):
        return len(self.file_data)
```

No dependency on the file system

- load_data() does not touch disk

Test becomes repeatable and fast

- no I/O delays or failures

Focus on logic

- calculate_file_size() is tested in isolation

Easily controllable input

- file_data can be replaced with any string

Simplifies testing

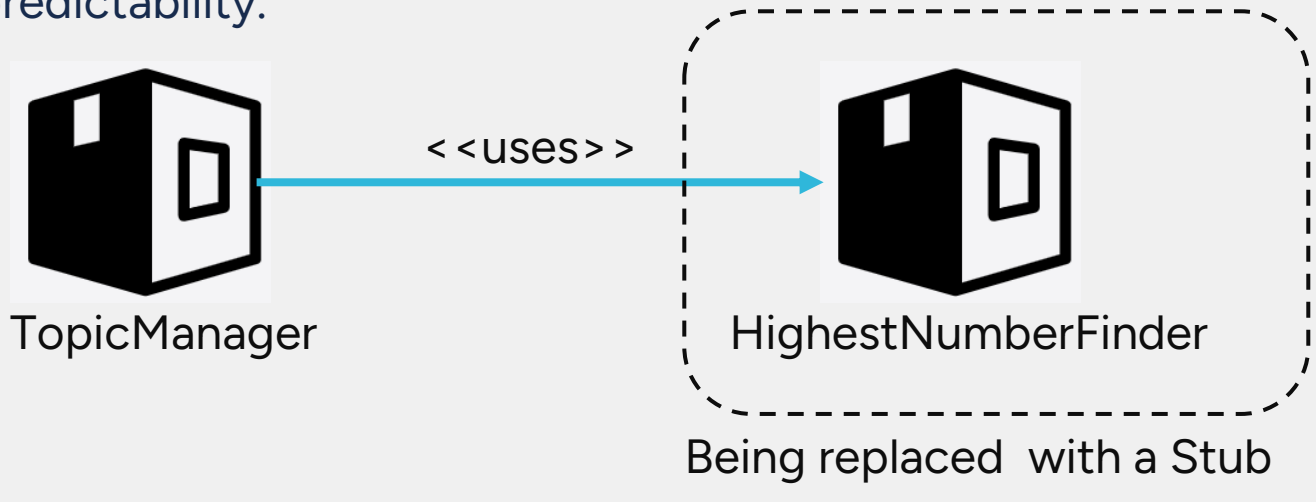
- avoids the need to create actual files for the test

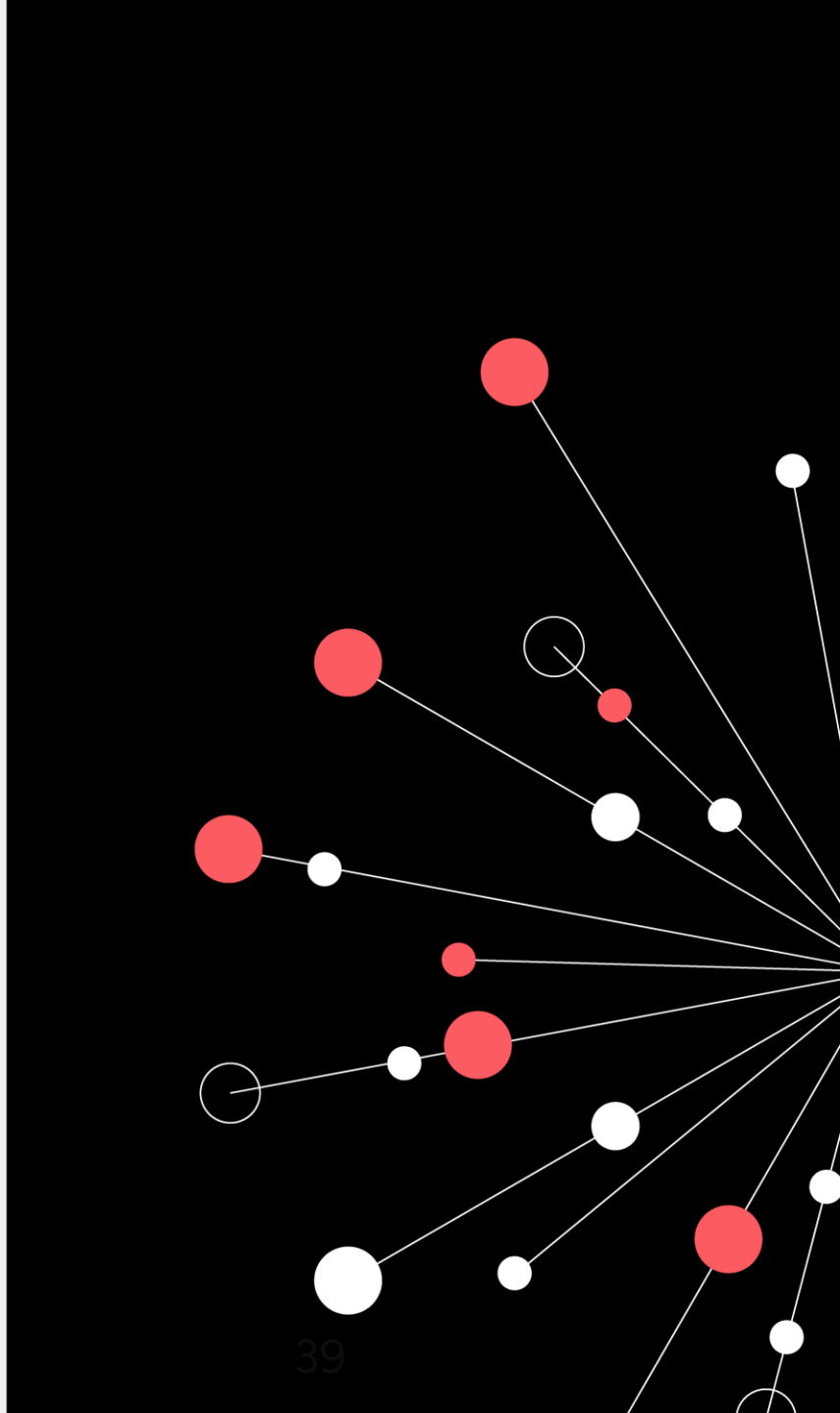
QuickLab – TDD Case Study

QLC 2.2 – TopicManager

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.



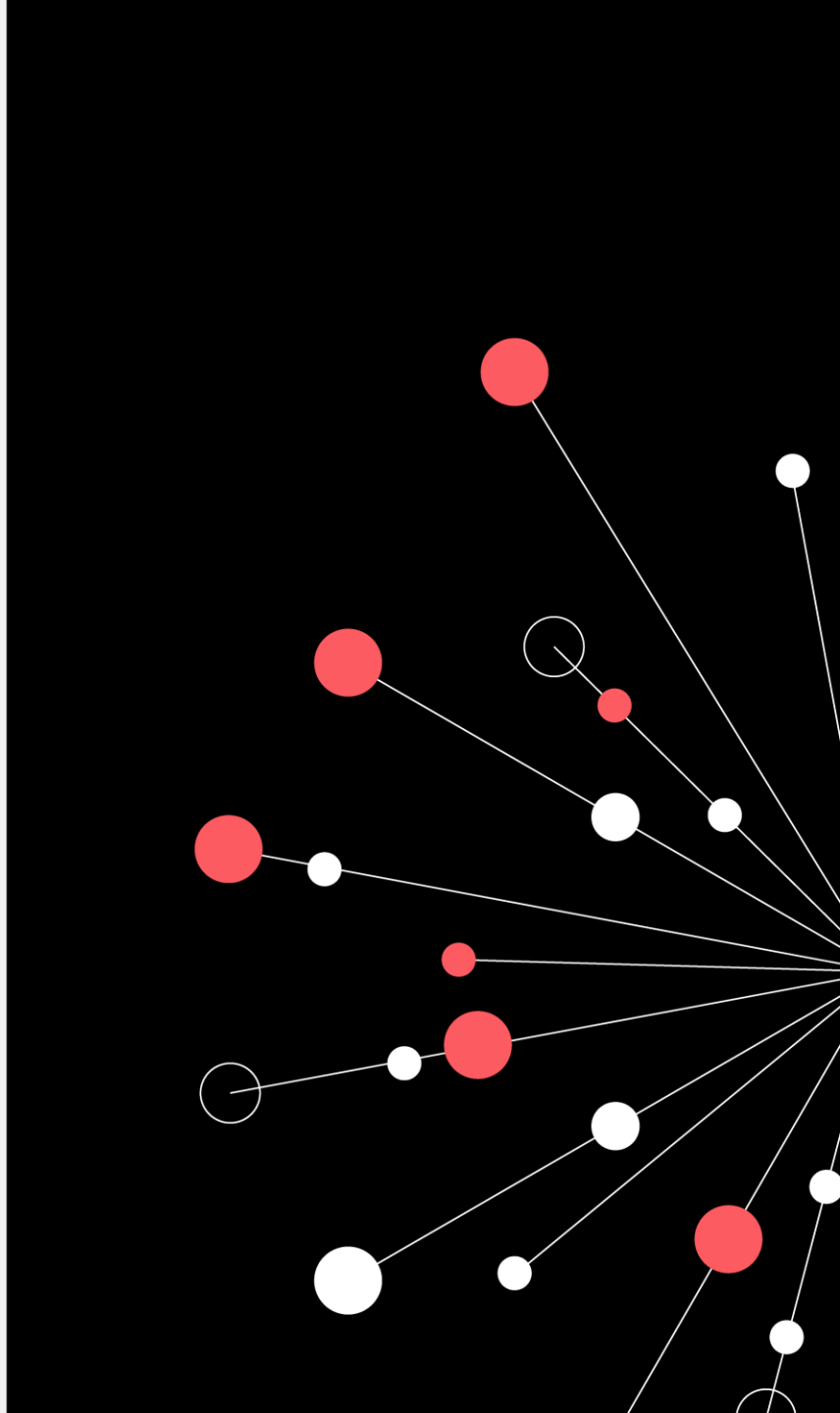


TDD Takeaway

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



QA

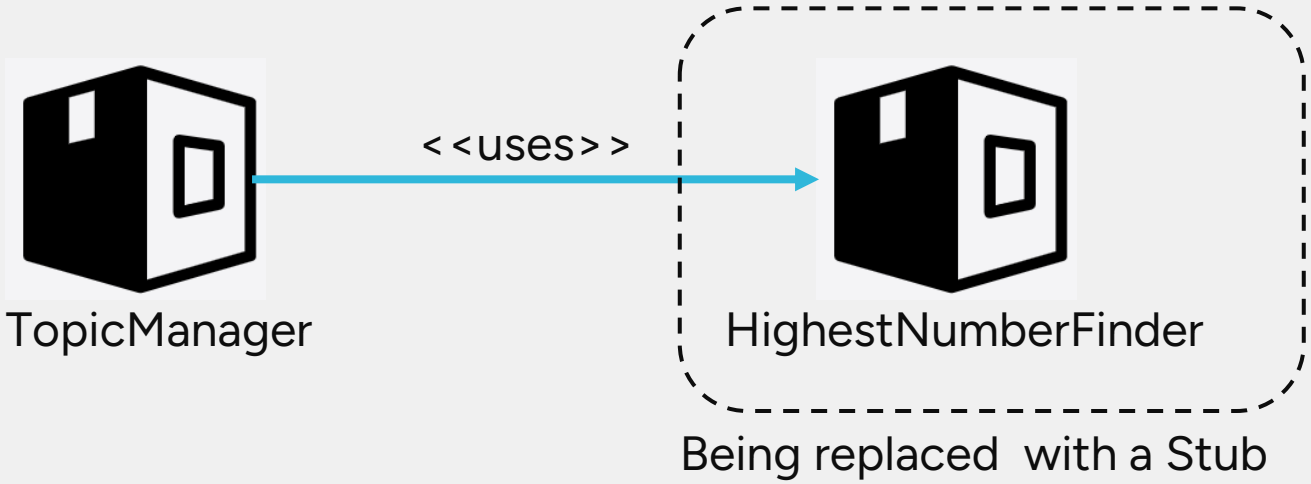
QuickLab – TDD Case Study

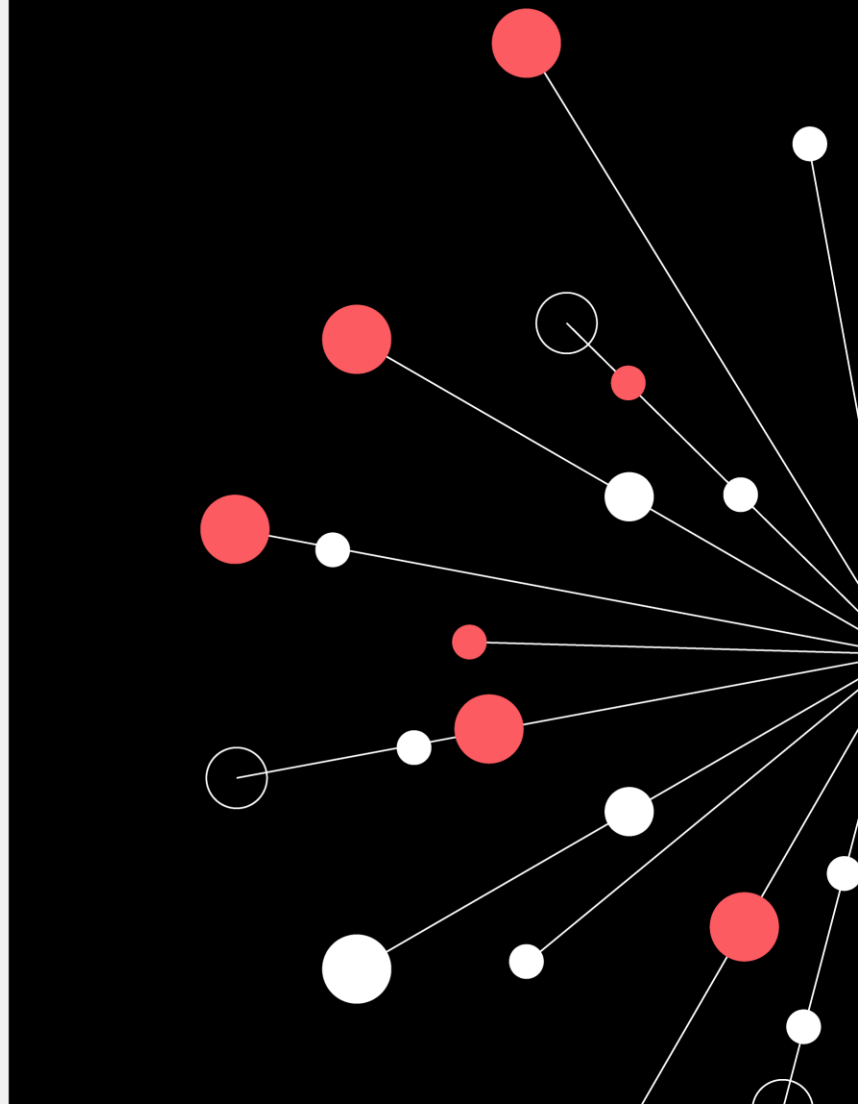
QLC 2.3 – TopicManager

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?



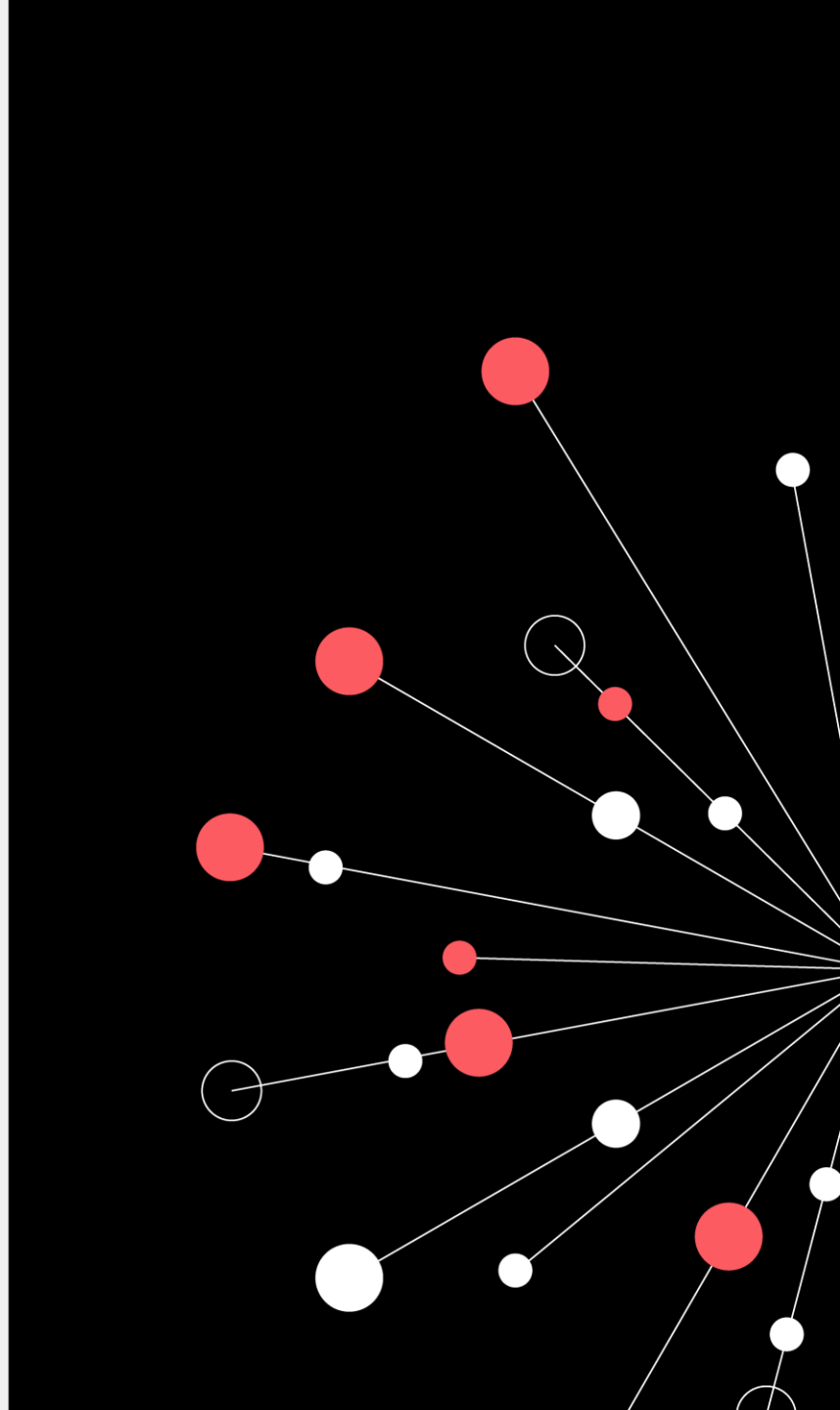


Stubs Takeaway

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

Stubs have their **limitations!**

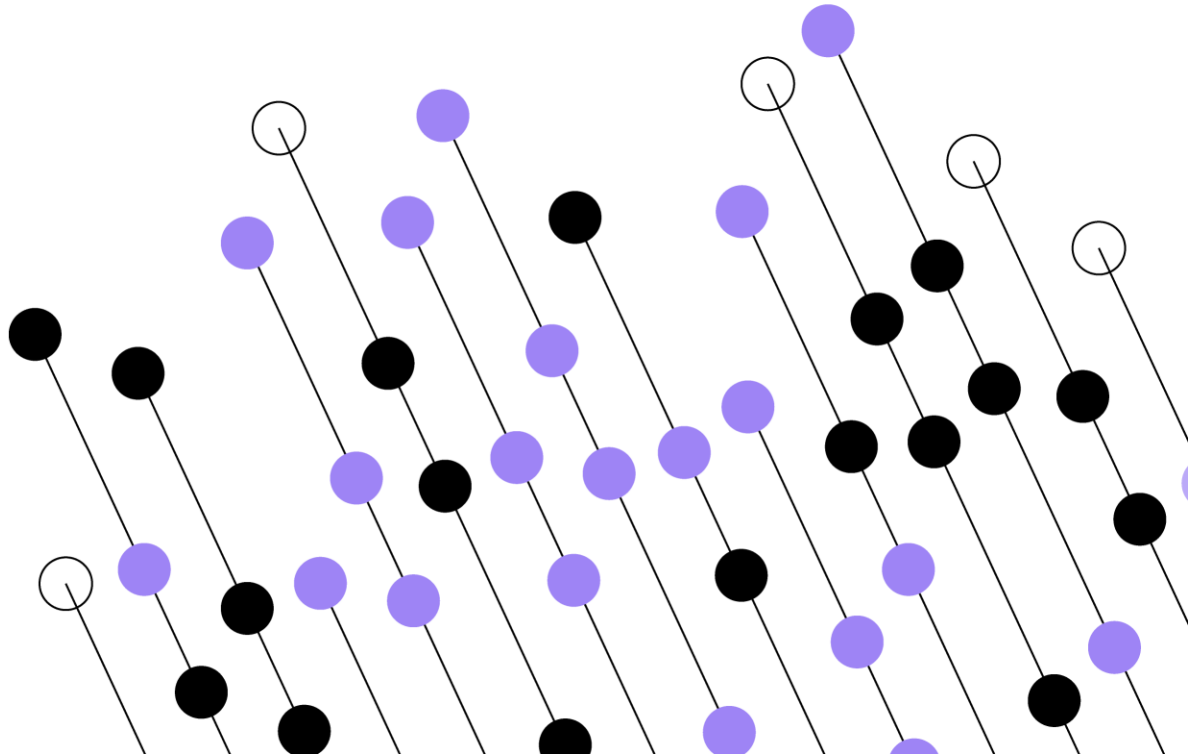
You might end up writing a significant amount of code to create a stub test double, which can introduce errors into your tests - as we've just seen.



QA

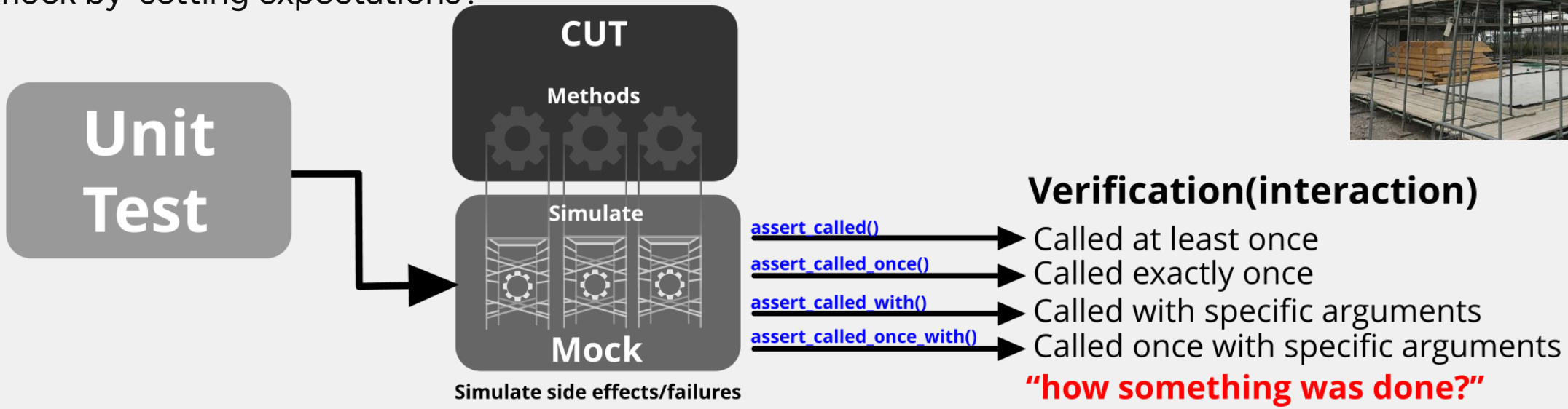
Test Doubles and Verification Testing

Mocks



Mock Objects

A **mock object** is a strongly-typed **test double** that **mimics** the **interface** of the real object it replaces. By default, its methods do nothing and return the default value for their return type. To simulate specific behaviour, you must explicitly configure the mock by 'setting expectations'.



Mocks provide the 'scaffolding' but no real structure – they DO NOT run the real code unless explicitly configured to do so. Structure is added through the use of expectations

QA

QuickLab QL-3 Code Demo

Mocks are NOT stubs!

QuickLab – TDD Case Study

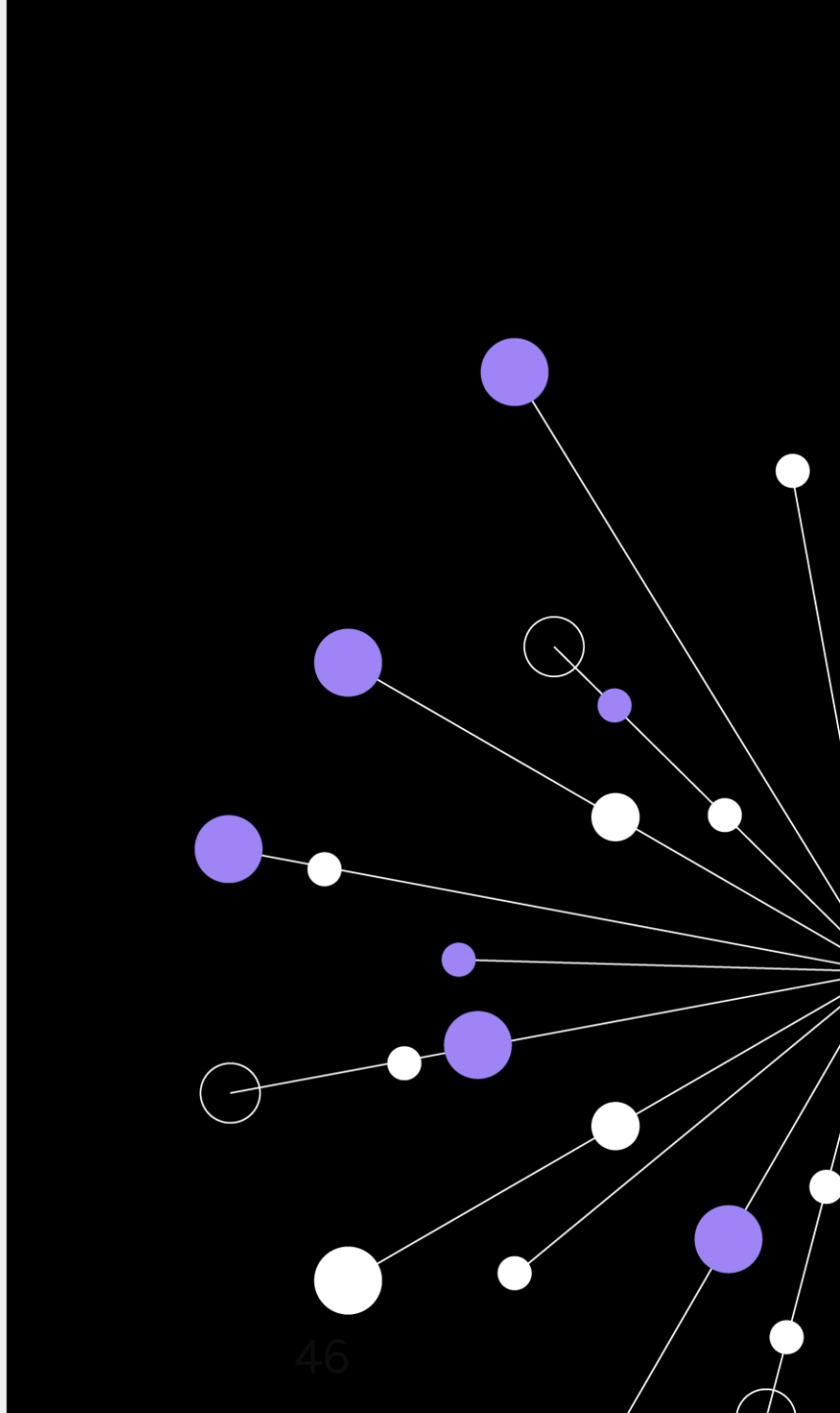
QLC 2.4 – TopicManager

Continue working with **Mocks**

Continuing in a TDD manner, complete the last 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]]

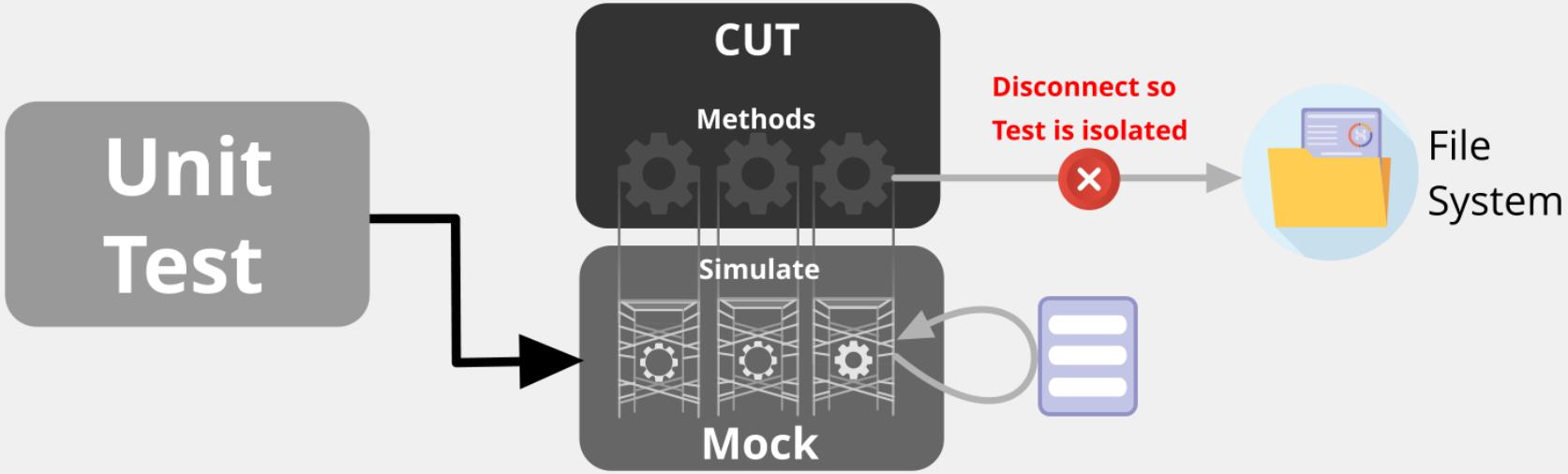
Use a Mock in place of the production HighestNumberFinder class



Working with Mocks

We are going to design and refactor our 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.



QA

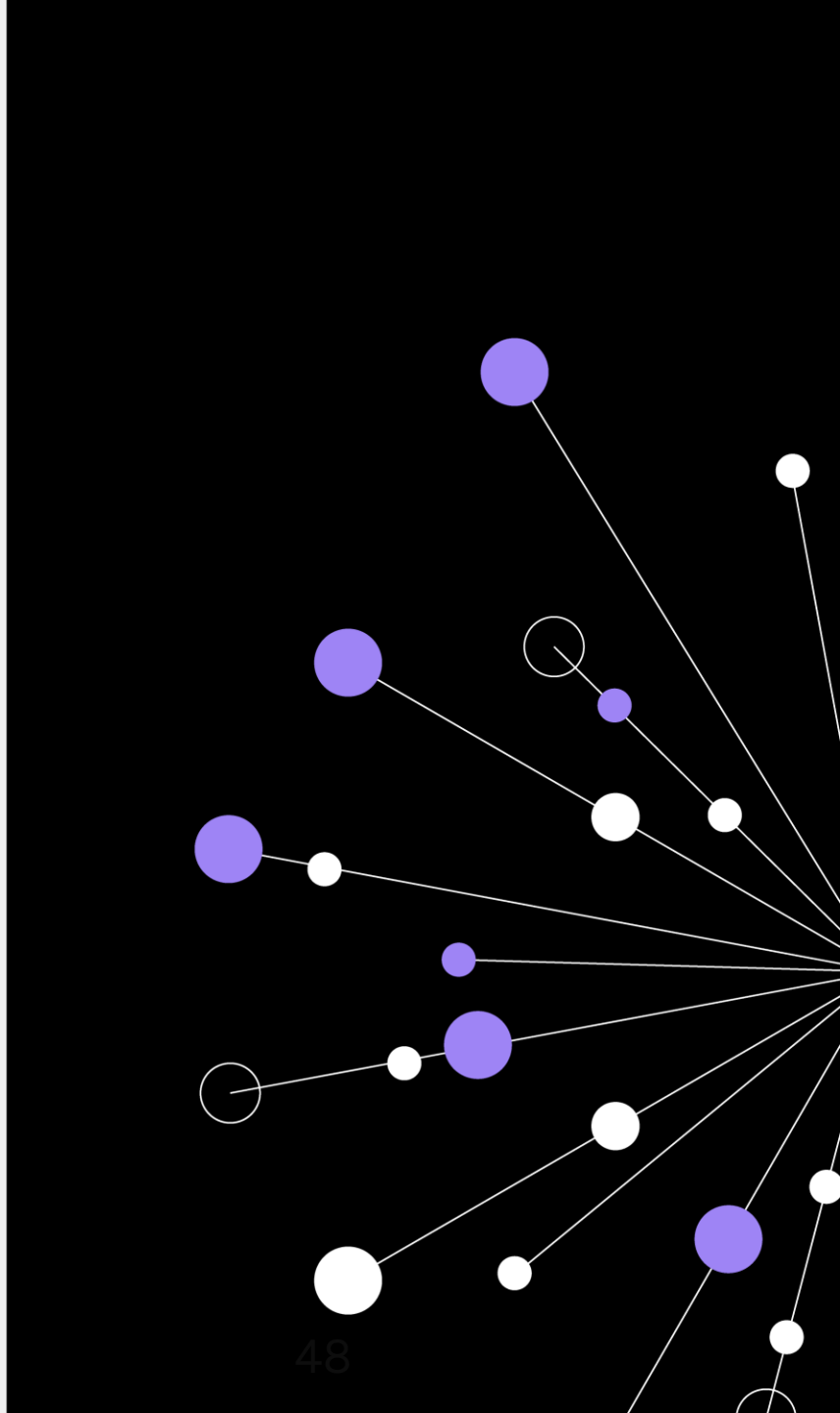
QuickLabs – TDD Demo

QL 4.1 – FileLoader

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

```
def load_file(self, fname: str) -> int:
    """
    Loads the file specified by fname and returns the
    total number of characters.
    """
```

Question: What's the weakness in this design?



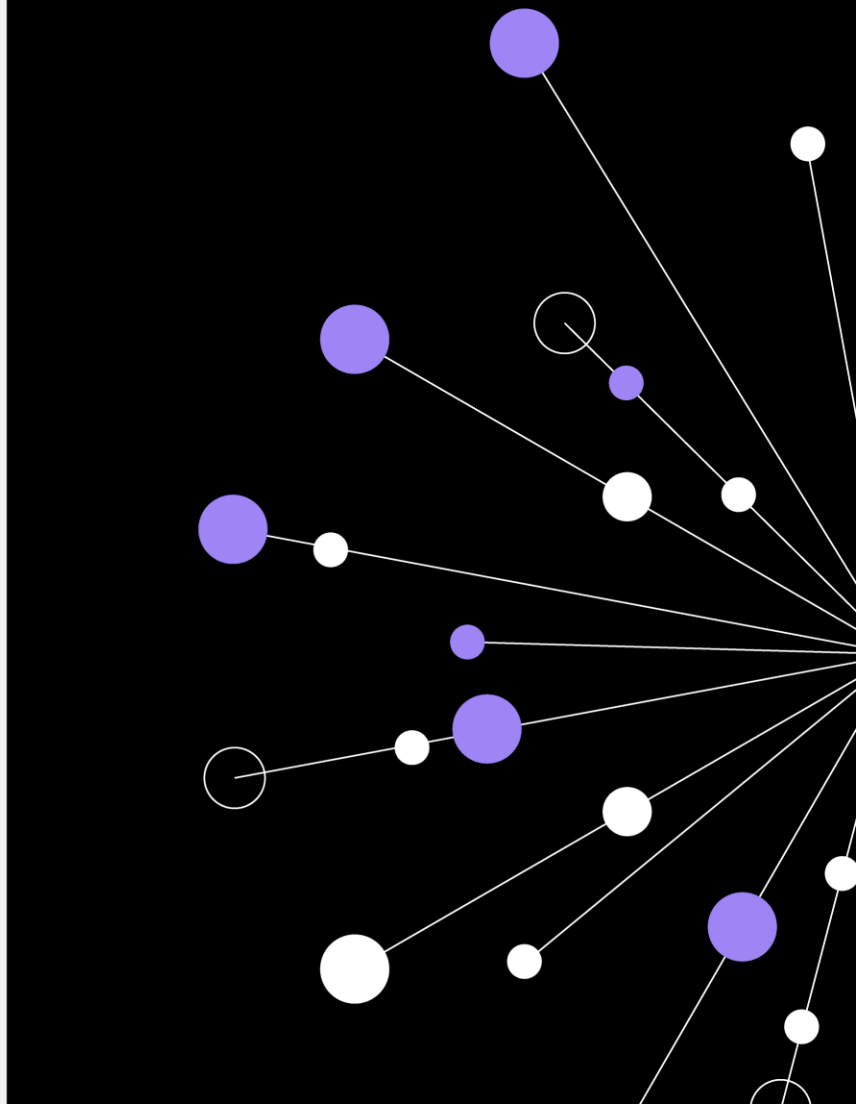
QuickLabs – TDD Demo

QL 4.2 – Using Lambdas

Create a new unit test that passes a **lambda** expression to the **load_file()** method in the class under test (CUT).

Add a new method to the FileLoader class that accepts a callable (e.g., a lambda function) to handle file loading logic.

This allows you to inject a stub function for reading files, improving flexibility and testability.



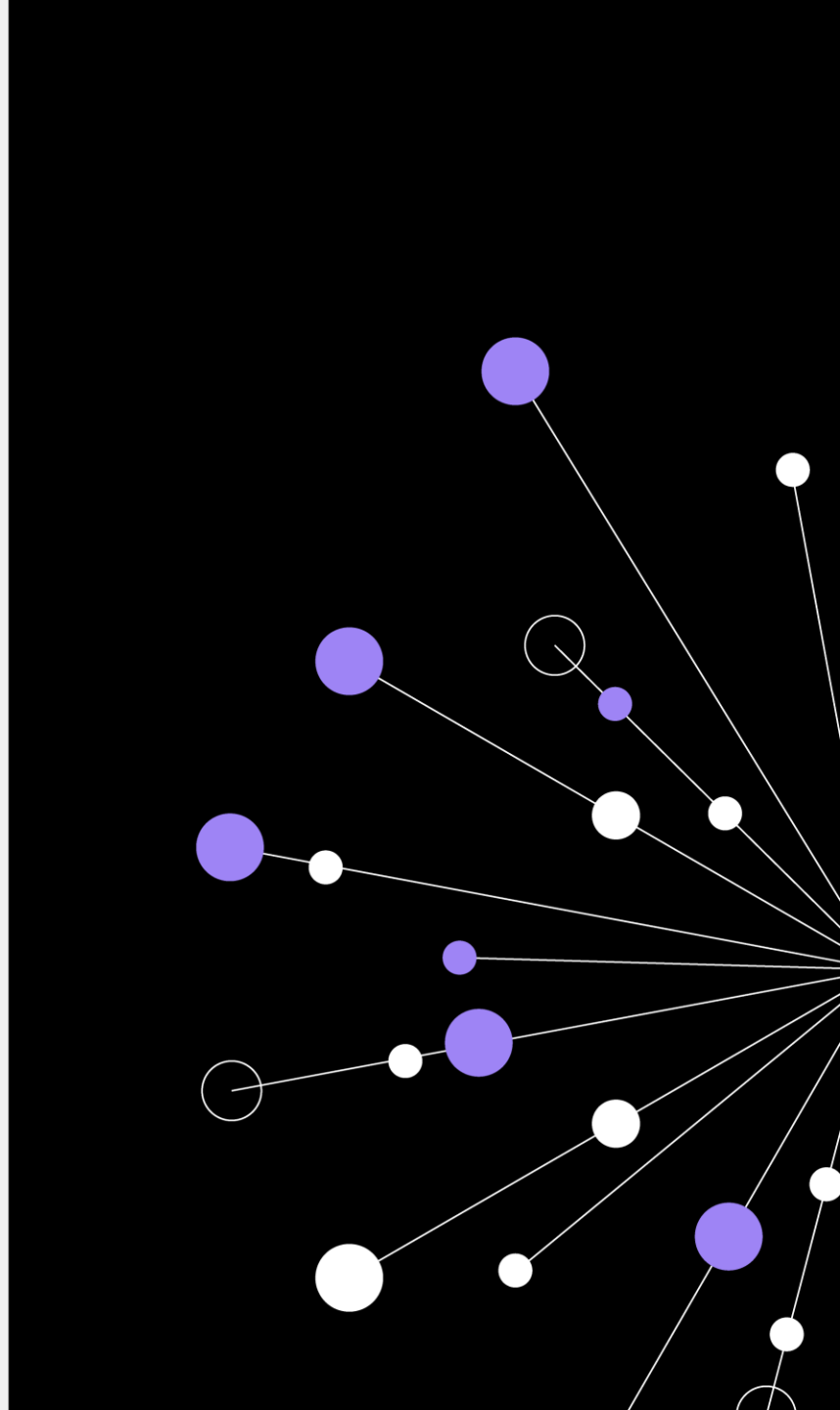
QA

QuickLabs – TDD Demo

QL 4.3 – Using Stubs

Add a new test that demonstrates passing a lambda to the loadFile() method on the class under test (CUT).

Within the lambda, substitute actual file reading with a list of dummy data — each item in the list should represent a line from a file.



QA

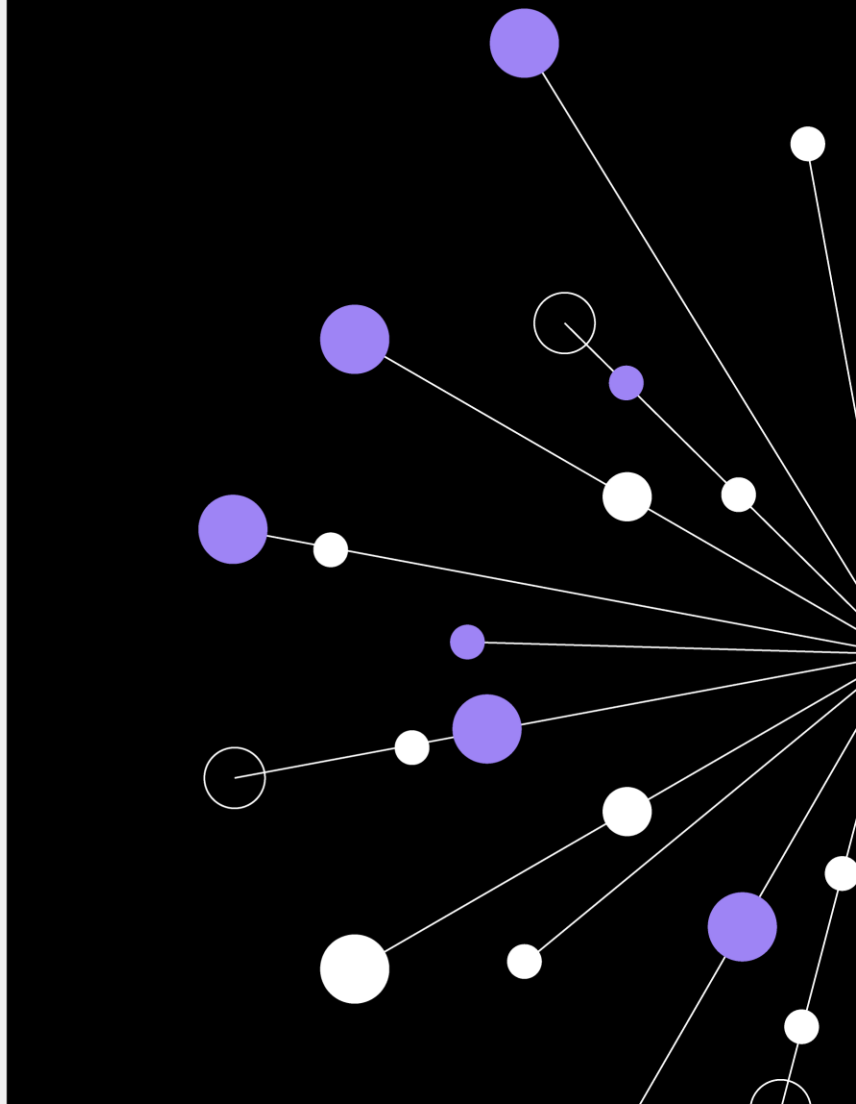
QuickLabs – TDD Demo

QL 4.4 – Replacing the Stub with a Mock

Wrap Python's built-in open().readlines() in a class method, mimicking the behavior of File.ReadLines().

Use unittest.mock to mock your wrapper class so that its read_lines() method returns dummy data.

Pass a lambda to load_file_with_func() that calls your mocked wrapper's read_lines() method.



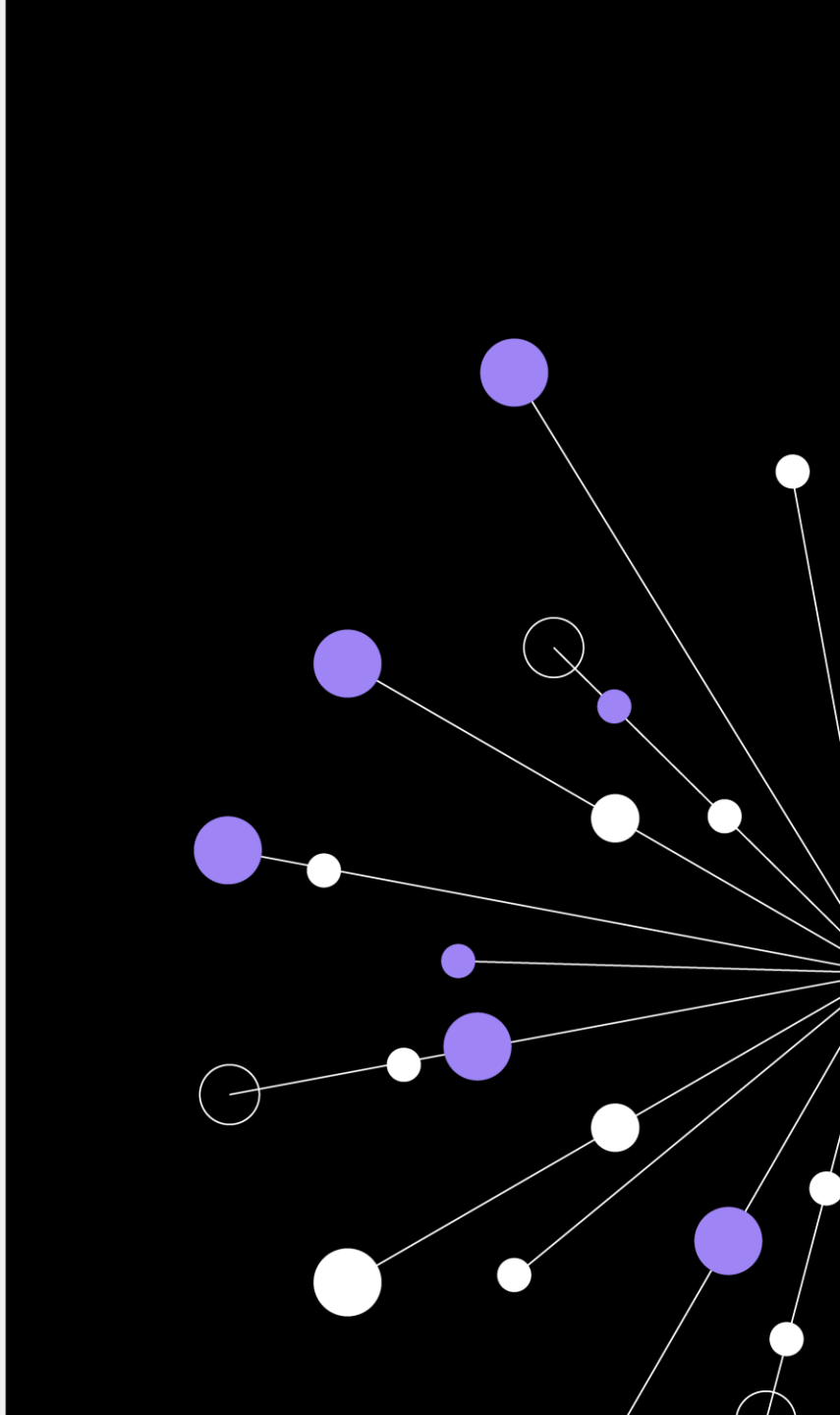
QA

QuickLab – TDD Case Study

QLC 3 – TopicScoreWriter

Working with Test Doubles and Mocks

See Lab Exercise.



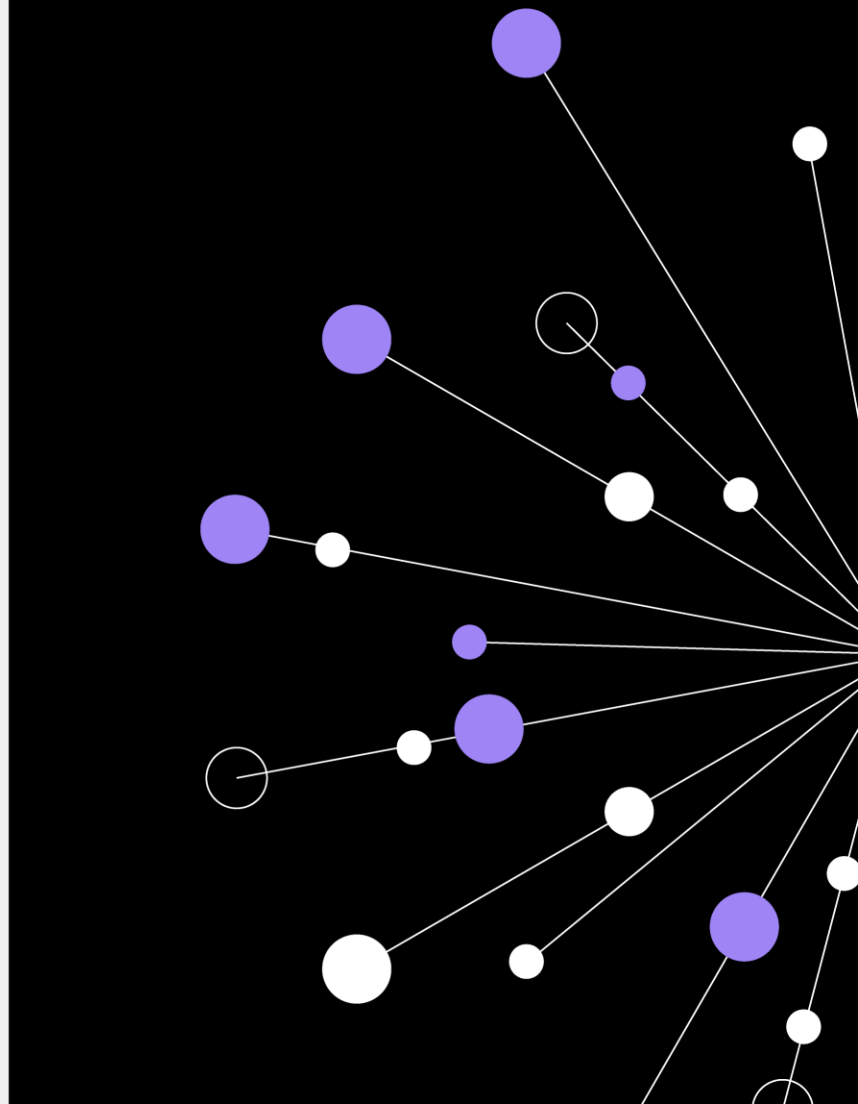
Mocking Objects Takeaway

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



QA

Spy Object

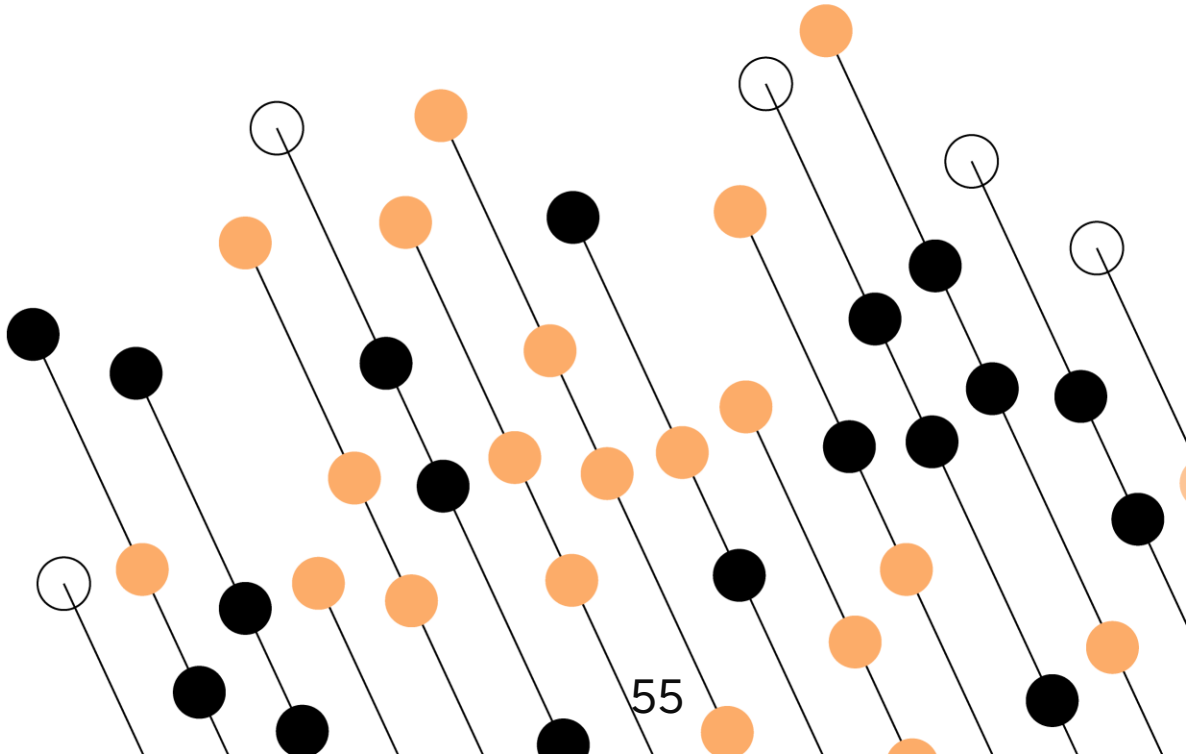
A spy is a strongly-typed object that **replicates** the **interface** of the **real object** it replaces.

By default, its methods delegate calls to the real object, allowing the actual behaviour to run.

However, if specific expectations or overrides are set on the spy, those will take precedence over the real method

While a mock is just scaffolding with nothing inside, a spy is like scaffolding around a real house—you can still access and use the actual structure unless you cover it with something new.

Untestable Code



What is Untestable Code?

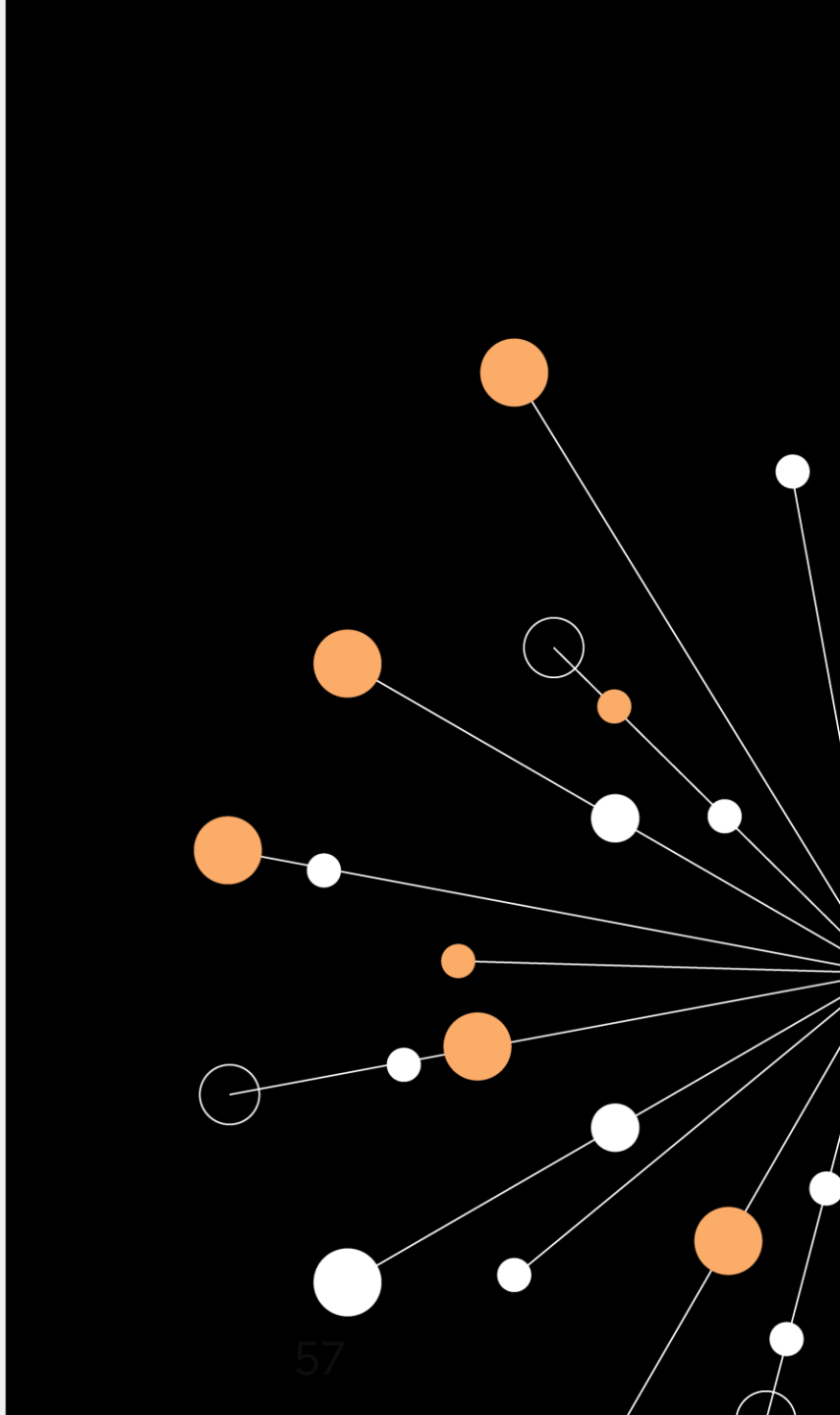
Code that is difficult to test often has one or more of the following characteristics:

- **Hard to isolate:** relies on I/O, networking, or external databases.
- **Opaque state:** it's difficult to verify the internal state after method calls, even when inputs are known.
- **Inaccessible logic paths:** certain paths are only triggered under specific conditions, such as time-based constraints.
- **Tightly coupled:** depends heavily on other components that require complex setup or configuration.
- **Legacy dependencies:** interacts with code you can’t modify or don’t have source access to but still need to rely on.

QA

QuickLab – TDD Case Study

QLC 4 – There is no Lab

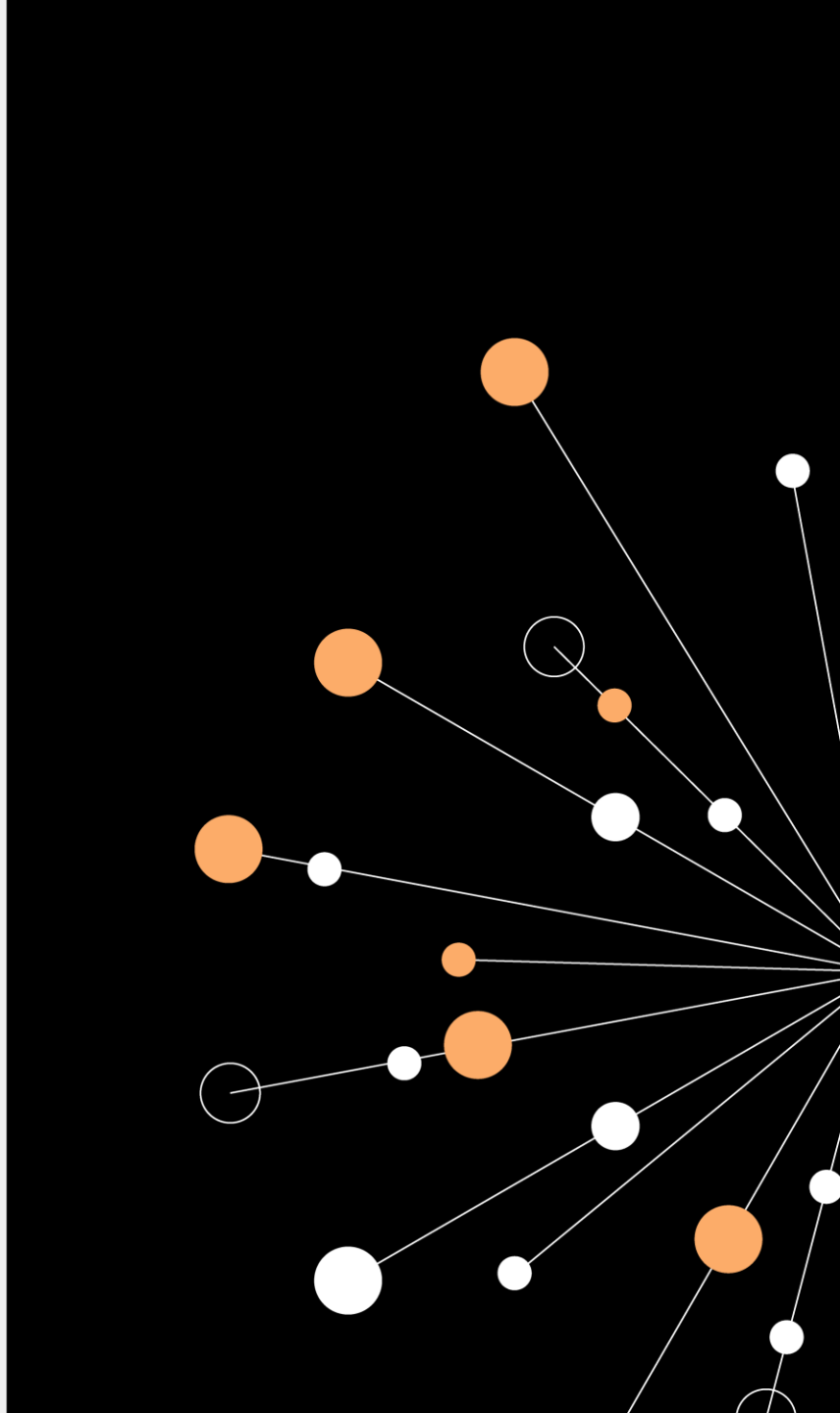


Conclusion

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



QA