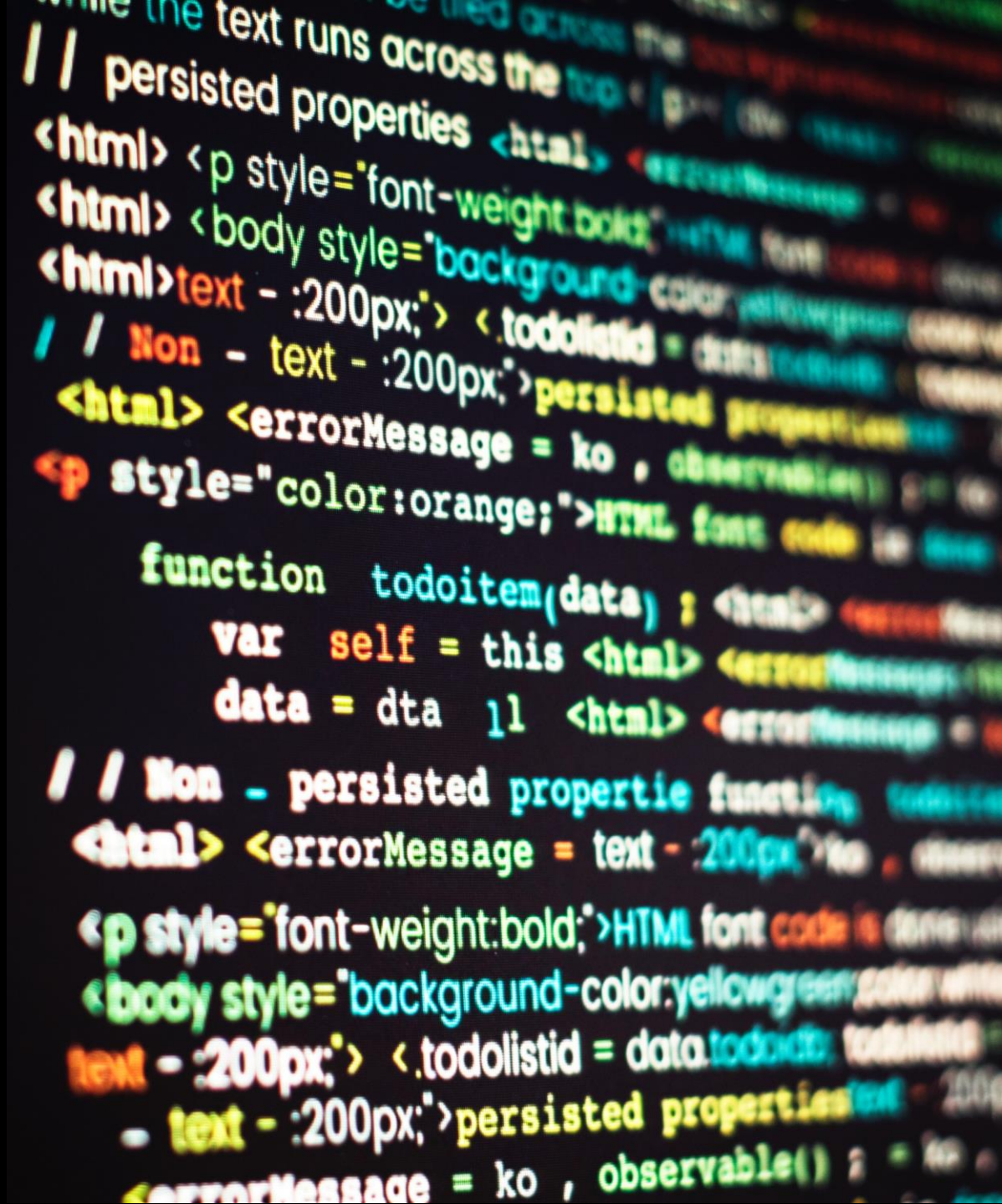
# Test-Driven Development

GoLang

QA

Learn. To Change.

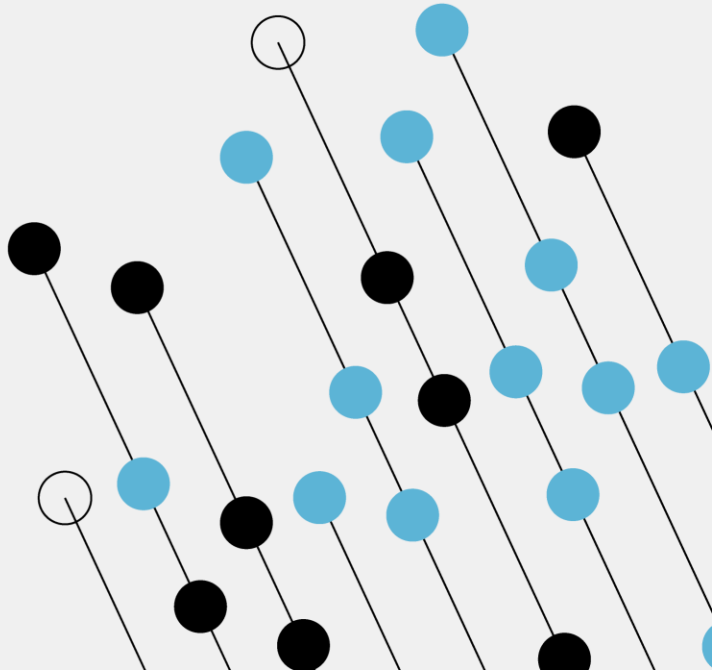


# Course Slides

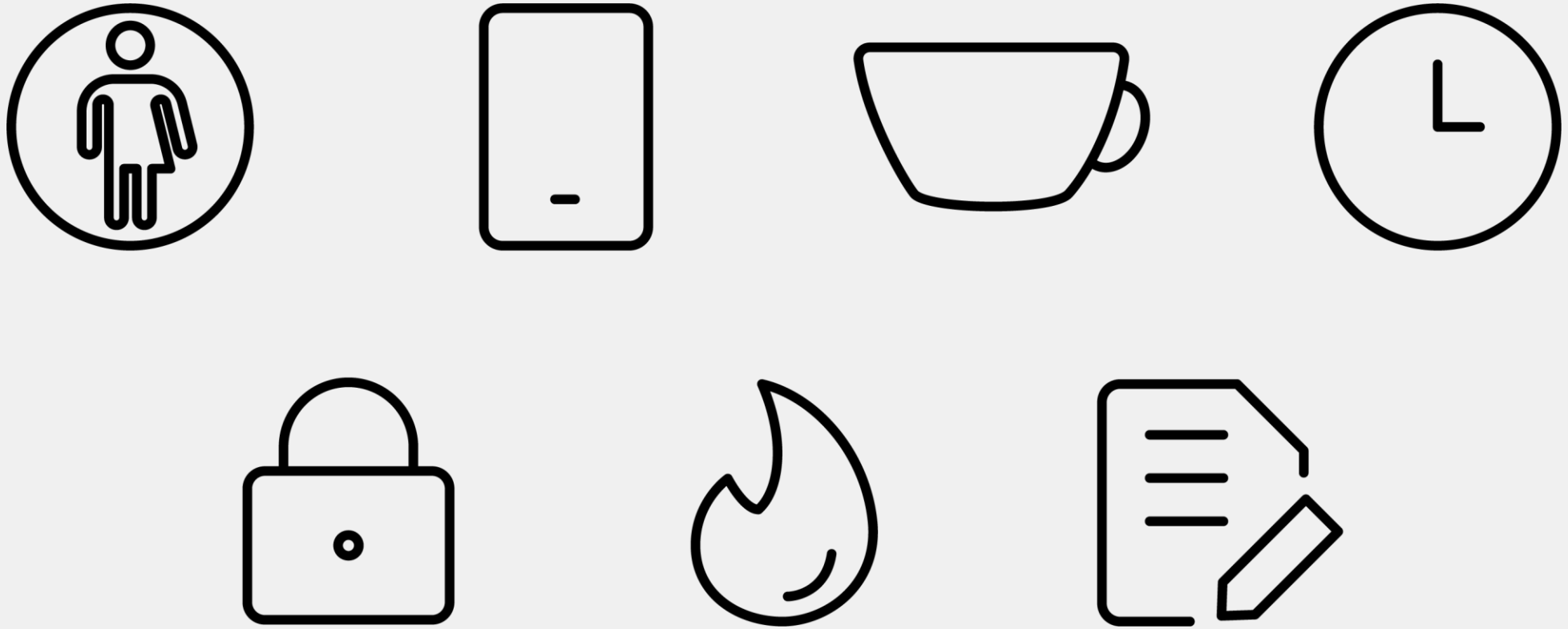
TDD – Test Driven Development

QA

Learn. To Change.



# Housekeeping



# Course Delivery


Hear and forget

See and remember

Do and understand

# Your Training Experience

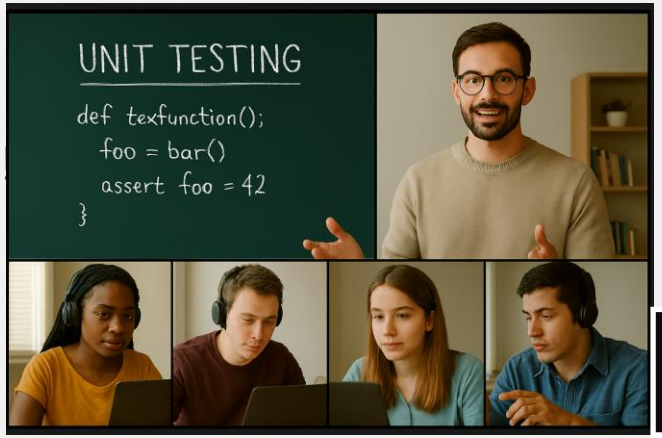
A course should be:

- 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!

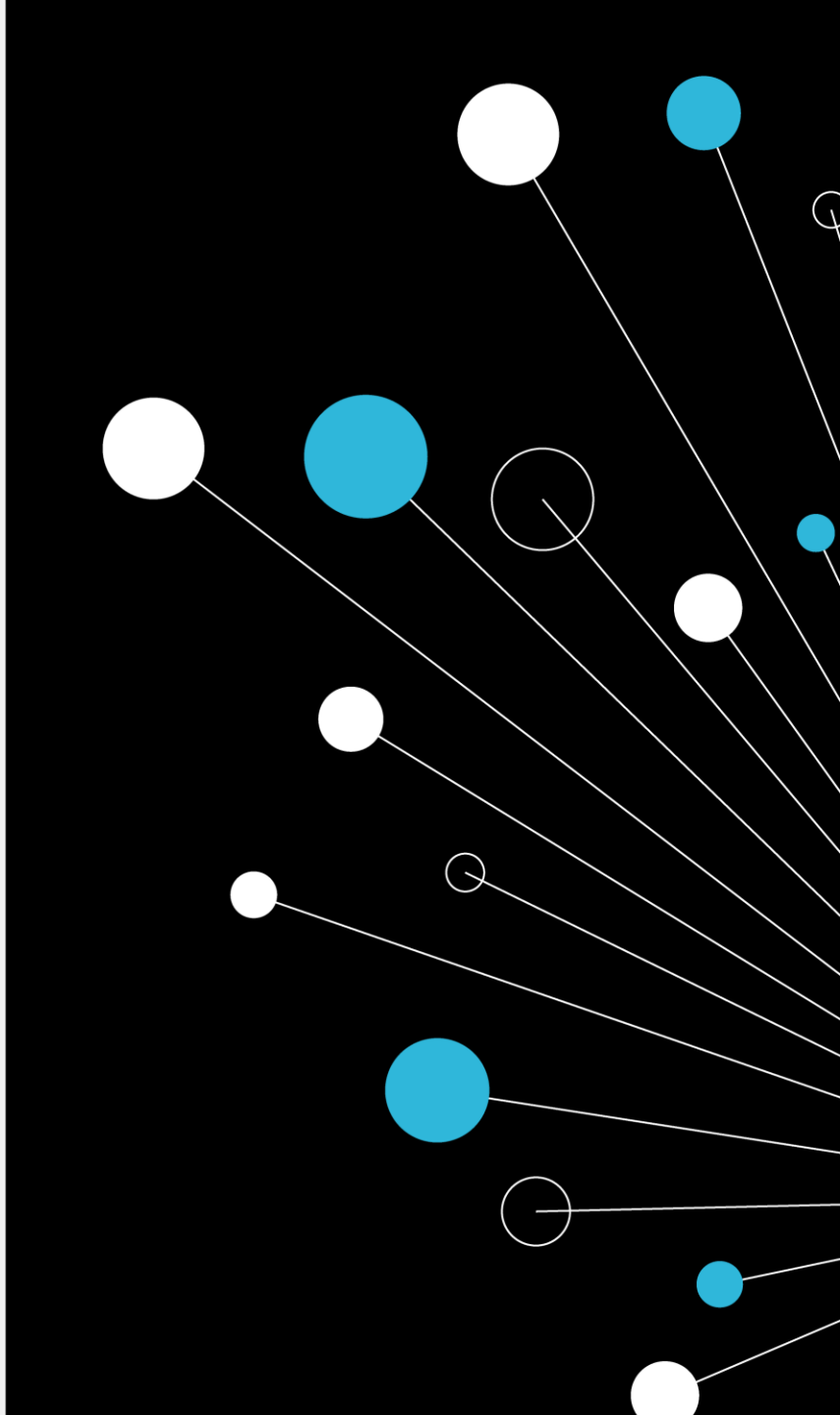



Keep Webcams ON

# 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 GoLang
- 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



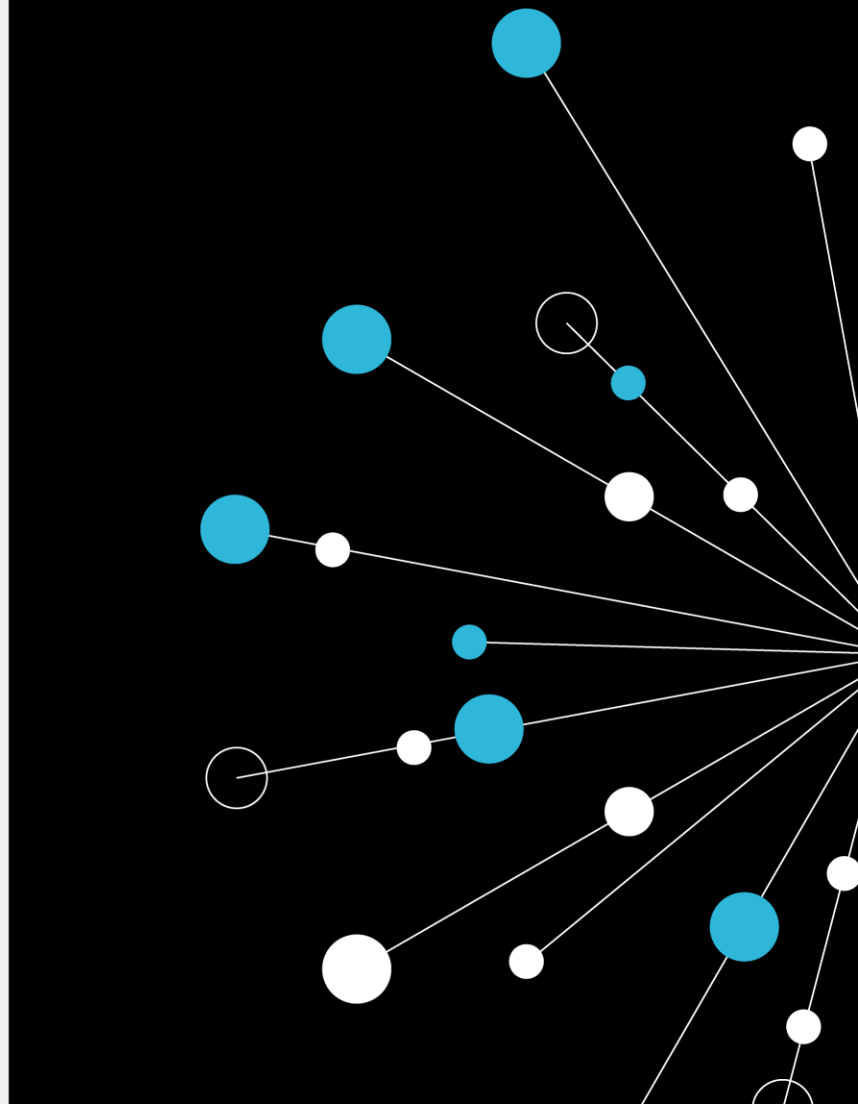
QA

# Pre-requisites

This course assumes the following prerequisites:

- You have a basic knowledge of GoLang
- You have a basic knowledge of OOP
- You have used an IDE like VSCode

If there are any issues, please tell your instructor now



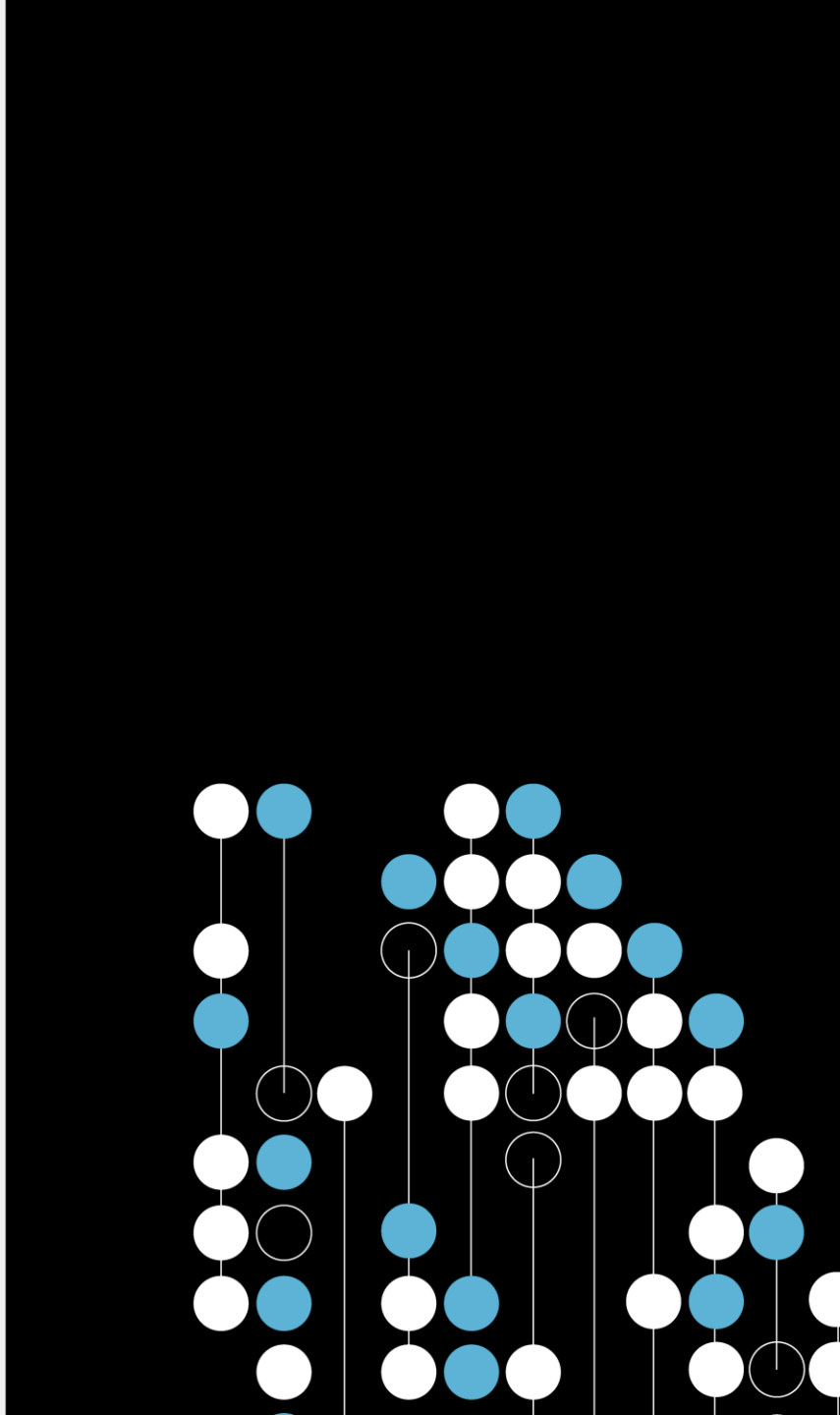
QA

# Introductions

Please say a few words about yourself:

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

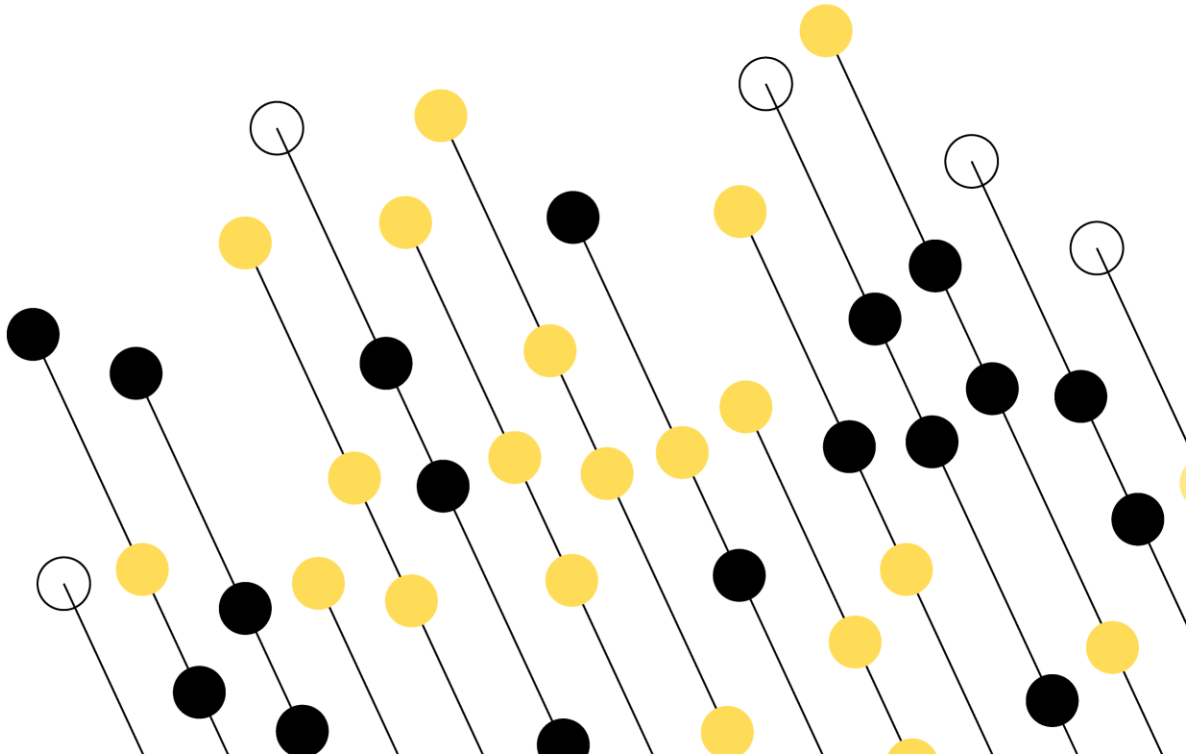
What is your main objective for attending the course?



QA

# TDD

Test Driven Development



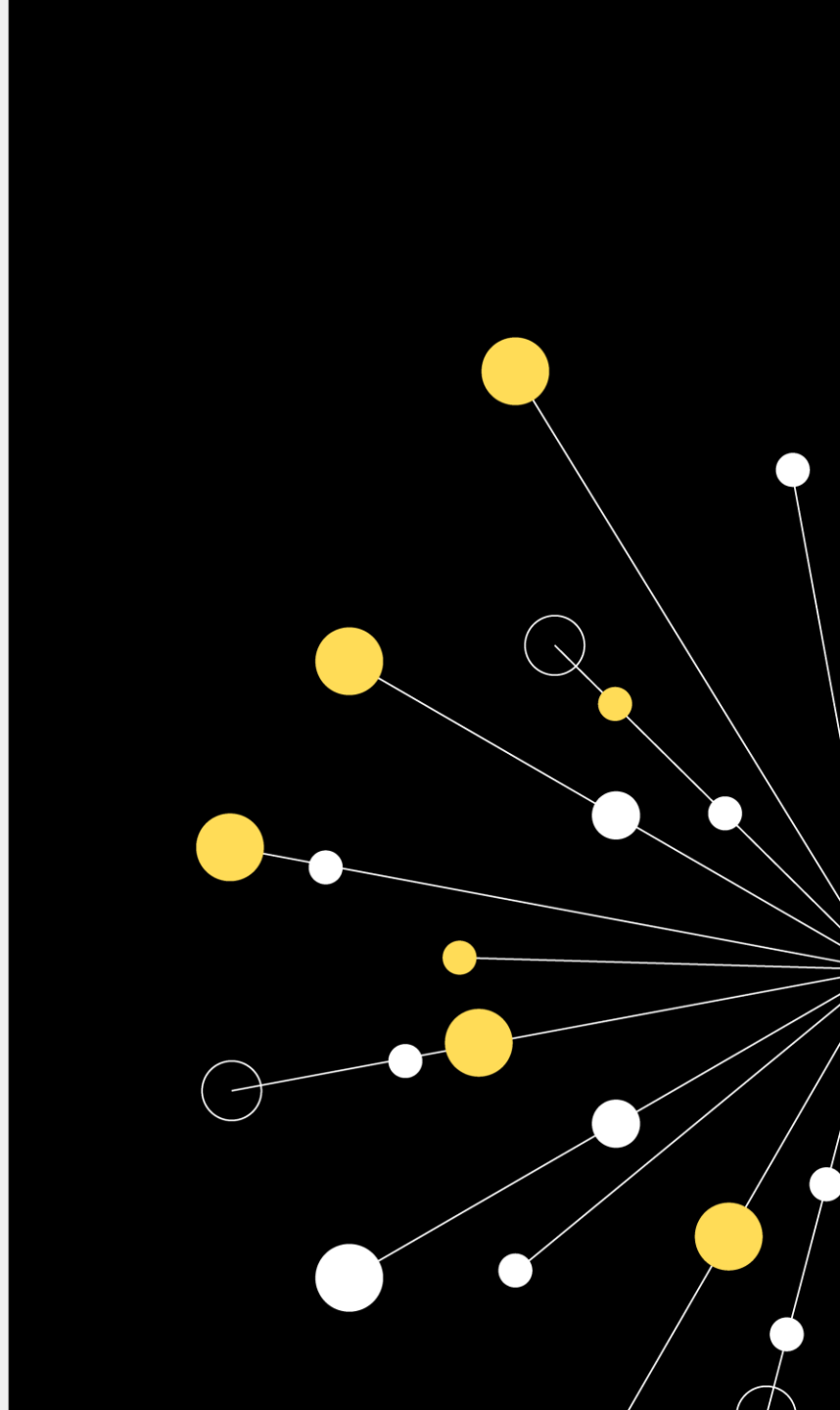
# What’s the problem?

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



QA

# Tests to Development

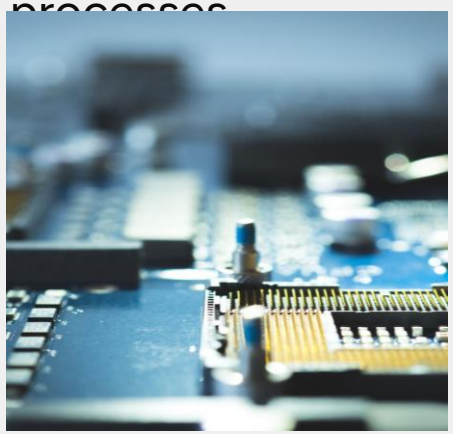
The software industry didn't invent the idea of tests first, development after (TDD).

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

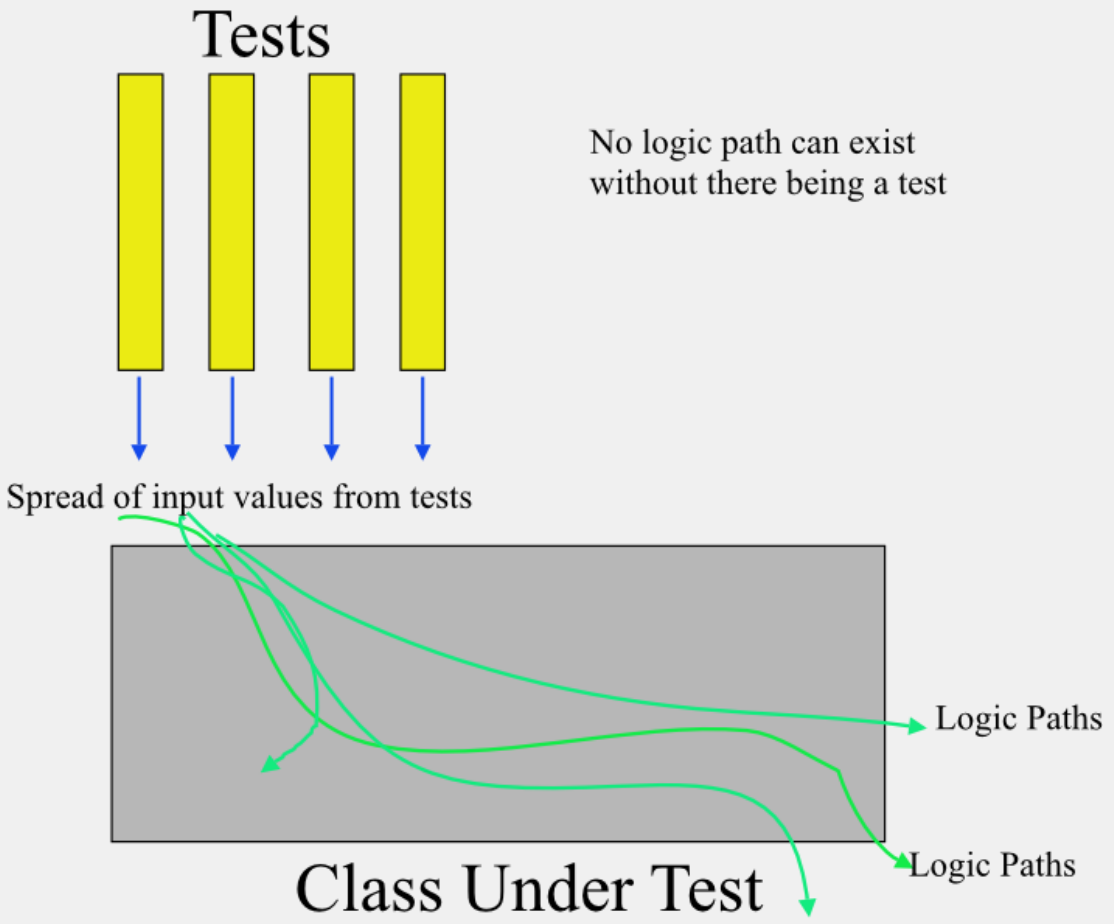
**Test before approach**

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.

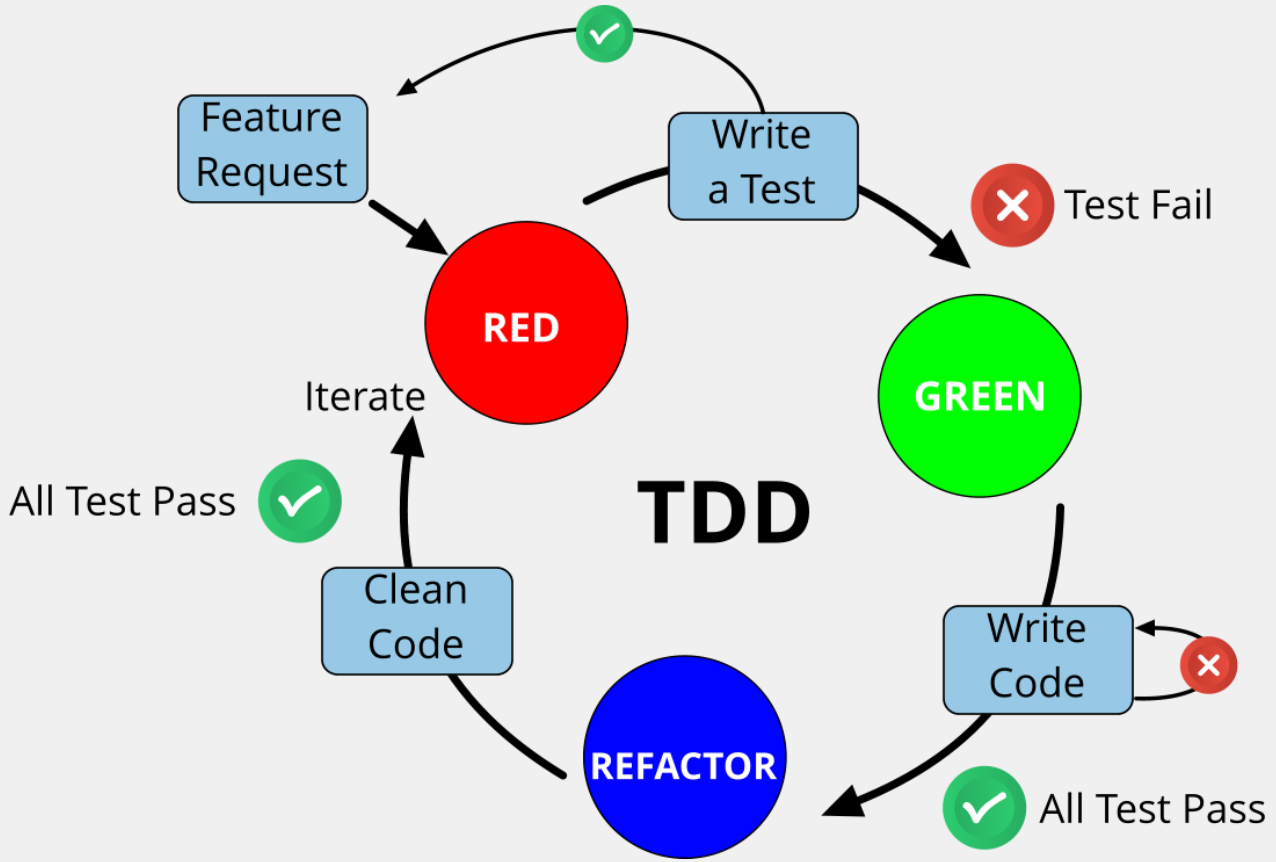
Many organisations have policies of not testing setters and getters.

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



# Scenario

You are designing a **payment processing service** for an e-commerce platform.
The system must handle:

- Charging the customer’s card.
- Applying discounts, cashback, or loyalty points.
- Preventing **double charges** if the request is retried.

# Step 1: Define Business Rules as Tests

Before writing any code, think in terms of **expected behaviors**:

1. Charging $100 reduces customer balance by $100.
2. If a 10% discount applies, only $90 is charged.
3. If balance < amount, reject the payment.
4. If the same request is sent twice, charge only once (idempotency).
5. At this point, **all tests would fail** because no code exists yet.

# Step 2: Implement Just Enough to Pass

- Start by writing the simplest implementation to satisfy Rule #1.
- Add more tests for Rules #2–#4, and evolve the code until they pass.
- Keep repeating the Red → Green → Refactor cycle.

# Step 3: Importance of TDD here

**Without TDD**

- Double charging could cost millions.
- Edge cases (negative values, retry handling) may be missed.
- Fear of refactoring slows down development.

**With TDD**

- Tests encode business rules as a **safety net**.
- Edge cases are caught early.
- Developers can add features (multi-currency, refunds) **with confidence**.

# Key Takeaways

TDD here is not about testing syntax — it's about **protecting critical business logic**.

Think of tests as **contracts**: once defined, they guarantee that payment rules will always hold true, no matter how often the system evolves.

# Go Testing Basics

**Run tests** with:

*go test ./...*

**Test files** must end with:_

test.go → parser_test.go

**Test functions**:

- Start with Test
- Signature: func TestXxx(t *testing.T)

**Naming convention**: use descriptive names showing intent

Example: TestParseTags_SingleWord

Example: TestParseTags_TrimsSpaces

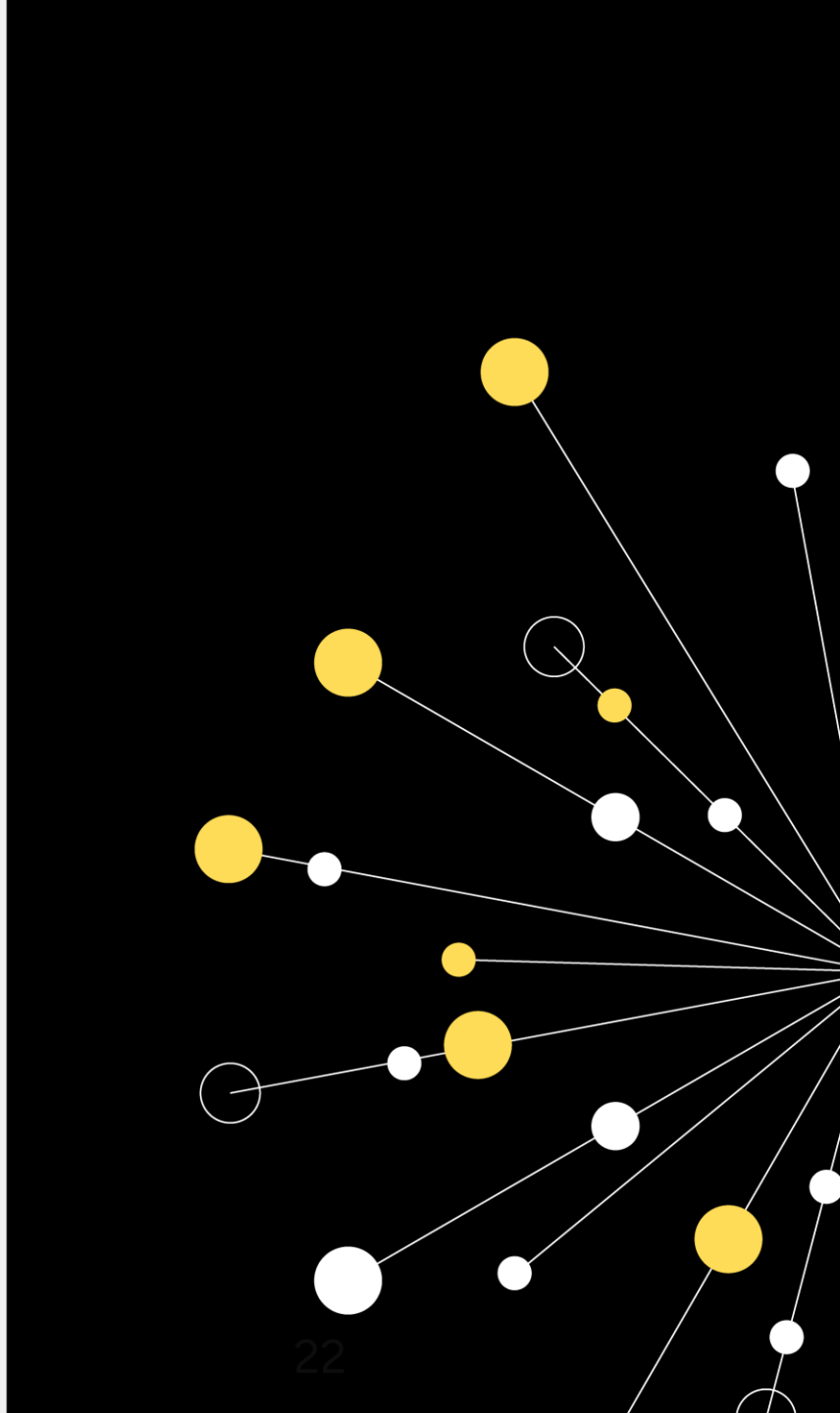
**Best practice**: keep tests short, isolated, and readable

# Example1 – TDD Walkthrough

### Q1 - 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**.



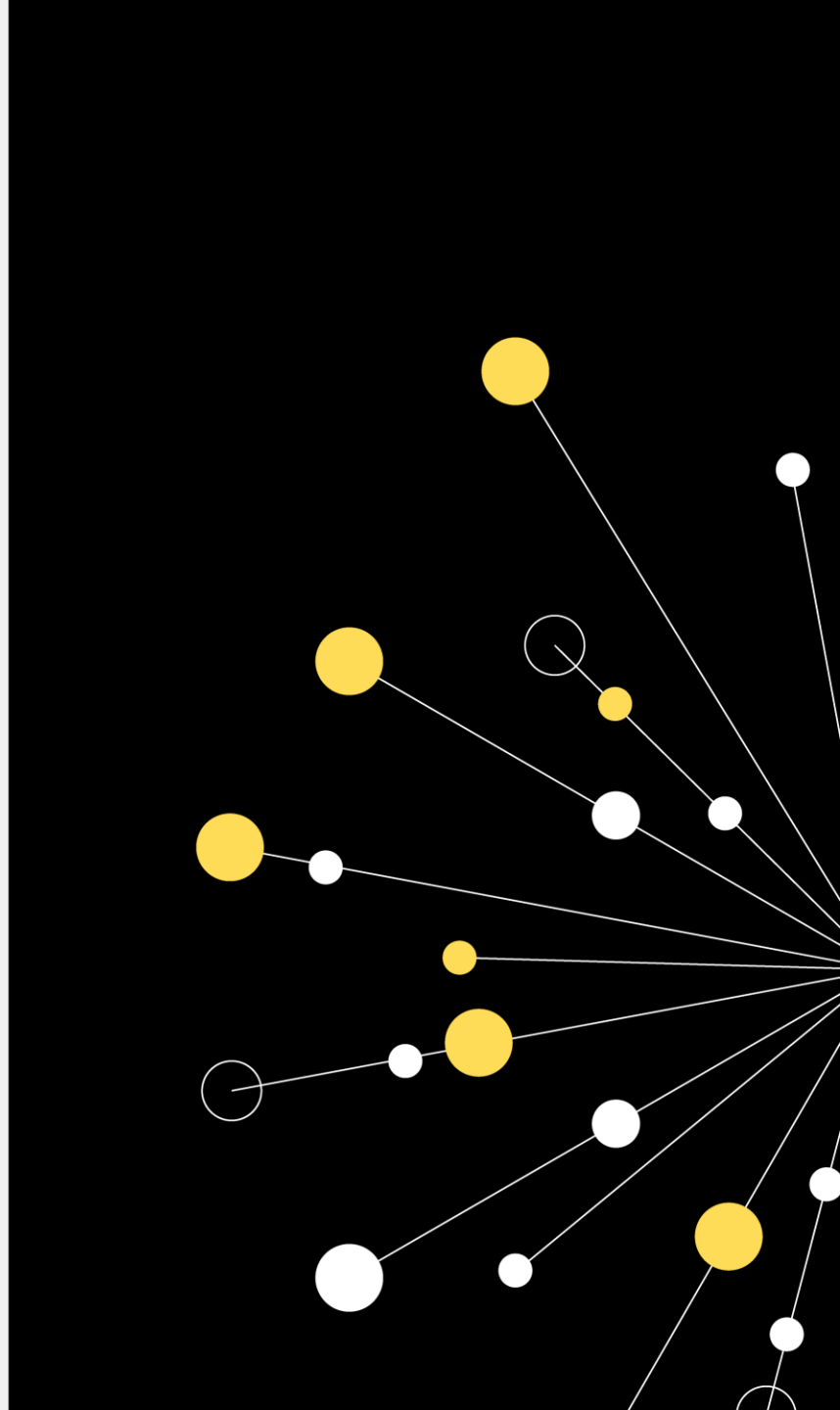
# TDD Takeaways

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



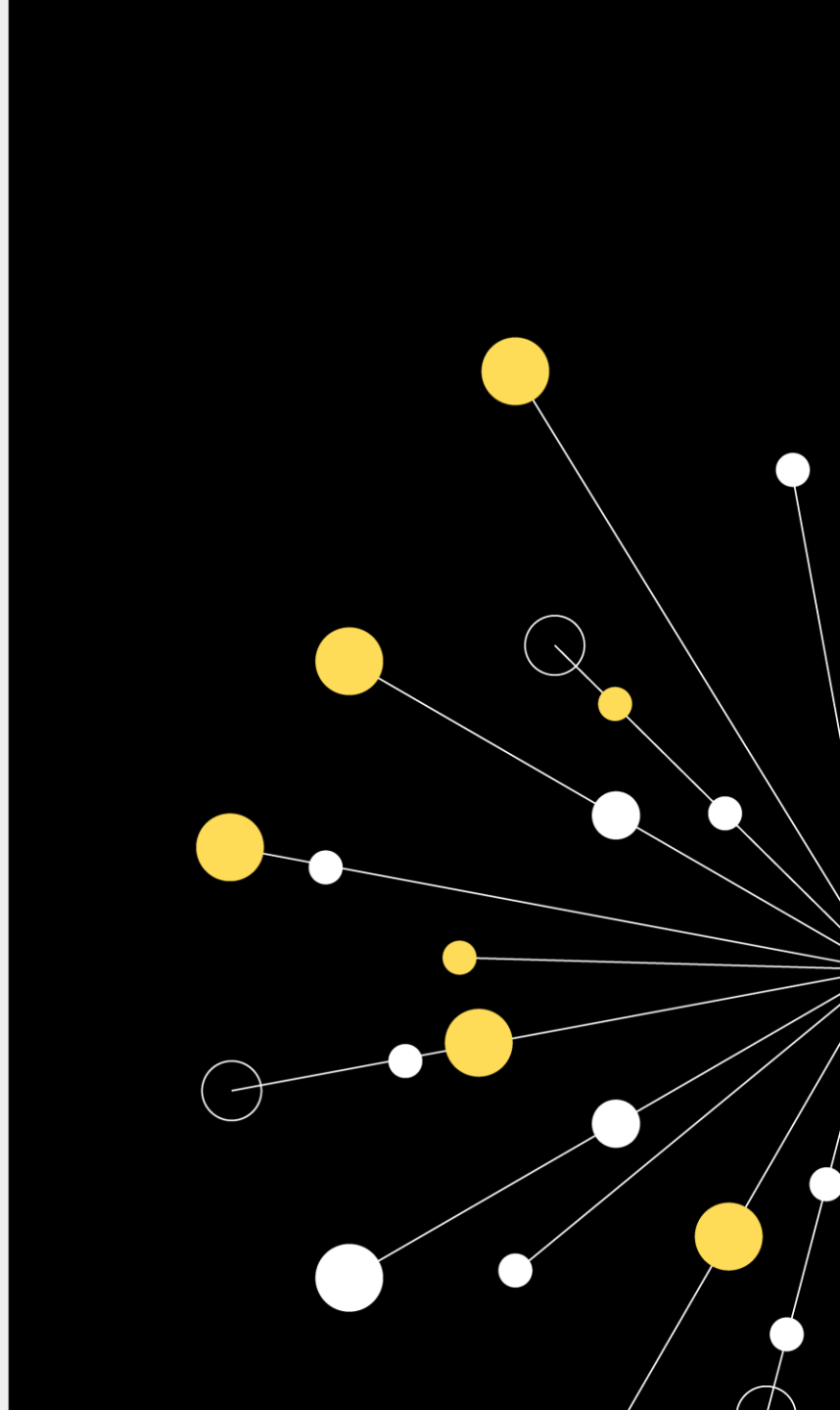
# Example 2 - TDD Case Study

**Example 2 - 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.

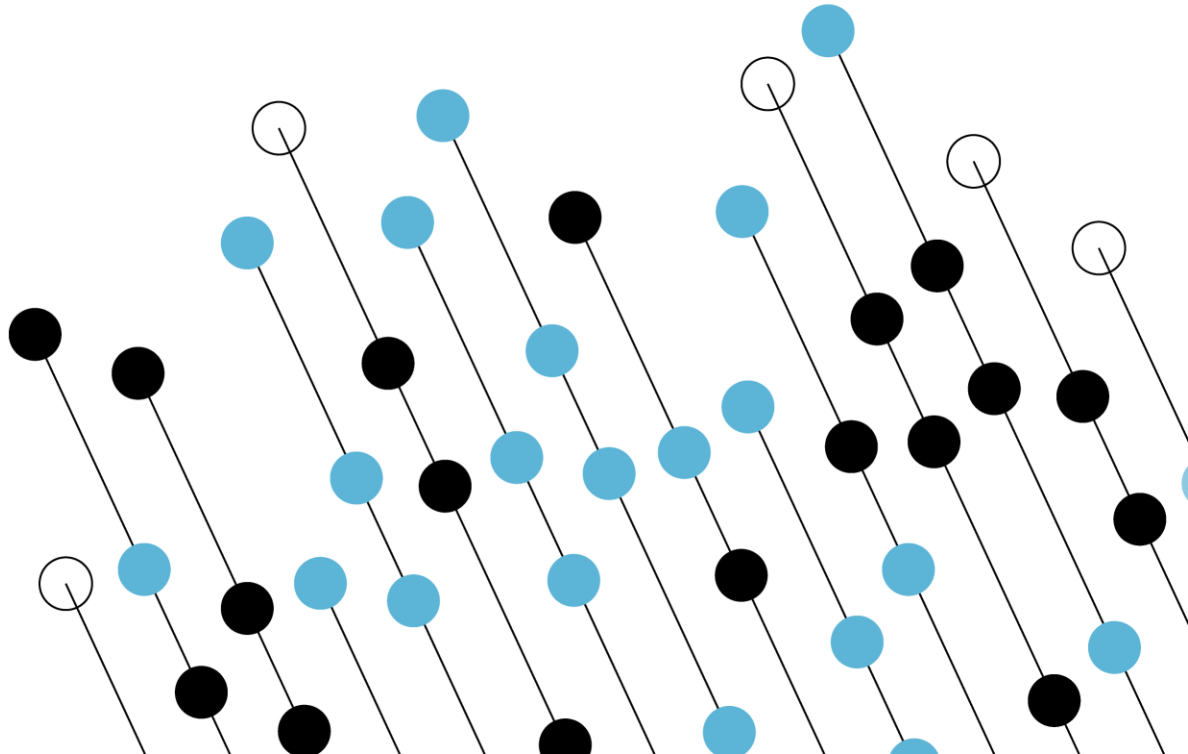


QA

# 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

```
func TestParseTagsSingleCase(t *testing.T) {
    // Arrange
    // Given a comma-separated string with extra spaces
    input := " golang, python "
    expected := []string{"golang", "python"}

    // Act
    // When we parse the tags
    result := ParseTags(input)

    // Assert
    // Then the result should be a slice of trimmed tags
    if !reflect.DeepEqual(result, expected) {
        t.Errorf("expected %v, got %v", expected, result)
    }
}
```



QA

# Qualities of a Good Unit Test

**Isolated – does not depend on any other unit test.**

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

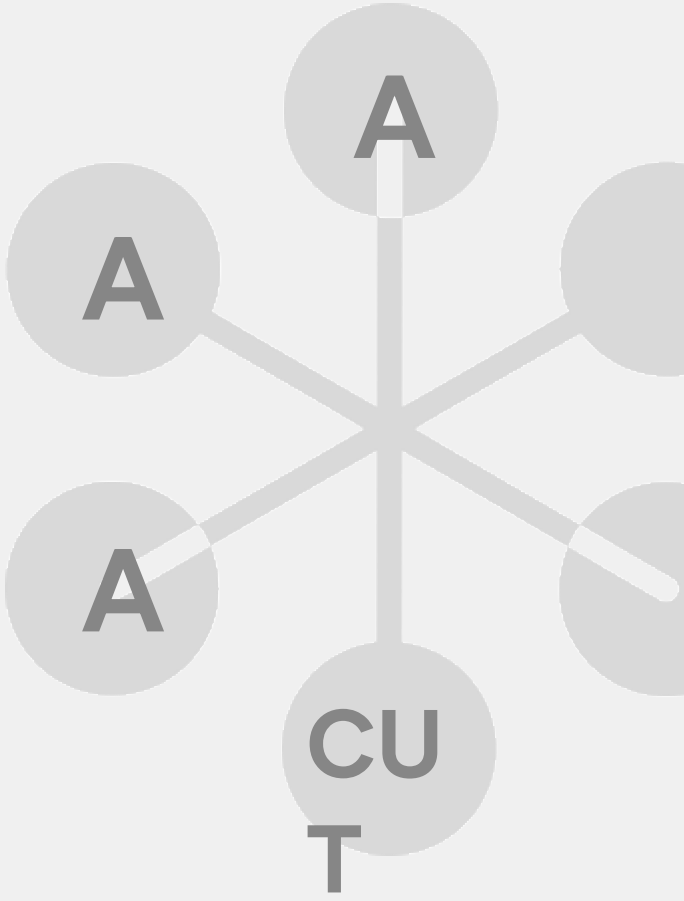
1. **A**rrange
2. **A**ct
3. **A**ssert

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

```
func ParseTags(input string) []string {
    //  just enough to pass this test
    return []string{"golang"}
}
```

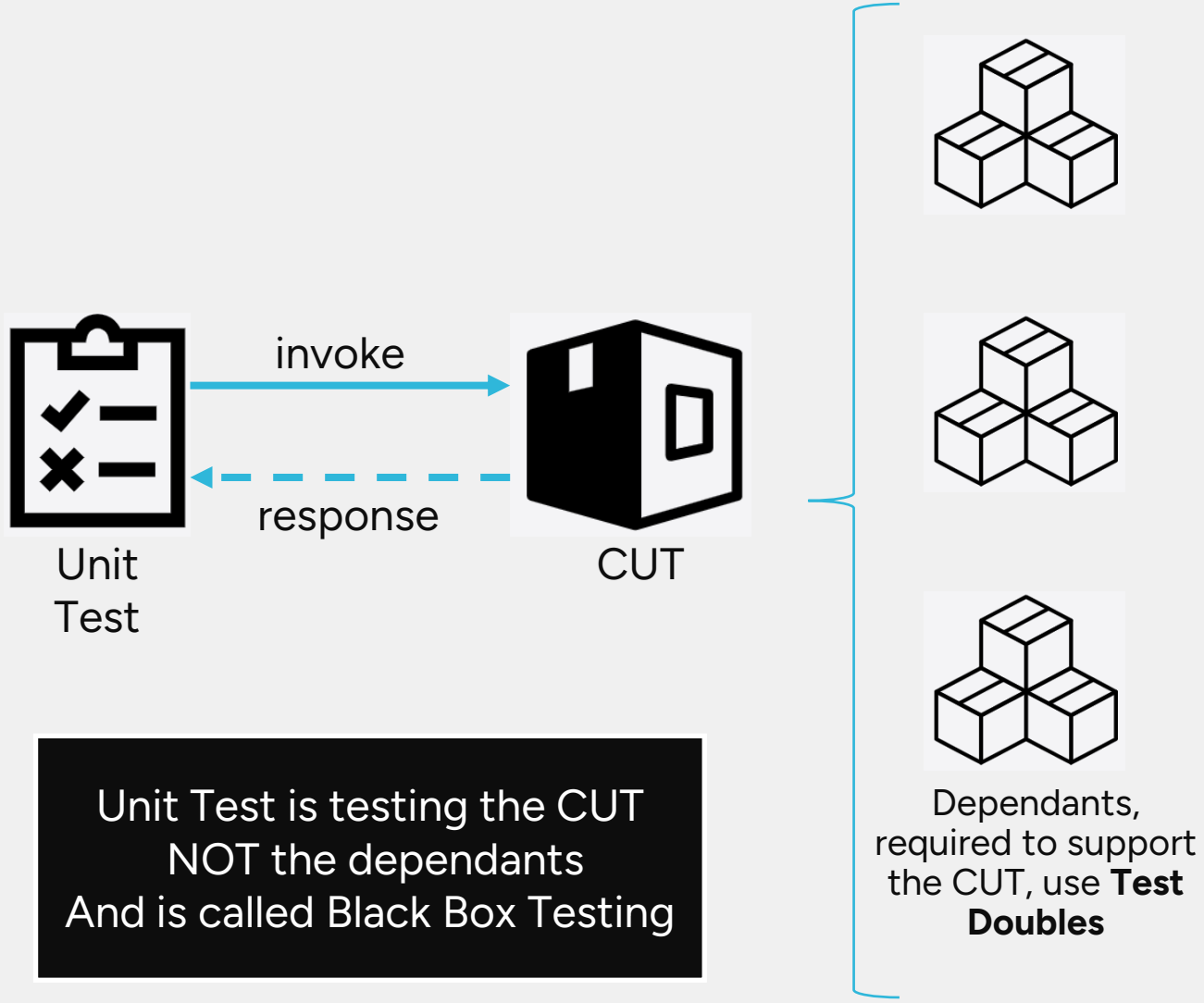
Only write enough implementation code to pass the test

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

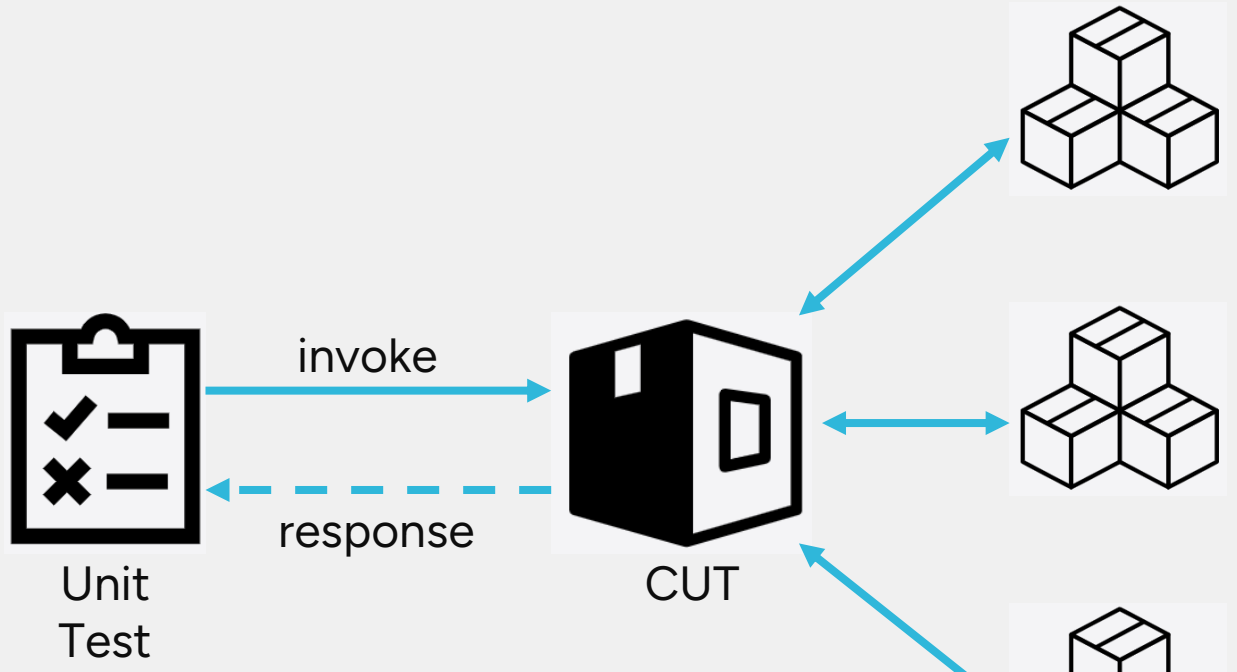
Only write what is needed to maintain good test coverage

# 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

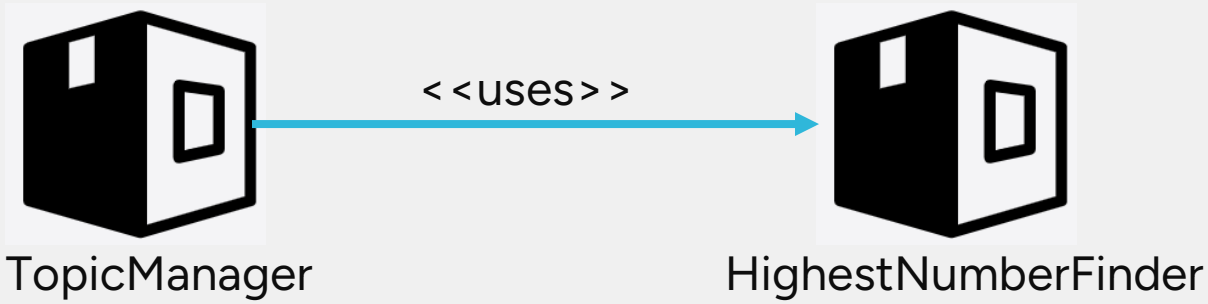
# Extended Lab – Find Highest Number

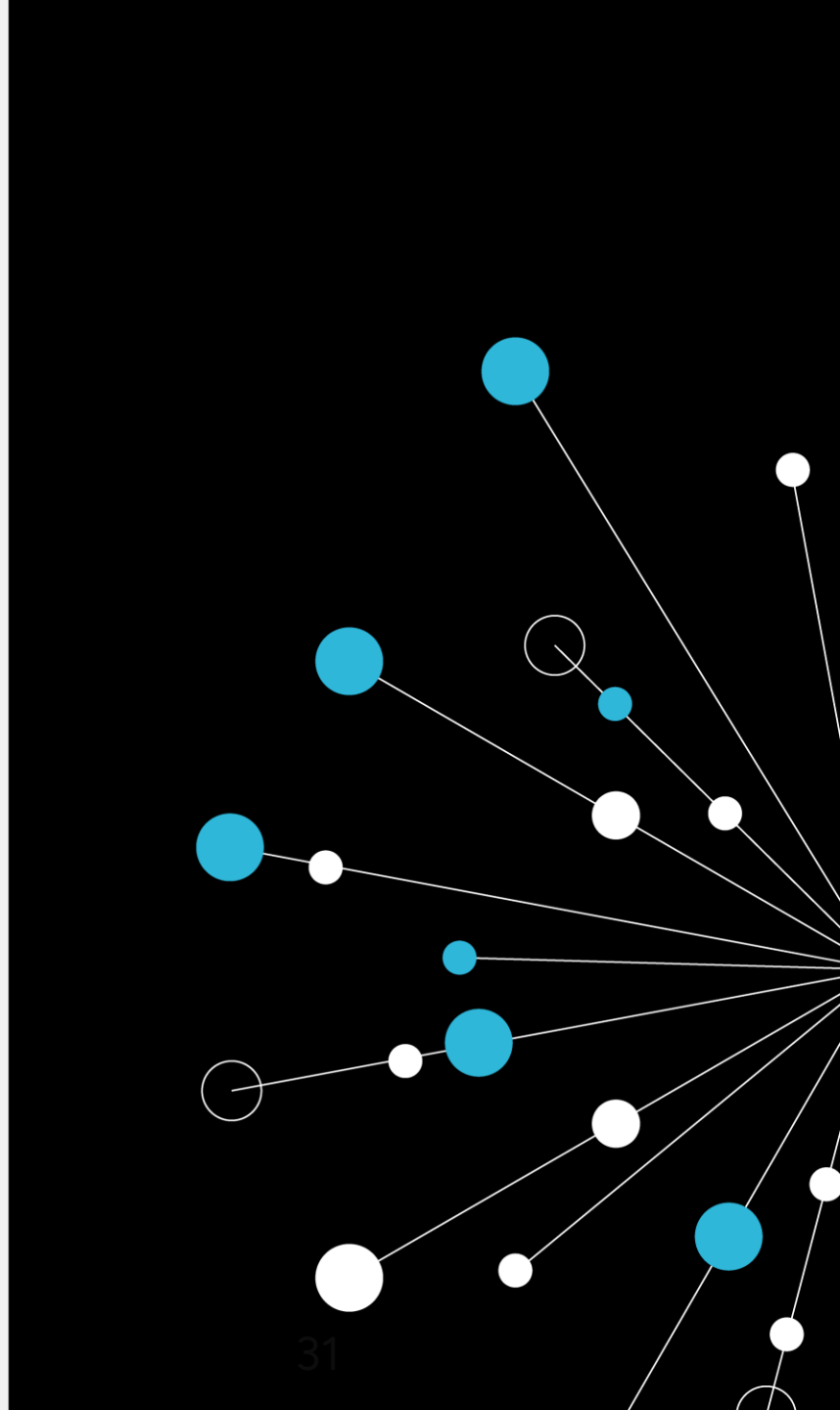
A requirement has come through that the highest number finder must deal with finding the highest number for a series of subjects being taught.

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



# Example 3

**Q3 - Topic Manager**

Requirements -

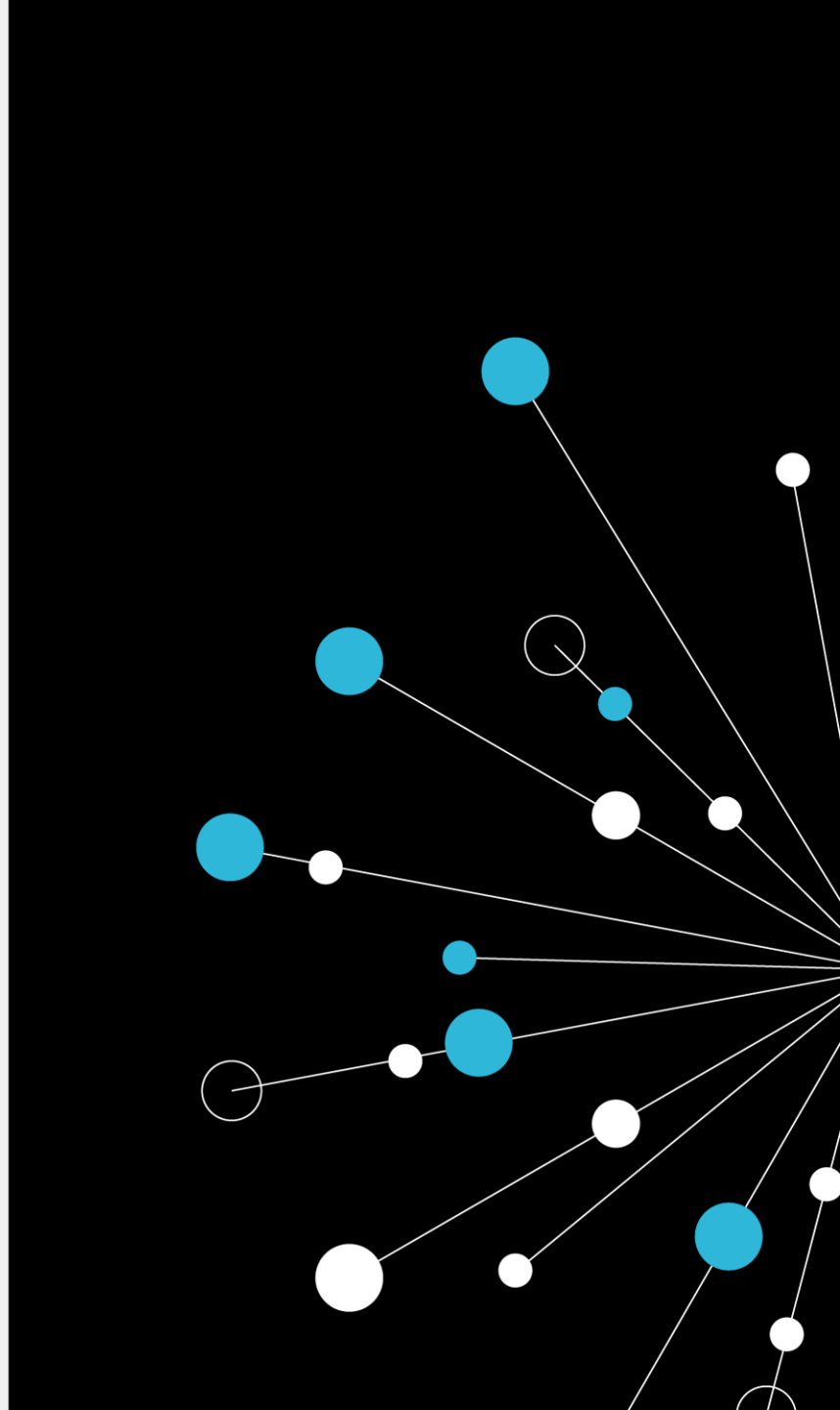
The system should manage **topics** and their **highest scores**.

Implement the following functionality:

- **Find the highest number** in a list of integers.
- **Assign the highest score to a topic** (e.g., "Math → 95").
- **Write the topic and highest score to a file** on disk.

Ensure that your implementation meets these conditions:

- An **empty list** should throw an error.
- A **single-item list** should return that item as the highest.
- If multiple numbers are equal and highest, return only one occurrence.
- Topic and score must be **persisted in a file** in a readable format.



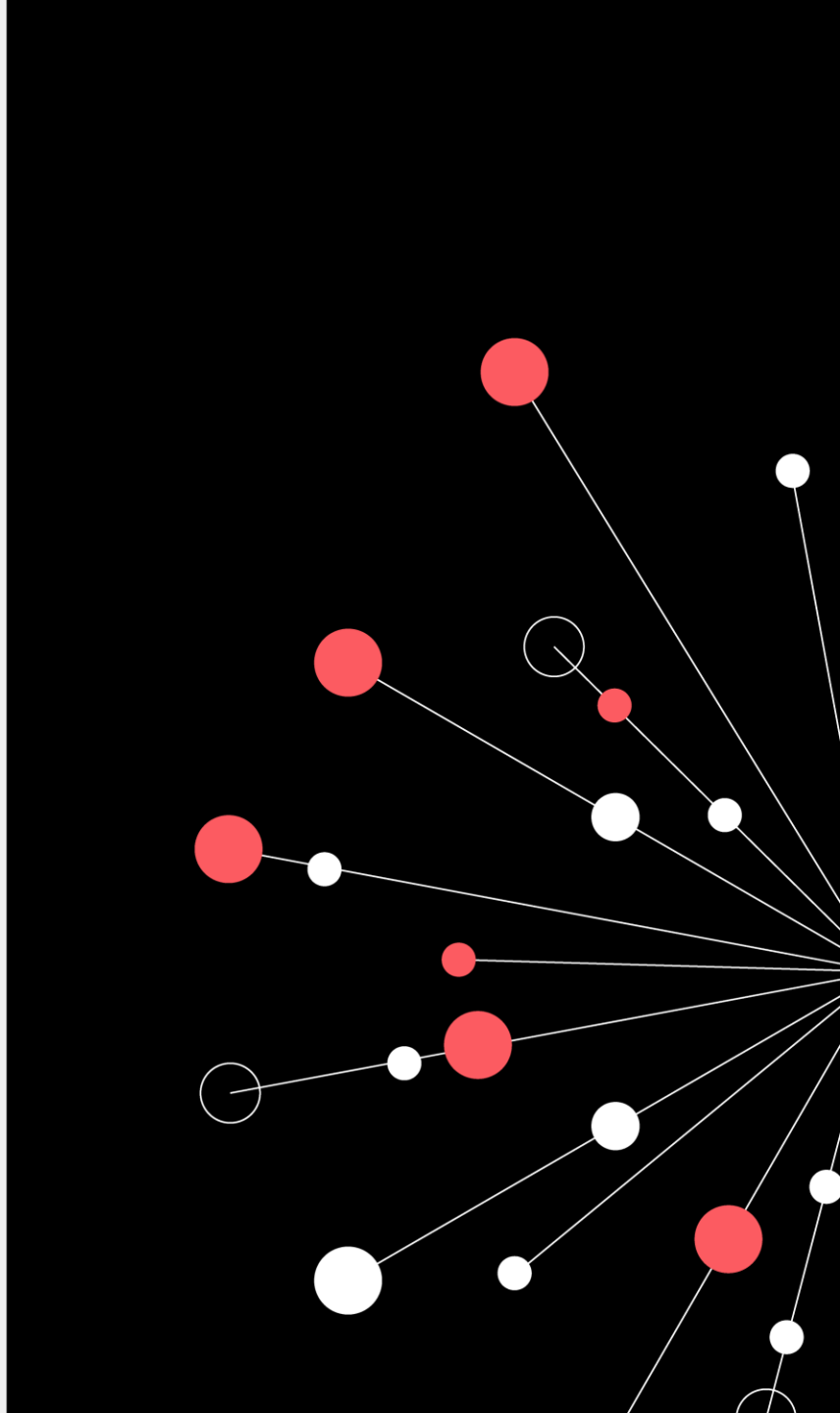
QA

# TDD Takeaway

Having written the **tests first**, we were able to:

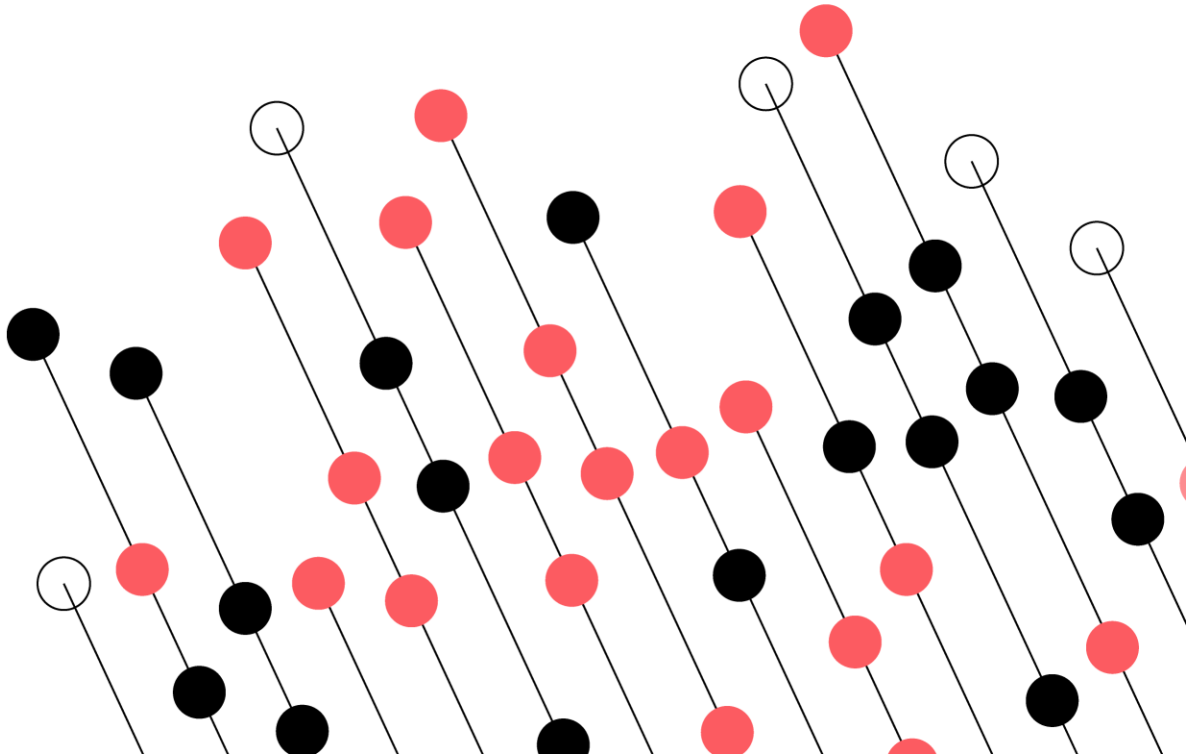
- Identify code smells
- Perform regression testing easily
- Think more clearly about the design of the code

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



QA

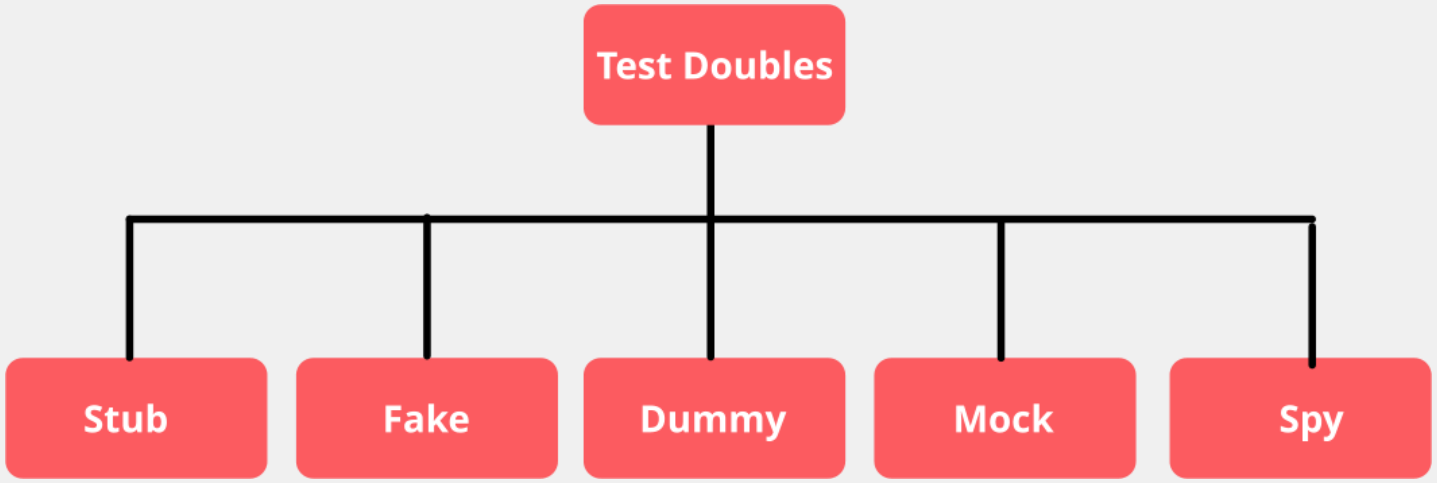
# Test Doubles



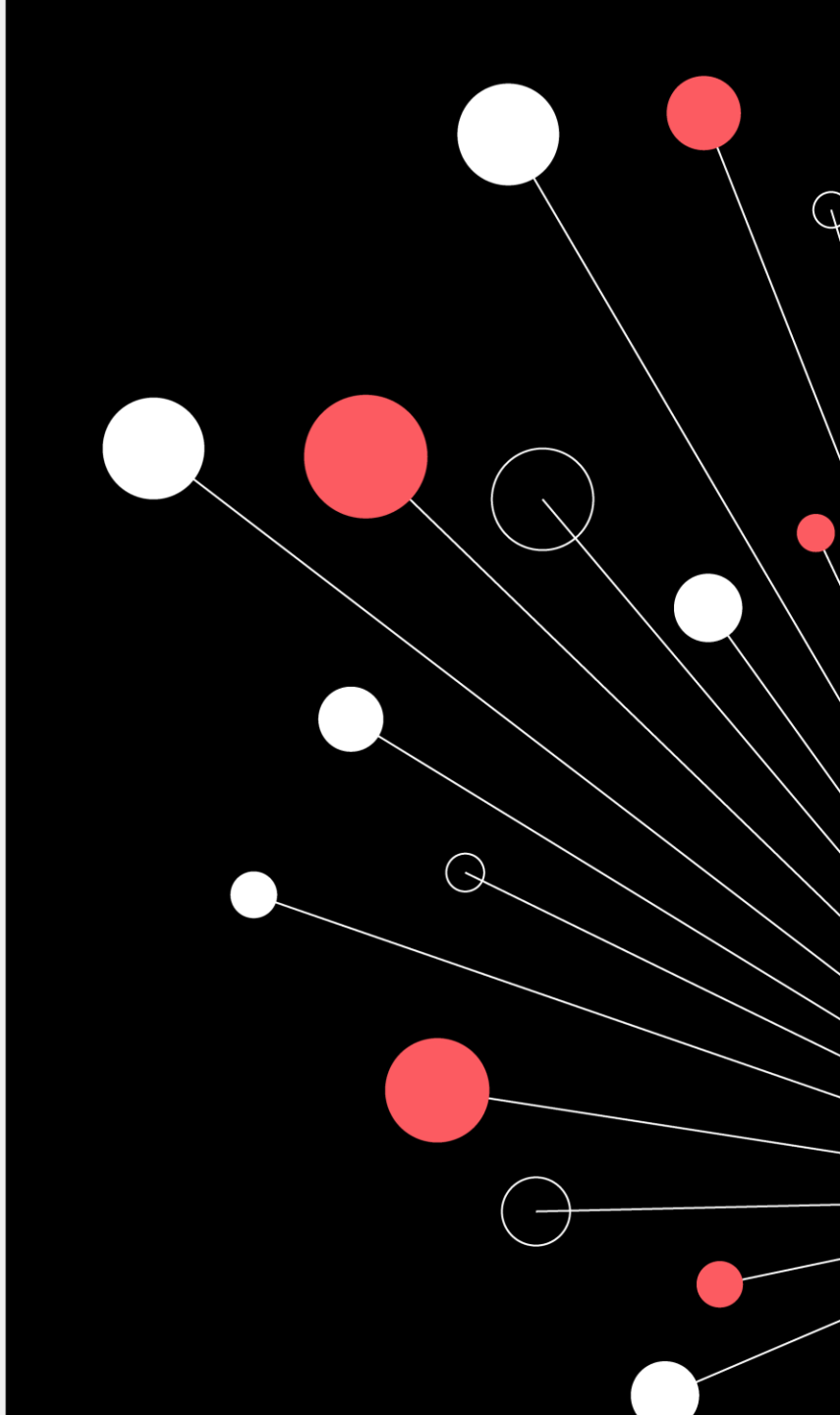
# Learning objectives

**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

Real dependencies (DB, API, File System) are:

- Slow
- Unavailable
- Irrelevant to the current test

**What is a Stub?**

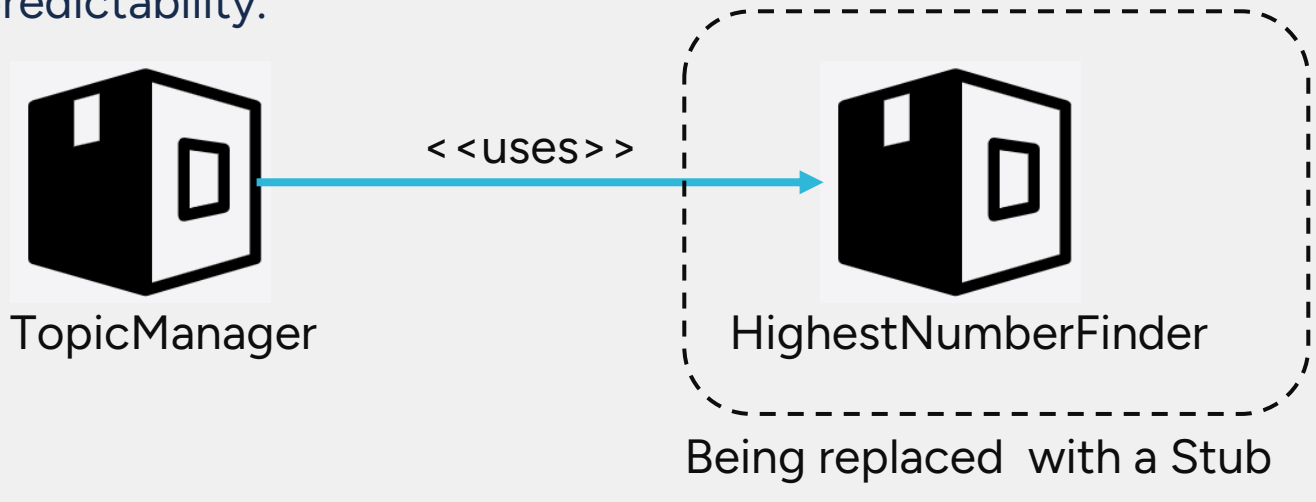
- A **simple fake implementation** of a dependency.
- Returns **fixed, predictable values**.
- Keeps tests **fast, isolated, and reliable**

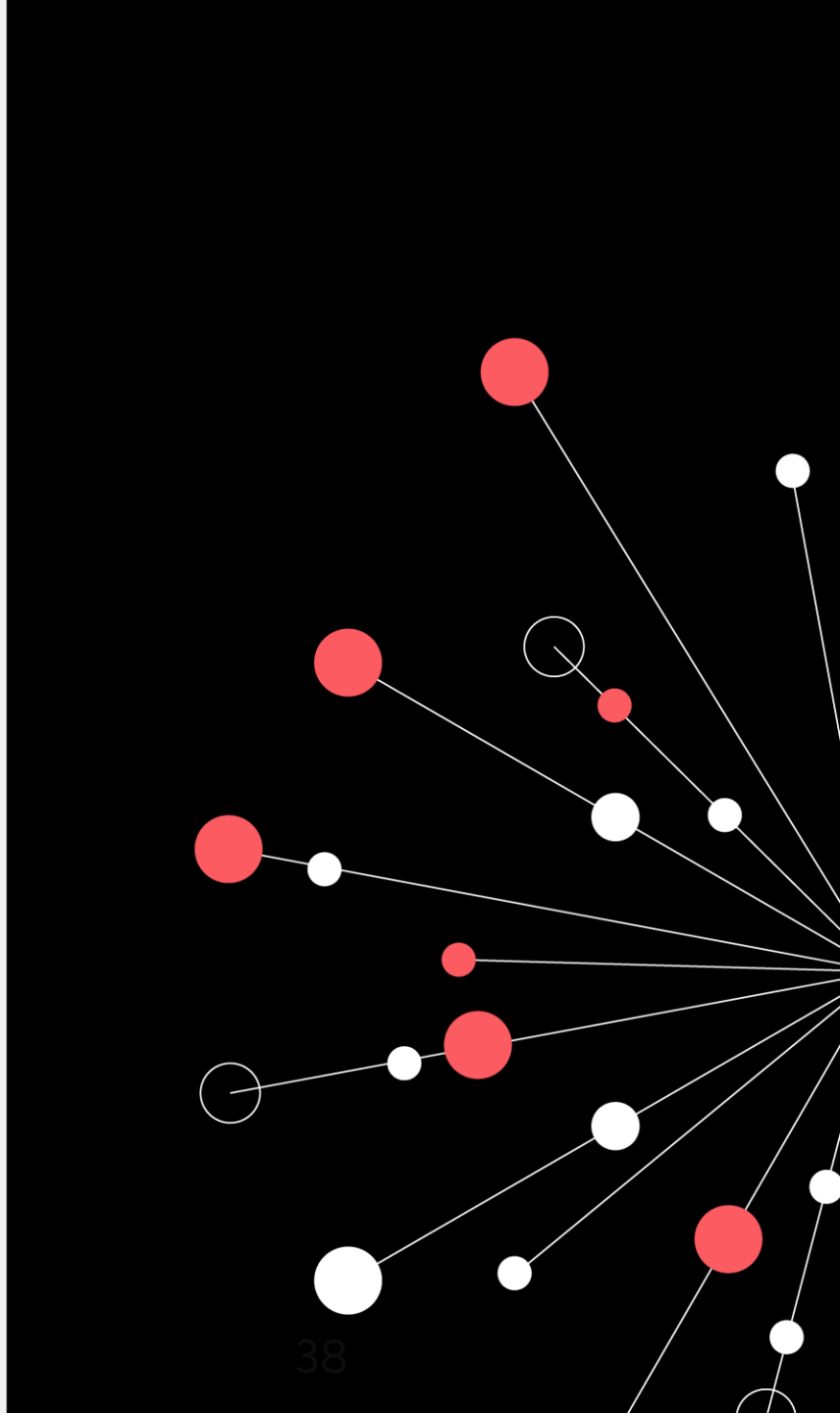
# Stub Example

**Q 4 – TopicStubExample**

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.

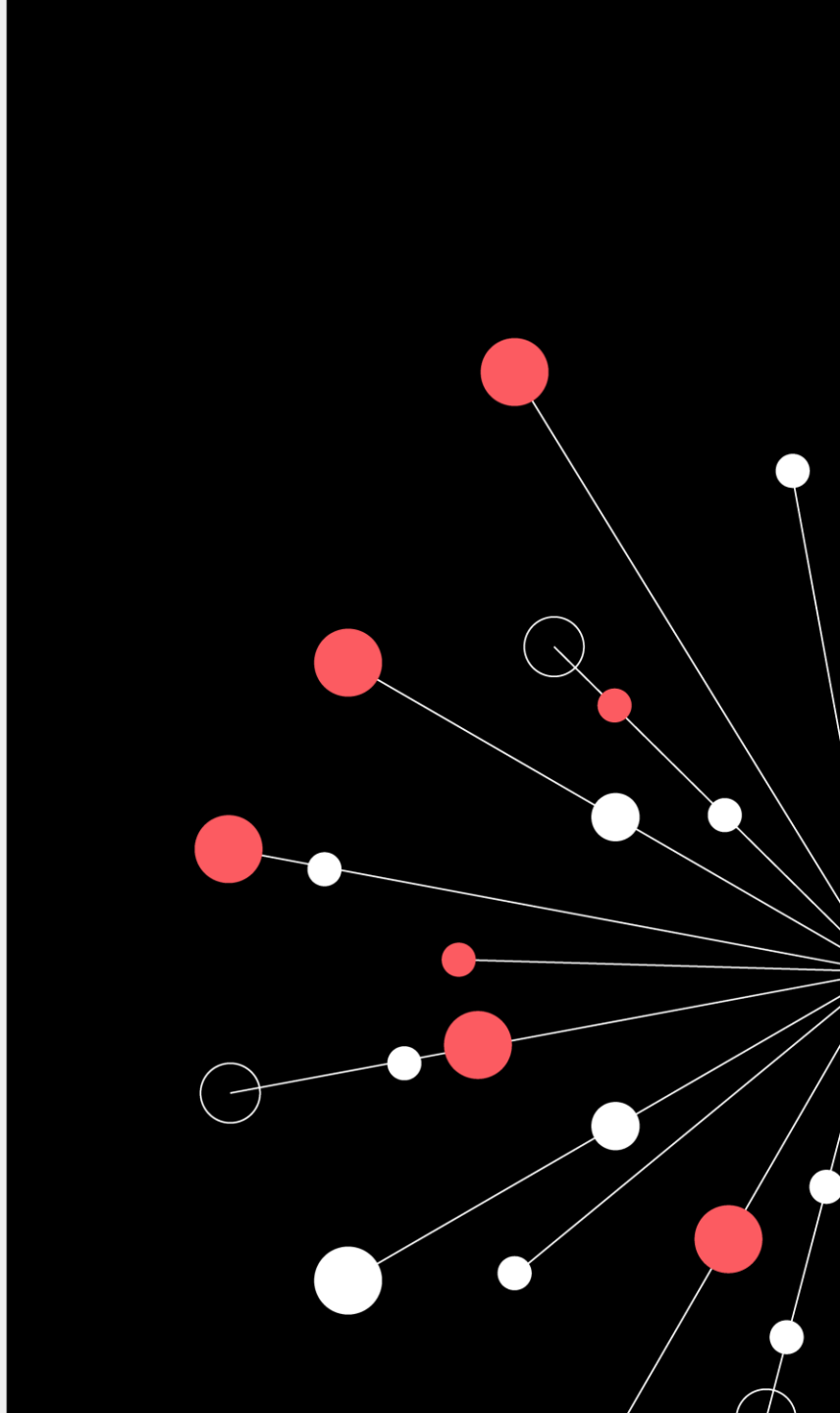


# Stubs Takeaway

- Always return **fixed values** → can’t cover all scenarios
- May not reflect **real dependency behavior**
- Useful only for **simple cases**
- Too many stubs → **hard to maintain** in large systems

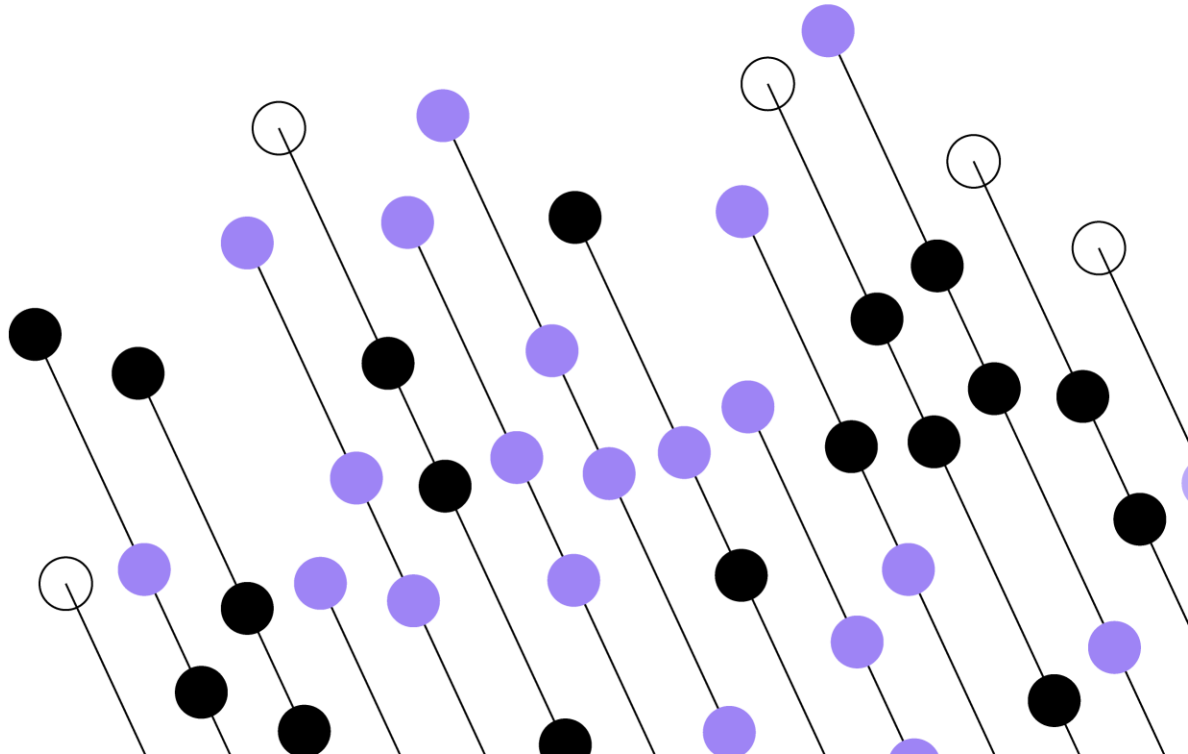
- Takeaway: Stubs keep tests simple, but they have limited realism.



QA

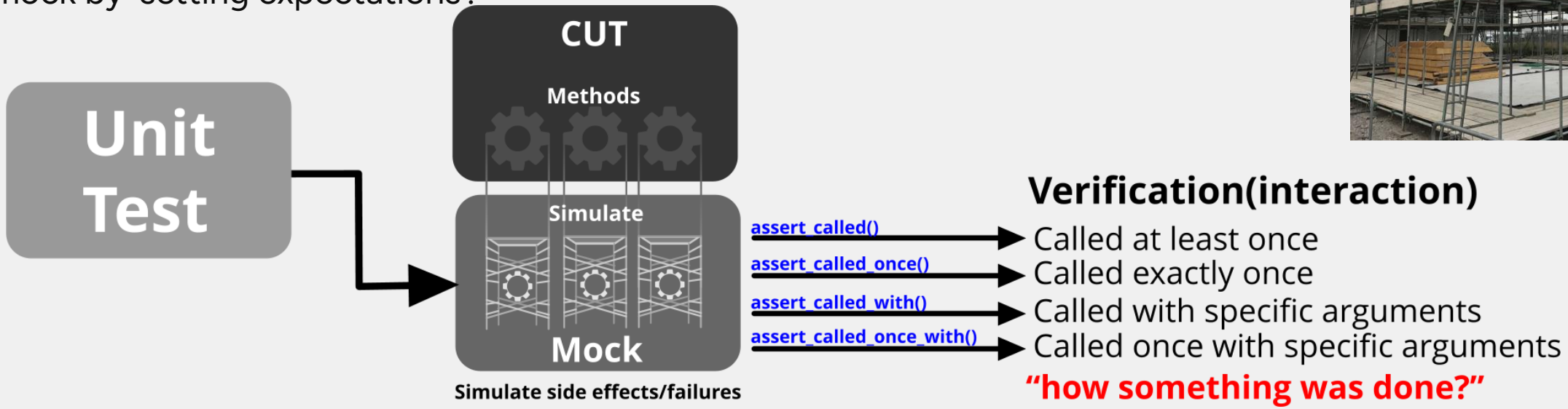
# Test Doubles and Verification Testing

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

# Mocks Goals

Verify **interactions** with dependencies

- *Was a method called?*
- *How many times?*
- *With what arguments?*

Ensure the CUT (Code Under Test) uses dependencies **correctly**

Replace real dependencies (DB, API, etc.) with a **controlled, testable version**

Increase **confidence in behavior**, not just outputs



QA

# Stubs vs Mocks

**Q5 – TopicManager (Extension of Q4)**

**Test Data Returned by the Mock**

var mockTopics = []Topic{

{Name: "English", Scores: []int{40, 65, 55}},

{Name: "History", Scores: []int{85}},

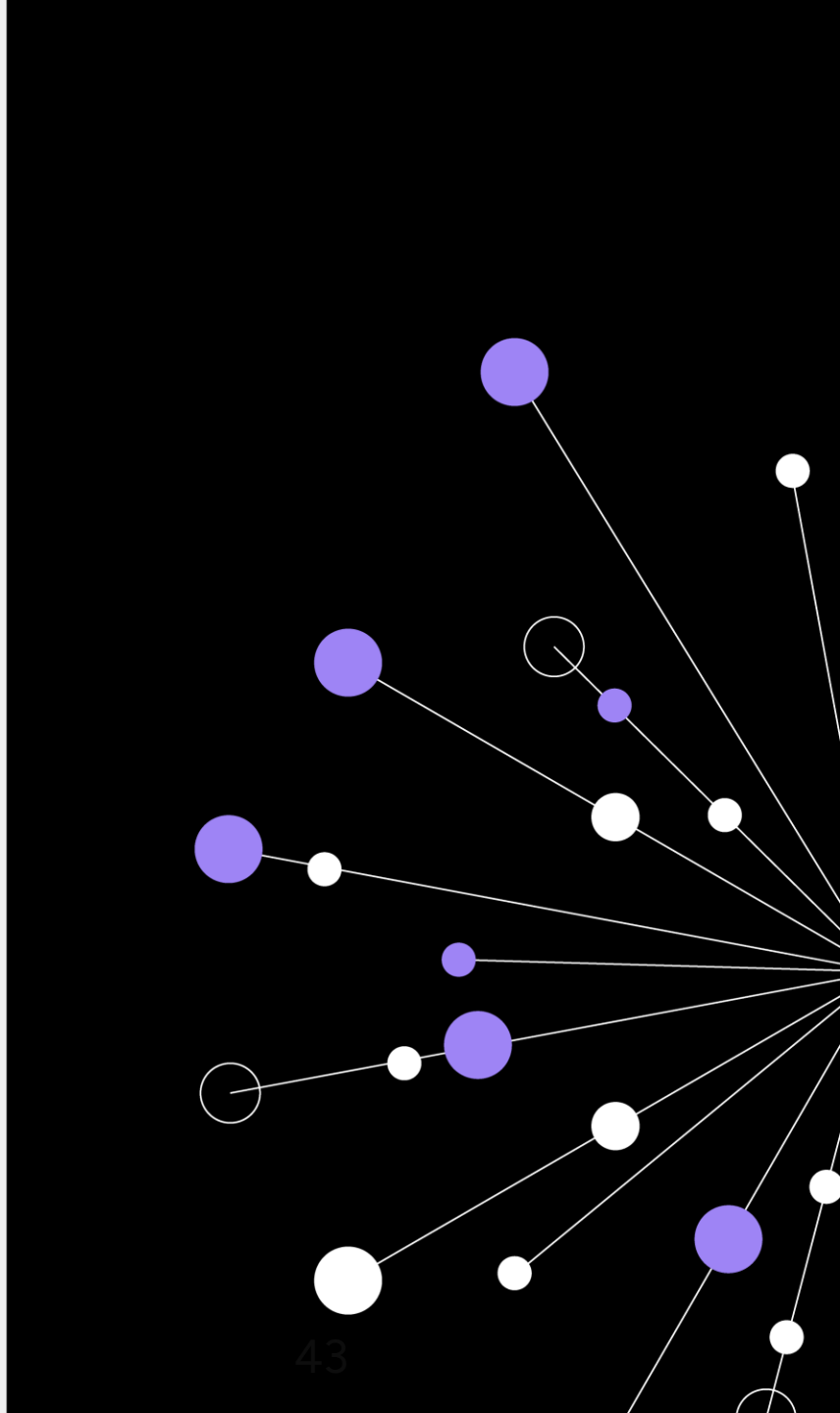
{Name: "Empty", Scores: []int{}}, // should be ignored }

**Expected result from CUT:**

expected := map[string]int{ "English": 65, "History": 85, // "Empty" excluded (no scores) }

**Interaction expectation:**

GetTopics() is called exactly once.



QA

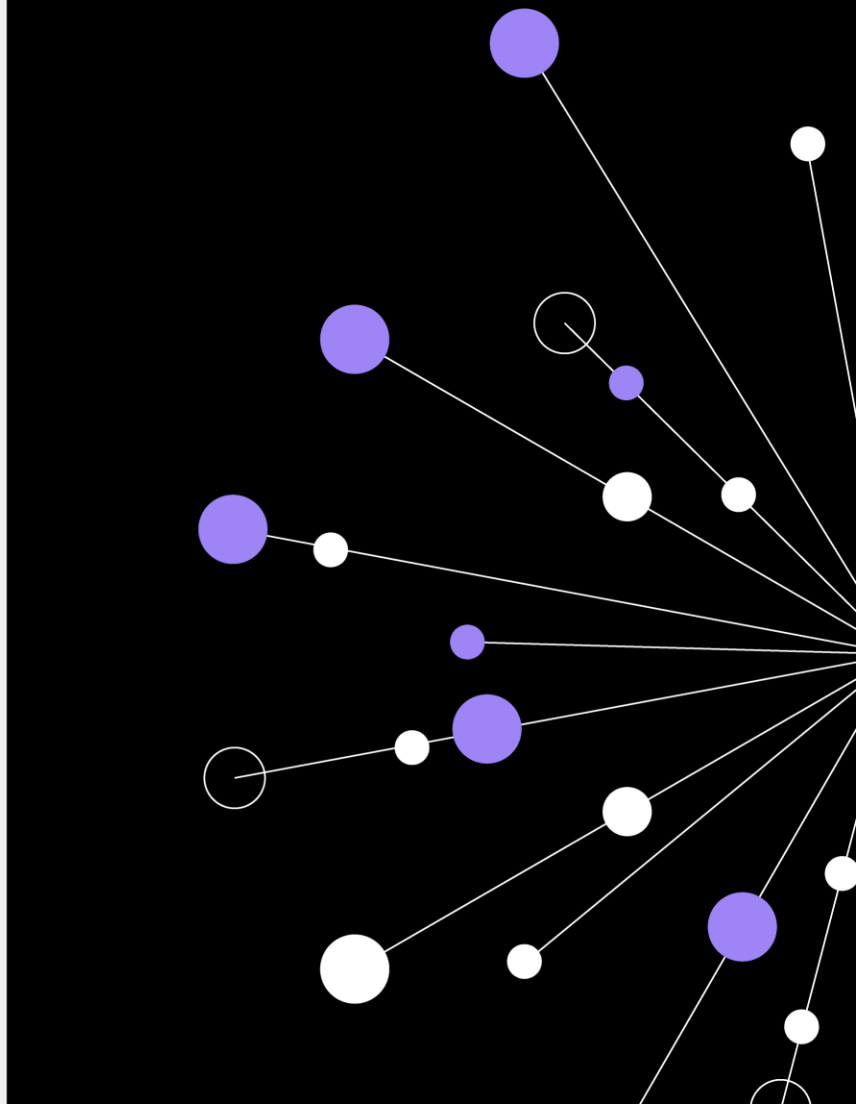
# GoMock

What is GoMock?

- Official mocking framework for Go
- Part of golang/mock package
- Generates mock implementations of interfaces

Why use GoMock?

- Isolate the Code Under Test (CUT) from real dependencies
- Simulate external systems (DB, APIs, services)
- Verify interactions



QA

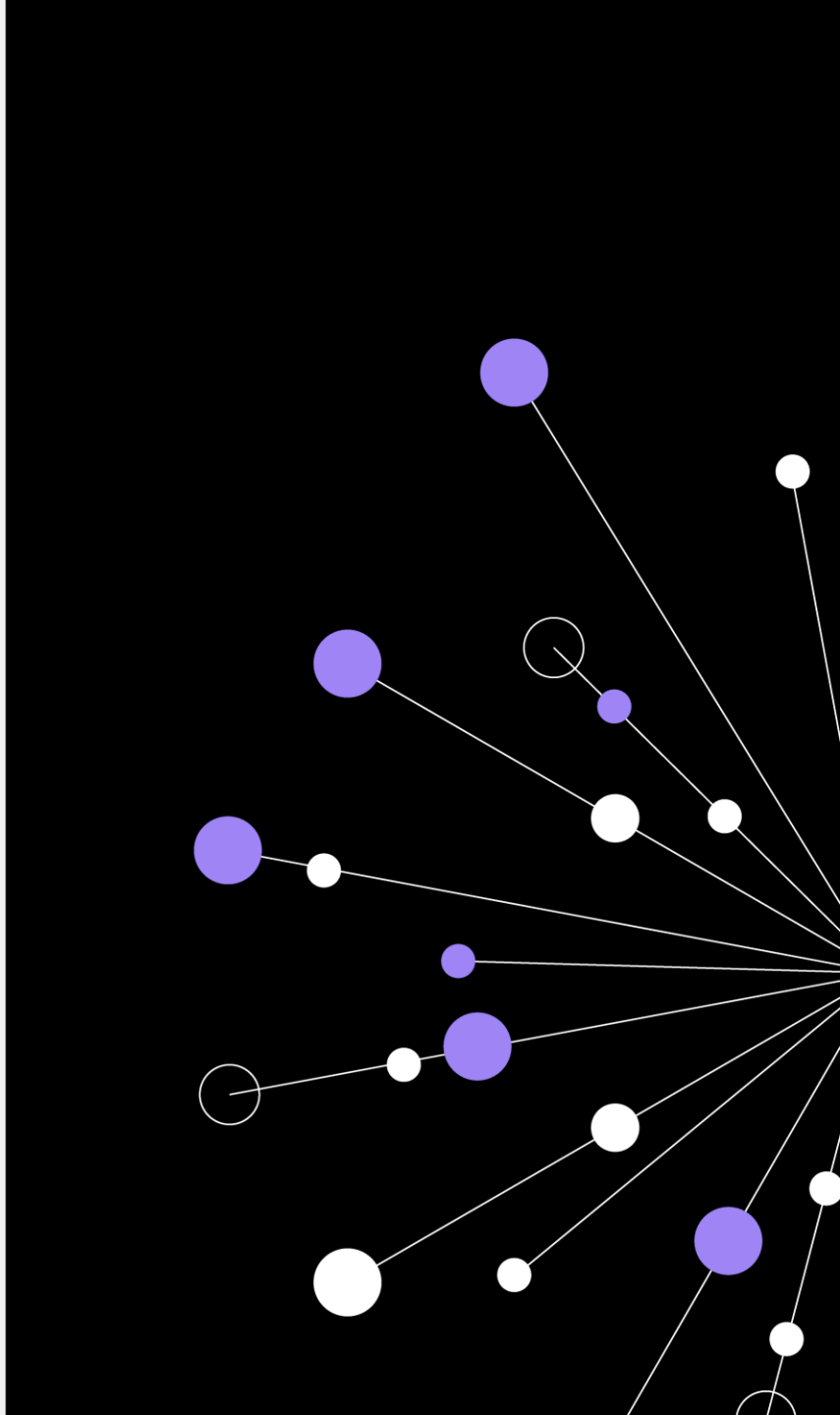
# GoMock

How it Works

- Define an **interface** (e.g., OTPService)
- Run **mockgen** → generates a mock struct
- Use mock in tests with **EXPECT()** to set behavior and check calls

Example :

mockgen -source=otp.go -destination=mock_otp/mock_otp.go -package=otp



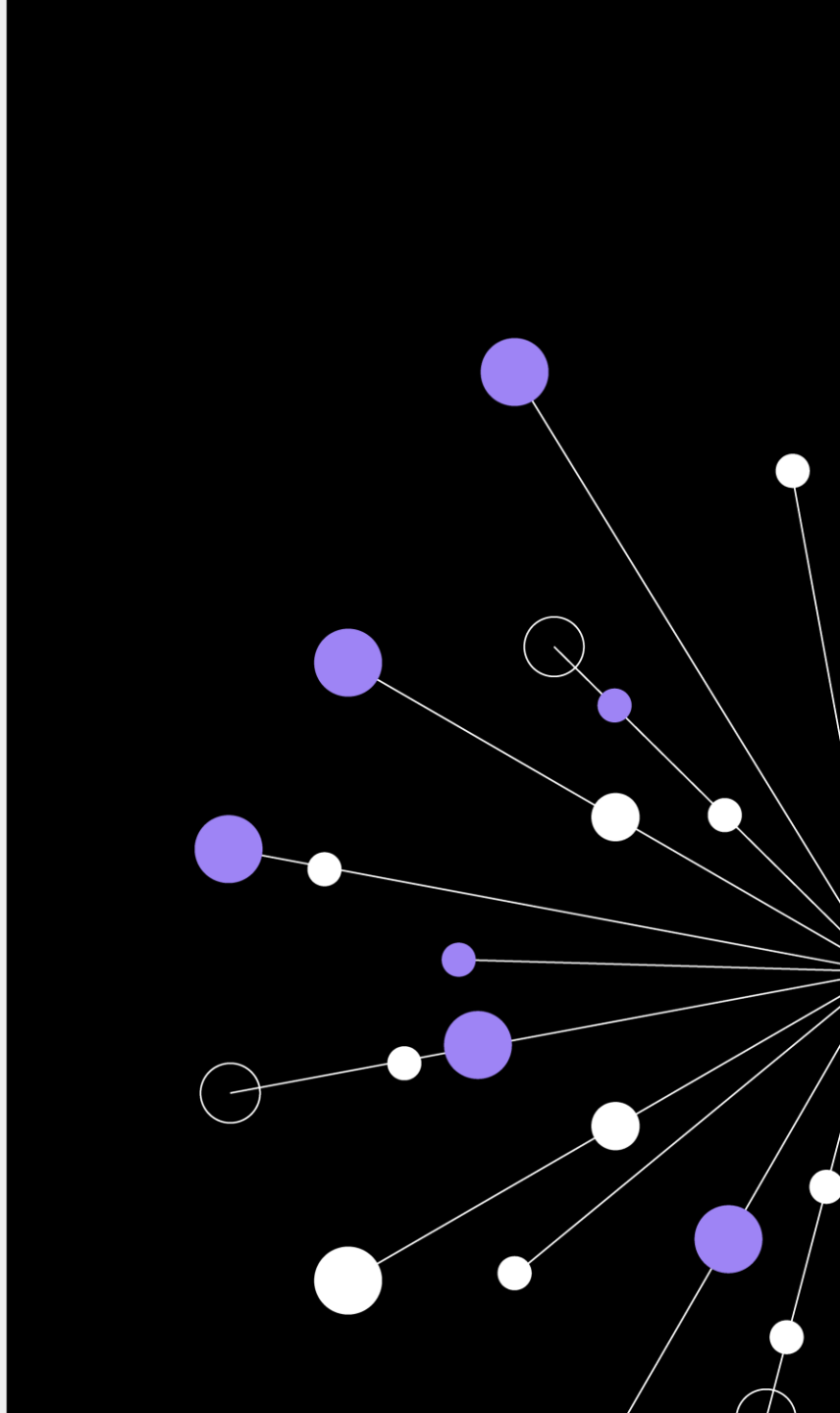
QA

# Example – OTP Authentication

Q6 - Build an AuthManager that sends and verifies one-time passwords (OTPs) via an external OTPService.

The AuthManager must:

- Request an OTP from OTPService.SendOTP(email) and **temporarily store** it against the email.
- Verify a user's OTP via OTPService.VerifyOTP(email, otp) **and** ensure the OTP matches what was stored.



QA

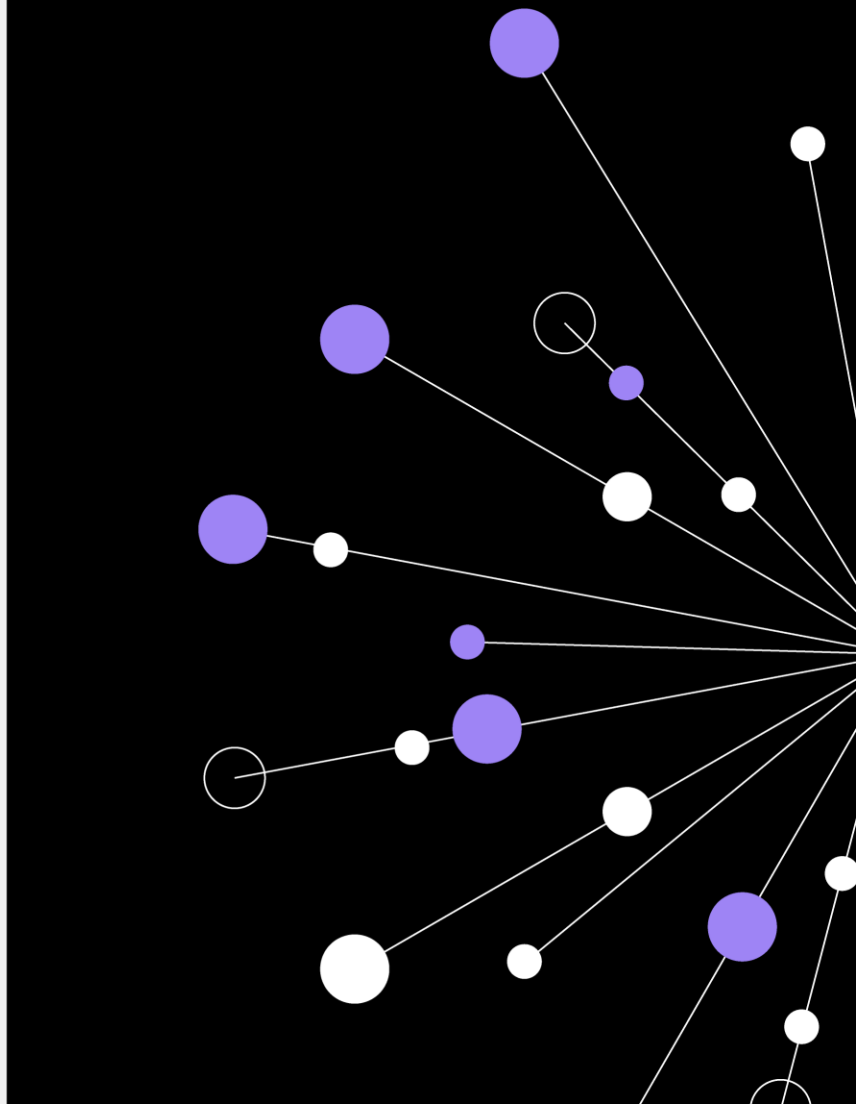
# Mocking Objects Takeaway

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.

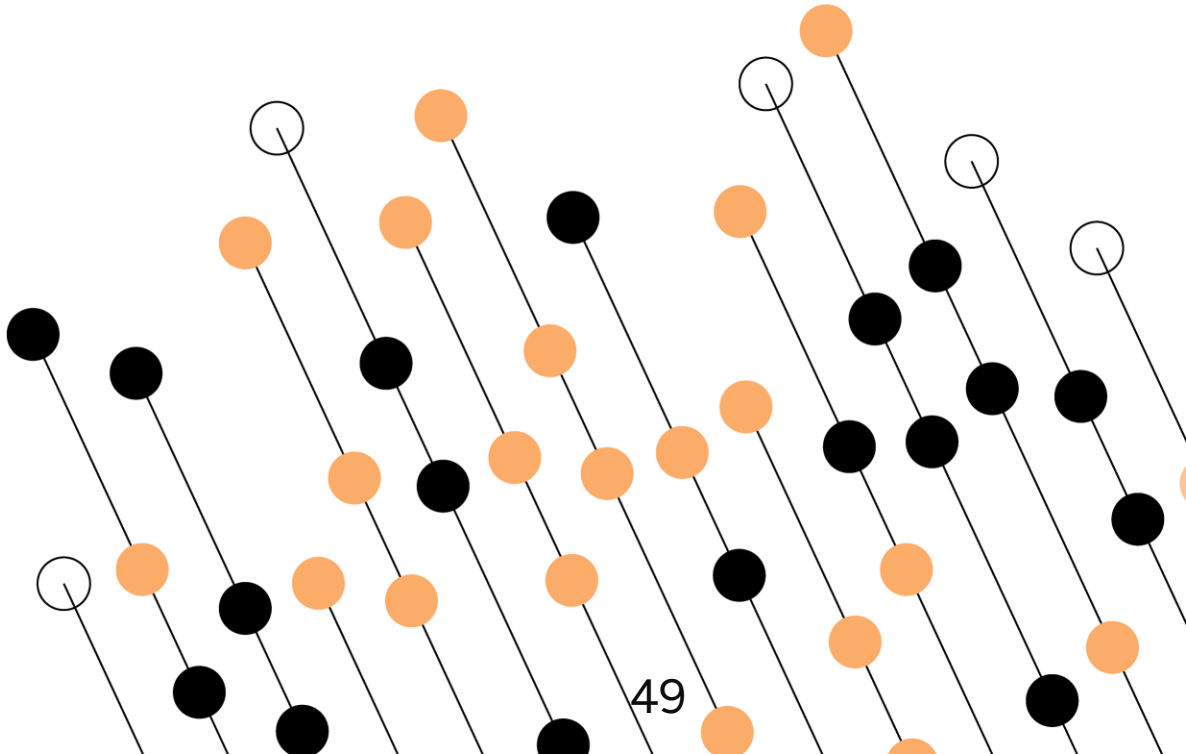
By default, its methods delegate calls to the real object, allowing the actual behaviour to run.

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.

QA

# 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

# Code Example

```
func GenerateReport() string {
    now := time.Now()                    // hard dependency
    fmt.Println("Report:", now)          // side-effect
    if now.Hour() < 12 {
        return "Morning Report"
    }
    return "Afternoon Report"
}
```

# Why is the code hard to test?

**1. Hidden Dependency (time.Now):**

- Always uses the current system clock.
- You cannot **inject a fake time** to test morning vs afternoon.

**2. Side Effects (fmt.Println):**

- Always writes to the console.
- You cannot easily **assert printed output**.

**3. No Interface Abstraction:**

- Logic is tightly coupled to real packages (time, fmt)
- No way to replace with stubs/mocks.

**4. Non-deterministic:**

- Running tests in the morning vs afternoon may produce **different results**.

# Steps to make it testable

**Identify hard deps** → time, DB, I/O, globals

**Extract interfaces** → wrap deps (e.g. Clock, Mailer)

**Inject dependencies** → pass via constructor/params

**Separate side-effects** → keep I/O at edges, core logic pure

**Make deterministic** → use fakes/stubs in tests

**Write small units** → focus on inputs/outputs, not internals

# What is Mutation Testing?

**Mutation testing** (or *mutation analysis* or *program mutation*) is used to design new software tests and evaluate the quality of existing software tests.

**Idea**: Intentionally introduce small code changes (mutants) to see if tests fail

**Goal**: Measure test effectiveness, not just coverage

**Outcomes**:

**Killed** (tests fail) = good

**Survived** (tests still pass) -> weak tests

Metric: **Mutation Score** = killed / total mutants



QA

# Why do this in TDD?

- Forces **better assertions** (behavior, boundaries, error paths)
- Reveals **false positives** where tests pass but logic is unprotected
- Encourages small, **well-specified** units and refactoring with confidence

# Tooling in Go - Gremlins

- **Modern mutation testing tool for Go**
- **Mutates source, re-runs tests, reports killed/survived mutants**

***Install***

**go install github.com/go-gremlins/gremlins/cmd/gremlins@latest**

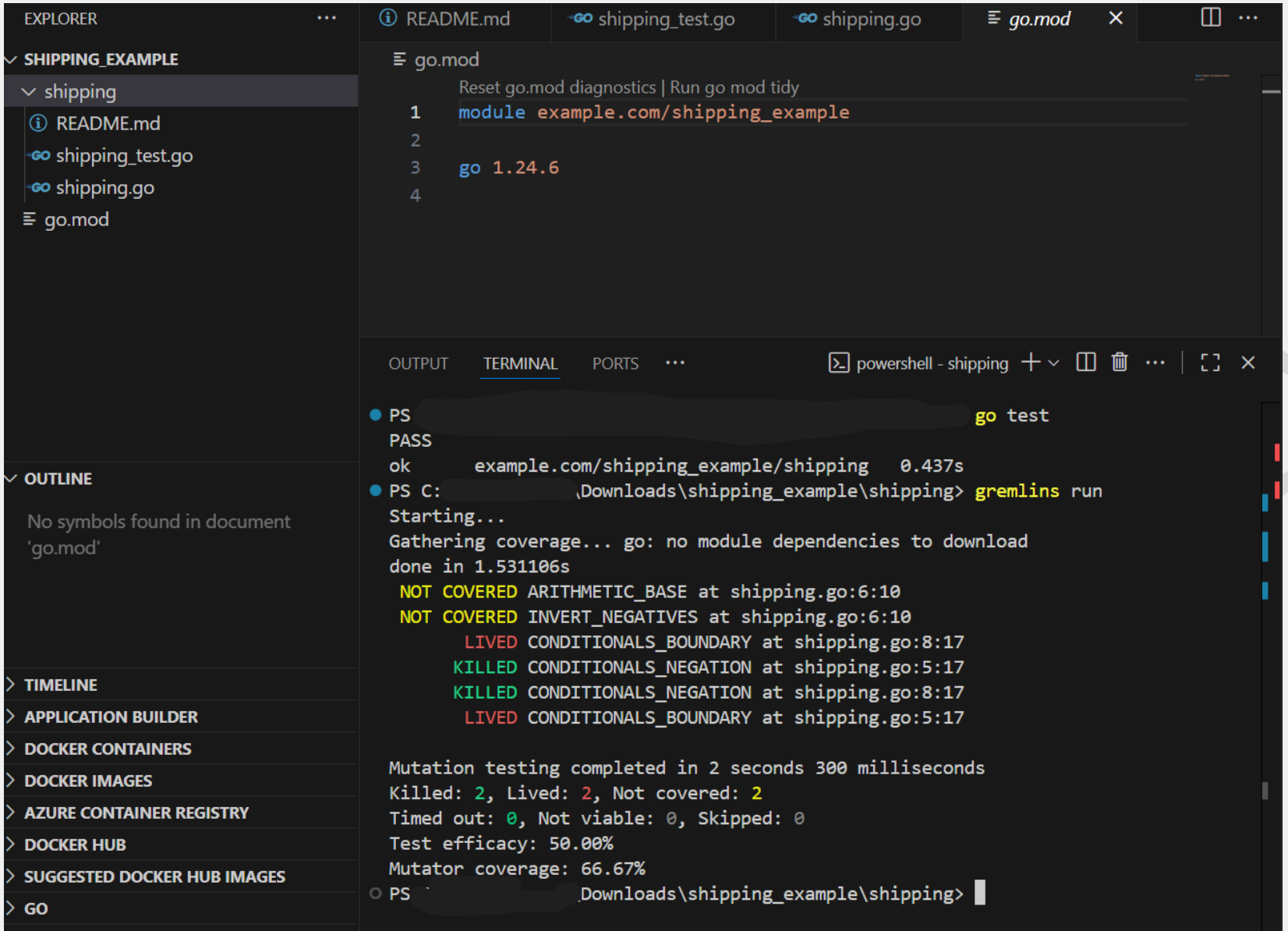
***Run***

**gremlins unleash**



QA

# Example Output



# How to fix Surviving Mutants

- **Tighten assertions**

Assert exact outputs, error values, and **side-effects** (state changes, calls)

- **Add boundary tests**

Empty/zero, min/max, negatives, duplicates, overflow, timeouts

- **Cover branches**

True/false paths, error returns, early exits, default cases

- **Reduce over-mocking**

If behavior is externalized, assert **interactions** (called/args), not just “no error”

- **Make code testable**

Inject time/IO/random; remove hidden globals; separate side-effects

- **Remove dead code**

If a mutant survives in truly unused logic, delete or quarantine it

# TDD + Mutation Workflow

1. Red → Green → Refactor (write tests first)
2. Measure coverage (go test -coverprofile)
3. Run Gremlins (gremlins unleash)
4. Fix survivors (add/strengthen tests; refactor for testability)
5. Repeat until mutation score is acceptable for the module



QA

# Example – Shipping Cost Calculator

Q7 – **Shipping Cost Calculator (TDD & Mutation testing)**

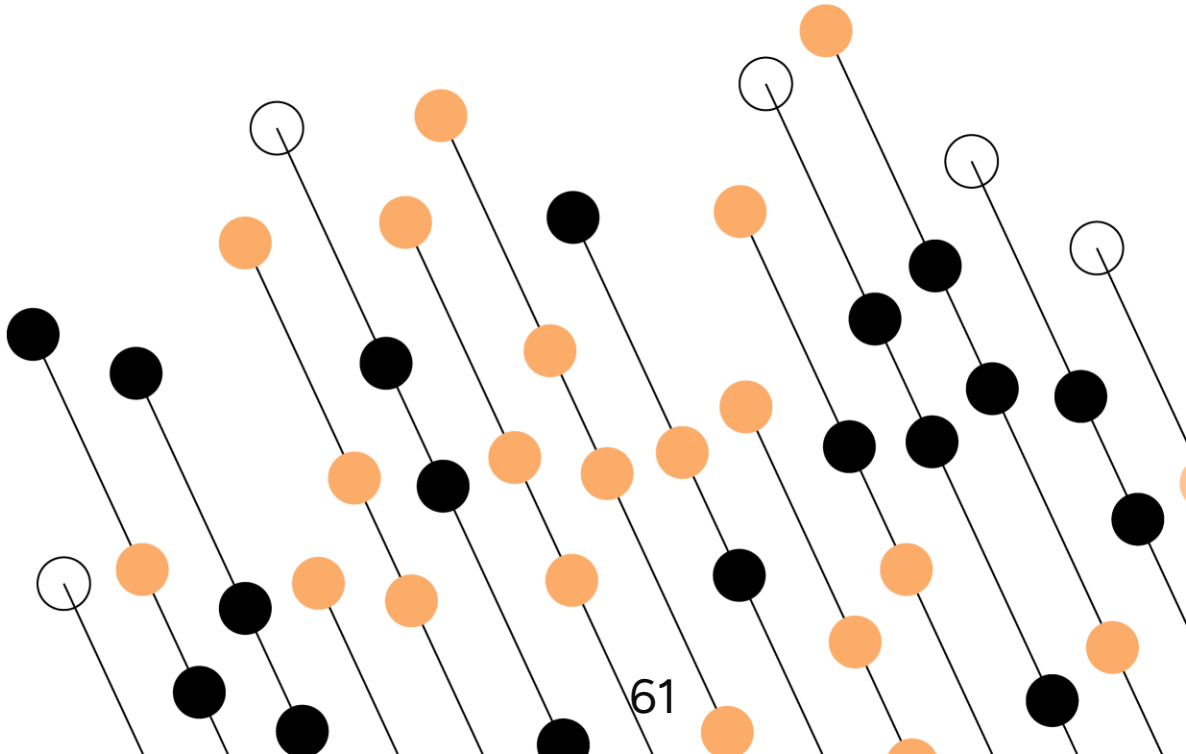
**Business Rules**

1. **If the order value ≥ 1000, shipping is free (₹0).**
2. **If the order value < 1000, apply a flat shipping fee of ₹50.**
3. **If the order value < 0, it is an invalid input and should return -1.**

**Requirements**

- Implement the function CalculateShipping(orderValue int) int.
- Write **unit tests** that cover all rules above.
- Run **mutation testing (Gremlins)** to check if your test suite catches all injected mutants.
- Improve your tests if mutants survive.

# Advanced Structures



# Functions as Values

- In Go, functions are first-class citizens
- You can assign them to variables, pass them as arguments, return them



```
double := func(x int) int { return x * 2 }

fmt.Println(double(3)) // 6
```

# Closures

- A closure is a function that captures variables from its surrounding scope
- Useful for building reusable, configurable behaviors



```
func MultiplierBy(k int) func(int) int {
    return func(x int) int { return x * k } // captures k
}
times10 := MultiplierBy(10)
fmt.Println(times10(3)) // 30
```

# Lambdas

- Anonymous functions (defined inline)
- Great for short, single-use behaviors
- Often used directly in tests or when passing a function



```
nums := []int{1,2,3}
squares := Map(nums, func(x int) int { return x * x })
```

# Relevance to TDD

**Inject Behavior**
- Pass rules/strategies as functions instead of hardcoding
- Makes code flexible & testable

**Test Different Scenarios Easily**
- Swap lambdas/closures in tests to cover edge cases
- No need to rewrite production code

**Composable & Reusable**
- Small, pure functions can be combined for complex logic
- Each function is unit-testable in isolation

**Drive Design with TDD**
- Write a failing test → add a new function rule → pass test
- Encourages incremental, safe growth of features

# Example – Advanced Structures

Q8 – **Pricing Engine with Discount Rules**

Build a **checkout system** where discounts are applied via functions.

**Requirements:**

- Define Item with Name, Price, Qty
- Implement Subtotal(items []Item) int
- Implement Checkout(items []Item, rules ...DiscountRule)
- Define type DiscountRule func(items []Item, subtotal int) int
- Implement closures:
  - NewPercentOffAbove(threshold int, rate float64)
  - NewBOGO(itemName string)
- Implement FilterItems using a function as a value

# Example – Case Study

## Q9 – Bank Management

The service should support:

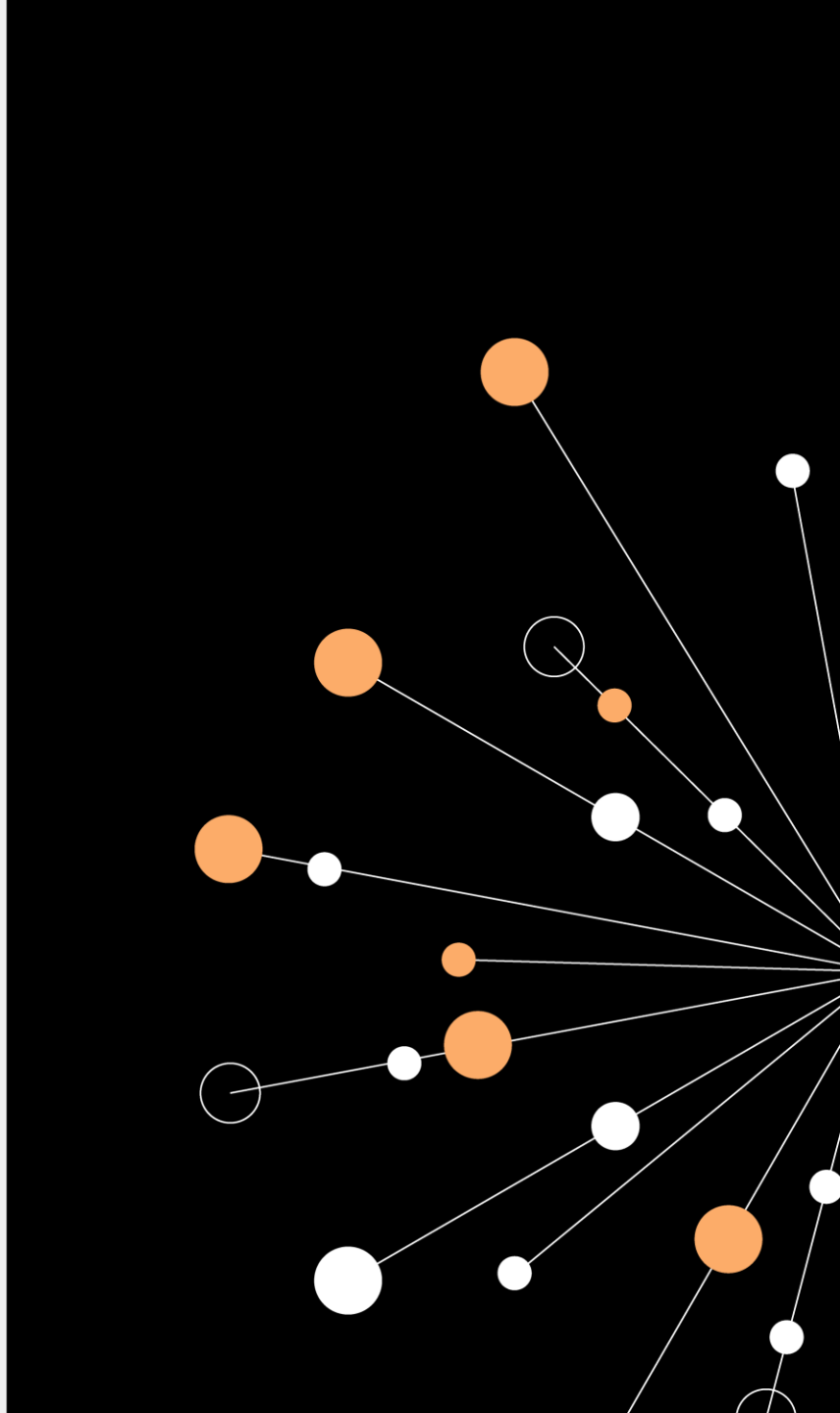
1. **Account Creation**
   - Create a new bank account with an initial balance.
2. **Deposit Money**
   - Add funds to an account.
3. **Withdraw Money**
   - Withdraw funds from an account.
   - Prevent withdrawals when funds are insufficient.
4. **Check Balance**
   - Retrieve the current account balance.
5. **Transaction Logging**
   - Log every transaction (deposit/withdrawal) through a `Logger` interface.
   - Use **GoMock** to mock and verify logging behavior in tests.

# 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