Unit Testing

Why do we test?

The goal of testing is to ensure that the software that we produce meets our objectives when deployed into the environment where it will be used.

How do we do this?

- We run our software under controlled conditions i.e. specific inputs and environments, and check that expected & actual outcomes match.

Benefits of testing?

- Improve confidence in your software (cannot "guarantee correctness").
- Produce an improved design, usually as a by-product of having written tests. *The process of writing tests forces us to structure our code more thoughtfully*.
- Occasionally find defects, deficiencies or design flaws from our tests.

How do we test?

We create a range of tests to provide "complete" coverage across our code. We can identify three levels of testing:

1. **Functional (system) tests**: testing functionality from the perspective of the user; end-to-end feature testing.

2. **Integration tests**: testing across multiple classes or functional units, to check interaction between objects.

3. **Unit tests**: tests operating at the class level (or smallest functional unit), which are check low-level interfaces and behaviours.
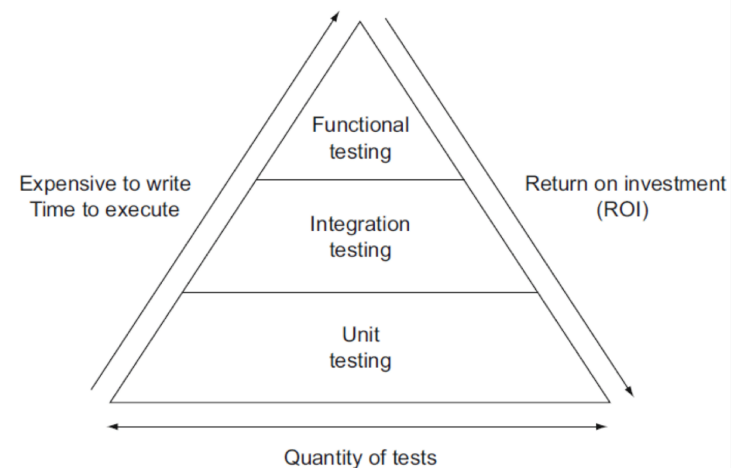
When should we test?

Traditional views suggest that testing should be done after implementation. This is a poor approach!

Testing should be done *early* in the development process.
- It gives you more opportunity to incorporate the feedback from testing.
- It's much cheaper to "fix bugs" earlier in the process.

Tests should be matched to the stage of the development process:
- **Unit Tests**: done *during implementation*, when you are working on a class.
- **Integration Tests**: done *after implementation*, when you want to ensure that classes work together (on all platforms).
- **System Tests**: done when features are *complete and merged*, to ensure that the system continues working.

Test-Driven Development (TDD)

Promoted by Kent Beck around 2002 as an **Extreme Programming (XP) practice**.

- The basic idea is that you write tests *before* writing the corresponding implementation code.

TDD development cycle

1. Define an interface or specification for your class or module.
2. Write a test against that interface.
3. Write the implementation code that causes the test to pass.
4. Repeat until completed.
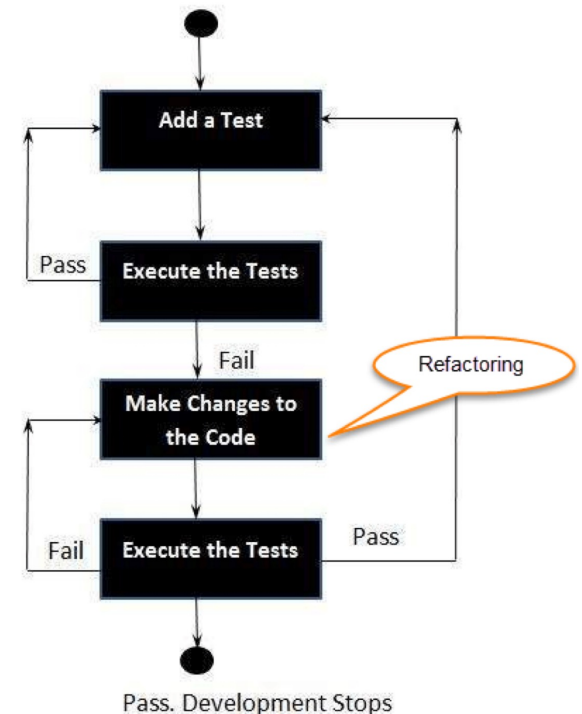
Advantages of continuous testing

There are some clear benefits of TDD:

- **Early bug detection**. You are building up a set of tests *as you write code*.
- **Better designs**. Making your code testable often means improving your interfaces, having clean separation of concerns, and cohesive classes. Testable code is by necessity better code.
- **Confidence to refactor**. *Refactoring* is the process improving your code incrementally over time. You can only do this if you can easily verify that you haven't "broken anything" in the process. TDD helps you have the confidence to refactor!
- **Simplicity**. Code that is built up over time this way tends to be simpler to maintain and modify.

Our initial goal is to generate unit tests

Unit tests are meant to exercise the interface of a single class or module.

- Unit tests should be very quick to execute and report results.
- They should return consistent results from a specified input.
- They should be integrated into our development workflow, so that they are routinely executed. i.e. they need to be automated.

- Unit testing is behavioural testing i.e., test how classes behave across a range of valid and invalid inputs.

Unit testing configuration

Setup the Kotlin test framework for unit testing.

Installing test dependencies

Junit is the standard test framework for Java testing. It's possible to use it with Kotlin, but Junit is not supported equally well across platforms.

Kotlin has a cross-platform test framework which is similar, but will work across all Kotlin platforms e.g., iOS, Android, desktop). We'll use it instead.

Make sure that you have these lines in your `build.gradle.kts` file.

```
dependencies {
    testImplementation(kotlin("test"))
}
tasks.test {
    useJUnitPlatform() // we'll use Junit as the test runner when possible
}
```
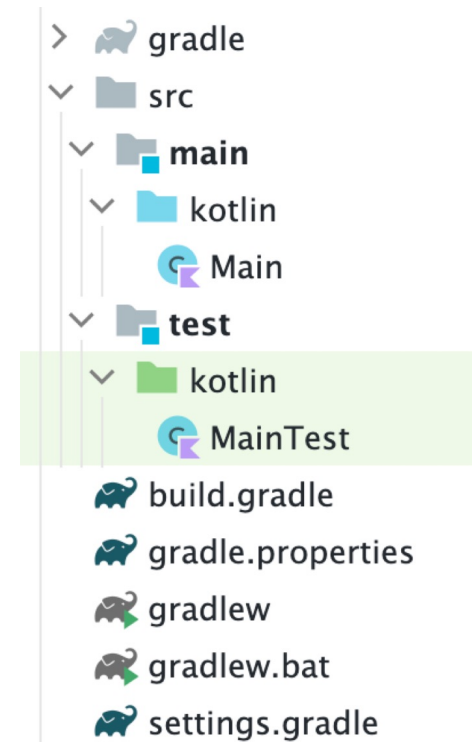
Unit tests are just functions

Unit tests are just Kotlin classes and functions that check inputs and outputs for what they are testing.

- Unit tests should be placed under `src/test/kotlin`.

- It's best practice to have one test class for each class that you want to test. e.g., classes `Main` and `MainTest`.

- Unit tests are automatically executed with `gradle build` or can be executed manually with `gradle test`.

```
$ gradle build
BUILD SUCCESSFUL in 928ms
8 actionable tasks: 8 up-to-date // this includes tests

$ gradle test
BUILD SUCCESSFUL in 775ms
3 actionable tasks: 3 up-to-date
```

> gradle
v src
  v main
    v kotlin
      Main
  v test
    v kotlin
      MainTest
  build.gradle
  gradle.properties
  gradlew
  gradlew.bat
  settings.gradle

A Simple Unit Test

1. Create a class to test under src/main/kotlin.

```kotlin
class Sample() {
    fun sum(a: Int, b: Int): Int {
        return a + b
    }
}
```

2. Create a test class under `src/test/kotlin`. Add functions as tests.

```kotlin
import kotlin.test.Test
import kotlin.test.assertEquals

internal class SampleTest {
    @Test
    fun testSum() {
        private val testSample: Sample = Sample()
        assertEquals(42, testSample.sum(40, 2))
    }
}
```

Running tests
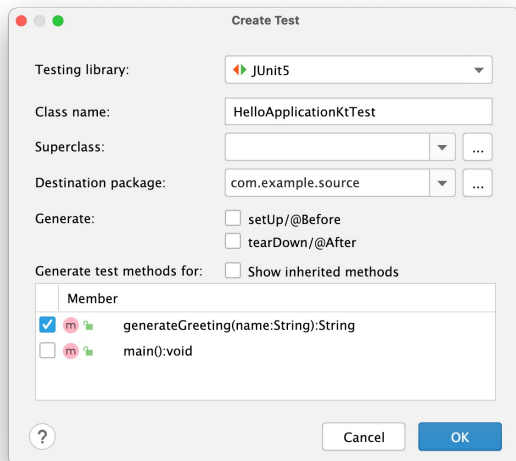
Press Cmd-N to generate a new test for a selected class.

```
1   package com.example.source
2
3   import org.junit.jupiter.api.Test
4   import org.junit.jupiter.api.Assertions.*
5
6   internal class HelloApplicationKtTest {
7
8       @Test
9       fun generateGreeting() {
10          val expected = "Hello world!"
11          assertEquals(expected, generateGreeting( name: "world"))
12      }
13  }
```

In the test class, you can execute a particular test by clicking the Run icon in the gutter.

Assertions

We call utility functions to assert how the function should successfully perform.

Function	Purpose
assertEquals	Provided value matches the actual value
assertNotEquals	The provided and actual values do not match
assertFalse	The given block returns false
assertTrue	The given block returns true

Test Annotations

The @Test annotation tells the compiler that this is a unit test function. The `kotlin.test` package provides annotations to mark test functions, and denote how they are managed:

Annotation	Purpose
@AfterTest	Marks a function to be invoked after each test
@BeforeTest	Marks a function to be invoked before each test
@Ignore	Mark a function to be ignored
@Test	Marks a function as a test

Writing Unit Tests

What are the characteristics of well-written tests?

Unit Test Characteristics

A unit test is a test that meets the following three requirements:
1. Verifies a single unit of behaviour,
2. Does it quickly, and
3. Does it in isolation from other tests.

Unit tests are the lowest-level tests that you can write:

- Tests should be small and quick to execute and return results.

- Each test focuses on a specific class or component, tested in isolation.

- Tests cannot have dependencies on other tests! i.e., can execute in any order.

- As an author, favour many small tests that each check a single thing over monolithic tests.

If a test exercises more than a single class, it's not a unit test.

Unit Test Composition

Every unit test should be a separate function, with the following steps:

1. <u>Arrange</u>:
- Setup the conditions for your test.
- Initialize variables, load data, setup any dependencies that you might need.
- Do NOT reuse anything from a different test.

2. <u>Act</u>:
- Execute the functionality that you want to test and capture the results.

3. <u>Assert</u>:
- Check that the actual and expected results match.
- Use asserts appropriately - see next page.

```kotlin
class CalcTest {
@Test
fun validPlus() {
    val input = arrayOf("1", "+" , "2")
    val results = Calc().calculate(input)}
    assertEquals(3, results)
}

@Test
fun invalidPlus() {
    val input = arrayOf("1", "+", "2")
    val results = Calc().calculate(input)
    assertNotEquals(5, results)
}

@Test
fun insufficientArguments() {
    try {
        val input = arrayOf("1", "+")
        Calc().calculate(input)
    } catch (e:Exception) {
        assertTrue(true)
    }
}
```

Test valid input conditions.
Create a unit test like this for each operation or function.

Test invalid input conditions.
Create a unit test like this for each operation or function to ensure that you handle input errors correctly. Choose representative values (or important outliers)

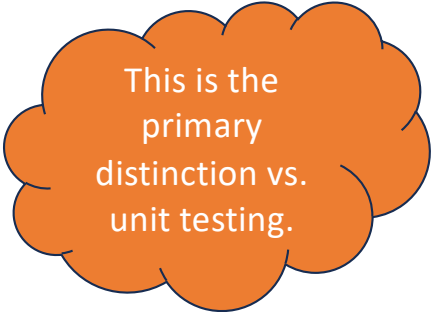Special-purpose unit test to check a specific error condition.

Integration Tests

The next step; handling more complex interactions.

Integration tests

"Unit tests are great at verifying business logic, but it's not enough to check that logic in a vacuum. You have to validate how different parts of it integrate with each other and external systems: the database, the message bus, and so on." — Khorikov (2020).

- A **unit test** is a test that verifies a single unit of failure, in isolation.
- An **integration test** is a test with a broader scope.
  - It checks multiple potential units of failure.
  - **Seeks to understand the interaction between components**.
  - **Tests component *dependencies*.**

This is the primary distinction vs. unit testing.

What is a dependency?

- When you are examining a software component, we say that your component may be dependent on one or more other software entities to be able to run successfully. e.g. a library, or a different class, or a database. Each of these represents code that affects how the code being tested will execute.

- **We often call the external software component or class a dependency**. That word describes the relationship (classes dependent on one another), and the type of component (a dependency with respect to the original class).

- *A key strategy when testing is to figure out how to control these dependencies, so that you're exercising your class independently of the influence of other components.*

Dependencies

Managed vs. unmanaged dependencies. Distinction between dependencies that we control (managed), and those that may be shared (unmanaged).

- A managed dependency suggests that we directly control the state.
- e.g., A database could be single-file and used only for your application (managed) or shared among different applications (unmanaged).

Internal vs. external dependencies. Distinction between running in the context of our process (internal) or out-of-process (external).

- *External intrinsically means unmanaged (and usually untrusted)*.
- e.g., A library is internal. If statically linked, we manage its state.
- e.g., An external library is external and probably unmanaged.

An unmanaged dependency cannot be tested directly.

- How can we trust that its state isn't changing independently?

Test Doubles (aka Mocks)

How do you test unmanaged dependencies?

1. You test to the interface and not the concretion.

2. You can also create a "mock" or a test double that substitutes for the concretion in testing.

A **mock** is a fake object that holds the expected behaviour of a real object but without any genuine implementation. For example, we can have a mock File System that would report a file as saved but would not actually modify the underlying file system.

Mocks, or test doubles, remove dependencies and allow for controlled testing. They are extremely useful!

Mocking & Dependency Injection

[Dependency injection](#) is the practice of supplying dependencies to an object in its argument list instead of allowing the object to create them itself.

Problem: Here's a class that manages the underlying database connection. How do you test it separately from the database?

```
class Persistence {
  val repo = UserRepository() // Create the required repo instance

  fun saveUserProfile(val user: User) {
    repo.save(user)
  }
}

val persist = Persistence()
persist.saveUserProfile(user) // saves using the real hard-coded database
```

Example: Mock DB

Solution: change the Persistence class so that we pass in the dependency.
This allows us to control how the repository is created and even replace it
with a mock for testing.

```
class Persistence(val repo: IUserRepository) {    // pass in the repo
    fun saveUserProfile(val user: User) {
    repo.save(user)
  }
}
class MockRepo : IUserRepository {
    // body with functions that mirror how the repo would work
    // but simpler/fake implementation
}

val mock = MockRepo()
val persist = Persistance(mock)
persist.saveUserProfile(user)   // save using the mock database
```

References

- JetBrains. 2025. kotlin-test documentation.
- Khorikov. 2020. Unit Testing Principles, Practices, and Patterns. Manning. ISBN ISBN 978-1617296277.