Unit Testing

CS 346: Application Development

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 in which it will be used.

Why do we test?

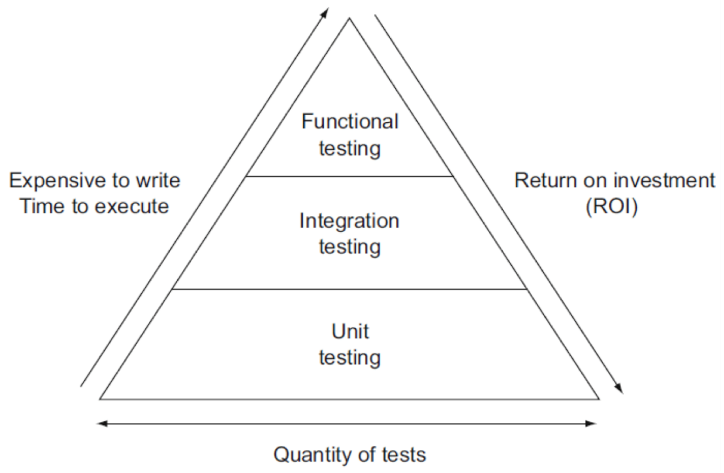
- Improve your confidence in the correctness of your software (we cannot “guarantee correctness”).
- Ensure that you are handling errors properly, which will result in a better user experience and a more stable application.
- Produce an improved design, usually as a by-product of having written tests. *The process of writing tests forces us to structure our code more thoughtfully*.
- Identify deficiencies and flaws in both design and implementation.

Layered Testing

We should produce a range of tests that can confirm that our system is working as expected.

We can identify three types of tests:

1. **Unit tests**: tests operating at the class level (or smallest functional unit), which are meant to check low-level interfaces.
2. **Integration tests**: testing across multiple classes or functional units, to check interaction between objects.
3. **Functional (system) tests**: testing functionality from the perspective of the user; end-to-end feature testing.



When should we test?

Traditional views suggest that testing should be done after implementation. This is a poor approach; testing is more useful when done *earlier* in the process.

- It gives you more opportunity to incorporate the feedback from testing.
- It's much cheaper to "fix bugs" earlier in the process.

Different tests suit for different parts 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.

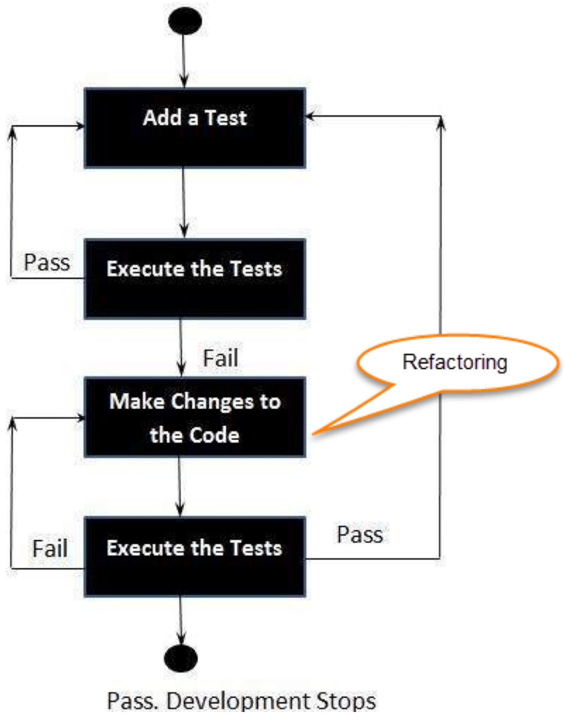
Test-Driven Development (TDD)

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:

- **Early bug detection**. You are building up a set of tests that can verify that your code works as expected.
- **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**. *What is refactoring*? The process is 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.
- **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.

- *How many unit tests should you have?* As many as required to cover your critical classes and workflows. (You will NOT get 100% code coverage, despite best intentions).

Using Junit

Setting up unit tests for a Kotlin project

Installing JUnit

To manage our tests, we're going to use **JUnit 5,** a popular testing framework. We'll let Gradle manage JUnit for us as a project dependency: add these lines to your project `build.gradle.kts` file.

The "test" dependency will pull in Kotlin test classes for JUnit.

```
dependencies {
    testImplementation(kotlin("test"))
}
tasks.test {
    useJUnitPlatform()
}
```

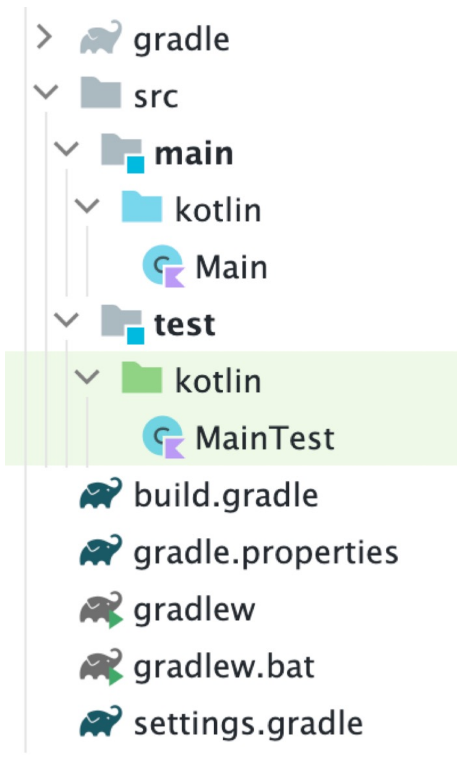
Unit tests are just functions

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.

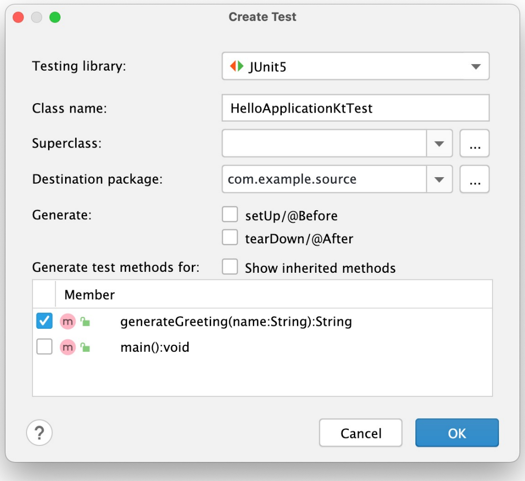
```kotlin
class Sample() {
    fun sum(a: Int, b: Int): Int {
        return a + b
    }
}
```

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))
    }
}
```

Sidebar: IntelliJ Tips



Testing library: JUnit5

Class name: HelloApplicationKtTest

Superclass:

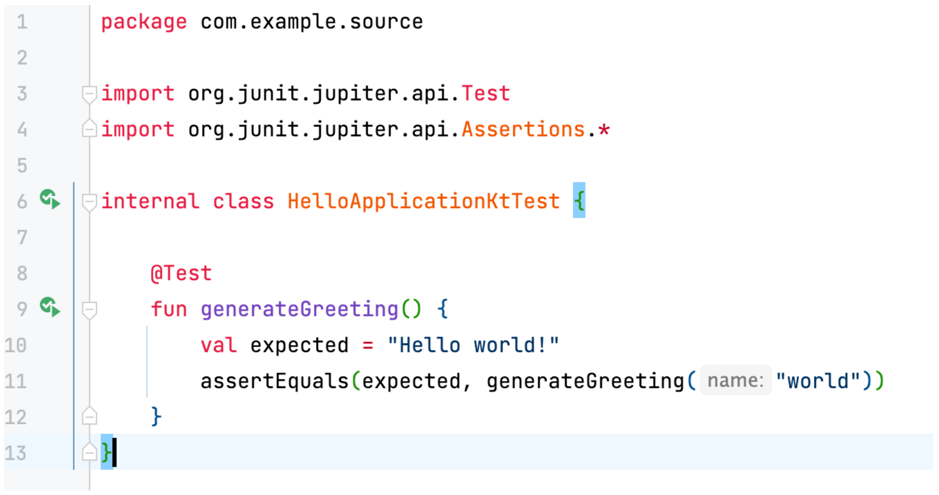
Destination package: com.example.source

Generate: [ ] setUp/@Before [ ] tearDown/@After

Generate test methods for: [ ] Show inherited methods

	Member
✓	generateGreeting(name:String):String
	main():void

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



```kotlin
package com.example.source

import org.junit.jupiter.api.Test
import org.junit.jupiter.api.Assertions.*

internal class HelloApplicationKtTest {

    @Test
    fun generateGreeting() {
        val expected = "Hello world!"
        assertEquals(expected, generateGreeting( name: "world"))
    }
}
```

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. Arrange:

- 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. Act:

- Execute the functionality that you want to test and capture the results.

3. Assert:

- Check that the actual and expected results match.
- Use asserts appropriately - see next page.

```kotlin
import kotlin.test.Test


class MainTest {

    @Test
    fun saveFiles() {
        // FILE 1
        val f1 = "file1.txt"         // arrange
        val file1 = File(f1)
        val status1 = file1.save()  // act
        assert(status1 == FILE.OK)  // assert

        // FILE 2
        val f2 = "file2.txt"         // arrange
        val file2 = File(f2)
        val status2 = file2.save()  // act
        assert(status2 == FILE.OK)  // assert
    }
}
```

```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.

Testing > Integration Tests

CS 346: Application Development

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.
  - Tests components that have dependencies between them.
  - Seeks to understand the *interaction* between components.

Area of Responsibility

- **Unit tests** primarily test the domain model, or business logic classes.
- **Integration tests** focus on the point where these business logic classes interact with external systems or dependencies.
- Code that we shouldn't bother testing:
  - **Trivial code** is low complexity, and typically has no dependencies or external impact so it doesn't require extensive testing.
  - **Overcomplicated code** likely has so many dependencies that it's nearly impossible to test.

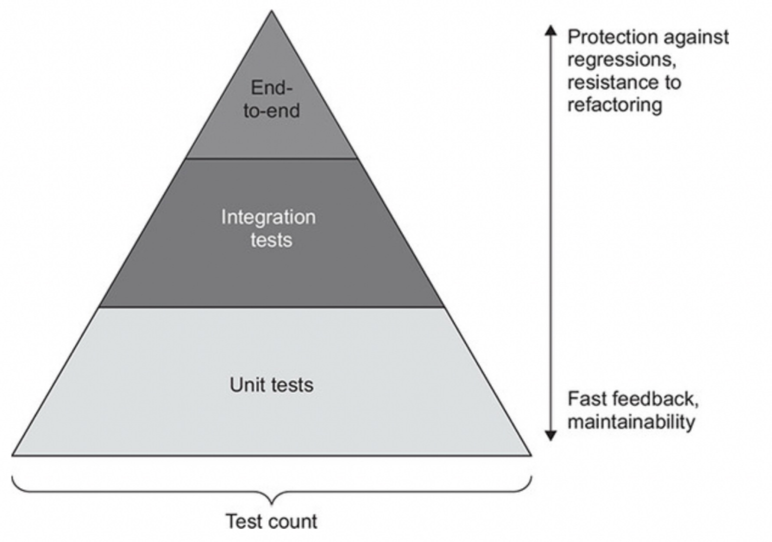
Integration Test coverage

You should have fewer integration tests than unit tests; they are more complex.

To determine what to test, identify a "**happy path**": an execution path that will be your first (aka main) integration test.

- It should reflect the most used execution path. Ensure that common functionality works before testing more obscure functions.
- This main integration test should exercise all external dependencies i.e. libraries, interfaces, database.

The primary purpose of integration tests is to **exercise dependencies** (on libraries, other classes).


End-to-end
Integration tests
Unit tests
Protection against regressions, resistance to refactoring
Fast feedback, maintainability
Test count

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.*

Unmanaged vs. Managed Dependencies

- **Managed vs. Unmanaged dependencies**. There is a difference between those that we control directly (managed), vs. those that may be shared with other software (unmanaged). A managed dependency suggests that we control the state of that system.
  - 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**. Running in the context of our process (internal) or out-of-process (external). External dependencies tend to be unmanaged.
  - e.g., A library is internal. A database, as a separate process is always external.
- An unmanaged dependency means that we do not control its state and we may not be able to test that specific component or dependency. The best we could do it test our interface against it, since we may not be able to trust the results of any actions that we take against it.
  - ***We cannot test unmanaged dependencies**. We also tend to be limited in our ability to test external systems, since we do not manage their state.*

Test Doubles (aka Mocks)

If you cannot test an unmanaged dependency, then you need to test "around it". We can do this with test doubles (mocks).

- 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.
- The value in a mock is to remove dependencies and allow controlled testing.

Strategy: Extract Interfaces

- One important recommendation is that you introduce **interfaces** for out-of-class dependencies. e.g., you may see code like this:

```
public interface IUserRepository
public class UserRepository : IUserRepository
```

- This is common when testing, even in cases when that class may represent the *only* realization of an interface. This allows you to easily **write mocks against the interface**, where it's relatively easy to determine what expected behaviour should be.
- Replacing concretions with interfaces is one of our SOLID principles.

Strategy: 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 the `saveUserProfile()` method 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) // save using the real database
```

Example: Mock DB

To reduce coupling, we could instead change our Persistence class so that we pass in the dependency. This allows us to control how it is created, and even replace the `UserRepository()` with a mock.

```
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
```

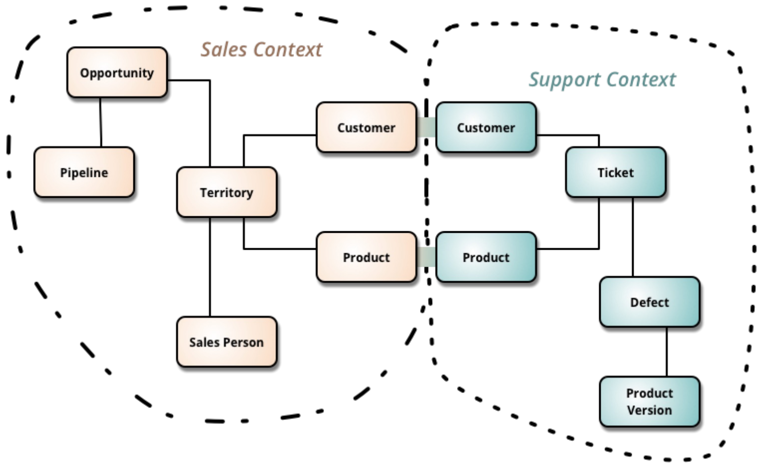
Tip: Integration Tests

1. Keep your code simply and reduce abstractions when you can.

- Many abstraction layers will make it very difficult to write integration tests for your code. e.g., Java libraries and "class bloat".
- Try and keep the overall number of architectural layers relatively small:
  - UI, controller, persistence.
  - Domain model, application services, infrastructure.

2. Make domain boundaries explicit

- You should be testing end-user functionality. Design classes to emphasize different problem domains.
  - e.g., user-creation/editing tests; note-creation and note-deletion.



https://martinfowler.com/bliki/BoundedContext.html