Testing

CS 346 Application Development

https://student.cs.uwaterloo.ca/~cs346

Contents

Introduction 3
- Why Test? 4
- Types of Testing 5
- Test Frameworks 7

Unit Tests 8
- What is a Unit Test? 9
- Writing Tests 11
- Running Tests 15

Integration Tests 16
- What is an Integration Test? 17
- Writing Tests 18

Dependencies 19
- What is a Dependency? 20
- Types of Dependencies 21
- Test Doubles 23

TDD 26
What is it? 27
Bibliography 29

Introduction

Why Test?

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

Purpose of testing?

- Find problems aka bugs and address them before shipping.
- Find design flaws and incrementally improve the product.
- Intended to demonstrate "fitness" of our software.

Benefits of testing?

- Improve confidence that you have met your goals.
- Occasionally find defects, deficiencies or design flaws from our tests.
- Produce an improved design, sometimes as a by-product of testing.

Types of Testing

Business Facing

Guide Development	Functional examples Examples – third party integrations Contract testing on APIs	Exploratory testing Usability Monitoring Observability Testing in Production	Critique the Product
	Unit testing Static analysis Component integration Test doubles for external integrations Data integrity	Recoverability Performance Load/Stress Reliability/Safety Infrastructure	

Technology Facing

Figure 1: The testing quadrants demonstrate different concerns. We'll focus on the bottom-left, including unit tests and test doubles. [1]

Types of Testing

We will focus on developer focused tests.

- Unit testing, or class-level tests.
- Component integration, or features that span multiple classes.
- Test doubles for external integrations.
- Contract testing for APIs, where applicable.

Our goals:

1. We want **broad coverage**. Every class and every interface should have some tests against it. Start with common scenarios, and expand over time.
2. Our tests should be **automated**. They should run automatically without manual intervention, and report the results. You should only perform manual testing if there is no other choice!
3. Tests should be **easy to run**. There should be no manual steps required to ‘setup tests’. Make it easy so that there are no roadblocks to testing.

Test Frameworks

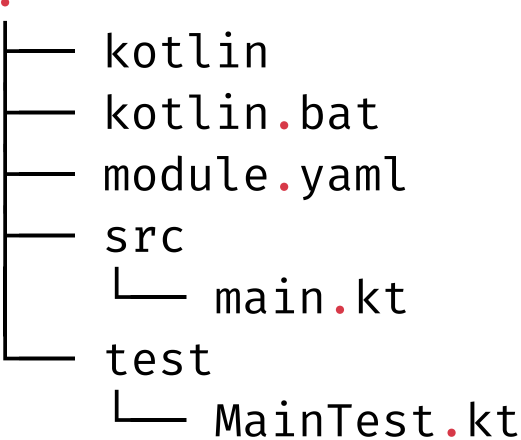
There are specialized software packages that are designed to help automate and simplify testing e.g., JUnit for Java, Catch2 for C++.

Kotlin has a cross-platform test framework: kotlin.test.

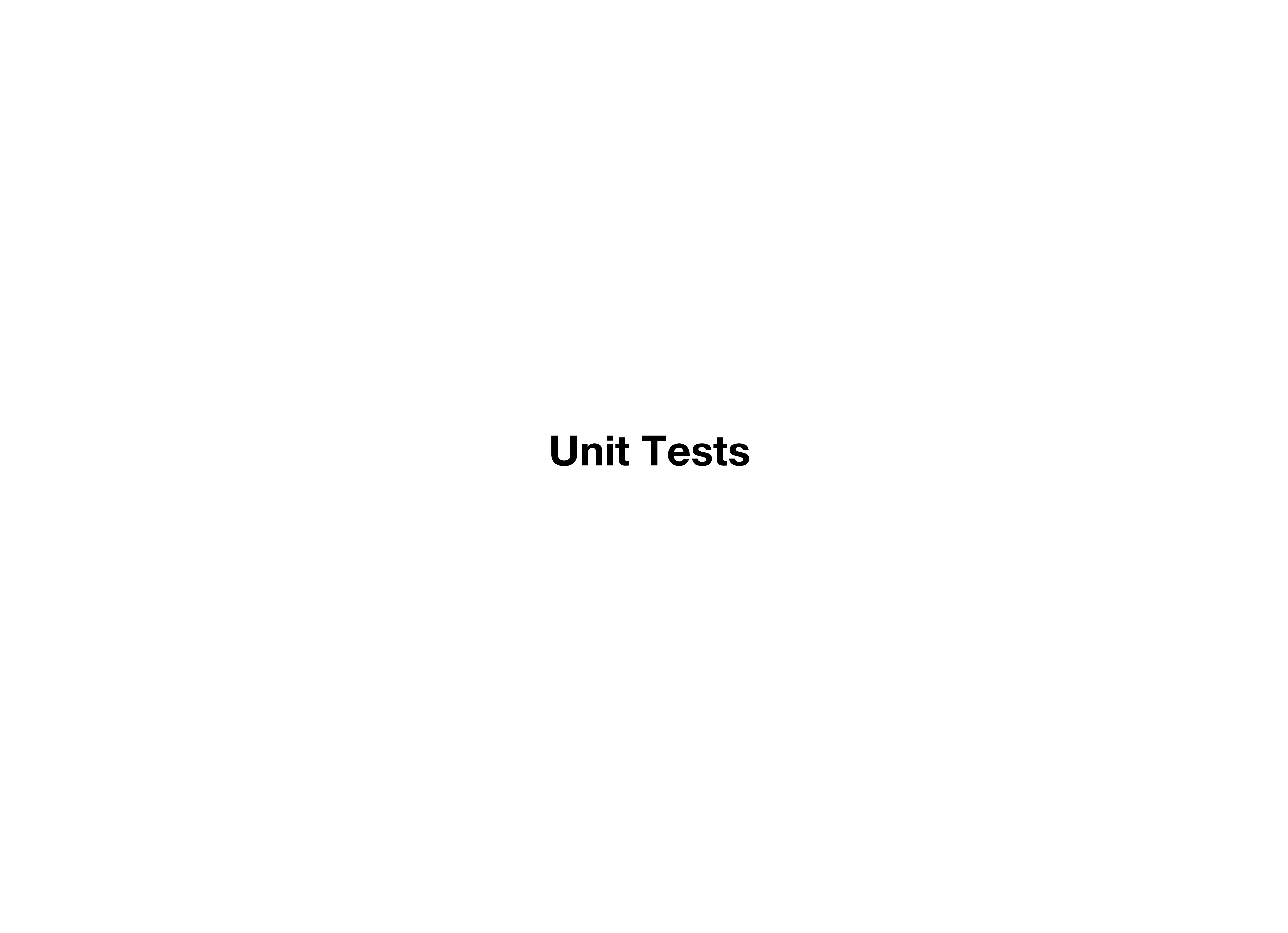
- Similar to Junit (Java standard) but it works across all platforms.
- Make sure that you import the dependencies in your `module.yaml` file.

Our tests will ultimately be Kotlin classes and functions that check that inputs produce expected outputs for a range of values and conditions.

- Unit tests should be placed under `/test` folder.



```
.
├── kotlin
├── kotlin.bat
├── module.yaml
├── src
│   └── main.kt
└── test
    └── MainTest.kt
```


Unit Tests

What is a Unit Test?

A **unit test** is a test that verifies a single class or unit of failure in isolation.

Unit tests are typically designed to verify that your class reacts in an expected way to a specific input. i.e. if I call function X with input Y, result Z is returned.

Mechanically, unit tests are just Kotlin classes and functions that you create.

- Keep them in separate files, under a top-level `test` folder.
- Create a unit test file that corresponds to each class that you test.
- Unit tests are run automatically by the build system when you build.

Tip

You should have unit tests for every class that you create.

What is a Unit Test?

Here's a simple example of creating a single-class function and a corresponding unit test.

1. Create a class to test under `/src/Calculator.kt`.

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

2. Create a test class under `/test/CalculatorTest.kt`.

```
internal class CalculatorTest {
  @Test
  fun testSum() {
    private val calc = Calculator()
    assertEquals(42, calc.sum(40, 2))
  }
}
```

Writing Tests

A unit test should meet the following requirements:

1. Verifies a single class
   - Unit tests only interact with a single class.
   - Write tests to ensure that you check all valid states for that class.
2. Executes quickly
   - Unit tests are automated, and will run every time you build.
   - You may have hundreds of unit tests, so keep them small and simple.
3. Executes in isolation from other tests
   - Tests should not rely on the outcome of a previous test for their execution.
   - There is no guarantee that they will run in any particular order!

Tip

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

Writing Tests

Three Steps

Every unit test should be a separate function, with the following steps [2].

1. Arrange
   - Setup the conditions for your test.
   - Initialize variables, load data, setup any dependencies.
   - 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).

Writing Tests

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

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

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

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

Assertions

We call assertions to state how the function should successfully perform.

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

Annotations

The `@Test` annotation tells the compiler that this is a unit test function. The kotlin.test package provides additional annotations:

Function	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

Running Tests

Running unit tests is simple. All of these options work:

1. Beside each unit test, there is a Play button in the gutter of the IDE. Click on Play to run an individual test.
    - You can also click on Play beside the enclosing class to run all tests.

2. Unit tests can also be run by building the project. All unit tests will be executed automatically after the build, and the results posted.

Integration Tests

What is an Integration Test?

"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" [2].

- Khorikov, Unit Testing Principles

A **unit test** is a test that verifies a single class or 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.

Writing Tests

They look exactly like unit tests, except with a broader scope.

You still need to ensure that you are using the three-A approach:

- Arrange
- Act
- Assert

Integration tests do not replace unit tests

- Having integration tests doesn’t meant that you skip writing unit tests.
- Continue to write unit tests for all classes, just add additional integration tests to cover cases where two or more classes overlap.

Dependencies

What is a Dependency?

Your application will be composed of many components (classes).

Some of your components may be dependent on one or more other entities e.g., you may use an external library, or store data in an external database. We call the external software component or class a **dependency** of the component that you are examining.

You may not have control over how that entity is working.

- How do you test something that you don't control?

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

Your strategy may be different based on the type of dependency.

Types of Dependencies

There is a distinction between dependencies that we control (`managed`), and those that may be shared (`unmanaged`) [2].

Managed Dependencies

With a managed dependency, you directly control the state of the dependency, and you trust that there are no other external influences on its state. e.g.,

- A library that you import. It's "dedicated" to your application, and in this case, running in-process with your application.
- An inline database that is dedicated to your application.

In both of these examples, you don't have access to the underlying source code, but you can still trust that any state changes during testing will be predictable.

- You can and should write unit tests, but limit them.
- Your assumptions are different. You are no longer testing how well your code is working, but rather how that dependency reacts to state changes.

Types of Dependencies

Unmanaged Dependencies

With an unmanaged dependency, you don't have any guarantees that you have exclusive access to a dependency. e.g.,

- A web service which other applications can access.
- A shared database where other users/processes have the ability to modify your data, or the way that the system is running.

In these examples, you cannot fully trust the dependency.

- The system can fail for reasons unrelated to changes to your source code.
- Not recommended to unit test in this case.

Tip

So what do you do? There are other types of tests e.g., acceptance tests that are structured around these assumptions. It's beyond the scope of this course or a single lecture like this.

Test Doubles

Although we should not test unmanaged dependencies directly, we still need to know that any component that interacts with them is working properly e.g., a database class that interacts with an external database.

How do you test the class that interacts with an external dependency? You replace the thing that you *don't* trust with something that you *can* trust.

A mock (test double) is a fake object that holds the expected behaviour of a real object but without any actual implementation. e.g.,

- a mock file system that would report a file as saved but would not actually modify the underlying file system.
- a mock database that would report a record as saved, but just keeps it in a list in memory.

You can now write unit tests against the combination of your class + the mock class. Since you control everything, you can trust that you will get consistent results during testing.

Test Doubles

In practice, creating a mock involves these steps:

1. Specifying a interface for class you wish to mock.
2. Creating a bridge class that communicates with the actual dependency.
3. Creating a mock class that implements that interface in a trivial way.
4. Creating unit tests that use your mock class.

```
interface IFileSystem {
  fun writeFile()
  fun readFile()
  fun fileExists()
}
class FileSystem() : IFileSystem { /* actual implementation */}

class FileRepo(val fileSystem: IFileSystem) {
  fun read(): File { return fileSystem.readFile() }
  fun write(file: File) { fileSystem.writeFile(file) }
  fun exists(file: File): Boolean { return fileSystem.fileExists() }
}
```

We use Dependency Injection to pass the mock dependency into the `FileRepo` instance when we construct it. In our production code, we would construct `FileRepo` with an instance of the `FileSystem` instead.

```
class MockFileSystem : IFileSystem {
  val files = emptyList<File>()
  fun readFile(): File { return files.last() }
  fun writeFile(file: File) { files.add(file) }
  fun fileExists(file: File) { return files.contains(file) }
}

@Test
class FileSystemTest {
  @Test
  fun writeFileTest() {
    val fs = MockFileSystem()
    val repo = FileRepo(fs)
    repo.writeFile(...)
    assert(...)
  }
}
```

TDD

What is it?

Test-Driven Development (TDD) was developed by Kent Beck as an Extreme Programming (XP) practice around 1999. It emphasizes writing tests first, and then writing code to make the tests pass. This results in tests and code being written togther, with the expectation of better test coverage. [3].

TDD development cycle

- Define a specification that you wish your software to meet.
- Define an interface for your class or module.
- Write a test against that interface. It should initially fail.
- Write the implementation code that causes the test to pass.
- Repeat until the specification is met, with complete test coverage.

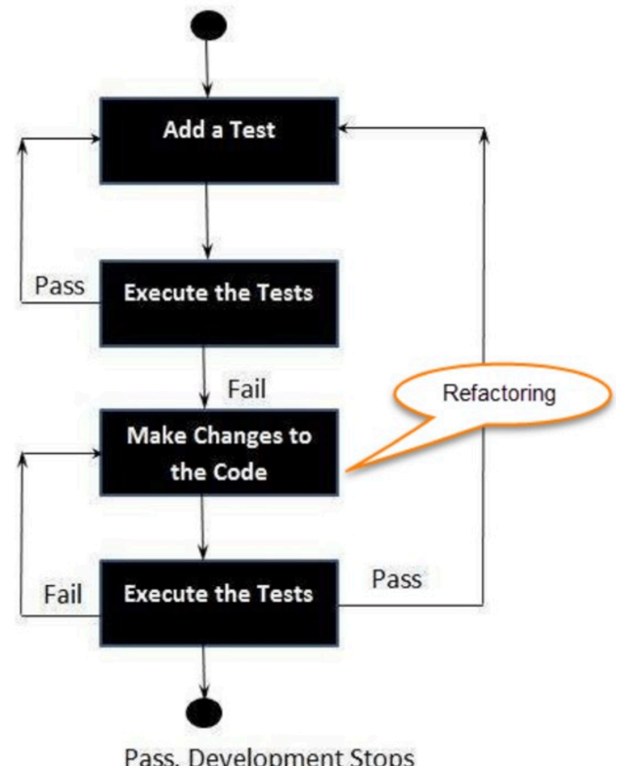


Figure 2: TDD promotes writing tests as a means of aligning tests and implementation code.

What is it?

Advantages of TDD

Early bug detection

- Tests are written alongside code, so you should catch bugs earlier.
- You should complete implementation with a full set of tests.

Better designs

- Writing tests forces you to write clean code. e.g., improved interfaces, clean separation of concerns, cohesive classes.
- Making code testable also makes it cleaner i.e. more flexible, readable.

Confidence to refactor

- Refactoring is improving your code incrementally.
- To refactor, you need to know that you haven’t “broken anything”.
- TDD means that you can trust your tests to catch any mistakes you make.

Bibliography

[1] I. Sommerville, *Engineering Software Products - An Introduction to Modern Software Engineering*. London, UK: Pearson, 2021. [Online]. Available: https://iansommerville.com/engineering-software-products

[2] V. Khorikov, *Unit Testing Principles, Practices, and Patterns*. Manning Press, 2020.

[3] K. Beck and C. Andres, *Extreme Programming Explained*, 2nd ed. Boston, USA: Addison-Wesley Professional, 2004. [Online]. Available: https://www.amazon.ca/Extreme-Programming-Explained-Embrace-Change-ebook/dp/B00N1ZN6C0