WILD CODE SCHOOL

Clever Testing

Testing in Java, Kotlin and Scala with Mockito & AssertJ

OUR GOAL IS TO WRITE BUG-FREE SOFTWARE. I'LL PAY A TEN-DOLLAR BONUS FOR EVERY BUG YOU FIND AND FIX.
YAHOO!
WE'RE RICH
YES!!! YES!!! YES!!!
I HOPE THIS DRIVES THE RIGHT BEHAVIOR.
I'M GONNA WRITE ME A NEW MINIVAN THIS AFTER-NOON!
S. Adams E-mail: SCOTTADAMS@AOL.COM
© 1995 United Feature Syndicate, Inc.(NYC)

WILD
CODE
SCHOOL

- Unit Testing & TDD
- Mocking & Assertions
- Deep Dive: Mocking with Mockito
- Deep Dive: Asserting with AssertJ
- Sample Application with Tests

What is a unit test?

Unit tests are typically automated tests written and run by software developers to ensure that a section of an application (known as the "unit") meets its design and behaves as intended.

In Java, it’s possible to write special classes to test if the code in your program gives the expected results.

(Wikipedia)

What's unit testing?

1. Fixes bugs **early** in the development cycle
2. **Helps** developers **to understand** the code base
3. Enables them to **make changes quickly** (because it’s modular)
4. Good unit tests serve as **project documentation**
5. Unit tests promote **code reuse**

Example scenario

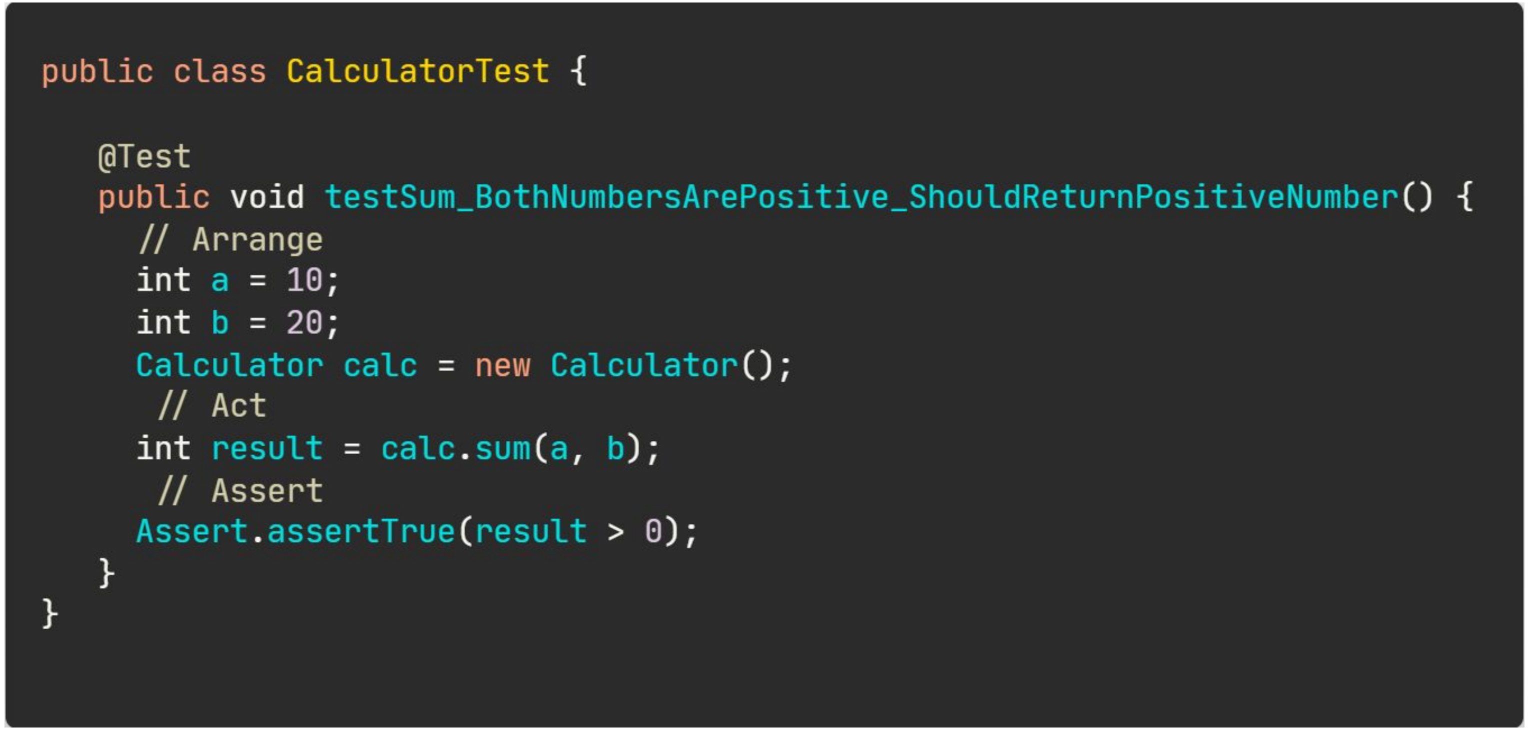
If you were to implement a calculator that supports the sum operation.

Your **acceptance criteria** might be:

- if numbers are positive the result should be positive.
- if numbers are negative the result should be negative.
- if numbers are opposite the result should be zero.

Example of a test written in Java

Acceptance criteria: if numbers are positive the result should be positive.



```java
public class CalculatorTest {

   @Test
   public void testSum_BothNumbersArePositive_ShouldReturnPositiveNumber() {
     // Arrange
     int a = 10;
     int b = 20;
     Calculator calc = new Calculator();
      // Act
     int result = calc.sum(a, b);
      // Assert
     Assert.assertTrue(result > 0);
   }
}
```

Take a look at the full example code

Unit testing frameworks

Unit testing frameworks for various languages:

- ➢ JsUnit for Javascript
- ➢ Mockit for Angular
- ➢ PyUnit for Python
- ➢ CppUnit for C++
- ➢ PhpUnit for Php
- ➢ **JUnit** for Java (Kotlin/Scala)

JUnit

https://junit.org/junit5/

JUnit is a unit testing framework for the Java programming language. It is an open-source framework, and is used to write and run repeatable automated tests.

- ❖ Easy to setup and run.
- ❖ Supports annotations.
- ❖ Allows certain tests to be ignored or grouped and executed together.
- ❖ Supports parameterized testing, i.e. running a unit test by specifying different values at run time.
- ❖ Supports automated test execution by integrating with build tools like Ant, Maven, and Gradle.

Some Vocabulary

Unit: block of code to be tested

Assertion: verification of an expected result. If the check fails, an exception is thrown and the current test stops.

Fixture: initialization / termination common to all unit tests

Suite: a set of executable unit tests

JUnit

Common JUnit 5 Annotations

Annotations are like meta-tags that you can add to your code, and apply them to methods or in a class.

Some common annotations for JUnit 5 are:

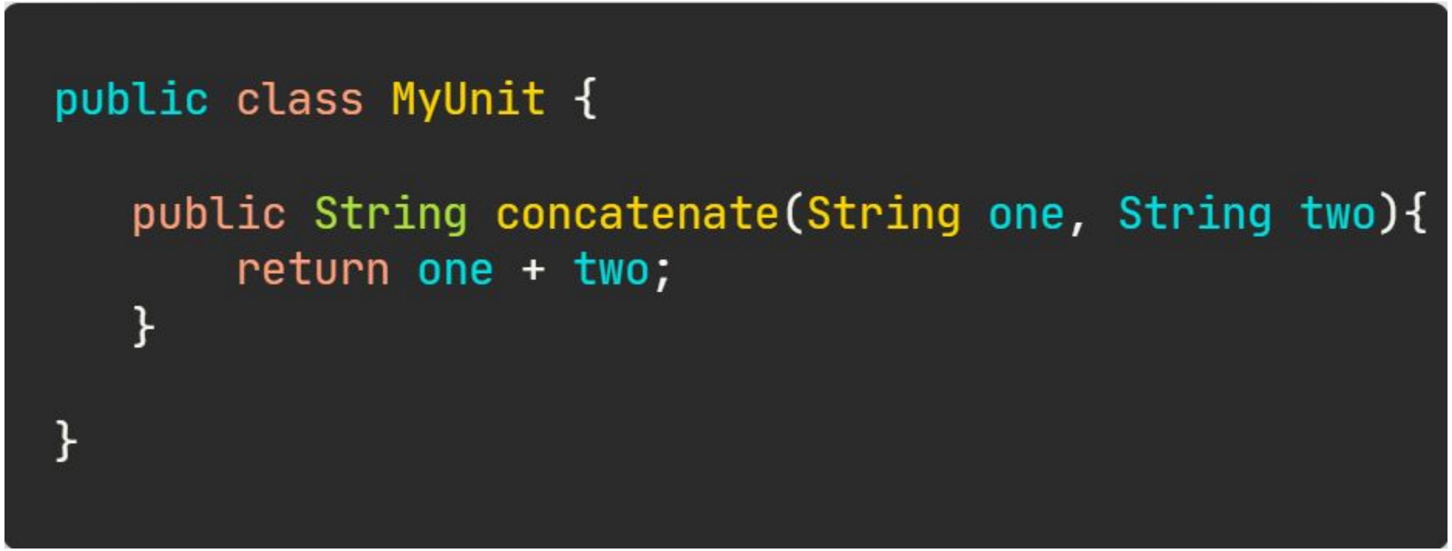
@Test - Marks the method as a test method.

@BeforeEach and @AfterEach sandwiches each test method in the class.

@BeforeAll and @AfterAll sandwiches all of the test methods in a JUnit test class.

Example of a JUnit test written in Java

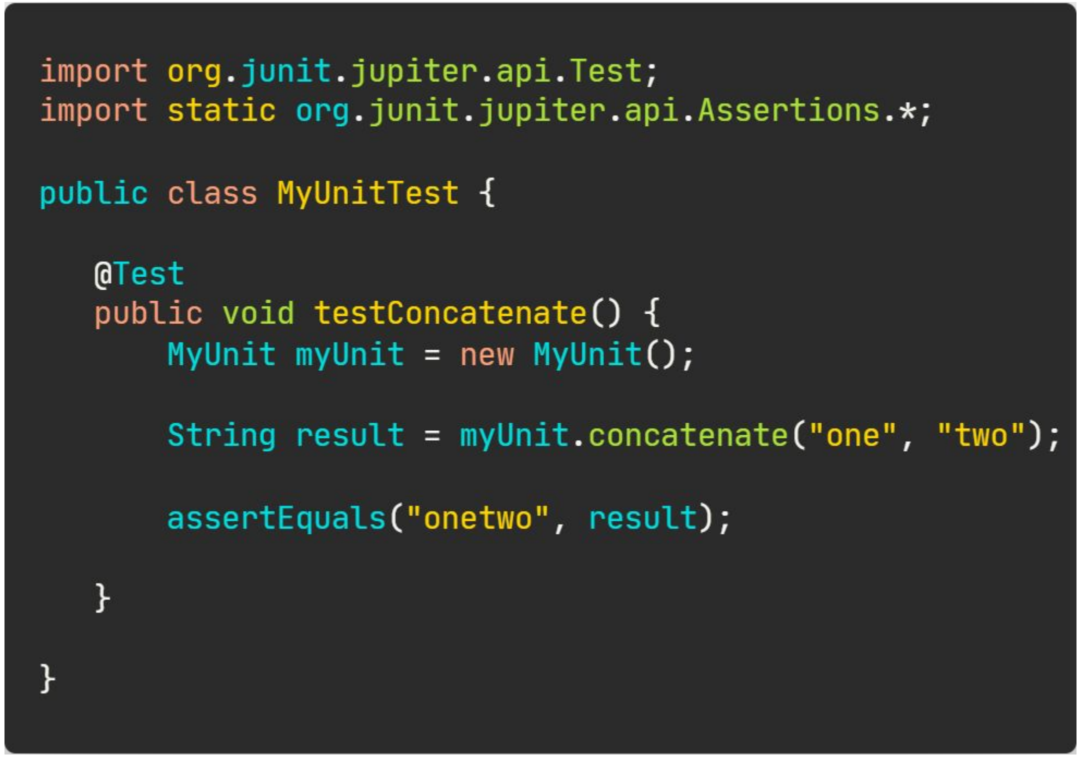
The class we want to test (System Under Test)



```java
public class MyUnit {

    public String concatenate(String one, String two){
        return one + two;
    }

}
```

Example of a JUnit test written in Java

The JUnit unit testing the concatenate() method



```java
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class MyUnitTest {

    @Test
    public void testConcatenate() {
        MyUnit myUnit = new MyUnit();

        String result = myUnit.concatenate("one", "two");

        assertEquals("onetwo", result);

    }

}
```

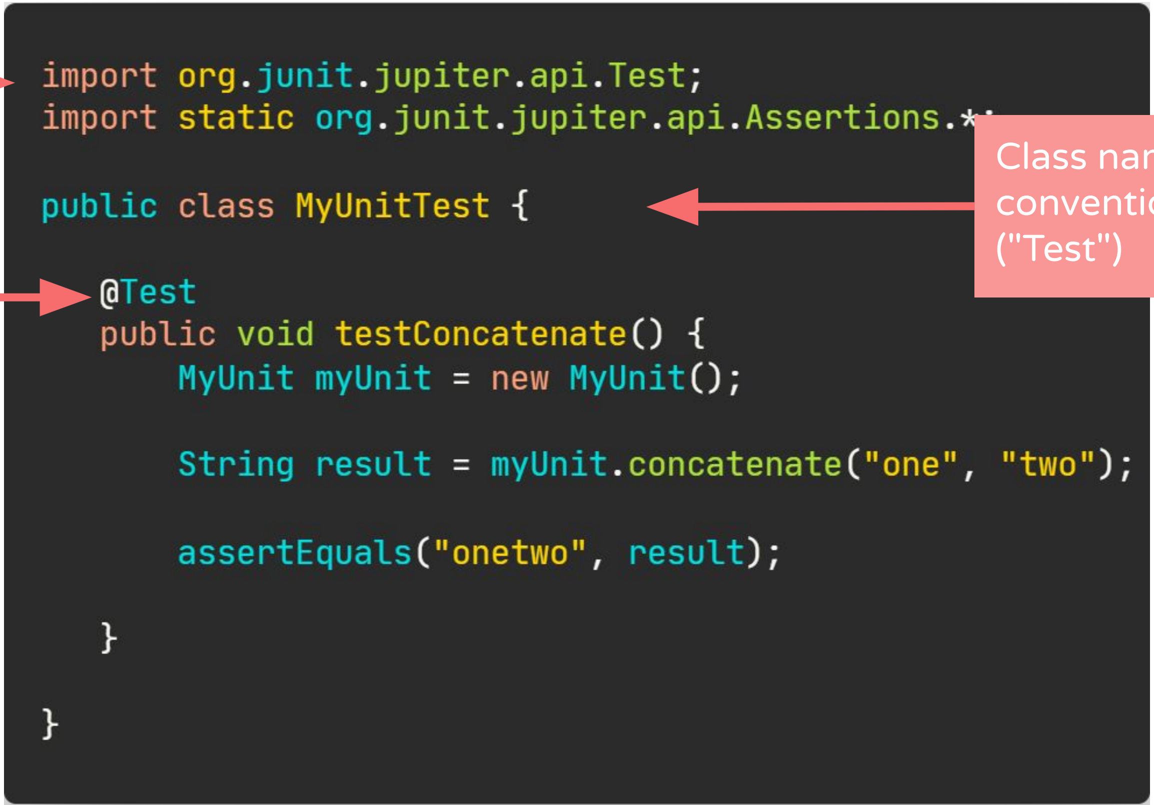
Example of a JUnit test written in Java

The JUnit unit testing the concatenate() method

Import annotation @Test

Annotation "@Test" indicating that the method is a test to be executed

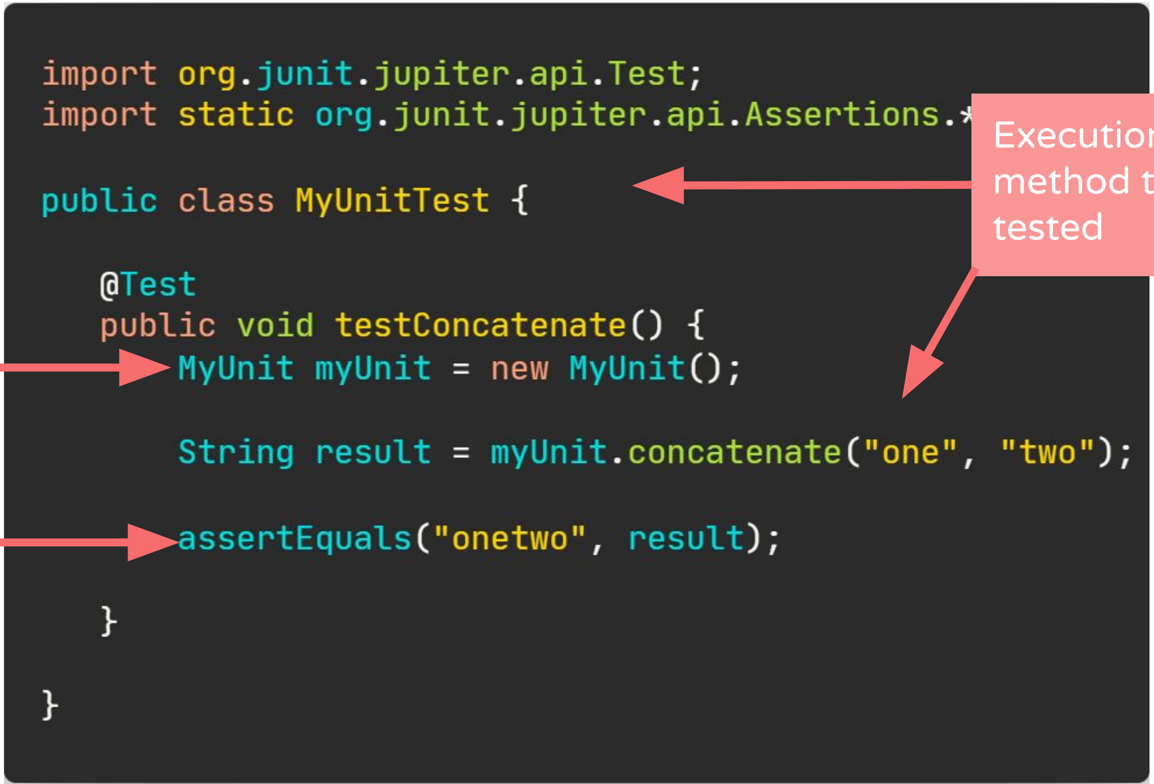
Class naming convention - suffix ("Test")



```java
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class MyUnitTest {

    @Test
    public void testConcatenate() {
         MyUnit myUnit = new MyUnit();

         String result = myUnit.concatenate("one", "two");

         assertEquals("onetwo", result);

    }

}
```

Example of a JUnit test written in Java

The JUnit unit testing the concatenate() method



```java
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*

public class MyUnitTest {

    @Test
    public void testConcatenate() {
        MyUnit myUnit = new MyUnit();

        String result = myUnit.concatenate("one", "two");

        assertEquals("onetwo", result);

    }

}
```

Execution of the method to be tested

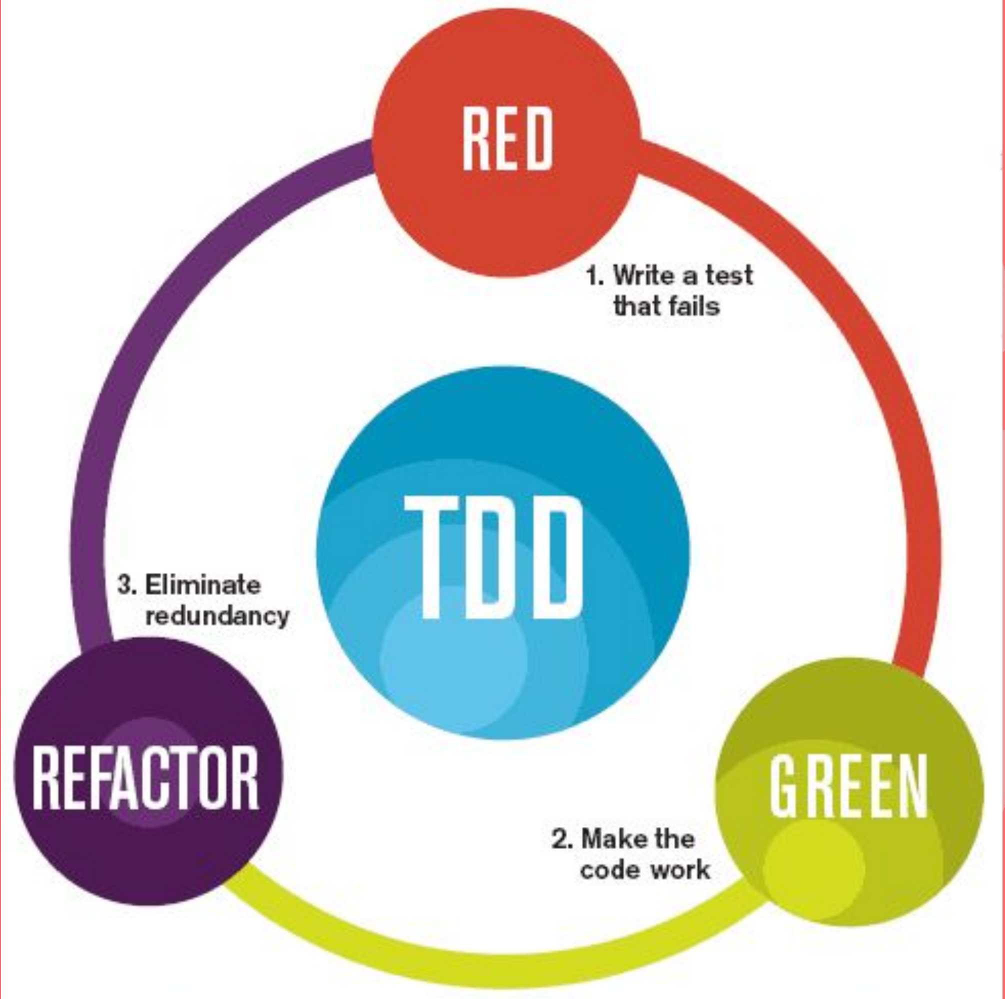
Creating an instance of the class to be tested

Assertion verifies the execution results of the code to be tested

WILD CODE SCHOOL

Test Driven Development (TDD)

1. **Add a** test
   Before writing the application code
   - **Forces you to think about your code before writing it!**
2. **Check** that the test does not pass
   - Normal since the code is not written
3. **Implement** the functionality
   - At the very least, just so it can send back a proper test…
4. **Perform** unit tests
   - Check that the tests pass, otherwise debug
5. **Refactor** the source code and re-run unit tests
6. If still ok, move on to the next objective and start the process again at step 1.



The mantra of Test-Driven Development (TDD) is "red, green, refactor."

Benefits of TDD

- Forces you to write tests in a **systematic** way: more tests = less bugs and regressions
- Forces to **think ahead about the** implementation before coding (class names, methods, parameters, return values, behaviors...)
- Encourages you think of **all the** possible **scenarios** that could lead to malfunctions.
- Allows a better code **breakdown** by limiting the complexity of each step (step by step realisation)

In the end, **it's the code that adapts to the tests,** writing tests beforehand means writing code that is easily testable! Otherwise, there is a risk of testing portions of badly written code which is therefore difficult to write and test.

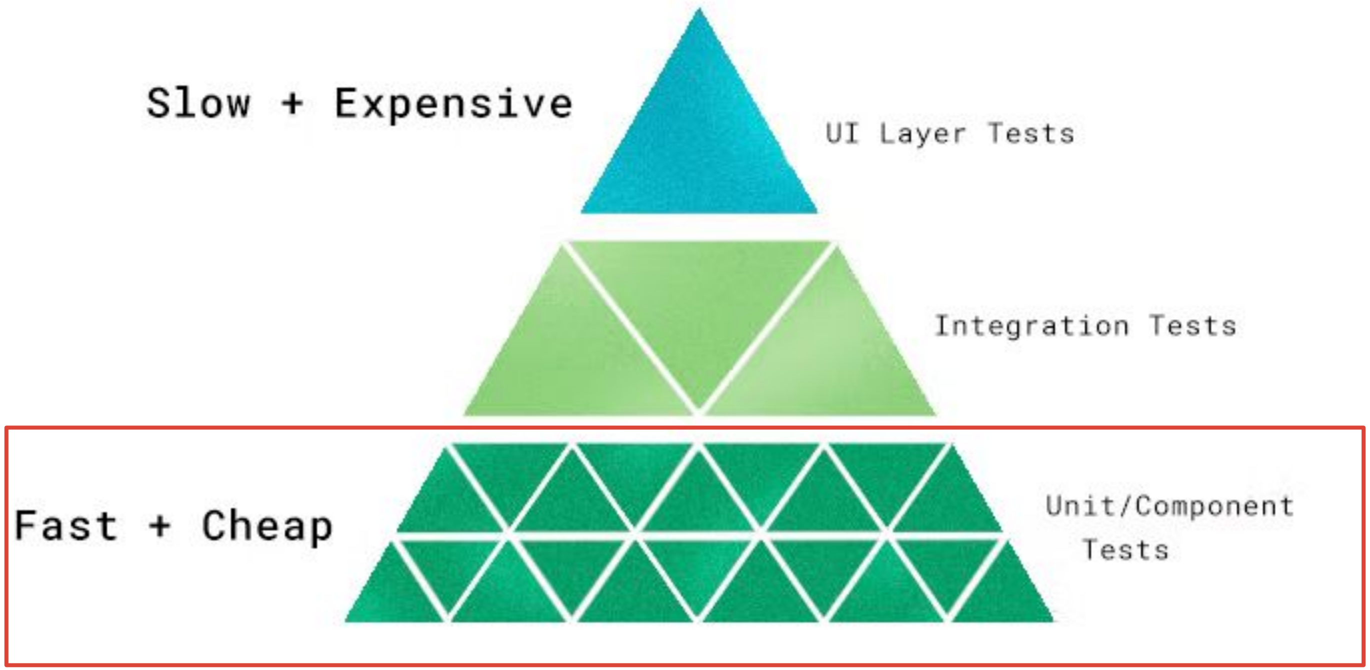
WILD CODE SCHOOL

Mocking with Mockito

Different Types of Tests

- **Unit test** all smaller, pure, core components like entities, domain services, aggregates, value objects, utility classes and so on
- Also **unit test** your use cases with **mocks** to confirm that we perform commands against infrastructure dependencies; use **stubs** to force code to go down different code paths — do this for all of your acceptance tests
- **Integration test** all of your incoming and outgoing adapters (GraphQL API, database, cache, event subscribers, etc)
- **End to End test** — from the front-end to the GraphQL API, through the database and the cache — a select few acceptance tests to get that extra bit of confidence

Optimize Testing: Left Shift



Scope of Unit Tests

What's *not* a unit test?

Michael Feathers says a test is not a unit test if:

1. It talks to the **database**
2. It communicates across the **network**
3. It touches the **file system**
4. It can’t run at the same time as any of your **other unit tests**
5. You have to do special things to your **environment** (such as editing config files) to run it.

Mocks vs Stubs

Use mocks for commands, use stubs for queries.

Do not perform assertions on queries.

Assert commands that were invoked using mocks.

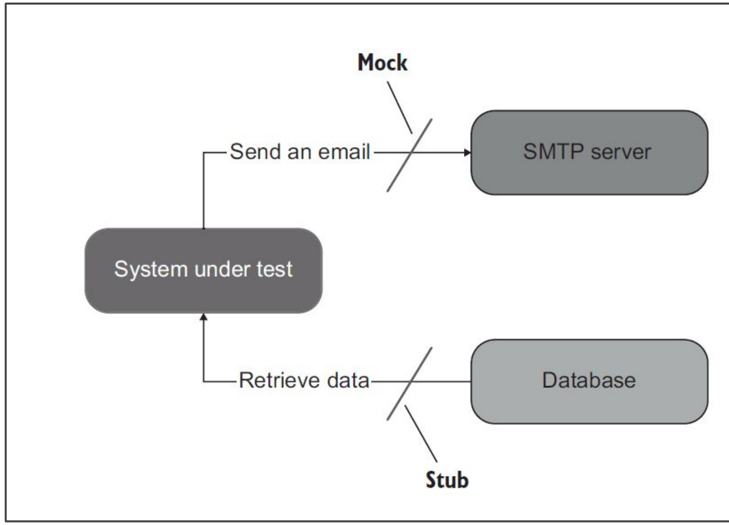


Figure 5.2 Sending an email is an *outcoming* interaction: an interaction that results in a side effect in the SMTP server. A test double emulating such an interaction is a *mock*. Retrieving data from the database is an *incoming* interaction; it doesn't result in a side effect. The corresponding test double is a *stub*.

Unit Testing Principles, Practices, and Patterns - Manning
https://khalilstemmler.com/wiki/test-doubles/

Mocks vs Stubs

Use mocks for commands, use stubs for queries.

Do not perform assertions on queries.

Assert commands that were invoked using mocks.

- A **stub** is a **fake class** that comes with **preprogrammed return values**. It’s injected into the class under test to give you absolute control over what’s being tested as input. A typical stub is a database connection that allows you to mimic any scenario without having a real database.
- A **mock** is a **fake class** that can be examined after the test is finished for its **interactions with the class under test**. For example, you can ask it whether a method was called or how many times it was called. Typical mocks are classes with side effects that need to be examined, e.g. a class that sends emails or sends data to another external service.

https://khalilstemmler.com/wiki/test-doubles/
https://semaphoreci.com/community/tutorials/stubbing-and-mocking-with-mockito-2-and-junit

Different Types of Fake Objects

Mocks

- **Mocks**: We use a mock to stand in and **assert** against command-like operations (outgoing interactions or state changes against dependencies). We pass mocks in to the system under test (SUT) and later check during the *assert* phase of a test that the correct calls to change the dependencies' state were made.
- **Spies**: These are exactly the same thing as traditional mocks except that we *hand-roll* spies manually, whereas with traditional mocks, mocking libraries like ts-auto-mock help you create mocks in a single line or two (I highly recommend this package over basic mocking with Jest). You can see an example of a hand-rolled spy here.

Stubs

- **Dummies**: These don't do anything. Nor are they used during a test. These are objects that are just used to fill up parameter lists so that a constructor, function, or method will execute. They can often be null or empty-string.
- **Stubs**: A stub is a dependency that we can configure to return different values in query-like scenarios. This generally takes some effort to do.
- **Fakes**: A fake is practically the same thing as a stub. They only differ in the sense that we create a fake to sub in for a dependency that doesn't exist yet. This can be done very quickly.

https://khalilstemmler.com/wiki/test-doubles/

WILD CODE SCHOOL

Assertions with JUnit, Hamcrest & AssertJ

Assert methods

JUnit provides a class named **Assert**, which provides a set of assertion methods useful in writing test cases and to detect test failure.

The methods are as follows (click each one to learn more)

- assertArrayEquals()
- assertEquals()
- assertTrue() + assertFalse()
- assertNull() + assertNotNull()
- assertSame() and assertNotSame()
- assertThat()

Hamcrest Assertions

In JUnit, Assertions just check if the condition is false, the test fails.

```
assert(x == y)
```

This syntax fails to produce a sufficiently good error message if 'x' and 'y' are not equal. More recent unit test frameworks provide a family of assertion statements, which produce better error messages.

```
assert_equal(x, y)
assert_not_equal(x, y)
```

But this leads to an explosion in the number of assertion macros, as the above set is expanded to support comparisons different from simple equality. Modern unit test frameworks use a library such as Hamcrest to support an 'assert_that' operator that can be combined with 'matcher' objects, leading to syntax like this:

```
assert_that(x, equal_to(y))
assert_that(x, is_not(equal_to(y)))
```

AssertJ Assertions

What is AssertJ Core?

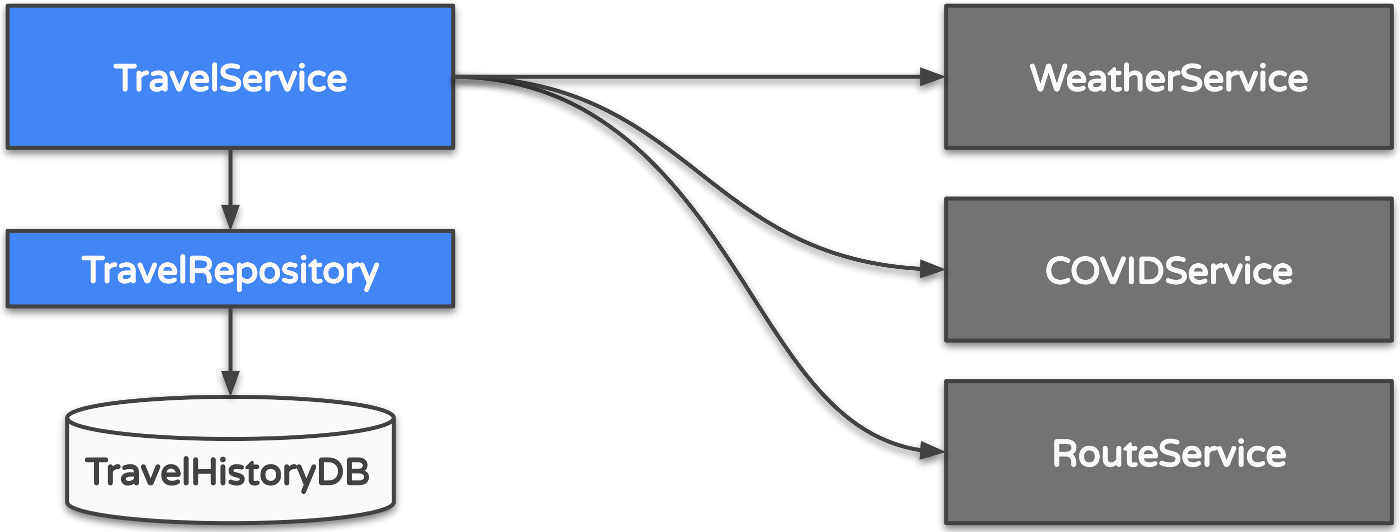
AssertJ is a Java library that provides a rich set of assertions and truly helpful error messages, improves test code readability, and is designed to be super easy to use within your favorite IDE.

WILD
CODE
SCHOOL

Extending the TravelService with ZIP Codes

The Travel Service System Context

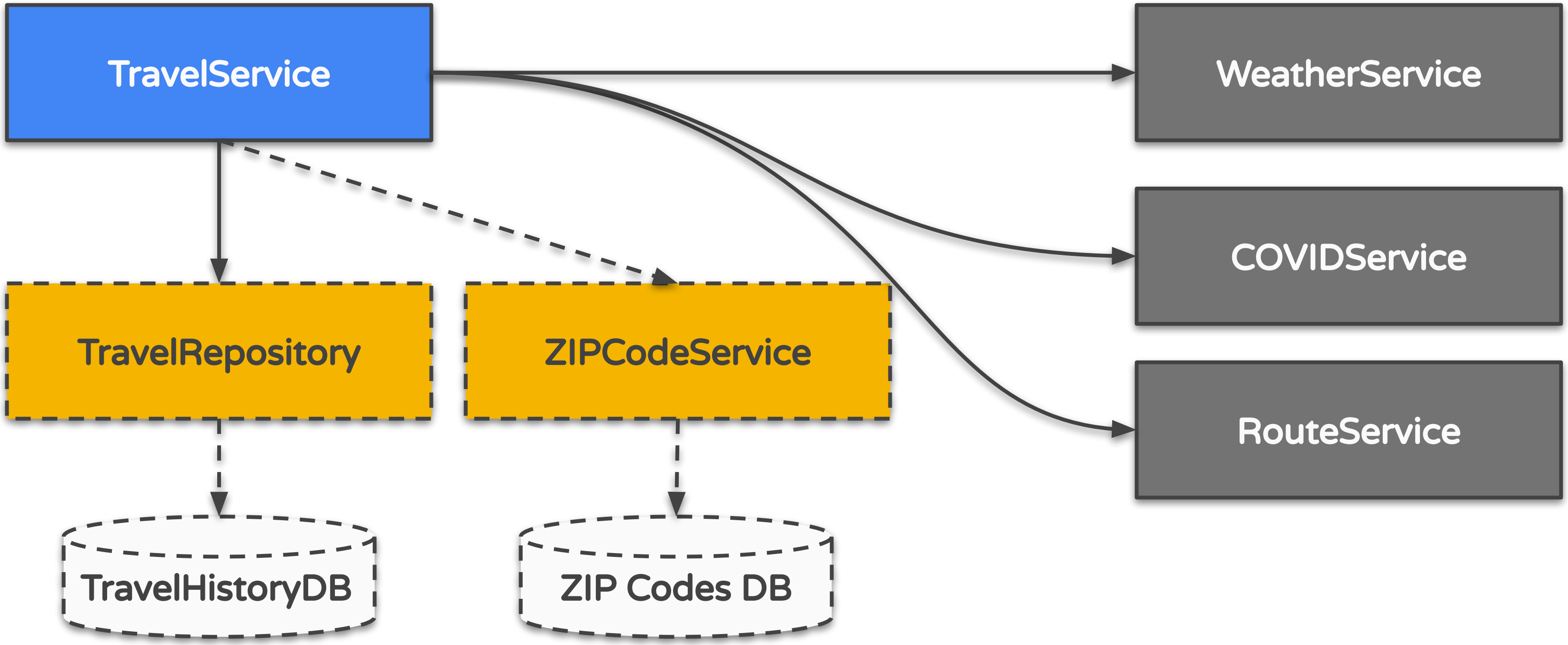
The ***TravelService*** uses **REST APIs**: ***WeatherService***, ***COVIDService*** and ***RouteService*** for Service Integration



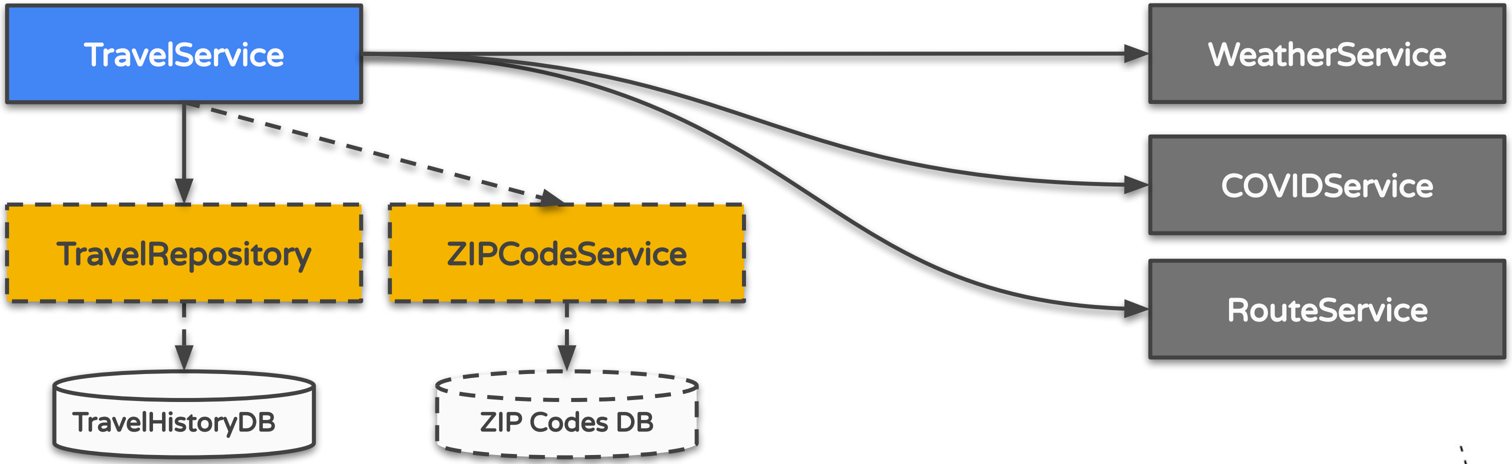
The extended Travel Service System Context

Extend the TravelService for ZIP codes

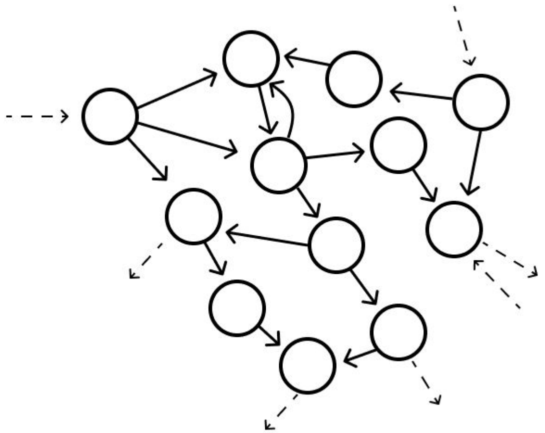
As a user, if I type in a **5 digit number**, the **zip code service is queried** for the city name.
The returned city name is used for further processing.



The extended Travel Service System Context



- How to mock the dependent classes?
- How to stub data classes?
- How to assert results and invocations?



Links and other information

About Mocking

- **About Mocking:** https://khalilstemmler.com/wiki/test-doubles/
- **Mockito at Baeldung:** https://www.baeldung.com/mockito-behavior, https://www.baeldung.com/mockito-verify
- **Stubbing & Mocking with Mockito:** https://semaphoreci.com/community/tutorials/stubbing-and-mocking-with-mockito-2-and-junit

Assertion Frameworks

- **Comparison Hamcrest, AssertJ, Truth:** https://www.sigs-datacom.de/uploads/tx_dmjournals/philipp_JS_06_15_gRfN.pdf
- **AssertJ Tutorial:** https://www.vogella.com/tutorials/AssertJ/article.html

What do you want to learn about next…?

We want to hear from you!

- ❖ Write comments, suggestions, questions to: softwaredeveloper_tutoring@wildcodeschool.com
- ❖ Please vote at: