Unit Testing

For Java, Using JUnit, Mockito, and PIT

André Restivo

Index

Introduction

Software Testing

A **process** to evaluate the **quality** and **functionality** of a software system:

- Does the software meet the specified **requirements**, both **functional** and **non-functional**?
- Are there any **defects** (*aka* bugs)?

Software testing comes in **many forms** and can be done at **different levels** of the software development cycle.

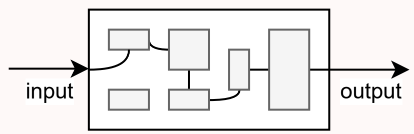
Automated Testing

Traditional software testing was done by **deploying** your application to a **test environment** and **manually** performing **black-box** tests. For example, by **clicking** through the **user interface** to find if something was **broken**.

Automated testing is a **technique** where the **tester/developer** writes **scripts** to test and compare the **actual** outcome with the **expected** outcome.

Black-box vs. White-box

In **black-box** testing, the actual **internal** structure of the item being tested is **unknown** or **not taken into consideration**.



In **white-box** testing, the design of the test cases is **based** on the **internal structure** of the system being tested, so that the **maximum number** of different **code paths** are **covered**.

Testing Levels

- **Unit Testing** – testing **individual units** of a software system in order to validate if they perform as designed.
- **Integration Testing** – **individual units** are **combined** and tested **as a group** in order to expose faults in the **interaction** between them.
- **System Testing** – the **complete software system** is **deployed** and **tested** to evaluate its **compliance** with the specified **requirements**.
- **Acceptance Testing** – the complete system is tested for **acceptability** to evaluate if it is **compliant** with the business **requirements** and acceptable for **delivery**.

Testing Types

- **Smoke** - ensure that the **most important** features work.
- **Functional** - verify if **functional requirements** are met.
- **Usability** - verify if the system is easily **usable** by end-users.
- **Security** - uncover **vulnerabilities** of the system.
- **Performance** - test the **responsiveness** and **stability** of the system under a certain **load**.
- **Regression** - ensure that **previously** developed and tested software still performs after a **change**.
- **Compliance** - determine the **compliance** of a system with any **standards**.

Unit Testing

Unit Testing

Testing **individual units** of a software system in order to **validate** if they perform as designed.

There are several **advantages** to unit tests:

- Increases **confidence** in **changing/maintaining** code.
- In order to make unit testing **possible**, codes need to be **modular**, which makes them more **reusable**. Good unit testing **promotes** good code.
- Development becomes **faster** as the whole system does not need to be run to test newly written code.
- When a test fails we know **which unit** is the **culprit**.

FIRST

The FIRST principles of unit testing:

- **Fast** - Unit tests should be **fast** so we can run them often.
- **Isolated / Independent** - Only test **one unit** at a time. Only test **one thing** at a time. **Order** of tests should **not matter**.
- **Repeatable** - Results should be deterministic and not depend on the environment (time, available data, random values, ...).
- **Self-validating** - No manual checking necessary.
- **Thorough / Timely** - Cover every **use case** scenario (different from 100% code coverage). Test for **corner cases**, **large** data sets, **different** roles, **illegal** arguments and **bad** inputs...

The 3 As

A unit test should be divided into **three** different parts:

- Arrange - Where the test is **setup** and the data is **arranged**.
- Act - Where the the actual method under test is **invoked**.
- Assert - Where a **single logical assert** is used to test the outcome.

Helper classes can be used to **setup** data to be **reused** in **several** tests cases.

Test Doubles

Test doubles are pretend objects that help reduce complexity and verify code independently from the rest of the system. They come in many flavours:

- **Dummy** - never actually used; just to fill parameter lists.
- **Fake** - working implementations, but not suitable for production.
- **Stubs** - provide canned answers to calls made during the test.
- **Spies** - stubs that also record some information based on how they were called.
- **Mocks** - pre-programmed with expectations which form a specification of the calls they are expected to receive.

State vs. Behavior Testing

- **State Testing**: determine whether the exercised method worked correctly by examining **the state** after the method was exercised.

- **Behavior Testing**: specify **which methods** are to be invoked, thus verifying not that the ending state is correct, but that the **sequence of steps** performed was **correct**.

Spies and Mocks are usually needed for **behavior** testing.

JUnit

JUnit

JUnit is a **testing framework** for **Java** specialized in **unit tests**.

A **JUnit** test is a **method**, contained in a **class**, which is **only used** for **testing**.

A **JUnit** test must have the **@Test** annotation.

A simple **test class** looks like this:

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

public class DogTest {
    @Test
    public void testDogName() {
        Dog dog = new Dog("Max", "German Shepherd");
        Assertions.assertEquals("Max", dog.getName());
    }
}
```

Asserts

JUnit provides a series of **assert methods** (as static methods of the *Assertions* class) to help test for certain **conditions**:

- **fail**([message]) - Fails the test.
- **assertTrue**(condition[, message])
- **assertFalse**(condition[, message])
- **assertEquals**(expected, actual[, message])
- **assertEquals**(expected, actual[, tolerance][, message])
- **assertNull**(object[, message])
- **assertNotNull**(object[, message])
- **assertSame**(expected, actual[, message])
- **assertNotSame**(expected, actual[, message])

Message is an **optional message** specifying why the test failed.

Set Up and Tear Down

The **@BeforeEach** and **@AfterEach** annotations allows us to define methods that run **before** or **after** each test method.

These can be used to **setup** and **dispose** of any **data/classes** that are used by all tests, thus simplifying the **Arrange** phase.

There are also **@BeforeClass** and **@AfterClass** annotations that define methods that should be run only **once** for the **entire class**. These might help when test methods share a computationally **expensive** setup.

```
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import java.util.List;

public class DogTest {
    private SQLDogDatabase database;

    @BeforeEach
    public void connectToDatabase() { database = new SQLDogDatabase(); }

    @Test
    public void testFindByBreed() { List<Dog> dogs = database.getAllDogs(); }
}
```

Test Isolation

Test Isolation

One of the key features of **unit testing**, is that of test isolation. The whole point of **unit tests** is to **reduce the scope** of the system under test (SUT) to a **small subset** that can be tested in isolation.

Most of the times this can be difficult without **changing our design**. For example, consider the following **class** and **test**:

```java
public class DogFinder {
  private SQLDogDatabase database = new SQLDogDatabase();

  public List<Dog> findBreed(String breed) {
    List<Dog> allDogs = database.getAllDogs();
    List<Dog> breedDogs = new ArrayList<>();

    for (Dog dog : allDogs)
      if (dog.getBreed().equals(breed))
        breedDogs.add(dog);

    return breedDogs;
  }
}
```

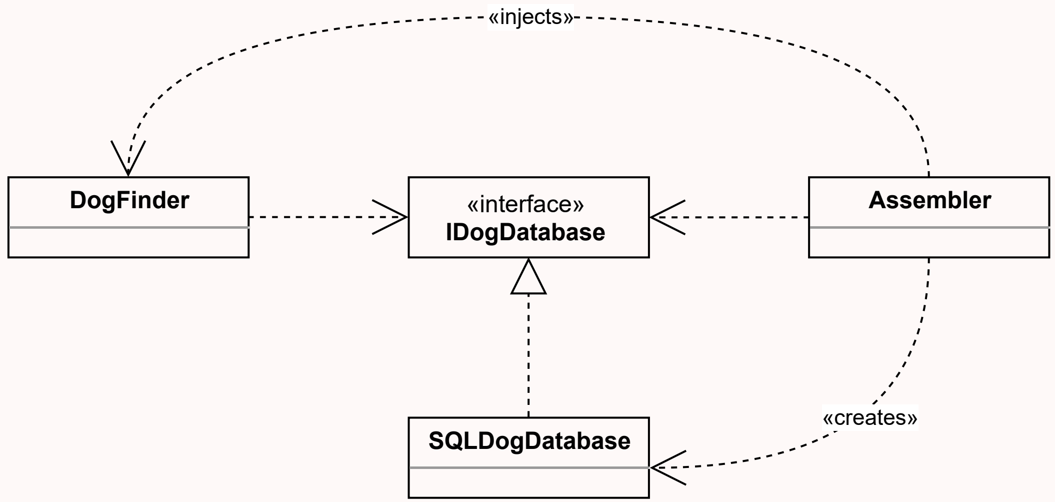
```java
public class DogFinderTest {
    @Test
    public void testFindByBreed() {
        DogFinder finder = new DogFinder();
        List<Dog> dogs = finder.findBreed("Border Collie");
        for (Dog dog : dogs)
            if (!dog.getBreed().equals("Border Collie"))
                Assertions.fail("Got dog from wrong breed!");
    }
}
```

Any test on the **DogFinder** class will depend on the **SQLDogDatabase** class.

Dependency Injection

One way to achieve **test isolation**, is to use **Dependency Injection**.

With this technique, **classes** no longer **depend** on other classes but **on interfaces**. The **concrete instantiation** of each interface is **injected** into the class by a third-party class (the **Assembler**).



Dependency Injection Example

```
public interface DogDatabase {
    public List<Dog> getAllDogs();
}
```

```
public class SQLDogDatabase implements DogDatabase {
    @Override
    public List<Dog> getAllDogs() { /* ... */ }
}
```

```
public class DogFinder {
    private DogDatabase database;

    public DogFinder(DogDatabase database) {
        this.database = database;
    }

    public List<Dog> findBreed(String breed) {
      /* Same code as in previous example */
    }
}
```

```
public class Application {
    public static void main(String[] args) {
        DogFinder finder = new DogFinder(new SQLDogDatabase());
        finder.findBreed("Border Collie");
    }
}
```

Stub Example

Using a **stub** to **isolate** *DogFinderTest* class from *SQLDogDatabase*.

```
public class DogFinderTest {
  class StubDogDatabase implements DogDatabase {
    @Override
    public List<Dog> getAllDogs() {
      List<Dog> dogs = new ArrayList<>();
      dogs.add(new Dog("Border Collie", "Iris"));
      dogs.add(new Dog("Border Collie", "Floyd"));
      dogs.add(new Dog("German Shepherd", "Max"));
      return dogs;
    }
  }


  @Test
  public void testFindByBreed() {
      DogFinder finder = new DogFinder(new StubDogDatabase());

      List<Dog> dogs = finder.findBreed("Border Collie");
      for (Dog dog : dogs)
          Assertions.assertEquals("Border Collie", dog.getBreed());

      Assertions.assertEquals(2, finder.findBreed("Border Collie").size());
  }
}
```

Mockito

Mockito

A simpler way to create **Mocks** and **Stubs** is to use a specialized framework like **Mockito**.

If we are using **Gradle**, the only thing we have to do to be able to use **Mockito** is add the **dependency** in our **"build.gradle"** file:

```
testImplementation 'org.junit.jupiter:junit-jupiter-api:5.6.0'
testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine'

testImplementation('org.mockito:mockito-core:3.7.7')
```

Mockito Stubs

Creating **stubs** with **Mockito** is very simple:

```
import org.mockito.Mockito; //...

public class DogFinderTest {
    private DogDatabase stubDogDatabase;

    @BeforeEach
    public void setUp() throws Exception {
        List<Dog> dogs = new ArrayList<>();
        dogs.add(new Dog("Iris","Border Collie"));
        dogs.add(new Dog("Floyd", "Border Collie"));
        dogs.add(new Dog("Max", "German Shepherd"));

        // A stub with canned answers
        stubDogDatabase = Mockito.mock(DogDatabase.class);
        Mockito.when(stubDogDatabase.getAllDogs()).thenReturn(dogs);
    }

    @Test
    public void findBreed() throws Exception {
        DogFinder finder = new DogFinder(stubDogDatabase);
        List<Dog> dogs = finder.findBreed("Border Collie");
        Assertions.assertEquals(2, dogs.size());
    }
}
```

When and Then

The **when** and **then*** keywords allows to configure **Mockito stubs** to return **canned answers** very easily:

```
stubDogDatabase = Mockito.mock(DogDatabase.class);
Mockito.when(stubDogDatabase.isConnected()).thenReturn(true);
Mockito.when(stubDogDatabase.runSQL(null)).thenThrow(NullPointerException.class);
```

When the method returns *void*, the syntax is slightly different:

```
ArrayList stubList = Mockito.mock(ArrayList.class);
Mockito.doThrow(NullPointerException.class).when(stubList).clear();
```

When/Then Cookbook

Verify

Until now we have been doing **state testing**. If we want to do **behavior testing** we need to use **mocks**, and **Mockito**, as the name implies, can help us with that.

```
@Test
public void findBreedCallsDatabaseOnlyOnce() throws Exception {
    DogFinder finder = new DogFinder(stubDogDatabase);
    List<Dog> dogs = finder.findBreed("Border Collie");
    // Verify if the getAllDogs methods was called only once
    Mockito.verify(stubDogDatabase, Mockito.times(1)).getAllDogs();
}
```

Verify CookBook

Code Coverage

Code Coverage

- Measures the **number** of **code lines covered** by the **test cases**.
- Reports the **total** number of lines in the code and **number** of lines **executed** by tests.
- The **degree** to which the source code of a program is exercised when a **test suite** runs.
- The **higher** the code **coverage**, the **lower** the chance of having undetected software **bugs**.

But, code coverage doesn't tell the **whole story**...

Code Coverage Problems

- **High** coverage numbers are **too easy** to reach (we don't even need **asserts**).
- **Good testing practices** should result in **high coverage**. The inverse is not true.

So **why** do code **coverage** analysis:

- It helps us **find untested** parts of our source code that should be tested but are not.

Code Coverage in IntelliJ

In **IntelliJ** you can **run** your **tests with coverage** to get a **percentage** of code covered per **class** and/or **package**, for **all test suites** or just for **a few**.

```
Right Click on Test Class -> More Run/Debug -> Run ... with Coverage
```

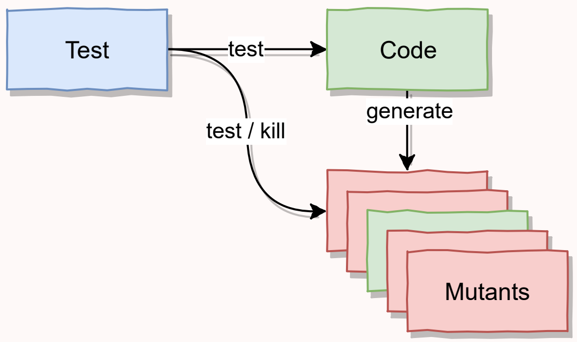
You also get **indicators** throughout your code showing which lines are tested and which are not.

Mutation Testing

Mutation Testing

A type of **software testing** where we **mutate** (change) certain statements in the **source code** and **check** if the test cases are able to **find** the errors.

The **goal** is to assess the **quality** of the **test cases** which should be **robust** enough to **fail mutant code**.



In the mutation testing **lingo**, **tests** are trying to **kill** as many **mutants** as possible (optimally 100% of them).

PIT Mutation Testing

PIT is a mutation testing system, providing gold standard test coverage for **Java**.

With **Gradle**, installing **PIT** for your project in **IntelliJ** is as easy as adding this second line to your **plugins** section in your **"build.gradle"**:

```
plugins {
    id 'java'
    id 'info.solidsoft.pitest' version '1.6.0'
}
```

PIT can be configured directly in your **"build.gradle"** using the same command line parameters as the command line version uses. For example, this enables JUnit 5 support:

```
pitest {
  junit5PluginVersion = '0.12'
}
```

Target Classes

By default, PIT uses the group defined in the **"build.gradle"** file to automatically infer the **targetClasses** parameter. For example, if your **"build.gradle"** file has:

```
group 'com.example'
```

Then it will **automatically infer** the following:

```
pitest {
  targetClasses = ['com.example.*']
}
```

Running Mutation Tests

PIT will **automatically** generate a **Gradle task** called **"pitest"**. So you can **run mutations** tests simply by doing:

```
./gradlew pitest
```

Reports will be created under **"build/reports/pitest/<timestamp>/"** in **HTML** format by default.