SOFTWARE DESIGN AND TESTING LABORATORY

LAB-04/05: UNIT TESTING WITH JUNIT AND SPOCK

1. Project Setup

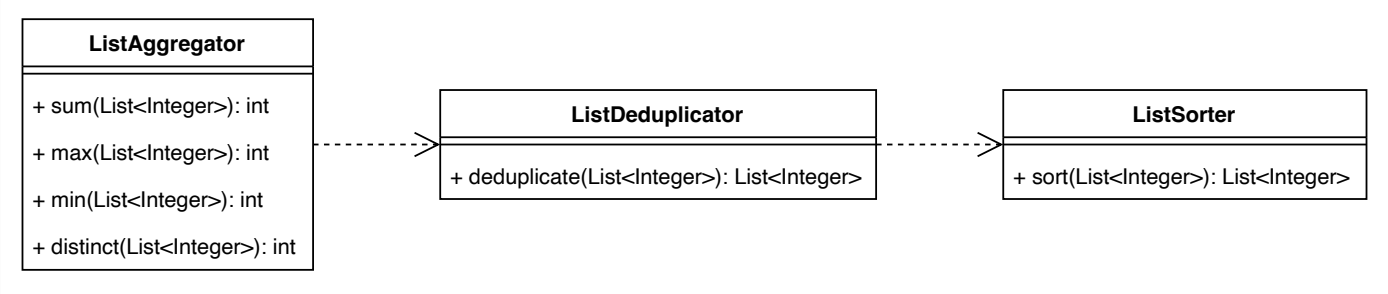
- **Download** and **unzip** the following project into a suitable folder on your computer.
- Start **IntelliJ** and **open** the project.
- Run all tests by selecting `src/test/java`, **right-clicking** and selecting **"Run All Tests"**.
- Verify that all tests **pass** — you might need to turn on showing tests that are passing by selecting the checkmark icon.

2. Inspect and Analyze the Project

The project contains three classes:

- `ListAggregator`: Contains several methods that calculate values from lists of integers (sum, min, max and distinct). The `distinct()` method returns the number of distinct numbers in the list.
- `ListDeduplicator`: Is capable of removing duplicates from a list of integers.

- `ListSorter` : Is capable of sorting a list of integers.



As you can see, the `distinct` method in the `ListAggregator` class, depends on the `ListDeduplicator` class in order to calculate the number of unique elements in a list.

Also, the `ListDeduplicator` class depends on the `ListSorter` class as it is much easier to remove duplicates in an already sorted list.

3. Simplify Test Setup

Take a moment to notice that our test methods are **organized** along **three** different **phases** (the 3 As):

- **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.

Notice that the setup for the `ListAggregator` tests is always the same:

```
List<Integer> list = Arrays.asList(1,2,4,2,5);
```

Do **one of two** things:

- Create a **helper** method, that gets **called** from **each one** of the tests and returns the list to be tested. This solution is better if you are not using the same values in every test.

- Create a **helper** method, having a `@BeforeEach` **annotation**, setting up the list as an **attribute**. Methods with a `@BeforeEach` annotation are called before **each test**.

Do the same for the **other test classes** making sure that all tests **still pass**.

4. Corner Cases

You received a **bug report**:

Bug report #7263

Created a list with values `-1, -4 and -5`.

Tried to calculate the maximum of these values but got `0` instead of `-1`.

- Create a **test** that **confirms** the **bug** (call it *max_bug_7263*).
- Observe that the test **fails**.
- **Fix** the **code** so the test **passes**.

5. Distinct

You received a **bug report**:

Bug report #8726

Created a list with values `1, 2, 4 and 2`.

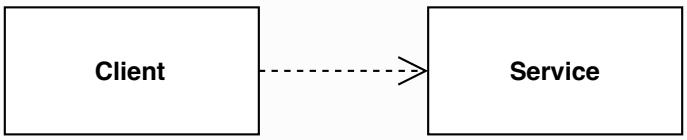
Tried to calculate the number of distinct values in the list but got `4` instead of `3`.

- Start by creating a **test** that **confirms** the **bug**.
- Observe that the test **fails**.
- Then, **look** into the `ListAggregator.distinct()` method code.

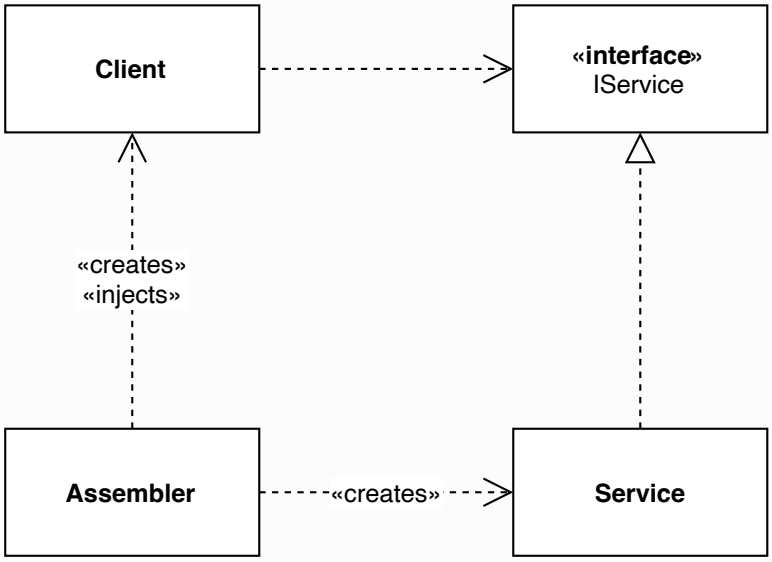
> ! **Spoiler alert**: you won't find anything wrong…

The problem is that when we are testing the `distinct()` method, we are also testing the `ListDeduplicator.deduplicate()` code. **Before** fixing the bug, **lets fix the test**.

To test the `distinct()` and the `deduplicate()` methods independently from each other, we must go from a design that looks like this:



Where our client (the `ListAggregator`) depends directly on its service (the `ListDeduplicator`). To something like this:



Where the **client depends** on an **interface** (lets call it `GenericListDeduplicator`) instead, and some `Assembler` class (the `ListAggregatorTest`) is responsible for creating the **concrete service** (the `ListDeduplicator`) and **injecting** it into the client (the `ListAggregator`).

Like this:

```
ListDeduplicator deduplicator = new ListDeduplicator();
int distinct = aggregator.distinct(Arrays.asList(1, 2, 4, 2),
deduplicator);
```

This is what is called **Dependency Injection** and it allows our test to inject into the `ListAggregator` any list deduplicator **service**. Even one that always responds with the same canned answer (a **Stub**).

To remove the dependency between the `ListAggregatorTest` and the `ListDeduplicator` class using a stub, we first need to:

- Create a `GenericListDeduplicator` interface containing only the definition of the `deduplicate()` method.
- Modify the `ListAggregator.distinct()` method so that it can receive a class that implements this `GenericListDeduplicator` interface.
- Make `ListDeduplicator` implement this interface.
- Change the tests so that a `ListDeduplicator` is injected into the distinct method.

And then create the stub:

- Create a stub that always returns the correct answer for the data we are testing as a **inner-class** inside our `ListAggregatorTest.distinct()` method.
- Modify both `distinct()` tests so that they inject this **stub** class. This should make **both tests pass**.

This did **not fix** any bug, we simply **corrected** the failing **test** as it should not be the one failing. To **fix** our code we still have to:

- Create tests for `sorting` and `deduplicating` using these same values: `1, 2, 4 and 2`.
- Make sure to use **Dependency Injection** in the `deduplicator()` method as it also depends on the `sort()` method.
- **Fix** the code that **needs fixing**. Only **one** of the tests should be failing now and that should point you in the **correct direction**.

6. Mockito

Redo the previous exercise but this time use Mockito to create the stubs.

To use **Mockito**, you must first add this to the dependencies on your **build.gradle** file:

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

Creating a deduplicator using Mockito, should look like this:

```
GenericListDeduplicator deduplicator =
Mockito.mock(GenericListDeduplicator.class);
```

Making the stub return the correct list can then be done like this:

```
 Mockito.when(deduplicator.deduplicate(Mockito.anyList())).then
Return(Arrays.asList(1, 2, 4));
```

7. COVERAGE

- Run all tests again, but this time **right-click** on `src/test/java`, and select **"Run All Tests with Coverage"**.

The report should appear on the right side of the screen.

Enter inside the **com** package, then inside the **aor** and **numbers** packages and verify if all classes, methods and lines are covered by your tests. If not add more tests until they are.

8. FILTERS

Create a new class `ListFilterer` that will be capable of **filtering** a **list of numbers**. This class should have a **constructor** that **receives a filter**:

```
    public ListFilterer(GenericListFilter filter) { ... }
```

And single method called `filter` with the following **signature**:

```
    public List<Integer> filter(List<Integer> list);
```

As you can see, this method **returns** a **list of numbers** that have been **filtered** by a certain **filter** (**Dependency Injection** again).

The `GenericListFilter` interface, should have only one method that returns true if a certain number should be accepted for that filter and false otherwise:

```
    public boolean accept(Integer number);
```

Create **two classes** that follow this interface: `PositiveFilter` (that accepts only **positive** numbers) and `DivisibleByFilter` (that receives an **integer** upon construction and accepts only numbers **divisible** by that number).

- Create tests for all these classes (`ListFilterer`, `PositiveFilter` and `DivisibleByFilter`).
- Use **stubs** when necessary.
- Verify the test **coverage** again.

9. Mutation Testing

Test coverage allows us to **access** the **percentage** of lines **covered** by our **tests** but it does not verify the **quality** of those tests.

Mutation testing tries to **mitigate** this problem by creating **code mutations** (that should not pass the tests) and **verifying** if any of those mutations **survive** our test suite.

To use **PIT** (a **test mutation system** for **Java**) we must first add the following line to the **plugin section** of our **build.gradle** file:

```
    id 'info.solidsoft.pitest' version '1.6.0'
```

Also add the following section anywhere in your **build.gradle** file:

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

By default, **PIT** runs all tests **under** the **package** with the **same name** as the **group** defined in your **build.gradle** file. So if all your classes and tests are under the **com.aor.numbers** package, no other configuration should be necessary.

PIT should have **automatically** created a **gradle task** called **pitest** that you can execute by doing (or using the IntelliJ gradle panel):

```
./gradlew pitest
```

This will run **PIT** and create a **report** under `build/reports/pitest/<date>`. You can open this report using your **browser** and check if any mutations **survived**.

Try **improving** your tests so **all mutations die**.

10. Hero Testing

- With your new found knowledge, create **tests** for the `Hero` code you created last class.
- Try using **Dependency Injection** to **remove** the **dependency** between your `Element` classes and the **lanterna** library.
- Try using **Mocks**, with **Mockito**, to test if the correct **lanterna functions** are being **called** by your code.
- Verify the **coverage** of your tests.
- Try **mutation** testing and **improve** the results by writing **more** and **better** tests.

11. Advanced Material: Spock Testing Framework

Next, we'll take a look at **Spock**, a Groovy testing framework. Mainly, Spock aims to be a more powerful alternative to the traditional JUnit stack, by leveraging Groovy features.

Groovy is a JVM-based language which seamlessly integrates with Java. On top of interoperability, it offers additional language concepts such as being a dynamic, having optional types and meta-programming.

By making use of Groovy, Spock introduces new and expressive ways of testing Java applications, which simply aren't possible in ordinary Java code. We'll explore some of Spock's high-level concepts next, with some practical step by step examples.

11.1 Gradle Dependency

To use spock, one must first add the following dependencies into Gradle. We also need a dependency on Groovy, since Spock tests are written in Groovy. We're only going to use Groovy for testing, so we'll declare this as testImplementation too.

> ! You can delete the other JUnit dependencies since we don't need them.

Your `build.gradle` dependencies should look like:

```
dependencies {
    testImplementation 'org.spockframework:spock-core:2.0-
groovy-3.0'
    testImplementation 'org.codehaus.groovy:groovy-all:3.0.8'
}
```

Since we're using Groovy, we need to add the groovy plugin to Gradle. When we're using the Groovy plugin, we don't need to specifically add the Java plugin as well, since all that functionality is included in the Groovy plugin. We might want to delete java plugin just to keep the noise down in our build file. We can keep the plugin to be explicit, or delete it to be more succinct, it's really down to our own

preferences.

```
plugins {
    id 'groovy'
}
```

11.2 Structure of a Spock Test

Specification and Features

As we are writing our tests in Groovy, we need to add them to the `src/test/groovy` directory, instead of `src/test/java`. Let's create our first test in this directory, naming it `FirstSpecification.groovy`:

```
class FirstSpecification extends Specification {

}
```

Note that we are extending the `Specification` interface. Each Spock class must extend this in order to make the framework recognize it. It's doing so that allows us to implement our first *feature*:

```
def "one plus one should equal two"() {
  expect:
    1 + 1 == 2
}
```

Before explaining the code, it's also worth noting that in Spock, what we refer to as a feature is somewhat synonymous to what we see as a test in JUnit. So whenever we refer to a feature we are actually referring to a test.

Now, let's analyze our feature. We should immediately be able to note some differences between it and Java.

The first difference is that the feature method name is written as an ordinary string. In JUnit, we would have had a method name which uses camelcase or underscores to separate the words, which would not have been as expressive or human readable.

The next is that our test code lives in an expect block. We will cover blocks in more detail shortly, but essentially they are a logical way of splitting up the different steps of our tests.

Finally, we realize that there are no assertions. That's because the assertion is implicit, passing when our statement equals true and failing when it equals false. Again, we'll cover assertions in more details shortly.

Blocks

Sometimes when writing JUnit a test, we might notice there isn't an expressive way of breaking it up into parts. For example, if we were following behavior driven development, we might end up denoting the *given-when* then parts using comments:

```
@Test
public void givenTwoAndTwo_whenAdding_thenResultIsFour() {
    // Given
    int first = 2;
    int second = 4;

    // When
    int result = 2 + 2;

    // Then
    assertTrue(result == 4)
}
```

Spock addresses this problem with blocks. Blocks are a Spock native way of breaking up the phases of our test using labels. They give us labels for given when then and more:

- **Setup** (Aliased by **Given**) – Here we perform any setup needed before a test is run. This is an implicit block, with code not in any block at all becoming part of it
- **When** – This is where we provide a stimulus to what is under test. In other words, where we invoke our method under test

- **Then** – This is where the assertions belong. In Spock, these are evaluated as plain boolean assertions, which will be covered later
- **Expect** – This is a way of performing our stimulus and assertion within the same block. Depending on what we find more expressive, we may or may not choose to use this block
- **Cleanup** – Here we tear down any test dependency resources which would otherwise be left behind. For example, we might want to remove any files from the file system or remove test data written to a database

Let's try implementing our test again, this time making full use of blocks:

```
def "two plus two should equal four"() {
    given:
        int left = 2
        int right = 2

    when:
        int result = left + right

    then:
        result == 4
}
```

As we can see, blocks help our test become more readable.

11.3 Leveraging Groovy Features for Assertions

> ! Within the then and expect blocks, assertions are implicit.

Mostly, every statement is evaluated and then fails if it is not true. When coupling this with various Groovy features, it does a good job of removing the need for an assertion library. Let's try a list assertion to demonstrate this:

```
def "Should be able to remove from list"() {
    given:
        def list = [1, 2, 3, 4]

    when:
        list.remove(0)

    then:
        list == [2, 3, 4]
}
```

While we're only touching briefly on Groovy in this lecture, it's worth explaining what is happening here.

First, Groovy gives us simpler ways of creating lists. We can just able to declare our elements with square brackets, and internally a list will be instantiated.

Secondly, as Groovy is dynamic, we can use def which just means we aren't declaring a type for our variables.

Finally, in the context of simplifying our test, the most useful feature demonstrated is operator overloading. This means that internally, rather than making a reference comparison like in Java, the equals() method will be invoked to compare the two lists.

It's also worth demonstrating what happens when our test fails. Let's make it break and then view what's output to the console:

```
Condition not satisfied:

list == [1, 3, 4]
|    |
|    false
[2, 3, 4]
 <Click to see difference>

at FirstSpecification.Should be able to remove from
list(FirstSpecification.groovy:30)
```

While all that's going on is calling `equals()` on two lists, Spock is intelligent enough to perform a breakdown of the failing assertion, giving us useful information for debugging.

11.4 Asserting Exceptions

Spock also provides us with an expressive way of checking for exceptions. In JUnit, some our options might be using a try-catch block, declare expected at the top of our test, or making use of a third party library. Spock's native assertions come with a way of dealing with exceptions out of the box:

```
def "Should get an index out of bounds when removing a non-
existent item"() {
    given:
        def list = [1, 2, 3, 4]

    when:
        list.remove(20)

    then:
        thrown(IndexOutOfBoundsException)
        list.size() == 4
}
```

Here, we've not had to introduce an additional library. Another advantage is that the thrown() method will assert the type of the exception, but not halt execution of the test.

11.5 Using Datatables in Spock

One easy win for Spock when compared to JUnit is how it cleanly it implements parameterized tests. Again, in Spock, this is known as **Data Driven Testing**. Now, let's implement the same test again, only this time we'll use Spock with **Data Tables**, which provides a far more convenient way of performing a *parameterized test*:

> ! Essentially, **data driven testing** is when we test the same behavior multiple times with different parameters and assertions. A classic example of this would be testing a mathematical operation such as squaring a number. Depending on the various permutations of operands, the result will be different. In Java, the term we may be more familiar with is parameterized testing.

```
def "numbers to the power of two"(int a, int b, int c) {
  expect:
      Math.pow(a, b) == c

  where:
      a | b | c
      1 | 2 | 1
      2 | 2 | 4
      3 | 2 | 9
  }
```

As we can see, we just have a straightforward and expressive Data table containing all our parameters.

Also, it belongs where it should do, alongside the test, and there is no boilerplate. The test is expressive, with a human-readable name, and pure expect and where block to break up the logical sections.

11.6 Mocking

Mocking Using Spock

Spock has it's own mocking framework, making use of interesting concepts brought to the JVM by Groovy. To guide you through the process of creating a mock, lets use the test we created before to test bug #8726.

```
@Test
public void distinct_bug_8726() {
  ListAggregator aggregator = new ListAggregator();

  GenericListDeduplicator deduplicator =
Mockito.mock(GenericListDeduplicator.class);

 Mockito.when(deduplicator.deduplicate(Mockito.anyList())).then
Return(Arrays.asList(1, 2, 4));

  int distinct = aggregator.distinct(Arrays.asList(1, 2, 4, 2),
deduplicator);

  Assertions.assertEquals(3, distinct);
}
```

First, let's instantiate a `Mock`:

```
GenericListDeduplicator deduplicator =
Mock(GenericListDeduplicator)
```

In this case, the type of our mock is inferred by the variable type. As Groovy is a dynamic language, we can also provide a type argument, allow us to not have to assign our mock to any particular type:

```
def deduplicator = Mock(GenericListDeduplicator)
```

Now, whenever we call a method on our PaymentGateway mock, a default response will be given, without a real instance being invoked:

```
when:
    def result = deduplicator.deduplicate(Arrays.asList(1, 2,
4, 2))

then:
    result == null
```

The term for this is *lenient mocking*. This means that mock methods which have not been defined will return sensible defaults, as opposed to throwing an exception. This is by design in Spock, in order to make mocks and thus tests less brittle.

Stubbing Method Calls on Mocks

We can also configure methods called on our mock to respond in a certain way to different arguments. Let's try getting our `GenericListDeduplicator` mock to return the list `[1,2,4]` when we invoked with `[1,2,4,2]`:

```
given:
    def deduplicator = Mock(GenericListDeduplicator)
    deduplicator.deduplicate(Arrays.asList(1, 2, 4, 2)) >>
Arrays.asList(1, 2, 4)

when:
    def result = deduplicator.deduplicate(Arrays.asList(1, 2,
4, 2))

then:
    result == Arrays.asList(1,2,4)
```

What's interesting here, is how Spock makes use of Groovy's operator overloading in order to stub method calls. With Java, we have to call real methods, which arguably means that the resulting code is more verbose and potentially less expressive.

Now, let's try a few more types of stubbing.

If we stopped caring about our method argument and always wanted to return true, we could just use an underscore:

```
deduplicator.deduplicate(_) >> Arrays.asList(1, 2, 4)
```

If we wanted to alternate between different responses, we could provide a list, for which each element will be returned in sequence:

```
deduplicator.deduplicate(_) >>> [Arrays.asList(1, 2, 4),
Arrays.asList(6, 7)]
```

There are more possibilities, and these are left for your own investigations!

Verification

Another thing we might want to do with mocks is assert that various methods were called on them with expected parameters. In other words, we ought to verify interactions with our mocks.

A typical use case for verification would be if a method on our mock had a `void` return type. In this case, by there being no result for us to operate on, there's no inferred behavior for us to test via the method under test. Generally, if something was returned, then the method under test could operate on it, and it's the result of that operation would be what we assert.

Let's try verifying that a method with a void return type is called:

```
def "Should verify notify was called"() {
    given:
        def notifier = Mock(Notifier)

    when:
        notifier.notify('foo')

    then:
        1 * notifier.notify('foo')
}
```

Spock is leveraging Groovy operator overloading again. By multiplying our mocks method call by one, we are saying how many times we expect it to have been called.

If our method had not been called at all or alternatively had not been called as many times as we specified, then our test would have failed to give us an informative Spock error message. Let's prove this by expecting it to have been called twice:

```
2 * notifier.notify('foo')
```

Following this, let's see what the error message looks like. We'll that as usual; it's quite informative:

```
Too few invocations for:

2 * notifier.notify('foo')   (1 invocation)
```

Just like stubbing, we can also perform looser verification matching. If we didn't care what our method parameter was, we could use an underscore:

```
2 * notifier.notify(_)
```

Or if we wanted to make sure it wasn't called with a particular argument, we could use the not operator:

```
2 * notifier.notify(!'foo')
```

Again, there are more possibilities, and these are left for your own investigations!

12. Hero Testing with Spock

Re-implement your tests, now using Spock and Spock Mocking.

13. Other resources:

- https://blog.jetbrains.com/idea/2021/01/tutorial-spock-part-1-getting-started/