Workshop - TDD

Part 1 - Unit testing

In this part, you're going to implement a calculating system. It is a very simple and unfortunately useless system, but does have some nice pitfalls when it comes to testing.

These exercises are meant to be done in the "test first" style. This means that when you're asked to develop a class or implementation, you should first develop the test for it.

Materials

The teacher will provides maven base project already set up for

- JUnit
- Mockito

Exercise 1

Create two classes, `Divider` and `DivisionResult`. The `Divider` will have one method `divides(Integer dividend, Integer divisor)` that takes two integers as parameter. The return value should be a `DivisionResult` object with three properties: both parameters and the result value, which is an integer too.

Note: The remainder of the division may be discarded.

Hint: Test first, remember? Use CTRL-1 in Eclipse or ALT-ENTER in IntelliJ to enter the quick-fix mode and create a class on-the-fly from within your test.

Exercise 2

The functional guys decided that the `ArithmeticException`, which is thrown when you divide by 0 is not very sexy. The exception should be wrapped in an "`ImpossibleCalculationException`". The `ImpossibleCalculationException` should be a checked exception (so don't extend `RuntimeException`).

Note: You should catch the `ArithmeticException` and throw an `ImpossibleCalculationException`

Exercise 3

Someone discovered a "bug" in your system. When you divide 3 by 2, the result is 1, but should be 2. You must have misread the spec that said "The result of the division must be round up".

Hint: Build a test first that isolates the bug. Then fix the bug and check whether the test passes.

Hint: You'll have to use `Math.ceil()` and cast all parameters to (double). Notice that dividing a double by zero doesn't throw an `ArithmeticException` anymore. One of your tests should point this out.

Exercise 4

Your customer has read about a system that keeps track of the calculations it has done in the past. They decide they want that too. Change the implementation of your Divider to maintain a *static* List of "`DivisionResult`" objects (thus *private static* `List<DivisionResult>`). Failed operations should also *be stored, with null as the result property.*

Note: Remember to change your test(s) first before making this change! Make sure you clean up any leftovers of your test after your test has been executed.

Exercise 5

Your client forgot to mention that they have bought an external system to keep track of all the performed calculations. For the sake of uselessness, the system is implemented by a singleton that keeps track of a list in the same way your `Divider` does.
Refactor your `Divider` implementation to make use of a singleton `DividerHistory` object. Since we don't want our code to be tied to this implementation, we use a setter to set the `DividerHistory` instance in the `Divider`. The `Divider` may assume that the DivisionHistory system is correctly started.
Make sure this assumption is documented in your test somehow.
You can find the implementation of the `DivisionHistory` class in the supplied training material.
As you can see, starting and stopping the `DividerHistory` system is a time consuming operation. It is best to do that only once for each test class.

Note: since `DividerHistory` is a singleton, its state is maintained throughout all test methods. Since you cannot depend on the order of your tests, it is impossible to predict the exact state of the object. Use the set up or tear down method to clear your added calculations.

Bonus Exercise

If you're done, and are still eager to learn some more, try out the following things:

- Ignore one or more of your test methods using `@Ignore` and see how your IDE reports ignored tests
- See if you can run your tests with maven (mvn clean test). See what happens when a test fails or when one is ignored. Check out the /target/surefire-reports folder and see how maven generates the test reports
- Install jacoco eclipse plugin and check the results for your code

Part 2 – Mockito

Until now, our application doesn't really test a unit in isolation. In this part, we'll be using **Mockito** to create a real unit test for our `Divider`.

Exercise 1:

We don't really want to test our `Divider` in combination with the `DivisionHistory` "system" each time.
As you've noticed, it is quite slow. And imagine what happens if the `DivisionHistory` system is offline when you're trying to run your test. Or worse: they're charging you for every time you access the system.
Create a Mock for the `DividerHistory` and use the mock to stub responses and verify calls.

Hint: you don't have to start and stop the `DividerHistory` from within your `Divider`, so you don't have to program that behavior into your mock.

Hint: Did you verify that a call was actually made?

Exercise 2:

As a new requirement, whenever a division by zero occurs, we want a message to be printed to `System.out` like "Division of 12345 by zero". Implement this.
To do so, create a new method `log(String message)` on `Divider` that prints a message to System.out. Do test that this method is called as expected when a division by zero occurs.

Hint: replace the divider instance for this test by a partial mock… spy()

Hint: Did you remember to "verify" the expected calls on your mocks?

Exercise 3:

Often, the need for partial mocking is an indication that the class under test has too many responsibilities. In this case, the logging to `System.out` can be considered such a responsibility, and it is better moved to a separate class.
Move the `log()` method to a separate class Logger, and refactor the `Divider` to use a `Logger` instance to do its logger. Make sure this eliminates the use of partial mocking from the previous exercise.