"Software Engineering 2"
– *Advanced Testing [02]*

Introduction

In this exercise you will create a test environment which utilizes Mockito, AspectJ, and Selenium. The first three tasks are for becoming acquainted with these tools and libraries. If you feel confident enough you may skip these and go directly to the actual task 4: "Testing an Application". For the exercises you find the file `SE2 - Exercise. Test 2 - Task Code.zip` containing all applications to be tested in the following tasks.

Technical Hints

- The exercise is meant to run on a recent Eclipse-version mainly with maven. As of May 2021, you will find the current maven-dependencies, the current AspectJ and Selenium (for Chrome) in the tasks.
- If you are working on the university lab computers, keep your workspace on the C-drive (don't forget to save your results before you log out) otherwise the Selenium-Driver will not be found.
- Maven doesn't seem to interact well all the time with the use libraries and lambdas; if you experience any issues, these (individual) actions might help:
  - Update you maven-project with Maven → Maven Update …
  - *No Main Class Found*: Run As → Maven clean + Project → Clean …
  - *Imports for test-classes not resolved*: Move package to src/main/java, check if it compiles there and move it back to src/test/main
  - *Message-box with cryptic closure-error → Lambda is not updated after recompile*: Change the Lambda somehow and back again to force a recompile; alternatively use (an ugly) anonymous interface implementation instead of an lambda.

0 "Hello World" for maven/JUnit

Implement and run a simple JUnit-example, to see that your installation works.

0.1 Setting up the Maven-Project

Create a new maven-project with the following setting:

- Create a simple project (Skip Archetype Selection)
- Use default workspace location
- Group-Id: de.stuttgart.hft.se2
- Artifact-Id: 00.junit5.samples

0.2 Configure pom.xml

Then open the `pom.xml` and go to the tab pom.xml and add at the end (before `</project>`)

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.8.1</version>
            <configuration>
                <release>11</release>
            </configuration>
        </plugin>

        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>3.0.0-M5</version>
            <dependencies>
                <dependency>
                    <groupId>org.junit.jupiter</groupId>
                    <artifactId>junit-jupiter-engine</artifactId>
                    <version> 5.8.0-M1</version>
                </dependency>
            </dependencies>
        </plugin>

    </plugins>
</build>

<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>

<dependencies>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-engine</artifactId>
        <version> 5.8.0-M1</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Execute Maven → Maven Update…

Remark: Maven, by default, works with Java 1.5, therefore you need the compiler-plugin. The surefire-plugin is not really needed for this exercise, but you would need it later, if you would want to execute maven-test. We will only use the Eclipse-JUnit-Launcher.

0.3 Implement an Application and Test

Implement the following class in `src/main/java`

```java
package de.stuttgart.hft.se2.app;

public class App {

    public static int sum(int n, int m) {
        return n + m;
    }

    public static void main(String[] args) {
        System.out.println("Hello World");
        System.out.println("1 + 2 = " + sum(1, 2));
    }
}
```

Implement the following test case in `src/test/java`

```java
package de.stuttgart.hft.se2.app;


import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;

import de.stuttgart.hft.se2.app.App;

class AppTest {

    @Test
    void testSum() {
        assertEquals(3, App.sum(1, 2));
    }
}
```

0.4 Running the Test

Select the test-class and RunAs → JUnit Test; everything should be green.

1 Using Mockito

Implement the example of the lecture and test a small application, where an interface needs to be mocked.

1.1 Setting up a Maven-Project

Create a new maven-project as before with artifact-id: `01.mockito.samples` and add (to the previous dependencies the following dependency:

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version> 3.9.0</version>
</dependency>
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-junit-jupiter</artifactId>
    <version>3.9.0</version>
    <scope>test</scope>
</dependency>
```

1.2 Implementing a Test Case

Now implement the example of the lecture (see Slides 14-15):

- The folder `01.mockito.samples` contains the application to be tested. It consists of the class under test (`Client`) which uses a class and an interface (`Data` and `Service`). Copy this application to your java-folder
- You should verify the behavior of the method `action()` of the class under test in a Junit-Test (in the test-folder).
- To do so create a JUnit-test class, which creates calls the method `action()` and verifies, that two calls of the method `doSomething(…)` have happened.
- As there is no existing implementation of `Service` you have to mock and inject it.

2 Using AspectJ

2.1 Setting up an AspectJ-Project

AspectJ has to be installed for Eclipse through update: http://download.eclipse.org/tools/ajdt/410/dev/update. Attention this is the Update for the current Eclipse-Version (4.18): If you are using a different Eclipse-version look for the corresponding AspectJ-version!

For utilizing aspects you have to make a new AspectJ-project in Eclipse (instead of a Java-Project). This can be achieved by New → New Project → AspectJ > AspectJ-Project. New aspect-classes can be created by New → Other… → AspectJ → Aspect.

2.2 Implementing a Test Case

Now implement an injecting aspect for testing (see Slides 32-33):

- Create an AspectJ-project `02.aspectj.samples`, add JUnit5 as Library and create a new source-folder “test”.
- The folder `02.aspectj.samples` contains the class and method to be tested. The method `StringReader.main(…)` reads a line from standard input and returns the string. Add this class to your project
- Implement a test case which reads a line from the console and checks, if the expected line has been returned. To automate this, use the class `Redirect` in the samples-folder. This class offers the method `Redirect.input(String … inputs)` which allows to provide some inputs, which are fetched from `System.in` (instead of typing it on your own). Execute `Redirect.output()` before calling `main(…)`. This forces the output to be redirected to an Sting-Array. You can access the outputs by calling `Redirect.getOut()`. Your test can check if you got the to be shown output line.
- Now add a second test method where you test the behavior when an `IOException` is thrown while reading.
- For testing create an aspect, which injects an exception, when `readLine()` tries to read. To make sure the test runs in any case again Redirect an arbitrary input (otherwise, of no exception would be thrown, the test would wait for an input – Timout does not work in this case either).
- *Again: If you receive a message-box with cryptic closure-error* means your *lambda is not updated after recompile*: Change the Lambda somehow and back again to force a recompile; alternatively use (an ugly) anonymous interface implementation instead of an lambda.

3 Selenium

Implement the example of the lecture and test if you can access an internet page.

3.1 Setting up a Maven-Project

Create a new maven-project as before with artifact-id: `03.selenium.samples` and add (to the previous dependencies the following dependency:

```
<dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-java</artifactId>
    <version>3.141.59</version>
</dependency>
```

3.2 Implementing a Test Case

Now implement a test case which verifies the behavior of some web pages (see Slides 37-38):

- The folder `03.selenium.samples` contains three html-pages which should be tested. You should verify that when you open the page `buttons.html` and press “Yes” `yes.html` pops up.
- To do so create a set-up method (`@BeforeAll`), which establishes the connection
- Then create the actual test-method which opens buttons.html, clicks the Yes-button and verifies, that yes.html is now shown (you may check the title of the page for this). For opening a local web-Page the prefix “`file://`” has to be added to the absolute file path.
- Finally, tear everything down (`@AfterAll`) by quitting the driver.

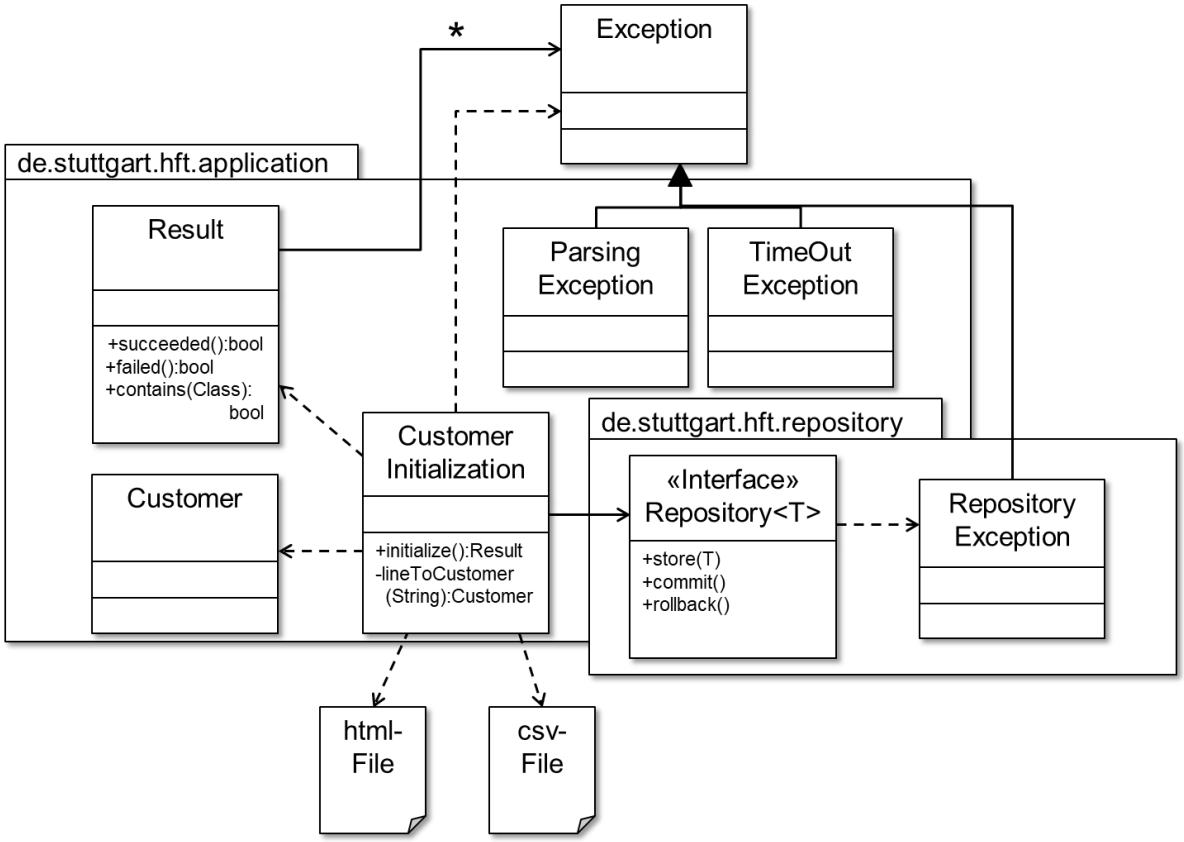
Be aware that you need to download the fitting chromedirver.exe for your Chrome-Browser. For setting up the connection, use this code:

```
System.setProperty("webdriver.chrome.driver","PATH/TO/chromedriver.exe");
ChromeOptions options = new ChromeOptions();
options.addArguments("--no-sandbox");
options.setPageLoadStrategy(PageLoadStrategy.EAGER);
WebDriver driver = new ChromeDriver(options);
```

4 Testing an Application with Advanced Techniques

Now apply these techniques for testing an application. During all the tests it is not allowed to change any code or file of the application under test.

The folder `04.sample.application` contains a simple application which reads data from a file, creates records and hands over the records for storing in a database (the “Repository”). The following UML shows the basic structure:



The application works in the following way:

- For an instance of `CustomerInitalization` the method `initialize()` is called.
- The method opens the csv-file for reading and reads line by line.
- If a line could be parsed a `Customer`-object is created and stored in the repository with the `store(...)`-method.
  - If a line failed to parse a `ParsingException` is thrown, however, the reading continues.
  - If the reading / parsing / storing takes longer than 2 seconds a `TimeOutException` is thrown, again the reading continues.
  - If the storing fails a Repository Exception is thrown.
- Each exception is caught by the `initialize()`-method and recorded.
- At the end, when no exception happened, the repository is committed, if there were exceptions, a rollback is done.
- A `Result`-Object is returned, which can be asked for success / failure and if a certain exception has happened.

- Also, an html-file is written, with the current date and the overall result (“successfully initialized” or “initialization failed”).

The component under test is the class `CustomerInitialization`. Your task is to test the method `initialize()` within this class. For performing the tests it is not allowed to change any code within this or the other classes, or to change the enclosed csv-file at this location.

The main problems for this test are:

- There is no implementation for the Repository-interface available yet; furthermore any instance for the `CustomerInitialization`-class has to be injected, as there is no constructor or setter available.

```
public class CustomerInitialization {
 …
 private Repository<Customer> repository;
 …
}
```

- The path to the csv-and html-file is hard-coded in the `CustomerInitialization`-class and the supplied data is all correct and does not trigger any exceptions and the data should not be altered (at the current position).

```
public class CustomerInitialization {
…
 private static final String CUSTOMER_DATA = "data/CustomerData.csv";
 private static final String RESULT_PAGE = "Result.html";
 …
 public Result initialize(){
     …
     InputStream is = getClass().getClassLoader()
         .getResourceAsStream(CUSTOMER_DATA);
     …
 }
…
}
```

- Testing the time-out is difficult, as the test cases perform faster than 2 seconds.

```
public class CustomerInitialization {
 …
 private static final int MAX_DELAY = 2000;

 public Result initialize(){
     …
     long start = System.currentTimeMillis();
     … // reading and storing in between
     long end = System.currentTimeMillis();
     if(end - start > MAX_DELAY)
         throw new TimeOutException(…);
     …;
 return result;
 }
…
}
```

- Testing the IO-Exception and Repository-Exception is difficult, as the reading/storing does not cause any exceptions in the running application.

```
public class CustomerInitialization {
 …
 public Result initialize(){
     …
 try {
     …
     buffer.readLine();
     …
     customer = this.lineToCustomer(line);
     repository.store(customer);
     …
 } catch (IOException e) {
//           e.printStackTrace();
     result.addProblem(CUSTOMER_DATA, e);
     return result;
 }catch (RepositoryException e) {
//           e.printStackTrace();
     result.addProblem(line, e);
 }
 …;
 return result;
 }
…
}
```

For performing the tests, the following steps should be performed:

4.1 Setting up the Project

Create a maven-project 04.sample.application. Add the properties/dependencies of the initial hello-world-project (that is the compiler and JUnit5). Load the project from 04.sample.application to the java-folder. Add the csv-file to main/resources. You can run the project by executing `de.stuttgart.hft.dummy.DummyMain` – however, this package should be put in the test-folder and the classes `DummyMain`, `DummyDB` and `Injector` are only for demo-purpose and are not allowed to be used for the test.

Create for each of the following stages a test class, containing the respective test methods. Create individual packages for each stage.

4.2 Initalization Stage 1 Test

Add the Mockito-dependencies to your project.

4.2.1 Regular Test

The enclosed csv-file allows reading 999 records successfully. Provide one JUnit-Testcase, which checks this:

- Provide a mock for the Repository and inject it to the `CustomerInitalization`-object
- Execute the test (by calling the method `initialize():Result`).
- Assert, that the reading was successful by checking that the returned result succeeded and verify, that the method `store(Customer)` of the repository has been called 999 times and the method `commit()` has been called once.

Hint: Provide an ArgumentMatcher which can be used directly in Mockito-matches.

```
private ArgumentMatcher<Customer> any = arg -> arg instanceof Customer;
verify(…).…(argThat(any));
```

4.2.2 Irregular Scenarios

Add an additional method to your previous test class and try to provoke a Repository-Exception:

- Instruct the repository-mock that store(…) should throw a RepositoryException.

Hint (from the Mockito-manual for a mocked List):

Stubbing void methods with exceptions

```
doThrow(new RuntimeException()).when(mockedList).clear();
//following throws RuntimeException:
mockedList.clear();
```

- Assert, that the writing failed by checking that the returned result failed and check, that the returned result contains a RepositoryException.
- Verify, that the method `rollback()` of the repository has been called once.

4.3 Initalization Stage 2 Test

Prepare your project for using AspectJ. Add the following dependencies:

```
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjrt</artifactId>
    <version>1.9.5</version>
</dependency>
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.9.5</version>
</dependency>
```

Add AspectJ-Nature to your project: Properties → Project Natures → Add… → AspectJ Nature

4.3.1 Using an Alternative Sample-File

Now create a new test class which exchanges the csv-file with a smaller sample:

- The sample file (only 10 lines) is already to be found in the sample folder `test/resources`
- Create an Aspect `FileReaderInjector`
  - which crosscuts calls to `getResourceAsStream(…)`
  - and returns instead a `InputStream` which points at your test-file instead
- Execute the test (by calling the method `initialize():Result`).
- Assert, that the reading succeeded, there were ten `store()` and one `commit()`.

Hint: Create the Aspect in a way that you can supply the filename with a setter. By default the filename should be null – in this case, the original behavior should be used.

4.3.2 Using a Faulty Sample File

Now add a test method to exchange the csv-file with a “corrupted” sample:

- The sample file (again 10 lines, this time with 5 errors) is already to be found in the sample folder `test/resources`
- Create a test-data-file which contains several problems.
- Initialize your aspect accordingly.
- Execute the test (by calling the method `initialize():Result`).
- Assert, that the reading failed by checking that the returned result failed and check, that the returned result contains a `ParseException`.
- Verify, that the method `rollback()` of the repository has been called once.

Hint: Check the source code in the method `lineToCustomer(…)`, where entries might cause exceptions: (e.g. wrong date format, no number as id, etc.)

4.3.3 Testing the Time-Out

Now add a test method and try to provoke a time-out:

- Create an Aspect `TimeInjector`
  - which crosscuts calls to `System.currentTimeMillis()`
  - and returns a “slowed down” time, which causes the reader to fail
- Execute the test (by calling the method `initialize():Result`).
- Assert, that the reading failed by checking that the returned result failed and check, that the returned result contains a `TimeOutException`.
- Verify, that the method `rollback()` of the repository has been called once.

Hint: Create the Aspect in a way, that you supply the duration between two consecutive calls of `currentTimeMillis()`, if the duration is 0, the application should behave as normal, if larger than 0 each call should return the previous time plus the interval.

4.3.4 Force IO-Exception on Reading

Now add another method and try to provoke an `IOException`:

- Create an Aspect `ReadIOExceptionInjector` which generates an IOException at the relevant cross cutting points.
- Execute the test (by calling the method `initialize():Result`).
- Assert, that the reading failed by checking that the returned result failed and check, that the returned result contains an `IOException`.
- Verify, that the method `rollback()` of the repository has been called once.

Hint: Check, which statement in `initialize()` may cause an `IOException`, make a pointcut to this statement and throw an `IOException`.

4.4 Initalization Stage 3 Test – Testing the Reports[1]

Add the dependencies for Selenium. Create a new test class and use Selenium for testing the generated html-file.

- Now call the application with the correct sample and check with Selenium that the opened page includes the string "`initialized successfully`".
- In a second test method use the corrupted sample and assert that the file contains: "`initialization has failed`".

4.5 Checking the Coverage

Group all your previous test cases in a test-suite. For this you have to add the following dependencies:

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.13.2</version>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.junit.platform</groupId>
    <artifactId>junit-platform-runner</artifactId>
    <version>1.8.0-M1</version>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.8.0-M1</version>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-params</artifactId>
    <version>5.8.0-M1</version>
    <scope>test</scope>
</dependency>
```

1 Actually we imagine that the application deploys the page at a server, where we would call it.

Now create a test-suite which includes the previous test-packages. Check with eclEmma which code pieces have not been covered so far.

```
package de.stuttgart.hft.all;

import org.junit.platform.runner.JUnitPlatform;
import org.junit.platform.suite.api.SelectPackages;
import org.junit.runner.RunWith;

@RunWith(JUnitPlatform.class)
@SelectPackages({"de.stuttgart.hft._01.mockito",
                 "de.stuttgart.hft._02.aspectj",
                 "de.stuttgart.hft._03.selenium"})
public class TestSuite {
}
```

Now create additional tests (with aspects) to achieve 100% coverage. Add these Tests to a new test-class and create a new suite. (Remark: you have to reset the eclEmma -Console first manually).