

# Exercises for “Software Engineering 2“

## - *Advanced Testing with Java 1*

### Foreword

This exercise will be counted as Pre-Exam. As part of this exercise, you will simulate the testing process in a project. Please read the following description carefully. Also, document your solution steps in the course of the exercise (make screenshots, copy code, make notes, ...). You may do this exercise in groups. In this exercise you are the test team who tries to find errors in some developer's code. The more errors you find the more successful you are.

As a test-team member you first have to come up with test cases, based on a given specification. Together with the test cases you have to implement the corresponding JUnit-Tests. In parallel a developer (**not you or your team!**) comes up with the implementation, hands it over to your team. You do the tests and evaluate the results. If the test results are not satisfying you may reject the current version and request a new, fixed version from the developer. Thus, the main effort of this task lies in the steps 1-3, where you set up your environment and the tests.

We will start the exercise online in Zoom-breakout-sessions 10<sup>th</sup> May, at 14.00. You may organize yourself in groups. The exercise (and the following exercises) are done with the current Eclipse version and OpenJDK 11 (although it may also run with Java 1.8 upwards).

The role of the developer is either Manfred Besner ([manfred.besner@hft-stuttgart.de](mailto:manfred.besner@hft-stuttgart.de)) or me ([marcus.deininger@hft-stuttgart.de](mailto:marcus.deininger@hft-stuttgart.de)) who are both available for the lecture. That is, if you have set up the test environment and the JUnit-Tests are running with your dummy implementation you may request the developed functionality (you unit under test) either from him or me (the procedure to do so is described below). If there are any issues or questions during the course of this exercise, please contact us also (this time we will answer in the role as supervisors or product owners).

Through the course of the test you should document steps and upload them in the end to [Pre-Exam: Overall Test Documentation](#) until 16<sup>th</sup> May.

## Specification

A company has a program commissioned which should calculate the Christmas bonus of the employees depending on the company affiliation. In the description of the requirements the following excerpt can be found:

“Employees working for three years at the company will receive a bonus of 50% of their monthly salary as Christmas bonus. Employees, who work for more than five years in the company, will receive 75%, with a corporate affiliation of more than eight years a bonus of 100% will be granted.”

The design provides the following specifications:

A class **Employee** is to be realized. The class should hold two attributes: **yearOfHiring** and **salary** (both of type **int**). With the help of the constructor **Employee(int yearOfHiring, int salary)** both are set. An instance method **bonus(int yearOfCalculation)** provides the bonus of the employee (as an integer) for the calculation year as a result.

If the constructor receives a wrong entry year (prior to its foundation in 1990) or a negative salary, an exception (RuntimeException) is to be thrown. If an incorrect year is entered (before entering year) when calculating the bonus, an exception should be thrown, too.

**Attention:** *Your job is not to implement these functions, the classes will be available in Moodle – you should test these classes!*

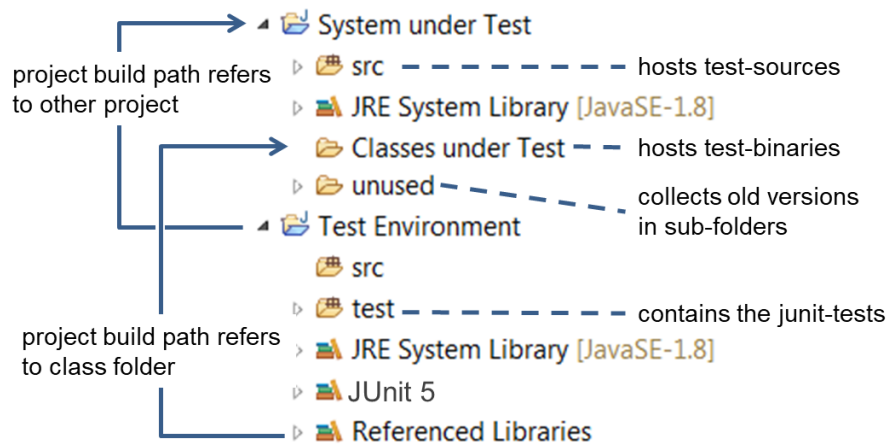
## 1 Step 1: Create Black Box-Test Cases

Define the equivalence classes for a black-box test of the constructor **Employee(int, int)** and the method **bonus(int):int** of an employee; select the equivalence classes, select representatives and define the associated test cases with expected result. Document your test cases in the following table. **Actually this is the most time-consuming part of the exercise.**

No.	Equivalence class (as a test comment)	Parameter for construction an Employee	Parameters for bonus	Expected result of Employee or bonus
1	Create Employee with negative sallary	1990, -2000		throws RuntimeException
2	Create Employee with wrong year of hiring	1989, 2000		throws RuntimeException
3	Create correct Employee but incorrect year in bonus	1990, 2000	1989	throws RuntimeException
4	Calculate bonus for less than three years	1990, 1000	1991	0
5	Calculate bonus for more than three years	1990, 1000	1994	50
6	Calculate bonus for more than five years	1990, 1000	1996	75
7	Calculate bonus for more than eight years	1990, 1000	1999	100

## Step 2: Setting up the Test System

Set up the test system (with your current Java-Version) as shown in the following figure:



- Create two Java projects: "System under Test" (short SUT) and "Test Environment".
- In the SUT create a folder (no package or source folder!) called "unused" – in this folder old versions will be stored in sub-folders.
- In the SUT create a folder "Classes under Test" – in this folder class files are stored while testing.
- In the "Test Environment" create a new "Source Folder" called "test". This is where your JUnit tests go.
- Put the project "System under Test" in the Build Path of the Test Environment (Build Path > Configure Build Path ...> Projects > Add ... " ) - this is necessary so that you can use the dummy implementation of the following step.
- Now enter the "Classes under Test" in the Build Path of the Test Environment (Build Path > Configure Build Path ...> Libraries > Add Class Folder ... " ) - this is necessary to use the test class files in step 4.

## Step 3: Programming JUnit Test Cases

For compiling the test cases, use the following (minimal, provisional) dummy implementation for the above given specification. Put it in the `src`-folder of "System under Test".

```
public class Employee {

    public static String version () {return "0 (dummy implementation)";}

    public Employee(int yearOfHiring, int salary) {}

    public int bonus(int yearOfCalculation) {return 0;}

}
```

Now create the JUnit test cases for your test cases (in the folder “test” the Test Environment).  
*Note:* Add a method

```
@BeforeAll
public static void info(){
    System.out.println ("Employee Version" + Employee.version ());
}
```

in your test class. Each version of the program provides its own version information - that helps you to keep track and determine if you are testing the correct version.

Create one test class and for each test case identified in step 1 create a test method within this class. When running the tests, almost all test cases should fail.

## Step 4: First JUnit test

*A programmer implemented the class `Employee` and releases it for testing.*

Create in the folder “unused” a sub-folder “version 0 (source)” and move the (Dummy) **Employee.java** from the previous test into it.

Request the first version of the “Unit under test” from Manfred Besner or me – to do so, call us to your breakout-session (or after the exercise send one of us an email with the topic “Unit under Test v1 requested”). As a proof, that you have prepared the setup, please upload the list of test cases and a screenshot of the executed junit-test-cases (of course most of them should have failed) to the respective Moodle-Task [Request for Unit under Test v1 \(Binary\)](#). In return you will receive **Employee.class** that should implement the above-specified behavior. Put the class into your “Class under Test”-folder.

Now run the JUnit tests. Which test cases are successful and which not? Copy the code for the JUnit test into a Word document. Insert a screenshot of JUnit tests from which the test result can be seen.

## Step 5: Second JUnit test


Let’s assume not all test cases were successful – in this case you contact the programmer (Manfred Besner or me) again, and you request a fixed version. To do so, call us to your breakout-session (or after the exercise send one of us an email with the topic “Unit under Test v2 requested”). Please upload a list of test cases which have failed to [Request for Unit under Test v2 \(Binary\)](#) so we can fix our errors.

In return you will receive a fixed version of the **Employee.class** from us. In the folder “unused” create a sub-folder “version 1 (binary)” and move the **Employee.class** from the previous test into it. Now put the new, fixed **Employee.class** into the folder “Classes under Test”.

Run the JUnit test; which test cases are successful and which not? Copy the code for the JUnit test into the Word document. Insert a screenshot of JUnit tests from which the test result can be seen.

## Step 6: Check the Code Coverage

If all test cases have been running without errors, check test the code coverage of the previous test case.<sup>1</sup> Since you do not want to measure the coverage of your test code, you must first

define the projects which actually should be measured. Select through  (the EclEmma launcher) “Coverage Configuration ...> Coverage“, the “System under Test – src” and “Classes under Test“ folders. Now, run your test by the EclEmma launcher and check the coverage of your test cases.

## Step 7: Inspection of the Executed Code

If you did not achieve 100% coverage with the previous test, request the source code of the currently tested binary, to inspect it closer. For this call us (or send one of us an email with the topic “Unit under Test v2 Sources requested”) in parallel upload a screenshot of the coverage console to [Request for Unit under Test v2 \(Source\)](#).

Now, move the previously tested file **Employee.class** to a (new) directory “unused / version 2 (binary)”. Then add the **Employee.java**, which you will receive, in the src-folder of your Eclipse project and check again the coverage of your test cases (the percentage should be the same, as it is effectively the same code). Document the actually covered source code by the screenshot of the measured file.

## Step 8: Third JUnit Test

Examine the previously achieved code coverage and find out, which (if any) code has not been covered. Create more JUnit test cases to achieve now 100% coverage. Ideally, the new test should end up in a new test class and both, old and new test classes, are run by a JUnit-test suite. Again, document your results. Document the test-case and coverage results.

*As before, it's not your job to correct errors, but to achieve through appropriate test cases 100% code coverage and find possible errors in the program. As a tester, you are actually successful when JUnit tests are red.*

## Step 9: Optional Bonus-Test

Implement an additional JUnit-Test which tests the constructor of the Employee-class. The test should create a new Employee-object and then check if the provided parameters have actually been set to the private int-fields yearOfHiring and salary of new object. Use reflection to access these fields. Document the test-case.

Upload the final document with all the test resultsto [Pre-Exam: Overall Test Documentation](#) until 16<sup>th</sup> May.

---

<sup>1</sup> The tool EclEmma is already installed at the university. If you want to install it at home use the menu item “Help> Install New Software ...” with the upload-address <http://update.eclEmma.org/>.