Practical Work: Advanced Java Programming – Mob Programming

The Goal of this practical work is to assess the student’s ability to write and complete Java code, to show his operational skills in real situation.

This Practical Work is composed of 6 domains:

1. Maven
2. Junit
3. DI with Spring
4. Persistence With JDBC through DAO pattern

Each domain needs to be completed to go on, as those domains are ordered from the most general to the most specific.

You’ll have to deliver all the projects created for that practical workshop; you must place all of them in the same folder.

Remark: you’ll have a global bonus If you follow good coding practices, like putting comments on critical code, using loggers instead of *System.out* and so on.

# Domain 1: Maven (35 minutes)

**Exercise MVN1**

*5 minutes* ***0.5pts***

Build a new java project names using maven. You must enforce the maven standard layout. You must define correct maven coordinates for your project. The project name should be **${id}-adv-java** which will be referenced as the “the project” and replaced by the variable **${project}** in the rest of this document.

**Exercise MVN2**

*20 minutes* ***2pts***

1. Have a look to the attached csv file (contains 17 contacts)

|  |
| --- |
|  |

1. Then create a class “**Contact**” with as attributes names the column titles to represent the files data.
2. Create a class (think well about the class name) able to read data from this file and transform it as a list of instances of the **Contact** Class.
3. Add a method in that Class that sorts the list by the Contact state.
4. Create a third Class **TestMVN2** that will be able to do test (not using JUnit) the deserialization feature of this program.

Think well about where you place the text file and the 3 classes.

**Exercise MVN3**

*5 minutes*

Perform a build (mvn install), store the console output in the file “**${project}/output/mvnbuild1.out**” (in the folder “output”) in your project.

**Exercise MVN4**

*5 minutes*

Add the following dependencies to the project :

* Hibernate 5.4.5-Final
* Spring-context 5.1.9.RELEASE
* Spring-orm 5.1.9.RELEASE
* Spring-test 5.1.9.RELEASE
* Log4j2-core 2.12.1
* JUnit 4.12
* Javax-inject 1.0

Launch a build (mvn -U install) and store the output in the file “**${project}/output/mvnbuild2.out**

Domain 2. Unit Testing with JUnit (20 minutes)

**Exercise JUN1**

*5 minutes*

Create a JUnit test class **TestJUN1**, make this class produce the output “*Hi from JUnit*" while building the project through Maven

**Exercise JUN2**

*10 minutes*

Create a JUnit test named **TestJUN2** in package **fr.epita.tests.${id}** and use correctly the @After, @AfterClass, @Before and @BeforeClass to manage the instance from the class that does the deserialization from exercise . The junit test itself should just display the list of persons present in the file.1

**Exercice JUN3**

*5 minutes*

Create a JUnit test with correct assertions to validate that the list produced by your deserialization class is matching the results that one would expect. Remember to follow the “given-when-then” approach.

Domain 3. DI with Spring (35 minutes)

**Exercise SPR1**

*10 minutes*

Declare your first bean thanks to the spring xml file.

This bean should be named “**myFirstBean**”, should be of type **java.lang.String** and should contain as a value **“Hello from Spring, ${id}”**. Replace ${id} by your id.

Beware of the location where you place the spring file. It should be loadable from **the root** of the classpath.

**Help :** the file root content is as follows : you can copy paste this content in your own file:

|  |
| --- |
|  |

**Exercise SPR2**

*10 minutes*

Inject the previously declared bean by using the standard JEE annotations (2 annotations to be used) in a JUnit test case. This testcase has to be named “**TestSPR2**” and has to be in the package **fr.epita.tests.${id}** in the appropriate source folder.

**Help :** Remember that Spring and JUnit do not integrate by default, you have to make appropriate use of **@RunWith** and **@ContextConfiguration** annotations

There’s a Spring Junit 4 class runner class in the Spring test dependency

**Exercice SPR3**

*5 minutes*

Considering the Person class from exercise MVN2, Create a new JUnit Test named “**TestSPR3**” in the **fr.epita.tests.${id}** package

Inject then an instance of person constructed by spring with whatever values you want, except for the person name which should be “${id}”

**Exercice SPR4**

*10 minutes*

Create an H2 embedded database instance through spring using this bean:



Write a testcase **TestSPR4** in which you will inject this **DataSource** instance and write an assertion that shows that this Database is ready to use.

Domain 4. JDBC using DAO pattern

**Exercise DAO0**

Prepare a CREATE TABLE query using H2, to create a table matching the data constraints (type, format) of the provided csv file. Run this statement before any other **@Test** method in the next DAO\* exercises.

**Exercise DAO1**

*40 minutesz*

Write a DAO **ContactJDBCDAO** able to deal with read/write operation regarding the Contact datamodel class. This DAO will use the datasource defined in the previous exercises.

Define a bean in the application context, allowing to have a singleton-scoped instance of **ContactJDBCDAO**

Write a JUnit test class **TestDAO1** to test **ContactJDBCDAO** on *create()* and *search()* methods.

You can use the csv file as a source to provide data for your test cases.

**Exercise DAO2**

*15 minutes*

Consider refactoring your code and adding abstraction to avoid using the **ContactJDBCDAO** implementation directly.

Make **ContactJDBCDAO** a Spring @Repository and use component-scan directive to make it available in the application context.

Create a Junit test class **TestDAO2** to verify that your refactoring is working properly (no regression compared to TestDAO1)

**Exercise DAO3**

*20 minutes*

Use your DAO to fill the database with all the lines present in the provided csv file. Do this operation before any other test in the **TestDAO3** class.

Use your DAO to implement the following use cases:

1. The contact Sage Wieser needs to be updated (because address contains a typo). The admin searches by email address and then modify the address (street to be set to 5 Boston Ave #188, instead of 5 Boston Ave #88), and apply the update
2. The admin needs to find all people living in New York state (code = NY) and display them in the UI (console output in our case).
3. Count the number of people per state.

Create a **@Test** method *testUCa(),* *testUCb(),* *testUCc()* per use case in the **TestDAO3** class.