Lab 3 - Object-Oriented Software Development / ... with Design Patterns TOMK18 / TOUK18 - H22

Java, OO, MVC, basic testing

last presentation opportunity: Thursday, 06 October

submission deadline: Friday, 07 October, 23:59

Jönköping University, School of Engineering

Tuesday, 13 September 2022

General instructions

In this lab we learn and explore the basics of Java by implementing a simple MVC application. Relevant background for this lab is found in lecture 3,4,5 and the corresponding readings. At the end we explore basic testing with JUnit.

1 Java tutorial

We do not give a self-contained Java tutorial. We rather give a few pointers, recommendations and aspects to focus on. On the web there is good documentation of Java and there is good tutorials on Java.

We recommend to start with a simple "Hello World". Go to `https://docs.oracle.com/javase/tutorial/tutorialLearningPaths.html`.

Under *New to Java* → *Getting Started* you find among other items *The "Hello World!" Application*. It gives you detailed instructions, using the command line and javac, java. A look at *A Closer Look at "Hello World!"* can deepen your understanding and a look at *Common Problems (and Their Solutions)* can help when you encounter problems. You can also find guidance on how to achieve "Hello World" from within the Eclipse IDE. Search the web for "hello world eclipse" and you find several alternatives.

Under *New to Java* → *Learning the Java Language* you get the basics of Java. We recommend to look at least at the first three items. Feel free to use further web resources. As preparation for this and the next lab we recommend to focus on the following aspects.

1. OO (define classes, attributes, methods, interfaces, abstract classs, inheritance, ...)
2. GUI with swing and awt, look at a few simple examples
3. arrays, 2-dimensional arrays
4. String versus char[], differences, transformations
5. java.util.ArrayList

6. Integer.parseInt(textWithNumber)
7. exceptions, try, catch, NumberFormatException

2 An MVC application

The program we want to create shall provide a GUI where the user can input text, an encryption key, press on an 'encrypt' or 'decrypt' button and the input text will be correspondingly encrypted or decrypted (there is some 'output text' on the GUI, too). To implement this program, we follow the Model-View-Controller (MVC) pattern (see lecture 5 and literature). The system lies in Model, the GUI in View and the Controller mediates between them. In our case the system is rather simple, it provides only encryption/decryption functionality according to a set encryption key. We keep things simple and we want to implement the *Caesar Cipher* https://en.wikipedia.org/wiki/Caesar_cipher.

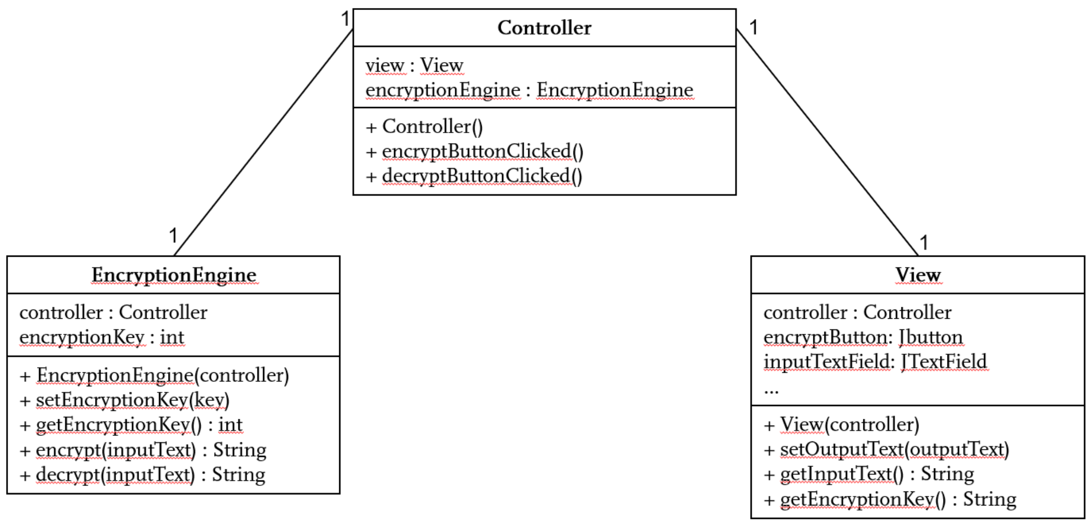
The responsibilities of our design can be defined as follows.

EncryptionEngine's responsibilities: provide encryption and decryption using a set encryption key.

Controller's responsibilities: a) take user input from the View and transform it into meaningful actions in the EncryptionEngine; b) take output from the EncryptionEngine and forward it to the View for display.

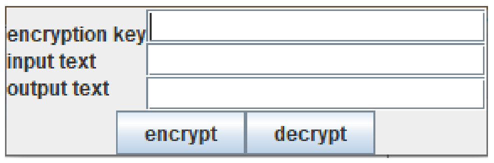
View's responsibilities: a) create and display the GUI; b) take user input and forward it to the Controller.

The UML class diagram could look like this.


Controller
view : View
encryptionEngine : EncryptionEngine
+ Controller()
+ encryptButtonClicked()
+ decryptButtonClicked()
EncryptionEngine
controller : Controller
encryptionKey : int
+ EncryptionEngine(controller)
+ setEncryptionKey(key)
+ getEncryptionKey() : int
+ encrypt(inputText) : String
+ decrypt(inputText) : String
View
controller : Controller
encryptButton: Jbutton
inputTextField: JTextField
...
+ View(controller)
+ setOutputText(outputText)
+ getInputText() : String
+ getEncryptionKey() : String
1
1
1
1

2.1 The View

The GUI shall look something like this:



Since it is cumbersome to create such a GUI when new to a language, we have already done this for you (see *Canvas→Modules→Labs→Lab03View.java* or also as archive file *Lab03.zip* to import in Eclipse via *File->Import...->General->Existing Projects into Workspace, Select archive file*). What remains to do here: implement the method `getEncryptionKey()` and the ActionListeners of the buttons need to call a corresponding method in the Controller.

2.2 The EncryptionEngine

You create the EncryptionEngine yourself. The getters and setters and constructor are straight forward. The methods `encrypt` and `decrypt` are a little more sophisticated. Hints: use `char[] inCharArr = inputText.toCharArray();` to transform the String `inputText` to a char array. Then loop over its elements and translate every element according to the Caesar cipher rules (substract from `inCharArr[i]` the char 'A', then add the encryption key, then take modulo 26, then add again the char 'A'). For this lab it is sufficient if your EncryptionEngine can handle capital letters A to Z.

2.3 The Controller

You create the Controller yourself. The constructor shall create the View and the EncryptionEngine and store it in the Controller's private attributes. `encryptButtonClicked()` shall extract the input text from the View, extract the encryption key from the GUI, set in the EncryptionEngine the encryption key, use the EncryptionEngine to encrypt and finally make sure that the encrypted text is displayed in the GUI under 'output text'. `decryptButtonClicked()` works analogously.

2.4 How to start it all?

As you may have noticed, in our design the Controller creates all other required objects (View and EncryptionEngine). Therefore, creating a Controller object will launch our program. A standard way to do so is to create an additional class, say Launcher, that contains a static main method which creates a Controller object. The Launcher file can be run (because it has a main method). For instance like this:

```
public class Launcher {
    public static void main(String[] args) {
        Controller controller = new Controller();
    }
}
```

3 Your Task

1. Implement the design as discussed. Deliver a working Caeser cipher with GUI! (that means, a click on the 'encrypt' button encrypts according to the indicated encryption key, analogously for 'decrypt')

2. If all works as required, think about invalid input (e.g. user enters letters into the textfield for the encryption key). Think about the responsibility of taking care of

invalid input. Who does this? View or EncryptionEngine or Controller? Why? Have a motivation for your choice.

3. Take care of invalid input, according to your choice from the previous point. What happens if the user enters letters into the textfield for the encryption key? An error message of some kind should be displayed[1]. Connected to this: should the method `getEncryptionKey` return an integer or a String? What happens if the user enters symbols or characters into the input text that are not in the range A-Z? An error message of some kind should be displayed.

4 Testing

When you feel you are done with the implementation, we want to write some simple tests with JUnit and see if your encryption and decryption methods work correctly. Writing tests is typically done even before implementing the code to be tested. But in this lab we do it afterward.

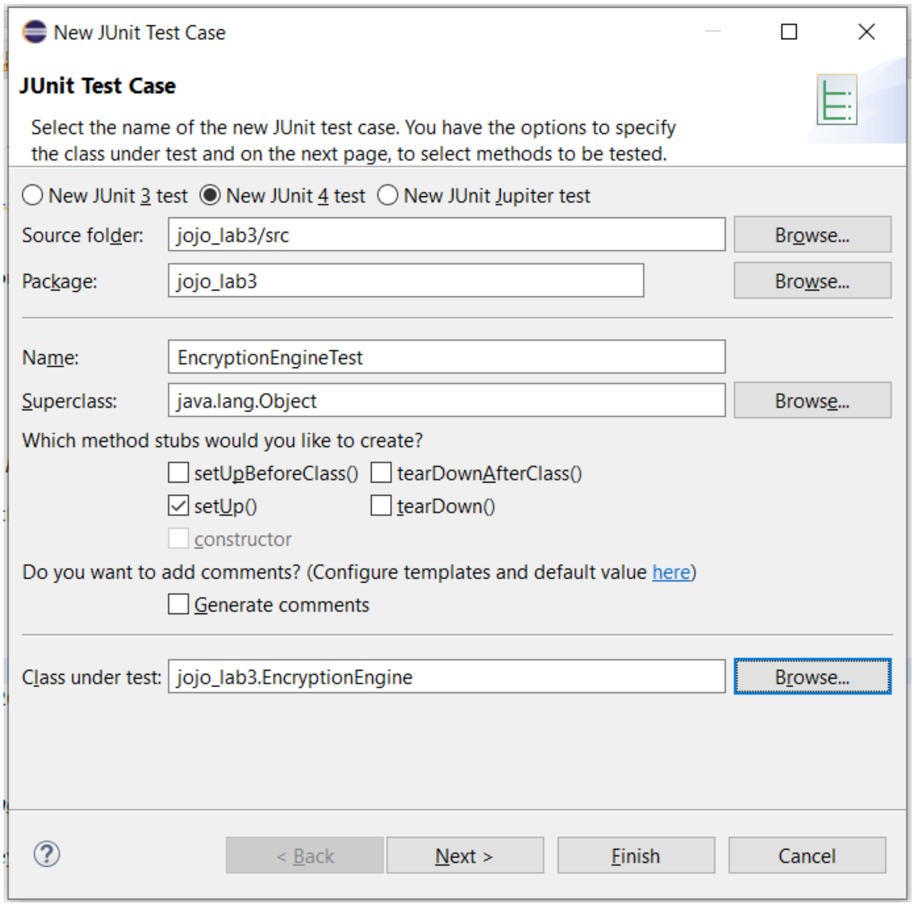
Right-click on the project's source folder (usually 'src') and select to add new *JUnit Test Case*. A configuration window opens that you should fill in as in Figure 1 (make sure to name it *EncryptionEngineTest*, select *New Junit-4 test*, check *setUp()* and select your EncryptionEngine class to test). Click on *Next* and select to test only the encryption and decryption methods. Click finish. If Junit 4 is not on the build path, accept to add it. Now your unit tests are created. You run the tests by right-clicking on the 'src' folder and selecting *Run As* and then *JUnit Test*. If you do so, two failures should occur, because the tests are not implemented yet.

Your task is now to implement the two tests. Tests are usually constructed in three steps: 1. *Arrange* (perform required set-up), 2. *Act* (call the methods to be tested), 3. *Assert* (verify that the methods return the expected data). You can do some more complex setup in the setUp() method, or you can do it in the test methods directly. For instance, `testEncrypt()` could be implemented as follows[2].

```
public void testEncrypt() {

    // Arrange / set-up
    Controller controller = null; // dummy controller
    EncryptionEngine e = new EncryptionEngine(controller);
    e.setEncryptionKey("3"); // String or int, depending on your implementation

    // Act and Assert in one line
    assertEquals(e.encrypt("ABC"), "DEF");

    // Another Act and Assert
    assertEquals(e.encrypt("XYZ"), "ABC");
}
```

[1] An easy way is to let some messagebox pop up. A nicer way would be to have a special area in the GUI that displays error messages when needed. This area needs to be cleared from error messages when the error is resolved...

[2] In case your EncryptionEngine does work with a null-object as controller, you will find a workaround


New JUnit Test Case

JUnit Test Case

Select the name of the new JUnit test case. You have the options to specify the class under test and on the next page, to select methods to be tested.

( ) New JUnit 3 test [✓ New JUnit 4 test] ( ) New JUnit Jupiter test

Source folder: jojo_lab3/src

Package: jojo_lab3

Name: EncryptionEngineTest

Superclass: java.lang.Object

Which method stubs would you like to create?

☐ setUpBeforeClass() ☐ tearDownAfterClass()

☑ setUp() ☐ tearDown()

☐ constructor

Do you want to add comments? (Configure templates and default value here)

☐ Generate comments

Class under test: jojo_lab3.EncryptionEngine

Figure 1: JUnit configureation window.

Feel free to add more Act/Assert statements, also with different set-ups (test different encryption keys, especially also 0). Note that there are many more assert methods, not just assertEquals(), confer `https://junit.org/junit4/javadoc/4.11/org/junit/Assert.html`. Implement also `testDecrypt()` accordingly. Finally, run the tests. In case they still fail, figure out where the error lies (either in the tested code, or in your test case) and fix the error. When the tests run without failure, introduce an error in your implementation (e.g. in the method `encrypt()`) and check whether your tests detect it. If you are done with all that: congratulations, you now know the basics of (automated) testing!

5 Deliverables and presentation

Show your working program to your lab assistant and orally present your code. Be prepared to answer any question regarding your code. Then submit your code on Canvas under the assignment "Lab 3". Either in a zip-file containing several text files, or as a zip file generated by exporting your project from eclipse (*File->Export...->General->Archive File*).