UNIVERSITÄT ZU LÜBECK
INSTITUTE FOR SOFTWARE ENGINEERING
AND PROGRAMMING LANGUAGES

isp

Development of Safety Critical Systems (Sichere Software) Winter term 18
Martin Leucker, Karam Kharraz, Felix Lange November 8, 2018

Exercise Sheet 1

Deadline: 8th/12th of November 2018

Task 1.1 Writing Mock Objects

Mock objects are used to write unit tests for classes that communicate with other classes. Mock objects allow us to test one specific class and not the whole system. Consider the following interface of a provider that can pass on a notification to all its registered listeners.

```
1 public interface Listener {
2   void notify(Object argument);
3 }
4 public class Provider {
5   public void addListener(Listener listener);
6   public void removeListener(Listener listener);
7   public void clearListeners();
8   public void notifyListeners(Object argument);
9 }
```

The implementation of `Provider` can be downloaded on the website. We want to implement the following test cases for this (correct) implementation: The setup contains a provider `provider`, and a listener `a` added once and a listener `b` added twice to the provider. After running

```
1 provider.notifyListeners("Two");
2 provider.notifyListeners("One");
3 provider.notifyListeners("Two");
```

we assert that

- `notify("One")` was invoked once on `a`,
- `notify("Two")` was invoked twice on `a`,
- `notify("One")` was invoked twice on `b` and
- `notify("Two")` was invoked four times on `b`.

To implement these test cases we need a mock object implementing the `Listener` interface and counting the invocations of `notify` grouped by their arguments.

a) Implement the test cases using a mock object written manually.
b) Inform yourself about Mockito[1], which is already included as dependency in the maven project. Write the same test suite again using Mockito.
c) Install EMMA in your favorite IDE[2] and check the coverage of the test cases. Adopt them to get full coverage of the `Provider` class.

Use the maven project provided in the Moodle course.

[1] `http://mockito.org`
[2] E.g. install EclEmma (`http://www.eclemma.org`) in Eclipse as described in the lecture.

Task 1.2 Test Driven Development (TDD)

TDD has been introduced in the lecture. In this exercise we want to learn how TDD works based on a real example. In the following, we want to write a method that takes an Integer resembling a number and return the corresponding Roman numerals[3] as a String. The method `toRoman` is given in the class `Roman` of the code which you can find in the Moodle course. Furthermore, a test class `RomanTest` is given to write tests for the method `toRoman`.

Now, do the following in order:

a) Add a new test case which tests an easy case. Choose this method

```
1 @Test
2 public void testI() {
3   Roman r = new Roman();
4   assertEquals(r.toRoman(1), "I");
5 }
```

If you run this test case, it will fail.

b) Extend the method `toRoman` in the easiest way, such that it fulfills the test case you wrote before. You should now refactor the code to make it look nicer, but there should not be much to refactor, so leave it as is.

c) Repeat this for 2 and 3 which should result in "II" and "III" by first writing a test method `testII` and `testIII`, respectively, and then implementing the additional behaviour in the easiest way again (using if-clauses). Now you have to refactor the code. You see that it looks ugly with all the if-clauses and these are just producing a number of I depending on the input value. Replace the code with a for-loop which produces I's corresponding to the input number. Use for example an empty string and extend it (+ on a String) with "I". Afterwards, your tests should still be fulfilled.

d) Go on with 4 which should result in "IV". Again, write a test and implement the behaviour in your code. Do the same thing for 5 which should result in "V". Refactor your code when you see a possibility to make it look nicer or if you just think it is hard to read, but keep the cases covered, do not go too crazy.

e) Go on by writing more tests for increasing numbers. Only write tests for cases which you think should not work by now with your code. You can see how your method becomes more complex while still being refactored to be readable. Also you have tests for all cases you had a look at by now and if you change your method, you can directly see if you broke some test case that already worked. It is a really structured way of writing code. If you like, you can now still go on and implement more and more numbers. Over time, you should get a good feeling of which cases are still missing and how you add code for new cases.

[3] https://en.wikipedia.org/wiki/Roman_numerals