Learning goals

Before the next day, you should have achieved the following learning goals:

- Understand the importance of automated testing.
- Write your own tests for classes already defined.
- Execute your own tests for classes already defined.

You should be able to finish most of non-star exercises in the lab. Remember that star exercises are more difficult. **Do not try star-exercises unless the other ones are clear to you**.

1 Install JUnit 4

Use your favourite search engine to find JUnit ("JUnit download" will do, and probably just "JUnit" will do too).

- Download the latest stable version of JUnit 4. You may download it either as a ZIP with a JAR file inside or as the JAR file directly. The JAR file will have a name similar to `junit.jar` (maybe including a version number, similar to `junit-4.11.jar`).
- Download the latest stable version of HamCrest too.
- Place both JAR files in a place where you can find it. It is probably a good idea to create a folder "lib" where you place all the external libraries that you use (e.g. `h:\lib` on windows or `/opt/lib` on Unix systems).
- Now you can add both JAR files to your classpath at the command line (as in the notes) or by modifying the environment variable CLASSPATH.

2 Testing mathematical functions

On Day 7 you implemented a simple hash. Write a battery of tests to verify its behaviour, paying special attention to border cases.

Hint: Implement a loop that tries a fair amount of random numbers (around two thousand, for the purposes of this exercise) and verify that the output is within the range.

3 Practice "Find bugs once"

The method `getInitials(String)` has a bug! If you introduce a name with more than one space between words, it throws an exception.

Create a class that contains the method `getInitials(String)` as described in the notes. Create also the test class as described in the notes.

Then follow the "find bugs once" algorithm: reproduce the bug manually, reproduce the bug programmatically by adding a new test to the testing class, then fix the bug and check that all tests pass.

4 Test implementations of a given interface

You already know that an interface is a way of describing the behaviour of a class without any knowledge about the implementation details. Sometimes, one party provides the interface of a component and the other party implements the interface. This is very common in big projects, where small teams of programmers make parts of a bigger program (e.g. web browsers, word processors, multiplayer games), and the different modules need to communicate with each other. Defining clear and simple interfaces is usually the first step in the design, as it allows different teams to work in parallel and then bring their code together.

Sometimes, the first party does not only define the interface, it also implements the tests that the implementation (i.e. the class that implements the interface) must pass[1]. This is a good idea when the development is sub-hired to an external company.

[1] When this approach is taken to its logical conclusion, we are talking about Test-Driven Development as we will see very soon.

Take the role of a project leader and implement the tests for two of the interfaces you have implemented in past weeks.

4.1 Stack

The notes from Day 8 implemented a Stack interface in two different ways. Create a battery of tests that verify that the classes implementing the interface is working as expected. Use it to test both implementations.

4.2 Queue

You implemented a Queue interface —maybe in two different ways— on Day 8. Create a battery of tests that verify that the class(es) implementing the interface work/s as expected.

4.3 Set (*)

You implemented a Set interface —possibly in two different ways— on Day 9. Create a battery of tests that verify that the class(es) implementing the interface work as expected.

5 Testing dynamic structures

Write batteries of tests to verify the functionality of the dynamic structures you have created in past weeks:

- doubly-linked list (day 7)
- circular list (day 7)
- simple map (day 8)
- sorted list (day 7)

Make sure that you test border cases, including situations like:

- Adding the first element.
- Removing the last element.
- Adding the first element and then removing it... and then adding another one.

You should have time to do at least one the four cases in the lab.

6 More tests (**)

If you have finished with the other exercises, write additional batteries of tests for other programs (in particular, the exercises marked with a star) that you have written in past weeks. Some exercises that provide a harder challenge to test properly are:

- the anti-aircraft game from day 6
- any of the sort algorithms from day 7
- any of the unfair queues from day 8
- the hash-table (day 8)
- deletion of elements in a tree (day 9)
- re-balancing of a tree (day 9)
- the abstract syntax tree (day 9)
- the pseudo-git tree (day 9) (**)