Unit test/TDD lab - Malmö Högskola.

Magnus.Krampell@mah.se

1. Introduction

This is a lab aimed at giving an introduction to Unit Tests and TDD (Test-Driven Development).

- Unit testing is a central concept within development and test, where the developer (most often) creates test cases and executes these.
- Unit tests are often created and executed within a test framework (e.g. JUnit)
- **Unit tests with JUnit will be tried in this lab**
- TDD (Test-Driven Development) is a method, where test cases are written before the tested code has been written. The test cases thus fails when executed the first time. The tested code is then written and the test cases are re-run. This time the test cases should pass.
- Test cases written in a test framework can be used as a basis for automatic tests. Whenever a piece of code has been modified, the corresponding tests can be re-run to make sure test cases that passed earlier still pass.
- Automatic tests are useful when performing refactoring – restructuring the code without changing its interface or functionality.
- **TDD will be tried in this lab**

JUnit and the plugin EclEmma (for code coverage measurement in Eclipse) are already installed in the university environment. If you use your own computer, you need to install these tools yourself (read installation instructions on the web).

Help each other to install and get the development environment up and running! As soon as one person succeeds, tell the others how to do it!

2. Assessment

When you have done both parts of the lab, report to your lab assistant for assessment. The assessment will look at:

- Your answers to the questions (see below)
- You test cases (code in JUnit) for part 1
- You test cases (code in JUnit) for part 2
- Your code for implementing the Clock (including Time and Date) in part 2

If you do not finish in time to be assessed during the lab, your results (see list above) shall be uploaded to It's Learning (in the assignment for the lab).

3. Overview

In the first part of the lab, a piece of code is given and the task is to write Unit test cases for it. The JUnit framework, including a tool to measure code coverage, shall be used.

Test cases shall be written so that a certain level of code coverage is reached. The test cases should also reveal any defects in the code. Once these defects have been fixed, the test cases shall pass.

In the second part of the lab, a specification is given. Test cases shall be written and executed. They shall fail when executed. The specification shall then be implemented and test cases re-run. They shall pass when executed.

4. Part 1

a. Task

Start up the development environment (e.g. Eclipse for Java) and get acquainted with the test framework (JUnit) and the code coverage tool used (a special button is used instead of the “Run” button).

QUESTION (3a1): What type of code coverage is measured by the tool (statement, branch, path or condition)? Is it possible to choose more than one type?

b. Task

Load the code **rovar.java** from It’s learning (unit test lab) into the development environment and make sure it compiles. The code converts strings to and from “rövarspråket” – all consonants are duplicated and an “o” is inserted between them. vowels, numbers and other characters are kept as they are. “rovar” thus becomes “rorovovaror.

Rövarspråket (English: The Robber Language) is a Swedish language game. It became popular after the books about Kalle Blomkvist by Astrid Lindgren, where the children use it as a code, both at play and in solving actual crimes.
The principle is easy enough. Every consonant (spelling matters, not pronunciation) is doubled, and an *o* is inserted in-between. Vowels (and other characters, e.g. numbers) are left intact. It is quite possible to render the *Rövarspråket* version of an English word as well as a Swedish, *e.g.*:
sos-tot-u-bob-bob-o-ror-non or *sostotubobboborornon*
that syllable chain would mean *stubborn*. Needless to say, the code is not very useful in written form, but it can be tough when spoken by a trained (and thus quick) user. On the other hand, for an untrained speaker, a word or phrase can often be something of a tongue-twister or a shibboleth.
Today, the books (and subsequent films) are well known in Sweden, so the language has become integrated in the culture of schoolchildren. Most Scandinavians are familiar with it.

c. Task

Create a test module in JUnit and create a set of test cases (Equivalence Partitioning yields minimum 3+3 test cases, but then the class of “non-empty string” will be a very long string, so it pays off to split this into smaller chunks, creating more test cases) that tests the method(s) of the tested code. Use different test methods to **find suitable test data** to be used. Make sure you have considered the following:

- Have you:
    - Checked “non-empty string”, containing:
        - checked all characters (Lower case as well as UPPER CASE)?
        - checked allNumbers?
        - checked a reasonable set of other characters?
- Have you:
    - checked “empty string”?
    - checked “null pointer”?

The following web site may be useful for creating the strings:

http://www.rövarspråket.se

QUESTION (3c1): What is the maximum code coverage you reach when executing your tests?

QUESTION (3c2): Which defects did you find?

QUESTION (3c3): How did you correct the defects?

QUESTION (3c4): Do all test cases pass once the defects have been corrected?

5. Part 2 – Test and implement a Clock, using TDD.

The functionality and corresponding state machine for this task has been presented in Lecture 2 (including state diagrams). **It is assumed that the student is acquainted with the state machine.**

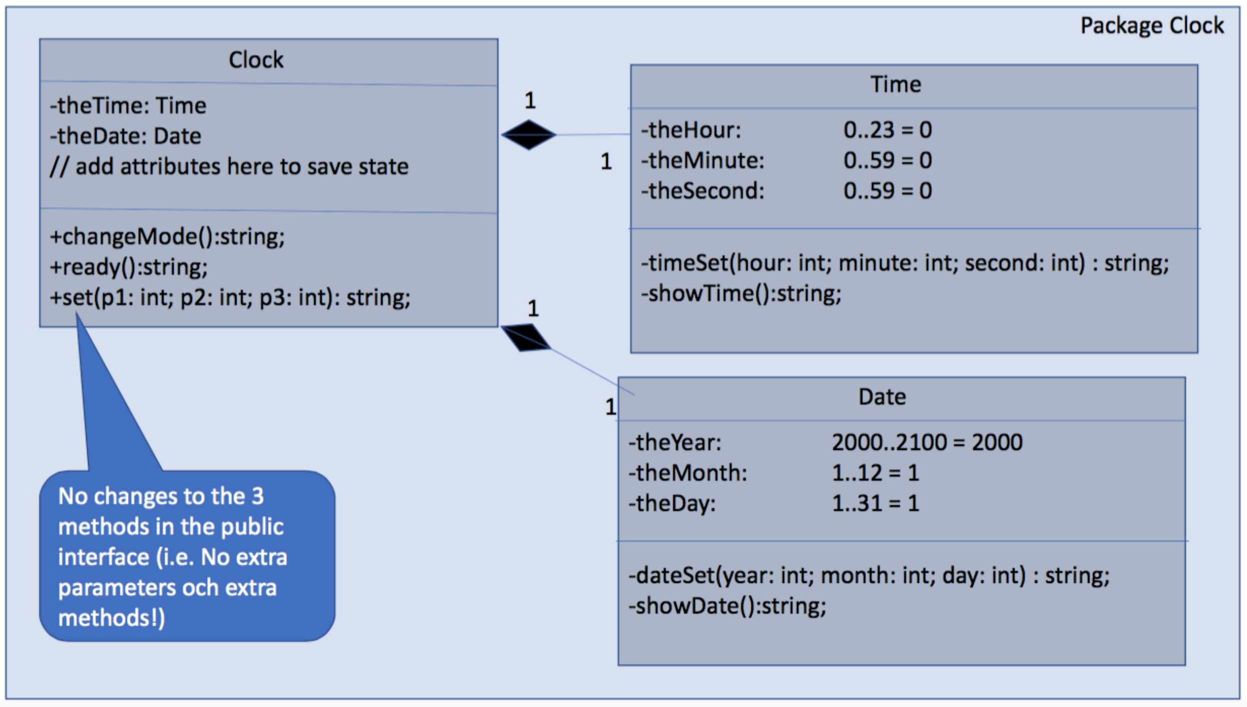
Note, that the ambitions here is to practice testing, NOT describing the best way to implement a clock! DO NOT propose a different way to do the state machine!

Also, Use the following concepts when implementing the Clock:

- Use an `enum` type to represent the 4 states. Give them appropriate names.
- Use a switch statement and **handle ALL 4 STATES in ALL 3 methods**!

a. Task 2.1

Consider the following class diagram:


Package Clock
Clock
-theTime: Time
-theDate: Date
// add attributes here to save state
+changeMode():string;
+ready():string;
+set(p1: int; p2: int; p3: int): string;
Time
-theHour: 0..23 = 0
-theMinute: 0..59 = 0
-theSecond: 0..59 = 0
-timeSet(hour: int; minute: int; second: int) : string;
-showTime():string;
Date
-theYear: 2000..2100 = 2000
-theMonth: 1..12 = 1
-theDay: 1..31 = 1
-dateSet(year: int; month: int; day: int) : string;
-showDate():string;
1
1
1
1
No changes to the 3 methods in the public interface (i.e. No extra parameters och extra methods!)

The task is to test the **3 methods** in the Clock class (**no extra methods may be defined!**) forming the interface of the Clock class. The Time and Date classes are internal and shall NOT be tested directly – only through the Clock class.

No user interface or main program is needed. The JUnit test shall call the methods and verify that the expected result is given in each situation.

Each of the 3 methods returns a string. Depending on the internal state of the clock**, the string shall be different, so that the current state can be inferred**. Assume that S1 (show time) is the default state.

Where the specification is not clear, ask or make decisions about how your implementation shall work (e.g. what is returned from the "Ready" method? What happens when you try something invalid?)

b. Task 2.2

First create the Clock class, but only containing empty implementations of the 3 methods. (all methods shall return null).

c. Task 2.3

Then create enough test cases in the test framework (JUnit) to test the Clock class.

- Create one test case for **each state transition test** you need (see also question 4c1 below). Proposal: check the default values in the Time and Date objects during these tests.
- Test that the **correct operations can (only) be performed in the correct state** – check the 6 illegal transitions - see table in the lecture slides.
- Create **test cases for each border in the Time and Date object.** (remember, there are 3+3 values, each with 2 borders. A minimum of 2 test per border yields (3+3)*2*2 = 24 test cases!)

Consider test methods presented in the lectures so far and create test cases so that the clock class is well tested.

Note! The Time and Date classes shall only store the time and date set so that a set time or date can be tested as the returned result. (i.e. they shall **NOT update time and date from the operating system** automatically)

QUESTION (4c1): How many state transitions (compare 0-switch, 1-switch, etc. mentioned in the lecture) are reasonable to test in this case?

d. Task 2.4

Run all tests. All test cases should fail. If they do not, correct/update them!

e. Task 2.5

Implement the functionality of the clock, including the time and date classes and the 7 related methods in the 3 classes. Do not overwork (e.g. no need to handle different number of days in different months – ok with 31 days in all!), but consider error conditions. Make sure to consider the state!

f. Task 2.6

Run the tests again. All tests should now pass. If they do not, correct the implementation of the clock so that the tests pass. If the test was incorrect, correct it and think about why you created an erroneous test case.

QUESTION (4f1): How much code coverage do you reach with your tests?

g. Task 2.7

Add test cases and Run the tests again until you reach close to 100% statement coverage or >80% branch coverage.