Day 9: Software Testing

1 Software testing

I know your code has bugs. If it is a small project it will have a few bugs. If it is a big projects it has a lot of bugs. I know.

How do I know? Because I know you are a human being, and human beings make mistakes. When a programmer is writing code there are too many things to keep in the short-term memory at the same time: data structures, communication between different parts of the program, flows of information... Every program except the most trivial ones is too complex for the human mind to see in its entirety. This is why we use methods to isolate operations, and why we use classes to group different behaviours and states in our program around some conceptual ideas that we can grasp (e.g. a supermarket has queues, a queue has people, a person has a name and knows who is behind in the queue; a company has products and employees, employees have names and take money from the company, products give money to the company).

Structured and object-oriented programming are ways in which programmers can reduce the complexity of their programs to levels that are (more or less) manageable for their human minds, but it is still impossible to write programs that do not contain bugs. Humans, even the best programmers, have a tendency to forget some details of the program. That is the way the human mind works: it concentrates on the most fundamental aspect and forgets the details until needed... which in computing means when the program is already executing and nothing can be done about it.

Long story short, as long as programming is performed by humans, programs will have bugs. The compiler can make sure that a program is syntactically correct but it cannot tell if it makes sense or even whether it will do what the programmer expects.

That is why software testing is important. Testing a program is a way to ensure that the program does what it should. There are two types of testing: manual and automatic (Table 1). You are already familiar with the former type: you have been doing it for weeks, for those programs you have been writing until now. Now we will learn how to do things properly in an automated way.

	Manual	Automated
Speed	Very slow	Very fast
Focus	Most important bugs known	Every single bug known
Thoroughness	Sometimes forgets to test some things	Tests everything always
It feels...	Very repetitive and boring	Nothing: work is done by computer

Table 1: Manual testing vs automated testing

```
Programmer 1: The program has messed-up and now we have lost data
    from our clients!
Programmer 2: How is that possible?
Programmer 1: The function returnPastPurchases() returned null,
    the system got a NullPointerException and crashed.
Programmer 2: Oh! It returned null because there were no past
    purchases.
Programmer 1: But it has always returned an empty array!
Programmer 2: But I thought that null was more elegant than an
    empty array. If there is nothing to return, why not return
    null?
Programmer 1: What about... because the rest of the program
    expects an empty array! Now we face our clients sueing us!
Programmer 2: I am sorry...
```

Figure 1: Scenario 1: unmet expectations

1.1 Automated testing

The idea behind automated testing is very simple: make a "testing" program that verifies whether your "main" program behaves as expected. You can execute this "testing" program every day as your original program grows to make sure that everything works as it should. Figures 1 and 2 provide two examples from real life when programs did not behave as expected.

Programs can behave badly for an infinite number of reasons, including:

- bad programmers writing bad code.
- lack of communication between programmers in a big project.
- programs without documentation that have to be modified, usually by a programmer that has nothing to do with the original programmers.
- programs with *obsolete* documentation that have to be modified, usually by a programmer that has nothing to do with the original programmers.
- version changes in external libraries (or even the programming language) that are not backwards-compatible.
- changes in one part of the program affect some other part of the program, apparently unrelated.

The last reason is particularly important because it is the most common and it is quite difficult to fight. Big programs are usually written by many programmers, and none of them knows everything about the code. Sometimes a change in one region of the code breaks something else that had been working seamlessly until then. The problem becomes worse as programmers leave the project and other programmers join in with no previous knowledge about the program. This results in programs that are more and more difficult to modify and adapt as they evolve over time. This is know as *code viscosity*, and it is bad. Automated testing, by means of regression tests, is a way of fighting

```
Programmer 1: Good, now we have found the bug. Method
    getTaxReturns() had problems when it received a null
    TaxPayer. We have fixed the method that was returning
    a null TaxPayer. Let's create a test for it to make sure it
    does not happen again.
Programmer 2: No need to. It will never happen again. I will never
    forget this week working 20h a day! I promise you no method
    will return a null TaxPayer ever again.
(...one year later...)
Programmer 1: There is a serious bug in the system!
Programmer 2: I have found it! Someone returned a null TaxPayer.
Programmer 1: What? A year ago, you promised me that it was not
    going to happen again!
Programmer 2: It is not my fault! Someone else must have made
    changes to the core classes!
```

Figure 2: Scenario 2: Star Bugs: Return of the Bug

code viscosity: when the automated tests verify that everything is still working —even those parts of the program unknown to the programmer—, one can be confident that the last change did not brake anything.

1.2 Creating tests

How do you write your own tests for your Java programs? You write them as Java programs, using a special library called JUnit. There are other ways, but JUnit is very simple, powerful, and well-known. JUnit does not come with Java by default, but you can easily install it from the web as it is free software[1].

An automated test is just a collection of methods that will be executed by JUnit. These methods will test the methods in your "main" classes, and check that they return the right values. If they do not, for any reason, JUnit will raise a flag so that you can look into your code and find out why it is not behaving as expected.

Let's see an example. Imagine that we have a method that returns the initials of a name in a class called `Person`:

```
public String getInitials(String fullName) {
    String result = "";
    String[] words = fullName.split(" ");
    for (int i = 0; i < words.length; i++) {
         String nextInitial = "" + words[i].charAt(0);
         result = result + nextInitial.toUpperCase();
    }
    return result;
}
```

[1] *Free* as in free speech, not free beer.

A test program would be a Java class that could look like this:

```
import org.junit.*;
import static org.junit.Assert.*;

public class PersonTest {
   @Test
   public void testsNormalName() {
        Person p = new Person();
        String input = "Dereck Robert Yssirt";
        String output = p.getInitials(input);
        String expected = "DRY";
        assertEquals(output, expected);
     }
}
```

As you can see, there are several things that make a test class (using JUnit) different from a normal Java class. First of all, there is no main method. JUnit tests are not executed directly, they are executed by JUnit, so there is no need for a main method.

A second important difference are those `import` statements at the beginning, before the class definition. These statements are the way in which a programmer can tell Java to use classes that are not in the current folder (they must be in the *classpath*, though). If you want to import just some static methods from some class, you can use *static imports*. In this example, the method `assertEquals(String,String)` is statically imported from class `Assert`. This is only a convenience so that you do not need to write `Assert.assertEquals(....)` every time you need it, only the method's name is enough.

The third difference is the most important one. There is a special kind of word before the method definition, a word starting with an at symbol: `@Test`. These at-words are called *annotations* in Java, and have a special meaning for the compiler. In this case, the annotation `@Test` tells JUnit that the method `testsNormalName()` is a test to be run by JUnit. This is the way JUnit knows which methods to run and which methods to ignore. (We will learn more about annotations in the future). In conclusion, you must write `@Test` before each of your test methods.

You can also appreciate some conventions in the code. The test class for class `Person` is called `PersonTest` (some people prefer `TestPerson`). The method's name tells the story of the test: it can be read as "this method"...`testsNormalName()`: this method tests a normal name. You can also appreciate some of the variable names: `input`, `output`, `expected` (for expected output).

The tests ends with the method `assertEquals(String,String)`, statically imported from class `Assert`. This method performs an *assertion*, a test that the parameters verify some condition (equality in this case). If that is not true, the method will throw an exception and JUnit will report a failure in your test. Assertion can be made on equality, difference, nullity, and truth value. The most common methods are `assertEquals()`, `assertTrue()`, and `assertFalse()`. The full list of methods of `Assert` can be found in the documentation of the class[2].

[2] http://junit.sourceforge.net/javadoc/org/junit/Assert.html

1.3 Running tests

To run your tests, you must run JUnit from the command line. In order to do so, you must bear in mind some facts:

- JUnit is not part of core Java, so it must be added to the *classpath* either manually or by modifying the environment variable CLASSPATH.
- When you modify the CLASSPATH, you must remember to add the current folder (or Java will not find your class!).
- The core class in JUnit is called, perhaps unsurprisingly, `JUnitCore`.

With this in mind, you are now able to run the following command:

```
> javac -cp .:junit-4.10.jar PersonTest.java
```

This will compile your test class(es). The `-cp` parameter is the way you can modify the classpath when executing `javac` or `java`. Remember that elements in the classpath are separated by colons on Linux and Mac, but they are separated by semicolons on Windows. To run the tests, you must execute:

```
> java -cp .:junit-4.10.jar org.junit.runner.JUnitCore PersonTest
```

If all tests run successfully, JUnit will congratulate you. If one or more of the tests fail, JUnit will inform you of which tests failed and why.

1.4 Designing tests

So far we have seen how to create automated tests, and how to execute them in order to find bugs in our *production code* (i.e. our "main code", the program that we are creating to solve a problem). Programmers usually create a lot of tests to verify the behaviour of their classes (i.e. as defined by its public methods, its interface).

But there is something that we need to bear in mind at all times: tests do **not** prove that your code does not contain bugs. No matter how many tests you have written, no matter how many cases you cover with them, there is no way to prove that a code does not contain bugs. We can only try to push the chance of bugs as low as possible, but it is never zero.

You can think of tests as torches in a cavern, where the cavern represents your code. The more torches you place in the cavern, the more light there will be, so there will be fewer shadows where bugs can hide. But you must place your torches carefully. If you just put a lot of torches in one place, most of the cavern will be in shadows, and your bugs will lurk in there until the time is right to appear (usually to make your company look bad or lose money or both). If you place your torches carefully and illuminate as many corners as possible, your bugs will find it more difficult to hide undetected. How can we ensure that we keep our cavern as illuminated as possible? There are three basic strategies.

1.4.1 Testing basic functionality

The first tests that you must write, the first torches you have to place in your cavern, are those that shed light on the main features of your code: if you have written a class with a method that returns the number of colours in an image, write a test that provides an image and checks that the right answer is returned; if you have written a method that returns the initials in a name, write a test that checks that the right initials for a name are returned.

This is the simplest strategy, and everybody has it in mind, so there is no need to hammer it anymore. The next two strategies are usually forgotten by many programmers, so it is important that you think about them carefully.

1.4.2 Testing border cases

Good code is simple, clear, and general. Programmers create programs that repeat the same operations most of the time: checking the salaries of all employees in a company, verifying the medicine doses of all patients in a hospital, calculating all stress tensors for every point on the wing of a plane, etc. Usually, this is performed correctly in most programs. If it is not, the error tends to be quite evident and is detected with the simple "basic" tests (or even at compile time).

Basic functionality is not the main source of bugs. Programs have a tendency to break around the *borders*. Using the metaphor of the cave, you can think that corners are usually darker than the rest of a cave room, so they are good hiding places for bugs. Border cases (the first, the last, etc) are usually slightly different from the general case and your implementation may break there: a list may not remove the first element correctly, or the last; a loop may not check the last element in a list, or go out of bounds; a method may break when it is given an empty list as a parameter (or null).

Border cases must always be checked in your tests. This is very important. Leaving corners and borders in the shadows is just a disaster waiting to happen. Many programmers forget to test border cases and are later surprised by bugs in a code that seemed to be "working fine".

1.4.3 Find bugs once

This is the most important rule in software testing, yet it is usually forgotten by many programmers (especially lazy programmers). Whenever you find a bug in a code, and you will find many in your life as a programmer, you have to follow this simple algorithm:

1. Find the bug again. Repeat the same steps until you know exactly how and when the bug appears. This is called *reproducing the bug* and it is not always easy in large, complicated programs.

2. Once you have been able to reliably reproduce the bug manually, write a test that reproduces the bug programmatically. Check that the bug is always fired when you run your test with JUnit. Make the test as simple as possible.

3. Once you have written the simplest test that reproduces the bug —**and only then**— fix the bug. Verify that the tests passes.

Following this simple algorithm will ensure that the same bug does not appear again and you never find yourself in the situation described in Figure 2. You are a human being and you make

mistakes. No matter how careful you are, your code will have some bugs. Most of the time you will find bugs in code that was written by other programmers that were not as careful as you. Following the "find bugs once" algorithm will at least ensure that you do not have to track the same bug twice and repeat work you have already done. This is a algorithm to make sure your projects only move forward, never backwards, and you do not waste your precious time.

Final note: Running tests overnight

The good thing about automated tests is that you do not need to run the tests yourself over and over again. You write the tests and then tell JUnit to run them. Every day, you can tell JUnit to run every single test that you have written so far. Serious professional projects have thousands or millions of tests: no human being can do all of that without dying of boredom. That is why automated testing is important.

Testing everything frequently ensures that the code does not go backwards, e.g. that introducing new features does not break other features that were already working. This why it is sometimes called *regression testing*: it makes sure that the code does not regress, i.e. it does not go backwards.

It is common for most professional projects to run *all tests* on a daily basis, usually overnight when the programmers involved are sleeping[3]. This is done by means of an automatic script that retrieves the sources from a public repository (e.g. on GitHub), compiles everything, and then runs all the regression tests. If anything is wrong, the programmers can fix it as soon as they are back to their computers. Usually programmers run a subset of all tests before pushing their changes, but a serious project can have millions of tests that take several hours to run; therefore, a full comprehensive test run is necessary every day to keep projects on track.

[3] Well, most of them at least.