Object Oriented Software Design - I

Unit Testing

Giuseppe Lipari
`http://retis.sssup.it/~lipari`

Scuola Superiore Sant'Anna – Pisa

November 28, 2011

Testing

- Testing can never be exhaustive
  - you can never discover all the program bugs by running tests, simply because (except in trivial cases), the number of tests to be executed is badly exponential
  - nevertheless, testing is useful because you can catch **some** bugs, the most evident and probably the most important bugs in a program
  - also, tests give some evidence that the program is conforming with the specification, and this is useful for the customer
  - therefore, even if testing does not prove correctness, it proves something, and something is better than nothing!
  - Every programmer acknowledges that testing is an important activity in coding
- There are many types of testing:
  - Unit testing, functional testing, integration testing, etc.
- Here we discuss Unit Tests, a tool for the programmer

Unit testing

- Unit testing is a method by which individual units of source code are tested to determine if they are fit for use
- A unit is the smallest testable part of an application, typically a class or a function
- Unit tests are written by programmers as part of their coding work
  - The idea is to test an unit (a small piece of code) in isolation
- The goal is to:
  1. Find bugs in code
  2. Be able to refactor (i.e. change the code) later, and check that the unit continues to work
  3. Check that modifications to other units (on which the unit under test depends upon) will not break the assumption and invalidate the behaviour of the unit
- item 2 and 3 are also called *regression testing*, and are fundamental to check properties (as the Liskov Substitution Principle, for example)

The problem with test

- Writing tests is a boring activity
  - It works against human nature
  - it is a lot of additional code to write (set up the environment and all the necessary objects, write the testing code, run the test, check the result, etc.)
  - the test code needs to be maintained and kept constantly in sync with the tested code
  - most of this work is not very funny, whereas a programmer wants to write useful application code (a more inventive and rewarding activity)

Testing is boring

- As a consequence, most programmer simply do not test
  - Testing is time consuming
  - There is never enough time, so the programmer concentrates on *important* things, like producing application code
  - Tests are code, and we may introduce bugs in testing code, producing “false positives”
  - and many other excuses
- Forcing programmers to write tests is not the right approach
  - It is against human nature, so after a while, he will stop anyway
  - ... unless ...

Making testing more attractive

- The first rule is to make the make the testing activity automatic
- The programmer should not spend time in *checking* the output of the test to see if everything is OK
  - Let the PC automatically check the outcome
  - the only thing the programmer wants to see is: **OK**, or **FAIL**, and not endless screens of text output to be checked
- Then, make it easy to run suites of tests
  - All tests should be grouped in test suites, and it should be possible to compile and execute them with one single command
  - as soon as a new test is written, it is added to the test suite and executed along with the other tests

Run tests often

- Tests should be written by the programmer *before* coding the unit
  - In other words, first write the test, then write the functionality
  - Run the test after compiling
- “Run tests often” means “Run tests every time you compile”
  - The habit of running tests should be automatically embedded in the development tool
  - in addition to “I should make my program compile”, the objective is now also “I should make all the tests run smoothly”

Advantages

- Substitute “debugging” with “testing”
  - Testing is done while coding, while debugging is done later,
  - Testing **is** a special kind of coding, while debugging means slowly going through existing code in a painful way and under pressure
  - Tests are there to stay, and can be run automatically at any time, while debugging does not stay, it is wasted time
- Certainly, you will be convinced that testing may be more funny than debugging
- Maybe you will also become convinced that testing may be very productive
  - If you can drastically reduce the amount of time spent in debugging, more that the time you spend in testing, then you productivity increases

Testing to addressing change

- Testing is a fundamental tool to address the “need for change”
- If you need to change existing code (or add additional code/functionality), you want to make sure that existing tests do not break
- if they do break,
  - Maybe you introduced a bug in the code that breaks existing code
  - Maybe you violated an assumption made by existing code (see LSP)
  - Maybe you need to change the test because the specification has changed and the test is not valid any more
- In any case, since you test every time you compile, you can immediately spot the error
  - As opposed to discovering the bug during integration testing, when you will need to trace back the problem and debug you code to find where the problem is
- which alternative is more time-consuming?

Testing coverage

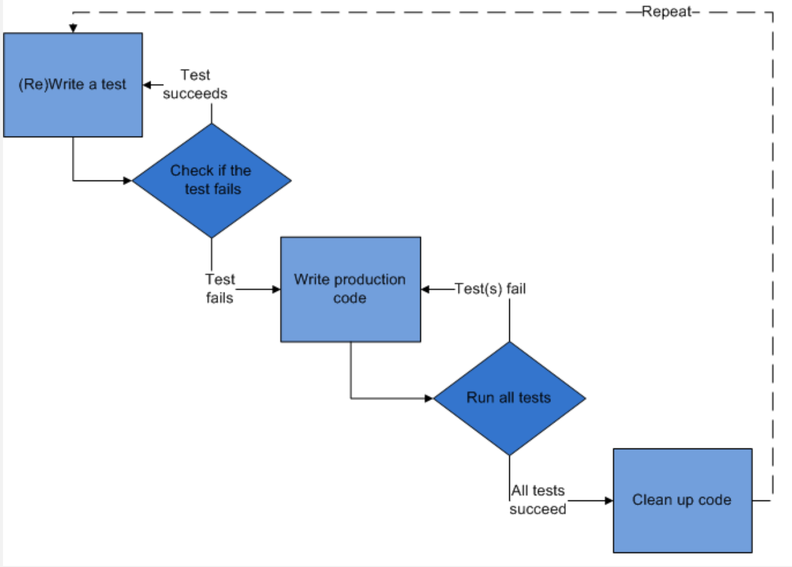
- Of course, you can never do a complete testing coverage in the unit test, because it is too time consuming
- However, you can try to write **a few** meaningful tests
- for example, checking boundary conditions (the source of most errors)
  - what if the function needs a file, and the file does not exist?
  - what if I try to extract from an empty container?
  - what if I pass a negative value (while the function expects a positive one?)
  - etc.
  - Also, check a couple of “normal” cases
- Of course, testing can never completely substitute debugging
  - Since you cannot be exhaustive, it may happen that your tests do not spot a subtle bug
  - Don’t worry: when you discover it by debugging, add immediately a unit test to checks that the error will never occur again in future modifications

Other advantages

- When writing tests, you concentrate on the interface
  - while writing tests, you wear the “client hat”
  - while writing code, you wear the “implementer hat”
  - writing tests is similar to writing a specification for the code
- The test is useful as “documentation”
  - If you want to know how to use a function, you can sometimes look at the test
- You have a clear point at which you are done: when the all tests work!

Test Driven Development

- The practice of writing tests first is called *Test Driven Development* (TDD), and it is one of the main points of the Extreme Programming (XP) methodology proposed by Kent Beck



Continuous Integration

- Another related practice is Continuous Integration
  - This consists in automatic compilation, testing and commit in the repository in one development cycle
  - committing on the server automatically compiles and test the code
  - therefore, an integration step is performed at every commit!
- Of course, an appropriate tool support is needed to automate all steps
- Also, a certain rigorous and structured approach is needed to impose the practice to all programmers

Tests, test suites and fixtures

- Testing should be automatic:
  - No need to manually check the results of the test, it should only say **OK** or **FAIL**
- Tests are grouped into **Suites**
  - A test suite is just a set of tests on the same “portion” of code
  - for example, a set of tests on the same class, or on the same package
- What if you have two or more tests that operate on the same or similar sets of objects?
  - Tests need to run against the background of a known set of objects, called a **test fixture**.

JUnit

- JUnit is a library written by Kent Beck
- Its goal is to make the job of writing tests and running them as easier as it is possible
- It uses **annotations** to make the writing easier
  - Java annotations are lines of code beginning with a @
  - An example is the **@Override** annotation
    - *It indicates that a method declaration is intended to override a method declaration in a superclass. If a method is annotated with this annotation type but does not override a superclass method, compilers are required to generate an error message.*
  - Annotations are processed by the compiler, and usually do not produce extra code
    - However, they can be processed by other tools to produce extra code or data
  - It is possible to define new annotations
  - See the Java documentation

Structure of the test case

- A Test case is just a class with a very simple structure:

```
import junit.framework.TestCase;
import org.junit.*;

public class TestFoobar extends TestCase {
    @BeforeClass
    public static void setUpClass() throws Exception {
        // Code executed before the first test method
    }
    @AfterClass
    public static void tearDownClass() throws Exception {
        // Code executed after the last test method
    }
    @Before
    public void setUp() throws Exception {
        // Code executed before each test
    }
    @After
    public void tearDown() throws Exception {
        // Code executed after each test
    }
    @Test
    public void myTest() {
        // test code
    }
    @Test
    public void anotherTest() {
        // test code
    }
}
```

Assertion

- When writing a test, it is important to check that everything goes well
  - This can be achieved through *assertions*
- Example:

```
@Test
public void testAddLeft() {
    tree.addLeft(leftTree);
    assertTrue(tree.getLeftSubtree().getRootElem() == "L");
}
```

- The function `assertTrue` checks that the condition evaluates to True
  - If it evaluates to False, it throws an exception
- There are many other assertions that can be used