Testing of Java Packages

This Project aims at rewriting java libraries by using a designing method: “Clean Room”. For this purpose, three groups have been formed, one for each package to be rewritten: Crypto, Math, RMI, and the Testing group.

As part of an industrial process, the testing phase adds value to the product that is being handled: every programmer makes mistakes, leaving flaws in the programs that are only discovered by making the program’s execution fail. The testing phase consists of discovering those mistakes by identifying the execution faults and, thus, detecting flaws in the program. The removal of these flaws adds value to the product. The specific aim of the tests is to find as much mistakes as possible.

Eclipse 3.1 and a CVS server have been established as the working tools, plus one repository for each group. A small alteration on the extreme programming methodology was made and a framework for testing JUnit has been used for Unit tests. Each group was in charge of the private classes, while the public interfaces are tested by the Testing group.

The testing group initiated its activity estimating the time availability, choosing the tools and dividing work based on the public classes of the API.

Given the infrastructure of this project, there is one Pentium IV PC available per member, each PC having two operating systems installed. A good balance has been reached between the operating systems, since two of the members work with Windows XP and two others work with Mandriva Linux.

Policies to be taken into account in Testing

- To test is to use a program in order to find flaws in it. A program should never be used to show that it works, that is not the objective.
- The testing phase should not be made after the creation of a program, but parallelly or even previously in some cases, first the testing and then the programming.
- A test has been successful if it finds a flaw. It is key and very normal to find flaws in the programs, not to find a perfect execution of the program.
- In order not to be “comprehensive with the flaws” the tests must be designed and passed by someone different form the one that created the code.

- It is not necessary to wait until the whole code has been written to start passing the tests. They must be passed as the code is being written so that the mistakes are discovered as soon as possible and thus avoid that they spread to other units. Actually, the name “testing phase” is not so accurate, because there are plenty of activities that are developed during this “phase”, and there is no need to finish one phase in order to start another.
- The tests can find flaws, but never prove that there are no flaws.
- The test should be as much precise as possible, i.e., every time it is realized it must be able to be executed and be repeated the same way. Special attention must be paid to the parameter generator.
- Unit tests must be grouped according to their common initialization.
- Unit tests fail when the method that is being tested gives an unexpected result or an unexpected exception.
- Unit tests do not end correctly when the method that is being tested is not responsible for the flaw.
- Unit tests end correctly when the method that is being tested responds as it was expected.
- If the initialization is optimized with static variables that are initiated only once, it must be made certain that they are not altered in the unit tests.

Organization of the Testing Phases

- UNIT
    - BLACK BOX
    - WHITE BOX
- INTEGRATION
- PERFORMANCE
- CODIFICATION STYLE
    - Identifiers denomination
    - Code documentation

Units test

When the unit seems presentable (or there is one, as in our case, we have Sun's version of the JDK), we enter in what is called a systematic testing phase. On this stage faults

are searched following some criteria so that nothing remains untested. The most common criteria are the Black Box and the White Box.

We refer to a black box test when it lacks the code details and it limits itself to what it observes form the outside. It aims at discovering cases and situations in which the unit does not do what is expected.

Contrary to what the black box test does, the white box test analyzes closely the code that has been written and it tries to make it fail. Perhaps the most appropriate way to call these tests is “transparent box tests”.

The Junit Framework has been adopted to perform these tests. For these reason we will create a class which inherits form TestCase for each class to test. They will follow the nomenclature, starting with the prefix “Test”, followed by the class name to be tested (for instance, TestCipher). Besides, the testing methods must start with “test”, followed by the name of the method that is being tested, the types of arguments that each method receives and finally a number. They will be invoked by the runner of JUnit that we use. A number of conditions about its return will be verified, as well as any exit that said method may generate. This is done through the use of asserts. (the assert methods are inherited by the class TestCase from the class Assert, and they are in charge of verifying if certain condition is fulfilled. In case we need other assert methods that are not implemented for the evaluation of specific conditions, we will implement them in our utility class)

In order to simplify the manual running of the test batteries, TestSuite is used. This will allow us to realize at once, all the test methods of one of these classes, which will be done as follows:

TestSuite testSuite = new TestSuite(TestCipher.class)

However, it should be noticed that when we call the class builder this way, we are including all the testing methods (those that take the prefix “test”) of our test class in the test batteries as follows:

Junittextui.testRunner.run (testSuite)

Furthermore we can realize our test classes as follows:

TestResult testResult = new TestResult( )
testSuite( ).run (testResult)

Black Box

White box tests are also known as: opaque box tests, functional tests, entrance/exit tests, data induced tests.

Black box tests center on what is expected of a module, i.e., they try to find cases in which the module does not comply with its specification, that is why they are called functional tests. The tester only supplies the data as entrance and studies the exit, without being concerned about what the module might be doing on the inside.

Black box tests are based on the specification of the module's requirements. In fact, the phrase "specification coverage" is used to give a measure of the number of requirements that have been tested. It is easy to obtain 100% of coverage on internal modules, but it can be harder with modules that have an external interface. In any case, it is highly recommended to get high coverage on this line.

The problem with black box tests does not usually lie on the number of functions provided by the module (which is always a very limited number, on reasonable designs) but the data that are passed to these functions. The group of possible data can be wide. (For instance, it can be an integer).

Several scripts have been realized, initially on Perl, to generate massive test codes. This was quite beneficial, since a large number of tests were realized in a very short amount of time and with very little difficulty. After that, said scripts were realized directly in Java, which allowed us to generate the code with the expected results.

Since the projects are developed with the designing method "Clean Room", we have at our disposal a binary of a different implementation to calculate the expected results on the Unit cases.

This working methodology permits us to write unit tests quickly. As a consequence, the ranges searched were not a priority to us. The test cases were those that are "frontier" or just interesting.

To achieve a good coverage with black box tests is a desirable aim, but certainly not enough. A program might pass widely thousands of tests and have nevertheless internal flaws that will arise on the most unfortunate moment.

Black box tests show us that a program will do what it has to do, but not that it will do (in addition) other less acceptable things.

White box

White box tests are also known as: structural tests, transparent box tests.

With these tests we are always observing the code, which the tests execute in order to "try
it all". This idea of total test formalizes in what is called "coverage" and it is nothing but a percentage measure of the amount of code we have covered.

To achieve a good coverage with white box tests is a desirable aim, but certainly not enough. A program might be perfect and nevertheless not be good for the function that is being required.

Once the development of every class was finished, the code was analyzed and the necessary tests were generated complementing the black box tests.
White box tests show us that a program does what it does but not that it will do what we need.

Integration Tests

Integration tests are generally carried out during the construction of the system, they involve an increasing number of modules and they end up testing the system as a whole. However, we will treat these tests as acceptance tests, since the package will be tested from the user's point of view, generating in this way some application that uses as much functions as possible.

Performance test

We may typically be concerned about the response time or how much time it takes for a system to process so much information, or how much memory it consumes, or how much disk space it uses, or how many data it transfers by a communications channel, or…. In this context, it is usually useful to know how they evolve when the dimension of the problem is modified (for instance, when the volume of the entrance data is doubled).
However, here we will concentrate on the execution time of every method, in order to compare our implementation with that of Sun's.

Codification styles

In relation to any human activity it can be stated that there is style when: the level of similarity of certain attitudes, expositions and actions of a person, or a group of people,

is such that it is possible to identify this similarity and deduce general behavioral schemes.

- Every programmer ends up developing their own codification style.
- The homogenization of the different styles of the programmers of the group facilitates the programming work.
- In our case, we control that the codification specifications of SUN's are fulfilled.