“javax.crypto” package

We started with the Crypto package. Crypto is a framework where a provider scheme that contains the encryption algorithms and generation of parameters and keys.

Given the diversity of functions, because of the selection of algorithms and the supplier of nearly any usable class of the Crypto package, dynamic and static tests were divided.

A static test is always the same test, while a dynamic test might run with other algorithms. Subsequently, these dynamic tests transformed into an integration test that consisted of testing certain particular algorithms. This will be explained below.

Eventually, we came to the conclusion that the tests should not be of brute force, but that they should be as much independent as possible from other classes. That is how the implementation of Key interface came up as a valid solution to obtain symmetric keys. However, public key algorithms were not as lucky. We later found out that “Secretkeyspec” was already implementing key interface in a similar way.

The tests were grouped in cases that extend from TestCase, each of which tests completely the other class. This methodology would very soon be in crisis because of the rapid growth of the code.

This problem was produced by the massive code generator, because the amount of generated code was such that it made impossible to work with Eclipse’s editor. This obliged us to make a division on the classes. This problem was made even worse because of the decision of using especially chosen individual tests and not test bags. Cipher is one of the classes that has been divided the most, since there are TestCases devoted exclusively to test one method, for instance; “update”, “wrap” and “unwrap”.

Another problem we were faced with in this methodology was the initialization of tests (setup). Since one test class tests a whole class and the activities to test were so diverse, in most cases the test was initiated in the test itself. This leads itself to some errors such as the generation of the key every time the test is executed, provoking an unnecessary indefiniteness. Another problem was found when the tests ended with a flaw and it was in the initialization, making difficult later performance tests.

In order to distinguish these cases, it has been decided to use the three termination forms in a clear and precise way:

- Correct termination when the test ended correctly
- Failing termination when we found in the test an unexpected result or an ill-executed exception
- Error termination, when the test failed in the initialization or any part that was not related to the function to be tested

Initializations have been repeated several times and a lot of these times they were not required. As an optimization way, in some cases static variables have been used, they are initialized only once and said variable is not altered.

Even though several flaws have been discovered in the development of the tests, the constant growth of the test code prevented us from correcting the code already written, unless a critical fault that makes the test unusable was found on it.

The tests were executed on Linux and Windows operating systems, and for Sun's implementation as well as for our own. The results were kept on XML files and were stored on the server Saturno in the folder named “Intel / Testing/Ejecución de pruebas”.

Inside said folder there are other folders with the name of each package. In this case we search for Crypto and then one folder to separate Linux executions from Windows executions. An XLS design sheet was designed to achieve a better reading of the XML file.

Unit tests

Black box

The specifications of the classes and their methods can be found in the API of SUN's JDK. Since the API does not mention all the possibilities, in case of doubt about the operation we assume that SUN´s implementation puts the contract into effect adequately.

If a group disagrees with SUN´s implementation, the group will have to provide an explanation for this disagreement, as we will see in the case of Crypto. We also detected bugs on Sun's implementation, leaving design observations to the groups specialized in that package.

Certificate tests that were not incorporated into the general tests were realized. 15540 tests of certificates resulted from 15 basic initialization tests combined with all kinds of certificates.

Only one Bug was found in Sun, and even this was already corrected. The class javax.crypto.DESKeySpec has a method called “isWeak”. An array of Byte is passed to this method and it returns true if the array is material for a weak key or a semi-weak key.

The arrays "1F1F 1F1F 0E0E 0E0E" and "E0E0 E0E0 F1F1 F1F1" are material for a weak key and the method “isWeak”, when applied to any of these arrays, returns false and it is thus material for a strong key.

These bugs are detected on the following failing tests:

- stestIsWeak003(ar.org.fitc.test.crypto.spec.TestDESKeySpec)
  ```
  junit.framework.AssertionFailedError: Should not raise an
    Exception...junit.framework.AssertionFailedError: Must be a week key
  at ar.org.fitc.test.crypto.spec.TestDESKeySpec.testIsWeak003
    (TestDESKeySpec.java:211)
  at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
  at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
  at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
  at java.lang.reflect.Method.invoke(Unknown Source)
  ```

- testIsWeak004(ar.org.fitc.test.crypto.spec.TestDESKeySpec)

```
junit.framework.AssertionFailedError: Should not raise an
  Exception...junit.framework.AssertionFailedError: Must be a week key
at                    ar.org.fitc.test.crypto.spec.TestDESKeySpec.testIsWeak004
  (TestDESKeySpec.java:220)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
at java.lang.reflect.Method.invoke(Unknown Source)
```

White Box

In order to initiate the white box test two conditions need to be fulfilled. The first one is that all the unit tests that constitute the black box must be prepared. The second one is that Crypto must have the code ready and must enter into a test and revision phase.

While Testing was generating the group of white box test, Crypto executed the unit tests and started correcting the code problems. This simultaneous activity is part of this open code process, with several changes on it.

As a policy we chose not to allow changes in the code on the part of Testing. However, some propositions were made and they were quickly approved. The report of every flaw was communicated orally, modified by Crypto immediately, and then the changes merged in the CVS.

As we were concerned both with function and presentation, the control flows were visually checked, and if in doubt about the operation, a unit test was realized. Our aim was branch coverage of one hundred per cent, facilitating this process initiating with the coverage obtained from the black box tests.

In some cases with only one unit test a general problem arises, as it was the case of the initialization of Cipher. In this case, the private variables were set before initiating the CipherSpi, being the second initialization of possible exceptional termination. The order was reversed, now we initiate first the CipherSpi and then the private variables. Although only Cipher has been referred to, this case was repeated in several initializations which were corrected without the need of unit tests.

The unit tests were executed with a tool used to calculate the coverage. The results of the coverage are found on a folder named “cobertura” inside the “crypto” folder, in a html form for simple reading. The classes that are covered %100 do not need further correction, while the classes that are not completely covered have marks on the items that are not covered. Following we will enumerate and briefly explain the items without coverage:

- All the *Exceptions* have two constructor methods at their disposal, one with a string-shaped message and another one without parameters. Because of this the exceptions only have 50% of coverage, since the exceptions are thrown using the constructor that receives a string.
- The class *CipherInputStream* reached 95.5% of coverage. In its testing process the protected constructor was not used, a flow Cipher and to initiate it with an InputStream that returns zero bytes to the array reading.
- The class *CipherOutputStream* reached 84.4% of coverage. In its checking process we did not use the protected constructor and close it several times.

- The class *CipherSpi* reached 70.4% of coverage. Tests were not realized for this class. In its checking process we did not use the methods engineGetKeySize, engineUnwrap and engineWrap. All of them end exceptionally.
- The class *NullCipherreached* reached 74.5% of coverage. In its testing process we did not use more intensively the method getKeySpec.
- The classes *ExemptionMechanism*, *ExemptionMechanismException*, *ExemptionMechanismSpi* were not tested at all because they were not implemented; thus, they have no coverage (%0)
- The class *Mac* reached 94.8% of coverage. In its checking process we did not use the call with the provider String type that is null and a call to doFinal without initialization. Both cases end exceptionally.
- The class *NullCipher* reached 100% of coverage, but its subclass NullCipherSpi reached 20% of coverage. Given its simple implementation, we did not test methods that did not do anything or that returned the same value.
- The class *SealedObject* reached 65.9% of coverage. In its checking process we did not use its protected constructor and a Cipher with AlgorithmParameters. On the other hand, the existence of a dead code that throws an AssertionError was detected on the code.
- The class *Cipher* reached 91.2% of coverage. In its checking process we did not test all the possible initialization techniques, to request the ExemptionMechanism and to use a certificate with no critical extension of the OID in the initialization. Plus, tests with non signed provider and files of the lacking supplier were not performed.
- The class *DESKeySpec* reached 85.2% of coverage. In its checking process we failed to test the functions isWeak and isParityAdjusted with a null key and wrong length material.
- The classes *RC2ParameterSpec*, *RC5ParameterSpec* and *SecretKeySpec* reached more than 94% of coverage. In none of these clauses was it proved that an instance of any of them equals itself (using the equals method).

Performance test

Performance tests require comparisons between XML; there is even a HTML to XML table translator. The comparison is based on the relative difference of the results, i.e. there is a difference if it is a %10, %20 or a %30 differences, and any percentage is a valid unit. Once the difference has been discovered we observe which is faster, SUN's or ITC's.

Performance tests were executed over the general group of correction tests. The results were organized by classes and only time was measured. These tests initially showed differences of 10 to 1, that is, SUN executed 10 times faster than ITC.

That is why the organization was reduced to unit tests, and after some tests we discovered the cause of such differences. Only the classes that used Spi generated a delay. This delay was due to the signature control of the provider, and this control was realized every time that service was required from the provider. This difference was removed with a quick modification of the code. Our Crypto package is now faster especially because it does not have exportation limits (unrestricted exportation).

The difference that generates having unrestricted exportation can be observed in methods such as *javax.crypto.spec.PBEKeySpec.(char[], byte[], int),* where the implementation of

SUN delays 13,196,104 while our implementation delays 0.07148704. In this case we say that our implementation is %100 more efficient, since there is no circumstance analysis.

For performance tests, the profiler of Netbeans was used. At least one test case per method was executed. If the method had any exception, it was executed as well, together with the integration test.

Clean tables will be found in an excel file on folder “crypto” and on folder “Windows” or “Linux” according to the operating system wanted.

The file is named “cryptoperformance” and it disposes of one spreadsheet for the jce times of SUN's, another for the times of our jce and a third one with the time differences.

The negative values mean that the difference is in favor of SUN and the positive values mean that the difference is in favor of our project.

While the performance was estimated in the two operating systems, in windows it seemed more unstable, with random jumps, and in Linux it seemed more symmetric. In Windows our jce varies from the best to the worst performance with differences of over %50. In some cases reached 5000 milliseconds, whereas in Linux the differences only reached a few milliseconds.

We can finally say that, even some of the majority of the methods is even as regards its execution time; there are many methods that are widely faster than our jce.

Integration Tests

We first made an exchange of keys and a symmetric key passage. Then we started to encrypt the channel with said key. After that we made a client and a chat server, which was later on modified in order to be able to execute all the tests in an automatic and centralized way. We achieved this through the utilization of the GNU SuiteRunner tool.

This tool offered us the possibility storing the records of the executed tests in an XML file. Nevertheless, it is also possible to execute these tests with the JUnit Plugin inside Eclipse.

A dynamic test for cipher was realized. It consisted in testing all the algorithms and providers that were already set up, with a specific kind of tests. This was repeated in every algorithm and in every provider. We then found out that when we performed the dynamic tests, much of them failed because of the lack of AlgorithmParameters for the algorithm that was being tested; others failed because of the mode of said algorithm. As a consequence, very few of the tests were actually useful.

Although this may seem as an innovative integration test, it soon failed because the different types of ciphers were not distinguished, which caused a great number of failed tests that did not test what should have tested. However, when we combined these tests with the providers we reached really tempting numbers (around 70000) and just using the BouncyCastle provider.

As a result of this inconvenient, we developed a new type of test, in which the framework JUnit was not used. This test tried with certain particular algorithms -DES, 3DES, RSA, RSA/ECB/PKCS1 and Blowfish-. SealedObject, CipherInputStream and

CipherOutputStream were also included in the tests. This test works for the providers that are set up and is takes a time measure that takes some time to perform each algorithm.