Unit Tests of Software in a University Environment

Darlene Gómez, Dalila Jústiz, and Martha Delgado

Instituto Superior Politécnico José Antonio Echeverría (CUJAE), La Habana, Cuba

{dgomez, djustiz, marta}@ceis.cujae.edu.cu

Abstract. Quality is a necessary feature to be achieved by a system or application after its development is completed. Tests contribute to software quality, but testing is a process that requires much time. This process starts at the beginning of the construction of a system and ends before the implementation. This paper presents an analysis of a set of tools for automatic test execution, with emphasis on unit testing, and describes a proposal of using such tools in a university environment of project development. This proposal responds to the need of combining commercial tools with other path generation tools and test cases.

Keywords. Software quality, test cases, test tools, software test.

Pruebas unitarias en proyectos de software en el entorno universitario

Resumen. La calidad es una característica necesaria que debe ser alcanzada por el sistema o aplicación una vez finalizado su desarrollo. Las pruebas contribuyen a la calidad del software, aunque es un proceso que requiere de un alto porcentaje de tiempo. Estas deben comenzar desde que el desarrollador inicia la construcción del sistema y deben finalizar antes del despliegue del mismo. Este trabajo se centra en las pruebas que se hacen a los pequeños componentes que conforman el sistema. En él se presenta un análisis de un grupo de herramientas de ejecución automática de pruebas, haciendo énfasis en las pruebas unitarias, y esboza una propuesta de utilización de estas en un entorno de desarrollo de proyectos en el marco universitario. En esta propuesta se sustenta la necesidad de combinar estas herramientas comerciales con otras de generación de caminos y casos de prueba.

Palabras Claves. Calidad de software, pruebas de software, diseño de pruebas, herramientas de pruebas.

1 Introduction

The testing phase is important for the software development process to meet the requirements set by the users and the clients. But one does not have to postpone testing till this stage. When a developer starts developing software, verification and debug of the code must be performed, but these processes are usually ignored by the development team.

A successful test does not mean that there are no errors, but rather that no other errors were detected by this particular test [21].

A university environment is characterized by the presence of students and professors in a software development group where they perform research and accomplish production tasks. The proposal in [22] is to perform a development process in such environment. This proposal includes definition of methodological aspects and selection of tools to computerize the process. But it does not consider aspects related to the execution of unit tests.

Therefore it is necessary to have techniques for designing test cases and tools to support unit tests.

There are many authors who deal with the issue of software testing [7, 13, 18, 23, 24, 24] but it is generally agreed that software testing is merely the process of executing a system or a component with the purpose to measure and improve quality, under specified conditions, and also with the intention of finding errors, observing and recording the results, and evaluating some aspects of the system or component.

According to [23, 24, 26], two basic approaches or methods of testing are white box testing, or structural approach, and black box testing, or functional approach.

The black box approach tends to discover functional errors occurring in the implementation of requirements or design specifications. They are focused on input and output functions. They also check the correct handling of external functions provided or supported by the software and the observed behavior. Transformations which occur may not be seen, only the input and output functions are known [23, 24].

The white box or structural approach discovers errors which occur in the coding of a program. They are focused on the internal structure of the program (analyze execution paths). They also verify the correct implementation of the internal units, structures and their relations. They emphasize internal error reduction [23, 24, 26].

White box tests are also known as unit tests focused on the internal processing logic and data structures within a component. This type of test can be applied at the same time to multiple components.

This article aims at contributing to test process improvement and outlines the idea of generating test cases from a source code linking it to test execution tools. We also intend to show developers the importance of test path generation from their source code using as a basis the techniques for designing test cases.

2 Unit Test

According to [23], unit tests are focused on each individual component, ensuring that it works properly as a unit, thus verifying the smallest unit of software design. In [29], it is stated that generally in object-oriented tests it is assumed that a test unit is a class. Thus it checks that the state of an instance of a class is correct for input data. Unit tests are designed to verify the functionality and structure of each component individually once it has been coded [18].

White box testing should be able to run at least once all independent paths from each module, and to use the true and the negative part of a decision which is no more than running unlimited cycles and using all the internal data structures [4].

White box testing is a design method that uses the control structure described as part of the design at the component level to derive test cases. Therefore, the software engineer can derive test cases which [23].

1. Ensure that all independent paths within the module have been exercised at least once;
2. Exercise the true and false options of all logical decisions;
3. Execute all loops at their boundaries and within their operational limits;
4. Exercise internal data structures to ensure their validity.

There are different techniques of designing white box tests [1, 2, 17, 23, 31] including condition testing, data flow testing, loop testing, basic road test, and coverage testing of decision/condition. In this work, the last two tests are emphasized.

A basic path test has the aim of finding a logical complexity measure of procedural design and uses this measure to guide the definition of a basic set of execution paths. The obtained test cases guarantee that during the test each program statement is executed at least once (statement coverage). For designing tests using the basic path principle, one must follow the steps: (1) get the flow graph from the design or code module; (2) get the cyclomatic complexity of the flow graph; (3) define the basic set of independent paths; (4) determine the test cases that allow the execution of each of the components mentioned previously; (5) run each test case and verify that the results are as expected [23].

In a flow graph, each node represents one or more procedural statements. A single node may correspond to a sequence of steps of a process and a decision. The arrows (edges) represent the flow of control. A node predicate contains a condition and is characterized because two or more edges start in it. The regions are the areas that limit edges and nodes, and they include the areas located outside the graph [23].

The cyclomatic complexity is a measure that gives an idea of the logic complexity of a program; it is used to determine the number of paths to search. The following aspects must be considered: (1) if there is coincidence with the number of regions of the flow graph; (2) the cyclomatic complexity, $V(G)$ of a flow graph G, is defined as $V(G) = Edges - Nodes + 2$; (3) the cyclomatic complexity, V (G) of a flow graph G is also defined as $V(G)$ = *Predicate Nodes* + 1 [23].

An independent path is any path that introduces to a program at least one new set of processing instructions or a new condition, which from the point of view of the flow chart must travel along at least one edge which has not been run before [23].

A test of coverage decision/condition is described as follows. The coverage is the amount of code covered by a set of test cases. There are many ways to measure how much code has been covered. Here are some of them. A condition is a pair of algebraic expressions connected by a relational operator (<,>, =,> =, <=, <>). A decision is a list of conditions connected by logical operators (AND, OR). The coverage decision criterion is satisfied when each decision and each condition are evaluated true or false at least once. This guarantees the decision coverage, i.e., that the decision of which part of conditions is to be made true or false is accomplished at least once [29].

3 Tools for Automatic Execution of Unit Tests

A testing process should be supported by tools that help to design and execute test cases. In order to characterize the environment, 15 projects were interviewed, of which approximately 67% use Visual Studio 2010 as IDE in C#, while the 13% use Eclipse for JAVA. Making the use of these technologies as a starting point, a search for tools which support unit testing was done. The features are described as follows.

JUnit: a set of libraries of the xUnit family developed by Erich Gamma and Kent Beck, it is free software, open source component. It supports testing Java applications. It integrates Eclipse development environment, NetBeans and JDeveloper and performs unit testing. There are two versions of JUnit families: 3.x and 4.x. The 4.x versions make use of new features of Java. They automatically generate test cases [14].

TestNG: a framework for tests that works with Java. It is based on JUnit (for Java) and NUnit (for. NET), but introduces new features which make them more powerful and easy to use. It is open source and integrates with three major Java IDEs: Eclipse, IntelliJ IDEA, and NetBeans. As supporting evidence, it incorporates Hudson as a continuous integration server and Maven as a build system. It performs different categories of tests such as unit, functional, end to end and integration tests. It does not generate test cases [15].

Jtest: a software quality testing platform that allows development teams to increase productivity and quality. It is not a free tool; it focuses on practices for validating Java code. Jtest is a customized version of Eclipse IDE and its applications. It perfectly integrates with ParasoftSOAtest, IntelliJ, IDEA, and RAD, as well as with CVS, ClearCase, Subversion, and StarTeam. It develops unit tests and functional tests. It is able to automatically generate all necessary unit tests, taking into account the parameters of code coverage and trying to find evidence that result in runtime errors [16].

NUnit: it belongs to the family called xUnit testing tools. It is free software, open source, and is integrated with the development environment. It is a unit testing framework written in C# for all languages .NET. It supports the basic languages such as NET and C#, J#, VB and C++. It performs unit testing. Developers can easily complete NUnit tests; this tool also offers a graphical interface to view the results of a test. NUnit compares expected values and values generated, if these are different, the test does not pass, otherwise the test is successful [20].

Visual Studio Unit Testing Framework (MSTest): Microsoft Visual 2010 has a testing framework known as MSTest. This is not free software; it has a complete set of functions for the trial test of Visual Studio Team System that runs in the IDE. It incorporates the code coverage analysis once the tests have run as well as the code generation testing methods. This tool allows unit testing, load tests, and fitness tests. One of the key features of the test team for Visual Studio is the ability to load test data from a database and then use this data in the test methods [19].

Unitils: an open source library in Java, integrated with Eclipse, Hibernate, and Spring. Unitils is used to implement the business and persistence layers and access to data. It is also used with JUnit and TestNG. It manipulates different libraries like DBUnit for testing a database and EasyMock for testing the integration between objects aimed at achieving unit tests, integration tests and transactional tests. It does not generate test cases [29].

Taking into account the characteristics of the environment and the interview with project managers, the following comparison criteria for tool selecting were defined: (1) it is free software, (2) it is open source, (3) it is integrated with Eclipse, (4) it is integrated with Visual Studio, and (5) it generates test cases.

The determining factors in this selection were the automatic generation of test cases and the integration with the Eclipse IDE and Visual Studio which are most commonly used in development projects. Therefore, the selected tools were JUnit and MSTest Visual Studio.

Although MST is better than NUnit for being a free software and open source, it was not selected because most of the projects use Visual Studio as IDE and they are accustomed to this environment. Although Unitils and TestNG meet the same criteria, JUnit was selected due to its high level of usability.

4 Tools for Generating Test Cases

There is a set of tools that automatically generate test cases based on different parameters [8,10, 11, 12, 26, 27].
A university environment is made up of students, teachers and specialists engaged in software development, but the teaching practice consumes much of their time. To speed up the testing process, there is a need for tools that generate test cases automatically.

CP generation is a key element to consider during the development of a software product, because it reduces the time the test team dedicates to this activity. The paper [8] documents the development of a component for generating automatic test paths, starting with the detailed descriptions of functional requirements guided by patterns proposed in the component.

The idea outlined in the work is to combine elements used in the solution of [8] with the existing tools designed to run unit tests. It is intended to complement the tools to perform unit tests with the case generation component from the solution code. It allows developers to save time when designing test cases.

Table 1. Comparison of tools

Tools	Approaches				
	1	2	3	4	5
JUnit	x	x	x		
Jtest			x		x
Unitils	x	x	x		
TestNG	x	x	x		
NUnit	x	x		x	x
Visual Studio (MSTest)				x	x

5 Experiment

This experiment displays the feasibility of generating test paths from the source code and achieving an automatic analysis of the basic path and decision/condition coverage techniques exemplified below manually.

5.1 Unit Testing in Java code and Using JUnit

To show how JUnit tools are used, a test applied to the "Withdraw" method has been developed, which belongs to a small project of bank money transactions.

This method receives as a parameter a value which is the amount of money to be withdrawn. If this amount of money is a negative value, it throws an exception, and if the balance is less than the amount to be withdrawn, it displays another exception that the balance is insufficient; otherwise it creates a new movement of transactions, passes the parameters it has, and adds them to a list.

```
public void withdraw (double  x) throws Exception
{
    if (x <= 0)
      throw new Exception (“Not possible to withdraw
        a negative quantity”);
    if (getBalance() < x)
      throw new Exception (“Balance insufficient”);
    Movement m = new Movement();
    m.setConcept (“Withdrawn in cash”);
    m.setImport (-x);
    movements.add (m);
}
```

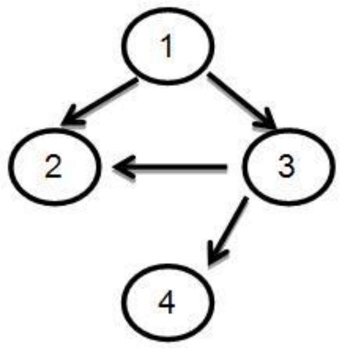
Fig. 1. "Withdraw" method

To set the number of test cases needed to be designed for conducting a test as completely as possible; the design technique applied is basic path and decision/condition coverage. At the initial step, the flow graph of the method is generated. At the second step, the cyclomatic complexity of the function is calculated using any of the ways outlined above. In this case, we used the formula $V(G) = P + 1$, where P is the number of node predicates in the function and $V(G)$ is the cyclomatic complexity. Assigning a value to the variable P, we obtain $P = 2$ and if the full calculation is done, the result is $V(G) = 3$. So there are three test cases needed to cover all lines of code in the function, as well as the verification of conditionals in their true and false options.

Taking as an input source code of the method in Fig. 1 the solution proposed in [8], a set of paths is obtained which represent the graph in Fig. 2. From this information, we can generate values for each path by analyzing its structure, specifically the conditionals, and applying the decision/condition coverage technique explained above. For the example of the method presented in Fig. 1, there are two choices D1 >> "x <= 0" and D2 >> "getBalance() <= x". The specific data for test cases can be as in Table 3.

Considering the decisions D1 and D2, we have the following three test cases:

A result of the application of this technique is that three test cases must go across the code completely. We noted that there is a match between the results of both techniques. In this case, three test cases should be performed for the sample method.



$$\begin{tikzcd} & 1 \arrow[dl] \arrow[dr] & \\ 2 & & 3 \arrow[ll] \arrow[dl] \\ & 4 & \end{tikzcd}$$

Fig. 2. Flow graph of the method "Withdraw"

Table 2. Value assignment

	True value	False value
D1	X<=0	x>0
D2	getBalance() <=x	getBalance() >x

Table 3. Test cases and combination of values

	Test Case 1	Test Case 2	Test Case 3
D1	True	False	False
D2	False	True	False
Exit	The exception is raised, balance cannot be withdrawn.	The exception of insufficient balance rises.	Records a new movement of money.

Next, a Java class that contains the test method of Fig. 1 is implemented.

Fig. 3 shows how to make a statement which contains the test method. A void type method for

```
public void testWithdraw()
{
  try
   {
     account.withdraw (1000);
   }
   catch (Exception e) {}
   assertTrue(account.getBalance() == 0.0)
}
```

Fig. 3. Test method declaration

each test to be performed must be created. The name of the method to be tested must be with the ¨test¨ prefix. Test cases are considered successful or unsuccessful depending on the sentence to be included in the test cases, and it is of the Assert type which is the claim of a proposition (code line) in a program where the developer places it wherever he/she considers that its statement is always true. Then *assertTrue* is used whenever we want to validate that the condition is true.

```
public List<LEVEL_COMP> LevelComp_Employee
(string pCI)
{
     NO_EMPL employee = GetEmployeeByCI(pCI);
     if (employee!= null)
     {
        String pos =
                      Convert.ToString(employee.ID_EMPL);
       NO_POS position = PositionEmpl(employee.ID);

      var query = (from c in context.POS_COMP
                         join d in context.LEVEL_COMP on
                      c.NO_COMPEID equals
                      d.NO_COMPEID
                         where c.NO_POSID == position.ID
                         select d.Distinct();
     List<LEVEL_COMP> listQuery;
     try
     {
         listQuery = query.ToList();
     }
     catch (System.Exception ex)
    {
      throw new FaultException<UnknowException>(new
        UnknowException    (ex.Message), ex.Message);
    }
    List<LEVEL_COMP> list =
                         new List<LEVEL_COMP>();
    foreach (var item in listQuery)
    {
        list.Add (item);
    }
      return list;
     }
  return null;
}
```

Fig. 4. "Employee Competency Levels" method

5.2 Unit Testing in C# Code and Using Visual Studio (Mstest)

For demonstrating how to use the Visual Studio MSTest tool, a test method applied to the method "LevelComp_Employee" for “Manage Skills levels” has been developed, where a competency can be generic or technique, the first are the emotional and behavioral characteristics and the other is specific skills or techniques which people present.

The method is shown in Fig. 4. This method receives as a parameter an identification card, and calls another method which searches for a particular identity card in a list of employees to verify if the employee exists. If the employee exists, he/she is sought by the employee position as its identifier, then due to that assignment, the query is performed by searching for office skill IDs, and a list of those skill levels which the

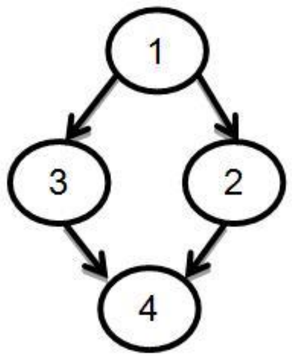


Fig. 5. Flow graph of the method "Employee Competency Levels”

Table 4. Value assignment

	True value	False value
D1	Emp!=null	Emp!=null

Table 5. Combination of test cases and securities

	Test Case 1	Test Case 2
D1	True	False
Exit	Full list	EmptyList

employee seeks is returned, otherwise the method returns an empty list.

The same basic path technique as in “Withdraw” method is applied. Due to this, the flow graph is made up (1) using the cyclomatic complexity whose value is equal to two, it is determined with the formula $V(G) = P + 1$. So two test cases are obtained which are necessary to cover all lines of code in the function; (2) using the technique of decision/condition coverage.The decision D1 >> "emp! = Null" is obtained. Specific data for the test cases can be as in Table 4.

Considering the decision D1, we have two test cases presented in Table 5.

As a result of the application of this technique, there are two test cases necessary to go across the code completely. It can be noted that the results of both techniques match. There are two test cases to be performed for the sample method. Then a Visual Studio class containing each of the test cases with data required is implemented.

As shown in Fig. 6, TestMethod() is automatically assigned to each test method. Each test corresponds to a unique method in the test code to be tested. Test methods are stored in a test class that is assigned to the attribute TestClass().

```
Namespace EtesTesting
{
 [TestClass]
  public class  CompetenciesTesting
   {
     [TestMethod]
     public void LevelComp_Employee()
      {

   EtesTesting.Indy.Competencies.NomenclCompetClient
   client = new
   Indu.Competencies.NomenclCompetClient();
    string ci = “87032209140”;
    client.LevelComp_Employee(ci);
        }
     }
   }
```

Fig. 6. Test method declaration

6 Future Work

This experiment has opened new opportunities for further research. The proposed method currently works in the following directions:

5. Adding new functionality to the solution in [8] which can generate a sequence of instructions to be executed by each test case paths, from a code to a specific method.
6. Suggesting algorithms to process each of the paths identified, determining the conditional execution flow resulting in the combination of values needed to test this way.
7. Integration of these results with automatic test execution tools so that test cases with corresponding values are generated in the language of this tool.

7 Conclusions

During this work, the techniques for designing test cases have been identified and implemented as practical examples. Tools that support unit testing for the codes C# and Java were also selected, as they are the languages most used in projects of software development in the university environment. We identified the need to combine, in the environments of software development at a particular university, automatic test execution tools with other tools which generate paths and values for each of the test cases.

References

1. **Alba M.,*“*** *A Test Generation Solution to Automate Software Testing”.* Journal Advances in Systems and Computer Science, ISSN 1657-7663, Medellín, Vol. 8, No. 2, 2011.
2. **Bardin, S. & Hermann, P. (2008).** Structural Testing of Executables. *1st International Conference on Software Testing, Verification, and Validation*, Lillehammer, Norway, 22–31.
3. **Bouquet, F., Grandpierre, C., Legeard, B., & Peureux, F. (2008).** *A* Test Generation Solution to Automate Software Testing. *3rd International Workshop on Automation of Software Test (AST’08)*, Leipzig, Germany, 45–48.

4. **Braude, E.J. (2001).** *Software engineering: an object-oriented perspective.* New York: Wiley.
5. **Bregieiro, J.C., Zenha, M., & Fernandéz, F. (2008).** A Strategy for Evaluating Feasible and Unfeasible Test Cases for the Evolutionary Testing of Object-Oriented Software. *3rd International Workshop on Automation of Software Test* (AST'08), Leipzig, Germany, 85–92.
6. **López, C., Yañez, C., Gutierrez, A., & Felipe, E. (2008).** Adequacy Checking of Personal Software Development Effort Estimation Models Based upon Fuzzy Logic: A Replicated Experiment. *Computación y Sistemas*, 11(4), 333–348.
7. **Craig, R.D. & Jaskiel, S.P. (2002).** *Systematic Software Testing*. Boston: Artech House.
8. **De la Torre W.,** "*Component for automatic generation of test cases paths".* Thesis, ISPJAE, Havana, Cuba, 2012.
9. **Gutiérrez, J.J., Escalona, M.J., Mejías, M., & Torres, J. (2006).** Modelos y Algoritmos para la Generación de Objetivos de Prueba. *XV Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2006).* Retrieved from *www.lsi.us.es/~javierj/publications/JISBD35.pdf.*
10. **Gutiérrez, J.J., Escalona, M.J., Mejías, M., & Reina, A.M. (2006).** Modelos de Pruebas para Pruebas del Sistema, *XV Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2006),* Retrieved from *http://users.dsic.upv.es/workshops/dsdm06/files/dsdm06-07-Gutierrez.pdf.*
11. **Gutiérrez, J.J., Escalona, M.J., Mejíuas, M., & Torres, J. (2007).** Derivation of test objectives automatically. *Advances in Information Systems Development*, (435–446), New York, NY : Springer.
12. **Gutiérrez, J.J., Escalona, M.J., Mejías, M., Torres, J., & Torres-Zenteno, A. (2007).** Generación automática de objetivos de prueba a partir de casos de uso mediante partición de categorías y variables operacionales, *XII Jornadas de Ingeniería del Software y Bases de Datos*, Retrieved from *www.lsi.us.es/~javierj/publications/JISBD07.pdf.*
13. *IEEE Standard Glossary of Software Engineering Terminology.* 610.12-1990.
14. **JUnit., (s.f.).** *JUnit Official Site* [Ref. of May 16, 2012]. Retrieved from www.junit.org.
15. **TestNG. (s.f.).** *TestNG Official Site* [Ref. of May 24, 2012]. Retrieved from www.testng.org.
16. **Jtest. (2008).** *Parasoft Jtest* [Ref. of May 30, 2012]. Retrieved from http://odinlatin.com/wp-content/uploads/2010/10/Jtest_NEWDS06_ESP_PDF21.pdf.
17. **Mendoza, L.E., Pérez, M.A., & Grimán A.C. (2005).** Prototipo de Modelo Sistémicode calidad (MOSCA) del Software. *Computación y Sistemas*, 8(3), 196–217.
18. **Myers, G.J. (2004).** *The Art of Software Testing*. (2nd ed.). Hoboken, N.J.: John Wiley & Sons.
19. **MSDN. (s.f.).** *Información general de pruebas unitarias.* [Ref. of June 8, 2012]. Retrieved from http://msdn.microsoft.com/es-es/library/ms182516%28v=vs.80%29.aspx.
20. **NUnit. (2002-2007).** *NUnit Official Site*, [Ref. on June 3, 2012]. Retrieved from http://www.nunit.org/.
21. **Pfleeger, S.L. (2006).** *Software engineering: Theory and Practice* (3rd ed.). Upper Saddle River, N.J.: Pearson/ Prentice-Hall.
22. **Polo, D.** "*Definition of a software development process in a university setting".* Studies Center and Systems Engineering. Havana, Cuba, Instituto Superior Politecnico Jose Antonio Echaverría, 2011.
23. **Pressman, R.S. (2005).** *Software Engineering, A practitioner's Approach (6th ed.).* Boston, Mass.: McGraw-Hill,
24. **Piattini., *"Analysis and Design of Computer Applications Management. A software engineering perspective"*, 2007.**
25. **Patton, R. (2006).** *Software Testing* (2nd Ed.). Indianapolis, IN: Sams Publishing.
26. **Rodríguez E., *"Importance of software testing"*, 2011.**
27. **Sevilla et al.,** "*Open HMI Tester*", 2010, [Ref. on October 5, 2012]. Available at: http://www.catedrasaes.org/trac/wiki/ProjectsOht
28. **TheFreeLibrary. (2002).** *I-Logix Launches Statemate MAGNUM Automatic Test Generator*, [Ref. on October 5, 2012]. Retrieved from *http://www.thefreelibrary.com/ILogix+Launches+Statemate+MAGNUM+Automatic+Test+Generator.-a083021729.*
29. **Usaola**., *"Testing Information Systems"*. University of Castilla-La Mancha Department of Technology and Information Systems
30. **Unitils. (2011).** *Unitils Official Site*, [Ref. on June 4, 2012]. Retrieved from http://www.unitils.org.
31. **Yagüey, A. & Garbajosa, J. (2009).** Comparativa práctica de las pruebas en entornos tradicionales y ágiles. *Revista Española de Innovación, Calidad e Ingeniería del Software (REICIS).* 5(4), 19–32.

Darlene Gómez received the B.Sc. degree in Informatics Engineering from the Higher Polytechnic Institute José Antonio Echeverría (CUJAE), Havana, Cuba, in 2010. Her research areas are Software Engineering and Software Quality.

Dalila Jústiz received the M.Sc. degree at Higher Polytechnic Institute José Antonio Echeverría (CUJAE), Havana, Cuba. Her research areas are Software Engineering and Software Quality.

Martha Delgado received her M.Sc. and Ph.D. degrees from Higher Polytechnic Institute José Antonio Echeverría (CUJAE), Havana, Cuba. Her research areas are Software Engineering and Software Quality.

Article received on 23/10/2012; accepted on 07/01/2013.