ISSN 2395-8618

GeCaP: Generador de casos de pruebas unitarias a partir del código fuente en lenguaje Java

D. Larrosa, P. Fernandez, and M. Delgado

***Resumen*—Las pruebas de software, no obstante que son costosas, aumentan considerablemente la confiabilidad y calidad de los sistemas, contribuyendo así a su posicionamiento en el mercado. Específicamente, las pruebas unitarias se encargan de probar que las unidades individuales del diseño de software, componente o módulo de software, funcionan correctamente. Aunque existen herramientas que se encargan de ejecutar pruebas unitarias de manera automática, éstas carecen de funcionalidades que proporcionen apoyo y asistencia al desarrollador en el diseño de los casos de prueba; además, las propuestas existentes para el diseño de los casos de pruebas unitarias, no se han insertado al entorno productivo y no permiten generar código de pruebas. En el presente trabajo se propone una herramienta que permite la generación automática de casos de pruebas unitarias a partir del código fuente en lenguaje Java. En la nueva propuesta se utiliza la técnica del camino básico para el diseño de los casos de prueba. De forma automática se genera el grafo de control de flujo del código fuente a probar, para luego generar los caminos independientes; por último, se generan las combinaciones de valores de prueba que satisfagan todos y cada uno de los caminos independientes. En el proceso de implementación de la nueva herramienta, se diseñó un caso de estudio para efectos de validación; se aplicaron algoritmos metaheurísticos para la generación de valores de prueba y para la generación de combinaciones de valores para cada camino, y se compararon estas combinaciones de valores con las obtenidas por otros algoritmos del estado del arte. Dado que en el caso de estudio se alcanza un 100% de cobertura de los caminos independientes la nueva herramienta exhibe resultados competitivos respecto de los resultados obtenidos por herramientas propuestas por otros autores.**

***Palabras clave*—pruebas unitarias, técnica del camino básico, generación automática de casos de prueba, algoritmos metaheurísticos.**

Manuscrito recibido el 10 de octubre de 2017, aceptado para la publicación el 17 de febrero de 2018, publicado el 30 de junio de 2018.

Danay Larrosa está con la Universidad Tecnológica de La Habana José Antonio Echeverría (CUJAE), La Habana, Cuba (correo: dlarrosau@ceis.cujae.edu.cu).

Perla Beatriz Fernández Oliva está con la Universidad Tecnológica de La Habana José Antonio Echeverría (CUJAE), La Habana, Cuba (correo: perla@ceis.cujae.edu.cu).

Martha Dunia Delgado Dapena está con la Universidad Tecnológica de La Habana José Antonio Echeverría (CUJAE), La Habana, Cuba (correo: marta@ceis.cujae.edu.cu).

GeCaP: Unit Testing Case Generation from Java Source Code

***Abstract*—Software testing, despite its cost, considerably improves the reliability and quality of systems, contributing to their positioning in the market. Specifically, unit testing is the process by which the correct individual functioning of modules, components, and design is ensured. Even though tools that execute unit testing automatically exist, these lack the ability to provide the developer with support and assistance in the design of test cases; furthermore, the current proposals for test case design in unit testing have not been inserted into the production environment and are unable to generate testing code. The present work proposes a tool that allows developers to automatically generate test cases for unit testing from Java source code. In this new proposal the basis path testing technique is used for the design of the test cases. The control flow graph is automatically generated from the source code being tested, in order to subsequently generate the independent paths. Finally, the combinations of test values that satisfy each and every one of the linearly independent paths are generated. In the process of implementing this new tool a case study was designed for the purpose of validation; metaheuristic algorithms were applied to generate test values and value combinations for each path. These combinations were compared against the ones obtained by other state-of-the-art algorithms. Since in this case study a 100% coverage of the independent paths is reached, the proposed tool exhibits competitive results with respect to the ones reported by tools proposed by other authors.**

***Index terms*—unit tests, basic path technique, automatic generation of test cases, metaheuristics algorithms.**

I. Introducción

En la actualidad, la demanda de software ha aumentado considerablemente como consecuencia del avance de la tecnología. De esta forma, se hace necesario que los productos de software obtengan una certificación de calidad siendo la mejor manera de competir en un mercado en crecimiento que es cada vez más exigente [1].

Las pruebas de software continúan ocupando espacio en los trabajos científicos de múltiples investigadores: En particular,

ISSN 2395-8618

se mantienen como problemas abiertos la generación de caminos y valores de pruebas para apoyar el diseño de los casos de prueba [2; 3; 4; 5; 6], así como los procesos vinculados con las pruebas de software [2; 7; 8; 9; 10; 11]. Estas propuestas van desde la utilización de algoritmos de optimización e inteligencia artificial para resolver el problema de la explosión combinatoria de los caminos y valores de pruebas, hasta propuestas de *frameworks* para lograr la automatización de algunos elementos del proceso. En este último caso las propuestas son prototipos para validar la solución teórica, pero no se han incorporado a las soluciones comerciales los elementos de generación de los casos de prueba, de forma tal que puedan ser utilizados por desarrolladores y equipos de probadores, reduciendo así el esfuerzo vinculado con esta actividad de diseño que es altamente costosa. Además, las propuestas mencionadas anteriormente no llegan a la generación de código de pruebas unitarias.

Existen diferentes herramientas como JUnit [12], NUnit [13] y PHPUnit [14] que permiten ejecutar pruebas unitarias de forma automática, pero carecen de funcionalidades que asistan al desarrollador en el diseño de los casos de pruebas. Ello se debe a que, a pesar de que estas herramientas crean automáticamente una clase de prueba con un método de prueba vacío, el desarrollador debe llenar el método de prueba y crear el resto de los métodos que necesite.

Se hace necesario automatizar la generación de casos de pruebas unitarias a partir del código fuente. Para solucionar la problemática existente, el objetivo del presente trabajo es desarrollar una herramienta para la generación de casos de pruebas unitarias a partir del código fuente en lenguaje Java. La herramienta estará insertada en el propio ambiente de desarrollo, por lo que brinda apoyo al programador en cuanto al diseño de los casos de pruebas y se disminuye el tiempo y esfuerzo dedicado a esta tarea.

II. MATERIALES Y MÉTODOS

En el presente trabajo se utilizó la técnica del camino básico para el diseño de los casos de pruebas. Como se ilustra en la Figura 1, esta técnica permite ejecutar todas las instrucciones del código fuente al menos una vez.

Las actividades que aparecen sombreadas en gris corresponden a propuestas que han llegado a soluciones para los entornos productivos, y las que aparecen sombreadas en azul, corresponden a propuestas teóricas que no han sido insertadas en el entorno industrial. Como se puede observar, el diseño de casos de prueba cuenta con algunas propuestas no incorporadas al entorno productivo, por lo que el problema sigue sin resolverse en el entorno de producción; además, no cuenta con herramientas que generen el código de pruebas necesario para su posterior ejecución con las herramientas existentes.

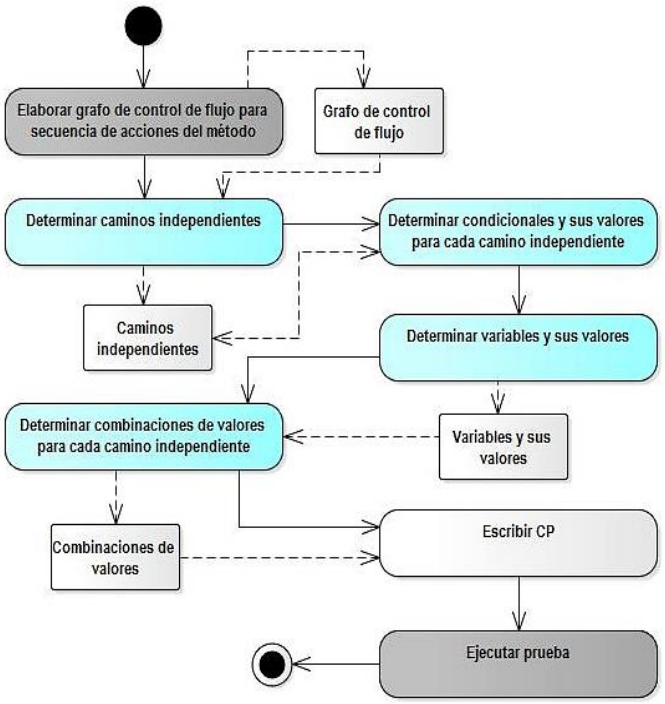

Fig. 1. Procedimiento para realizar pruebas unitarias.

Teniendo en cuenta este procedimiento, en la herramienta desarrollada en el presente trabajo se genera el grafo de control de flujo de forma automática a partir del código fuente de un método en lenguaje Java. Para la generación del grafo de control de flujo se utiliza la herramienta ANTLR (Herramienta para Reconocimiento de Lenguaje).

ANTLR es una herramienta que provee un *framework* para construir reconocedores, compiladores y traductores de descripciones gramaticales para lenguajes de dominio específico. Los lenguajes de dominio específico incluyen formatos de datos, formatos de ficheros de configuración, protocolos de red, lenguajes de procesamiento de texto, secuencia de genes, lenguajes de control de sondeo de espacio, y lenguajes de programación de dominio específico. Tiene soporte de generación de código en diferentes lenguajes de programación, tales como: Java, C#, Python, Ruby, Objective-C, C y C++. Además, permite generar un árbol de sintaxis abstracta (AST por sus siglas en inglés) con la secuencia de acciones del método [15].

Un AST es una representación de árbol de la estructura sintáctica abstracta (simplificada) del código fuente escrito en cierto lenguaje de programación. Cada nodo del árbol denota una construcción que ocurre en el código fuente. La sintaxis es abstracta en el sentido que no representa cada detalle que aparezca en la sintaxis verdadera [15].

En la figura 2 se muestra un diagrama UML con las actividades necesarias para obtener un grafo de control de flujo mediante la utilización de la herramienta ANTLR.

Con objeto de determinar los caminos independientes a partir del grafo de control de flujo, se desarrolló un algoritmo

ISSN 2395-8618

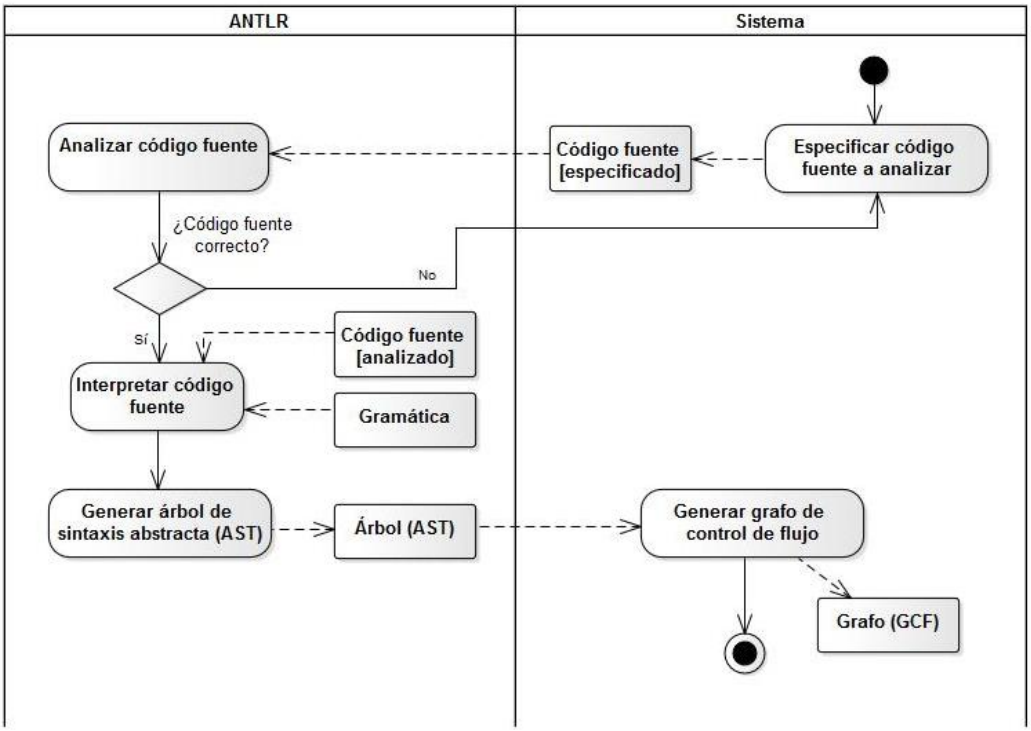

Fig. 2. Flujo de actividades para generar un grafo de control de flujo.

encargado de transformar el grafo de control de flujo a un grafo de condicionales que permite obtener los casos de prueba independientemente del lenguaje del código fuente.

Este algoritmo se describe de la siguiente manera:

1) Procesar sólo los nodos que representen una condicional.

2) En caso de que exista una sentencia *switch*, cambiar a un conjunto de condicionales, donde cada vértice tenga grado de salida dos.

3) En caso de que exista una condición compuesta (esto ocurre cuando uno o más operadores booleanos se presentan en una instrucción condicional), ésta se transforma en varias condicionales simples; para lograrlo, se tienen en cuenta los operadores booleanos asociados a cada una de las condicionales simples.

El grafo de condicionales facilita la obtención de los caminos independientes, debido a que sólo contiene los nodos que generan nuevos caminos, los cuales son precisamente los nodos condicionales. Además, la cantidad de nodos del grafo de condicionales más uno representa la cantidad de caminos independientes (caminos que poseen al menos una nueva arista), y la cantidad de casos de prueba. En la Tabla I se muestran las principales diferencias entre el grafo de control de flujo y el grafo de condicionales.

Una vez transformado el grafo de control de flujo en un grafo de condicionales, se utiliza el algoritmo de búsqueda en profundidad para obtener todos los posibles caminos del grafo, y luego se procede a eliminar los caminos redundantes. De esta forma, sólo se obtienen los caminos independientes.

A partir de los caminos independientes, se obtienen los casos de pruebas (caminos y valores asociados); para ello, se utilizaron algoritmos metaheurísticos de búsqueda para la generación de los valores y sus combinaciones para cada camino, como se indica en [1] y [2].

TABLA I
DIFERENCIAS ENTRE EL GRAFO DE CONTROL DE FLUJO Y EL GRAFO DE CONDICIONALES

Aspectos a tener en cuenta	Grafo de control de flujo	Grafo de condicionales
Vértices del grafo	Todas las instrucciones del código fuente.	Instrucciones que representan una condicional.
Grado de salida de los vértices del grafo	Si el vértice es una instrucción secuencial, grado de salida 1; si es una instrucción condicional, grado de salida 2 o más.	Grado de salida 2 a lo sumo.
Condicionales compuestas	Las instrucciones que representan condicionales compuestas se mantienen igual.	Las instrucciones que representan condicionales compuestas se transforman en condicionales simples, teniendo en cuenta el o los operadores que las relacionan.

III. PROPUESTA DE SOLUCIÓN

Para la generación de casos de pruebas unitarias a partir del código fuente en lenguaje Java, se desarrolló un *plug-in* en el

ISSN 2395-8618

entorno de desarrollo Eclipse, de forma tal que el programador puede seleccionar el método a probar y, en el propio proyecto bajo prueba, se genera una clase que contiene los métodos de prueba del método seleccionado. El método de prueba incluye el caso de prueba que responde a un determinado camino y su valor esperado.

En la Figura 3 se pueden observar, a través de un diagrama UML de casos de uso, las funcionalidades del *plug-in* desarrollado y su relación con los componentes de generación de valores y de combinaciones de valores para cada camino independiente.

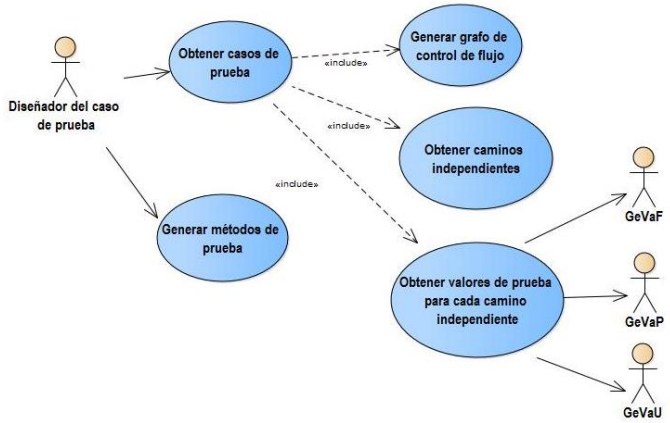

Fig. 3. Funcionalidades del plug-in para generar casos de pruebas unitarias en lenguaje Java.

El *plug-in* permite obtener los casos de pruebas unitarias a partir del código fuente de un método en lenguaje Java, mediante la generación automática de un grafo de control de flujo; luego, se obtienen los caminos independientes.

A fin de obtener las combinaciones de valores de prueba para cada camino, se utilizan tres componentes, GeVaF: encargado de generar valores de pruebas a partir de la descripción del dominio de las variables que intervienen en el grafo de control de flujo; GeVaP: encargado de generar valores de prueba teniendo en cuenta las técnicas de diseño de casos de prueba: de bucles y de condiciones; y GeVaU: encargado de generar combinaciones de valores de prueba para cada camino independiente, a partir de los valores generados por GeVaF y GeVaU.

Posteriormente, se genera un conjunto de métodos de prueba que pueden ser ejecutados con la herramienta JUnit.

IV. RESULTADOS Y DISCUSIÓN

La solución propuesta permite generar casos de pruebas unitarias mediante la utilización de técnicas de diseño de casos de prueba de Ingeniería de Software y algoritmos metaheurísticos. Se realizó una comparación de los valores de prueba generados para cada camino independiente con algoritmos propuestos por autores que trabajan el tema en la comunidad científica; paara ello, se utilizó el problema de clasificación del triángulo.

A continuación se describe el problema de clasificación del triángulo. Los resultados obtenidos se presentan en la tabla II.

TABLA II
RESULTADOS OBTENIDOS AL GENERAR COMBINACIONES DE VALORES PARA EL ALGORITMO DE CLASIFICACIÓN DEL TRIÁNGULO

Algoritmo propuesto por	Cantidad de combinaciones	Tiempo promedio (en segundos)
Jones	17789	8.4
Díaz	587	1.09
Lanzarini	51	1.03
Solución propuesta	5	0.853

El problema de clasificación del triángulo tiene tres variables de entrada (A, B, C) que representan la longitud de los lados de la figura. El programa determina, en cada caso, si la entrada corresponde, o no, a un triángulo; y en caso afirmativo, genera el tipo de triángulo: escaleno, equilátero, isósceles.

En la tabla II se muestra la cantidad de combinaciones que requirió cada uno de los algoritmos que se compararon experimentalmente, luego de realizar 2000 iteraciones en el problema de clasificación del triángulo; se incluye, además, el tiempo promedio que tardó cada propuesta en lograr el 100% de cobertura.

Al aplicar la solución propuesta en el presente trabajo de investigación al problema de clasificación de triángulo, se identificaron 5 caminos de prueba que permiten el 100% de cobertura. Considerado que en el comparativo se invirtió el menor tiempo en segundos (0.853), es evidente la superioridad de los resultados obtenidos con la nueva propuesta sobre los algoritmos respectivos de Jones, Díaz y Lanzarini: la nueva propuesta exhibe cobertura en el 100% de los caminos de prueba, genera un conjunto reducido de valores para esos caminos y, además, obtiene los resultados en menos tiempo que el propuesto en [20], [19], [18].

Adicionalmente a los resultados experimentales previos que muestran la superioridad de la nueva propuesta respecto de los algoritmos del estado del arte, en el presente trabajo de investigación se diseñó un caso de estudio para evaluar el valor práctico de la solución propuesta.

Se describe el contexto utilizado en el caso de estudio, se propone un conjunto de preguntas de estudio y se ilustra cómo la solución propuesta genera respuestas válidas y prácticas a esas preguntas de estudio. El código se incluye en la Figura 4.

Contexto: Para generar el código de prueba del caso de estudio, se utiliza el código fuente del algoritmo para la Serie de Fibonacci (ver Figura 4). El entorno de desarrollo en el que se muestra la solución es Eclipse debido a que el *plug-in* se construyó para ese entorno. Además, la herramienta de ejecución de pruebas unitarias que se seleccionó fue JUnit porque permite realizar pruebas para el lenguaje de programación Java.

ISSN 2395-8618

```java
public int SerieFibonacci (int limiteSerie) {
    int result=-1, a, b;
    if(limiteSerie==0 || limiteSerie==1) {
        result=limiteSerie;
    }
    else {
        a = -1;
        b = 1;
        for(int i = 0; i <= limiteSerie; i++) {
            result = a + b;
            a = b;
            b = result;
        }
    }
    return result;
}
```

Fig. 4. Código fuente del caso de estudio.

Preguntas de estudio y proposiciones:

1) ¿Cómo obtener caminos independientes a partir de código fuente en Java?

Para obtener caminos independientes a partir de código fuente en Java es necesario contar con un grafo de control de flujo del código fuente en Java, el cual es generado por el *plug-in* diseñado en el presente trabajo de investigación; luego, se generan los caminos independientes, a partir del grafo de control de flujo. En la Figura 5 se pueden observar los caminos generados.

Caminos independientes:

Caminos/Condiciones	limiteSerie==0	limiteSerie==1	i<=limiteSerie
C1	T	-	-
C2	F	T	-
C3	F	F	T
C4	F	F	F

Fig. 5. Caminos independientes generados por el *plug-in*.

2) ¿Cómo obtener casos de pruebas unitarias de forma automática para ejecutar todas las instrucciones del código fuente?

Una vez generados los caminos independientes, se generan los valores interesantes con los componentes GeVaF y GeVaP. Posteriormente, a partir de los caminos, las condiciones y los valores interesantes, se generan las combinaciones de valores de prueba para cada camino independiente mediante el uso del componente GeVaU. En la Figura 6 se muestran los casos de prueba generados (caminos y valores asociados).

Como se puede observar en la figura anterior, el diseñador del caso de prueba debe especificar el resultado esperado en cada caso de prueba, teniendo en cuenta la combinación de valores de prueba generada para cada camino independiente.

Casos de prueba:

Caminos/Variables	limiteSerie	Valor esperado
1	0	0
2	1	1
3	5	5
4	-1	-1

Fig. 6. Casos de pruebas (caminos y valores asociados) generados por la herramienta.

3) ¿Cómo insertar los casos de pruebas unitarias en un entorno de prueba específico?

Para insertar los casos de pruebas unitarias en un entorno de prueba específico se utiliza el *plug-in* desarrollado en el presente trabajo, lo cual se llevó a cabo en el entorno Eclipse para el lenguaje Java.

En caso que se quieran insertar los casos de pruebas en otros entornos, bastaría con desarrollar un *plug-in* para el entorno requerido, siempre y cuando lo permita. Se recomienda el desarrollo de un *plug-in* debido a que facilita el trabajo del diseñador del caso de prueba; se debe, además, seleccionar la herramienta a utilizar para realizar las pruebas unitarias.

A diferencia de la herramienta JUnit, que solamente genera un método de prueba vacío (ver Figura 7), el *plug-in* desarrollado ofrece la implementación de los métodos de prueba necesarios para satisfacer cada camino independiente.

```java
package pruebas_unitarias;

import static org.junit.Assert.*;

import org.junit.Test;

public class TestEjemploTesisJUnit {

    @Test
    public void testIsPrime() {
        fail("Not yet implemented");
    }

}
```

Fig. 7. Método de prueba creado por JUnit.

En la Figura 8 se muestran los métodos de prueba generados por el *plug-in*, a partir de los casos de pruebas obtenidos previamente para ejecutarlos con JUnit.

Como se puede observar en la figura anterior, el diseñador del caso de prueba debe especificar el resultado esperado en cada caso de prueba, teniendo en cuenta la combinación de valores de prueba generada para cada camino independiente.

Como se puede observar, se genera un método de prueba por cada caso de prueba obtenido, teniendo en cuenta la combinación de valores generada para cada camino, además del resultado esperado especificado previamente.

ISSN 2395-8618

```java
package pruebas_unitarias;

import org.junit.Assert;
import org.junit.Test;
import ejemploTesis.EjemploTesis1;

public class TestEjemploTesis1  {

    //Este método de prueba hace referencia al camino C1: limiteSerie==0,
    @Test
    public void test_SerieFibonacci_CP1() {
        EjemploTesis1 ejemploTesis1 = new EjemploTesis1();
        int expected = 0;
        Assert.assertEquals(expected, ejemploTesis1.SerieFibonacci(0));
    }
    //Este método de prueba hace referencia al camino C2: limiteSerie==0,
    @Test
    public void test_SerieFibonacci_CP2() {
        EjemploTesis1 ejemploTesis1 = new EjemploTesis1();
        int expected = 1;
        Assert.assertEquals(expected, ejemploTesis1.SerieFibonacci(1));
    }
    //Este método de prueba hace referencia al camino C3: limiteSerie==0,
    @Test
    public void test_SerieFibonacci_CP3() {
        EjemploTesis1 ejemploTesis1 = new EjemploTesis1();
        int expected = 5;
        Assert.assertEquals(expected, ejemploTesis1.SerieFibonacci(5));
    }
    //Este método de prueba hace referencia al camino C4: limiteSerie==0,
    @Test
    public void test_SerieFibonacci_CP4() {
        EjemploTesis1 ejemploTesis1 = new EjemploTesis1();
        int expected = -1;
        Assert.assertEquals(expected, ejemploTesis1.SerieFibonacci(-1));
    }
}
```

Fig. 8. Código de pruebas generado por la herramienta JUnit.

4) ¿Cómo ejecutar pruebas unitarias en un entorno específico?

Para ejecutar pruebas unitarias en un entorno específico, se hace necesario utilizar una herramienta para realizar pruebas unitarias para un determinado lenguaje. En la solución propuesta se utiliza la herramienta JUnit porque ejecuta pruebas unitarias para el lenguaje de programación Java, lenguaje para el cual fue creado el plug-in. En la Figura 9 se muestran los resultados de una ejecución de los métodos de prueba generados por el *plug-in*.

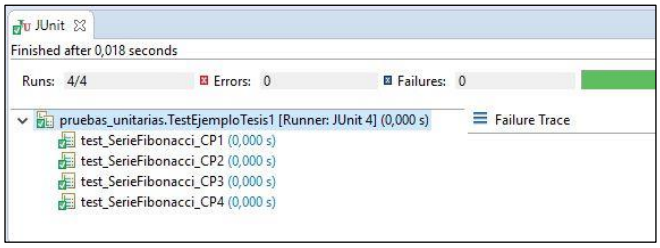

Fig. 9. Ejecución del código de prueba generado con la herramienta JUnit.

V. CONCLUSIONES

Los resultados del presente trabajo de investigación facilitan el proceso de pruebas unitarias durante el desarrollo de productos de software, debido a que permiten automatizar el diseño de casos de pruebas unitarias y la generación de código de pruebas unitarias en lenguaje Java. Adicionalmente, estos resultados permiten reducir el tiempo y esfuerzo dedicados por desarrolladores, probadores y diseñadores de casos de prueba en el diseño y ejecución de pruebas unitarias. Debido a que los valores de prueba generados tienen en cuenta las técnicas de bucles y condicionales, las combinaciones de valores generados satisfacen todos los caminos independientes. De esta forma, se alcanza un 100% de cobertura de caminos independientes permitiendo ejecutar cada instrucción del código fuente del método a probar, con el objetivo de detectar errores.

REFERENCIAS

[1] Equipo del Producto CMMI, "CMMI para Desarrollo, Versión 1.3," CMMI-DEV, V1.3. Software Engineering Institute, Hanscom AFB, Massachusetts, Tech. Rep. ESC-TR-2010-033, Nov. 2010.

[2] M. B. Chrissis, M. Konrad, and S. Shrum, *CMMI for Development. Guidelines for Process Integration and Product Improvement.* 3rd. ed., USA: Pearson Education, 2011, pp. 123–135.

[3] S. Sekhara, M. L. Hary, U. Kiran, et al. (2012, marzo). Automated Generation of Independent Paths and Test Suite Optimization Using Artificial Bee Colony. *Procedia Engineering.* [Online]. *30*, pp. 191-200. Available: isiarticles.com/bundles/Article/pre/pdf/7408.pdf

[4] A. Pachauri and G. Srivastava. (2013, enero). Automated test data generation for branch testing using genetic algorithm: An improved approach using branch ordering, memory and elitism. *The Journal of Systems and Software.* [Online]. *86(5)*, pp. 1191-1208. Available: https://www.researchgate.net/publication/256991955_Automated_test_data_generation_for_branch_testing_using_genetic_algorithm_An_improved_approach_using_branch_ordering_memory_and_elitism

[5] P. Ranjan, B. Mallikarjun and X. Yang. (2012, septiembre). Optimal test sequence generation using firefly algorithm. *Swarm and Evolutionary Computation.* [Online]. *8*, pp. 44-53. Available: https://pdfs.semanticscholar.org/bbdc/692a58b3517d66b4b0e000a7e0fcb8cc9e3d.pdf

[6] G. Carvalho, D. Falcão, F. Barros, et al. (2014, junio). NAT2TESTSCR: Test case generation from natural language requirements based on SCR specifications. *Science of Computer Programming.* [Online]. *95(3)*, pp. 275-297.

[7] I. Hermadi, C. Lokan and R. Sarker. (2014, enero). Dynamic stopping criteria for search-based test data generation for path testing. *Information and Software Technology.* [Online]. *56(4)*, pp. 395-407. Available: https://www.researchgate.net/publication/260009614_Dynamic_Stopping_Criteria_for_Search-based_Test_Data_Generation_for_Path_Testing

[8] F. Elberzhager, A. Rosbach, J. Münch and R. Eschbach. (2012, mayo). Reducing test effort: A systematic mapping study on existing approaches. *Information and Software Technology.* [Online]. *54(10)*, pp. 1092-1106. Available: http://www.juergenmuench.com/publications/uploads/cd60a34afaa5f50e7422113e7d8941ef4ce84456.pdf

[9] T. Rongfa, "Adaptive Software Test Management System Based on Software Agents," in Advanced Technology in Teaching - Proceedings of the 2009 3rd International Conference on Teaching and Computational Science (WTCS 2009), vol. 117, Y. Wu, E d . Berlin: Springer Berlin Heidelberg, 2012, pp. 1–9.

[10] T. Chen, X. Zhang, S. Guo, et al. (2012, marzo). State of the art: Dynamic symbolic execution for automated test generation. *Future Generation Computer Systems.* [Online]. *29(7)*, pp. 1758-1773. Available: https://pdfs.semanticscholar.org/02af/b1a4a45cfea0c1aeffca4a441e6541a5d34b.pdf

[11] X. Ying, G. Yun-zhan, W. Ya-wen and Z. Xu-zhou. (2014, abril). Intelligent test case generation based on branch and bound. *The Journal of China Universities of Posts and Telecommunications.* [Online]. *21(2)*, pp. 91-97. Available:

ISSN 2395-8618

http://www.juergenmuench.com/publications/uploads/cd60a34afaa5f50e7422113e7d8941ef4ce84456.pdf

[12] JUnit. (2017, Enero). Sitio official de JUnit. [Online]. Available: www.junit.org

[13] NUnit. (2015). Sitio official de NUnit. [Online]. Available: www.nunit.org

[14] S. Bergmann. (2017). Sitio official de PHPUnit. [Online]. Available: http://phpunit.de/

[15] T. Parr, "The Definitive ANTLR Reference," Texas, USA: Pragmatic Bookshelf, 2007, pp. 15–17.

[16] A. Macías, M. D. Delgado, J. Fajardo and D. Larrosa. (2016, enero-junio). Generador de valores de casos de pruebas funcionales. *Lámpsakos.* [Online]. *15,* pp. 51-58. Available: https://dialnet.unirioja.es/descarga/articulo/5403332.pdf

[17] P. B. Fernández, W. Cantillo, M. D. Delgado, et al. (2016, mayo-agosto). Generación de combinaciones de valores de pruebas utilizando metaheurísticas. *Ingeniería Industrial.* [Online]. *36(2),* pp. 200-207. Available: http://www.redalyc.org/pdf/3604/360446197009.pdf

[18] B. F. Jones, H. -H. Sthamer and D. E. Eyres. (1996, septiembre). Automatic structural testing using genetic algorithms. *Software Engineering Journal.* [Online]. *11(5),* pp. 299-306. Available: ieeexplore.ieee.org/iel1/2225/11679/00533215.pdf

[19] E. Díaz, J. Tuya, R. Blanco and J. J. Dolado. (2008, octubre). A tabu search algorithm for structural software testing. *Computers & Operations Research.* [Online]. *35(10),* pp. 3052-3072. Available: giis.uniovi.es/testing/papers/caor-2007-tabustesting.pdf

[20] L. C. Lanzarini, and P. E. Battaiotto. (2010, junio). Dynamic generation of test cases with metaheuristics. *Journal of Computer Science & Technology.* [Online]. *10(2),* pp. 91-96. Available: http://sedici.unlp.edu.ar/bitstream/handle/10915/21338/Documento_completo.pdf?sequence=1