Revista Cubana de Ciencias Informáticas
Vol. 15, No. Especial UCIENCIA II, Septiembre, 2021
ISSN: 2227-1899 | RNPS: 2301
http://rcci.uci.cu
Pág. 268-285

Tipo de artículo: Artículo original
Temática: Pruebas de software
Recibido: 30/06/2021| Aceptado: 01/10/2021

Patrones de implementación para extender la generación de código de pruebas a nuevos lenguajes en GeCodP

Implementation patterns to extend test code generation to new languages in GeCodP

Alejandro Miguel Güemes Esperón [1] https://orcid.org/0000-0001-9704-9449
Martha Dunia Delgado Dapena [1] https://orcid.org/0000-0002-2601-3462
Danay Larrosa Uribazo [1] https://orcid.org/0000-0002-3993-1393

[1] Facultad de Ingeniería Informática, Universidad Tecnológica de La Habana "José Antonio Echeverría", Cujae. Calle 114, No. 11901 entre 119 and 127, Marianao, Código Postal: 19390, La Habana, Cuba. aguemes@tesla.cujae.edu.cu, {marta, dlarosa}@ceis.cujae.edu.cu

*Autor para la correspondencia: (aguemes@tesla.cujae.edu.cu)

RESUMEN

En la actualidad existen herramientas para la generación de código de pruebas unitarias. Sin embargo, se centran en un lenguaje específico. Esto trae como consecuencia, que siempre que se necesite generar pruebas para un nuevo lenguaje se tenga que realizar todo el producto desde cero. En la CUJAE se ha desarrollado la herramienta GeCodP, que integra la generación de código de pruebas en JUnit, PHPUnit y NUnit a partir de código fuente en lenguaje Java. Este trabajo presenta un grupo de patrones de implementación que permiten incorporar el tratamiento de las llamadas a métodos y adicionar nuevos

lenguajes de programación de entrada y nuevos lenguajes de código de prueba, de forma tal que la herramienta pueda ser utilizada en otros entornos de desarrollo de software.

Palabras clave: Pruebas de Software; Pruebas Unitarias; Generación automática de Código de Pruebas; Herramientas de Ejecución de Pruebas.

ABSTRACT

Currently there are tools for generating unit test code. However, they focus on a specific language. This means that whenever you need to generate tests for a new language, you have to make the entire product from scratch. At CUJAE, the GeCodP tool has been developed, which integrates the generation of test code in JUnit, PHPUnit and NUnit from source code in Java. This work presents a group of implementation patterns that allow incorporating the treatment of method calls and adding new input programming languages and new test code languages, in such a way that the tool can be used in other development environments of software.

Keywords: Software Testing; Unit Testing; Automatic Test Code Generation; Test Execution Tools.

Introducción

En la actualidad trabajos científicos consideran importante trabajar la temática de automatización del proceso de pruebas, como (Bregieiro, 2008) (Myers, 2012) (Anand, 2013) (Carvalho, 2014) (Pressman, 2015) (Polo, 2017); y algunos de ellos (Jones, 1996) (Díaz, 2008) (Sekhara, 2012) (Gambi, 2017) (Havrikov, 2017) se centran en la generación de pruebas unitarias. Específicamente en (Varshney, 2013) (Hermadi, 2014) (Khari, 2016) (Han, 2016) (Moadab, 2016) (Soltana, 2019) (Wu, 2019) (Chen, 2019) (Musa, 2019) (Marino, 2019) se encargan de la generación de valores de pruebas mediante la utilización de Inteligencia Artificial para evadir la explosión combinatoria de valores. Existen además diversas herramientas de la familia XUnit para la ejecución automática de pruebas unitarias para diferentes lenguajes

de programación, tales como: JUnit (JUnit, 2018), NUnit (NUnit, 2018) y PHPUnit (Bergmann, 2018). Sin embargo, estas herramientas no cuentan con funcionalidades que contribuyan en el diseño de las pruebas. Esto trae como consecuencia, que los desarrolladores y probadores, en gran medida no cuenten con un buen diseño de los casos de pruebas.

En los últimos años, se han desarrollado herramientas que tienen en cuenta el diseño y ejecución de las pruebas unitarias a partir de código fuente, como (Mirshokraie, 2015) (Prasetya, 2015) (Panichella, 2017) (Sakti, 2017). Para ello, generan código de pruebas que puede ser ejecutado en una herramienta de la familia XUnit. Sin embargo, estas herramientas se centran en un solo lenguaje de programación. Esto trae como consecuencia, que siempre que se necesite generar pruebas para un nuevo lenguaje se tenga que realizar todo el producto desde cero.

En (Delgado, 2016) se ha definido un modelo para la generación automática de pruebas basado en búsquedas, conocido como MTest.search. El modelo consta de Flujos de Trabajo para la ejecución de pruebas tempranas en el entorno de producción, Modelos de Optimización para reducción de casos de pruebas funcionales y unitarias, y Herramientas Automatizadas Integradas (Fernández, 2016) (Macías, 2016) que dan soporte a la ejecución de los flujos de trabajo. Los Modelos de Optimización están compuestos por: un modelo de dominio de entrada, un modelo de pruebas, un modelo de pruebas reducido y un modelo de ejecución. La utilización de este modelo permite realizar extensiones en los modelos de dominio de entrada y de ejecución para la generación de pruebas en diferentes lenguajes sin modificar los modelos de pruebas.

La herramienta GeCodP (Larrosa, 2018) (Pérez, 2019), que da soporte al modelo, permite generar código de pruebas, unitarias y funcionales, en lenguajes de la familia XUnit. Sin embargo, no están definidos los patrones a seguir para la extensión a nuevos lenguajes de entrada y salida en la generación del código de pruebas, así como su inserción en los diferentes entornos productivos.

En este trabajo se exponen un grupo de patrones de implementación que permiten extender la arquitectura de la herramienta GeCodP para adicionar nuevos lenguajes de entrada y de salida. La adopción de estos patrones posibilita generar código de pruebas utilizando toda la generación de casos de pruebas ya implementada en la herramienta teniendo en cuenta las llamadas a métodos incluidos en el código bajo

prueba. De esta forma se pudiera extender la generación de código de pruebas en diferentes entornos de aplicación.

Métodos o Metodología Computacional

Arquitectura de GeCodP

GeCodP es una herramienta para la generación de código de pruebas en diferentes lenguajes y a partir de diferentes dominios de entrada. Se sustenta en el Modelo de Generación automática de Pruebas basado en búsquedas (MTest.search) descrito en (Delgado, 2017). Aunque surge como un componente del modelo se puede utilizar de forma independiente. Esto se hace posible a través de los mecanismos de extensión que ofrece y que permiten personalizarlo para ser utilizado en diferentes entornos de desarrollo tecnológico.

Como soporte al modelo se han desarrollado un conjunto de componentes y herramientas que cubren la generación de casos y código de pruebas en diferentes entornos y los mecanismos de extensión que permiten entenderlos y adaptarlos a las necesidades del entorno productivo.

Las herramientas están organizadas mediante el patrón Arquitectura basada en componentes y está orientado a la experticia del usuario de MTest.search. Se expresa a través de una arquitectura compuesta por tres capas: capa de generación de suit de pruebas, capa de extensión y capa de entorno usuario (ver Fig. 1). La herramienta GeCodP (paquete Generación de código) integra el resto de los componentes que dan soporte al modelo.

La primera capa, contiene el componente GeCaP que integra otros componentes que hacen posible la generación de la suit de pruebas reducida. A esta capa solo deben tener acceso los desarrolladores del equipo de proyecto de MTest.search, de forma tal que se garantice la consistencia y flexibilidad del modelo y sus mecanismos de extensión. La extensión del modelo de reducción basado en búsquedas es privativa de esta capa.

Revista Cubana de Ciencias Informáticas
Vol. 15, No. Especial UCIENCIA II, Septiembre, 2021
ISSN: 2227-1899 | RNPS: 2301
http://rcci.uci.cu
Pág. 268-285

La segunda capa, contiene el componente GeCodP implementado en Java que permite a los usuarios avanzados utilizar los mecanismos de extensión previstos para el modelo de dominio, el modelo de pruebas y el modelo de ejecución.

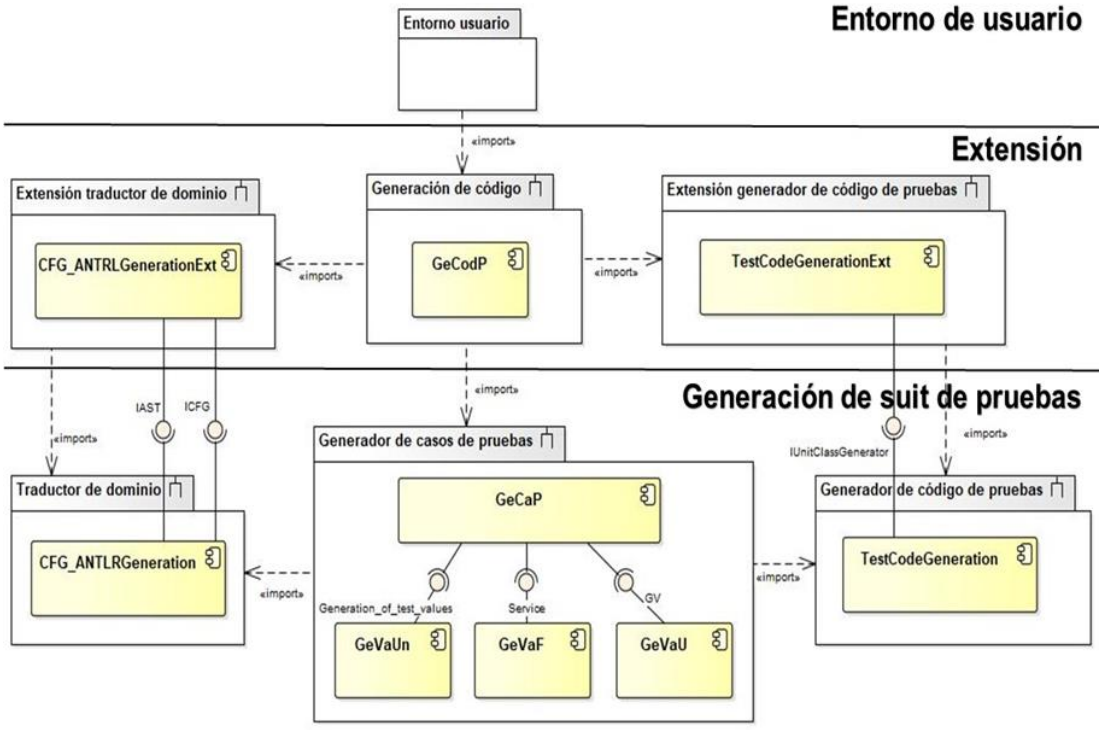

Fig. 1 - Arquitectura basada en componentes de la herramienta GeCodP.

Por último, la tercera capa está concebida para la personalización de MTest.search a los entornos productivos específicos del usuario. En ella se deben implementar aplicaciones clientes que capturan la información del dominio de entrada y que visualizan la suit de pruebas reducida en el lenguaje o artefacto de salida correspondiente.

Para la inserción de estas herramientas en los entornos productivos debe tenerse en cuenta que en el mercado existen diversas herramientas para la ejecución automática de pruebas, una vez que los desarrolladores o probadores han diseñado adecuadamente los casos de prueba. Por tal motivo, estas

herramientas de ejecución de pruebas del mercado, que hayan sido seleccionadas por la empresa, y las herramientas que automatizan la generación de los casos de prueba propias de MTest.search, serán integradas en el entorno productivo.

Patrones de implementación

Se definieron tres patrones de implementación para los mecanismos de extensión propuestos que contribuyen a la implementación de las capas de extensión y de entorno de usuario en la arquitectura definida.

Patrón 1. Extensión a nuevos dominios de entrada para el modelo de código fuente.

Problema: Un usuario avanzado necesita agregar una nueva especificación del código fuente de un lenguaje de programación específico, como dominio de entrada a las herramientas de soporte del modelo MTest.search.

Solución: Para agregar un nuevo dominio de entrada se deben tener en cuenta los pasos siguientes:

1. Crear un nuevo componente que importe el componente CFG_ANTLRGeneration del paquete Traductor de dominio y el componente GeCaP_Classes para el trabajo con el grafo de control de flujo y sus dependencias.
2. Implementar una clase CFGGenerator que implemente la interfaz InterfaceGraph. Para ello, redefinir el método que genera el grafo de control de flujo a partir del código fuente en el nuevo dominio.

En caso de utilizar la herramienta ANTLR para analizar el código del nuevo dominio:

1. Crear un nuevo componente que importe ANTLR (versión 4.1.1).
2. Implementar la gramática del nuevo dominio de entrada, de modo que se genere un AST (Árbol de sintaxis abstracta) en lenguaje Java.

3. Implementar las estructuras gramaticales necesarias para detectar las llamadas a métodos en correspondencia con el nuevo dominio de entrada.
4. Implementar las estructuras gramaticales necesarias para el análisis de las llamadas a métodos (determinar si es una llamada a un atributo parámetro de entrada del método bajo prueba, si es una llamada a un método de una clase externa, si es la creación de un objeto).
5. Importar el componente creado en paso 3, en el componente creado en paso 1.
6. Implementar una clase ASTGenerator que implemente la interfaz InterfaceAST en el componente creado en paso 1. Para ello, redefinir el método que genera el AST a partir del código fuente en el nuevo dominio, mediante la utilización del componente creado en paso 3.
7. Implementar en la clase ASTGenerator los métodos que permitan obtener el listado de las llamadas a métodos existentes en el método bajo prueba.
8. Utilizar el método que genera el AST al principio del método que genera el grafo de control de flujo.

La Figura 2 muestra las clases que necesarias para la definición del patrón, donde la clase CFGGenerator es la encargada de redefinir la generación del grafo de control de flujo; y la clase ASTGenerator redefine la generación del árbol de sintaxis abstracta, en caso necesario.

Consecuencias: La aplicación de este patrón permite generar código de prueba a partir de diferentes lenguajes de entrada, mediante la redefinición de las funcionalidades encargadas de generar el grafo de control de flujo a partir del código fuente en un lenguaje específico.

Revista Cubana de Ciencias Informáticas
Vol. 15, No. Especial UCIENCIA II, Septiembre, 2021
ISSN: 2227-1899 | RNPS: 2301
http://rcci.uci.cu
Pág. 268-285

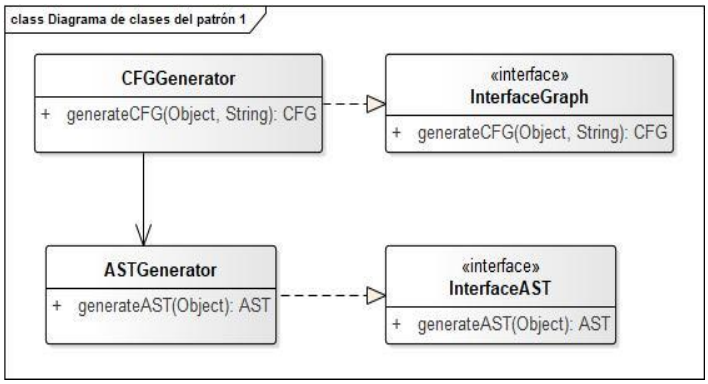

Fig. 2 - Diagrama de clases para el patrón de implementación de un nuevo código fuente de entrada.

Patrón 2. Extensión a nuevos formatos para código de prueba.

Problema: Un usuario avanzado necesita agregar un nuevo formato para código de prueba a las herramientas de soporte del modelo MTest.search.

Solución: Para agregar un nuevo formato para código de prueba se deben tener en cuenta los pasos siguientes:

1. Crear un nuevo componente que importe el componente TestCodeGeneration.
2. Implementar una clase XUnitClassGenerator que implemente la interfaz IXUnitClassGenerator. Para ello, redefinir los métodos para crear la clase de prueba a partir de los casos de pruebas generados (incorporar el tratamiento a los objetos que se devuelven), la clase bajo prueba y el método bajo prueba; y para crear los métodos de pruebas teniendo en cuenta el framework de XUnit a utilizar para la ejecución de las pruebas.
3. Una vez creadas las extensiones descritas previamente, en la capa de extensión, el usuario avanzado debe importar el componente GeCodP y redefinir el método que genera el código de prueba teniendo en cuenta el nuevo dominio de entrada y el nuevo código de salida.

Revista Cubana de Ciencias Informáticas
Vol. 15, No. Especial UCIENCIA II, Septiembre, 2021
ISSN: 2227-1899 | RNPS: 2301
http://rcci.uci.cu
Pág. 268-285

La Figura 3 muestra las clases necesarias para la definición del patrón 2, donde la clase XUnitClassGenerator es la encargada de redefinir la generación del código de prueba en un formato específico.

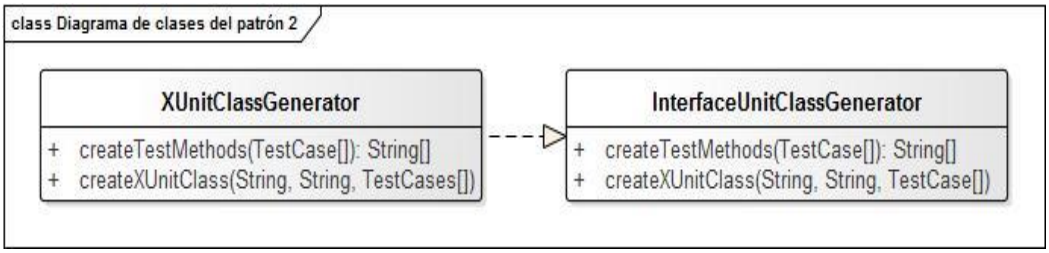

Fig. 3 - Diagrama de clases para el patrón de implementación de un nuevo formato de salida de código de pruebas.

Consecuencias: La aplicación de este patrón permite generar código de prueba para diferentes formatos de salida, mediante la redefinición de las funcionalidades encargadas de generar el código de prueba en un formato determinado.

Patrón 3. Personalización de MTest.search a los entornos productivos específicos del usuario.

Problema: Un usuario necesita capturar la información del dominio de entrada y visualizar la suit de pruebas reducida en el lenguaje o artefacto de salida correspondiente mediante la utilización de los componentes de MTest.search.

Solución: Para personalizar MTest.search a los entornos productivos específicos del usuario se deben tener en cuenta los pasos siguientes:

1. Crear una nueva aplicación cliente que importe el componente GeCodP del paquete Generación de código y sus dependencias.
2. Implementar una clase NombreClase que utilice los métodos necesarios para generar código de pruebas para diferentes lenguajes:

a. Utilizar método para generar el grafo de control de flujo generateCFG (Object methodCode, String path, String language) a partir de la especificación del método bajo prueba, su lenguaje de programación, las especificaciones de las llamadas a métodos dadas por el usuario, y la dirección donde se guardarán los ficheros de intercambio.
b. Utilizar método para generar las combinaciones de valores de prueba para cada camino independiente del grafo de control de flujo generateValuesCombs(LinkedList<VariableXML>varDesc) a partir de la descripción del dominio de las variables asociadas al método bajo prueba.
c. Utilizar método para generar código de prueba, que se corresponde con la suit de pruebas para el método bajo prueba dado) generateTestCode (String language, String classUnderTestPath, LinkedList<String>expectedValues) mediante la especificación del formato de salida del código de prueba, la dirección de la clase que contiene el método bajo prueba y el listado de resultados esperados para cada una de las combinaciones de valores de pruebas generada.

3. Especificar descripción del dominio de las variables que intervienen en el código a probar.
4. Especificar descripción de las llamadas a métodos que intervienen en el código a probar.
5. Especificar valor esperado para las diferentes combinaciones de valores de prueba que satisfacen cada camino independiente.
6. Visualizar el código de prueba generado en el formato especificado.

Consecuencias: La aplicación de este patrón permite la creación de aplicaciones cliente para los diferentes entornos productivos, mediante utilización del componente GeCodP que contiene las diferentes herramientas de MTest.search que permiten generar pruebas unitarias para diferentes lenguajes.

Revista Cubana de Ciencias Informáticas
Vol. 15, No. Especial UCIENCIA II, Septiembre, 2021
ISSN: 2227-1899 | RNPS: 2301
http://rcci.uci.cu
Pág. 268-285

Resultados y discusión

Con el objetivo de evaluar la efectividad de los patrones definidos para generar código de pruebas en diferentes lenguajes, teniendo en cuenta las llamadas a métodos existentes, a partir del modelo ya implementado en GeCodP se diseñaron dos casos de estudio que fueron conducidos por las preguntas siguientes:

1. ¿Es posible extender a nuevos lenguajes de entrada y a nuevos códigos de pruebas utilizando los patrones definidos para extender GeCodP?
2. ¿Es posible generar código de pruebas prueba para diferentes lenguajes de la familia XUnit, a partir de los patrones propuestos?

Para ello, se implementaron aplicaron los patrones e implementaron las extensiones a la herramienta GeCodP para Java y C#. Además, se personalizó el modelo para el entorno de desarrollo Eclipse 4.5.2 o inferior, que permite el trabajo con código Java y C#.

La generación de código de pruebas se realizó en una PC con las siguientes características: Procesador Intel Core i3-4160 de 3.60 GHz y, Memoria RAM de 4 GB. Se utilizó el formato de xUnit 4, en Eclipse 4.5.2.

Pregunta 1. ¿Es posible extender a nuevos lenguajes de entrada y a nuevos códigos de pruebas utilizando los patrones definidos para extender GeCodP?

Para ello, se crearon los componentes necesarios que implementan las interfaces del subsistema Traductor de dominio, teniendo en cuenta el *Patrón 1* descrito en el epígrafe anterior para a extender a nuevos dominios de entrada.

Para la creación del componente con la gramática de ANTLR para un nuevo lenguaje de entrada, ver Fig. 4 a. Para la creación del componente que genera el grafo de control de flujo a partir del código fuente del nuevo dominio de entrada, ver Fig. 4 b. Luego, para extender a nuevos códigos de pruebas, también se tuvo

Revista Cubana de Ciencias Informáticas
Vol. 15, No. Especial UCIENCIA II, Septiembre, 2021
ISSN: 2227-1899 | RNPS: 2301
http://rcci.uci.cu
Pág. 268-285

en cuenta el *Patrón 2* descrito en el epígrafe anterior. Para ello, se crearon los componentes necesarios que implementan las interfaces del subsistema Generador de código de pruebas (ver Fig. 4 c).

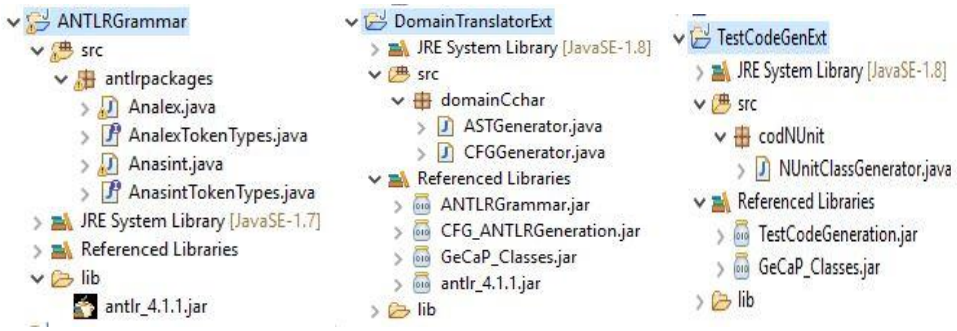

Fig. 4 - Vista de implementación de los componentes a) ANTLRGrammar, b) DomainTranslatorExt y, c) TestCodeGenExt.

Pregunta 2. ¿Es posible generar código de pruebas prueba para diferentes lenguajes de la familia XUnit, a partir de los patrones propuestos?

Para la generación de los casos de pruebas se utilizó el código fuente del algoritmo de clasificación de triángulos definido en (Myers, 2012), en lenguaje de programación Java y C#. Para ello, se crearon los componentes de extensión necesarios, como se mostró en la respuesta de la pregunta anterior. Además, se generó el código de pruebas para su posterior ejecución en JUnit y NUnit, para la ejecución de pruebas unitarias en Java y C# respectivamente.

Para la visualización del trabajo de la herramienta se utilizó el plugin desarrollado para el entorno de desarrollo Eclipse siguiendo el *Patrón 3*. Partiendo del código fuente en ambos lenguajes se obtuvo el código de pruebas que se muestra en Fig. 5 y Fig. 6 respectivamente.

```java
import triangleClassification.TriangleClassification;
import org.junit.Assert;
import org.junit.Test;

public class TestTriangleClassification {

    @Test
    public void test_triangleClassification_CP1() {
        TriangleClassification triangleClassification = new TriangleClassification();
        String expected = "Los lados -1, 0 y 0 no forman un triángulo.";
        Assert.assertEquals(expected, triangleClassification.triangleClassification(-1, 0, 0));
    }

    @Test
    public void test_triangleClassification_CP2() {
        TriangleClassification triangleClassification = new TriangleClassification();
        String expected = "Los lados 17, 0 y 0 no forman un triángulo.";
        Assert.assertEquals(expected, triangleClassification.triangleClassification(17, 0, 0));
    }

    @Test
    public void test_triangleClassification_CP3() {
        TriangleClassification triangleClassification = new TriangleClassification();
        String expected = "Los lados 10, 9 y -1 no forman un triángulo.";
        Assert.assertEquals(expected, triangleClassification.triangleClassification(10, 9, -1));
    }

    @Test
    public void test_triangleClassification_CP4() {
        TriangleClassification triangleClassification = new TriangleClassification();
        String expected = "Los lados 27, 16 y 10 no forman un triángulo.";
        Assert.assertEquals(expected, triangleClassification.triangleClassification(27, 16, 10));
    }

    @Test
    public void test_triangleClassification_CP5() {
```

Fig. 5 - Fragmento de código de prueba a partir de lenguaje Java.

Revista Cubana de Ciencias Informáticas
Vol. 15, No. Especial UCIENCIA II, Septiembre, 2021
ISSN: 2227-1899 | RNPS: 2301
http://rcci.uci.cu
Pág. 268-285

```
using System;
using NUnit.Framework;
using System.Text;

namespace UNitTesting
{
[TestFixture]
public class TriangleClassificationCSTestClass1 {

    [Test]
    public void test_triangleClassification_CP1() {
        TriangleClassificationCS triangleClassification = new TriangleClassificationCS();
        String expected = "Los lados -1, 0 y 0 no forman un triángulo.";
        Assert.assertEquals(expected, triangleClassification.triangleClassification(-1, -1, -1));
    }

    [Test]
    public void test_triangleClassification_CP2() {
        TriangleClassificationCS triangleClassification = new TriangleClassificationCS();
        String expected = "Los lados 17, 0 y 0 no forman un triángulo.";
        Assert.assertEquals(expected, triangleClassification.triangleClassification(1, -1, -1));
    }

    [Test]
    public void test_triangleClassification_CP3() {
        TriangleClassificationCS triangleClassification = new TriangleClassificationCS();
        String expected = "Los lados 10, 9 y -1 no forman un triángulo.";
        Assert.assertEquals(expected, triangleClassification.triangleClassification(21, 20, 0));
    }

    [Test]
    public void test_triangleClassification_CP4() {
        TriangleClassificationCS triangleClassification = new TriangleClassificationCS();
        String expected = "Los lados 27, 16 y 10 no forman un triángulo.";
        Assert.assertEquals(expected, triangleClassification.triangleClassification(27, 16, 10));
    }
```

Fig. 6 - Fragmento de código de prueba a partir de lenguaje C#.

Como se puede observar en las figuras anteriores, la utilización del modelo MTest.search permite la generación de código de pruebas, sin modificar la generación de los casos de pruebas. Solo es necesario utilizar los patrones para diseñar e implementar los componentes correspondientes, tal y como se describió previamente para agregar un nuevo lenguaje, lo cual evita comenzar a automatizar desde cero todo el proceso de generación de pruebas unitarias.

Conclusiones

Como resultado de la definición de los patrones que permiten extender la herramienta GeCodP basada en el modelo MTest.search para implementar la generación en diferentes lenguajes de código de pruebas, se puede extender a nuevos lenguajes de entrada y de ejecución sin modificar la generación de los casos de pruebas y se asegura el tratamiento de las llamadas a métodos involucradas en el código a probar. De esta

forma se facilita el proceso de pruebas durante el desarrollo de productos de software, debido a que permite automatizar el diseño de casos de pruebas y generar código de pruebas unitarias a partir de código fuente integrándolo a diferentes entornos.

Referencias

Myers, G.J., Badgett, T., Sandler, C.: The Art Of Software Testing. John Wiley & Sons, Inc., New Jersey, Usa (2012)

Anand, S., Burke, E.K., Chenc, T.Y., Clark, J., Cohen, M.B., Grieskamp, W., Harman, M., Harrold, M.J., Mcminn, P.: An Orchestrated Survey Of Methodologies For Automated Software Test Case Generation. The Journal Of Systems And Software 86, 1978– 2001 (2013)

Pressman, R.S., Maxim, B.R.: Software Engineering. A Practitioner's Approach. Mcgraw-Hill Education, New York, Usa (2015)

Junit, Sitio Oficial Junit, Www.Junit.Org, Access Date: 2018-07-18, Last Update Date: 2018-04-13

Nunit, Sitio Oficial Nunit, Www.Nunit.Org, Access Date: 2018-07-13, Last Update Date: 2017

Bergmann, S., Sitio Oficial Phpunit, Http://Phpunit.De/, Access Date: 2018-07-18, Last Update Date: 2018-06-05

Bregieiro, J.C.: Search-Based Test Case Generation For Object-Oriented Java Software Using Strongly-Typed Genetic Programming. In: Proceedings Of The 10th Annual Conference Companion On Genetic And Evolutionary Computation, Pp. 1819-1822. Acm, (2008)

Carvalho, G., Falcão, D., Barros, F., Sampaio, A., Alexandremota, Leonardomotta, Blackburn, M.: Nat2testscr: Test Case Generation From Natural Language Requirements Based On Scr Specifications. Science Of Computer Programming 95, 275-297 (2014)

Polo, M., Ruiz, F., Rodríguez-Bobada, R., García, I.: Test Case Generation With Regular Expressions And Combinatorial Techniques. In: 10th Ieee International Conference On Software Testing, Verification And Validation Workshops, Pp. 189-198. Ieee, (2017)

Díaz, E., Tuya, J., Blanco, R., Dolado, J.J.: A Tabu Search Algorithm For Structural Software Testing. Computers & Operations Research 35, 3052–3072 (2008)

Gambi, A., Kappler, S., Lampel, J., Zeller, A.: Cut: Automatic Unit Testing In The Cloud. In: Issta 2017 Proceedings Of The 26th Acm Sigsoft International Symposium On Software Testing And Analysis Pp. 364-367. Acm, (2017)

Havrikov, N., Gambi, A., Zeller, A., Arcuri, A., Galeotti, J.P.: Generating Unit Tests With Structured System Interactions. In: Ast '17 Proceedings Of The 12th International Workshop On Automation Of Software Testing, Pp. 30-33. Ieee, (2017)

Jones, B.F., Sthamer, H.-H., Eyres, D.E.: Automatic Structural Testing Using Genetic Algorithms. Software Engineering Journal 11, 299-306 (1996)

Sekhara, S., Hari, M.L., Kiran, U., Ch, S., Ranjan, P.: Automated Generation Of Independent Paths And Test Suite Optimization Using Artificial Bee Colony. Procedia Engineering 30, 191-200 (2012)

Han, X., Lei, H., Wang, Y. S.: Multiple Paths Test Data Generation Based On Particle Swarm Optimisation. Iet Software 11, 41-47 (2016)

Hermadi, I., Lokan, C., Sarker, R.: Dynamic Stopping Criteria For Search-Based Test Data Generation For Path Testing. Information And Software Technology 56, 395-407 (2014)

Khari, M., Kumar, P.: A Novel Approach For Software Test Data Generation Using Cuckoo Algorithm. In: Ictcs '16 Proceedings Of The Second International Conference On Information And Communication Technology For Competitive Strategies Acm, (2016)

Moadab, S., Rashidi, H.: Automatic Path-Oriented Test Data Generation By Boundary Hypercuboids. Journal Of King Saud University - Computer And Information Sciences 28, 82-97 (2016)

Varshney, S., Mehrotra, M.: Search Based Software Test Data Generation For Structural Testing: A Perspective. Acm Sigsoft Software Engineering Notes 38, 1-6 (2013)

Soltana, G., Et Al.: Practical Model-Driven Data Generation For System Testing, Supported By The European Research Council Under The European Union's Horizon 2020 Research And Innovation Program (Grant Agreement Number 694277), Cornell University (2019)

Wu, H., Nie, Ch., Petke, J., Jia, Y., Harman, M.: A Survey Of Constrained Combinatorial Testing, Cornell University (2019)

Chen Et Al.: A Taxonomic Review Of Adaptive Random Testing: Current Status, Classifications, And Issues, Supported By National Natural Science Foundation Of China (Nsfc Grant Numbers: U1836116,

61462030, And 61762040), The Project Of Jiangsu Provincial Six Talent Peaks (Grant Number: Xydxxjs-016), And The Graduate Research Innovation Project Of Jiangsu Province (Grant Number: Kylx16_0900), January 2019.

Musa, J. Et Al.: An Analysis On The Applicability Of Metaheuristic Searching Techniques For Automated Test Data Generation In Automatic Programming Assessment, Baghdad Science Journal, Vol.16 (Special Issue), 515-523 (2019)

Marino, J. And V. Alexandre: A Systematic Mapping Addressing Hyper-Heuristics Within Search-Based Software Testing, Information And Software Technology, No. 114, Pp 176–189, (2019)

Mirshokraie, S., Mesbah, A., Pattabiraman, K.: Jseft: Automated Javascript Unit Test Generation. In: 2015 Ieee 8th International Conference On Software Testing, Verification And Validation (Icst), Pp. 1-10. Ieee, (2015)

Prasetya, I.S.W.B.: T3: Benchmarking At Third Unit Testing Tool Contest. In: 2015 Ieee/Acm 8th International Workshop On Search-Based Software Testing, Pp. 44-47. Ieee, (2015)

Panichella, A., Molina, U.R.: Java Unit Testing Tool Competition: Fifth Round. In: Sbst '17 Proceedings Of The 10th International Workshop On Search-Based Software Testing Pp. 32-38. Ieee, (2017)

Sakti, A., Pesant, G., Guéhéneuc, Y.-G.: Jtexpert At The Sbst 2017 Tool Competition. In: Sbst '17 Proceedings Of The 10th International Workshop On Search-Based Software Testing Pp. 43-46. Ieee, (2017)

Delgado, M.D., Macias, A., Larrosa, D., Verona, S., Fernández, P.B.: Model For Automatic Generation Of Search-Based Early Test. Computación Y Sistemas 21, 503-513 (2017)

Fernández, P.B., Cantillo, W., Delgado, M.D., Rosete, A., Yáñez, C.: Generación De Combinaciones De Valores De Pruebas Utilizando Metaheurística. Ingeniería Industrial 37, 200-207 (2016)

Macías, A., Delgado, M.D., Fajardo, J., Larrosa, D.: Generador De Valores De Casos De Pruebas Funcionales. Lámpsakos 51-58 (2016)

Pérez, Z.; Rojas, D.; Delgado, M.: "Generador de valores interesantes para casos de pruebas unitarias", Ingeniería Industrial 40 (2019)

Larrosa, D.; Fernández, P.; Delgado, M.: "GeCaP: Unit Testing Case Generation from Java Source Code", POLIBITS 57 (2018)

Conflicto de interés

Los autores autorizan la distribución y uso de su artículo.

Contribuciones de los autores

1. Conceptualización: Alejandro Miguel Güemes Esperón y Martha Dunia Delgado Dapena.
2. Curación de datos: Alejandro Miguel Güemes Esperón.
3. Análisis formal: Danay Larrosa Uribazo.
4. Adquisición de fondos: Martha Dunia Delgado Dapena.
5. Investigación: Alejandro Miguel Güemes Esperón.
6. Metodología: Danay Larrosa Uribazo.
7. Administración del proyecto: Martha Dunia Delgado Dapena.
8. Recursos: Martha Dunia Delgado Dapena
9. Software: Alejandro Miguel Güemes Esperón.
10. Supervisión: Martha Dunia Delgado Dapena.
11. Validación: Danay Larrosa Uribazo.
12. Visualización: Alejandro Miguel Güemes Esperón.
13. Redacción – borrador original: Alejandro Miguel Güemes Esperón.
14. Redacción – revisión y edición: Martha Dunia Delgado Dapena y Alejandro Miguel Güemes Esperón

Financiación

Universidad Tecnológica de La Habana “José Antonio Echeverría”, Cujae.