Generación automática de casos de prueba de JUnit

María del Mar Jiménez, Macario Polo, Mario Piattini

Grupo Alarcos
Escuela Superior de Informática
Paseo de la Universidad, 4
13071 Ciudad Real (España)
{Mar.Jimenez, Macario.Polo, Mario.Piattini}@uclm.es

http://alarcos.inf-cr.uclm.es

Resumen. Se presenta una técnica para la generación automática de casos de prueba de JUnit para Sistemas Orientados a Objetos (*SOO*). A partir del conjunto de clases del Sistema Orientado a Objetos, se aplican una serie de algoritmos para obtener secuencias de prueba para cada una de las clases que forman el sistema, formadas dichas secuencias por un constructor y un conjunto de métodos de la clase. Se utilizan los diagramas de estado de las clases para conseguir reducir el conjunto de secuencias de prueba obtenido. Se asignan valores a los parámetros de cada uno de los métodos y al constructor que forman cada una de las secuencias, obteniendo de este modo los casos de prueba. Las clases y sus operaciones pueden anotarse con restricciones que usaremos para generar un formato válido de casos de prueba para JUnit.

1 Introducción

La fase de pruebas del software es una de las partes más importantes del proceso de desarrollo de software. Probar una aplicación involucra la generación de casos de pruebas, la ejecución de la aplicación contra estos casos de prueba y la observación del comportamiento de la aplicación para determinar su corrección. Cuando una aplicación se comporta de manera incorrecta necesita ser depurada para determinar la causa. La depuración nos conduce a la identificación y la corrección de errores [9].

Las pruebas software nos permiten comprobar cómo una serie de datos de entrada producen una serie de datos de salida en un determinado trozo de código [14]. La generación manual de los casos de prueba puede consumir bastante tiempo, y la fase de pruebas del software ha llegado a ser una de las más caras del ciclo de vida. La realización de las pruebas cuesta entre un 40 y un 75 por ciento del tiempo de las fases de desarrollo y mantenimiento del ciclo de vida software [3] [10]. A pesar de esto, las pruebas software han sido la técnica más usada para asegurar la calidad del software [1].

En este trabajo se estudian únicamente las pruebas de caja negra, en las que el código fuente no está disponible o no interesa y para ello elegiremos como unidad de prueba las clases que componen los sistemas orientados a objetos. La noción de clase

como unidad de prueba ha sido utilizada en muchos proyectos de sistemas orientados a objetos [15].

Las pruebas de unidad se centran en la verificación de pequeños bloques del diseño del software [19]. Las pruebas unitarias nos sirven para asegurar que partes individuales de un sistema complejo trabajan correctamente de manera aislada, antes de su integración con otras partes del sistema. A pesar de que las pruebas unitarias son consideradas una excelente idea no son muy usadas y no se practican formalmente. Por ello, nosotros intentaremos automatizar la generación y ejecución de dichas pruebas partiendo de un metamodelo del SOO obtenido de forma automática considerado este tipo de pruebas debido a la gran importancia que tienen dentro de la Programación Extrema (*XP*) [7] y al extendido uso de entornos de pruebas de caja negra como es el caso de JUnit [12].

Este artículo está organizado de la siguiente manera: en la sección 2 describimos la estructura del metamodelo usado, en la sección 3 examinamos las técnicas utilizadas para la generación de casos de prueba y algunos valores problemáticos para estos casos de prueba, y describimos el proceso de ejecución de estos casos de prueba haciendo uso de restricciones y del marco de pruebas JUnit. Por último obtenemos algunas conclusiones a cerca del método presentado y proponemos distintas líneas para trabajos futuros.

2 Descripción formal de las clases de un sistema orientado a objetos

Los sistemas orientados a objetos se pueden representar con diagramas estructurales (de clases, objetos, componentes, paquetes,...) o con diagramas de comportamiento (de secuencia, colaboración, interacción, estados, actividad,...). La mayoría de las técnicas [5] [6] [11] se basan en los diagramas de comportamiento para generar casos de prueba, en particular, en los diagramas de estado y para ello es necesario definir primero el comportamiento de los objetos. Nuestro método se basará en una especificación formal obtenida automáticamente a partir del diagrama de clases del sistema o en la especificación individual de cada una de las clases, módulos o componentes que forman el sistema, por lo que no será necesario definir el dinamismo del sistema para generar los casos de prueba.

El metamodelo propuesto especifica la representación de cada una de las clases del sistema usando una representación formal basada en una descripción matemática de algunos de los elementos existentes en un sistema orientado a objetos. Para ello hemos considerado las ideas propuestas por [13] y hemos utilizado la notación algebraica descrita posteriormente por los mismos autores en [20].

Teniendo en cuenta estas consideraciones, una posible descripción textual de un sistema orientado a objetos es [14]: "*conjunto de clases y de relaciones entre ellas: asociaciones, agregaciones, dependencias y relaciones de herencia*". Además, cada clase del sistema puede ser anotada con invariantes descritas en algún lenguaje ad hoc, como OCL, y cada una de sus operaciones puede también serlo con precondicio-

nes y postcondiciones. De este modo, todas estas estructuras pueden representarse como se muestra en la siguiente figura:

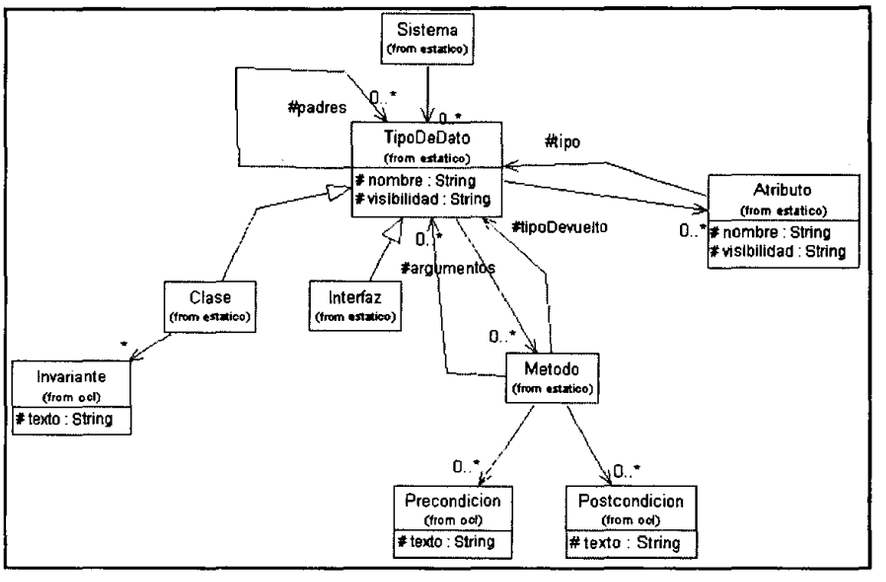

Fig. 1. Diagrama de clases que muestra parte del metamodelo utilizado para representar SOO

Incluso, con cada clase puede acompañarse un diagrama de transición de estados que represente las secuencias válidas de llamadas a métodos de la clase. Esto implica la construcción de otro metamodelo para representar este tipo de diagramas, que no ofrecemos por razones de brevedad, pero en el que se incluye la descripción completa de los estados y de las transiciones entre éstos. En este nuevo metamodelo, la clase *Clase* mantendría una relación de asociación con una instancia de clase *DTE*.

Obviamente, cualquier SOO (o, al menos, las porciones de él que nos interesan para nuestro propósito) puede ser representado mediante este metamodelo.

3 Generación de clases Test para JUnit

En *JUnit,* dada una clase *X* que se va a probar, se procede por lo general construyendo una clase *TestX* que incluye métodos cuyo nombre es del estilo *testFuncionalidad()*, en cuyo interior se crean dos objetos: el primero se corresponde con el resultado esperado y es construido normalmente de forma manual; el segundo objeto representa el resultado obtenido y se construye utilizando los servicios suministrados por la clase *X*. A continuación, se comparan ambos objetos mediante una serie de funciones *assert* suministradas por el *framework*.

Uno de los métodos *assert* más utilizados es el *assertEquals(Object expected, Object obtained)*, que comprueba que el objeto esperado y el obtenido sean iguales. No

obstante, *JUnit* también permite utilizar métodos como *assertTrue(boolean)*, que comprueba la veracidad de la condición pasada como parámetro. Con nuestro método, generamos clases *TestX* para una clase *X* si ésta incluye restricciones que representen invariantes o restricciones sobre sus operaciones, que son comprobadas con *assertTrue*.

Así, dadas las anotaciones mostradas en la parte izquierda de la tabla siguiente, se podrían aprovechar los casos de prueba obtenidos para generar la clase *TestCuenta*. En la parte derecha de la misma tabla se muestra el código generado por un caso de prueba que hace una llamada al constructor de la clase y al método *ingresar(importe)*.

context **Cuenta** inv :
saldo>=0.0
 	
context Cuenta ::
ingresar(importe:Double
)
pre :
importe>=0.0
 	
context Cuenta ::
retirar(importe:Double)
pre :
importe>=0.0
	public class **TestCuenta** extends TestCase {
	...
	public void **testSecuencia_5_1**() {
	Cuenta c=new Cuenta(new String());
	assertTrue(c.saldo>=0.0); // Prueba de la invariante
	Double importe=new Double(-1.0);
	assertTrue(importe>=0.0); // Prueba de la precondición
	c.ingresar(importe);
	assertTrue(c.saldo>=0);
	}

Tab. 1. Algunas restricciones para la clase *Cuenta* y un fragmento de la clase *TestCuenta*

El número de casos de prueba generado para una secuencia depende de la longitud de ésta, del número total de parámetros contenidos en su constructor y métodos, y del número de valores *CE* del tipo de cada parámetro, pudiéndose representar por la expresión:

$$\prod_{\forall p \in s.Constructor.parámetros} |CE(p)| \cdot \prod_{\forall m \in s.Métodos} \left(\prod_{\forall p \in m.parámetros} |CE(p)| \right)$$

Ejemplo

Un famoso ejercicio de *testing*, que nos sirve para hacernos una idea de los costes reales de un proceso de pruebas exhaustivo, es el dado por Glenford Myers en *The Art of Software Testing* [15]: se pide que se construyan casos de prueba para probar el funcionamiento correcto de un programa que admite tres números enteros que se interpretan como las longitudes de los tres lados de un triángulo; el programa debe decir si el triángulo es equilátero, isósceles o escaleno. La solución puede parecer trivial, pero muy probablemente se haya olvidado de construir alguno de los siguientes casos de prueba:

1. Triángulos escaleno/ equilátero/ isósceles válidos. 2. Tres casos de prueba que representen triángulos isósceles válidos, de modo que se han considerado las tres permutaciones de dos lados iguales. 3. Un lado es cero. 4. Un lado es negativo	5. Tres números enteros positivos tal que la suma de dos lados es igual al tercero, y sus permutaciones. 6. Suma de dos de los números menores que el tercero 7. Tres casos del tipo anterior tal que se han intentado todas las permutaciones.

Tab. 2. Algunos casos de prueba para el problema de los triángulos

Para una clase *Triángulo* como la descrita anteriormente la masiva generación de casos de prueba ofrecida por nuestro algoritmo permite comprobar que, con un conjunto reducido de valores de conjetura de error (MININT, -aleatorio(), -1, 0, +1, +aleatorio(), MAXINT), se cubren prácticamente todos los casos enumerados en la tabla (se garantizan todos excepto el supuesto 5).

```
public class TestTriangulo extends
TestCase {
  public void testSecuencia_X_1_1() {
      Triangulo t=new Triangulo();
        t.setX (-1);
        t.setY (0);
        t.setZ (+1); ... }
  public void testSecuencia_X_1_2() {
      Triangulo t=new Triangulo();
        t.setX (+1);
        t.setY (+1);
        t.setZ (+1); ... }
  public void testSecuencia_X_1_3() {
      Triangulo t=new Triangulo();
        t.setX (+1);
        t.setY (0);
        t.setZ (+1); ... }
  public void testSecuencia_X_2_1() {
      Triangulo t=new Triangulo();
        t.setX (+1);
        t.setY (0);
        t.setZ (+1); ... }
  public void testSecuencia_X_2_2() {
      Triangulo t=new Triangulo();
        t.setX (+1);
        t.setY (+1);
        t.setZ (0); ... }
  public void testSecuencia_X_2_3() {
      Triangulo t=new Triangulo();
        t.setX (0);
        t.setY (+1);
        t.setZ (+1); ...}
```

```
  public void testSecuencia_X_5_1() {
      Triangulo t=new Triangulo();
        t.setX (+aleatorio); // Vale 2
        t.setY (+1);
        t.setZ (+1); ... }
  public void testSecuencia_X_5_2() {
      Triangulo t=new Triangulo();
        t.setX (+1);
        t.setY (+1);
        t.setZ (+aleatorio); // Vale 2
      ... }
  public void testSecuencia_X_5_3() {
      Triangulo t=new Triangulo();
        t.setX (+1);
        t.setY (+aleatorio); //Vale 2
        t.setZ (+1); ... }
  public void testSecuencia_X_6_1() {
      Triangulo t=new Triangulo();
        t.setX (MAXINT);
        t.setY (+1);
        t.setZ (+1); ... }
  public void testSecuencia_X_7_1() {
      Triangulo t=new Triangulo();
        t.setX (MAXINT);
        t.setY (+1);
        t.setZ (0); ... }
  public void testSecuencia_X_7_2() {
      Triangulo t=new Triangulo();
        t.setX (+1);
        t.setY (0);
        t.setZ (MAXINT); ... }
```

```
public void testSecuencia_X_3_1() {
        Triangulo t=new Triangulo();
            t.setX (+1);
            t.setY (0);
            t.setZ (+1); ... }
    public void testSecuencia_X_4_1() {
        Triangulo t=new Triangulo();
            t.setX (+1);
            t.setY (-1);
            t.setZ (0); ... }
```

```
public void testSecuencia_X_7_3() {
        Triangulo t=new Triangulo();
            t.setX (+1);
            t.setY (MAXINT);
            t.setZ (0); ... }
}
```

Tab 3. Casos de prueba de JUnit para el problema de los triángulos

Con nuestro método cada uno de los casos quedaría probado por la combinación de valores que están disponibles para la clase *Integer*. La última observación que hace Myers es, sin embargo, muy difícil de lograr: "especificar el resultado esperado para cada caso de prueba"; sin embargo, la anotación de la clase y de sus restricciones permite comprobar con cada caso la consistencia del estado del objeto antes y después de la ejecución de cada operación.

Por último, para poder ejecutar estos test cases hacemos una llamada a la clase *TestTriangulo*:

```
public static void main (String [] args){
        junit.swingui.TestRunner.run(TestTriangulo.class);
}
```

4 Conclusiones y trabajos futuros

El trabajo ha presentado un método para generar y ejecutar de manera totalmente automática casos de prueba de JUnit sobre sistemas orientados a objeto.

El sistema genera una cantidad de casos de prueba muy grande, que no obstante no garantiza el cubrimiento del código que se está probando (si bien la aplicación práctica nos indica que es bastante alta). Con el fin de conocer el nivel de cubrimiento conseguido por los casos de prueba generados estamos desarrollando un conjunto de herramientas que utilizan mutación [17] para comprobar la calidad de los casos de prueba.

Agradecimientos

Este trabajo forma parte del proyecto DOLMEN parcialmente financiado por la Subdirección General de Proyectos de Investigación, Ministerio de Ciencia y Tecnología (TIC 2000-1676-C06-06).

Referencias

1. Bashir, I., Goel, A.L. Testing Object-Oriented Software. Springer, 1999.
2. Beizer B. Black-Box Testing: Techniques for Functional Testing of Software and Systems. Wiley: New York, 1995.
3. Beizer, B., Software Testing Techniques, New York, New York: Van Nostrand Reinhold, 1983.
4. Binder, R. Automated Java Testing. Object Magazine, Jul. 1997.
5. Booch, G., Rumbaugh, J., Jacobson, I. The Unified Modeling Language User Guide. Object Technology Series. Adisson Wesley Longman, Inc., October 1998.
6. Chevalley, P., Thevenod-Fosse, P. Automated Generation of Statical Test Cases from UML State Diagrams. 25th Annual International Computer Software and Applications Conference. COMPSAC 2001, 2001, Pages 205-214.
7. Crispin, L., House, T. Testing Extreme Programing, Adison-Wesley, 2002; ISBN 0-321-11355-1.
8. Doong, R.K. y Frankl, P.G. (1994). The ASTOOT approach to testing object-oriented programs. ACM Transactions on Software Engineering and Methodology, 3(2):101-130.
9. Fewster, M., Graham, D. Software Test Automation. Addison-Wesley, ACM Press Books, 1999.
10. Ghiassi, M., Woldman, K.I.S. "Dual Programming Approach to Software Testing," Software Quality Journal, 3:45 58, 1994.
11. Harel, D. Statecharts: a Visual Formalism for Complex Systems. Science of Computer Programming, 8:231-274, 1987.
12. JUnit web site. http://www.junit.org.
13. Krikhaar, R.L. Software Architecture Reconstruction. PhD thesis, University of Amsterdam, 1999.
14. Michael, CC., McGraw, G., Schatz MA. (2001). Generating Software Test Data by Evolution. IEEE Transactions on Software Engineering, 27(12), 1085-1109.
15. Murphy, G.C., Townsend, P., Wong, P.S. Experiences with Cluster and Class Testing. Communications of the ACM, 37(9):39-47, Septiembre, 1994.
16. Myers, G.J. (1979). The Art of Software Testing. John Wiley and Sons, New York.
17. Offutt J. (1995). *Practical Mutation Testing*. Twelfth International Conference on Testing Computer Software, pages 99-109, Washington, DC.
18. Overbeck, J. Testing Object-Oriented Software: State of the Art and Research Directions. In Proceedings of the Firs European International Conference of Software Testing, Analysis and Review, London/UK, Oct. 1993.
19. Pressman, R.S., Software Engineering A Practitioner's ApproachK. McGraw-Hill, 1997.
20. Reinder, J., Loe, M.G. Feijs, André Glas, René L. Krikhaar, Thijs Winter (2000). Maintaining a legacy: towards support at the architectural level. Journal of Software Maintenance, 12(3): 143-170.