AUTOMATIZACIÓN DEL PROCESO DE PRUEBAS UNITARIAS

Macario Polo Usaola y Mario Piattini Velthuis

Grupo Alarcos - Departamento de Tecnologías y Sistemas de Información -
Escuela Superior de Informática
Universidad de Castilla-La Mancha
Paseo de la Universidad, 4; 13071-Ciudad Real
e-mail: Macario.Polo@uclm.es, Mario.Piattini@uclm.es

Palabras clave: Testing, Pruebas unitarias, Automatización de las pruebas

Resumen. *Este artículo describe algunos de los trabajos realizados en la línea de automatización del proceso de pruebas, que se enmarcan en el proyecto Mantenimiento Ágil del Software (TIC2003-02737-C02-02). Principalmente, se ha trabajado en la integración de diferentes técnicas ya existentes en la literatura, pero que no habían sido utilizadas conjuntamente para automatizar el proceso.*

1. INTRODUCCIÓN

De acuerdo con Meudec [1], la investigación en la automatización de las pruebas software destina sus esfuerzos, fundamentalmente, a las tres siguientes líneas de trabajo: (1) tareas administrativas (registro de especificaciones de prueba, generación de informes); (2) tareas mecánicas (ejecución y monitorización, utilidades de captura y reejecución); y (3) tareas de generación de casos de prueba.

Respecto de la tercera línea citada (generación de casos), Ball et al. [2] indican así mismo tres enfoques principales: (1) dado un fragmento de código fuente, generar automáticamente datos de entrada para lograr un criterio determinado de cobertura de sentencias, ramas o caminos; (2) dado el conocimiento del ingeniero de pruebas del código fuente y de su comportamiento esperado, desarrollar algoritmos que generen automáticamente los datos de entrada y que comprueben las salidas; (3) dada una especificación formal, generar automáticamente casos de prueba a partir de ella.

Si bien la especificaciones formales aumentan la capacidad de automatizar el proceso de pruebas [2], es también cierto que son difíciles de escribir y mantener [3], difíciles de aplicar en la práctica, y son raramente utilizadas en entornos industriales [2]. Así, lo más habitual es que el programador escriba los casos para probar una característica de un nuevo sistema después de que el código que implementa dicha característica haya sido escrito [4], siendo de este modo el código la fuente más habitual para escribir los casos de prueba [5].

En los últimos años se han desarrollado una serie de frameworks (los conocidos entornos X-Unit, como JUnit o NUnit) para automatizar las pruebas unitarias, y hay multitud de entornos de desarrollo que ya incorporan plugins y componentes que permiten su integración.

Algunos plugins y herramientas utilizan X-Unit para generar automáticamente casos de prueba: normalmente, el usuario elige un método de la CUT (por ejemplo, el método *deposit* en la clase *Account)* y la herramienta genera un método cuya implementación debe ser suministrada por el usuario (así, crearía un método *testDeposit,* que el usuario debería completar con la llamada de-

seada, asignando los valores adecuados a los parámetros y escribiendo el oráculo).
Así pues, en lo tocante a pruebas unitarias, la segunda de las líneas apuntadas por Meudec (y que citábamos al principio de este artículo) se encuentra suficientemente automatizada con entornos X-Unit; pero en las otras dos queda todavía mucho trabajo por hacer.
Precisamente a estos dos tipos de tareas se han dedicado los esfuerzos de investigación en la línea de trabajo de automatización de pruebas del proyecto (TIC2003-02737-C02-02), que finaliza en noviembre de 2006. A continuación se presentan los principales resultados obtenidos.

2. GENERACIÓN AUTOMÁTICA DE CASOS DE PRUEBA

Kirani y Tsai propusieron la "especificación de secuencias de métodos" como una forma de documentar "el orden en que los métodos de una clase pueden ser invocados por otras clases clientes" [6]. Una secuencia de métodos se puede describir utilizando una expresión regular, cuyo alfabeto estaría compuesto por el conjunto de operaciones públicas de la CUT. La idea, desde luego, resulta sumamente atractiva para describir casos de prueba para una clase.
Así, suponiendo que una clase *Account* (que representa una cuenta bancaria) disponga de un constructor y las operaciones *deposit, withdraw* y *transfer*, una posible expresión regular para describir casos de prueba podría ser la siguiente, en la que se obliga a que los casos de prueba comiencen realizando un ingreso (operación *deposit)* y, posteriormente, una secuencia compuesta por cualesquiera de las tres operaciones indicadas en cualquier orden:

*Account·deposit·[deposit|withdraw|transfer]**

Los lenguajes Java y los de la plataforma Microsoft .NET permiten la utilización de introspección (reflexión o *reflection*), lo cual posibilita cargar en tiempo de ejecución una clase que era desconocida en tiempo de compilación, para luego extraer sus características, crear instancias de esa clase, invocar a sus métodos, etc. Mediante introspección, por tanto, es posible cargar el *bytecode* o el ensamblado correspondientes a la CUT, mostrar al usuario (que sería el ingeniero de pruebas) el conjunto de operaciones públicas, darle la opción de escribir una expresión regular y, a partir de ella, generar casos de prueba.
Con el fin mencionado se han desarrollado dos herramientas, *testooj* y *NTesting*, capaces de bregar respectivamente con clases Java (ficheros *.class,* que pueden estar empaquetados en *.jar* o *.zip*) y con ensamblados de .NET (bibliotecas *dll,* ejecutables *exe*, etc.). El ingeniero de pruebas escribe la expresión regular y, a partir de ella, se genera un conjunto de "plantillas de casos de prueba", que son utilizadas posteriormente para generar los casos. En la Figura 1 se muestra la ventana principal de *tezstooj:* una vez cargada la CUT, en el árbol de la izquierda se muestra el conjunto de operaciones públicas de la clase; el usuario, entonces, escribe la expresión regular en la zona habilitada a tal efecto utilizando letras (la *A* para la primera operación, la *B* para la segunda, etc.); el usuario puede seleccionar la longitud máxima de las secuencias (en la figura, se ha elegido 6), que se corresponde con el número máximo de operaciones de los casos de prueba que se generarán; por último, el árbol del lado derecho se carga con las plantillas de prueba.

3.1. Asignación de valores y descripción del oráculo

Las plantillas de prueba generadas a partir de la expresión regular deben combinarse con valores, que serán asignados a los diferentes parámetros de las operaciones involucradas en cada plantilla. Así, por ejemplo, la plantilla *Account().deposit(double, String)* no representa nada si no va acom-

pañada de valores que puedan ser pasados a los dos parámetros de *deposit*.
A los valores que se utilizan en los casos de prueba se los conoce como "valores interesantes" [7]. Desde luego, interesa que estos valores tengan la mayor probabilidad de encontrar errores en la CUT, y para ello se utilizan técnicas muy conocidas, como la partición en clases de equivalencia, valores límite, etc.

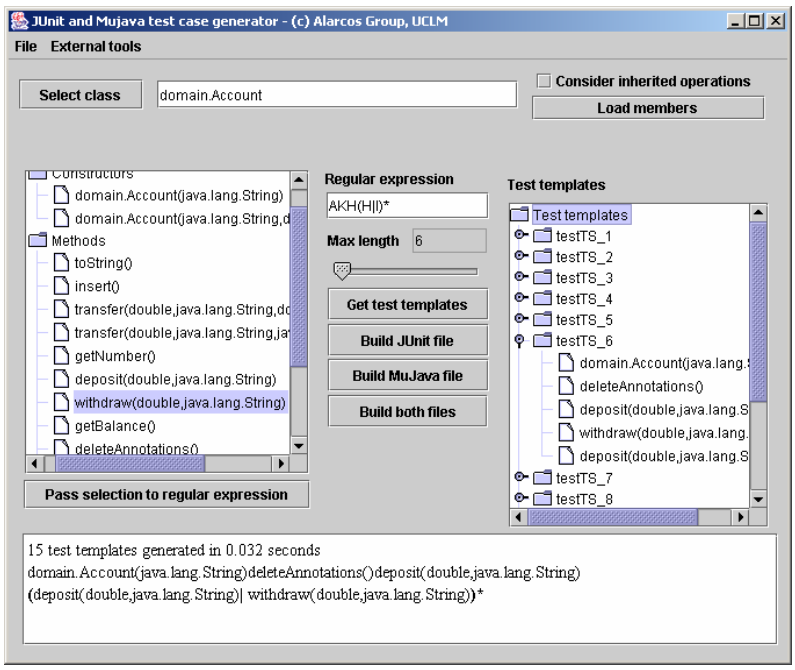

Figura 1. Pantalla principal de testooj, la herramienta de generación de casos para Java

De este modo, suponiendo que se asignen al primer parámetro de *deposit* los valores 0 y 100, y la cadena "Ingreso en efectivo" al segundo parámetro, de la plantilla mencionada más arriba podrían obtenerse dos casos de prueba de este estilo:

Account obtained=Account()
*obtained.deposit(**0, "Ingreso en efectivo"**)*

Account obtained=Account()
*obtained.deposit(**100, "Ingreso en efectivo"**)*

También resultaría sumamente interesante poder describir el oráculo para ambos casos de prueba de una forma genérica. Tanto en *testooj* como en *NTesting* se puede describir utilizando un pequeño conjunto de "palabras reservadas":

- Con *obtained* se hace referencia a la instancia de la CUT que está manipulando el caso de prueba.
- Con *result* se hace referencia al resultado devuelto por las operaciones involucradas en el caso de prueba.
- Con *arg1, arg2,* etc. se hace referencia a los distintos argumentos pasados como parámetros.

La Figura 2 muestra la ventana de *testooj* utilizada para asignar valores interesantes a parámetros y para describir el oráculo. Como se observa, para cada operación puede escribirse "precódigo" y "postcódigo", que es código que se inserta, respectivamente, antes y después de la operación que se está anotando. En el ejemplo, antes de llamar a *deposit* en cada caso de prueba, se declarará una

variable llamada *balancePre* a la que se asigna el resultado de ejecutar la operación *getBalance* sobre el objeto *obtained* (nótese el texto mostrado en el precódigo)*;* tras la llamada a la operación, se insertará una instrucción que hace uso de la función *assertTrue* de JUnit para comprobar que el saldo se incrementa adecuadamente (obsérvese que se hace referencia al primer argumento con la palabra *arg1)*, y que constituye el oráculo para esta operación.

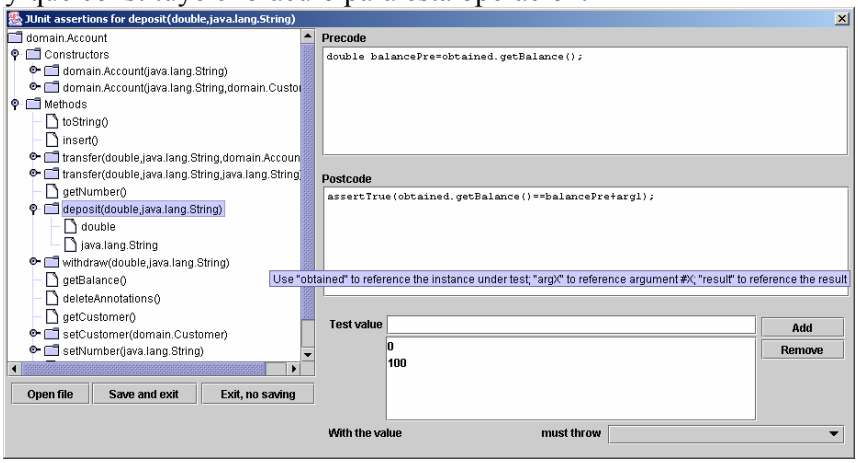

Figura 2. Ventana para la asignación de valores interesantes y descripción del oráculo

Una vez descritas las plantillas, asignados los valores interesantes y descritos el pre y el postcódigo, se está en disposición de generar los casos de prueba en diversos formatos. Centrándonos en el formato de JUnit, *testooj* incluye una ventana, en la que se muestran todos los casos de prueba generados.

3.2. Medida de la calidad de los casos

Incluso aunque se genere un número muy grande de casos de prueba, su calidad (es decir, su capacidad para detectar fallos en la CUT) no está garantizada si el ingeniero de pruebas no conoce el grado de cobertura que se haya alcanzado (Erdogmus et al. [4] han presentado un estudio en el que se muestra que no hay correlación estadísticamente significativa entre el número de casos de prueba y la calidad del programa). Por ello es necesario utilizar algún criterio de cobertura para conocer cuánto código de la CUT es ejercitado por los casos de prueba.

Para código fuente, Cornett [8] presenta algunos criterios de cobertura comunes, como número de sentencias, condiciones, decisiones, condiciones/decisiones, caminos, flujo de datos, etc. La mutación es otro criterio de cobertura cada vez más utilizado para comprobar la calidad de una batería de casos, como para validar estrategias de generación de casos de prueba [9-12].

testooj (no así NTesting) incluye la posibilidad de generar casos de prueba para MuJava [13], un sistema de pruebas mediante mutación para código Java. El formato de los casos de prueba para MuJava es muy similar al de los casos de JUnit: los primeros devuelven una cadena que representa el estado de la instancia de la CUT y no tienen oráculo y no tienen oráculo (puesto que MuJava no se utiliza para encontrar fallos en la CUT); en JUnit, los casos de prueba son métodos *void* que sí disponen de oráculo.

3.3. Descripción del proceso de pruebas unitarias

A la vista de los resultados presentados en la sección anterior, la combinación de ambos tipos de técnicas (pruebas de caja negra mediante JUnit, y de caja blanca mediante MuJava) permite la definición de un proceso de pruebas unitarias riguroso y muy completo (Figura 3).

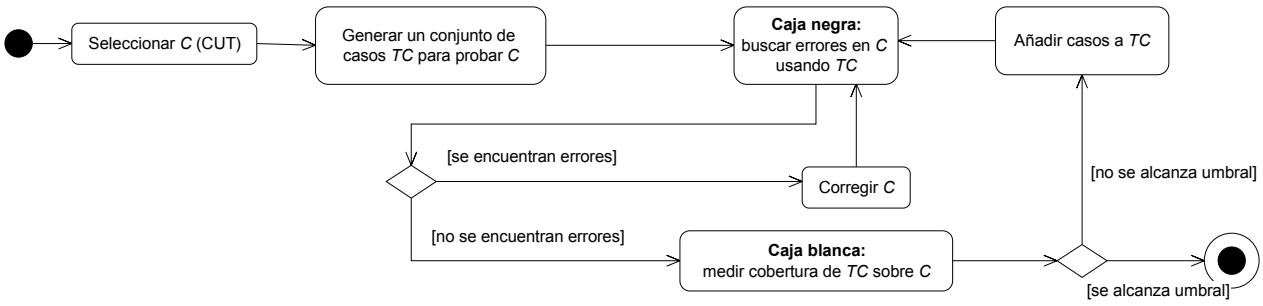

Figura 3. Estructura del proceso de pruebas unitarias

3.4. Resultados experimentales

Con objeto de validar el proceso de pruebas descrito y de comprobar la utilidad de la herramienta, se realizó un experimento controlado con 28 estudiantes de 5º curso de Ingeniería Informática. El problema consistió en probar la conocida clase del Triángulo [14]. Los alumnos se dividieron en dos grupos de 14 alumnos: un grupo probó la clase manualmente, utilizando JUnit; el otro grupo generó los casos de prueba utilizando *testooj*. Por razones de espacio no presentamos los detalles técnicos de preparación y realización del experimento, si bien todos sus detalles pueden consultarse en [15]. La hipótesis nula establecía que "La eficiencia del proceso de pruebas con y sin *testooj* no es significativamente distinta"; la alternativa, que sí lo era. Tras evaluar la calidad de los casos de prueba de ambos grupos traduciéndolos a formato MuJava (bien manualmente, bien con *testooj*) y ejecutándolos con este entorno, se obtuvo 0,000 como *p-value,* rechazándose por tanto la hipótesis nula.

4. OTROS RESULTADOS

testooj permite generar casos de prueba utilizando tres algoritmos diferentes [7]: *All combinations* (que obtiene todos los casos de prueba posibles mediante la combinación de todos los valores interesantes), *Antirandom* (que consiste en partir de un caso de prueba base y generar los siguientes de manera que tengan la mayor distancia Hamming o Euclídea a los ya generados) y *Each choice* (cada valor de cada parámetro se incluye en, al menos, un caso de prueba). Para un mismo conjunto de valores interesantes, *All combinations* es el algoritmo que obtiene la mayor cobertura posible; sin embargo, es también el que genera el número más alto de casos de prueba, muchos de los cuales son además redundantes (en el sentido de que ejercitan exactamente los mismos fragmentos de código). Los otros dos algoritmos generan menos casos, pero alcanzan también, por norma general, menos cobertura.

En *testooj* se han implementado dos algoritmos voraces que minimizan el conjunto de casos de prueba generado sin disminuir la cobertura: el primero (llamado *Decreasing)* va seleccionando aquellos casos que matan más mutantes; el segundo utiliza el criterio contrario, y toma aquellos casos que matan me-

nos (*Increasing)*. Éste es el problema conocido como "Reducción óptima de la batería de casos" [16], que es NP-completo y que ha sido muy estudiado en la literatura con técnicas de lo más variopintas (p. ej., [17-21]). *Decreasing* e *Increasing* consiguen reducciones de hasta el 97% en el número de casos (Figura 4).

Programa	**Nº de casos de prueba generados (*All combinations)***	**Nº de casos en el conjunto reducido (*Decreasing)***	**Nº de casos en el conjunto reducido (*Increasing)***
Bisectriz	64	4	9
Burbuja	125	5	8
Triángulo	216	22	31

Figura 4. Resultados experimentales en reducción de casos de prueba

Por otro lado, se ha descrito una técnica para "mutar casos de prueba". Un mutante de un caso de prueba para una CUT es un caso de prueba que procede de otro, al cual se le ha realizado algún tipo de cambio (p.ej., repetir o eliminar una llamada a un servicio, cambiar el orden de invocación a servicios…). Sorprendentemente, se han encontrado baterías de casos de prueba que, sin encontrar fallos, mataban al 100% de los mutantes de la CUT pero que, al generar los casos de prueba mutados, encontraban fallos que habían permanecido ocultos para la batería original (Figura 5).

	Casos de prueba			**Casos de prueba (mutantes)**		
	Nº de casos	**Casos que encuentran fallo**	**% de mutantes muertos**	**Nº de casos**	**Casos que encuentran fallo**	**% de mutantes muertos**
Burbuja.java	125	0	98%	4500	625	100%
Encontrar.java	256	0	100%	13056	3328	100%
Medio.java	27	0	88%	567	0	100%
Bisectriz.java	9	0	100%	135	11	100%
CuatroBolas.java	324	0	100%	10692	0	100%
Triángulo.java	216	0	100%	6372	53	100%

Figura 5. Algunos resultados de la técnica de mutación de casos de prueba

5. CONCLUSIONES

En este artículo se han presentado algunos resultados conseguidos en la línea de investigación de automatización del proceso de pruebas, dentro del proyecto Mantenimiento Ágil del Software. Todas las técnicas diseñadas se han ido plasmando en *testooj,* una herramienta de automatización del proceso para código Java y, en menor medida, en *NTesting,* para entornos .NET. En próximas fechas el proyecto citado llega a su fin; no obstante, las líneas futuras de investigación, ya iniciadas, se encaminan hacia la utilización del UML Testing Profile para automatizar el proceso de pruebas desde los primeros momentos del ciclo de vida.

AGRADECIMIENTOS

Este trabajo ha sido parcialmente financiado por el Plan Nacional de I+D+I del Ministerio de Educación y Ciencia y fondos FEDER, acción especial RePRIS (TIN2005-24792-E).

6. REFERENCIAS

1. Meudec C, *ATGen: automatic test data generation using constraint logic programming and symbolic execution.* Software Testing, Verification and Reliability, 2001. **11**(2): p. 81-96.
2. Ball T, Hoffman D, Ruskey F, Webber R, and White L, *State generation and automated class testing.* Software Testing, Verification and Reliability, 2000. **10**: p. 149-170.
3. Hoffman D, Strooper P, and Wilkin S, *Tool support for executable documentation of Java class hierarchies.* Software Testing, Verification and Reliability, 2005. **15**(4): p. 235-256.
4. Erdogmus H, Morisio M, and Torchiano M, *On the effectiveness of the test-first approach to programming.* IEEE Transactions on Software Engineering, 2005. **31**(3): p. 226-237.
5. Offutt J, Liu S, Abdurazik A, and Amman P, *Generating test data from state-based specifications.* Software Testing, Verification and Reliability, 2003(13): p. 25-53.
6. Kirani S and Tsai WT, *Method sequence specification and verification of classes.* Journal of Object-Oriented Programming, 1994. **7**(6): p. 28-38.
7. Grindal M, Offutt J, and Andler SF, *Combination testing strategies: a survey.* Software Testing, Verification and Reliability, 2005(15): p. 167-199.
8. Cornett S. *Code Coverage Analysis*. 2004 [cited June 15, 2005]; Available from: http://www.bullseye.com/webCoverage.html.
9. Demillo RA and Offut AJ, *Experimental results from an automatic test case generator.* ACM Transactions on Software Engineering and Methodology, 1993. **2**(2): p. 109-127.
10. Daniels F and Tai K. *Measuring the effectiveness of method test sequences derived from sequencing constraints*. *Technology of Object-Oriented Languages and Systems*. 1999. Santa Barbara, CA, USA.
11. Kim SW, Clark JA, and McDermid JA, *Investigating the effectiveness of object-oriented testing strategies using the mutation method.* Software Testing, Verification and Reliability, 2001. **11**: p. 207-225.
12. Vincenzi AMR, Maldonado JC, Barbosa EF, and Delamaro ME, *Unit and integration testing strategies for C programs using mutation.* Software Testing, Verification and Reliability, 2001. **11**(4): p. 249-268.
13. Ma Y-S, Offutt J, and Kwon YR, *MuJava: an automated class mutation system.* Software Testing, Verification and Reliability, 2004. **15**(2): p. 77-93.
14. Myers GJ, *The Art of Software Testing*, ed. J.W. Sons. 1979.
15. Polo M, Tendero S, and Piattini M, *Integrating techniques and tools for testing automation.* Software Testing, Verification and Reliability, 2007. **15**(1).
16. Jones JA and Harrold MJ, *Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage.* IEEE Transactions on Software Engineering, 2003. **29**(3): p. 195-209.
17. Harrold M, Gupta R, and Soffa M, *A methodology for controlling the size of a test suite.* ACM Transactions on Software Engineering and Methodology, 1993. **2**(3): p. 270-285.
18. Jeffrey D and Gupta N. *Test suite reduction with selective redundancy*. *International Conference on Software Maintenance*. 2005. Budapest (Hungary): IEEE Computer Society Press.
19. Tallam S and Gupta N. *A concept analysis inspired greedy algorithm for test suite minimization*. *6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 05)*. 2005.
20. Heimdahl MPE and George D. *Test-Suite Reduction for Model Based Tests: Effects on Test Quality and Implications for Testing*. *19th IEEE International Conference on Automated Software Engineering (ASE'04)*. 2004.

21. McMaster S and Memon AM. *Call Stack Coverage for Test Suite Reduction*. *21st IEEE International Conference on Software Maintenance*. 2005. Budapest (Hungary).