Revista
Española de

Innovación, Calidad e Ingeniería del Software

REICIS

Volumen 5, No. 3, octubre, 2009

Web de la editorial: www.ati.es
Web de la revista: www.ati.es/reicis
E-mail: calidadsoft@ati.es
ISSN: 1885-4486

Publicado por la Asociación de Técnicos de Informática (ATI), Via Laietana, 46, 08003 Barcelona.

Secretaría de dirección: ATI Madrid, C/Padilla 66, 3º dcha., 28006 Madrid

Revista Española de Innovación, Calidad e Ingeniería del Software (REICIS)

Contenidos

REICIS

Una herramienta para la reducción de conjuntos de casos de prueba

Pedro Reales
Escuela Superior de Informática – Universidad de Castilla-La Mancha – España
pedro.reales@uclm.es

Macario Polo
Escuela Superior de Informática – Universidad de Castilla-La Mancha – España
macario.polo@uclm.es

Resumen

Este artículo presenta un algoritmo de reducción para conjuntos de casos de prueba en formato JUnit, así como una herramienta (en forma de plugin para Eclipse) que lo implementa, y que puede descargarse desde http://geclipsetesting.sourceforge.net/. El objetivo de la reducción de conjuntos de casos de prueba es la obtención de una nueva versión del conjunto, pero con menos casos de prueba, mientras que se mantiene la cobertura alcanzada se mantiene. El algoritmo, y la implementación que se la ha dado en el plugin, consiguen estos resultados.

Palabras clave: Generación de casos de prueba, reducción de conjuntos de casos de prueba, JUnit, cobertura de código.

A tool for minimizing sets of test cases

Abstract

This article presents an algorithm for reducing JUnit test suites, as well as a tool (implemented as an Eclipse plugin) that realizes it. The tool can be downloaded from http://geclipsetesting.sourceforge.net/. With test suite reduction, the test engineer may get smaller test suites that, however, keep the same coverage that the original one. The algorithm and its implementation fulfil these goals.

Key words: Test case generation, test suite reduction, JUnit, code coverage.

Zulueta,Y., Despaigne,E. y Hernández A., "Una herramienta para la reducción de conjuntos de casos de prueba", REICIS, vol. 5, no.3, 2009, pp.21-37. Recibido: 15-3-2009; revisado: 10-4-2009; aceptado: 19-9-2009.

1. Introducción

La priorización de los casos de prueba es una práctica recomendada para reducir los costes en las pruebas de regresión: básicamente, la idea consiste en ejecutar aquellos casos de prueba que son más importantes de acuerdo a diferentes criterios de calidad. Una manera de priorizar los casos de prueba consiste en reducir el tamaño del *test suite* (conjunto de casos de prueba) sin perder calidad. Más formalmente, se transforma el conjunto de casos de prueba T en un nuevo conjunto $T' \subseteq T$, siendo $|T'| \leq |T|$ y preservando T' la misma cobertura alcanzada que con el conjunto T. Este problema ha sido analizado en muchos contextos por diferentes investigadores. Jones y Harrold [1] plantean el problema de "reducción al conjunto óptimo de casos de prueba" como se muestra en la Figura 1.

Dados: Un conjunto de casos de prueba *T*, un conjunto de requisitos de prueba $r_1, r_2, ..., r_n$, que deben ser satisfechos para alcanzar la cobertura deseada en el programa bajo prueba.
Problema: Encontrar $T' \subset T$ *de manera que T' satisface todos los* r_i *y* $(\forall T'' \subset T, T''$ *satisface todos los* $r_i \Rightarrow |T'| \leq |T''|)$

Figura 1. Problema de reducción al conjunto óptimo de casos de prueba [1]

El problema de la reducción óptima (es decir, la obtención de un *test suite* nuevo cuyo cardinal sea el mínimo posible) es NP-completo [2], por lo que no es un problema resoluble en tiempo polinomial. Por esta razón, todos los algoritmos que tratan este problema obtienen soluciones próximas a la óptima, pero no ofrecen garantías de que la solución ofrecida sea, desde el punto de vista del cardinal del *test suite*, la mejor posible. La mayoría de los algoritmos desarrollados son voraces, y obtienen *test suites* reducidos que cumplen los mismos requisitos de calidad (normalmente, algún criterio de cobertura) que el original.

Aunque durante años las actividades de prueba se han llevado a cabo de una manera relativamente descuidada, la introducción, hace algunos años, de los *frameworks* automatizados tipo X-Unit (JUnit, NUnit, etc…), ha permitido a las organizaciones desarrolladoras de software ir introduciendo, progresivamente, buenas prácticas de *testing* en sus proyectos de desarrollo [3]. En torno a estos *frameworks*, que supusieron realmente un avance importante para la "democratización" el *testing*, diferentes investigadores y

compañías han desarrollado extensiones que permiten algunas funcionalidades adicionales, como pruebas de interfaces de usuario (mediante UISpec, por ejemplo), pruebas de caja blanca (con herramientas o plugins como Coverlipse o Eclemma Code Coverage [4], descargable éste desde http://www.eclemma.org/), y pruebas de mutación (como Muclipse, un plugin para Eclipse basado en MuJava [5]).

El uso conjunto de JUnit y Eclemma permite detectar fallos en el sistema bajo prueba, a la vez que se obtiene una estimación del código recorrido por los casos de prueba, en forma de un valor de la cobertura de sentencias de código fuente o de instrucciones de *bytecode*. Para disminuir el coste derivado de la reejecución de los casos de prueba hemos desarrollado e implementado un algoritmo que reduce el tamaño de un conjunto de casos de prueba en formato JUnit, mientras que preserva la cobertura alcanzada en el sistema bajo prueba.

Este artículo se organiza de la siguiente manera: en la sección 2 se hace una introducción al problema de la reducción de los casos de prueba y se describen algunos algoritmos que lo resuelven. En la sección 3 se presenta el algoritmo que hemos desarrollado, cuya implementación se describe en al sección 4. En la sección 5 se presenta un "ejemplo motivacional" extraído de la literatura de referencia. En la sección 6 se describen algunos experimentos llevados a cabo. Por último, la sección 7 muestra las conclusiones a las que se han llegado.

2. Algoritmos para reducir conjuntos de casos de prueba.

Supongamos que tenemos una versión orientada a objetos del clásico problema de la determinación del tipo de un Triangulo [6]: el lado izquierdo de la Figura 2 representa la posible estructura de esta clase: dispone de tres métodos de configuración (*setI*, *setJ* y *setK*, que asignan valor a los tres lados del triángulo), y de un método *getType* (que devuelve un valor numérico que representa si la figura es un triangulo o no, y el tipo de triángulo en caso afirmativo: equilátero, isósceles o escaleno).

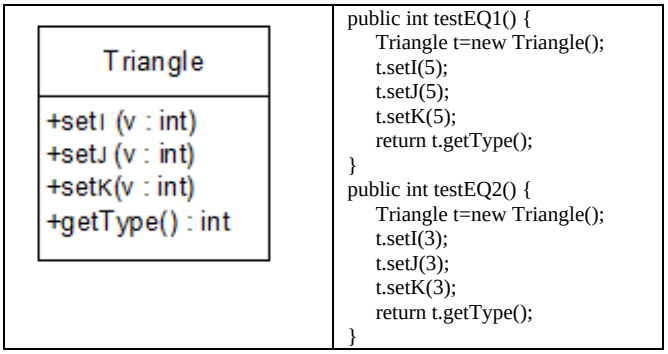

Figura 2. Representación del clásico problema del triangulo mediante una clase

En el lado derecho de la figura anterior aparecen dos posibles casos de prueba para la clase: ambos corresponden a un triángulo equilátero y, probablemente, ambos ejecutarán el mismo conjunto de instrucciones (o cualquier otro criterio de cobertura) de la clase bajo prueba. Desde el punto de vista de las pruebas, el segundo caso de prueba es redundante con respecto al primero, por lo que, si se fuese a reducir el tamaño de este *test suite*, el ingeniero de pruebas podría quedarse solamente con uno de ellos.

El problema de la redundancia de casos de prueba puede ser serio cuando existen múltiples casos de prueba y su redundancia no es tan evidente como ocurre en este sencillo ejemplo. En estas situaciones, el ingeniero de pruebas debería aplicar algún tipo de estrategia de reducción al conjunto de casos de prueba para obtener un conjunto más pequeño, pero que alcance la misma cobertura que el conjunto original en el sistema bajo prueba. En esta sección se hace una revisión de algunos algoritmos desarrollados por diferentes investigadores para reducir conjuntos de casos de prueba.

2.1 El algoritmo HGS

Harrold, Gupta y Sofa [7] proponen un algoritmo voraz (normalmente referenciado como HGS) para reducir un conjunto de casos de prueba preservando los requisitos de las pruebas cubiertos por el conjunto original. Los “requisitos de las pruebas” (*test requirement*) son, normalmente, uno o más criterios de cobertura, de manera que este algoritmo permite seleccionar un subconjunto de casos que verifique más de un criterio de cobertura.

Los principales pasos de este algoritmo son:

1. Inicialmente, todos los requisitos están sin marcar.

2. Añadir los casos de prueba que ejecutan sólo un requisito al conjunto de casos de prueba reducido y marcar todos los requisitos cubiertos por los casos de prueba seleccionados.
3. Ordenar los requisitos sin marcar de acuerdo al número de casos de casos de prueba que ejecutan cada requisito. Si varios requisitos son cubiertos por el mismo número de casos de prueba, se marca el mayor número de estos requisitos. Si además hay múltiples casos de prueba "empatados", se selecciona el caso de prueba que marque más requisitos en total. Si aun así sigue habiendo un empate, se selecciona un caso de prueba arbitrariamente. Entonces, se marcan los requisitos ejecutados por el caso de prueba seleccionado, y se eliminan los casos de prueba redundantes con respecto a los casos de prueba ya seleccionados.
4. Repetir el paso 3 hasta que todos los requisitos de pruebas hayan sido marcados.

2.2 Mejoras de Gupta

Con diferentes colaboradores, Gupta ha propuesto una serie de mejoras para mejorar el algoritmo HGS:

- Con Jeffrey [8], Gupta añade al algoritmo "redundancia selectiva". "La redundancia selectiva" hace posible la selección de un caso de prueba que, por cualquier requisito de prueba dado, alcanza la misma cobertura que otro caso de prueba seleccionado anteriormente, pero que ahora añade cobertura de un requisito de prueba diferente. Por ejemplo, se puede dar la posibilidad de que T' alcance el criterio de todas las ramas, pero no el de *def-uses*; por lo tanto, se puede añadir un nuevo caso de prueba t a T' si éste incrementa la cobertura de los requisitos de *def-uses*: ahora, T' no incrementará el criterio de todas las ramas, pero sí que incrementa el criterio de *def-uses*.
- Con Tallam [9], la selección de los casos de prueba se basa en técnicas de análisis de conceptos. De acuerdo con los autores, esta versión alcanza mejores reducciones que las versiones previas, sin un empeoramiento en el tiempo de ejecución.

2.3 Algoritmo de Heimdahl y George

Heimdahl y George [10] también proponen un algoritmo voraz para reducir conjuntos de casos de prueba. Básicamente, lo que hacen es tomar un caso de prueba aleatoriamente, lo ejecutan y comprueban la cobertura alcanzada. Si esta cobertura es mayor que la cobertura alcanzada hasta ahora, se añade el caso de prueba al conjunto final.

Este algoritmo es repetido cinco veces para obtener cinco conjunto reducidos de casos de prueba diferentes. Puesto que el azar es un componente esencial, este algoritmo no puede garantizar una buena calidad de los resultados.

2.4 Algoritmo de McMaster y Memon

McMaster y Memon [11] también presentan un algoritmo voraz. El parámetro tomado en cuenta para incluir los casos de prueba en el conjunto reducido se basa en el número de llamadas hechas a la pila del programa bajo prueba. Como se puede ver, este criterio de selección no es un requisito de pruebas frecuente.

2.5 Resumen

Puesto que el problema de reducir un conjunto de casos de prueba es un problema NP-Completo, todas las propuestas discutidas presentan un algoritmo voraz para encontrar una buena solución en un tiempo polinomial. Los requisitos de las pruebas para seleccionar los casos de prueba pueden ser cualquiera: cobertura de bloques, de caminos, etc.

El algoritmo presentado en este trabajo es también un algoritmo voraz, garantizando que la cobertura alcanzada por T’ es la misma que la alcanzada por T. Las principales diferencias con los trabajos discutidos anteriormente son la implementación amigable de este algoritmo en una herramienta fácilmente usable, su aplicación con casos de prueba en formato JUnit (un *framework* ampliamente extendido) y la posibilidad de usar las ventajas de herramientas de terceros (como se verá más adelante, nuestra herramienta hace uso de las capacidades de Eclemma).

3. Un algoritmo para reducir conjuntos de casos de prueba basado en cualquier criterio de cobertura

El algoritmo voraz presentado aquí puede hacer uso de cualquier criterio de cobertura para incluir casos de prueba en el conjunto reducido de casos de prueba final.

Inicialmente, el algoritmo ejecuta todos los casos de prueba del conjunto original T y registra los elementos de código fuente (por ejemplo, sentencias) alcanzados por cada caso de prueba. Después, el algoritmo sigue los siguientes pasos: (1) añade al conjunto reducido T' el caso de prueba *t* que alcance el mayor valor de cobertura para el criterio seleccionado (por ejemplo, selecciona el caso de prueba que alcanza un mayor número de sentencias); (2) elimina aquellos casos de prueba que sólo ejecutan elementos que ya han sido visitados por los casos de prueba que están en T' (por ejemplo, la primera vez se borran los casos de prueba que solo ejecutan sentencias ejecutadas por *t*); (3) se repiten los pasos (1) y (2) hasta que todos los elementos alcanzados por los casos de prueba de T sean alcanzados por los casos de prueba de T'.

A modo de ejemplo, la Tabla 1 muestra la matriz de alcance que un conjunto de casos de prueba (compuesto por los casos *tc1* al *tc6*) obtiene en una supuesta clase (compuesta por las sentencias *s1* a *s7*). Esta matriz se construye en el primer paso del algoritmo. Después, el algoritmo toma el caso de prueba que ejecuta más sentencias, que en el ejemplo pueden ser *tc2* o *tc3*. Supongamos que se selecciona *tc2*, el cual se añade a T'. Puesto que las sentencias alcanzadas por *tc1* y *tc5* también son ejecutadas por *tc2*, se eliminan *tc1* y *tc5*, dejando la matriz como en la Tabla 2.

	tc1	tc2	tc3	tc4	tc5	tc6
s1	X	X				
s2	X	X			X	
s3		X				
s4			X			
s5			X			
s6			X	X		
s7				X		X

Tabla 1. Matriz de alcance (I) de un programa

	tc2	tc3	tc4	tc6
s1	X			
s2	X			
s3	X			
s4		X		
s5		X		
s6		X	X	
s7			X	X

Tabla 2. Matriz de alcance (II) de un programa

Después, el algoritmo selecciona el siguiente caso de prueba que ejecuta más sentencias (*tc3*) y lo añade al conjunto reducido de casos de prueba. Hecho esto, el algoritmo selecciona *tc4* y elimina *tc6*, de manera que el conjunto final queda como sigue:

$$T'=\{tc2, tc3, tc4\}.$$

Realizar una comparación de este algoritmo con los algoritmos comentados en la sección 2 presenta muchas dificultades: (1) en primer lugar, porque a veces es técnicamente complicado implementar los algoritmos, por ejemplo, el criterio usado por McMaster y Memon ("llamada únicas a la pila"). (2) El algoritmo de Heindahl y George selecciona el conjunto reducido de casos de prueba mediante azar, lo cual no garantiza la calidad y, además, la comparación sería diferente después de cada ejecución. (3) El agoritmo HGS y sus mejoras usan dos o más tipos de requisitos de las pruebas para realizar la selección de los casos de prueba: Nuestro algoritmo toma el criterio del plugin Eclemma Code Coverage (instrucciones de código byte, sentencias de código fuente, métodos, tipos y bloques), entre los cuales existen relaciones subsunción (por ejemplo, el criterio de sentencias subsume al criterio de métodos). Además, cuando reducimos por el criterio más estricto de los que soporta Eclemma, estamos también reduciendo por los otros criterios, por lo que la comparación con el algoritmo HGS no tiene sentido. Si el algoritmo HGS fuera aplicado con únicamente un criterio de los permitidos por Eclemma, los resultados serían similares a los nuestros.

4. Implementación del algoritmo en Geclipse

El algoritmo ha sido implementado en *Geclipse*, un plugin para Eclipse el cual, de forma amigable, ofrece algunas de las funcionalidades de *testooj* [12] dentro del entorno de desarrollo. Geclipse permite también la generación automática de casos de prueba para una clase determinada con diferentes estrategias de generación [13] (*each choice*, algunas variantes de *pair-wise* y *all combinations*).

Geclipse utiliza Eclemma para acceder a la cobertura alcanzada por cada caso de prueba y de esta manera poder construir la matriz de alcance. Hecho esto, Geclipse implementa el algoritmo descrito en la sección 3 para reducir la matriz.

El diseño arquitectónico de Geclipse, Eclemma y Eclipse se muestra en la Figura 3: el paquete Eclemma representa los elementos del plugin Eclemma invocados por Geclipse, el

cual esta compuesto por los paquetes presentación, dominio y persistencia. Como se muestra en la Figura 3. Arquitectura de Geclipse, *ControladorGCP* (el controlador de Geclipse) invoca a la clase *SessionAnalyzer* de Eclemma para obtener la cobertura alcanzada por cada caso de prueba. Después, estas coberturas se procesan mediante las clases del paquete *conjuntoMinimo* de Geclipse.

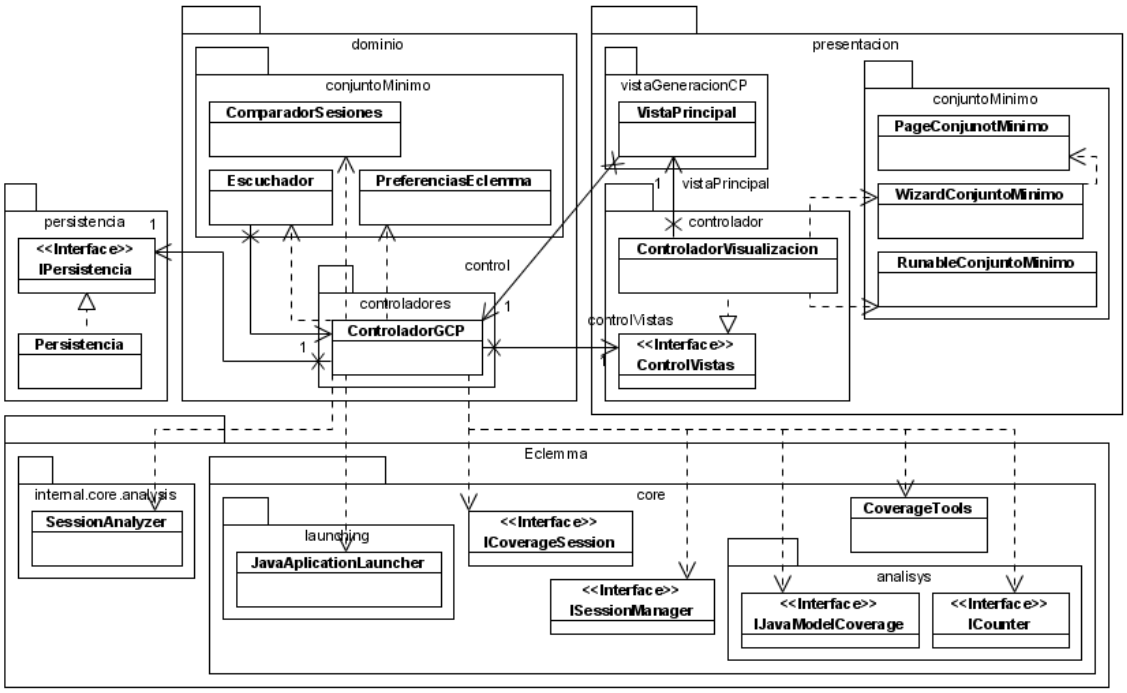

Figura 3. Arquitectura de Geclipse

Eclemma es capaz de medir cinco criterios de cobertura diferentes: instrucciones de código byte, sentencias de código fuente, métodos, tipos y bloques. En una sesión de pruebas, este plugin recoge todas las medidas y las guarda en un objeto de tipo *ICoverageSession* (Figura 3); después, Geclipse procesa esos resultados para realizar los análisis correspondientes.

Geclipse también ofrece la posibilidad de generar casos de prueba usando expresiones regulares (las cuales de diseñan usando el conjunto de operaciones públicas de la clase bajo prueba), asignar valores de prueba a los parámetros de las operaciones y ejecutar los casos de prueba. Esta técnica de generación de casos de prueba se inspira en la propuesta de Kirani y Tsai [14], realizada en 1994, y ya había sido implementada en la herramienta *testooj* [12].

En la Figura 4 (lado derecho), el ingeniero de pruebas está escribiendo una expresión regular para generar casos de prueba para la clase *TriangleType* (problema de la determinación del tipo de un triángulo antes mencionado). Esta expresión será procesada usando el paquete *java.util.regex* para, mediante su expansión, obtener "plantillas de prueba". Una plantilla de prueba consiste en una secuencia de expresiones que, más tarde, debe recibir valores de prueba para, mediante su combinación, pueden poder obtener casos de prueba ejecutables. Así, por ejemplo, a partir de la expresión regular *TriangleType().[setI(int)|setJ(int)|setK(int)]*getType(),* podrían obtenerse, entre otras muchas, las plantillas de prueba siguientes:

TriangleType().getType()

TriangleType().setI(int).getType()

TriangleType().setI(int).setJ(int).setK(int).getType()

TriangleType().setI(int).setJ(int).setK(int).setI(int).getType()

…

Para obtener casos de prueba ejecutables, se asignan valores de prueba (*test data*) a los parámetros de las operaciones que intervienen en la expresión regular. Asignando, por ejemplo, los valores 1 2 y 3 al único parámetro de los métodos *setI*, *setJ* y *setK*, de la tercera plantilla de prueba que se acaba de poner como ejemplo, podrían obtenerse los siguientes casos de prueba:

TriangleType().setI(1).setJ(1).setK(1).getType()

TriangleType().setI(1).setJ(1).setK(2).getType()

TriangleType().setI(1).setJ(1).setK(3).getType()

TriangleType().setI(1).setJ(2).setK(1).getType()

…

En el árbol de la Figura 4 (lado izquierda) se muestran algunas de las plantillas de prueba procedentes de la expresión regular de la Figura 4 (Derecha). Como ya se ha comentado, estas plantillas de prueba deben ser combinadas con datos de prueba reales.

Por otro lado, también es posible que el resultado esperado de alguno de los casos de prueba sea el lanzamiento de una excepción. En la ventana mostrada en la Figura 5, el ingeniero de pruebas está, por un lado, asignando valores de prueba a los parámetros de las operaciones; por otro, puede asociar excepciones a los casos de prueba que se generarán.

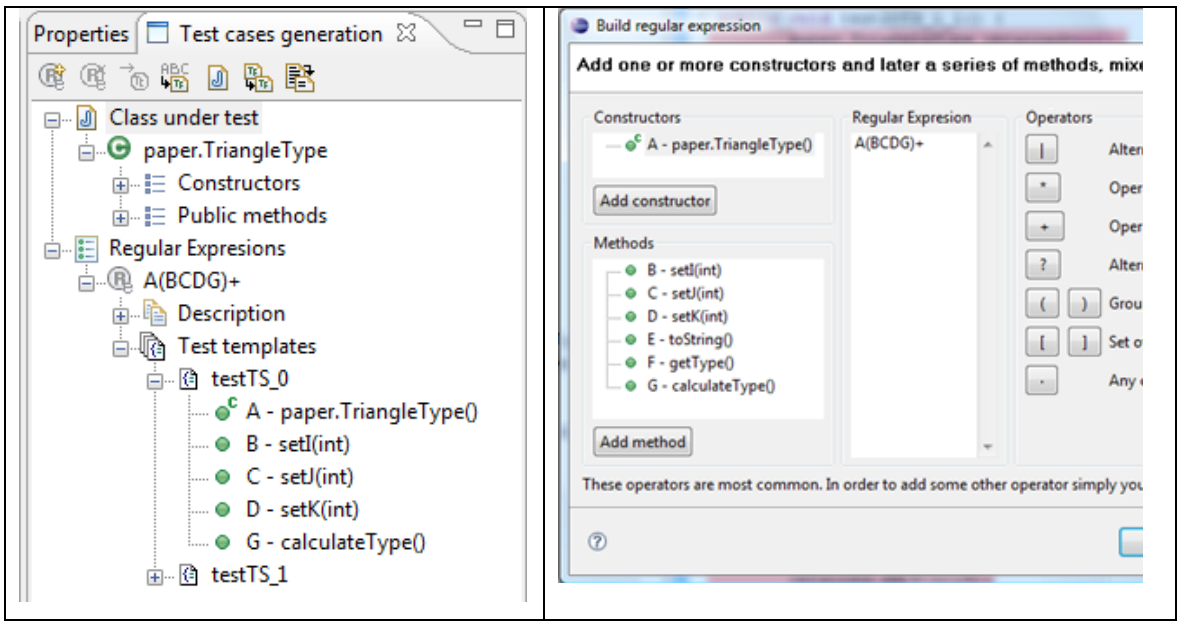

Figura 4. Vista de Geclipse con la estructura de una clase bajo prueba y las plantillas de pruebas generadas (izquierda). A la derecha, el *tester* escribe una expresión regular para generar casos de prueba

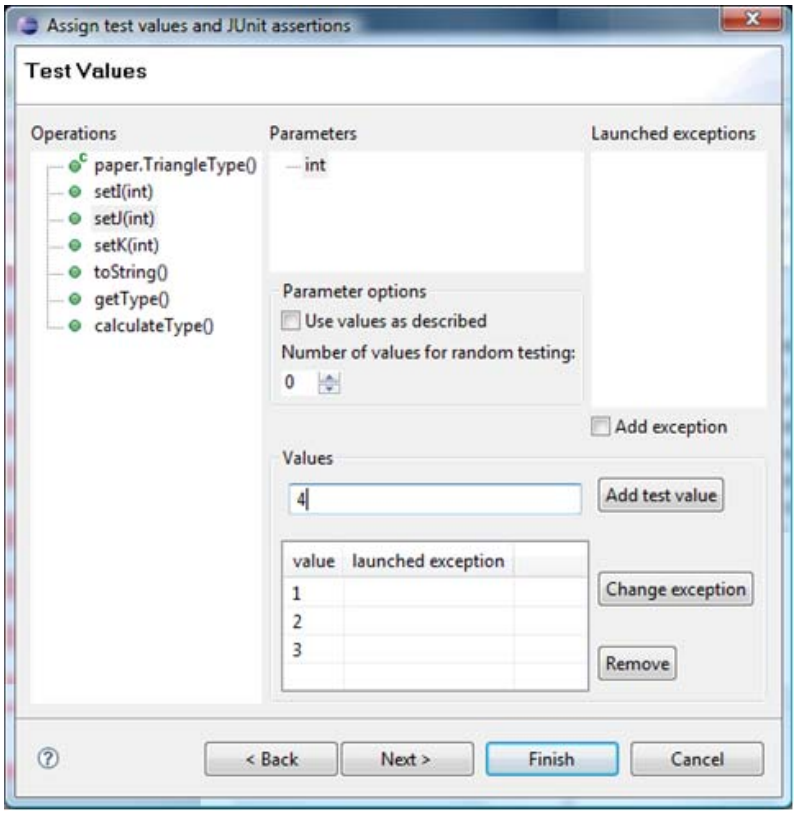

Figura 5. Asignación de valores de prueba

Cuando el conjunto de casos de prueba está disponible (ya sea mediante su confección manual, o mediante su generación automática a partir de la expansión de expresiones regulares), se pueden llevar a cavo las tareas de ejecución y reducción del *test suite.*

El lado izquierdo de la Figura 6 muestra las diferentes opciones de cobertura para reducir el conjunto de casos de prueba: Eclemma evalúa la cobertura que alcanzan los casos de prueba en todo el sistema que se está probando. Geclipse permite reducir el conjunto de casos bien fijándose en la cobertura del sistema completo, bien fijándose sólo en una clase. Igualmente, el usuario puede elegir cuál de los cinco criterios de cobertura medidos por Eclemma se utilizará para realizar la reducción. Durante la ejecución de los casos (lado derecho de la Figura 6), Geclipse utiliza la información de cobertura suministrada por Eclemma para realizar la reducción del *test suite.*

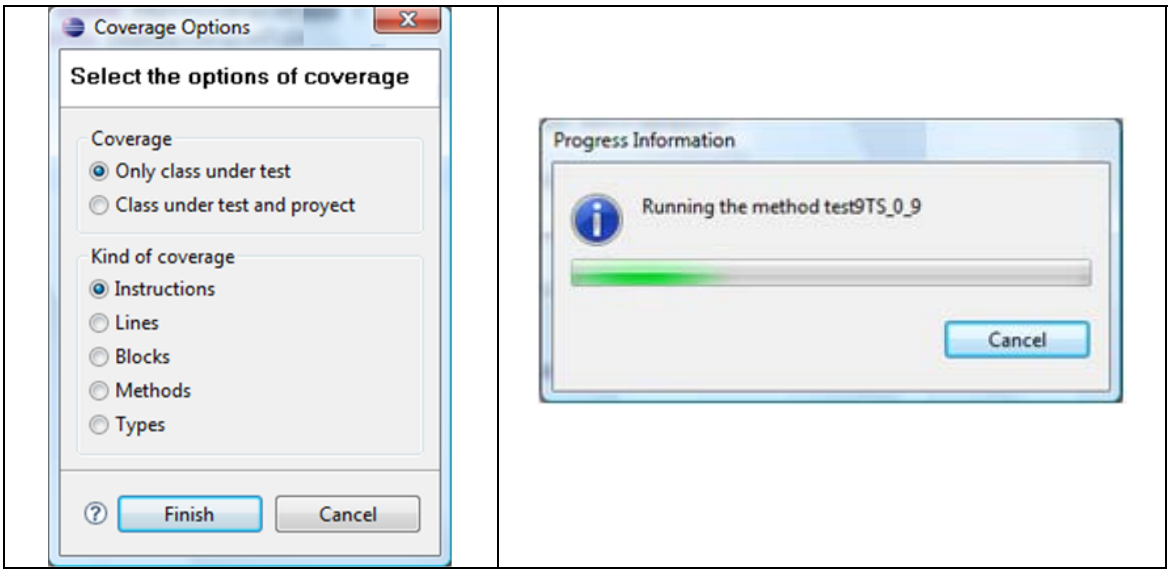

Figura 6. Selección del tipo de cobertura (Izquierda) y proceso de reducción del conjunto de casos de prueba (Derecha)

Finalmente, cuando la reducción se ha completado, la herramienta construye un segundo conjunto de casos de prueba que sólo contiene los casos de prueba seleccionados mediante el algoritmo (Figura 7), y el cual obtiene la misma cobertura que el conjunto original. De este modo, los costes de reejecución de casos (sobre todo, en situaciones de realización de pruebas de regresión) pueden verse disminuidos de forma importante, ya que sólo se ejecutan los ejemplares verdaderamente significativos.

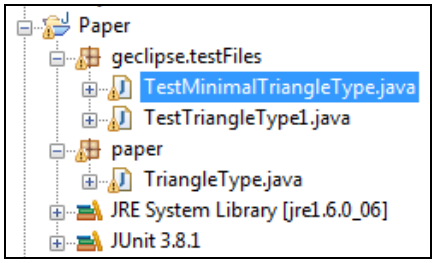

Figura 7. Conjunto de casos de prueba reducido

5. "Un ejemplo motivador"

En el trabajo [8], Jeffrey y Gupta muestran, con el mismo título que esta sección (*A motivational example*), un pequeño programa para ejemplificar su algoritmo con redundancia selectiva. Este programa ha sido traducido al código Java en la Figura 8.

```
public class JGExample {
    float returnValue;
    public float f(float a, float b, float c, float d) {
        float x=0, y=0;
        if (a>0)
            x=2;
        else
            x=5;
        if (b>0)
            y=1+x;
        if (c>0)
            if (d>0)
                returnValue=x;
            else
                returnValue=10;
        else
            returnValue=(1/(y-6));
        return returnValue;
    }
    public String toString() {
        return "" + returnValue;
    }
}
```

Figura 8. Una versión en Java del "ejemplo motivador" de Jeffrey y Gupta

Usando tres valores de prueba (-1.0, 0.0 y 1.0) para cada uno de los cuatro parámetros de la función *f*, y generando casos de prueba con el algoritmo *all-combinations*, Geclipse genera un archivo en formato JUnit con 3x3x3x3=81 casos de prueba, que alcanzan todas las sentencias de la clase bajo prueba.

Después de aplicar nuestro algoritmo, el conjunto de 81 casos de prueba se reduce a 3 casos de prueba (un 3,7% del conjunto original).

6. Validaciones adicionales

Además del ejemplo ilustrativo de Jeffrey y Gupta, la implementación del algoritmo ha sido aplicado a un conjunto de programas *benchmark*, que de acuerdo con [15], se pueden clasificar en dos categorías:

- *Toy programs* (programas "de juguete"): hemos usado algunos de los programas que Pargas y Harrold [16] usaron para validar sus algoritmos de generación de datos de prueba. Para estos programas generamos un conjunto de casos de prueba con el algoritmo *all-combinations* de *testooj*.
- *Industrial programs* (programas "industriales"): la clase *PluginTokenizer* de la aplicación *jtopas* incluida en el Software Infrastructure Repository (SIR) [17], una propuesta de Do, Elbaum y Rothermel que persigue la puesta a disposición de la comunidad científica de un conjunto de programas relativamente complejos, que puedan ser utilizados como referencia para la realización y replicación de experimentos de *testing*. Para nuestro experimento se usaron los 14 casos de prueba que se proporcionan en esta *infraestructura* para la prueba de esa clase. *jtopas* implementa un párser de archivos XML y está compuesto por 22 clases.

Los resultados obtenidos después de aplicar el algoritmo son los que se muestran en la Tabla 3. Como se puede ver, se consiguen reducciones próximas al 90% del tamaño del *test suite* original. En este ejemplo, el criterio de cobertura seleccionado es el más estricto que permite el plugin de "Eclemma code coverage", sentencias, que es uno de los más usados en el desarrollo de software no crítico.

Los resultados con las clases *TriTyp* y *PluginTokenizer* son especialmente ilustrativos: el primer ejemplo es el ejemplo más usado en el campo de la investigación sobre las pruebas del software; en el segundo caso, los desarrolladores de la clase podrían haber escrito únicamente uno de los casos de prueba para alcanzar la misma cobertura que con el conjunto de casos de prueba completo.

Obviamente, la aplicación de cualquiera de los algoritmos revisados en la sección 2 o de nuestro algoritmo sobre un conjunto de de casos de prueba que ya es mínimo, no producirá ningún tipo de reducción: es un conjunto reducido, y no puede volver a ser reducido usando el mismo criterio de cobertura y el mismo algoritmo.

			# of test cases		
	Programa	**Nº de sentencias**	**Original**	**Reducido**	**Reducción**
Toy	Bisect	19	8	1	87,5%
	Bub	27	320	1	99,7%
	TriTyp	49	216	7	96,7%
Industrial	PluginTokenizer	157	14	1	92,8%

Tabla 3. Resultados al aplicar el algoritmo de reducción al conjunto de programas benchmark

7. Conclusiones y trabajos futuros

En este artículo se ha presentado un algoritmo eficiente para reducir el tamaño de un conjunto de casos de prueba en formato JUnit. JUnit (y en general todos los entornos X-Unit) son los entornos de pruebas más utilizados actualmente en las compañías de desarrollo de software. La combinación de JUnit con otras herramientas para realizar pruebas de caja blanca (como pueden ser Eclemma o Coverlipse) permite a los desarrolladores conocer la cobertura alcanza por sus casos de prueba. A su vez, esto conduce a la adición de nuevos casos de prueba, que permitan el recorrido de las zonas de código que permanecían inexploradas. Sin embargo, muchos de estos casos de prueba son probablemente redundantes, lo que finalmente incrementa el coste de la re-ejecución de los casos de prueba. Por lo tanto, las habilidades de herramientas como Geclipse proporcionan una buena ayuda para disminuir los costes de las pruebas de regresión.

Según nuestras noticias, Geclipse es la primera herramienta que incluye la implementación de un algoritmo para reducir conjuntos de casos de prueba aplicable a una herramienta de pruebas ampliamente usada como es JUnit.

Como trabajos futuros, sería interesante la posibilidad de incluir en la herramienta más algoritmos de reducción de conjuntos de casos de prueba, como los revisados en la sección 2. Por ejemplo, implementando el algoritmo HGS se daría la posibilidad de usar más de un criterio de cobertura al mismo tiempo. Relacionado con los criterios de cobertura, la integración de nuestro plugin con otras herramientas, que soporten más criterios de cobertura, sería también interesante.

Por último, indicar que Geclipse puede descargarse e instalarse desde la siguiente URL: http://geclipsetesting.sourceforge.net/

Agradecimientos

Este trabajo está parcialmente soportado por el proyecto PRALÍN (Pruebas en Líneas de Producto): Junta de Comunidades de Castilla-La Mancha/Fondo Europeo de Desarrollo Regional (FEDER), PAC08-0121-1374.

Referencias

[1] Jones JA and Harrold MJ. "Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage". IEEE Transactions on Software Engineering, vol. 29, nº 3, pp. 195-209, 2003.

[2] Garey MR and Johnson DS. *Computers and Intractability*. New York: W.H. Freeman, 1979.

[3] Do H, Rothermel G and Kinner A. "Prioritizing JUnit test cases: An empirical assessment and cost-benefit analysis". Empirical Software Engineering, vol. 11, nº 1, pp. 33-70, 2006.

[4] Hoffmann M. "Code Coverage Analysis for Eclipse". Eclipse Summit Europe 2007. Ludwigsburg, 2007.

[5] Ma Y-S, Offutt J and Kwon YR. "MuJava: an automated class mutation system". Software Testing, Verification and Reliability, vol. 15, nº 2, pp. 97-133, 2005.

[6] Myers B. *The Art of Software Testing*: John Wiley & Sons, 1979.

[7] Harrold M, Gupta R and Soffa M. "A methodology for controlling the size of a test suite". ACM Transactions on Software Engineering and Methodology, vol. 2, nº 3, pp. 270-285, 1993.

[8] Jeffrey D and Gupta N. "Test suite reduction with selective redundancy". International Conference on Software Maintenance. Budapest (Hungary), pp. 549-558. IEEE Computer Society, 2005.

[9] 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, pp. 35-42. 2005.

[10] Heimdahl M 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, pp. 176-185. 2004.

[11] McMaster S and Memon A. "Call Stack Coverage for Test Suite Reduction". 21st IEEE International Conference on Software Maintenance. Budapest (Hungary), pp. 539-548, 2005.

[12] Polo M, Piattini M and Tendero S. "Integrating techniques and tools for testing automation". Software Testing, Verification and Reliability, vol. 17, nº 1, pp. 3-39, 2007.

[13] Grindal M, Offutt AJ and Andler SF. "Combination testing strategies: a survey". Software Testing, Verification and Reliability, vol. 15, nº, pp. 167-199, 2005.

[14] Kirani S and Tsai WT. "Method sequence specification and verification of classes". Journal of Object-Oriented Programming, vol. 7, nº 6, pp. 28-38, 1994.

[15] Juristo N, Moreno AM and Vegas S. "A Survey on Testing Technique Empirical Studies: How Limited is our Knowledge". International Symposium on Empirical Software Engineering (ISESE'02). Nara, Japan, pp. 161-172, 2002.

[16] Pargas RP, Harrold MJ and Peck RR. "Test-Data Generation Using Genetic Algorithms". Software Testing, Verification and Reliability, vol., nº 9, pp. 263-282, 1999.

[17] Do H, Elbaum SG and Rothermel G. "Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact". Empirical Software Engineering: An International Journal, vol. 10, nº 4, pp. 405-435, 2005.