Andorra La Vella, del 8 al 10 de julio 2015
ISBN: 978-99920-70-10-9

Mutantes como apoyo para la valoración de pruebas

Francisco Chicano y Francisco Durán
E.T.S. Ingeniería Informática
Universidad de Málaga, Andalucía Tech, 29071, Málaga
{chicano,duran}@lcc.uma.es

Resumen

En este trabajo proponemos el uso de la técnica de *mutation testing*, o pruebas basadas en mutación, en asignaturas de pruebas de software para ayudar en la labor de evaluación de las mismas, a la vez que su propio uso ayuda a los alumnos a entender y asimilar la técnica en sí, utilizándola para mejorar sus conjuntos de pruebas. Describimos nuestra experiencia usando *mutation testing* en una asignatura de pruebas de software. Gracias al uso de una herramienta que automatiza la generación de mutantes y su evaluación, el profesor puede obtener una idea rápida de la calidad de las pruebas que puede servir para guiar/confirmar su evaluación. El alumno, por su parte, puede usarla para obtener una primera evaluación de su trabajo y como guía para mejorar el conjunto de pruebas.

Abstract

In this work, we propose the use of the mutation testing technique in software testing courses, to help the instructor in the students' assessment. Students can also use it to better understand the technique and improve their test suites. We describe our experience using mutation testing in a software testing course. Supported by a software tool to automate the mutants generation and evaluation, the instructor can get a fast assessment of the test suites quality. This initial assessment can be used to guide/confirm the instructor's personal assessment. Students can use such a tool to get a first assessment of their job and guide the steps to improve their test suite.

Palabras clave

Mutation testing, pruebas de software, evaluación.

1. Introducción

Hace unos años era difícil, casi imposible, encontrar asignaturas sobre pruebas en las titulaciones relacionadas con la informática en las universidades españolas. Tampoco era fácil encontrarlas en otros países. Sin embargo, las necesidades de las empresas dedicadas al desarrollo de software, y sobre todo su aparición como área de conocimiento de la ingeniería del software en la guía del cuerpo de conocimiento de la ingeniería del software [1] provocaron que empezaran a aparecer cursos de formación en pruebas de software con distintos formatos. En una primera instancia fueron experiencias puntuales en másteres y titulaciones propias (véase, por ejemplo, el curso de experto Ingeniería de Pruebas y Testing del Software de la Universidad de Sevilla [3] o la asignatura *Pruebas y Control de Calidad* dentro del máster de desarrollo de aplicaciones y servicios web de la Universidad de Alicante[1] o la asignatura *Testing y Calidad* en el Máster en Ingeniería Web y Tecnologías RIA de la Universidad de Málaga[2].

La situación cambió drásticamente con los cambios de planes de estudio llevados a cabo para adaptar las titulaciones ofertadas por las universidades españolas al espacio europeo de educación superior. La flexibilidad introducida en la nueva legislación ha hecho, sin embargo, que la introducción de las pruebas de software se haya hecho de forma muy desigual en las distintas universidades. Así, además de la variedad en títulos de grado, nos encontramos con unas universidades que ofrecen asignaturas sobre pruebas como optativas y otras como obligatorias, en unas donde encontramos asignaturas dedicadas por entero a las pruebas de software, otras donde se combina el estudio de pruebas y mantenimiento, otras donde aparece combinado con algún método formal de especificación y análisis, y otras donde no aparece como una asignatura, sino transversalmente en varias. A modo de ejemplo, véase la asignatura optativa del Grado en Ingeniería Informática de la Universidad de Alcalá de Henares *Calidad, pruebas y mantenimiento del software*[3], la optativa del Grado en Ingeniería del Software de la Universidad Complutense de Madrid *Especificación, validación y*

[1] http://cv1.cpd.ua.es/consplanesestudio/cvFichaAsiEEES.asp?wCodAsi=38204
[2] http://riatec.lcc.uma.es/
[3] http://www.uah.es/estudios/asignaturas/programas/G780/780043_G780_2014-15.pdf

Andorra La Vella, del 8 al 10 de julio 2015
ISBN: 978-99920-70-10-9

testing[4], la obligatoria de los Grados en Ingeniería Informática e Ingeniería del Software de la Universidad de Sevilla *Diseño y Pruebas*[5] o la obligatoria *Ingeniería del Software I* de la Facultad de Informática de la Universidad Politécnica de Cataluña que incluye un tema sobre el diseño de pruebas a partir de la especificación de sistemas software[6].

La E.T.S.I. Informática de la Universidad de Málaga ofrece Grados en Ingeniería del Software, en Ingeniería de Computadores y en Ingeniería Informática, este último con menciones en Sistemas de Información, Tecnología de la Información y Computación. Estos tres grados ofrecen una asignatura obligatoria en segundo curso *Introducción a la Ingeniería del Software* en la que se presentan contenidos básicos sobre pruebas. En el Grado en Ingeniería del Software se ofrecen, entre otras asignaturas, y continuando con la formación en la validación y verificación de software, *Mantenimiento y Pruebas de Software* y *Métodos Formales en Ingeniería del Software*.

En la asignatura sobre pruebas se presenta un abanico de técnicas de prueba y los fundamentos teóricos tras ellas. Uno de los objetivos principales de la asignatura es que los alumnos sean capaces de seleccionar una estrategia de pruebas adecuada, definir casos de prueba adecuados y formular hipótesis de corrección de los mismos. Aunque son muchos los conceptos y contenido teórico que se imparten en la asignatura, esta es principalmente práctica. La necesidad de llevar a cabo todo tipo de pruebas ha provocado que existan una gran cantidad de herramientas tanto de uso público como comercial que implementan prácticamente todas las técnicas a utilizar. En entornos como el nuestro, en que la formación práctica es fundamental, inmediatamente surge la necesidad de establecer mecanismos de valoración del trabajo realizado. La evaluación de los alumnos es importante, por supuesto, pero cara al proceso formativo el poder darle una corrección del trabajo realizado es fundamental para que el alumno sepa si puede seguir como va o ha de cambiar su forma de realizar las tareas. Mucho se ha dicho sobre esto en el pasado en distintos contextos. En formación en programación es cada vez más habitual el disponer de sistemas de evaluación automáticos (véanse, por ejemplo, [5, 12]) o incluso herramientas on-line como las descritas en [9, 6, 13]). De hecho, muchas de estas herramientas están basadas en pruebas de unidad, utilizando JUnit o TestNG para ejecutarlas. Estas herramientas pueden además indicar con precisión qué es lo que produjo el fallo, lo cual puede ayudar al alumno a corregirlo.

En torno a las pruebas de software se han desarrollado una gran cantidad de medidas que permiten saber, con diferentes niveles de precisión, cuán buenas son las pruebas realizadas. Para el alumno, el obtener medidas que le digan que sus pruebas ejecutan el 90 % del código probado o el 30 % de las condiciones le sirve como una motivación adicional para conocer mejor dichas medidas de cobertura y para emplearlas mejor a la hora de desarrollar sus pruebas. Por supuesto, estas medidas no son la solución a todos los problemas, hay muchas situaciones que no quedarán reflejadas en ninguno de estos indicadores, pero la cuestión es que estos indicadores son útiles para su formación, de la misma forma que lo son para su evaluación o para el seguimiento de cualquier actividad. Una técnica de gran actualidad para dar una estimación de la bondad de nuestros conjuntos de pruebas es la técnica de los mutantes, que básicamente consiste en inyectar pequeñas modificaciones en el código fuente del software al que se aplican las pruebas para comprobar si nuestro banco de pruebas es capaz de detectarlas. Estas modificaciones son errores introducidos a propósito, pero que nuestras pruebas deberían ser capaces de detectar. Cuanto mejores sean nuestras pruebas más mutantes serán capaces de identificar como erróneos. Macario Polo y Pedro Reales presentaron en [10] una propuesta pedagógica para la enseñanza de los mutantes mediante una metáfora, en concreto comparar los mutantes con faltas de ortografía, y los conjuntos de casos de prueba con revisores ortotipográficos.

En este trabajo presentamos cómo hemos utilizado la técnica de generación de mutantes para evaluar la bondad de los conjuntos de casos de prueba aportados por los alumnos. El número de mutantes eliminados nos da un indicador muy útil para la valoración de las prácticas, lo cual puede permitir aumentar el número de prácticas valoradas de cada alumno, pero más importante aún, proporcionar una respuesta más rápida al alumno. Por último, un análisis de los mutantes que no se han podido eliminar permiten localizar con exactitud el fragmento de código cuya modificación no ha sido detectada por las pruebas, sirviendo de indicador de los casos de prueba que es necesario añadir para completarlo. Por supuesto, esta técnica tiene sus limitaciones. Dado el tipo de herramientas disponibles, la técnica está limitada a pruebas de unidad de caja blanca.

Tras dar una descripción más precisa de la asignatura en la sección 2 y una breve introducción a la técnica de pruebas basada en mutantes en la sección 3, presentamos la metodología utilizada en la asignatura relacionada con el presente trabajo en la sección 4, los resultados obtenidos en la sección 5 y una discusión sobre estos en la sección 6. El trabajo termina con unas

[4] http://www.fdi.ucm.es/Pub/ImpresoFichaDocente.aspx?Id=510

[5] http://www.us.es/estudios/grados/plan_205/asignatura_2050018

[6] http://www.fib.upc.edu/es/estudiar-enginyeria-informatica/enginyeries-pla-2003/assignatures/ES1.html

Andorra La Vella, del 8 al 10 de julio 2015
ISBN: 978-99920-70-10-9

conclusiones e ideas de trabajo futuro en la sección 7.

2. La asignatura

Los contenidos de la asignatura *Mantenimiento y Pruebas del Software* de la E.T.S.I. Informática de la Universidad de Málaga recogen las técnicas habituales de realización de pruebas, incluyendo pruebas de caja negra (particionamiento de valores, modelos combinatorios, grafos de transición de estados) y de caja blanca (revisiones formales, análisis de flujo de control y de flujo de datos, pruebas basadas en mutantes), pruebas de unidad, de integración y de sistema, pruebas de rendimiento, de estrés, de configuración, pruebas de regresión y pruebas de aceptación. Se hace un especial énfasis en las pruebas de sistemas orientados a objetos y sus particularidades, en la generación de pruebas a partir de modelos UML, y con atención a la existencia de estándares para la planificación y gestión del proceso de elaboración y ejecución de pruebas.

En concreto, en nuestra asignatura desarrollamos pruebas para Java en el entorno de Eclipse, y las herramientas que utilizamos son JUnit para pruebas de unidad, Mockito para pruebas de integración, EclEmma para medidas de la cobertura de pruebas, μJava para generación de mutantes y FEST para pruebas de interfaces gráficas de usuario. La elección de cualquiera de estas herramientas es discutible, pero el contexto en el que se imparte, donde se apuesta por Java como lenguaje vehicular, donde los alumnos están familiarizados con Eclipse, no deja muchas alternativas. El resto de las herramientas son herramientas de fácil acceso y que permiten poner en práctica conceptos y técnicas similares a los que encontramos en otras herramientas más potentes.

La asignatura se imparte en 13 semanas, con dos clases de 1 hora 45 minutos cada una. Se realizan 7 prácticas a lo largo del curso, cada una de las cuales les permite familiarizarse con una nueva técnica o herramienta. Cada práctica consta de una pre-tarea, a realizar antes de la sesión de laboratorio, una tarea a realizar en el laboratorio, y una post-tarea, a realizar con posterioridad. Las pre-tareas están diseñadas de forma que los alumnos lleguen al laboratorio en condiciones de realizar una práctica útil aprovechando el tiempo del que disponen. La post-tarea consiste en algún ejercicio adicional, más la compleción de la tarea, que es lo corregido y registrado para la evaluación final.

Las prácticas son realizadas en parejas, que forman ellos para cada práctica. Solo se les ponen dos condiciones: que no repitan pareja, cada práctica la realizarán con un compañero diferente, y que no se empareje un alumno que no ha realizado la pre-tarea con otro que sí la ha hecho. La primera restricción les ayuda a adaptarse a trabajar con personas distintas, sacándoles de su área de confort y obligándoles a ser autónomos. Aunque los dos autores de cada práctica obtienen la misma calificación, la variación de parejas permite minimizar bastante bien el impacto que una mala experiencia pueda tener. La segunda restricción funciona sorprendentemente bien como motivación para la realización de las pre-tareas, a la vez que garantiza el no perjudicar a los que sí lo han hecho y permitirles aprovechar mejor el tiempo de laboratorio. La experiencia de que dos personas que no han preparado la práctica es tan negativa que si algún alumno por cualquier circunstancia no ha podido realizarla se va a asegurar de que no vuelva a suceder. De hecho, en los dos años que se ha utilizado este mecanismo, solo en la primera clase hay algún alumno que no la realiza.

De las 7 prácticas que componen la asignatura, utilizamos los mutantes para valorar las tres primeras: una sobre triángulos y colas, la segunda para probar un paquete estadístico y una tercera en la que se prueba una implementación de árboles AVL. Realizamos una revisión manual de las prácticas solo en el paquete estadístico y en la implementación de árboles AVL, puesto que la primera práctica tenía como objetivo recordar los conocimientos de JUnit que los alumnos habían adquirido en una asignatura previa, y su evaluación no tenía mucho valor para la asignatura de pruebas.

3. Mutation testing

Las pruebas basadas en mutación, o *mutation testing*, es un enfoque propuesto por DeMillo, Lipton y Sayward [2] en 1978 para evaluar la calidad de un conjunto de casos de prueba. La idea consiste en crear ligeras variaciones, llamadas *mutantes*, del código a probar. Después se ejecutan los casos de prueba sobre el programa original y los mutantes y se cuenta el número de mutantes para los que la salida obtenida difiere de la del programa original (mutante muerto). Cuantos más mutantes sean estos mejor es el conjunto de casos de prueba. A continuación realizamos una descripción más detallada.

Dado un software a probar P, un mutante m es una copia de dicho software en la que se ha insertado un pequeño cambio. Este cambio se aplica al código fuente del software y normalmente en una sola sentencia del código[7]. Se han propuesto diversos tipos de mutaciones, llamados *operadores de mutación*, entre los que podemos destacar: reemplazo de un operador aritmético usado en una expresión por otro diferente (AOR), reemplazo de un operador relacional por otro diferente (ROR), reemplazo de un operador lógico usado en una condición por otro diferente (LCR) e inserción de un operador unario (UOI).

[7]Los *mutantes de orden superior* introducen cambios en varias sentencias simultáneamente.

Andorra La Vella, del 8 al 10 de julio 2015
ISBN: 978-99920-70-10-9

```
double normaInf(double x, double y) {
    return Math.abs(x)+Math.abs(y);
}
```

(a) Código original

```
double normaInf(double x, double y) {
    return Math.abs(x)-Math.abs(y);
}
```

(b) Mutante AOR

Figura 1: Fragmento de código original y un mutante de tipo AOR.

Estas mutaciones pretenden modelar los pequeños errores que un programador puede cometer al escribir el código. En la Figura 1 podemos ver un ejemplo de mutación de tipo AOR. Cada uno de los operadores de mutación se aplica a todas las sentencias donde sea posible y se genera el correspondiente *mutante*. De esta forma, a partir de un código de algunas decenas de líneas de código es posible generar cientos de mutantes.

Una vez que se han generado los mutantes, se ejecuta cada caso de prueba sobre el código original y cada uno de los mutantes. Si las salidas obtenidas al ejecutar un caso de prueba t sobre el código original y el mutante m difieren, decimos que el caso de prueba t *mata* al mutante m. Esta definición podemos extenderla al conjunto T de casos de prueba: si algún caso de prueba $t \in T$ mata al mutante m decimos que el conjunto T mata a m.

Un conjunto de casos de prueba bien diseñado no debería tener problemas para distinguir entre los mutantes y el programa original y, por tanto, debería ser capaz de matar a muchos de los mutantes. De aquí que cuantos más mutantes mate el conjunto de casos de prueba mejor se asume que es la calidad del mismo. Sin embargo, es posible que algunos mutantes queden siempre *vivos* tras aplicar cualquier conjunto de casos de prueba porque pueden ser *funcionalmente equivalentes* al código original.

Esta equivalencia funcional impide la completa automatización de la medida de calidad del conjunto de casos de prueba. En efecto, comprobar si dos programas son funcionalmente equivalentes o no es un problema indecidible. En la práctica, lo que se hace es una inspección manual del mutante que se sospecha que es funcionalmente equivalente. Si se puede afirmar que es funcionalmente equivalente se marca como tal y no se considera para el cálculo de la calidad del conjunto de casos de prueba. Empíricamente se ha observado que entre el 5 % y el 20 % de los mutantes generados son funcionalmente equivalentes al programa original [14].

Si llamamos M al conjunto de mutantes generados, K a los mutantes que ha matado el conjunto de casos de prueba T y E al conjunto de mutantes funcionalmente equivalentes a P, definimos el *mutation score* como el cociente:

$$\mathcal{MS}(P,T) = \frac{|K|}{|M| - |E|} \quad (1)$$

El *mutation score* es un número entre 0 y 1 que representa la calidad del conjunto de casos de prueba T. Cuanto más alto sea mayor será la calidad de T.

Offutt propuso una estrategia para diseñar casos de prueba basada en los mutantes [8]. Una vez diseñado el primer conjunto de casos de prueba, se ejecuta sobre el programa original y los mutantes tal y como se describió anteriormente. Aquellos mutantes que sobrevivan son inspeccionados para averiguar si son funcionalmente equivalentes al código original. Si es así, el mutante se marca adecuadamente, obviándolo en sucesivas ejecuciones del conjunto de casos de prueba. Si no es funcionalmente equivalente se diseña un caso de prueba que sea capaz de distinguir a dicho mutante del código original. Este caso de prueba se añade al conjunto de casos de prueba. Tras haber analizado cada uno de los mutantes supervivientes, se vuelve a ejecutar el conjunto de casos de prueba sobre ellos y se repite todo el proceso. En cualquier momento podría detectarse un error en el código original, en cuyo caso hay que volver a generar todos los mutantes tras modificar el código y empezar de nuevo.

Existen diversas herramientas de apoyo a *mutation testing*. Algunos ejemplos son μJava de Ma et al. [7], Bacterio de Reales et al. [11] y Evosuite de Arcuri y Fraser [4]. Nosotros optamos por μJava porque trabaja sobre código Java y pruebas en JUnit, y sobre todo porque desde hace pocos años posee una interfaz de línea de órdenes que permite crear fácilmente *scripts* de *shell* para aplicar *mutation testing* a las distintas entregas de los alumnos. μJava considera que la salida del código original difiere de la salida del mutante si el resultado del caso de prueba (éxito, fallo o error) es diferente en ambos.

4. Metodología

Durante el curso académico 2013/14 desarrollamos un experimento para evaluar la posible utilidad de las pruebas basadas en mutación como herramienta de evaluación de prácticas de diseño de casos de pruebas unitarias. Para ello nos centramos en las tres primeras tareas que los alumnos debían realizar en nuestra asignatura, como se describe en la Sección 2. En los tres casos se pidió a los alumnos que diseñaran un conjunto de casos de prueba adecuado para probar las clases que los profesores les proporcionaban (en el caso de los triángulos su propia implementación).

La primera tarea, denominada *Triangulo-Cola*, se realiza la primera semana, y su objetivo es que los

Andorra La Vella, del 8 al 10 de julio 2015
ISBN: 978-99920-70-10-9

alumnos refresquen el conocimiento de JUnit de asignaturas anteriores y perciban diversos problemas de no utilizar ninguna metodología para enfrentarse al proceso de pruebas. Esta práctica fue corregida automáticamente pero no hubo una corrección por parte del profesor aparte de los comentarios dados a los alumnos en el propio laboratorio durante su realización.

El objetivo de la segunda práctica, denominada *StatsNumber*, es poner en práctica el concepto de cobertura de código y sus distintos tipos, previamente presentado en clase de teoría, y utilizar EclEmma para medirla, y utilizarla para maximizar con sus pruebas la cobertura del código a probar. Los profesores proporcionaron una clase Java que tenía una serie de métodos para realizar cálculos estadísticos.

El objetivo de la tercera práctica, llamada *AVL*, es llevar a cabo un análisis funcional del sistema a probar para obtener las pruebas correspondientes. Aunque pueden hacer uso de EclEmma para completar sus pruebas, es importante el análisis previo, y no solo el conjunto de pruebas finalmente obtenido o su cobertura. La implementación proporcionada incluye un par de errores de los que ellos no son conscientes. En la post-tarea se les cuestiona sobre su detección y sobre la complacencia experimentada por suponer que no había errores. La implementación que se les proporciona incluye varias clases (*AB* de árbol binario, *ABB* de árbol binario de búsqueda, *AVL*, *Lista*, *ListaEnlazada*, además de clases auxiliares como nodos e iteradores). A los alumnos se les pidió que centraran sus esfuerzos en las clases que implementaban los árboles AVL y obviaran las clases relacionadas con las listas.

Los alumnos entregaron las soluciones de sus prácticas por parejas, las cuales fueron evaluadas y se les asignó una calificación entre 0 y 10. En el caso de *StatsNumber* esta calificación podía tomar cuatro valores: muy bien (10), bien (8), aceptable (5) e inaceptable (0). En el caso de *AVL*, dada su complejidad, los profesores utilizaron una escala más fina, añadiendo las calificaciones numéricas de 6, 7 y 9 a las mencionadas anteriormente. Adicionalmente, justificaban su calificación con una serie de comentarios para que los alumnos pudieran tener una retroalimentación que les permitiera aprender y mejorar en el futuro.

Posteriormente, los profesores (no los alumnos) utilizaron μJava para generar mutantes para el código fuente proporcionado en cada una de las tareas y calcularon el *mutation score* de cada conjunto de casos de prueba. Su análisis será presentado en la sección 5.

Para finalizar la descripción de la metodología, aclaramos algunos aspectos que es importante tener en cuenta al interpretar los resultados obtenidos:

- La evaluación manual no está sesgada por los *mutation scores* obtenidos. El cálculo del *mutation score* para cada conjunto de casos de prueba se realizó después de la revisión manual de las tareas por parte de los profesores.
- En el caso de *StatsNumber* el número de mutantes generado por μJava fue de 150 y para *AVL* fueron 1315, de donde 485 son mutantes relacionados con la implementación de los árboles y 830 están relacionados con la implementación de las listas. El número de mutantes para la práctica *TrianguloCola* es mucho menor.
- En *StatsNumber* podemos afirmar que ningún mutante era funcionalmente equivalente al código original porque algunos conjuntos de casos de prueba logran matar los 150 mutantes. Sin embargo, en *AVL* quedaron mutantes vivos tras el cálculo del *mutation score* de todas las prácticas entregadas. No sabemos, en este caso, si hay mutantes funcionalmente equivalentes al código original. No obstante, para los objetivos de nuestro estudio empírico este dato es irrelevante, ya que lo que pretendemos comprobar es si existe una correlación positiva entre las notas asignadas por el profesor y el *mutation score* calculado con ayuda de μJava. Por este motivo, calculamos el *mutation score* dividiendo el número de mutantes muertos entre el número total de mutantes generados: $\mathcal{MS}(P,T) = |K|/|M|$.
- μJava proporciona el número de mutantes generados y muertos por cada clase. En el caso de *StatsNumber*, con una sola clase, su *mutation score* se calcula dividiendo estos dos números. Sin embargo, en *AVL* no tiene sentido analizar los mutantes relacionados con las clases que implementan las listas, ya que al alumno se le dieron instrucciones de probar solamente el árbol AVL. En este caso el incluir todas las clases en el cálculo introduce cierto ruido en el *mutation score*. En este caso hemos calculado el *mutation score* de dos formas. En la primera solo hemos tenido en cuenta los mutantes de la clase AVL, mientras que en la segunda hemos considerado para el cálculo los de todas las clases relacionadas con la implementación de los árboles.
- Como se ha descrito en la sección 2, para cada tarea las parejas de alumnos son distintas. Entendemos que esta práctica, propia de la metodología aplicada en la asignatura, no tiene influencia en los resultados, ya que nunca relacionamos los resultados obtenidos en distintas tareas.

5. Resultados

En esta sección presentamos los resultados obtenidos en el experimento, primero analizaremos los resultados obtenidos para *StatsNumber* y a continuación para *AVL*.

Andorra La Vella, del 8 al 10 de julio 2015
ISBN: 978-99920-70-10-9

Pareja	Nota	Mutation score
P1	0.0	-
P2	0.0	-
P3	0.5	0.75
P4	0.5	0.85
P5	0.5	0.92
P6	0.5	0.92
P7	0.5	0.93
P8	0.5	0.93
P9	0.5	0.95
P10	0.8	0.71
P11	0.8	-
P12	0.8	0.92
P13	0.8	0.93
P14	0.8	0.94
P15	0.8	0.94
P16	0.8	0.95
P17	1.0	0.95
P18	1.0	0.95
P19	1.0	1.00

Cuadro 1: Notas normalizadas y *mutation score* obtenido por cada una de las entregas de las parejas de alumnos para la tarea *StatsNumber*.

5.1. *StatsNumber*

En el Cuadro 1 mostramos para cada una de las 19 parejas de alumnos la nota normalizada obtenida en la corrección por parte del profesor y el *mutation score* correspondiente. La nota normalizada es simplemente el resultado de dividir la nota del alumno entre la calificación máxima (10) para que resulte un número entre 0 y 1, tal y como ocurre con el *mutation score*. Para facilitar su análisis, se han ordenado las entradas del cuadro de acuerdo a la nota, y las entradas con la misma nota atendiendo al *mutation score*.

Hemos marcado en el cuadro con un guión en *mutation score* aquellas entregas que μJava no fue capaz de ejecutar. La razón es que dichas entregas no cumplían las especificaciones de la tarea. Los alumnos que realizaron esas entregas no respetaron el código entregado por el profesor, renombrando las clases en algún caso o creando clases nuevas en otros. Puede observarse que en el caso de las entregas P1 y P2, dicho incumplimiento de la interfaz fue considerado por los profesores como inaceptable.

Observamos la tendencia general de que las entregas con *mutation score* más alto son también aquellas a las que se le asignó una nota más alta. Existe una excepción notable en la entrega P10, donde la nota es 0.8 y el *mutation score* es de 0.71, mientras que el resto de entregas con 0.8 de nota tienen un *mutation score* por encima de 0.90.

También podemos apreciar que el *mutation score* en este caso no es capaz de distinguir adecuadamente entre las dos notas intermedias. Las entregas con valores en torno al 0.92 de *mutation score* parecen estar repartidas indistintamente entre el 0.5 y el 0.8 de nota.

Para poder interpretar adecuadamente estos resultados tenemos que tener presente que el objetivo de la práctica era maximizar los niveles de distintos criterios de cobertura utilizando EclEmma, y que la técnica de mutantes no deja de ser un criterio de cobertura adicional (que los alumnos no usaban). Con una implementación sencilla como esta, simplemente alcanzando un 100 % de cobertura en líneas de código se va a obtener un valor muy alto en el *mutation score*. Tenemos que tener en cuenta también que la calificación dada por el profesor no solo considera la cobertura, sino también el estilo de programación, la existencia de pruebas redundantes, etc.

Por último, a pesar de la tendencia observada en el cuadro, la dispersión de los valores hace que la correlación entre ellos sea mínima. Por ejemplo, el coeficiente de correlación de Pearson para los datos del cuadro se sitúa en torno al 0.08.

5.2. *AVL*

En el Cuadro 2 mostramos para cada una de las 17 parejas de alumnos la nota normalizada obtenida y el *mutation score* para la clase AVL y para todas las clases relacionadas con los árboles. Se han ordenado las entradas del cuadro de acuerdo a la nota, en primer lugar, y al *mutation score* de la clase AVL en segundo. Para facilitar la observación de los datos, se muestran también en la Figura 2.

Al igual que ocurría con la tarea *StatsNumber*, en esta observamos una tendencia de los dos *mutation scores* a aumentar con la nota. En este caso dicha tendencia resulta más clara que en *StatsNumber* y, de hecho, el coeficiente de correlación de Pearson es del orden de 0.30. También observamos que los dos valores de *mutation score* crecen simultáneamente, salvo por algunas excepciones en las entregas P5-P6, P9-P10 y P12-P13.

El valor del *mutation score* para los árboles resulta más bajo que el de la clase AVL. Esto no debe resultar sorprendente, ya que los alumnos se centraron en probar la funcionalidad del árbol AVL, y muchos de los métodos de las otras clases auxiliares pudieron quedar sin probar o incluso sin ejecutar durante las pruebas, permitiendo que muchos de los mutantes asociados a sentencias de dichos métodos sobrevivieran.

6. Discusión

En esta sección vamos a discutir sobre la validez del experimento, sobre el uso práctico de los mutantes en el contexto de la formación en pruebas de software,

Andorra La Vella, del 8 al 10 de julio 2015
ISBN: 978-99920-70-10-9

Pareja	Nota	$\mathcal{MS}$ AVL	$\mathcal{MS}$ árboles
P1	0.0	-	-
P2	0.5	0.59	0.34
P3	0.5	0.76	0.41
P4	0.6	-	-
P5	0.6	0.62	0.42
P6	0.6	0.63	0.30
P7	0.7	0.55	0.35
P8	0.7	0.58	0.41
P9	0.7	0.67	0.47
P10	0.7	0.72	0.45
P11	0.8	0.66	0.46
P12	0.8	0.71	0.49
P13	0.8	0.82	0.37
P14	0.9	0.66	0.29
P15	0.9	0.74	0.50
P16	0.9	0.74	0.50
P17	0.9	0.82	0.56

Cuadro 2: Notas normalizadas y *mutation score* obtenido para AVL y todas las clases de la implementación de árboles por cada una de las entregas de las parejas de alumnos para la tarea *AVL*.

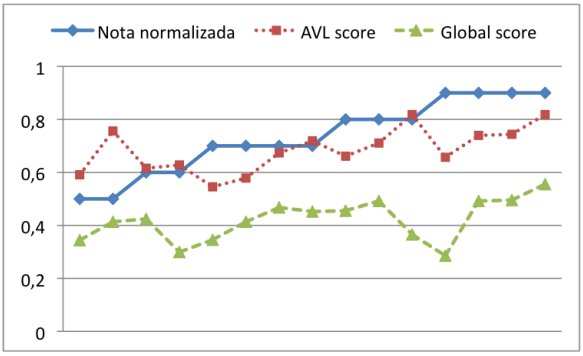

Figura 2: *Mutation score* de AVL y árboles frente a la nota normalizada para *AVL*.

sobre las dificultades encontradas y, por último, sobre las herramientas desarrolladas y utilizadas para su automatización.

6.1. Validez del experimento

La muestra utilizada para el experimento es demasiado pequeña como para sacar conclusiones refrendadas estadísticamente. Debemos ser cautos a la hora de confiar en la correlación observada. Por otro lado, también hay que tener en cuenta que los valores comparados están midiendo cosas distintas: mientras los *mutation scores* miden la cobertura de las pruebas y su efectividad, los profesores han valorado además otras cuestiones, como la correcta utilización de las diferentes técnicas para llegar a esos conjuntos de pruebas, la elegancia y estilo de la codificación, etc. Esto justifica, al menos en parte, que las correlaciones no sean altas.

6.2. Uso práctico

En principio descartamos el uso del *mutation score* para asignar una calificación automáticamente a los alumnos, pero el análisis de cada uno de los casos por separado nos permite ratificarnos en nuestra hipótesis de partida: el uso de la técnica de los mutantes nos sirve como un indicador muy informativo a la hora de valorar el trabajo realizado.

No podemos olvidar la utilidad cara al estudiante. El hecho de que se trate de una medida automatizada permite al alumno obtener una respuesta inmediata que puede ayudarle a completar sus conjuntos de pruebas, de forma similar a como usa otras medidas de cobertura, y en general su formación, no solo a la hora de generar buenos conjuntos de pruebas, sino también a la hora de entender la técnica de los mutantes en sí y su uso práctico.

6.3. Dificultades encontradas

Aunque existen herramientas más potentes, μJava tiene una versión ejecutable en línea de órdenes que hacía especialmente atractivo su uso cara a la tarea del profesor, pues permitía automatizar su ejecución utilizando scripts (véase la sección 6.4). Sin embargo, μJava presenta limitaciones importantes que han dificultado el procesamiento de las entregas de los alumnos para poder obtener los *mutation scores*.

En primer lugar, μJava solo puede generar mutantes y ejecutar estos mutantes para clases que se encuentran en el paquete por defecto, lo cual afecta al código proporcionado por el profesor y al de los alumnos. Por otro lado, hay ciertas características del lenguaje Java que μJava no soporta a la hora de generar mutantes pero pueden usarse en las pruebas. En particular, μJava no soporta clases anidadas, no permite restricciones de parámetros, ni la definición de clases abstractas o interfaces. Estas restricciones nos obligaron a modificar el código de las clases dadas a los alumnos. Sin embargo, dado que las modificaciones necesarias eran "poco elegantes", optamos por utilizar dos versiones distintas, una proporcionada a los alumnos en Java y otra modificada equivalente y con la misma interfaz utilizada solo para la generación de los mutantes. Obsérvese que los cambios realizados no afectaban ni a las pruebas ni a los mutantes. Por ejemplo, una clase anidada estática privada (p.ej., una clase *Nodo* de la clase *AB*) se define como una clase normal visible dentro del paquete, o una clase abstracta se define como una clase no abstracta en la que los métodos abstractos son reemplazados por implementaciones por defecto de estos.

Andorra La Vella, del 8 al 10 de julio 2015
ISBN: 978-99920-70-10-9

Para resolver el problema de los paquetes compilamos las clases de los alumnos usando el código original de los profesores en el *classpath*. Dado que μJava no requiere tener el código fuente de las clases de prueba, sino solo los ficheros *.class*, una vez compiladas las clases de pruebas utilizamos un pequeño programa para modificar los ficheros *.class* cambiando toda referencia a las clases a probar, al mismo tiempo que ubicaba en el paquete por defecto las clases de prueba.

6.4. Script

Para poder simplificar la labor de calcular el *mutation score* obtenido por las pruebas de los alumnos, hemos desarrollado un *script* de *shell* que permite hacer *mutation testing* usando μJava proporcionando: (1) el código a probar (el código fuente original proporcionado por el profesor sobre el que los alumnos han diseñado sus casos de prueba), (2) una versión del código original que respeta la interfaz hacia los alumnos pero que no incluya ninguna de las características del lenguaje Java no soportada por μJava, y (3) una carpeta con una subcarpeta por cada alumno conteniendo el código Java de sus pruebas implementadas en JUnit.

7. Conclusiones y trabajo futuro

En este trabajo proponemos el uso de la técnica de los mutantes en asignaturas de pruebas de software para ayudar en la labor de evaluación de las mismas, para ayudar a los alumnos estimando la calidad de sus conjuntos de casos de prueba, y para ayudarles a entender y asimilar la técnica en sí.

Nuestra experiencia muestra que los resultados automatizados sirven como un indicador muy informativo de la calidad de las pruebas que puede servir para guiar/confirmar al profesor en su evaluación.

Nos gustaría mejorar la automatización de la evaluación. Sobre todo cara a la formación alumno. Las restricciones de μJava nos hacen dudar que sea la herramienta más adecuada. Entre las alternativas que contemplamos está contactar con los desarrolladores de μJava para ofrecernos a ayudarles en la mejora de su herramienta, y contactar con desarrolladores de otras herramientas para estudiar la posibilidad de hacer un uso similar al que realizamos con μJava.

Agradecimientos

Este trabajo ha sido parcialmente financiado por Andalucía Tech, Campus de Excelencia Internacional, y la Universidad de Málaga.

Referencias

[1] Bourque, Pierre y Robert Dupuis: *Guide to the Software Engineering Body of Knowledge (SWEBOK)*. IEEE Computer Society, 2004.

[2] DeMillo, R. A., R. J. Lipton y F. G. Sayward: *Hints on test data selection: Help for the practicing programmer*. IEEE Computer, 11(4):34–41, 1978.

[3] Escalona, M.J., Tanja E.J. Vos y J.J. Gutiérrez: *Pruebas de software en la enseñanza universitaria de la informática: un título propio*. En *Actas XVIII JENUI*, páginas 409–412, 2012.

[4] Fraser, Gordon y Andrea Arcuri: *Achieving scalable mutation-based generation of whole test suites*. Empirical Software Engineering, páginas 1–30, 2014.

[5] Higgins, Colin A., Geoffrey Gray, Pavlos Symeonidis y Athanasios Tsintsifas: *Automated assessment and experiences of teaching programming*. ACM Journal of Educational Resources in Computing, 5(3):1–21, 2005.

[6] Joy, Mike, Nathan Griffiths y Russell Boyatt: *The BOSS on-line submission and assessment systems*. ACM Journal on Educational Resources in Computing, 5(3), 2005.

[7] Ma, Yu Seung, Jeff Offutt y Yong Rae Kwon: *MuJava: An Automated Class Mutation System*. Journal of Software Testing, Verification and Reliability, 15(2):97–133, June 2005.

[8] Offutt, A. J.: *A Practical System for Mutation Testing: Help for the Common Programmer*. En *Intl. Test Conference*, páginas 824–830, 1994.

[9] Pino, J.C. Rodríguez del, M. Díaz Roca y Z. Hernández Figueroa: *Hacia la evaluación continua automática de prácticas de programación*. En *Actas de XIII JENUI*, páginas 179–186, 2007.

[10] Polo Usaola, Macario y Pedro Reales Mateo: *Enseñanza de la mutación en pruebas software*. En *Actas de XVIII JENUI*, páginas 1–8, 2012.

[11] Reales Mateo, Pedro y Marcario Polo Usaola: *Bacterio: Java mutation testing tool: A framework to evaluate quality of tests cases*. En *Procs. of ICSM*, páginas 646–649, 2012.

[12] Saikkonen, Riku, Lauri Malmi y Ari Korhonen: *Fully Automatic Assessment of Programming Exercises*. En *Proceedings of ITiCSE*, páginas 133–136, 2001.

[13] Surrell, Joan, Imma Boada, Josep Soler, Ferran Prados y Jordi Poch: *Corrección automática de ejercicios de estructuras de datos a través de una plataforma de e-learning*. En *Actas de XVII JENUI*, páginas 75–82, 2011.

[14] Umar, Maryam: *An evaluation of mutation operators for equivalent mutants*. Tesis de Licenciatura, King's College, London, 2006.