Actas de las Jenui, vol. 5. 2020.
Páginas: 149-156

Diseño y desarrollo de prácticas coevaluables y autoevaluables para la etapa de prueba del desarrollo de software

Pedro Delgado-Pérez
Escuela Superior de Ingeniería
Universidad de Cádiz
Av. de la Universidad de Cádiz
10, 11519 - Puerto Real
pedro.delgado@uca.es

Inmaculada Medina-Bulo
Escuela Superior de Ingeniería
Universidad de Cádiz
Av. de la Universidad de Cádiz
10, 11519 - Puerto Real
inmaculada.medina@uca.es

Miguel Ángel Álvarez-García
Escuela Superior de Ingeniería
Universidad de Cádiz
Av. de la Universidad de Cádiz
10, 11519 - Puerto Real
miguelangel.alvarezgarcia
@mail.uca.es

Resumen

La prueba del software es una etapa clave en el desarrollo de programas. A pesar de su importancia, y que el coste asociado a defectos en los programas está en relación a su complejidad, esta fase no suele recibir la atención necesaria. En estudios de ingeniería del software, las actividades de diseño y programación tienen una presencia dominante, provocando que los alumnos puedan ver la prueba de software más como una carga que como un beneficio. En este artículo, presentamos una experiencia para hacer más conscientes a los alumnos del valor de desarrollar pruebas de calidad, usando técnicas y herramientas especializadas en comparación con una generación de pruebas manual y sin guías. Para ello, diseñamos unas prácticas que enfrentan la percepción del alumno sobre la adecuación de sus pruebas contra una evaluación automática basada en la prueba de mutaciones (técnica indicativa de la capacidad de detección de fallos de la batería de pruebas). Como resultado, la práctica cumple su objetivo, pues se observa un descenso de las valoraciones subjetivas que los alumnos otorgan a sus propios conjuntos de pruebas (autoevaluación) y a los de sus compañeros (coevaluación) tras conocer un resultado más objetivo como el de la prueba de mutaciones.

Abstract

Software testing is a key phase in the development of programs. Despite its importance, and the fact that the cost associated with software defects increases as programs become more complex, this phase often does not receive the necessary attention. In the context of software engineering studies, design and coding activities have a predominant presence; as a result, students may tend to see software testing as a burden rather than a benefit for the final quality of the product. In this paper, we present the experience carried out to try to raise student awareness about the value of creating quality software tests, using specialized techniques and tools in comparison with manual and unguided development. To this end, we designed a practice that confronts the student's perception of the adequacy of their tests against an automatic assessment based on the technique known as mutation testing (which is indicative of the fault detection ability of the test suite). In general, the practice meets its goals. We observe a decrease in the subjective assessments regarding their own test suites (self-assessment) and regarding those of their peers (co-assessment) after knowing the more objective result of mutation analysis.

Palabras clave

Prueba de software, prueba de mutaciones, autoevaluación, coevaluación.

1. Introducción y Motivación

Las pruebas del software son un conjunto de actividades que tienen por objetivo evaluar e incrementar la calidad de los proyectos software. Dependiendo del enfoque, podemos encontrar gran diversidad de tipos de pruebas, según si verifican propiedades funcionales o no funcionales, si son automáticas o manuales, si se conoce el código de la aplicación (caja blanca) o no (caja negra), etc. La importancia de esta fase dentro del ciclo de vida del software es mayor de la que habitualmente se le otorga. Tal y como sugiere Myers y Badgett [7], la fase de pruebas puede suponer hasta el 50 % del coste total de un proyecto software. De forma general, el esfuerzo en corregir defectos en el código

aumenta de forma exponencial cuanto más avanzada sea la fase en la que se encuentre el desarrollo.

A pesar de ser una etapa fundamental en el desarrollo del software, rara vez se exige a los alumnos durante sus estudios que dediquen un mínimo esfuerzo a probar sus desarrollos hasta alcanzar un determinado nivel de cobertura sobre el código. Mientras diseñar pruebas para ejercicios de programación en el ámbito del Grado puede no tener gran relevancia, saber abordar esas pruebas cuando nos trasladamos a un ámbito empresarial se vuelve en un asunto vital, especialmente en la industria del software dedicada al desarrollo de sistemas críticos [10]. Es un problema, por tanto, que los alumnos salgan al mercado laboral sin contar con recursos para afrontar esta fase o, peor aún, sin darle la importancia que requiere.

En nuestra Universidad, contamos con una asignatura llamada *Verificación y Validación del Software*, que puede ser cursada tanto por alumnos de tercer como de cuarto curso. Esta asignatura está enfocada por completo a enseñar a los alumnos técnicas y métodos diversos para que adquieran habilidades a la hora de encarar la fase de pruebas. No obstante, a pesar de todos los conceptos, técnicas y herramientas que se les presenta, notamos que los alumnos en cursos previos no terminan de interiorizar el porqué de la importancia de aplicar estas técnicas avanzadas a diferencia de implementar pruebas de una forma manual y sin tener en cuenta medidas de cobertura. Esta es la motivación que nos lleva a desarrollar unas prácticas dedicadas a mejorar este aspecto y que nos permitan confirmar esta hipótesis.

La práctica diseñada se divide en dos partes. En la primera, los alumnos deben diseñar un conjunto de pruebas para un código dado, y evaluar tanto sus pruebas como las de otros compañeros bajo ciertos criterios de diseño y representatividad. La propuesta de aplicar autoevaluación y evaluación entre compañeros tiene por objetivo que el alumno tenga un espacio de reflexión sobre su propio proceso [17], y que encuentre en la comparación con el desarrollo de otro compañero un punto de referencia para apreciar si su propia valoración está bien conmensurada. Posteriormente, en la segunda parte, se aplica la técnica conocida como *prueba de mutaciones* [13], que lleva a cabo una evaluación automática que actuará como juez imparcial, aportando una visión más realista de la facultad que tienen las pruebas diseñadas por el alumno en la detección de potenciales defectos en el código, que es el objetivo final de la creación de las pruebas. De esta manera, buscamos que la puntuación obtenida en esta evaluación automática no quede como un mero número, sino que adquiera significado al compararla con la puntuación derivada de su propia apreciación.

Los resultados de la experiencia durante dos cursos académicos muestran que, en primer lugar, hay un alto componente subjetivo en la apreciación que los alumnos tienen de las pruebas desarrolladas, con una tendencia a valorarse mejor que a los compañeros. En segundo lugar, se observa un claro descenso de estas valoraciones una vez los alumnos están en conocimiento de la puntuación automática proporcionada por la prueba de mutaciones, que resulta ser muy baja en la mayoría de casos. Este hecho da evidencias de que los alumnos tienen ahora mayor consciencia de la necesidad de emplear métodos de prueba más avanzados si quieren alcanzar unas pruebas de mayor calidad. Por otro lado, también se puede apreciar que los alumnos tienden a centrarse en ciertas características del lenguaje, pasando por alto otras características que igualmente requieren de atención en las pruebas.

En las próximas secciones se comentarán trabajos relacionados previos (Sección 2), se explicará en qué consiste la prueba de mutaciones (Sección 3), se describirá con detalle nuestra propuesta de innovación y la experiencia llevada a cabo (Sección 4), se mostrarán los resultados (Sección 5) y se discutirá su significación (Sección 6) y, finalmente, se ofrecerán las conclusiones del estudio (Sección 7).

2. Antecedentes

Es cada vez una práctica más frecuente que, en cursos de programación, se ofrezcan medios al alumno que le permitan determinar la validez de las soluciones que implementan para los ejercicios propuestos. Dentro de esos medios, suele ser muy habitual que los ejercicios vengan acompañados de un conjunto de datos de entrada y salida que permitan probar si el programa presenta el comportamiento esperado [8, 15]. Sin embargo, resulta menos común que sean los propios alumnos los que tengan que desarrollar esas pruebas siguiendo algún tipo de criterio de cobertura, como en [12], o que se realice algún tipo de valoración sobre las mismas.

Son varios los trabajos en los que el objetivo es la mejora del aprendizaje a través de la integración de los conceptos propios de la prueba de software en cursos de programación y relativos a la ingeniería del software en general [2, 3]. Asimismo, Lemos *et al.* [9] analizan en su estudio los beneficios de enseñar técnicas de prueba de software en la calidad del código desarrollado por los estudiantes.

La aplicación de la prueba de mutaciones en prácticas de programación con C++ fue propuesta por algunos de los autores de este trabajo previamente [5]. Varios son los docentes que en el pasado también han apoyado la introducción de esta técnica en sus clases, como Polo y Reales [14] o Chicano y Durán [1]. En el primero, se describe la fórmula empleada a través de

metáforas para conseguir que los alumnos comprendan en sus clases la utilidad de la prueba de mutaciones. En el segundo estudio, los autores van más allá ofreciendo resultados de la aplicación en clase de esta técnica, concluyendo que la prueba de mutaciones puede servir tanto al profesor como al alumno en la evaluación de los trabajos. Finalmente, Clegg et al. [3] también proponen la introducción de la prueba de mutaciones como un juego en sus clases para hacer más ameno para el alumno el aprendizaje de los conceptos de la prueba de software.

Por otra parte, otros autores señalan la importancia de la autoevaluación en relación con las prácticas de programación [12, 17], y también de la coevaluación, es decir, la evaluación entre pares por parte de los alumnos [16]. Estas técnicas ofrecen un espacio de razonamiento y evaluación sobre lo desarrollado, y pueden aportar una dimensión distinta a la corrección clásica realizada por el profesor. Comentar que en un artículo corto anterior se presentó un resumen de los objetivos del proyecto de innovación y de los resultados observados a nivel general durante el primer año [11]. En este artículo, se describe la experiencia en detalle, extendiendo su aplicación a dos cursos, y se presentan los resultados y el impacto de los mismos en mayor amplitud.

3. Prueba de mutaciones

La prueba de mutaciones es una técnica de prueba de software que busca comprobar la calidad de las pruebas creadas por el desarrollador. Estas pruebas se apoyan en la inserción de pequeñas modificaciones sintácticas en el código fuente de un programa dado, las cuales son denominadas como *mutaciones*. Estas modificaciones sintácticas planteadas se crean en base a unas reglas predefinidas, denominadas como *operadores de mutación*, que se corresponden con categorías de errores típicos que un programador podría cometer. Cabe destacar que existen múltiples operadores de mutación en la literatura, como se pueden observar en [4] o [6], entre otros. Por ejemplo, dado un programa con la sentencia $x = x + 1$, un operador de mutación que actúe sobre los operadores aritméticos podría generar varios mutantes, entre los que se encuentra el mutante $x = x - 1$ donde se cambia el operador suma por el de resta.

Existen diversos tipos de mutantes según sea el resultado de su ejecución. Aquellos mutantes que sí son identificados por el conjunto de casos de prueba al producir una salida distinta al programa original, son denominados como *mutantes muertos*. Por el contrario, aquellos que en su ejecución obtienen los mismo resultados se les denomina como *mutantes vivos*. En ciertas ocasiones, un mutante vivo puede producir la misma salida que el programa original dada cualquier entrada; estos mutantes se conocen como *mutantes equivalentes*. Por otro lado, puede ocurrir que un mutante no pueda ser ejecutado, denominándose en esta ocasión como *mutante inválido* [18].

En la prueba de mutaciones, la calidad del conjunto de casos de pruebas se mide mediante la *puntuación de mutación*, que indica el porcentaje de mutantes muertos con respecto al número de mutantes total (exceptuando los equivalentes). De forma general, cuanto mayor sea la puntuación de mutación, mayor se considera que será la capacidad del conjunto de casos de prueba en la detección de errores reales.

$$PM\ (P,\ C) = \frac{MM}{TM - ME}\ \text{x}\ 100$$

***PM**: Puntuación de mutación*
***P**: Programa bajo prueba*
***C**: Conjunto de casos de prueba*
***MM**: Mutantes muertos*
***TM**: Total de mutantes*
***ME**: Mutantes equivalentes*

4. Propuesta de innovación

4.1. Objetivo

El objetivo principal que se busca con la propuesta de innovación es la de conseguir que los alumnos sean más conscientes de la necesidad de incrementar la calidad de las pruebas que desarrollan, y de la importancia que estas tienen dentro del ciclo de vida de cualquier proyecto software y de los beneficios que pueden reportar a los desarrolladores y usuarios en general. De forma más concreta, se espera que la comparación directa de prácticas manuales y técnicas automáticas como la prueba de mutaciones les sirva para establecer la necesaria diferencia entre ambas. La autoevaluación y coevaluación, por un lado, y el cálculo de la puntuación de mutación, por el otro, buscan servir este propósito.

Es también objetivo de este trabajo ofrecer evidencias de qué características de la programación los alumnos suelen pasar por alto en las pruebas manuales diseñadas. En este caso, será de utilidad el análisis de la cobertura de cada mutante por parte de los alumnos.

En base a estos objetivos, se plantean las siguientes preguntas de investigación:

- **Pregunta 1:** *¿Existen diferencias entre la forma de evaluarse y coevaluar de los alumnos?*
- **Pregunta 2:** *¿Qué efecto produce en la forma de evaluar de los alumnos el conocimiento de la puntuación de mutación?*
- **Pregunta 3:** *¿Existen características de programación en las que los alumnos ponen una mayor o menor atención en sus pruebas?*

4.2. Asignatura

La innovación propuesta se presenta en la asignatura *Verificación y Validación del Software* del Grado de Ingeniería Informática de la Universidad de Cádiz. Esta asignatura de 6 créditos ECTS la cursan los alumnos de tercero de la rama de Ingeniería del Software y también alumnos de otras ramas en cuarto curso como optativa. Entre los resultados esperables de esta asignatura están el conocer la importancia de la verificación y validación del software, los conceptos fundamentales de la verificación y de la prueba del software, diversas herramientas para la prueba del software así como tipos de técnicas de prueba y criterios que se pueden emplear para generar casos de prueba.

La práctica diseñada para este estudio se ha llevado a cabo durante dos cursos consecutivos, contando con la participación de un total de 14 alumnos. No hubo modificaciones en la práctica de un año para otro. Los alumnos cuentan con conocimientos suficientes sobre la prueba de mutaciones, ya que es un concepto que se aborda previamente de forma extensa en las clases de teoría.

4.3. Diseño de las prácticas

La experiencia se proyectó para desarrollarse durante dos sesiones de prácticas de 2'30 horas cada una, cada cual con un enunciado diferente que se presenta a los alumnos al inicio de cada sesión con los pasos que debían seguir.

Sesión 1: Generación y valoración manual del conjunto de casos de prueba Como punto de partida de la primera sesión se proporciona un programa que consiste de dos clases en lenguaje C++. El programa se supone correcto, y se explica a los alumnos que no se trata de identificar ningún problema en el código, sino que el único objetivo es desarrollar un conjunto de pruebas para el programa en base a lo aprendido a lo largo de la asignatura y a unos criterios preestablecidos que se presentarán en la siguiente subsección.

El enunciado de la práctica ofrece unas breves instrucciones iniciales para que puedan abordar la práctica sin problemas sobre cómo funciona la biblioteca de pruebas que se les proporciona, cómo añadir un nuevo caso de prueba y cómo comprobar si un caso de prueba pasa o no con éxito.

Sesión 2: Evaluación automática del conjunto de casos de prueba El punto de partida de la segunda sesión es justamente el conjunto de pruebas que desarrollaron en la primera sesión. En esta práctica se lleva a cabo una evaluación automática basada en la prueba de mutaciones. La generación y ejecución de mutantes se lleva a cabo a partir de la herramienta *MuCPP* [4], que aplica tanto operadores de mutación clásicos como de orientación a objetos a programas en código C++.

El enunciado de la práctica ofrece instrucciones sobre cómo instalar la herramienta, cuáles son las órdenes de ejecución para generar y ejecutar los mutantes contra su conjunto de pruebas, cómo determinar qué mutantes han muerto y cuáles no y, finalmente, cómo calcular la puntuación de mutación.

4.4. Autoevaluación, coevaluación y evaluación automática

Antes de finalizar cada una de las dos sesiones, se les da un tiempo a los alumnos para que puedan evaluarse y valorar también las pruebas desarrolladas por otros compañeros (en ambas sesiones, los compañeros evaluados son los mismos). Los conjuntos de casos de prueba se valoran de acuerdo a los siguientes criterios, cada uno con un peso distinto para la valoración global:

- **Corrección (C1 - 10 %):** Se pasan con éxito todos y cada uno de los casos de prueba.
- **Completitud (C2 - 30 %):** En el sentido de que el conjunto cubre todas las funcionalidades previstas en el código bajo pruebas y ejercita los diferentes escenarios que pueden darse.
- **Diseño (C3 - 15 %):** Cada caso de prueba está diseñado para probar una funcionalidad particular. No existen diferentes casos de prueba que cubren una misma funcionalidad o incluso se solapen.
- **Comprobaciones (C4 - 30 %):** Los casos de prueba cuentan con un número de comprobaciones suficiente para cubrir las diferentes entradas posibles.
- **Claridad (C5 - 15 %):** El estilo de programación es claro, facilitando la comprensión del propósito de cada uno de los casos de prueba y de las comprobaciones que se añaden en ellos.

Los criterios C2 y C4 son los que tienen un mayor peso porque son los que tienen influencia en la capacidad de detección de fallos del conjunto de pruebas, mientras que el C1 es el de menor peso ya que solo se incluye como comprobación de que las pruebas no fallan. Cada criterio se valora en un rango de 1 a 4, y se obtiene una media ponderada final.

Respecto a la evaluación automática, los alumnos han de calcular la puntuación de mutación. El número total de mutantes para el código dado es de 46 provenientes de 15 operadores de mutación distintos. Se informa en la práctica de que 7 de los mutantes son equivalentes, por lo que el conjunto de pruebas desarrollado debería ser capaz de cubrir 39 fallos artificiales para conseguir un 100 % de puntuación.

Alumno	Tests	Sesión	Evaluación	C1	C2	C3	C4	C5	Media	PM
Alumno 1	12	Sesión 1	Autoevaluación	4	3	3	3	4	3.25	36 %
			Coevaluación	4	3	3	4	4	3.55	
		Sesión 2	Autoevaluación	4	2	4	2	4	2.8	
			Coevaluación	4	2	3	3	4	2.95	
Alumno 2	8	Sesión 1	Autoevaluación	4	2	4	4	4	3.4	28 %
			Coevaluación	4	2	3	4	4	3.25	
		Sesión 2	Autoevaluación	4	1	4	4	4	3.1	
			Coevaluación	4	2	3	2	4	2.65	
Alumno 3	6	Sesión 1	Autoevaluación	2	2	4	4	4	3.2	26 %
			Coevaluación	1	2	3	4	3	2.8	
		Sesión 2	Autoevaluación	4	1	4	4	4	3.1	
			Coevaluación	4	2	3	2	4	2.65	
Alumno 4	8	Sesión 1	Autoevaluación	4	3	4	2	4	3.1	33 %
			Coevaluación	1	2	4	4	2	2.8	
		Sesión 2	Autoevaluación	4	2	4	2	4	2.8	
			Coevaluación	4	1	4	4	4	3.1	
Alumno 5	-	Sesión 1	Autoevaluación	3	2	3	2	3	2.4	8 %
			Coevaluación	1	2	4	2	2	2.2	
		Sesión 2	Autoevaluación	-	-	-	-	-	-	
			Coevaluación	1	1	2	3	2	1.9	
Alumno 6	9	Sesión 1	Autoevaluación	3	2	4	3	4	3.0	26 %
			Coevaluación	3	3	4	4	4	3.6	
		Sesión 2	Autoevaluación	3	2	4	4	4	3.3	
			Coevaluación	-	-	-	-	-	-	
Alumno 7	12	Sesión 1	Autoevaluación	4	2	4	2	4	2.8	28 %
			Coevaluación	4	2	3	4	4	3.25	
		Sesión 2	Autoevaluación	4	1	4	2	3	2.35	
			Coevaluación	4	3	2	4	4	3.4	
Alumno 8	9	Sesión 1	Autoevaluación	3	2	4	3	4	3.0	10 %
			Coevaluación	2	3	4	3	4	3.2	
		Sesión 2	Autoevaluación	2	2	4	3	4	2.9	
			Coevaluación	2	1	4	1	3	1.85	
Alumno 9	17	Sesión 1	Autoevaluación	4	4	4	4	4	4.0	49 %
			Coevaluación	4	3	3	3	4	3.25	
		Sesión 2	Autoevaluación	4	3	4	3	4	3.4	
			Coevaluación	4	2	4	3	4	3.1	
Alumno 10	6	Sesión 1	Autoevaluación	4	3	4	1	4	2.8	18 %
			Coevaluación	1	1	3	3	4	2.35	
		Sesión 2	Autoevaluación	4	2	3	1	4	2.35	
			Coevaluación	1	1	3	2	4	2.05	
Alumno 11	9	Sesión 1	Autoevaluación	4	2	4	2	4	2.8	13 %
			Coevaluación	2	4	3	1	3	2.6	
		Sesión 2	Autoevaluación	1	1	3	1	4	1.75	
			Coevaluación	2	2	3	1	3	2.0	
Alumno 12	4	Sesión 1	Autoevaluación	4	1	3	3	4	2.65	23 %
			Coevaluación	3	2	2	2	4	2.4	
		Sesión 2	Autoevaluación	4	1	3	1	4	2.05	
			Coevaluación	2	2	3	2	4	2.45	
Alumno 13	16	Sesión 1	Autoevaluación	4	2	3	2	4	2.65	41 %
			Coevaluación	4	3	3	2	3	2.8	
		Sesión 2	Autoevaluación	4	2	3	2	3	2.5	
			Coevaluación	4	2	3	2	3	2.5	
Alumno 14	13	Sesión 1	Autoevaluación	4	3	3	3	4	3.25	59 %
			Coevaluación	4	2	3	3	2	2.65	
		Sesión 2	Autoevaluación	3	2	3	3	4	2.85	
			Coevaluación	4	3	3	2	2	2.65	

Cuadro 1: Valoraciones de cada alumno en la autoevaluación y coevaluación (la que otro alumno realiza) en la primera y segunda sesión de la práctica, así como su puntuación de mutación. Los datos marcados con (-) se debe a que el alumno 5 no asistió a la segunda sesión.

5. Resultados

El Cuadro 1 muestra los resultados recogidos en clase a partir de la experiencia descrita. En concreto, por cada alumno, se muestra el número de casos de prueba diseñados, su autoevaluación y la evaluación realizada por uno de sus compañeros en ambas sesiones, con la puntuación en cada uno de los 5 criterios fijados. Se recoge la media ponderada de esos criterios y, en la última columna, la puntuación de mutación obtenida tras la ejecución de la herramienta de mutación.

A continuación se analizan diferentes aspectos en base a los resultados de esta tabla:

Autoevaluación inicial: Lo más notorio de la evaluación que los alumnos realizan de sí mismos tras la primera sesión es que, en todos los casos, el resultado está por encima de 2 (siendo en más de la mitad de los casos igual o superior a 3). Esto indica que todos los alumnos aprueban el conjunto de casos de prueba que han desarrollado en base a los criterios establecidos.

Coevaluación inicial: De la evaluación que los alumnos se realizan entre sí, podemos observar que en 9 de los 14 casos el resultado de la coevaluación en la primera sesión es menor que la autovaloración. Resulta curioso observar que en el criterio C1 de corrección, en el que tan solo se evalúa si las pruebas pasan con éxito, hay una reducción de la puntuación en la mitad de casos, incluyendo caídas drásticas de 4 a 1 (alumnos 4 y 10). Esto podría deberse a que algunos alumnos no siguieran de forma exacta los pasos para la correcta ejecución de las pruebas. También hay descensos de puntuación en el criterio C5 de claridad en hasta 6 alumnos (en ningún caso se produce un incremento). Todos estos resultados sugieren que los alumnos se vuelven más críticos al juzgar los desarrollos de sus pares, o que cada uno tiene una interpretación muy diferente sobre la legibilidad de las pruebas.

Puntuación de mutación: Tal y como puede observarse, las puntuaciones de mutación conseguidas son, por lo general, bastante bajas. La mayor puntuación la obtiene el alumno 14 con un 59 %, siendo el único que llega a cubrir el 50 % de los mutantes. Estos datos sin duda favorecen el efecto que se esperaba conseguir con esta práctica, ya que una baja puntuación de mutación revela carencias significativas en las pruebas.

Tal y como ocurría en el estudio de Chicano y Durán [1], podemos observar que las prácticas con puntuaciones de mutación más alta llevan asociada, por lo general, unas valoraciones más altas. No obstante, dada la similitud de valores tanto en puntuación de mutación como en los criterios en el resto de alumnos, tampoco se puede establecer una correlación más allá. También se puede apreciar una tendencia a que las entregas con un mayor número de casos de prueba obtengan una puntuación de mutación mayor, aunque esto no es siempre así; si comparamos el caso de los alumnos 4 y 7 por ejemplo, el segundo obtiene un 28 % de puntuación con 12 casos de pruebas, mientras que el primero alcanza el 33 % con tan solo 8.

Autoevaluación y coevaluación final: Es patente, al comparar la autoevaluación de la primera y la segunda sesión, que el conocimiento de su puntuación de mutación produce un efecto en las valoraciones de los alumnos. Hasta en 12 de los 14 casos se aprecia una descenso de la propia nota, siendo el más crítico el alumno 11 con una bajada desde 2.8 a 1.75. También se produce un descenso en la coevaluación (en 8 casos). De forma generalizada, los criterios donde se observa la bajada de la nota son en el C2 y C4 (completitud y comprobaciones), aunque no siempre es así a pesar de que el código de las pruebas no cambia de una sesión a otra.

Para conocer el grado de cobertura de las características del lenguaje, también analizamos la puntuación de mutación de forma individual para cada operador de mutación (es decir, en qué medida los alumnos fueron capaces de detectar los mutantes generados por cada operador). El Cuadro 2 recoge los operadores de mutación que se aplicaron en la práctica junto con una breve descripción. La tabla se ordena según el promedio de los mutantes detectados por los alumnos de cada operador, dando un peso proporcional a la puntuación global lograda por cada alumno. Según estos datos, ciertas características relacionadas con los constructores, como la inicialización de los atributos (IPC), la presencia de varios constructores (OMR) o el uso de la palabra reservada *this* (CTD y CTI) fueron generalmente cubiertas por la mayoría de alumnos. En el otro extremo, sin embargo, podemos observar que otras características más particulares relacionadas con la orientación a objetos, como el polimorfismo (PVI), la copia (CCA) y la destrucción de objetos (CDD) o la herencia entre clases (IOP e IOD) son características que tienden a pasar más inadvertidas para los alumnos, a pesar de que como cualquier elemento, pueden presentar defectos.

6. Evaluación del impacto

La experiencia presentada tenía por objetivo fomentar la necesidad de tener en cuenta los criterios de cobertura a la hora tanto de generar como de evaluar las pruebas desarrolladas. En la práctica diseñada, los alumnos toman de hecho ambos roles: primero como creadores de las pruebas y después como evaluadores sobre su diseño y completitud. Después, gracias al cálculo de la puntuación de mutación, pueden compa-

Operador	Descripción	Alum.	Media pond.
IPC	Eliminación de inicialización de atributo	14	100 %
OMR	Reemplazo de métodos sobrecargados	14	86.2 %
CTD	Eliminación de la palabra clave *this*	9	79.4 %
CTI	Inserción de la palabra clave *this*	9	79.4 %
IHI	Inserción de atributo en clase derivada	7	67.7 %
COD	Eliminación de operador condicional (ej.: !)	8	67.1 %
ARS	Reemplazo de operadores de asignación aritméticos (ej.: $+=$ por $-=$)	8	49.2 %
ROR	Reemplazo de operadores relacionales (ej.: $>$ por $>=$)	7	33.4 %
AIS	Inserción de operador de incremento/decremento (ej.: $++$)	8	32.6 %
IOD	Eliminación de método sobreescrito en una clase derivada	3	31.6 %
CDD	Eliminación de destructor	1	14.8 %
ARB	Reemplazo de operadores aritméticos (ej.: $+$ por $-$)	2	4.2 %
IOP	Cambio de posición de la llamada a un método sobreescrito	1	2.8 %
CCA	Eliminación de constructor de copia y operador de asignación con copia	0	0 %
PVI	Inserción de palabra clave *virtual*	0	0 %

Cuadro 2: Mutantes cubiertos en cada operador de mutación. Se muestra el número de alumnos que detectaron algún mutante del operador (Alum.) y la media del porcentaje de mutantes cubiertos en cada operador ponderada en base a la puntuación de mutación obtenida por cada alumno (Media pond.).

rar la nota asignada por ellos y observar si concuerda con dicha puntuación. Como resultado de esta práctica, observamos que los alumnos tienden a asignar notas altas; sin embargo, las puntuaciones de mutación son bajas. Esta situación ha de alertarles rápidamente de que solo aplicar métodos manuales compromete la calidad del software, ya que la apreciación que realizamos suele tener un alto componente subjetivo.

Con base en los resultados presentados, creemos que así ha sido de forma mayoritaria. Los alumnos han reducido de forma generalizada sus valoraciones una vez en posesión del cálculo de la puntuación de mutación y tras apreciar cómo sutiles cambios del código han escapado de los asertos de las pruebas diseñadas. En el desarrollo de pruebas hay una tendencia casi irracional hacia tratar de probar que el programa funciona correctamente, cuando el objetivo de las pruebas debería de ser el contrario: detectar la presencia de fallos. Por esa razón, creemos que la aplicación del criterio de cobertura de mutantes juega un factor fundamental en esta práctica: al enfrentarles a potenciales fallos reales, se les muestra que lo importante no era demostrar que todo funcionaba como era de esperar, sino retarse en diseñar unas pruebas que no tuviesen deficiencias a la hora de detectar posibles defectos en el código.

En esta práctica también observamos el riesgo de que haya ciertas características que pasen más fácilmente desapercibidas en las pruebas, en especial las propias de cada lenguaje o dominio. Por ello, parece necesario que en las asignaturas de programación no solo se explique la utilidad de cada característica, sino que se incida en los posibles fallos de programación asociados a las mismas y las consecuencias de su aparición en los programas desarrollados.

Por último, hay que notar que la innovación se introdujo como una experiencia piloto en una única de una serie de prácticas que componen la asignatura. Por tanto, las valoraciones aquí mostradas no tienen una influencia alta en la nota global. En cuanto a la percepción de los alumnos, a partir de una encuesta observamos que estos valoran la experiencia de forma positiva dentro del proceso de aprendizaje de la asignatura, y en general recomiendan introducir conocimientos básicos de prueba de software, y también de forma más particular de la prueba de mutaciones, en las asignaturas de programación de los cursos iniciales.

7. Conclusiones

En este trabajo se ha presentado una experiencia de diseño y aplicación de prácticas basadas en los conceptos de autoevaluación y coevaluación para la etapa de prueba de software en asignaturas relacionadas con programación. En combinación con la puntuación automática que otorga la prueba de mutaciones sobre la calidad de las pruebas desarrolladas, pensamos que con esta innovación los alumnos han captado mejor la importancia de los criterios de cobertura que se les imparte a lo largo del curso a la hora de probar aplicaciones software. También, el propio proceso de generación de pruebas ha cobrado otra dimensión al hacerles ver que no basta con crear escenarios triviales de prueba que solo cubran las características de uso más común.

Como trabajo futuro, estaría bien incrementar el número de programas que son objetos de la práctica, desde casos muy sencillos (pero que favorezcan la comprensión de todo el proceso de prueba) hasta casos más complejos, de manera que se haga más evidente que al incrementar la complejidad se hace más necesaria una prueba de software más exhaustiva. También sería interesante agregar otros criterios de cobertura, de forma que, tanto los alumnos como los docentes, pudiéramos analizar la relación entre las pruebas diseñadas y los diversos criterios empleados.

8. Agradecimientos

Este trabajo fue realizado con la ayuda del proyecto de innovación docente de la Universidad de Cádiz con código: sol-201700083450-tra.

Referencias

[1] Francisco Chicano and Francisco Durán. Mutantes como apoyo para la valoración de pruebas. In *Actas de las XXI Jornadas de Enseñanza Universitaria de Informàtica, Jenui 2015*, pages 264–271, Andorra la Vella, Julio 2015.

[2] Peter J. Clarke, Debra Davis, Tariq M. King, Jairo Pava, and Edward L. Jones. Integrating testing into software engineering courses supported by a collaborative learning environment. *ACM Transactions on Computer Education*, 14(3):18:1–18:33, October 2014.

[3] Benjamin S. Clegg, José Miguel Rojas, and Gordon Fraser. Teaching software testing concepts using a mutation testing game. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering and Education Track*, ICSE-SEET '17, pages 33–36, Piscataway, NJ, USA, 2017. IEEE Press.

[4] Pedro Delgado-Pérez, Inmaculada Medina-Bulo, Francisco Palomo-Lozano, Antonio García-Domínguez, and Juan José Domínguez-Jiménez. Assessment of class mutation operators for C++ with the MuCPP mutation system. *Information and Software Technology*, 81:169–184, 2017.

[5] Pedro Delgado-Pérez and Inmaculada Medina-Bulo. Automatización de la corrección de prácticas de programación a través del compilador Clang. In *Actas de las XXI Jornadas de Enseñanza Universitaria de Informàtica, Jenui 2015*, pages 311–318, Andorra la Vella, Julio 2015.

[6] L. Deng, J. Offutt, and N. Li. Empirical evaluation of the statement deletion mutation operator. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 84–93, March 2013.

[7] C. Sandler G. J. Myers and T. Badgett. *The Art of Software Testing, 3rd Edition. (3rd ed.)*. John Wiley & Sons, 2011.

[8] Marco Antonio Gómez Martín, Guillermo Jiménez Díaz, and Pedro Pablo Gómez Martín. Test de unidad para la corrección de prácticas de programación, ¿una estrategia win-win? In *Actas de las XVI Jornadas de Enseñanza Universitaria de Informática, Jenui 2010*, pages 51–58, Santiago de Compostela, Julio 2010.

[9] Otávio Augusto Lazzarini Lemos, Fábio Fagundes Silveira, Fabiano Cutigi Ferrari, and Alessandro Garcia. The impact of software testing education on code reliability: An empirical assessment. *Journal of Systems and Software*, 137:497–511, 2018.

[10] Luiz Eduardo G. Martins and Tony Gorschek. Requirements engineering for safety-critical systems: A systematic literature review. *Information and Software Technology*, 75:71 – 89, 2016.

[11] Inmaculada Medina-Bulo, Pedro Delgado-Pérez, Antonia Estero-Botaro, M Carmen Castro-Cabrera, and Juan José Domínguez-Jiménez. Prácticas co-evaluables y autoevaluables para la etapa de prueba de software. In *Book of abstracts CIVINEDU 2018: 2nd International Virtual Conference on Educational Research and Innovation*, page 34. Adaya Press, 2018.

[12] Eduardo Mosqueira-Rey. La evaluación continua y la autoevaluación en el marco de la enseñanza de la programación orientada a objetos. In *Actas de las XVI Jornadas de Enseñanza Universitaria de Informática, Jenui 2010*, pages 223–230, Santiago de Compostela, Julio 2010.

[13] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Chapter six - mutation testing advances: An analysis and survey. volume 112 of *Advances in Computers*, pages 275–378. Elsevier, 2019.

[14] Macario Polo Usaola and Pedro Reales Mateo. Enseñanza de la mutación en pruebas de software. In *Actas de las XVIII Jornadas de Enseñanza Universitaria de Informàtica, Jenui 2012*, pages 1–8, Ciudad Real, Julio 2012.

[15] Óscar Sapena, Mabel Galiano, Natividad Prieto, and Marisa Llorens. Evaluación continua con CAP, un corrector automático de tareas de programación. In *Jornadas de Innovación Docente (ETS Ingeniería Informática)*, 2012.

[16] Pablo Sánchez and Carlos Blanco. Una metodología para fomentar el aprendizaje mediante sistemas de evaluación entre pares. In *Actas de las XIX Jornadas de Enseñanza Universitaria de Informàtica, Jenui 2013*, pages 37–44, Castellón de la Plana, Julio 2013.

[17] V. Javier Traver and Juan Carlos Amengual. Una propuesta de autoevaluación-reflexión para potenciar la responsabilidad individual. In *Actas de las XIX Jornadas de Enseñanza Universitaria de Informàtica, Jenui 2013*, pages 77–84, Castellón de la Plana, Julio 2013.

[18] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, December 1997.