Pruebas de Caja Negra: Una Experiencia Real en Laboratorio

Carlos López, Raúl Marticorena y David H. Martín
{clopezno, rmartico, dmartin}@ubu.es
Área de Lenguajes y Sistemas Informáticos
Universidad de Burgos

Resumen

En este trabajo se presenta la problemática de la realización de pruebas de caja negra en laboratorios de programación, así como la experiencia realizada con el objetivo de resolver dichos problemas.

Explicando las aproximaciones iniciales y experiencias previas efectuadas en cursos anteriores, se propone un proceso de preparación a la realización de prácticas, un desarrollo de las mismas, así como un resumen de los resultados obtenidos desde la óptica del alumnado. Se concluye con unas reflexiones sobre la experiencia por parte del profesorado.

De manera adicional, se plantean problemas y sus soluciones para cada uno de los temas que no son usualmente resueltos en las pruebas en orientación a objetos: pruebas de interfaces y pruebas de clases genéricas.

El propósito final es dar una propuesta que pueda ser reproducida en otros entornos y contextos universitarios, teniendo en cuenta los resultados aquí mostrados.

1. Introducción

El desarrollo dirigido por pruebas es, hoy en día, una realidad que no puede, ni debe, ser evitada en la docencia impartida en las universidades. La tendencia en auge de automatizar este flujo de trabajo, descrito en procesos de desarrollo actuales como RUP [6] o Extreme Programming [2], obliga a una mayor profundización en la realización de prácticas de laboratorio.

El presente trabajo analiza en primer lugar las experiencias previas que se han realizado sobre pruebas unitarias de caja negra, con sus pros y contras (Sec. 2). Identificado el problema de partida, se propone una solución y se detalla el cómo se ha llevado a cabo, enumerando y justificando todos los pasos realizados (Sec. 3). Todo trabajo de este tipo necesita una retroalimentación, por lo que se recogen (Sec. 4) los datos de las encuestas realizadas entre el alumnado, analizando razonadamente los resultados. Por último se finaliza con las conclusiones (Sec. 5) obtenidas de la realización de la experiencia y proponiendo líneas de trabajo futuro (Sec. 6) a realizar en cursos venideros.

2. Experiencias Previas

En cursos anteriores, se han venido desarrollando pruebas unitarias siguiendo el método de pruebas de caja negra. Para llevarlas a cabo nos hemos apoyado en JUnit [7] como framework de pruebas unitarias. Dicho framework permite la programación y ejecución automática de las pruebas.

Sin embargo, uno de los principios básicos a la hora de realizar las pruebas no se estaba cumpliendo: “el desarrollador no debe probar sus propios programas” [13]. Esta afirmación, aunque inicialmente es chocante, en la práctica se ha visto confirmada como una apreciación muy a tener en cuenta.

En cursos anteriores se planteaba una práctica de pruebas unitarias de caja negra, en la que el alumno debía implementar una interfaz dada e incorporar cuestiones de programación defensiva y/o de diseño por contrato a la clase implementada [11]. El alumno desarrollaba así tanto la clase a probar, como la propia batería de pruebas que debía superar dicha clase.

Aunque el proceso de desarrollo se asemeja a lo postulado por Extreme Programming [2] en cuanto al desarrollo paralelo de las pruebas y del software, hay que señalar que no se impuso a los alumnos la necesidad de iniciar antes la implementación de las pruebas que la de la propia clase.

Se observa que los alumnos de segundo curso no están preparados, ni confiados, en sus conocimientos de programación, como para afrontar directamente el desarrollo de la batería de pruebas automática sin una implementación tangible. Quizás esta experiencia pueda ser aplicada en cursos superiores, pero por el momento parece necesario tener una implementación real que probar.

Normalmente suelen implementar primero la clase para posteriormente probarla. Sin embargo hemos ido observando a lo largo de estos años que aunque el concepto de prueba se entendía, y que la utilización de JUnit se asimilaba, con mayor o menor dificultad, los tests realizados se veían normalmente "desviados" de lo que se esperaba.

Volviendo de nuevo al principio de prueba, se deduce que los alumnos no son agresivos con sus pruebas puesto que psicológicamente a nadie le gusta encontrar errores en lo que ha hecho. En muchos casos las pruebas eran excesivamente "suaves" a la hora de plantear todas las posibilidades de error.

Pese a insistir en la aplicación de métodos de particiones de equivalencia, y más particularmente en análisis de valores límites [13], los alumnos tienden a desviar sus pruebas hacia valores en los que su implementación particular funciona bien.

En segundo lugar, muchos de los tests estaban condicionados por la implementación concreta. Dependiendo de la estructura de datos o del tipo de recorridos y accesos realizados, los tests se diseñaban y programaban de distintas formas.

Recordando de nuevo que se les suele proporcionar una interfaz, y que las pruebas se realizan sobre la interfaz, conectando una instancia de la clase concreta implementada por ellos, no deberían plantearse las pruebas de cara a "su implementación", sino de cara a cualquier implementación que cumpla con el contrato establecido en la interfaz. En las horas de laboratorio se les sugería que cruzasen sus pruebas con las implementaciones de sus compañeros, pero como toda actividad voluntaria, nunca fue llevada a cabo.

A la hora de realizar las defensas orales de las prácticas, muchos de los recorridos de código realizados en presencia del profesor (siguiendo los propuesto por [15] en inspecciones de código) derivaban finalmente en: *"¿Por qué no has implementado esta prueba? Porque mi implementación no falla. He utilizado X y al utilizar X eso no puede pasar..."*

Como se puede deducir, se obtenían pruebas condicionadas a que el alumno conocía la implementación, convirtiendo finalmente las pruebas de caja negra en algo más cercano a pruebas de caja blanca.

3. Propuesta de Prácticas

La necesidad de resolver este problema que se nos estaba planteando año tras año, nos ha llevado a proponer otro tipo de prácticas en la que los alumnos no dispongan de la implementación.

Al no conocer detalles de la misma, se esperaba que ese tipo de razonamientos desaparecieran, aún siendo conscientes de que pueden aparecer nuevos problemas.

3.1. Contexto

Antes de plantear nada es necesario situarse en el contexto en que se ha llevado a cabo la experiencia. Se ha desarrollado en el laboratorio de prácticas de una asignatura de programación, en segundo curso de Ingeniería Técnica en Informática de Gestión.

Durante el desarrollo de una primera práctica previa, se ha solicitado a los alumnos que implementen el clásico juego del "Cuatro en Raya" en un lenguaje orientado a objeto. Se utiliza el lenguaje Java, sobre el entorno de desarrollo JDK 1.5 [17] con soporte de genericidad.

Para facilitar la labor, se les proporcionan los diagramas de clases UML, en vista de implementación, de la solución propuesta identificando: clases, atributos, métodos, modificadores de acceso y relaciones entre clases e interfaces. En concreto se les proporcionan los códigos comentados de la interfaz a implementar, para ayudarles a distinguir desde el principio entre especificación e implementación.

Una de las piezas claves del juego es la clase `Tablero` que, como veremos posteriormente, jugará un papel fundamental en las pruebas. El tablero es un simple contenedor de elementos de tipo `Ficha`, donde se depositan en el transcurso del juego.

Como resultado se espera que los alumnos adquieran fluidez en el desarrollo con un paradigma nuevo, orientado a objetos, frente a desarrollos estructurados. A la vez, dejamos a los alumnos familiarizarse con la temática que será objeto de la práctica final, meta de esta experiencia.

3.2. Enunciado de la Práctica

La práctica de pruebas se plantea desde el rol de *tester* o probador. Se supone que una empresa ficticia nos ofrece distintas implementaciones (cuatro en nuestro caso) de una interfaz genérica `ITablero`, cuyos elementos están acotados por subtipado (genericidad restringida) [11]. Respecto a la corrección del software se sigue un enfoque de programación defensiva, con la declaración de excepciones a lanzar ante casos inválidos.

A continuación se presenta el código sin comentarios.

```
package juego;
public interface ITablero<E extends
IElemento> {
   int MINIMO = 5;
   int MAXIMO = 100;
   void ponerElemento(E elemento,
     int fila, int columna) throws
     juego.PosicionOcupadaException,
     juego.ElementoNuloException,
     juego.FueraDelTableroException;
   E consultarElemento(int fila,
     int columna) throws
     juego.FueraDelTableroException;
   int consultarAlto(); // filas
   int consultarAncho(); // columnas
   void borrar();
} // interface ITablero
```

Algoritmo 1. Código de la `Interface ITablero`

El objetivo de la práctica es programar, utilizando el framework JUnit, las pruebas unitarias para las implementaciones de la interfaz. Las pruebas de caja negra se deben justificar en base a métodos de caja negra: particiones de equivalencia y análisis de valores límites [13].

Una vez implementadas las pruebas deben ser ejecutadas sobre las distintas implementaciones proporcionadas por nuestra empresa proveedora. En función del número de fallos (*failures*) y errores (*errors*) detectados sobre cada implementación, se insta a los alumnos a que cataloguen y clasifiquen de mejor a peor implementación, en base a los resultados, las cuatro implementaciones.

Para la identificación de los casos de prueba se les proporciona una plantilla a rellenar (Tabla 1) quedando a juicio del alumno el estimar cuántos tests tienen que realizar para estar seguros de cubrir completamente el caso de prueba. En esta tabla, T1 ... T4, se corresponden con las cuatro implementaciones proporcionadas.

Código	Descripción de prueba	T1	T2	T3	T4
CP01	instancia correctamente	☐	☐	☐	☐
CP02	permite generar instancias fuera de los límites	☐	☐	☐	☐
CP03	coloca el elemento en las coordenadas	☐	☐	☐	☐
CP04	no deja colocar fuera del tablero	☐	☐	☐	☐
CP05	no deja colocar en una posición ya ocupada	☐	☐	☐	☐
CP06	no deja colocar un elemento nulo	☐	☐	☐	☐
CP07	devuelve el elemento en las coordenadas	☐	☐	☐	☐
CP08	no deja consultar fuera del tablero	☐	☐	☐	☐
CP09	borra el tablero correctamente	☐	☐	☐	☐
nº total de tests con fallo (en JUnit)					
nº total de tests con error (en JUnit)					

Tabla 1. Plantilla de Casos de Prueba

La práctica concluye solicitando a los alumnos una propuesta de compra y modificación de las implementaciones ofertadas. La búsqueda de un resultado práctico concreto se traduce en un elemento de competición motivador para el estudiante.

3.3. Cómo Evitar Crear una Clase de Prueba para cada Implementación

A la hora de utilizar un framework de pruebas unitarias como JUnit surge la cuestión de cómo afrontar pruebas contra múltiples implementaciones de una interfaz. Aunque el problema de probar una clase abstracta ha sido tratado en mayor o menor profundidad [18], no era aplicable en la misma medida. Este problema también planteado y resuelto en [9] ofrecía una solución más compleja y que nos obligaba a implementar una clase para cada implementación a testar. Dicha solución se basa en el patrón de diseño *Método de Fabricación* [5].

En nuestro caso tenemos cuatro implementaciones de la misma interfaz Java. Una posible solución es implementar una clase con tests para cada una de las implementaciones, o bien establecer una jerarquía de herencia entre la clase con tests y las clases con instanciaciones particulares para cada implementación [9]. El

único punto de variabilidad en cada caso sería la instanciación concreta.

Sin embargo esta solución se antoja inadecuada puesto que ante un aumento del número de clases a probar cuando, todas ellas implementan la misma interfaz, implica un aumento del número de clases con test (heredan de `junit.framework.TestCase`).

Una solución a este problema es la utilización del patrón de diseño *Método de Fabricación* [5] combinada con un patrón *Fábrica Abstracta* [5]. En esta solución la instaciación se delega sobre otra clase (fábrica abstracta) que en función de algún parámetro elige la instanciación concreta. En nuestro caso el problema era mayor por tratarse de instanciaciones genéricas [11].

Obviamente, al alumno no se le solicita resolver este problema, sino que se le proporciona la clase que implementa el método de fabricación. Los patrones de diseño serán tratados más adelante en la titulación.

En concreto, en nuestro caso, `FabricaTablero`, recibe una cadena de caracteres e instancia un objeto cuya clase implementa la interfaz genérica `ITablero<E extends IElemento>`.

Esto nos ha permitido que al programar en una única clase las pruebas y delegar todas las instanciaciones al método de fabricación, el alumno pueda programar las pruebas para toda clase que implemente dicha interfaz.

3.4. Probar Interfaces o Clases Genéricas

Las clases genéricas no están ampliamente difundidas, por lo menos, en la medida de lo que conocemos. Mientras que el polimorfismo de inclusión (a través de la herencia) es enseñado y practicado en la amplia mayoría de facultades de Informática, el polimorfismo paramétrico no está tan arraigado.

Uno de los principales factores que pueden provocar dicha situación es que los lenguajes que incluyen esta posibilidad no son ampliamente utilizados, como el caso de Eiffel [11] siendo la única excepción C++ [16].

Un lenguaje que poco a poco ha copado la enseñanza, como es Java, no incluía hasta ahora dicha posibilidad, y por lo tanto no se utilizaban clases genéricas en las prácticas. Este vacío se cubría simulándolo con polimorfismo de inclusión.

Al utilizar la última versión del compilador de Java [17], se nos ha abierto todo un abanico de posibilidades a la hora de poder combinar ambos tipos de polimorfismo.

Sin embargo, esto aplicado a las pruebas, lleva a plantear un problema nuevo: ¿cómo se deben realizar las pruebas de una clase genérica? Aunque en nuestro caso se tratara de una interfaz, la cuestión era similar.

En la bibliografía relativa a pruebas, no es un tema que esté tratado con excesiva profundidad. En concreto, en [3] se recomienda repetir las pruebas con todas aquellas posibles instanciaciones genéricas posibles que van a llevarse a cabo en nuestro sistema.

En este caso particular, el tablero iba a ser utilizado con elementos de tipo `Ficha`, por lo que, para facilitar la tarea del alumno, sólo se han solicitado pruebas con instanciaciones genéricas de la forma `ITablero<Ficha>`. Al alumno se le mostraba algún ejemplo con código de instanciación a través de la fábrica para evitar problemas en la realización de la misma.

3.5. El Problema de la Copia de Prácticas

En años anteriores se ha observado también un problema habitual a la hora de realizar las prácticas y que se repite siempre: la copia de prácticas.

Como se señala en [4] es inevitable una cierta similitud entre las prácticas, más aún cuando se parte de una especificación común y existe una relación cotidiana entre los alumnos. Este problema de difícil solución podría intentar solventarse de una manera reactiva, con herramientas que detectan la copia de práctica [4]. Sin embargo, en este caso, se ha pretendido evitar que los alumnos inicien la copia de la práctica, desde un enfoque basado en la motivación al trabajo personal del alumno bajo una premisa de "premios" (detección de un mayor número de fallos y errores) y no basada en el miedo al castigo por la copia de la práctica.

Para ello se han producido varios lotes con distintas combinaciones de las implementaciones. En concreto partiendo de un número de N implementaciones de la interfaz, se combinan aleatoriamente en un lote (`.jar`) un número M de clases a probar por los alumnos, cumpliéndose que M<N.

Dado que era la primera experiencia que se realizaba de este tipo, y con el fin de tener delimitado el número de casos, se ha reducido el número de combinaciones.

Aunque obviamente esto no evita la copia de los tests entre los alumnos, sí que crea una cierta desconfianza, al no tener claro que los resultados obtenidos por sus compañeros se puedan aplicar de la misma forma a su combinación de clases. Se ha evitado además proporcionar ningún tipo de información al alumno sobre el número de clases implementadas originalmente o el modo de combinar las mismas para que no puedan hacer ningún tipo de suposición.

Particularmente se ha observado que el hecho de que cada pareja se enfrente a distintos casos anima a buscar el mayor número de fallos partiendo de lo que el propio alumno ha pensado, y no a través de una simple copia de los tests.

3.6. Evitar Pruebas de Caja Blanca: Ofuscar el Código

El problema de utilizar Java es que pese a dejar a los alumnos código binario (ficheros compilados con extensión `.class`) es relativamente sencillo decompilar el código y regenerar de nuevo el código fuente. Valga como ejemplo el uso de Jad [8] o DJ Java Decompiler [14] que es capaz de decompilar incluso ficheros generados con la última versión JDK 1.5 (aunque perdiendo información respecto a la genericidad).

Esto no sólo ocurre en este lenguaje, sino en muchos otros, por lo que parece interesante mostrar la solución adoptada de cara a poder ser aplicado en otros contextos.

Esto es especialmente preocupante si los ficheros implementación de las clases introducen "ligeras modificaciones" para ser localizadas en las pruebas de caja negra.

Si los alumnos decompilan las clases e inspeccionan el código, nos encontramos precisamente de nuevo en la situación que se pretendía evitar: convertir las pruebas de caja negra en pruebas de caja blanca viciadas por el conocimiento de la implementación particular.

Aunque no del todo carente de interés este otro enfoque, se pretendía evitar precisamente dicha situación. Para ello se ha utilizado un ofuscador de código Java, de tal forma que al intentar decompilar las clases se obtenga un código ininteligible. En concreto se ha utilizado yGuard [19] como ofuscador de clases.

Como resultado del proceso, los ficheros con las implementaciones empaquetados en un fichero `.jar` estaban ofuscados evitando (o dificultando al menos) el realizar las pruebas de caja negra con cierto conocimiento de la implementación concreta.

3.7. Proceso de Elaboración

Planteados todos los problemas y las soluciones propuestas para llevar a cabo una experiencia de este tipo se detalla todo el proceso en conjunto.

Tal y como se muestra en la Figura 1, el proceso se inicia con la definición de la interfaz y las implementaciones con pequeñas variaciones controladas por parte del desarrollador (rol ocupado por los profesores). En dichas variaciones se introducen modificaciones y fallos a propósito. Cada una de las clases se nombra con una letra distintiva (`TableroA...TableroZ`).

A partir de un número suficiente de implementaciones se hacen lotes de cuatro clases en los correspondientes ficheros empaquetados. Se conserva en ficheros externos un registro de las asignaciones (Tabla de Asignación), de manera que el docente sepa en todo momento cuál es la composición real de cada uno de los lotes.

Una vez generados los 23 lotes (`.jar`), uno por cada letra del NIF, se ofuscan con yGuard. Además, en dicho proceso de ofuscación se renombran las cuatro clases de cada fichero `jar` como `T1`, `T2`, `T3` y `T4`, en función de la información previa en la Tabla de Asignación. Todos los alumnos disponen de estas cuatro clases en cada fichero `jar`, aunque sus códigos serán distintos.

El fichero `jar` ofuscado se distribuye a través de la web de la asignatura para que cada pareja de prácticas descargue el fichero que le corresponde, en base a la letra del NIF del primero de los integrantes según la lista de alumnos.

El proceso se ha automatizado completamente con la herramienta Ant [1].

A partir de aquí comienza la labor de los alumnos, que deben implementar la clase con los tests, recoger resultados y rellenar la plantilla que se les proporciona. Tanto los enunciados como los

ficheros `jar` y fuentes disponibles se pueden consultar en línea en la web[1].

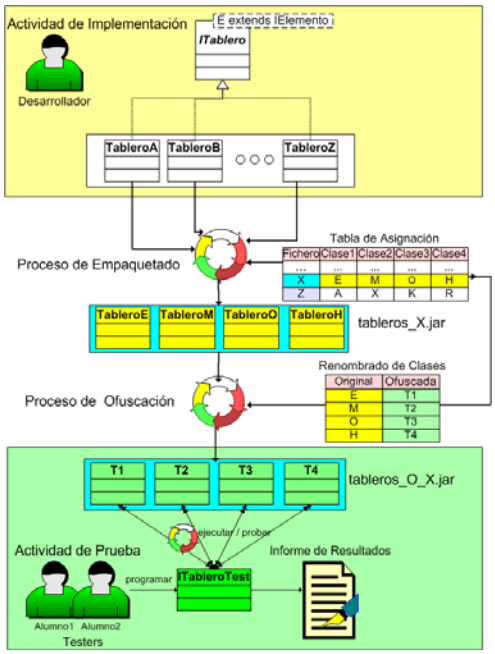

Figura 1. Proceso de Elaboración de la Práctica

4. Encuestas Recogidas

Para evaluar adecuadamente la experiencia es necesario conocer las opiniones de los alumnos. En concreto se ha realizado una serie de encuestas anónimas a los alumnos con las siguientes preguntas:

1. ¿Se ha comprendido el concepto de pruebas de caja negra por parte del alumno?
2. ¿Se ha logrado diferenciar por parte del alumno el papel de desarrollador frente papel de tester?
3. Valoración de la práctica para la comprensión de pruebas de caja negra.
4. ¿Es útil automatizar las pruebas con un framework de pruebas?
5. ¿Se ha diferenciado claramente entre pruebas de caja negra y caja blanca?
6. ¿Son adecuadas las pruebas de caja negra (sin código disponible)?
7. ¿Ayuda la realización de la práctica a comprender las pruebas de caja negra?
8. ¿Es adecuado el uso de JUnit como herramienta en prácticas para automatizar las pruebas?
9. ¿Dificultad de identificación de casos de pruebas a partir de la especificación?

Para poder comparar además con las decisiones tomadas en años anteriores, se consultó a los alumnos que no cursaban la asignatura por primer año, sobre la preferencia de realizar las prácticas como en años anteriores, o bien continuar con el enfoque tomado este curso.

Cada pregunta se valoraba de 1 a 5 (1 = muy mal, 5 = muy bien, las respuestas en blanco no se valoran). La encuesta se ha realizado sobre una muestra de 57 alumnos (de los cuales 24 no cursaban la asignatura en primer año.)

Los datos obtenidos se muestran a continuación en la Tabla 2.

	Puntuación / Frecuencia					Valor Medio
Pregunta	1	2	3	4	5	
1	3	8	8	20	18	3,74
2	2	3	14	22	16	3,82
3	2	10	24	14	7	3,25
4	1	1	18	16	17	3,61
5	2	6	14	15	20	3,79
6	1	10	17	19	10	3,47
7	4	5	17	17	13	3,47
8	1	1	12	27	16	3,98
9	3	4	21	22	6	3,37

Tabla 2. Resultados de Encuestas Realizadas

En líneas generales se puede observar que todas las puntuaciones superan el aprobado (valor 3). En la Figura 2 se visualizan los datos agrupados. Como se puede observar las áreas mayores están en los valores 3, 4 y 5.

Se analizan a continuación más en profundidad cada uno de los valores medios obtenidos. El concepto de pruebas de caja negra (pregunta 1) y su diferencia con las pruebas de caja blanca (pregunta 5) parece haber sido muy bien asimilado por los alumnos. Sin embargo es chocante que la pregunta 3 sea la que menor nota media haya conseguido cuando se ha preguntado (de otra forma) algo muy similar.

El hecho de que la práctica ayude especialmente a los alumnos en esta tarea (pregunta 7) se mantiene con valoración positiva.

[1] http://pisuerga.inf.ubu.es/rmartico/docencia/mp/practicas_obligatorias.html

Curiosamente la adecuación de las pruebas de caja negra (sin código disponible) ha sido de las cuestiones más discutidas (pregunta 6).

Otro de los puntos de interés, y no sólo relacionado con la asignatura en cuestión sino de cara a otras asignaturas de la troncalidad de ingeniería del software, es la diferenciación entre los roles de programador y tester/probador. Este concepto en particular ha sido claramente diferenciado en la práctica (pregunta 2).

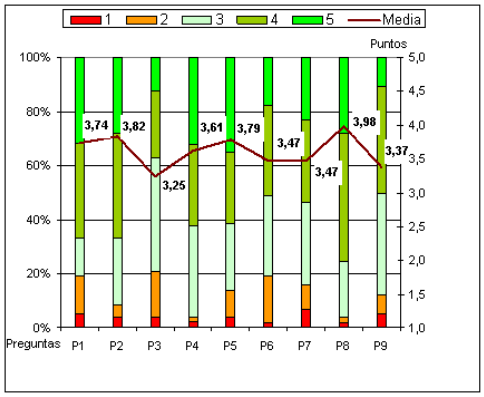

Figura 2. Resultados de la encuesta

Sobre el uso de *frameworks* de pruebas unitarias en prácticas, a lo largo de los años ha habido un gran número de discusiones con los alumnos sobre la adecuación de usar JUnit como herramienta. Aunque se ha convertido en un estándar de facto, y la idea ha sido ampliamente adoptada (PHPUnit, NUnit, etc.), es éste un punto abierto de discusión. La pregunta 4 es contestada por la mayoría positivamente: es adecuado utilizar un framework de pruebas que automatice el proceso. De igual manera los alumnos apoyan rotundamente la utilización de JUnit para realizar esta labor (pregunta 8), pese a que inicialmente se encuentran reacios, quizás motivado por "miedo a lo nuevo".

Por último cabe señalar las dificultades encontradas por los alumnos a la hora de definir los tests para realizar la práctica, como se refleja de los bajos resultados obtenidos en la pregunta 9.

Otro de los puntos de interés de la encuesta es cruzar los resultados de los alumnos nuevos frente a alumnos que ya habían cursado la asignatura. En general los alumnos nuevos son más críticos con el sistema, debido a que no han experimentado las pruebas conociendo el código fuente.

Por el contrario los repetidores apoyan de una manera más aplastante la realización de prácticas de caja negra sin implementar la clase a probar. Siempre cabe la sospecha de que prefieren trabajar menos, puesto que se les ha pedido menor esfuerzo de implementación, pero también es cierto que el esfuerzo a la hora del diseño e implementación de las pruebas es mayor.

En concreto respecto a la pregunta realizada aisladamente a los repetidores sobre la adecuación de las prácticas y mejora respecto a años anteriores, se tiene que: el 29% opinaron que el nuevo sistema era *peor*, el 54% que el sistema el nuevo sistema era *mejor* y el 17% que el sistema es *mucho mejor*. Las valoraciones *mucho peor* e *igual* tuvieron un 0% de votaciones.

Al respecto, el 71% consideran que se ha mejorado con el nuevo sistemas de prácticas. Sin embargo también se debe tener en cuenta el hecho de que esta solución no ofrece posturas intermedias, el 29% la rechaza y por lo tanto habrá de ser revisada para intentar descubrir las deficiencias de la misma, o bien suponer que todo tipo de prácticas obligatorias siempre crea un cierto rechazo por una parte del alumnado.

5. Conclusiones

A partir de una idea que se venía arrastrando desde que se incorporaron prácticas con pruebas de caja negra, se ha intentado mostrar y detallar como se puede cumplir el principio de no probar el código que uno mismo implementa.

Como toda experiencia, se han visto puntos fuertes y débiles. Sin embargo en este caso, como se puede apreciar en las encuestas realizadas, la tendencia es positiva.

Los alumnos han aprendido a diseñar y programar pruebas, no dependientes de una implementación, sino a partir de una especificación, creando pruebas que se puedan a su vez reutilizar con nuevas implementaciones.

Aún así, no todo ha sido positivo. En las defensas orales de las prácticas, se observa que los alumnos tienden a viciar o condicionar los tests a partir del comportamiento observado por medio de las pruebas, aunque no dispongan del código fuente. Este problema parece difícil de evitar.

La única solución posible, ante esta situación, sería no dar implementación real sobre la que ejecutar las pruebas, proporcionando un módulo de simulación (Stub o Mock Objects). Pero dicha solución creemos que sólo podría desmotivar y desorientar más a alumnos de primeros cursos, cuya confianza en las pruebas automáticas no está del todo afianzada.

6. Líneas de Trabajo Futuro

Somos conscientes de que queda mucho trabajo. Las pruebas se han enfocado desde un único punto de vista de la especificación, sin entrar en pruebas de la implementación concreta salvando problemas como la encapsulación.

Sin embargo, se ha creído que para ser la primera toma de contacto con los alumnos con este tipo de prácticas, el nivel alcanzado ha sido adecuado.

La combinación de pruebas de caja negra con pruebas de caja blanca es algo que no debería obviarse y que ya se plantea en prácticas de asignaturas de cursos superiores combinando herramientas como JUnit y jcoverage [12]. Dado el carácter cuatrimestral de la asignatura en la que nos hemos centrado, esto es imposible.

Como ya se ha señalado, se abre la posibilidad al posible cruce de implementaciones de alumnos contra pruebas de otras parejas.

Por último, se nos ha planteado la duda de poder solicitar implementaciones que engañen a ciertos tests que se les proporcionen a los alumnos. Este último enfoque seguiría lo planteado en [9] sobre el desdoblamiento de personalidad a la hora de probar el software. Además este tipo de enfoques de "desafío" parece ser muy bien aceptado por el alumnado.

Más aún, se debería repetir este tipo de prácticas en cursos superiores de ingeniería para observar la relación de la madurez del alumno con este tipo de prácticas siguiendo lo planteado en otros trabajos [10].

Referencias

[1] Apache Group. *Ant*. http://ant.apache.org.

[2] Beck, K. *Una Explicación de la programación Extrema. Aceptar el cambio.* Addison Wesley, 2002.

[3] Binder, R.V. *Testing Object-Oriented Systems. Models, patterns, and tools*. Addison Wesley, 2000.

[4] Clemente, P.J., Gómez, A. y González, J. *La copia de prácticas de programación: el problema y su detección*. Actas JENUI 2004 pp 203-210

[5] Gamma, E. Helm, R., Jonhson R., y Vlissides, J. *Patrones de Diseño*. Addison Wesley, 2003.

[6] Jacobson, I. et al., *El Proceso Unificado de Desarrollo de Software.* Addison Wesley, 2000.

[7] JUnit. *Testing Resources for Extreme Programming*. http://www.junit.org.

[8] Kounetsov, P. *Jad – the fase Java decompiler.* http://www.kdpus.com/jad.html.

[9] Link, J. et al. *Unit Testing in Java. How Tests Drive the Code.* Morgan Kaufmann, 2003.

[10] Marticorena, R., López, C. y Crespo, Y. *Estudio de la Distribución Docente de Pruebas del Software y Refactoring para la Incorporación de Metodologías Ágiles.* Actas JENUI 2004 pp 247-254.

[11] Meyer, B. *Construcción de Software Orientado a Objetos 2ª Edición.* Prentice Hall, 1998.

[12] Morgan, P. et al., *jcoverage* http://www.jcoverage.com/.

[13] Myers, G.J. *El Arte de Probar el Software*. El Atenero, 1984.

[14] Neskhovs, A. *DJ Java Decompiler*. http://members.fortunecity.com/neshkov/dj.html.

[15] Sommerville, I. et al. *Ingeniería del Sofware. 6ª Edición*. Addison Wesley, 2002.

[16] Stroustrup, B. *The C++ Programming Language.* 3rd Edition. Addison Wesley 1997.

[17] Sun Microsystems. *Java 2 Platform Standard Edition* 5.0. http://java.sun.com/j2se/1.5.0/download.jsp.

[18] Walton, C. *Abstract Test Cases*. 2004. http://c2.com/cgi/wiki?AbstractTestCases.

[19] yWorks. *yGuard-Ant Java Obfuscator.* http://www.yworks.com/en/products_yguard_about.htm.