Test de unidad para la corrección de prácticas de programación, ¿una estrategia win-win?

Marco Antonio Gómez Martín, Guillermo Jiménez Díaz, Pedro Pablo Gómez Martín
Facultad de Informática
Univ. Complutense de Madrid
{marcoa,gjimenez,pedrop}@fdi.ucm.es

Resumen

Los test de unidad permiten comprobar el correcto funcionamiento de las unidades básicas de una aplicación. Desde hace algunos años venimos proporcionando a nuestros alumnos, junto con el enunciado de las prácticas, una batería de test de unidad que debe ser superada por su solución para poder entregar la práctica. Desde el punto de vista del profesor, los test permiten comprobar automáticamente aspectos que de otra forma o bien son inviables de probar, o bien requieren un tiempo excesivo de revisión de la práctica.

En este artículo se presenta también el punto de vista del alumno, describiendo los resultados obtenidos en varias encuestas realizadas con alumnos que han hecho uso de estos test de unidad. Estos resultados se han usado para evaluar si el uso de los test es una estrategia "win-win" en las asignaturas de laboratorio: útil para los profesores, útil para los alumnos.

1. Introducción

En las asignaturas de laboratorio de programación los alumnos se enfrentan a la implementación de pequeñas aplicaciones que deben funcionar correctamente para sean consideradas como válidas.

La corrección, sin embargo, no debe consistir únicamente en la ejecución de sus programas para ver si funcionan tal y como les exigimos, sino que también debe incluir la inspección del código fuente para ver si hacen uso de buenas prácticas de programación, utilizan las estructuras o algoritmos que queremos que practiquen, etc. Lamentablemente esa tarea consume tanto tiempo y son tan pocos los recursos humanos que tenemos a nuestro alcance que en la mayoría de los casos el código no lo comprobamos con tanto detalle como deberíamos.

Muchas son las experiencias que describen la creación y uso de sistemas automáticos de corrección que alivian, al menos en parte, la tarea de la evaluación de las prácticas de los alumnos [5, 15]. En algunos de los casos estos sistemas se utilizan *a posteriori* para, incluso, proporcionar la nota del alumno, una vez que éste ya ha entregado la práctica [13].

Este artículo describe el uso que se puede dar a los test de unidad en asignaturas de laboratorio. Como veremos pueden cumplir una doble misión: como herramienta de corrección semiautomática para el profesor y como herramienta para comprobar su solución para los propios alumnos.

La sección siguiente hace una breve descripción del concepto de test de unidad. A continuación, en la sección 3, pasamos a describir cómo se pueden utilizar en las asignaturas de laboratorio de programación, identificando los distintos ejes de variabilidad que se tienen. De esta forma aquellos profesores que quieran comenzar a utilizar test de unidad en sus asignaturas tendrán una idea de qué decisiones deberán tomar antes de hacer uso de los mismos. La sección 4 describe exactamente cómo hemos utilizado los test durante estos últimos años en nuestras asignaturas de laboratorio. El artículo pasa después a evaluar, en la sección 5 el uso de test de unidad en el aula, basándonos en encuestas realizadas a los alumnos. El artículo se cierra con unas breves conclusiones.

2. Test de unidad

Los test de unidad son un método de prueba que verifica qué componentes individuales del código fuente funcionan correctamente. Para ello, el creador de los test implementa ejemplos de uso directamente en código fuente que es ejecutado durante el proceso de pruebas. Si la ejecución no obtiene los resultados esperados se dice que el test *falla* o *no pasa*, y por lo tanto se considera que el código probado es incorrecto.

El uso de test de unidad se ha hecho popular en

los últimos años, especialmente tras la aparición de las llamadas *metodologías ágiles* [14], como la programación extrema [6] o el desarrollo dirigido por test [7].

Para la programación y ejecución de los test de unidad existen multitud de *frameworks* en la mayoría de los lenguajes de programación mayoritarios. El más popular es, sin duda, JUnit, creado por Erich Gamma y Kent Beck a mediados de 1990 para realizar pruebas de código Java. Existen numerosos libros y artículos sobre JUnit [12, 16] y sus ideas se han llevado a librerías para otros lenguajes como C++ (UnitTest++, CppTest) o Python (unittest).

La popularidad de esta metodología de trabajo y la existencia de estos frameworks de uso extendido han provocado que numerosas herramientas de desarrollo o IDEs hayan incorporado facilidades para el uso de test de unidad. Por ejemplo Eclipse [2], uno de los IDE de Java más utilizados hoy día, dispone de distintos asistentes para la creación de paquetes de test. Además, permite la ejecución de los mismos desde el propio entorno, llegando a presentar una barra de progreso de color rojo o verde para indicar el fallo o éxito en la ejecución de los test.

3. Test de unidad en el aula

3.1. Motivación

En las asignaturas relacionadas con programación, y especialmente en los primeros cursos, suelen perseguirse objetivos relativamente modestos, como por ejemplo que los alumnos aprendan los métodos de lectura de ficheros teniendo en cuenta los posibles errores de entrada/salida. Para poner a prueba su destreza en estos quehaceres, el profesor puede plantear una serie de ejercicios cortos o la implementación de funciones o métodos concretos dentro de un contexto más general.

Sin embargo, en los laboratorios de programación es habitual aunar en una única práctica la implementación de una aplicación de pequeño tamaño que cumpla distintos objetivos. De esta forma reducimos el número de correcciones a realizar durante el curso y, creemos, aumentamos la motivación de los alumnos al construir una aplicación completa en vez de componentes dispersos sin una utilidad clara. Además, se ponen a prueba otras cuestiones como el trabajo en equipo, la planificación de tiempo y la organización del código.

Lamentablemente esta aproximación tiene una consecuencia negativa: los alumnos se centran en el funcionamiento correcto de la aplicación *completa* en detrimento de la corrección de cada componente individual, surgiendo lo que podemos llamar el síndrome del "así también funciona" [10]. Por seguir con el ejemplo anterior, en el método de lectura de fichero deja de tener importancia la gestión de los errores porque, aunque sobre el papel se pueden producir errores que impidan leer el fichero, en la aplicación que ellos han implementado raras veces ocurre.

El resultado es que la mayoría de los objetivos iniciales del profesor no se ven satisfechos pues los alumnos se olvidan de los detalles (que es lo que muchas veces se quiere poner a prueba) para centrarse en el funcionamiento global. Una posible solución es revisar todos esos componentes individuales durante la corrección, pero el número de alumnos hace difícil su puesta en práctica. Además, si la corrección de la práctica tiene asociada una calificación que repercute en la nota final del estudiante, retrasar la comprobación de todos estos detalles hasta el momento de la revisión de las prácticas resulta un tanto injusto para los alumnos. Al fin y al cabo, el último objetivo de la enseñanza es que los alumnos *aprendan*, por lo que resultaría mucho más interesante detectar estas desviaciones a tiempo y pedir a los alumnos que las corrijan antes de entregarlas y calificarlas. Naturalmente, esto añade aún más trabajo al profesor, que cargaría sobre sus espaldas no solo con la evaluación final de la práctica, sino con todas las revisiones intermedias para enmendar los errores que vayan cometiendo.

La solución que proponemos es utilizar test de unidad que comprueben esos elementos que nos ayudan a satisfacer los objetivos de la práctica. El *profesor* proporciona a los alumnos una batería de test que sus soluciones deberán pasar. Estos test amplían en cierto modo la *especificación* de cada práctica que, naturalmente, también se detalla mediante un enunciado tradicional. Los test comprueban todos aquellos detalles que se perseguían inicialmente (como el correcto uso de ficheros y su gestión de errores), pero que han quedado ocultos debajo de una práctica de una entidad mayor. De algún modo, los test se convierten en *correctores automáticos* de las prácticas, lo que permite a los estudiantes conocer si

van o no por el buen camino sin la intervención del profesor. Así mismo, los test impiden que se olviden de todos esos requisitos escondidos que, de otro modo, podrían estar pasando por alto.

En la fecha límite de entrega de la práctica, los alumnos deben realizar una *defensa* de su solución, cuyo prerrequisito es que supere *todos* los test. Gracias a esto, el profesor puede centrarse en comprobar el funcionamiento global de la aplicación porque sabe con total certeza que, al haber superado los test, se ha comprobado automáticamente que la solución satisface una gran cantidad de aspectos importantes, pero tediosos de verificar.

Aunque fue ésta la razón que nos impulsó a utilizar test en las clases de programación, su uso tiene una ventaja pedagógica adicional: los alumnos *se acostumbran a ellos*. De hecho, al proporcionárselos, no sólo estamos permitiendo la detección temprana de los errores que cometan, sino que también conseguimos que asimilen sus ventajas y vean ejemplos de test útiles. Esto nos crea el marco perfecto para *pedirles que realicen sus propios test* en los casos en que la asignatura pertenece a cursos superiores. La solución de una práctica debe, por tanto, ir acompañada de test de unidad adicionales, escritos por ellos mismos y que, naturalmente, su práctica deberá pasar.

3.2. Qué se puede probar

Dependiendo de la naturaleza de la práctica y del curso, los test entregados por el profesor pueden centrarse en diferentes aspectos:

- Métodos de acceso y modificación (`get` y `set`).
- Métodos de entrada de datos. En esta categoría entraría el ejemplo de los ficheros descrito anteriormente. Si forzamos a que se lea directamente de un flujo de datos dado (en vez de indicar el nombre del fichero), se pueden simular fácilmente ficheros con un formato incorrecto para ver cómo reacciona la implementación del alumno.
- Corrección de algoritmos (como los de ordenación, o el método que comprueba si en un tablero hay cuatro en raya). Dado que los test sólo comprueban *si el algoritmo funciona*, es complicado poder detectar automáticamente si se ha utilizado una implementación determinada, como el *Quicksort* en vez de *Mergesort*.
- Eficiencia de la implementación. Formalmente se puede hacer esta comprobación en aquellos algoritmos en los que la complejidad teórica es la misma pero no así el tiempo absoluto. Para eso los test pueden incorporar una implementación ingenua del mismo y comprobar que la solución del alumno mejora en una proporción determinada el tiempo utilizado.
- Restricciones en la implementación de clases. Por ejemplo, si les pedimos implementar una pila dinámica (utilizando listas enlazadas de nodos), podemos asegurarnos de que no están utilizando una implementación estática (con un array) comprobando que el tamaño de su tipo `Pila` no excede el tamaño de un simple puntero (a un nodo).
- Aplicación correcta de los patrones de diseño. Para crear la aplicación podemos pedirles la implementación de cierto patrón. Sin embargo, dado que el patrón es una solución *general* a un problema, es posible que el caso particular de uso no requiera la implementación completa del patrón. Por ejemplo, los test de unidad nos pueden asegurar que en su implementación del patrón *Observer* [9] lo alumnos han tenido en cuenta que no se puede registrar varias veces un mismo observador.
- Interfaz de usuario. Se puede verificar mediante test de unidad y los *frameworks* adecuados que una interfaz cumple con las especificaciones impuestas por el enunciado de la práctica, como la colocación de los distintos elementos de interfaz y su funcionalidad.
- Estilo de código. Técnicamente no se consideran test de unidad, pero lo incluimos aquí porque se puede utilizar la misma infraestructura para su ejecución. Existen herramientas (como Lint [8], Checkstyle [3] y PMD [4]) que analizan el código fuente para ver si el estilo es correcto, comprobando tabulaciones, declaración correcta de variables, cantidad de comentarios, etc.

3.3. Decisiones a tomar

A la hora de utilizar test de unidad en el aula, es importante destacar las distintas posibilidades o ejes de variación que el profesor tiene. Hemos detecta-

do cuatro ejes: *qué* se va a probar, *cómo* lo vamos a probar, *cuándo* y *dónde* se realizarán las pruebas.

Lo primero que hay que decidir es el tipo de comprobaciones que realizarán los test. En el apartado anterior hemos descrito las más significativas, pero eso no quiere decir que deban probarse todas o que no haya otras.

Una segunda decisión a tomar es si queremos publicar el código fuente de los test o no. En lenguajes como Java es posible empaquetar las clases de test *ya compiladas* en un `.jar`, omitiendo el código fuente. Para utilizarlas basta con que los alumnos *enlacen* ese fichero con la solución de su práctica compilada y lancen los test. Si alguno falla tendrán entonces que, guiándose por el nombre del test y algún texto de ayuda generado adicional, encontrar el fallo de su aplicación y solventarlo. Con esta filosofía, los test son *cajas negras* para los alumnos, que no saben a qué se enfrentan hasta que los prueban.

La otra alternativa es que el profesor les proporcione el código fuente de los test que deberán pasar, pudiendo así mirar exactamente qué caso concreto se está probando. Esto tiene también la ventaja añadida adicional de que el propio test les puede servir para *depurar* su código paso a paso en el caso de que falle. El problema de esta aproximación es que los alumnos más perezosos podrían resolver la práctica para conseguir que funcione con los casos particulares probados por los test, sin preocuparse de que, efectivamente, lo haga también de manera general.

La publicación del código fuente se puede realizar con dos objetivos distintos: publicarlo por el mero hecho de que sepan qué tipos de pruebas realizamos, o que deban *compilar* los test juntando su código fuente con el nuestro. En este último caso se pueden generar errores de compilación si hay alguna característica mal utilizada. Como ejemplo, el siguiente código (en un fichero de definición de test en C++) se asegura de que el fichero `Pila.h` del alumno incluye la guarda para evitar doble inclusión (con la pareja `#ifndef` - `#define`) y que no han utilizado la directiva `using namespace std;` en sus ficheros de cabecera:

```
// Incluimos ficheros del alumno
#include "MetodosOrdenacion.h"
#include "MideTiempos.h"
#include "Algoritmo.h"
#include "Pila.h"
#include "Pila.h" // ¿#ifndef-#define?

#include <string>
namespace { class string {}; }
// Si la línea siguiente da error, has
// utilizado el using namespace std en
// un .h, algo que nunca deberías hacer
string a;
```

No obstante, hay que tener en cuenta que la decisión sobre si dar o no el código fuente de los test puede que haya sido tomada por el propio lenguaje. Por ejemplo, en C++, sencillamente no es factible evitar dar el código fuente, pues la alternativa de dar los test en una librería ya compilada requiere conocer con exactitud el tamaño de los tipos que se prueban, así como las posiciones en memoria de cada atributo.

En tercer lugar debemos también plantearnos *cuándo* vamos a dar los test y realizar las pruebas. Existen hasta tres posibilidades:

- Publicar los test junto con el enunciado. De esta forma, pueden ir comprobando desde el principio si la funcionalidad pedida la están implementando bien. La principal pega de esta alternativa es que pueden convertir el desarrollo de la aplicación en una mera cuestión de superar los casos particulares probados por los test en lugar de programar con el objetivo de un funcionamiento global correcto.
- Publicar los test poco antes de la finalización del plazo de entrega. Por ejemplo, en prácticas de un mes de duración, podemos dar los test una semana antes. Idealmente en ese momento tienen la mayor parte de su práctica hecha y la última semana la dedican a los detalles finales y a arreglar los posibles errores detectados por los test de unidad. Así, los test no les *distraen* de su objetivo principal que es la implementación de la práctica. Esta alternativa tiene la desventaja de la falta de "monitorización automática" durante las primeras semanas, que podría llevar a algunos alumnos a soluciones que consideraban razonablemente válidas, pero que se han ido en realidad desviando abiertamente de la solución correcta que perseguía el profesor.

- No publicar los test, sino utilizarlos durante la defensa como un mecanismo de corrección semiautomática.

Por último, en algunos casos podemos plantearnos *dónde* se van a ejecutar las pruebas automáticas. Por ejemplo en Java podemos elegir proporcionar los test para facilitar su ejecución desde un IDE como Eclipse o desde un intérprete de comandos utilizando Ant [1]. Una alternativa más sofisticada (y que nos permitiría no dar el código fuente de los test independientemente del lenguaje) sería la ejecución remota de las comprobaciones: los alumnos envían la práctica a un servidor de pruebas que las lanza y emite un informe de vuelta.

4. Nuestra experiencia

Los autores de este artículo llevan utilizando test de unidad en asignaturas de laboratorio de programación desde el curso 2007/2008, en concreto en Laboratorio de Programación de Sistemas de 3º de la Ingeniería Técnica de Sistemas y en Laboratorio de Programación 2 de 2º de la Ingeniería Técnica en Informática de Gestión. En el primer caso el lenguaje utilizado es Java (con Eclipse como IDE) mientras que en el segundo caso es C++ (con Visual Studio).

4.1. Uso de test en Java

Para este laboratorio hemos utilizado una estrategia de prácticas incrementales que nos permite introducir de forma natural varios patrones de diseño [11].

Aunque para la implementación utilizan Eclipse como entorno de desarrollo, les imponemos una estricta estructura de directorios habitual en los proyectos Java: un directorio `./src` con el código fuente, un directorio `./test` en el que crearán sus propios test de unidad, y un directorio `./bin` donde se compila su código fuente.

Para los test, que hacen uso de JUnit, optamos por *no* publicar el código fuente sino proporcionarlos directamente compilados en un `.jar`, aunque sin preocuparnos de *ofuscarlo* para luchar contra la posibilidad de su decompilación. De hecho, de forma consciente introdujimos en clase una explicación sobre el funcionamiento de la máquina virtual de Java y la posibilidad de extraer el código fuente original a partir de la versión compilada usando *descompiladores*. Esto llevó a algunos alumnos a buscar en la Web más información sobre la decompilación para conseguir así el código fuente de nuestros test (obviamente, sin comentarios ni nombres de variables significativos). Aunque esto puede verse como contraproducente, en realidad nosotros lo consideramos una virtud, pues los alumnos interesados han aprendido a descompilar ficheros `.class` sin la intervención del profesor, lo que encaja perfectamente con las corrientes educativas en las que el alumno es el propio constructor de su aprendizaje.

La estructura de directorios pedida nos permite publicar un *script* de Ant que compila y ejecuta automáticamente todos los test. Eso nos permite tanto a nosotros durante la corrección como a los propios alumnos durante la elaboración de la práctica comprobar rápidamente si se superan todas las pruebas unitarias.

En los distintos cursos académicos que hemos impartido la asignatura hemos utilizado aproximaciones distintas en cuanto a *cuándo* darles los test, probando tanto su publicación a la vez que el enunciado de la práctica como unos días antes de la finalización del plazo. Las conclusiones de las dos estrategias se analizarán posteriormente.

4.2. Uso de test en C++

Nuestra experiencia con test en C++ se basa en el Laboratorio de Programación 2 utilizando Visual Studio como entorno para realizar prácticas relacionadas con la asignatura teórica de Estructuras de Datos y de la Información. Como se ha esbozado previamente, en este caso es técnicamente imposible proporcionarles los test compilados, por lo que les damos directamente el código fuente.

Igual que en el caso de Java, también les obligamos a utilizar una estructura de directorios determinada: el directorio `./Src` contiene los ficheros de implementación, y `./Projects` los ficheros de proyecto de Visual Studio.

Eso nos permite crear los test confiando en esa estructura. En concreto, les damos dos directorios, `./TestSrc` con el código fuente y `./TestProjects` con el fichero de proyecto. Para ejecutar nuestras pruebas automáticas deben abrir el proyecto que les damos, incluir en él sus ficheros de código fuente y compilarlo. El proyecto está configurado de forma que el propio Visual Studio ejecuta los test y, en ca-

so de error, los lista utilizando la misma ventana que usa cuando hay errores de compilación.

5. Evaluación

La sensación subjetiva que teníamos tras varios años de hacer uso de baterías de test como parte de la *especificación* de la práctica es que los alumnos los consideraban útiles. Sin embargo nos preocupaba que la opinión real fuera diferente, y que el uso de test les impusiera más desventajas que beneficios.

Para resolver nuestras dudas, tras el primer cuatrimestre del curso 2009/10 se realizó una encuesta anónima a los alumnos. Participaron 46 alumnos de la asignatura de Laboratorio de Programación 2 y 23 alumnos de la asignatura Laboratorio de Programación de Sistemas. El cuestionario de escala de Likert consistió en un conjunto de afirmaciones que los alumnos debían valorar usando una típica escala 1..5, donde 1 significa *"En completo desacuerdo"*, y 5 significa *"Completamente de acuerdo"*. Las preguntas eran del tipo:

- Me ha resultado fácil usar los test.
- Me he sentido más relajado/seguro con respecto a la evaluación de mi práctica al poder haber ejecutado los test con anterioridad a la entrega.

Sus respuestas no han podido ser más esperanzadoras (figura 1). Casi el 65 % de los alumnos está de acuerdo o totalmente de acuerdo con el uso de los test. Además, los test consiguen que cerca del 75 % de ellos afronten con más confianza la defensa de la práctica frente al profesor, al tener una mayor certeza de que no tenía errores graves gracias a los test. El resultado es tan positivo que, de nuevo, casi el 65 % querría que otros profesores tomaran medidas similares en sus asignaturas.

Estas buenas opiniones se ven perfectamente representadas en la utilidad de los test tal y como es percibida por los alumnos (figura 2). Así, el 84 % está de acuerdo o totalmente de acuerdo ante la pregunta de si los test les han servido para detectar fallos. En este caso, *ningún* alumno está en desacuerdo con esta afirmación, lo que demuestra que, efectivamente, los test sirven dan la oportunidad al estudiante de solucionar los problemas antes de ser evaluados.

Nos preocupaba que el uso de test supusiera una sobrecarga sobre los alumnos al tener que preocu-

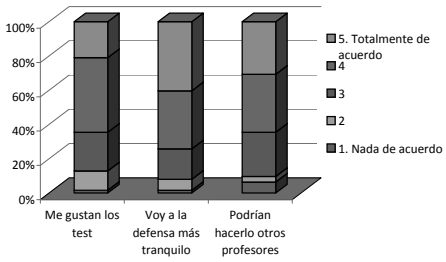

Figura 1: Opiniones de los alumnos sobre los test

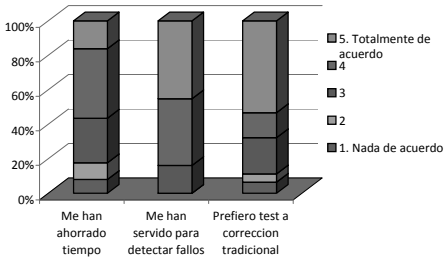

Figura 2: Utilidad de los test para alumnos

parse no sólo por programar bien la solución, sino por superar los test. Parece que nuestros alumnos no han sentido esta sobrecarga pues más del 55 % tienen la sensación de haber ahorrado tiempo. Esta percepción, unida a la tranquilidad a la hora de defender las prácticas mencionada antes, lleva a casi el 70 % a preferir la corrección de las prácticas con test frente a las alternativas tradicionalmente usadas sin test.

La opinión del alumnado diciendo que los test les ahorran tiempo resultó bastante gratificante, pues temíamos que la curva de aprendizaje de uso de los test fuera grande. El 60 % considera que el uso de test *es fácil*, sin apreciar una diferencia significativa entre los dos grupos. Esto es esperanzador, pues nos temíamos que el uso de test en C++ (y en segundo) fuera muy complicado, especialmente en comparación con su alternativa Java.

Tampoco se aprecia diferencia significativa en las respuestas de los dos grupos ante la pregunta de si consideran que disponer del código útil de los test es interesante (figura 3). Es necesario recordar que los alumnos de tercero *no* recibieron el código, por lo que la encuesta indica si les hubiera gustado tenerlo.

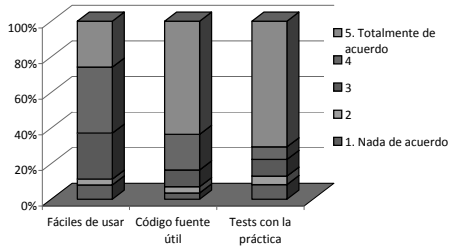

Figura 3: Ejes de variación del uso de test

Por su parte, el grupo de 2º, que tuvo el código desde el principio, lo agradece también. En total, casi el 85 % de los alumnos consideran interesante disponer del código fuente de los test.

Por último, respecto a la pregunta "¿cuándo?", más del 75 % prefiere tener los test desde el primer momento, formando parte de la especificación de la práctica, en lugar de esperar a que se acerque el día de la entrega para recibir los test a modo de prerrequisito para ser entregada.

Si pasamos a evaluar el otro punto de vista, el del profesor, es justo decir que la creación de los test supone un trabajo adicional, aunque el esfuerzo es compensado con el ahorro de tiempo durante la corrección. El uso de test obliga a implementar la práctica y crear los test para intentar cubrir todos los casos límite posibles. Curiosamente, en nuestra experiencia hemos visto que por mucho que intentemos ser exhaustivos, siempre hay alumnos que encuentran errores que no habíamos probado con nuestros test.

6. Conclusiones

En este artículo hemos presentado nuestras experiencias con el uso de test de unidad en dos asignaturas de laboratorio de programación distintas, una en C++ y otra en Java.

Los test de unidad nos permiten asegurarnos rápidamente de que los distintos elementos de las prácticas de los alumnos cumplen los requisitos pedidos. Eso nos evita tener que comprobar directamente sobre el código si han tenido cuidado con los casos límite, comprobación de errores de lectura, etc.

Al dejar disponibles esos test a los alumnos conseguimos además que ellos mismos puedan, durante la realización de sus prácticas, comprobar si la solución a la que están llegando es correcta o han olvidado cubrir algunos casos. Esto les proporciona un *feedback* inmediato que les permite corregir el error antes de que el profesor califique la práctica.

Para asegurarnos de que los test de unidad son vistos como recomendables no sólo por nosotros como docentes sino también por los propios alumnos, hemos realizado una serie de encuestas que avalan esas afirmaciones y que también hemos presentado en el artículo.

Por lo tanto, podemos concluir que el uso de test de unidad en los laboratorios de programación es una estrategia "win-win": útil para los profesores, útil para los alumnos.

7. Agradecimientos

Financiado por el Ministerio de Educación y Ciencia (TIN2009-13692-C03-03).

Referencias

[1] Apache ant home page. `http://ant.apache.org/` (last access June, 2009).

[2] Eclipse project home page. `http://www.eclipse.org/` (last access June, 2009).

[3] Página web de checkstyle. `http://checkstyle.sourceforge.net/` (último acceso, Febrero, 2010).

[4] Página web de pmd. `http://pmd.sourceforge.net/` (último acceso, Febrero, 2010).

[5] E. G. Barriocanal, M. Ángel Sicilia Urbán, I. A. Cuevas, and P. D. Pérez. An experience in integrating automated unit testing practices in an introductory programming course. *SIGCSE Bulletin*, 34(4):125–128, 2002.

[6] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 1999.

[7] K. Beck. *Test Driven Development: By Example*. Addison-Wesley Professiona, 2002.

[8] I. F. Darwin. *Checking C Programs with Lint*. O'Reilly Media, Inc., 1988.

[9] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addi-

son Wesley Professional, Massachusetts, USA, 1995.

[10] M. A. Gómez-Martín and P. P. Gómez-Martín. Fighting against the 'but it works!' syndrome. In M. C. Azevedo-Gomes, A. Mendes, and M. J. Marcelino, editors, *XI International Symposium on Computers in Education (SIIE 2009)*, 2009.

[11] M. A. Gómez-Martin, G. Jimenez-Díaz, and J. Arroyo-Gallardo. Teaching design patterns using a family of games. In *14th ACM SIGCSE Annual Conference on Innovation and Technology in Computer Science*, page In Press. ACM Press, 2009.

[12] J. Link. *Unit Testing in Java: How Tests Drive the Code*. Morgan Kaufman, 2003.

[13] L. Malmi, A. Korhonen, and R. Saikkonen. Experiences in automatic assessment on mass courses and issues for designing virtual courses. *SIGCSE Bull.*, 34(3):55–59, 2002.

[14] J. Shore and S. Warden. *The Art of Agile Development*. O'Reilly Media, Inc., 1st edition, 2007.

[15] J. Spacco, D. Hovemeyer, W. Pugh, F. Emad, J. K. Hollingsworth, and N. Padua-Perez. Experiences with marmoset: designing and using an advanced submission and testing system for programming courses. In *Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education*, pages 13–17, Bologna, Italy, 2006. ACM.

[16] P. Tahchiev, F. Leme, V. Massol, and G. Gregory. *JUnit in Action*. Manning Publications, 2008.