Una experiencia de generación personalizada de ejercicios de prueba del software

Agustín Cernuda del Río
Departamento de Informática
Universidad de Oviedo
guti@uniovi.es

Resumen

Las pruebas del software no son sólo una verdadera asignatura pendiente en la industria del desarrollo, sino que durante la etapa académica también resulta difícil que los alumnos escriban pruebas significativas y desarrollen el hábito adecuado para su realización.

A fin de paliar este problema, es posible generar, a partir de una única especificación y caso de estudio, ejercicios de prueba del software personalizados, con medios muy limitados y de uso libre y sin una gran inversión de tiempo del profesor. Este enfoque permite mejorar notablemente algunos indicadores del trabajo de los alumnos, crear una situación de aprendizaje más controlada por parte del profesor y, presumiblemente, producir un aprendizaje más significativo aprovechando con mayor eficiencia el tiempo de los alumnos.

1. Introducción

Es conocido que las pruebas del software son una fase del ciclo clásico de desarrollo que sufre de muy notable desatención; son muy sensibles a la presión de planificación [7], y también se ven afectadas por factores humanos que hacen que los desarrolladores no adopten la actitud necesaria para realizarlas con éxito. Las pruebas unitarias son un caso particularmente claro de esto [9]. Parece clara la necesidad de insistir en una formación eficaz de los ingenieros en informática sobre pruebas del software, y sobre pruebas unitarias en particular.

La enseñanza de pruebas del software suele tropezarse con problemas típicos, que aparecen también en explotación: los estudiantes realizan pruebas irrelevantes [10], realizan pruebas dependientes de la implementación [5][6], no son agresivos cuando prueban sus propios programas (creen que las pruebas "tienen éxito cuando revelan que no hay fallos"), se dan casos frecuentes de copia de ejercicios [5], etc. El problema se aprecia incluso en los proyectos fin de carrera, donde no se puede negar que muchos estudiantes actúan exactamente como se hace en la industria. En [3] se estudia una base de 30 proyectos fin de carrera reales, y en 7 de ellos (nada menos que el 23%) no se ofrece información alguna sobre pruebas. Sólo en un 20% del total se hicieron pruebas automatizadas, y sólo en el 28% de estos se incluye el código fuente de las mismas (un 7% del total de proyectos estudiados). Sólo el 17% del total incluye pruebas de regresión.

En nuestro caso se pretendía realizar unos ejercicios de pruebas unitarias relativamente sencillos (la materia ocupa sólo unas pocas horas dentro de una asignatura mayor), pero que resultaran mínimamente significativos, transmitieran al menos en parte el valor de las pruebas unitarias y la actitud necesaria para realizarlas, y evitaran los problemas mencionados. Esto parecía exigir un trabajo desmesurado por parte del profesor, diseñando ejercicios individualizados (para evitar la copia) y creando gran cantidad de enunciados, o bien multitud de implementaciones diferentes del mismo problema, o bien un enunciado y un diseño lo suficientemente flexibles para generar "familias de programas".

Sin embargo, se comprobó que con una carga de trabajo moderada, y con unos objetivos modestos y muy concretos, es posible crear ejercicios personalizados y paliar en cierta medida los problemas expuestos.

2. Situación de partida

2.1. Entorno

En este caso se trata de una asignatura del primer curso del Máster en Ingeniería Web de la Escuela Universitaria de Ingeniería Técnica en Informáti-

ca de Oviedo. Este máster tiene algunas asignaturas "de nivelación", entre ellas Programación Orientada a Objetos (POO), en la cual se incluyen dos días de clase sobre pruebas del software (6 horas presenciales y aproximadamente 10 de trabajo autónomo del alumno).

En esos dos días se habla de las pruebas en general (tipos según diversas clasificaciones), del ciclo de desarrollo y escritura de pruebas, de pruebas unitarias automatizadas mediante herramientas como JUnit, y otros temas. Se insiste en la necesidad y utilidad de las pruebas, se demuestra cómo es el ciclo de desarrollo dirigido por pruebas (*test driven development*), y se encaja la escritura de pruebas dentro de la escritura habitual de código, mostrando cómo minimizar el coste al hacer sólo pruebas significativas y tener un lugar para conservar ese código que habitualmente el programador escribe de todos modos y luego borra para continuar.

En ediciones anteriores (2006, 2007, 2008) la evaluación de este módulo tenía dos partes. Una de ellas (que es la que interesa aquí) obligatoria, consistente en la realización libre de un banco de pruebas con JUnit o NUnit sobre algunos módulos a elección del alumno (podían ser módulos desarrollados para otra parte de la asignatura). Con este ejercicio se podía aprobar el módulo de pruebas; había una segunda parte opcional, que podía ser la realización de un banco de pruebas web con HTTPUnit o algún otro trabajo abierto sobre otras tecnologías de prueba.

2.2. Objetivos de los ejercicios

En la parte práctica del módulo se pretende que:

- Los alumnos experimenten el ciclo de escritura de código de pruebas unitarias.
- Adquieran el concepto básico de que un defecto es una incoherencia entre especificación y comportamiento, y por tanto interioricen la importancia de las especificaciones.
- Encuentren realmente algún defecto con sus pruebas, si es posible (y así se indicaba en el enunciado).
- Perciban que un defecto encontrado tiene su código de prueba asociado (mentalidad favorable a las pruebas regresivas).
- No escriban pruebas irrelevantes, "de relleno", copiadas y pegadas; que no perciban las pruebas como un requisito burocrático.
- Se acostumbren a pensar en pruebas unitarias automáticas, y abandonen la idea de imprimir valores por pantalla para que un programador lo inspeccione visualmente.

2.3. Inconvenientes del método

El método de ejercicio abierto que se venía utilizando resultaba muy discutible en sus resultados didácticos; realmente, no se conseguía que los alumnos escribieran pruebas significativas. Ni las explicaciones, ni las demostraciones, ni las directrices escritas, ni las admoniciones consiguieron transmitir al alumno qué se esperaba de él. En los ejercicios presentados, aparecieron diversos problemas muy extendidos:

- Los alumnos no siempre tienen buenas ideas ni escogen el problema adecuado.
- Apenas encuentran defectos.
- No ejercitan tampoco la creación de casos de prueba regresivos para prevenir la reaparición de defectos encontrados por otra vía.
- Escriben una notable cantidad de código, pero las pruebas en general son poco significativas y muy rutinarias.
- Suelen utilizar los mismos casos de estudio (otros ejercicios de programación de la asignatura), y resulta difícil prevenir las copias.
- No tienen buenas especificaciones de esos casos de estudio; por tanto, la influencia de las mismas en las pruebas es mínimo. Ningún alumno, en tres ediciones del módulo, desarrolló una especificación para trabajar a partir de ella, por mucho que diga la teoría.
- En ocasiones caen en el error de hacer trazas (impresión por pantalla) en vez de pruebas totalmente automatizadas.

3. Nuevo sistema

3.1. Motivación

Parece evidente que el enfoque de "explicar cómo hay que hacerlo" y después "ordenar hacerlo" no ha funcionado, y este es un problema tradicional en las pruebas y otras buenas prácticas de ingeniería del software. Hacer explícitos los objetivos y luego pedir que se cumplan no resuelve el problema; hay que promover que los objetivos estén también implícitos en la actividad, de modo que en la medida de lo posible no se pueda hacer

esta sin cumplir, al menos en cierta medida, los objetivos. Y en el caso de las pruebas esto nos lleva a un clásico ejercicio de *bug hunt*, pero convenientemente adaptado.

Repasando los problemas descritos en el apartado 2.3, se decidió diseñar la actividad de la manera siguiente:

- *Diseñar un solo caso de estudio concreto, creado por el profesor*. Esto puede evitar la elección de módulos poco interesantes para las pruebas, o el "prejuicio del desarrollador" que surge al probar el código propio. También permitirá centrar y dimensionar el esfuerzo.
- *El dominio del problema será ajeno a los estudiantes*. En la medida de lo posible, se buscará un caso de estudio desconocido por los alumnos. Esto requerirá emplear bastante tiempo de clase para introducir y explicar el problema, pero a cambio se conseguirá que los estudiantes dependan mucho más de la especificación, y también que realicen las pruebas sin el prejuicio del desarrollador.
- *Los estudiantes tendrán que encontrar defectos introducidos deliberadamente por el profesor*. Además de ayudar a enfocar el esfuerzo y concretar el ejercicio, los estudiantes presumiblemente verán más utilidad a las pruebas. El objetivo del ejercicio será muy claro: encontrar defectos cuyo número se conoce de antemano (defectos que describirán en su informe final), no presentar unas líneas de código inútil.
- *El código de pruebas deberá indicar inexcusablemente qué caso de prueba pone de manifiesto qué defecto*. En el código de pruebas habrá pruebas que no encuentren ningún defecto, pero todos los defectos encontrados deben aparecer señalados con comentarios en el código de pruebas. Si el alumno encuentra un defecto por mera inspección visual, debe igualmente escribir código de pruebas para él antes de arreglarlo. Así se percibe la importancia de las pruebas regresivas y el ciclo adecuado de desarrollo cuando aparece un defecto ("probarlo primero y arreglarlo después").

3.2. Inconvenientes

Por supuesto, este enfoque plantea algunas dificultades para el profesor.

- *Mayor trabajo en la preparación de enunciados*. Hay que preparar un caso de estudio, una especificación del problema y su dominio que se pueda usar como base para las pruebas, y una exposición de ello en clase.
- *Tiempo de clase*. Como ya se ha indicado, introducir el dominio del problema requiere un tiempo de clase que hay que restar a las exposiciones. Pero, ya que exposiciones más largas no ayudaron a conseguir los objetivos planteados en 2.2, parece lógico correr el riesgo de "impartir" menos materia.
- *Calidad del código de estudio*. Si el caso de estudio tiene defectos introducidos deliberadamente, y se pretende tener pleno control sobre la tarea de los estudiantes y en qué medida la han cumplido, el código original debe estar libre de errores, para que sólo estén allí los errores controlados. Esto exige que el profesor valide concienzudamente su propio código antes de introducir los defectos deliberados para preparar los ejercicios.
- *Prevención ante copias*. Si todos los alumnos tienen exactamente el mismo objeto de sus pruebas, parece difícil evitar el fraude, y evidentemente no se aprende lo mismo averiguando algo que siendo informado de algo; precisamente la experiencia práctica con las pruebas, su viabilidad y su utilidad es el objetivo didáctico del ejercicio. Así que los defectos introducidos tienen que ser distintos para cada alumno, y además el profesor tiene que tener una relación de los mismos para la corrección. Esto implica aún más trabajo.
- *Incertidumbre sobre los resultados*. Siendo esta la primera experiencia, resulta difícil saber si los alumnos encontrarán los defectos, o cuántos de ellos; también cómo se comportarán en general manejando una especificación sobre un tema que les es ajeno.

Para asumir la carga de trabajo adicional, el mayor inconveniente es la generación de ejercicios individualizados, pero parece claro que la solución pasa por automatizar el proceso. Existen infinidad de sistemas tutores [2] o de apoyo [1] para enseñar pruebas del software, pero en ocasiones (como en este caso) no es viable instalar

algo complejo, ni su uso se adapta bien al tiempo disponible, y además no se resuelve la necesidad de generar ejercicios personalizados. También es interesante el enfoque de [1] en el que se integra las pruebas y el desarrollo, o el de [8]; pero no son aplicables tampoco a nuestro caso planteamientos didácticos específicos como estos, ya que no estamos en una asignatura introductoria ni en un curso completo, sino en un módulo aislado de una asignatura "de repaso" para alumnos de cursos superiores, con muy poco tiempo y unos objetivos muy concretos. Lo que se pretende aquí es algo tremendamente sencillo, autónomo, que no obligue a los estudiantes ni al profesor a utilizar herramientas web ni adherirse a determinados planteamientos educativos.

4. Implementación

4.1. Generación automática de variantes de ejercicios

Se decidió desarrollar de manera rápida y al menor coste posible un soporte software para generar estas variantes de ejercicios. De lo que se trata es de tener unos ficheros fuente que funcionan bien, y de introducirles aleatoriamente defectos. Lógicamente, deben ser defectos verosímiles y que no den lugar a errores de compilación, por lo que no se puede estropear los ficheros fuente de forma totalmente aleatoria.

El primer paso es tener una versión plenamente estable del código fuente del caso de estudio. Todos los defectos, tanto los introducidos por el profesor como los encontrados por los alumnos, van a basarse en números de línea, con lo que es fundamental que los fuentes no se modifiquen después de generar los ejercicios. Por supuesto, no importa que los alumnos lo hagan para ir corrigiendo los que encuentran, pero hay dos salvedades: una, que en todo caso corregir un error implica modificar una línea, sin necesidad de alterar el número de estas, y dos, que se advierte a los alumnos de que deben referirse a números de línea "originales". De hecho, no se les exige que corrijan los defectos (es decisión suya); sólo que los encuentren.

4.2. Bancos de defectos

Sobre esta versión estable, el profesor escribe unos "bancos de defectos" para cada fichero fuente. Estos defectos se escriben en un formato XML que facilitará el tratamiento posterior.

Un ejemplo de esto se ve en la Figura 1. Ese defecto afectaría a la línea 54 de cierto fichero fuente. El texto original, es decir, cuando no hay *bug*, es la inicialización `m_tipo = flags;` el texto erróneo, cuando "sí" hay *bug*, es `m_tipo = 0;`

```
<bug linea="54">
   <no>m_tipo = flags;</no>
   <si>m_tipo = 0;</si>
</bug>
```

Figura 1. Un defecto del banco de defectos escrito por el profesor.

No es imprescindible escribir cómo era el texto original (la sección **<**`no`**>**) pero ayudará a evitar errores, ya que el sistema (véase 4.3) dará un aviso si en la línea 54 no encuentra lo esperado.

Así, el profesor escribió una base de unos 75 defectos. Escribir directamente en XML (y en ocasiones con secciones `CDATA`) no es lo ideal, pero es aceptable y no resulta difícil copiar y pegar la información repetida. En cualquier caso, tampoco sería muy costoso preparar un sistema de edición de esos defectos más cómodo, una vez comprobado que merece la pena.

4.3. Generación de un ejercicio

Basta escribir un sencillo programa (en Java en este caso) que realiza un "sorteo" y genera una selección de 10 errores, en XML.

A continuación, mediante transformaciones XSLT, a partir de la información de esos errores se genera un fichero de texto en formato "patch" (véase la Figura 2). Se trata de un mecanismo utilizado típicamente en entornos Linux y similares para aplicar parches de actualización, y permite dar instrucciones al programa `patch` sobre cómo debe modificar un fichero de texto; en este caso, esas modificaciones consisten precisamente en introducir los *bugs* en el código fuente del ejemplo, y el fichero patch contiene los mismos errores del fichero XML, convenientemente formateados.

```
@@ -54,1 +54,1 @@
-       m_tipo = flags;
+       m_tipo = 0;
```

Figura 2. El defecto de la Figura 1, transformado al formato *patch* que se aplicará al código fuente.

También se genera un enunciado, en formato HTML, parcialmente personalizado para el alumno, en el que se le indica cuántos defectos tiene que buscar en cada clase.

Finalmente, se empaqueta el código fuente (ya con los errores) y el enunciado, y ese es el fichero que se enviará al alumno. Como puede verse, para generar todos los ejercicios de golpe basta con un pequeño programa (Java en este caso), utilidades de línea de órdenes (`patch`, ficheros `.bat` o similares, `gzip`), la lista de alumnos y un transformador de XSLT.

4.4. Caso de estudio

Todo lo anterior es genérico y perfectamente aplicable en otras asignaturas o centros; lógicamente, el enunciado concreto del ejercicio de 2009 no tiene por qué ser trasladable directamente, ni por su tamaño ni por su temática.

De todas maneras, a modo de ejemplo puede tener interés comentar que se trata de una pequeña biblioteca de armonía musical. Su función es generar las notas que forman parte de un acorde (conjunto de notas que suenan "bien" simultáneamente) de cuatriada (es decir, conjuntos de 4 notas). Por supuesto, puede haber alumnos para los que este dominio del problema sea conocido, pero en principio se puede suponer que la mayoría de ellos son extraños a este ámbito.

De modo que si se invoca a una función de la biblioteca para pedirle las notas que forman parte del acorde "sol menor séptima" (y que se codifica como G^{-7}), esta responderá con un objeto que, impreso por pantalla, mostrará las notas G, Bb, D, F (en notación americana).

Este puede ser un buen ejercicio porque la armonía musical no es más que una forma de aritmética sujeta a reglas bien precisas; se trata de calcular las distancias entre la nota de partida y las otras tres que la acompañan en el acorde. Pero estas reglas de aritmética están llenas de trampas ocultas y excepciones; la distancia entre dos notas consecutivas es de un tono en unos casos y un semitono en otros, la nomenclatura de los acordes dista mucho de ser consistente...

Además de explicar en clase los fundamentos de todo esto, con ejemplos prácticos e incluso escuchando ejemplos en algunos temas musicales, se suministró a los alumnos una descripción detallada, que sirve como especificación, de una página de longitud. Puede verse un fragmento de esa especificación en la Figura 3.

Intervalos

Intervalo	Notas	Semitonos
Tercera menor	2	3
Tercera mayor	2	4
Quinta disminuida	4	6
Quinta justa	4	7
Quinta aumentada	4	8
Séptima "doble bemol"	6	9
Séptima	6	10
Séptima mayor	6	11

Ejemplo: tanto la quinta justa como la quinta disminuida de C son forzosamente un G.
En concreto, la quinta justa es G (natural), y la quinta disminuida es Gb.
Aunque F# suena igual que Gb (son "la misma nota"), no sería correcto llamar F# a la quinta disminuida de C.

Acordes

	3ª	5ª	7ª
m	menor		
7			séptima
Maj7			sépt. mayor
b5		disminuida	
#5		aumentada	
Semidisminuido	menor	disminuida	séptima
Disminuido	menor	disminuida	doble bemol

Si no se dice nada sobre la tercera, se asume que es mayor.
Si no se dice nada sobre la quinta, se asume que es justa.

Figura 3. Un fragmento de la especificación.

El caso de estudio, finalmente, tiene 7 clases en Java, con un total de menos de 800 líneas (incluyendo los muy abundantes comentarios).

5. Resultados

Una vez presentados y corregidos los ejercicios del primer cuatrimestre de 2009-2010 (que aquí llamaremos "de 2009"), fue posible contrastar los

resultados con los ejercicios de años anteriores. Se tomó como medida de control principal la del curso precedente, 2008-2009 (en adelante, "de 2008").

Una vez descartados los ejercicios irregulares o no computables por alguna razón, se dispuso de 30 ejercicios de 2008, y 34 de 2009, lo que parece una cantidad razonable para sacar conclusiones. Los términos del ejercicio y el perfil de los alumnos son similares en ambos casos, salvo en lo dicho aquí.

Primero se evaluarán algunos aspectos directamente medibles, para pasar luego a discutir otros más bien cualitativos.

5.1. Defectos encontrados

En este apartado, lo cierto es que la comparación se reduce prácticamente a ver si los alumnos de 2009 encontraron o no los defectos ocultos, porque en la edición de 2008 no merece la pena hacer muchas estadísticas; el número de defectos encontrados en los ejercicios auto-planteados fue prácticamente anecdótico (4 defectos encontrados en total por 30 estudiantes), y la media es muy cercana a cero.

En 2009 estaba muy claro que había defectos que encontrar, y los alumnos sabían cuántos había en cada clase; finalmente, casi todos los alumnos encontraron los 10 defectos.

Más aún; 2 de los 34 alumnos encontraron independientemente un defecto adicional (el mismo) que había quedado oculto a las pruebas previas del profesor, cosa que puede calificarse como un notable éxito.

5.2. Código relevante

Como se estableció en 2.2, es deseable que los alumnos no escriban "código de relleno", que prueba cosas irrelevantes, o sea una sucesión de copias levemente modificadas del mismo caso de prueba, sino que el código esté realmente destinado a encontrar diversos defectos.

Esto es difícil de medir en algunos sentidos, pero se puede realizar una aproximación. Los datos están resumidos en la Tabla 1.

La primera conclusión, ya mencionada, es que los alumnos han encontrado una media de defectos cercana a 10 (si contamos sólo los aprobados la media es 9,8), mientras que en 2008 casi fue 0; no obstante, lo han conseguido escribiendo aproximadamente la mitad de líneas de código.

Al mismo tiempo, ha subido el número de métodos de prueba. Este dato, aunque por sí solo no es concluyente, parece un indicador de que las pruebas están mejor organizadas y divididas; la mitad de código, repartido en más métodos, arroja claramente muchas menos líneas de código por método, es decir, mayor modularidad en las pruebas, y eso es un buen síntoma.

	2008	2009	Δ
Líneas de código	380,93	185,29	-51%
Métodos de prueba	10,24	13,85	+37%
Asertos	35,20	32,03	-9%
Anotaciones `expected`	2,30	1,70	-26%
Ejercicios con `print`	40%	24%	
Líneas `print` por ejerc.	8,00	2,50	-69%
Defectos encontrados	0,13	9,37	
Horas reportadas	6,87(4)	8,25(12)	
Nota media	6,32	5,55	
Desv. estándar nota	0,86	1,6	

Tabla 1. Métricas de los ejercicios. En todo caso se refieren al código de pruebas escrito por los alumnos.

Respecto a los asertos (y hablamos de asertos de JUnit, es decir, expresiones de "resultado esperado") su número ha bajado ligeramente en términos absolutos, pero en realidad ha subido mucho en proporción a las líneas totales de código, y esos asertos han servido para encontrar muchísimos más defectos. Por tanto, a falta de evaluar casos individuales, también parece claro que esos asertos son mucho más significativos.

La cifra "anotaciones `expected`" se refiere a un tipo particular de comprobaciones para verificar si se genera una excepción cuando debería, de acuerdo con las especificaciones. Su número ha bajado. La comparación en este caso no es muy significativa ni fácil de interpretar, ya que las anotaciones `expected` son pocas de todos modos, y es probable que su número sea muy sensible al enunciado en concreto.

En conclusión, los indicadores objetivos que tenemos para la relevancia del código escrito parecen demostrar claramente que se ha hecho un trabajo de pruebas mucho más significativo con mucho menos código, y que las cosas han ido en la dirección correcta.

5.3. Sentencias `print`

Uno de los problemas identificados en 2.2 era cierta falta de rigor en la automatización de las pruebas, ya que los alumnos tendían a imprimir por pantalla valores para su inspección visual, en este caso mediante órdenes `print`. Tanto en 2008 como en 2009 se hicieron las habituales advertencias al respecto, pero con resultado desigual.

La Tabla 1 contiene también datos sobre el uso de este tipo de trazas en el código de pruebas. En 2008, un 40% de los alumnos utilizaron alguna sentencia `print`, lo cual es un mal resultado. Restringiendo la medición a estos, utilizaron una media de 8 de estas sentencias.

Sin embargo, en 2009, sin adoptar ninguna medida específica para evitarlo, el porcentaje de alumnos que utilizaron `print` en el código de pruebas cayó al 24%, y limitándose a estos, utilizaron una media de 2,5 sentencias.

Parece claro que el planteamiento del ejercicio influye en esto. Ya que el entregable principal es la lista de los defectos, dónde están y qué código de prueba los revela, las instrucciones `print` son inútiles para ese objetivo principal. Los alumnos saben que todo el código de pruebas que escriban importa, pero si no encuentran los defectos y no los revelan con asertos, faltará el mínimo exigido. En consecuencia, parece que caen menos en el error de escribir código de verificación por pantalla, lo cual es otro efecto claramente positivo.

Aun así, parece aconsejable introducir una instrucción específica en el enunciado, de modo que esté "prohibido" imprimir por pantalla.

5.4. Tiempo empleado

La información de tiempo empleado por el alumno es dudosa, porque se basa simplemente en el dato que este incluye en la documentación. Además, no es un dato obligatorio, por lo que hay pocas mediciones. En 2009 hay datos de 12 alumnos, que afirman haber empleado en los ejercicios una media de 8,25 horas. En 2008 sólo 4 alumnos informaron sobre el particular, siendo la media 6,875 horas.

La conclusión importante es que efectivamente un ejercicio de este tipo no tiene por qué disparar el tiempo de realización. Nótese que en ambos casos el tiempo está dentro de las 10 horas teóricamente disponibles (en muchos casos, los alumnos no hacen la parte opcional, ya que van directamente a por los objetivos mínimos).

5.5. Nota media

Simplemente para tener en cuenta el perfil de los alumnos, en 2008 la nota media de este módulo (sobre alumnos presentados) fue 6,32, con una desviación estándar de 0,86. En 2009 la media fue algo menor, 5,55, con desviación estándar mayor: 1,6. Si nos centramos sólo en los alumnos de 2009 que hicieron la parte opcional, fueron 9 (alrededor de una cuarta parte del total) y su nota media fue 8,55.

Es decir, que a grandes rasgos las calificaciones son similares, si bien se aprecia (también a simple vista) que en 2009 hay una variación mayor; muchos alumnos se centran en el objetivo mínimo, muy claro, mientras que en 2008 había más alumnos que hacían un ejercicio mediocre en el enunciado mínimo y les sobraba tiempo para hacer algo también en la parte opcional. Esto distorsionaba un poco las calificaciones; en 2009 son más estables.

6. Conclusiones

A la vista de los resultados, las conclusiones han de ser positivas. Por supuesto, es difícil saber hasta qué punto la formación de los estudiantes en técnicas de pruebas está funcionando de cara al futuro, pero como mínimo se ha conseguido un doble objetivo.

Por una parte, se ha resuelto el problema práctico de generar un conjunto de ejercicios personalizados, mediante el uso de transformaciones XSLT y ficheros *patch*, que puede utilizarse en cualquier ámbito. Gracias a esto, no es necesario hacer complejos diseños o "familias de productos" para poder hacer ejercicios distintos para cada alumno; se puede trabajar sobre un único enunciado, y generar los ejercicios con herramientas gratuitas y muy sencillas.

Por otra parte, de esta facilidad se derivan una serie de beneficios didácticos, que pueden comprobarse comparando las características de los ejercicios generados con los objetivos marcados en el apartado 2.2. También se puede comprobar, en los resultados obtenidos, que la experiencia parece demostrar la consecución de al menos algunos de esos objetivos.

Además de esos dos beneficios mensurables, es lógico pensar que el enunciado de 2009 resulta más claro para los alumnos; saben exactamente lo que tienen que hacer, saben que hay 10 defectos, y no pierden el tiempo cumpliendo un trámite burocrático de escritura de código, sino haciendo un entrenamiento. Claro está que esta no es del todo una situación real, ya que falta el factor de incertidumbre sobre si hay o no defectos, y cuántos; pero ese inconveniente es muy relativo, ya que el objetivo de este ejercicio era incidir en ciertas habilidades técnicas concretas, y una simulación totalmente fidedigna del trabajo de desarrollo requeriría un entorno mucho más complejo y amplio (un proyecto mayor, cierta presión de planificación, etc.).

Referencias

[1] Barbosa, Ellen F.; Silva, Marco A. G.; Corte, Camila K. D.; Maldonado, José C. *Integrated Teaching of Programming Foundations and Software Testing*. FIE (Frontiers in Education) 2008, Saratoga Springs, New York.

[2] Elbaum, Sebastian; Person, Suzette; Dokulil, Jon; Jorde, Matt. *Bug Hunt: Making Early Software Testing Lessons Engaging and Affordable*. Education Papers, ICSE 2007 (Minneapolis).

[3] Fernández Fernández, Eva María. *Estudio de defectos latentes de una base de proyectos y aplicación de técnicas de prueba*. Proyecto Fin de Carrera, Escuela Universitaria de Ingeniería Técnica en Informática de Oviedo, septiembre de 2007

[4] JUnit. http://www.junit.org/

[5] López, Carlos; Marticorena, Raúl; Martín, David H. *Pruebas de Caja Negra: Una Experiencia Real en Laboratorio*. JENUI 2005, actas del congreso.

[6] Marticorena, Raúl; López, Carlos; Crespo, Yania. *Estudio de la Distribución Docente de Pruebas del Software y Refactoring para la Incorporación de Metodologías Ágiles*. JENUI 2004, actas del congreso.

[7] McConnell, Steve. *Rapid Development: Taming Wild Software Schedules*. Microsoft Press, 1996. ISBN: 1-55615-900-5.

[8] Patterson, Andrew; Kölling, Michael; Rosenberg, John. *Introducing Unit Testing With BlueJ*. Proceedings of the 8th conference on Information Technology in Computer Science Education (ITiCSE 2003), Tesalónica, 2003

[9] Shroeder, Patrick J.; Rothe, Darrin. *Teaching Unit Testing using Test-Driven Development*. 4th Workshop on Teaching Software Testing, Melbourne, Florida, 2005.

[10] Tuya, Javier; de la Riva, Claudio; García Fanjul, José. *A laboratory exercise in testing database applications*. 7th Workshop on Teaching Software Testing, Melbourne, Florida, 2008.