Hacia un enfoque metodológico de cobertura múltiple para refactorizaciones más seguras

Carlos Fontela[1], Alejandra Garrido[2], Andrés Lange[1]

[1] Fac. de Ingeniería Universidad de Buenos Aires, Argentina

[2] Fac. de Informática, Univ. Nac. de La Plata y CONICET, Argentina

cfontela@fi.uba.ar, garrido@lifia.info.unlp.edu.ar, andres.lange@gmail.com

Resumen

Habitualmente se considera que la preservación del comportamiento del software después del refactoring puede ser verificada con pruebas unitarias automatizadas. Sin embargo, hay situaciones en las cuales estas pruebas dejan de funcionar precisamente por los cambios realizados en el código. Para agregar seguridad al refactoring es necesario entonces utilizar pruebas a distintos niveles que aumenten la redundancia en la cobertura, aunque esto agrega complejidad adicional al testing y al refactoring. Para mitigar este problema hemos desarrollado un enfoque metodológico basado en distintos niveles de prueba con cobertura redundante. Asimismo este artículo presenta una herramienta de chequeo de cobertura simultánea de varios tipos de pruebas, y un caso de estudio donde se usa la misma. Más allá de las ventajas conocidas de contar condistintos niveles de pruebas, una adicional es la colaboración que prestan en situaciones de refactoring, por lo que se esperaría una adopción mayor en la industria.

Palabras clave

Refactoring, preservación del comportamiento, cobertura, automatización, pruebas de aceptación

1. Introducción

El refactoring o refactorización de código es una práctica de la Ingeniería de Software que busca mejorar la calidad del código con vistas a facilitar su mantenimiento, pero sin alterar el comportamiento observable del mismo. La refactorización se aplica transformando un sistema una vez que se desarrolló su funcionalidad y la misma ya está codificada.

El problema con las refactorizaciones es que son riesgosas, ya que las mismas cambian código que se encuentra en correcto funcionamiento por otro, que supuestamente mejora algún atributo de calidad, pero no sabemos si cambiará o romperá el comportamiento existente. Para permitir refactorizaciones más seguras, una buena práctica es trabajar con pruebas unitarias automatizadas, escritas antes de refactorizar, y correrlas después de cada paso de transformación, para asegurarse de no haber introducido un error.

Sin embargo, resulta ingenuo creer que las pruebas unitarias pueden eliminar, o incluso mitigar, los riesgos de la refactorización. Hay ocasiones en que las pruebas resultan afectadas por las propias refactorizaciones y fallan, causando que la funcionalidad no pueda ser verificada a posteriori. Esto es más evidente cuanto menor sea la granularidad de las pruebas y cuanto mayor sea el alcance de la refactorización; aun cuando se trabaje con pruebas que integren varias clases o componiendo refactorizaciones más pequeñas, el problema sigue sin resolverse. Si bien algunos trabajos de investigación plantean el problema (por ej, [1]), distan mucho de resolverlo con un enfoque metodológico completo y seguro. Dado que este es uno de los supuestos más fuertes de la práctica del refactoring, nos hemos planteado encontrar una práctica metodológica que lo refuerce.

Hoy en día se utilizan distintos tipos o niveles de prueba, a saber: de unidad, técnicas de integración y de aceptación o comportamiento. Como explicaremos más adelante, en nuestro planteo utilizamos varios niveles de pruebas para cubrir el riesgo de incorrección de ciertas refactorizaciones. De esta manera, un nivel de pruebas de mayor granularidad va a servir como red de contención de pruebas que dejen de funcionar ante algunas refactorizaciones. En el nivel más alto de la red de contención se encuentran las pruebas de aceptación, que por su propia definición no deberían cambiar ante una refactorización, y así operan como garantía de refactorizaciones seguras, independientemente del alcance de las mismas.

Llamamos cobertura al grado en que los casos de pruebas de un programa llegan a recorrer dicho programa. Al plantearnos el problema de la cobertura de pruebas a distintos niveles, surgió la necesidad de contar con una herramienta que, en el marco del lenguaje y del entorno de desarrollo que estábamos utilizando, nos indicase cuáles pruebas cubrían cierta porción de código – el código a refactorizar – en forma simultánea. De esta manera, la herramienta nos indicaría cuáles son las pruebas de aceptación que asegurarían la corrección de las refactorizaciones ante la rotura de las pruebas de unidad. Así es que comenzamos a desarrollar *"Multilayer Coverage"* para Java, como una extensión a la herramienta de cobertura EclEmma, para el entorno de desarrollo Eclipse.

El presente trabajo se plantea también la necesidad de existencia de pruebas de aceptación automatizadas. Como sabemos, hay una práctica denominada Acceptance Test Driven Development (ATDD), que preconiza las ventajas de desarrollar aplicaciones siguiendo el protocolo de TDD [2, 3] utilizando como punto de partida a las pruebas de aceptación. ATDD aún no se encuentra tan difundida, pero nuestro planteo de la necesidad de pruebas de aceptación automatizadas para lograr refactorizaciones más seguras, es un argumento más en favor de la práctica de ATDD.

2. Trabajos relacionados

En resumen, hemos visto que hay ocasiones en que una refactorización es causa de que las pruebas unitarias se rompan o fallen, dejando a dicha refactorización sin su red de contención natural.

Según Pipka [1], el enfoque más habitual dentro del desarrollo dirigido por pruebas es realizar la refactorización y ver si las pruebas siguen funcionando. Si ninguna prueba se rompe, podemos seguir adelante. Si, en cambio, alguna prueba deja de compilar o falla, cambiamos esa prueba para que pase. El hecho de que este sea el método más frecuente suele ser consecuencia de que a los programadores no les resulta sencillo determinar de antemano qué pruebas van a fallar para un cambio dado. Pipka llama “enfoque rápido y sucio” a esta práctica, y la razón es obvia: se pierde mucho control y seguridad, al punto que la presunta reconstrucción de las pruebas no puede garantizar la preservación del comportamiento. Algunos autores han propuesto enfoques más elaborados que describimos a continuación.

2.1 Test-First Refactoring

Test-First Refactoring se basa en Test-Driven Development (TDD). TDD es una práctica iterativa e incremental de diseño de software, que fue presentada por Kent Beck y Ward Cunningham como parte de Extreme Programming (XP) en [4]. Consiste en desarrollar siguiendo un ciclo que comienza con la escritura de pruebas en código, luego el desarrollo de código que haga funcionar correctamente las pruebas, y finalmente una refactorización para mejorar la calidad de lo desarrollado.

El procedimiento de Test-First Refactoring es:

- Adaptar las pruebas antes de refactorizar.
- Comprobar que la nueva versión de las pruebas falla con el código antiguo.
- Refactorizar la aplicación.

- Correr nuevamente las pruebas en su segunda versión, que ahora deberían funcionar bien.

A este método, Pipka [1] lo llama **Test First Refactoring**. En un trabajo posterior [5] propone una ligera variante que llama **Test Driven Refactoring**. Este enfoque tiene una ventaja innegable: establece la interfaz del código a refactorizar antes de la propia refactorización, lo cual está en línea con la premisa de TDD de que especificar el uso de las clases antes de construirlas es positivo porque separa el qué del cómo. Claramente, el *Test First Refactoring* es preferible al "enfoque sucio".

Sin embargo, el *Test First Refactoring* no es la panacea. Si bien es cierto que modificar las pruebas antes que el código a refactorizar es una buena idea, es muy difícil garantizar que la nueva versión de la prueba comprueba el mismo comportamiento que antes. Además, si la refactorización no es una pequeña modificación localizada sino un cambio más amplio, la probabilidad de error crece sensiblemente.

2.2 Refactoring asegurado por los clientes

En refactorizaciones más amplias hay otra posibilidad, que es usar como red de contención las pruebas ya escritas para otras clases que dependan de la o las que están siendo cambiadas. En todo programa orientado a objetos, una clase brinda servicios a otras, a las que habitualmente denominamos *clientes* de aquélla. Si nuestra refactorización afecta a un conjunto de clases que deben preservar su comportamiento, entonces las clases que a su vez son clientes de este conjunto no deberían cambiar su comportamiento. Por lo tanto, las pruebas de las clases clientes deberían seguir corriendo sin problema.

En este caso, el procedimiento sería refactorizar, comprobar si las pruebas no se rompen, y si se rompen, eliminar provisoriamente las mismas del conjunto de pruebas, corriendo el resto de las pruebas del sistema. Si todo sigue funcionando bien, procederíamos a cambiar las pruebas que antes fallaban, para luego incorporarlas de nuevo en el conjunto de pruebas y asegurarnos finalmente que todo sigue corriendo bien. Este enfoque fue propuesto muy informalmente por Lippert y Roock [6], al decir que, en última instancia, el refactoring debería asegurarse con pruebas de integración. En este trabajo lo denominaremos **refactoring asegurado por los clientes**.

Sin embargo, el método, así enunciado, presenta algunas complicaciones. Primero, porque si las pruebas unitarias dejan de funcionar, es previsible que lo mismo ocurra con las clases clientes (al fin y al cabo, las pruebas también son clientes de la clase que se está cambiando). Por lo tanto, las propias clases

clientes deben ser cambiadas junto con las pruebas unitarias, y esto agrega una elevada complejidad a la solución. En segundo lugar, no siempre va a poder ser aplicado si estamos trabajando con algún enfoque que utilice abundantemente objetos ficticios (por ej., *Mock Objects* [7]). En efecto, los objetos ficticios buscan desacoplar las clases entre sí, para garantizar que las pruebas de los clientes funcionen a pesar de no estar vinculadas a los servidores. En estos casos, al desacoplar, perdemos esta red de seguridad provista por los clientes. A todo ello se suma el dilema de la cobertura, es decir, garantizar que el código a refactorizar sigue siendo recorrido cuando eliminamos del conjunto la o las pruebas problemáticas. De otra manera la eliminación de ciertas pruebas del conjunto va a disminuir el nivel de cobertura, y justamente en el tramo que debemos controlar. Aun con un nivel de cobertura del 100% antes de la refactorización no podríamos garantizar nada, ya que la eliminación de las pruebas que se rompen bajaría ese porcentaje.

2.3 Refactoring asegurado por pruebas de comportamiento

Con el tiempo, surgieron formas más evolucionadas de TDD, tales como Acceptance Test Driven Development (ATDD), Behavour Driven Development (BDD) y Storytest Driven Development (STDD). Éstas (que a los efectos de este trabajo vamos a considerar equivalentes) son formas de TDD que obtienen el producto a partir de las pruebas de aceptación de usuarios. De esta manera, las pruebas de aceptación de un requerimiento o funcionalidad se convierten en las condiciones de satisfacción del mismo, y los criterios de aceptación se convierten en especificaciones ejecutables.

Los impulsores de ATDD, afirmaron en diversos trabajos que las pruebas de comportamiento (entendidas como pruebas de aceptación que no incluyen la interfaz de usuario) sirven para garantizar refactorizaciones más seguras. Así lo afirmaron, por ejemplo, Mugridge y Cunningham [3]. También Pipka, en su segundo trabajo sobre el tema [5], propone partir de requerimientos, aunque plantea dividir los requerimientos en “tareas técnicas que puedan expresarse en pruebas unitarias” en vez de usar pruebas de aceptación.

En efecto, si la condición a verificar de una refactorización para garantizar su corrección es la preservación del comportamiento, lo mejor sería mantener pruebas de comportamiento, en el sentido de ATDD, y correrlas en cada refactorización para asegurarnos que siguen funcionando. En este caso, las pruebas de comportamiento estarían cumpliendo el rol de red de contención del refactoring. En consecuencia, podríamos proceder así:

- Refactorizar y determinar si las pruebas técnicas (unitarias y de los clientes) no se rompen.

- Si algunas pruebas se rompen, eliminar provisoriamente las mismas del conjunto de pruebas.
- Correr las pruebas de comportamiento, y asegurarse de que no se rompan.
- Cambiar las pruebas que antes fallaban, para luego incorporarlas de nuevo al conjunto de pruebas y asegurarnos finalmente que todo sigue corriendo bien.

Aunque esta parece una solución impecable, no está exenta de problemas. Uno de ellos es el escaso uso industrial de ATDD o sus variantes. Si el sistema no se hizo siguiendo el protocolo de ATDD, difícilmente tengamos pruebas de aceptación automatizadas que nos sirvan para este caso. Si, en cambio, se siguió el protocolo de ATDD, pero sin desarrollar suficientes pruebas de menor granularidad, sólo tendremos pruebas muy amplias, que verifican mucho código fuente cada una. Similarmente al enfoque previo, se presenta el dilema de la cobertura, que en este caso sería cómo garantizar que las pruebas de comportamiento recorren el mismo código que las pruebas técnicas. Además, al incluir ATDD pruebas mucho más amplias, es más difícil ver la correlación entre pruebas de comportamiento y pruebas técnicas.

Como hemos visto, ninguno de estos enfoques ofrece una respuesta integral. Esa es la razón que motiva este trabajo.

3. Refactoring con pruebas por niveles

Lo central del método propuesto es que se basa en las prácticas de refactoring asegurado por los clientes y refactoring asegurado por pruebas de comportamiento, más un análisis de cobertura.

Para clarificar un poco más los términos, definamos qué entendemos por cada tipo de prueba. Prueba unitaria es aquélla que ejercita una pequeña porción de código y en un escenario simple, como por ejemplo, una ejecución normal de un método. Llamamos pruebas de integración a las pruebas en código que chequean una funcionalidad simple, pero que involucra varios métodos, o incluso varias clases. Finalmente, prueba de aceptación o de comportamiento es aquélla prueba que verifica el funcionamiento de un escenario completo desde el punto de vista de un usuario o cliente de la aplicación, aunque salteando la interfaz de usuario.

La idea es usar, sucesivamente de ser necesario, las pruebas unitarias, las de integración y las de comportamiento, como redes de contención en cadena, que aseguren la preservación del comportamiento en una refactorización.

A nuestro método lo llamamos **refactoring asegurado por niveles de pruebas**, y es una técnica de aseguramiento de la corrección de una refactorización basada en brindar redes de seguridad en forma escalonada, mediante pruebas cada vez más abarcativas. En pocas palabras, el procedimiento consiste en:

- Refactorizar usando las *pruebas unitarias* como red de aseguramiento de la preservación del comportamiento.
- Si algunas pruebas unitarias dejan de compilar o fallan después de la refactorización, excluir las que no pasen y usar las *pruebas de integración* (pruebas unitarias de los clientes).
- Si las pruebas de integración también dejan de compilar o fallan, apartar las que no pasen, usando las *pruebas de comportamiento* para asegurar la preservación del comportamiento.
- Si las pruebas de comportamiento fallan, significa que la refactorización no era segura y debe deshacerse (volver al código antes de la refactorización).

En cada paso, al extraer pruebas del conjunto, se debe garantizar la cobertura con las pruebas del siguiente nivel. Asimismo, cada vez que se eliminaron pruebas del conjunto, hay que volver a introducirlas luego de la refactorización, cambiarlas para que compilen, y volver a probar todo.

En definitiva, las pruebas unitarias dan un primer nivel de seguridad. Si el primer nivel falla, las pruebas de integración funcionan como pruebas de un segundo nivel. Y en última instancia, están las pruebas de comportamiento.

Lo fundamental en el método es que se detiene en las pruebas de comportamiento, que en ningún momento deben ser cambiadas, ya que son los invariantes de la refactorización, las que garantizan su corrección. Por la propia definición del refactoring, si una prueba de comportamiento cambia, estamos ante un cambio de requerimientos, no de una refactorización.

Otra cuestión central es la condición de cobertura de unas pruebas por otras de mayor nivel. Si no pudiésemos garantizar que, al excluir un nivel de pruebas, hubiera pruebas de mayor granularidad que cubriesen al menos los mismos recorridos que las del nivel que se está omitiendo, nos quedaríamos sin ninguna garantía de preservación del comportamiento. En consecuencia, el buen funcionamiento de las pruebas de mayor granularidad, si ocurriera, sería un falso positivo. Por lo tanto, para que un conjunto de pruebas pueda ser reemplazado por otro conjunto como condición de corrección del refactoring, el nuevo conjunto tiene que cubrir como mínimo el código que cubría el primer conjunto.

El detalle de los pasos a realizar en el caso de usar un lenguaje con comprobación estática de tipos, y una vez que han fallado las pruebas técnicas, se muestra en el diagrama de actividades de la Figura 1 y se describe a continuación:

- Buscar entre las pruebas de comportamiento aquellas que cubran el código que se quiere refactorizar.
- Si no hay pruebas de comportamiento que cubran en su totalidad el código a refactorizar, no vamos a poder usarlas como garantía de la preservación del comportamiento. Lo más seguro es abandonar la refactorización iniciada y buscar mayor cobertura escribiendo nuevas pruebas.
- Si hay al menos una prueba o conjunto de pruebas de comportamiento que cubra la totalidad del código a refactorizar, seguir adelante con la refactorización, cambiando el código para mejorar su calidad.
- Cambiar también el código de los clientes que haya fallado o dejado de compilar, para que pueda volver a compilar y correr.
- Compilar todo el sistema, con sus pruebas de comportamiento, dejando de lado aquellas clases de pruebas técnicas que hubieran dejado de compilar, y que vamos a corregir más adelante.
- Si las pruebas de comportamiento hubiesen dejado de compilar, no vamos a tener garantía de corrección del refactoring. En principio, diríamos que estamos ante un cambio de requerimientos, que es lo único que admitiría un cambio en las pruebas de aceptación. En este caso la refactorización habría fallado, y debemos replantearla.
- Correr las pruebas de comportamiento y verificar que siguen pasando.
- Si las pruebas fallasen, la refactorización falló. Estamos ante un cambio de comportamiento, lo cual viola la definición de refactoring.
- Si las pruebas de comportamiento corren sin fallar, podremos concluir que la refactorización es exitosa, aunque haya pruebas técnicas que sigan fallando, que son las que hicieron que abandonásemos el refactoring asegurado por pruebas de menor granularidad.
- Corregir las pruebas técnicas, unitarias y de integración, que habían fallado en los pasos anteriores. Como medio de aseguramiento de la preservación del comportamiento de las mismas, usar el sistema que se ha cambiado, ya que sabemos que funciona bien gracias a las pruebas de comportamiento que así lo garantizan. Ahora sí la refactorización estaría completa y las pruebas todas funcionando nuevamente.

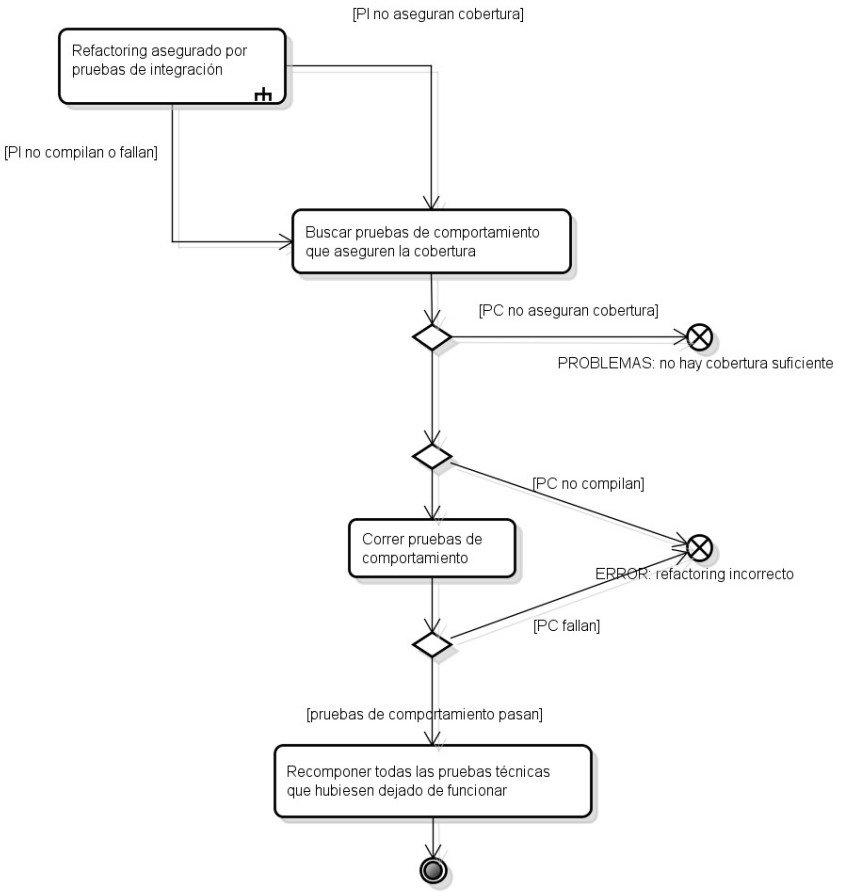

Figura 1: Procedimiento de refactoring asegurado por pruebas de aceptación

Ante este escenario, es factible preguntarse para qué realizar distintos niveles de pruebas, si con las de comportamiento solamente bastaría.

Sin embargo, este planteo no tiene en cuenta que hay refactorizaciones pequeñas que no precisan más que de las pruebas unitarias. Asimismo, hay refactorizaciones de gran envergadura que admiten ser descompuestas en refactorizaciones simples que se pueden probar con pruebas de integración. Las pruebas de comportamiento las dejamos solamente para aquellos casos en que los supuestos anteriores no se dan, como último recurso.

Esto último se debe a que, ante una falla en una prueba de comportamiento, es más difícil ver cuál fue la porción de código que la provocó. Finalmente, una cuestión no menor es que también son más lentas de ejecutar, debido a su mayor tamaño y a que suelen incluir acceso a recursos externos, lo cual hace que el proceso de refactoring sea más lento: esto está en clara contradicción con lo planteado por Beck y por Fowler [4, 8], que enfatizan que la práctica del refactoring debería ser realizada con herramientas que no entorpezcan el desempeño de las tareas del programador.

4. Automatización del método con Multilayer Coverage

Como ya hemos dicho, nuestro planteo metodológico es que, si las pruebas unitarias dejan de funcionar luego de una refactorización, y tampoco podemos utilizar las pruebas de integración, ya sea porque no aseguran la cobertura, porque no compilan más o porque fallan, no hay pruebas técnicas que nos sirvan para asegurar la preservación del comportamiento. El último recurso que nos queda son las pruebas de aceptación.

Por otro lado, como pasa tan a menudo en el desarrollo de software, la automatización es una gran ventaja del uso de las computadoras, sobre todo cuando nos referimos a tareas tediosas, repetitivas y propensas a errores. A pesar de esto, al día de hoy no existe ninguna herramienta que nos ayude a detectar en forma automática la intersección del código cubierto por distintas pruebas, o incluso cuáles son las pruebas que cubren una determinada porción de código. Por eso nos hemos planteado la construcción de *Multilayer Coverage*, una herramienta de chequeo de cobertura múltiple por pruebas técnicas y de aceptación para Java y el entorno de desarrollo Eclipse.

Marcando un método de una clase que se quiere refactorizar, es posible ver cuántas y cuáles son las clases de prueba que cubren ese método. Multilayer Coverage hace una corrida de todas las pruebas, y luego nos muestra las partes del método que están siendo cubiertas por una prueba, por dos, o por tres o más, con colores amarillo, azul y verde, respectivamente.

Además de conocer el grado de cobertura por niveles del código, Multilayer Coverage puede darnos a conocer más detalles sobre esta cobertura. De hecho, el método que proponemos requiere conocer cuáles exactamente son las pruebas que cubren el código a refactorizar. Multilayer Coverage permite conocer exactamente cuáles son las pruebas que cubren determinadas líneas. Adicionalmente, la herramienta provee información resumida sobre el grado de cobertura de un trozo de código.

4.1 Caso de estudio

Hemos realizado un caso de estudio con una aplicación de Ta-Te-Ti existente. Usando Multilayer Coverage, hemos planteado refactorizaciones que rompen las pruebas unitarias y de integración, por lo que necesitan de las pruebas de aceptación, y la herramienta nos ha ido mostrando las pruebas que cubren una misma porción de código, de manera tal de poder deshacernos de una prueba que se rompa, sabiendo que existe una de aceptación que también cubre la misma porción de código.

La Figura 2 muestra la clase *Jugador* después de haber corrido Multilayer Coverage. Allí vemos, por ejemplo, que tanto el constructor como los métodos *getSimboloJuego()* y *hacerJugada()*, al estar coloreados en verde, tienen un nivel de cobertura posiblemente suficiente para hacer la refactorización, aunque haya pruebas que se rompan. Si hubiese líneas en rojo o amarillo, eso indicaría que no tenemos cobertura suficiente. La presencia de líneas en azul indicaría cobertura por sólo dos pruebas.

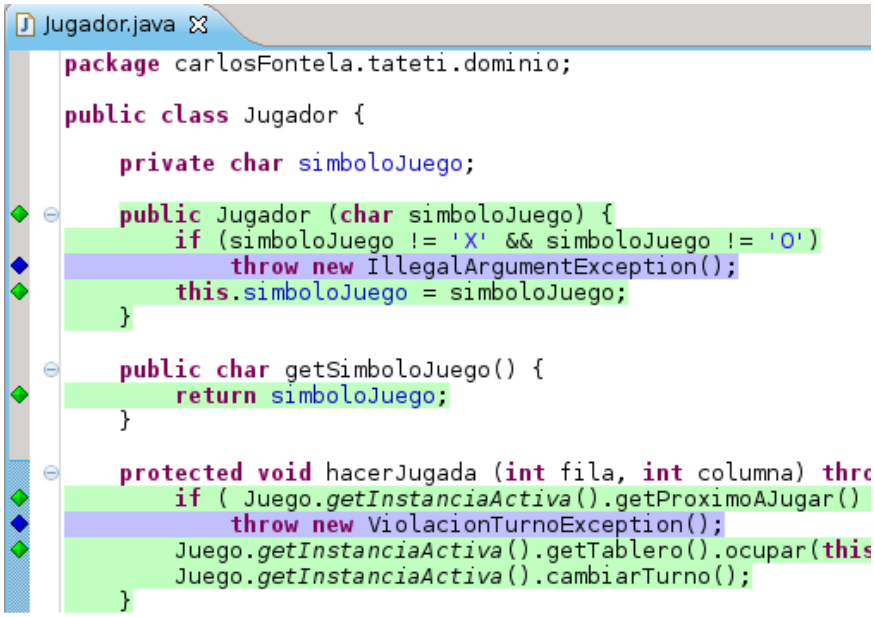

Figura 2: Grado de cobertura en Multilayer Coverage

Para comenzar entonces a refactorizar el método *hacerJugada()*, necesitamos primeramente conocer cuáles son las pruebas que puntualmente cubren las líneas de ese método. Como dijimos anteriormente, Multilayer Coverage permite conocer exactamente cuáles son las pruebas que cubren determinadas líneas. Para eso, posicionando el mouse sobre el diamante de color que Multilayer Coverage coloca a la izquierda en el entorno de desarrollo, se abre un cuadro en el que figuran las pruebas en cuestión, como se muestra en la Figura 3.

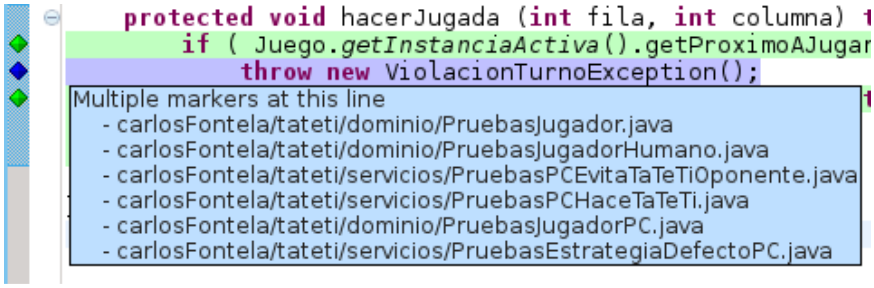

Figura 3: Clases que brindan cobertura en Multilayer Coverage

En este caso, procederemos a aplicar el refactoring sobre el método *hacerJugada()*, para que use una instancia de la clase *Juego* en vez de un Singleton siguiendo la refactorización que Cunningham [9] denomina Abstract Singleton. El problema que se plantea es que esa refactorización rompe las pruebas unitarias contenidas en la clase *PruebasJugador*, y también las pruebas de integración de las clases *PruebasJugadorHumano* y *PruebasJugadorPC*.

Sin embargo, como Multilayer Coverage nos indica que hay otras pruebas que cubren el mismo código, en las clases *PruebasEvitaTaTeTiOponente*, *PruebasPCHaceTaTeTi* y *PruebasEstrategiaDefectoPC*, y esas son pruebas de aceptación, deberíamos verificar si las mismas siguen compilando y pasando. Entonces, lo primero que hay que hacer es correr las pruebas de aceptación, las cuales pasan sin problema.

Ahora sí, entonces, podemos hacer la refactorización, cambiando el método *hacerJugada()* y comprobando con las pruebas de aceptación. Una vez que esto termine con éxito, podemos reconstruir las pruebas unitarias y de integración para dejarlas funcionando también.

Por el momento, MultiLayer Coverage evalúa solamente cobertura de sentencias [10] a nivel de *bytecode* de Java, porque de hecho esta cobertura es suficiente para aplicar nuestro método. No obstante, en un futuro planeamos extenderla para que chequee cobertura de ramas y de condiciones, entre otras. Otra mejora a la herramienta que planeamos a partir de la experiencia de uso de la misma, es que la información sobre la clase de prueba que cubre una determinada porción de código tenga un vínculo navegable hacia la clase de pruebas en cuestión.

El caso de estudio que se presenta en este trabajo no es el único realizado. También se está trabajando en aplicaciones web de mayor envergadura y en aplicaciones para dispositivos móviles en algunos trabajos de alumnos desarrollados en la Facultad de Ingeniería de la Universidad de Buenos Aires.

5. Discusión

5.1 Limitaciones metodológicas

El enfoque que proponemos en este trabajo para realizar una refactorización segura tiene algunos supuestos que lo hacen impracticable en determinadas situaciones. En efecto, podrían mencionarse las siguientes necesidades:

- Contar con pruebas técnicas automatizadas.
- Que las pruebas técnicas automatizadas ofrezcan suficiente cobertura para asegurar el comportamiento.
- Contar con pruebas de aceptación automatizadas.
- Que las pruebas de aceptación ofrezcan suficiente cobertura para asegurar el comportamiento.

La primera es la más fácil de cumplir, dado que TDD es una práctica cada vez más difundida. La segunda puede no verificarse, pero en todo caso sería cuestión de aumentar la cobertura antes de proceder a la refactorización. Si la tercera estuviera cubierta, la última no sería un problema tan serio si hubiese al menos pruebas automatizadas de aceptación, ya que estaríamos en un escenario parecido al del segundo caso. La tercera es, en cambio, la más limitante.

Lo que ocurre es que la práctica de ATDD (o sus prácticas emparentadas, BDD y STDD) es mucho más reciente que la de TDD y está mucho menos difundida. Este argumento es quizá uno de los que más mella le puede hacer a nuestra propuesta metodológica.

Como en el caso de las pruebas técnicas, las pruebas de aceptación automatizadas pueden desarrollarse a posteriori. Sin embargo, una prueba de acepta-

ción debería requerir de validaciones con los clientes, cosa difícil de obtener en un contexto de refactoring, a menos que se presente el mismo como etapa previa y necesaria para realizar una modificación al sistema. Además, escribir pruebas de aceptación automatizadas a posteriori es mucho más complejo que hacer lo mismo con las pruebas técnicas, no sólo por ser más abarcativas, sino también porque se pueden obtener menos garantías sobre su corrección.

5.2 La importancia de ATDD

El método que proponemos ofrece una razón más para desarrollar software siguiendo el protocolo de ATDD: las pruebas de aceptación que construyamos serán una última instancia de apoyo de la corrección de una futura refactorización.

Los impulsores de ATDD, como Mugridge y Cunningham [3] suelen citar varios beneficios de esta práctica, tales como la facilidad en la interpretación de las especificaciones, al expresarlas con ejemplos, y la mejora en la visibilidad de la satisfacción de requerimientos y del avance.

Como hemos visto en este trabajo, la ayuda en el análisis de la corrección de refactorizaciones es una razón más para desarrollar software siguiendo el protocolo de ATDD.

No faltan autores que coincidan con nosotros al respecto, aunque han expresado sus opiniones de forma muy escueta y sin detalles metodológicos. Entre ellos, destacan Adzic [2], Mugridge y Cunningham [3] y el sitio web de la herramienta Concordion[1]. Sin embargo, hasta donde conocemos, nadie ha presentado una práctica metodológica que incluya el problema de la cobertura. De allí la relevancia de nuestro trabajo.

6. Conclusiones

Este trabajo ha mostrado un enfoque metodológico que cubre, con pruebas de comportamiento automatizadas, las situaciones en las que una refactorización se torna riesgosa por la ruptura de pruebas unitarias. También hemos desarrollado una herramienta que ayuda en este proceso, al indicar qué pruebas de distintos niveles de granularidad cubren determinada porción de código a refactorizar. Finalmente, hemos concluido mostrando la importancia que tiene

[1] Concordion (http://www.concordion.org/) es un framework para el soporte de BDD.

la adopción de ATDD como práctica, no sólo por las ventajas que habitualmente se le reconocen, sino también por su soporte al refactoring.

Referencias

1. Pipka, J.U, “Refactoring in a ‘Test-First’ World”, Proceedings of Extreme Programming Conference, 2002, Sardinia, Italia.
2. Adzic, G., “Bridging the Communication Gap. Specification by example and agile acceptance testing”, Neuri, 2009.
3. Mugridge, R., Cunningham, W., “Fit for Developing Software: Framework for Integrated Tests”, Prentice Hall, 2005.
4. Beck, K., “Extreme Programming Explained: Embrace Change”, Addison-Wesley Professional, 1999.
5. Pipka, J.U, “Development Upside Down: Following the Test First Trail”, Daedalos Consulting GmbH, disponible en http://www.jupnet.de/publications/development_upside_down.pdf en diciembre de 2012.
6. Lippert, M., Roock, S.,“Refactoring in Large Software Projects”, John Wiley & Sons, 2006.
7. Meszaros, G., “xUnit Test Patterns: Refactoring Test Code”, Addison-Wesley Professional, 2007.
8. Fowler, M., “Refactoring”, Addison-Wesley Professional, 1999.
9. “Singleton Refactorings”, Cunningham & Cunningham, Inc., disponible en http://c2.com/cgi/wiki?SingletonRefactorings en diciembre de 2012.
10. Cornett, S., “Code Coverage Analysis”, estaba en http://www.bullseye.com/coverage.html en diciembre de 2012.