Connection Between Safe Refactorings and Acceptance Test Driven Development

C. Fontela and A. Garrido

***Abstract*— With the advent of improved strategies over Test Driven Development (TDD), like Acceptance TDD (ATDD), several benefits were recognized over the simple use of TDD with unit testing. In this article we propose an additional benefit of ATDD: the use of acceptance tests as ultimate invariants of the behavior that refactorings must preserve. Even when previous works have referred to this advantage of ATDD, the problem that remained unsolved was the lack of a complete and practical method that includes the different layers created by different types of tests. In this article we describe such a method, which uses multiple layers of tests and connects the layers through coverage analysis, in order to allow safe refactoring even when the refactorings break some tests. We also present *Multilayer Coverage*, an automatic tool for coverage analysis at different layers of tests and its intersection, to assist with the proposed method.**

***Keywords*— Test Driven Development, Acceptance Test Driven Development, Refactoring, Behavior Preservation, Coverage, Agile Methods.**

I. Introducción

HACE ya tiempo que los impulsores del desarrollo de software siguiendo metodologías ágiles vienen propugnando una serie de prácticas, entre ellas las que nos ocupan en este artículo: refactoring y TDD [1], [2]. También se viene insistiendo en la vinculación positiva entre ambas, cuando se dice que todo ciclo de TDD debe incluir un ciclo de refactoring, y cuando se afirma que el refactoring asume la preexistencia de pruebas automatizadas que permitan verificar la preservación del comportamiento a posteriori [2]. En efecto, una de las maneras de verificar la corrección del refactoring es tener una suite completa de pruebas automatizadas que, al ser ejecutadas luego de cada refactorización, nos permita afirmar que el comportamiento de la aplicación no ha variado.

Ahora bien, la granularidad de esas pruebas automatizadas puede variar mucho, desde pruebas de unidad hasta pruebas de aceptación [3]. Las pruebas de unidad tienen las ventajas de su directa vinculación con el código que se está cambiando y la facilidad de ejecución, al no requerir integración o "build" del sistema. La desventaja de estas pruebas es que muchas veces dejan de funcionar debido a la propia refactorización [4], [5]. Las pruebas de aceptación, en cambio, no se espera que dejen de funcionar luego de una refactorización, pero tienen el inconveniente de ser lentas de ejecutar (al requerir la potencial integración de varios subsistemas) y muy abarcativas, lo cual provoca que sea más difícil ver su vinculación con el código refactorizado.

Luego de la presentación de TDD y de su premisa de guiar el desarrollo a partir de pruebas en código escritas antes de la implementación de la funcionalidad, ha surgido una variante que incorpora pruebas de aceptación, y que se denomina ATDD. Un buen uso de ATDD debería incluir ciclos más abarcativos de pruebas de aceptación, y ciclos menos abarcativos de pruebas de unidad. Esto se lleva muy bien con la verificación de la corrección de las refactorizaciones, al permitir una cobertura múltiple que resulta útil para verificar la corrección, tanto en grandes refactorizaciones como en las más pequeñas.

Los autores hemos definido un enfoque metodológico, denominado *Refactoring asegurado por niveles de pruebas*, que define el procedimiento completo de verificación de la corrección de una refactorización, incluyendo distintos niveles de prueba y análisis de cobertura. Asimismo, hemos desarrollado una extensión al entorno de desarrollo Eclipse, denominada *Multilayer Coverage*, que permite analizar grados de cobertura múltiple que ayuden en este enfoque.

Lo interesante de todo esto es que el uso del *Refactoring asegurado por niveles de pruebas* nos ha permitido encontrar un nuevo argumento en favor de ATDD: la existencia de pruebas a distintos niveles que preconiza dicha práctica se convierte en un aliado a la hora de verificar la corrección de las refactorizaciones.

Este artículo se organiza de la siguiente manera: la Sección II describe los conceptos básicos sobre los que hemos trabajado: pruebas automatizadas a distintos niveles y refactoring, y presenta trabajos relacionados. La Sección III presenta la práctica metodológica que, integrando distintos niveles de pruebas en base a su cobertura, permite el refactoring de manera más segura. En la Sección IV se desarrolla lo que consideramos nuestro aporte para una utilidad adicional de ATDD y el análisis de la cobertura. La Sección V presenta una evaluación sobre la práctica metodológica desarrollada y los beneficios de ATDD, y para finalizar la Sección VI presenta conclusiones y trabajos futuros.

II. Antecedentes

A. TDD y ATDD

Test-Driven Development (TDD) es una práctica iterativa e incremental de diseño de software, presentada por Kent Beck y Ward Cunningham como parte de Extreme Programming (XP) [6]. Consiste en desarrollar siguiendo un ciclo que

C. Fontela, Facultad de Ingeniería, Universidad de Buenos Aires, Argentina, cfontela@fi.uba.ar

A. Garrido, LIFIA, Facultad de Informática, Universidad Nacional de La Plata y CONICET, Argentina, garrido@lifia.info.unlp.edu.ar

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 del código desarrollado.

Con el paso del tiempo, la práctica de TDD se ha ido convirtiendo en una técnica de diseño basada en el uso de pruebas *unitarias* automatizadas escritas antes del código productivo. Es decir, se han ido dejando de lado pruebas más abarcativas y ha quedado un uso casi exclusivo de pruebas unitarias. Esto es lo que los autores de este artículo hemos llamado Unit Test-Driven Development (UTDD), y que muchos miembros de la comunidad de desarrollo de software han criticado como insuficiente [7]. En algunos casos se utilizan pruebas de unidad de las clases clientes de la que se está refactorizando, a modo de pruebas técnicas de mayor alcance, que denominamos Pruebas de Clientes. Además, las críticas al uso de pruebas unitarias solamente, han hecho surgir nuevas formas de TDD, tales como Behavior-Driven Development (BDD, una forma de TDD que pretende expresar las especificaciones en términos de comportamiento esperado, de modo tal de lograr un grado mayor de abstracción respecto de UTDD) [7] , Story-Test Driven Development (STDD, una forma de ATDD que pone el énfasis en usar ejemplos como parte de las especificaciones, y que los mismos sirvan para probar la aplicación, haciendo que todos los roles se manejen con ejemplos idénticos) [8] y el ya mencionado ATDD [9].

A los efectos de este artículo, consideraremos a BDD y STDD como sinónimos de ATDD , que es el acrónimo con el que continuaremos trabajando. Esto es debido a que, si bien ATDD, BDD y STDD son formas de TDD que parten de premisas distintas, son equivalentes en cuanto a que obtienen el producto a partir de las pruebas de aceptación de usuarios, y en la industria no se hace distinción entre ellas.

Acceptance Test Driven Development (ATDD) [9] es una forma de TDD que obtiene el producto a partir de las pruebas de aceptación de usuarios, escritas en conjunto con ellos. La idea es tomar cada requerimiento, en la forma de una *user story*, construir varias pruebas de aceptación del usuario, y a partir de ellas construir las pruebas automáticas de aceptación, para luego, mediante ciclos de UTDD, ir desarrollando el código. De esta manera, las pruebas de aceptación de una *user story* se convierten en las condiciones de satisfacción de la misma, y los criterios de aceptación se convierten en especificaciones ejecutables.

En definitiva, ATDD se centra en empezar de lo general y las necesidades del usuario, para ir deduciendo comportamientos y generando especificaciones ejecutables. En muchos casos se trabaja con formularios en forma de tabla, que suelen ser mejor comprendidos por no informáticos, y de esa manera se consigue una mayor participación de personas no especializadas.

Lo importante de este enfoque es que puso el énfasis en que no eran pruebas de pequeñas porciones de código lo que había que desarrollar, sino especificaciones de requerimientos ejecutables. Se pone el foco en que el software se construye para dar valor al negocio, no debido a cuestiones técnicas, y esto está muy alineado con las premisas de los métodos ágiles.

Los objetivos generales de ATDD son:

- Mejorar las especificaciones.
- Facilitar el paso de especificaciones a pruebas.
- Mejorar la comunicación entre los distintos perfiles involucrados en el desarrollo: clientes, usuarios, analistas, desarrolladores y testers.
- Mejorar la visibilidad de la satisfacción de requerimientos y del avance.
- Disminuir la incorporación de funcionalidades o cualidades que el cliente no necesite ni haya solicitado (“gold-plating”).
- Usar un lenguaje único, más cerca del consumidor.

B. Refactoring

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 [1]. La refactorización se aplica transformando un sistema una vez que se desarrolló su funcionalidad y la misma ya está codificada.

Un problema central del refactoring es poder establecer su corrección, es decir, que el comportamiento observable del programa sea el mismo antes y después de cada refactorización. En la tesis seminal de Opdyke sobre refactoring se presentan “precondiciones” para que cada refactorización se considere correcta o segura [10]. Mientras que las herramientas de refactoring suelen chequear algunas de estas precondiciones, en muchos casos el análisis necesario es tan costoso que se evita (por ejemplo en el caso de uso de reflexión o manejo de memoria), dejando la decisión en manos del desarrollador. Por otro lado existen aún muchas refactorizaciones que no pueden ser automatizadas por su complejidad, y por lo tanto el análisis de precondiciones no es suficiente.

Otra solución mucho más habitual para asegurar la corrección de un refactoring es utilizar pruebas [2]. En este contexto, decimos que una refactorización es correcta si luego de la misma, las pruebas que antes funcionaban siguen funcionando sin problema.

C. Trabajos Relacionados

Varios autores plantean el uso de ATDD para asegurar la corrección de las refactorizaciones. Entre ellos, destacan Adzic [11], Mugridge y Cunningham [12] y el sitio web de la herramienta Concordion (http://www.concordion.org/). No es raro que así sea, pues las pruebas de aceptación, por su propia condición de especificaciones operacionales, deberían ser los invariantes de cualquier refactorización correcta.

Por lo tanto, la ayuda en el análisis de la corrección de refactorizaciones es una razón más para desarrollar software siguiendo el protocolo de ATDD, más allá de las ventajas conocidas de contar con distintos niveles de pruebas. Por ello se esperaría una adopción mayor en la industria de la práctica de ATDD.

Lo que sí resulta más extraño es que, hasta donde conocemos, nadie ha presentado una práctica metodológica completa y detallada que incluya el tratamiento de la cobertura. Tampoco se ha insistido demasiado en esta fortaleza de ATDD. De allí la relevancia de este artículo.

III. Uso de ATDD en el Marco del Refactoring Asegurado por Niveles de Pruebas

A. Refactoring asegurado por niveles de pruebas

Los autores de este artículo hemos desarrollado un enfoque metodológico denominado *Refactoring asegurado por niveles de pruebas*, que busca asegurar la corrección utilizando redes de seguridad en forma escalonada mediante pruebas cada vez más abarcativas [13].

El procedimiento consiste en:

• Refactorizar el código, usando las pruebas unitarias para chequear que el comportamiento se sigue manteniendo luego de la refactorización.

• Si algunas pruebas unitarias dejan de compilar o fallan después de la refactorización, se excluyen las que no pasan y se usan las pruebas de las clases clientes de la que estamos refactorizando.

• Si las pruebas de los clientes también dejan de compilar o fallan, se apartan las que no pasen. En este caso, las pruebas de aceptación son las que en última instancia deben seguir siendo exitosas, pues las mismas garantizan que hay preservación del comportamiento.

En cada paso, al excluir pruebas del conjunto, se debe asegurar la misma cobertura con las pruebas del siguiente nivel. Esta cuestión es ineludible en nuestro planteo: si no pudiésemos garantizar que, al excluir un nivel de pruebas, las pruebas del nivel inmediato superior cubren al menos los mismos recorridos que las del nivel que se está omitiendo, nos quedaríamos sin ninguna garantía de preservación del comportamiento. Y el buen funcionamiento de las pruebas de menor granularidad, si ocurriera, sería un falso positivo.

Asimismo, cada vez que se eliminaron pruebas del conjunto, hay que volver a introducirlas luego de cambiarlas para que compilen, y volver a probar todo, para no quedarnos con menos pruebas que las que teníamos antes de la refactorización. En esta instancia, la existencia de código que no ha cambiado su comportamiento, y de pruebas de menor granularidad que comprueban la preservación de ese mismo comportamiento, sirven como reaseguro del cambio que hagamos en las pruebas.

El diagrama de actividades de la Fig. 1 muestra el procedimiento.

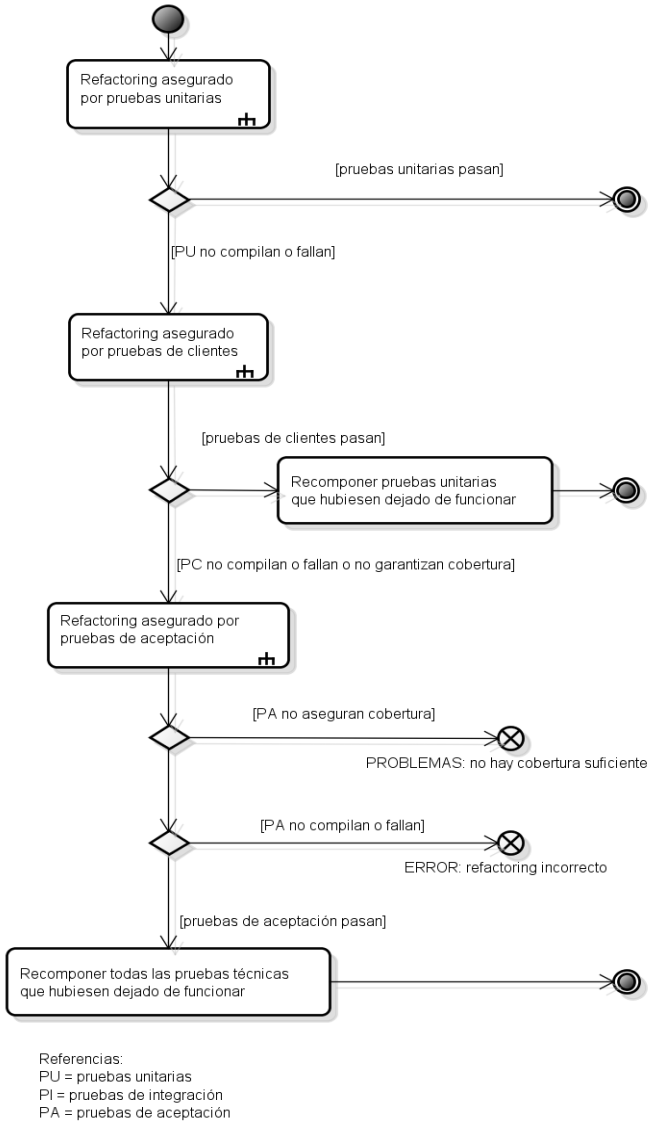

Figura 1. Refactoring asegurado por niveles de pruebas.

En definitiva, las pruebas unitarias dan un primer nivel de seguridad. Si este primer nivel falla luego de una refactorización, no significa necesariamente que la misma sea incorrecta; puede ser simplemente que las pruebas unitarias, que están tan adheridas al código, hayan sido afectadas por la refactorización. Por lo tanto, ante una falla en las pruebas unitarias, pasamos a las pruebas de integración, que funcionan como pruebas de un segundo nivel. Estas pruebas de integración, o pruebas de cliente, también pueden ser afectadas por una refactorización que transforma el protocolo de una clase, sin que signifique que la refactorización fuera incorrecta, de manera que tampoco podemos determinar ninguna conclusión. Al contrario, si contamos con pruebas de aceptación que cubren el mismo código, son estas las pruebas que usaremos en última instancia.

Así, ante refactorizaciones pequeñas alcanza con las pruebas unitarias, mientras que refactorizaciones mayores

requieren que usemos pruebas de aceptación. A pesar de ser más lentas de ejecutar, debido a su mayor tamaño y a que suelen incluir acceso a recursos externos, las pruebas de aceptación son la red de seguridad de última instancia del comportamiento que se debe preservar.

Las pruebas de aceptación son el último nivel a considerar, dado que una falla en las mismas implicaría que se ha cambiado el comportamiento especificado por los requerimientos y, por lo tanto, la refactorización es incorrecta al no garantizar la preservación del comportamiento. Esto significa que en última instancia, las pruebas de aceptación hacen las veces de invariantes para el refactoring.

B. Automatización del análisis de cobertura

El método en cuestión se hace más sencillo con el uso de *Multilayer Coverage*, una herramienta de chequeo de cobertura múltiple por pruebas técnicas y de aceptación para Java y el entorno de desarrollo Eclipse, que desarrollamos como extensión de EclEmma (herramienta de cobertura que corre sobre el motor de cobertura JaCoCo: http://www.eclemma.org/). 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* 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 del código, *Multilayer Coverage* puede darnos a conocer más detalles sobre esta cobertura. De hecho, el método que hemos desarrollado requiere conocer cuáles exactamente son las pruebas que cubren el código a refactorizar. Por eso, *Multilayer Coverage* permite conocer exactamente cuáles son las pruebas que cubren determinadas líneas. Ambas situaciones se muestran, en un ejemplo sencillo, en las Figs. 2 y 3.

Fig. 2 muestra una situación en la cual estamos refactorizando para eliminar un *Singleton* [14]. La misma nos muestra la clase *Jugador* después de haber ejecutado *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. Pueden verse más detalles en la tesis de magíster de Fontela en la Universidad Nacional de La Plata [15].

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 (ver Fig. 3).

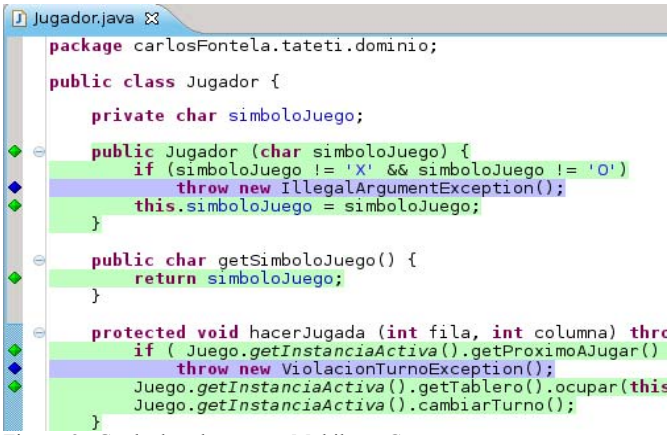

Figura 2. Grado de cobertura en Multilayer Coverage.

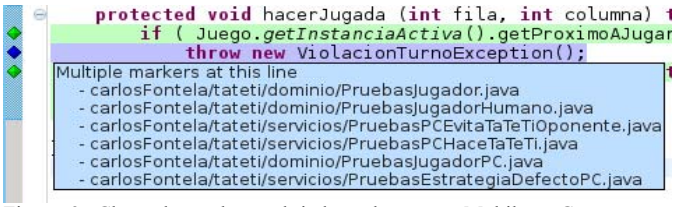

Figura 3. Clases de prueba que brindan cobertura en Multilayer Coverage.

IV. Beneficios del Método

A. La importancia de tener en cuenta la cobertura

Llamamos cobertura al grado en que los casos de pruebas de un programa llegan a cubrir dicho programa al recorrerlo. Se ha definido una gran cantidad de métricas de cobertura. Cornett habla de cobertura de sentencias, de ramas, de condiciones, entre otras menos comunes [16]. En nuestro caso, trabajamos con cobertura de sentencias, de métodos y de clases.

Nuestro enfoque metodológico explota el concepto de cobertura extendiéndola para asegurar la corrección de un refactoring. El enfoque considera la cobertura en múltiples niveles, que surge de agregar la cobertura de cada nivel de pruebas. Luego toma la intersección entre los distintos niveles de cobertura para encontrar la traza entre pruebas que permite intercambiarlas de a un nivel por vez. Este intercambio ocurre cuando el método elimina del conjunto una o más pruebas que dejan de funcionar, y las reemplaza por otras pruebas que necesariamente deben recorrer el mismo fragmento de código transformado por el refactoring. De allí que no busquemos un análisis de cobertura en el sentido tradicional. La misma razón hace que las herramientas tradicionales de cobertura tampoco nos den toda la información necesaria, y por ello nos hemos planteado la construcción de una herramienta que dé soporte al método.

B. La importancia de ATDD

Los autores de este trabajo recomendamos el uso de ATDD en forma integral en el desarrollo de software. Esto debería

incluir el uso de pruebas de aceptación y todos los ciclos de UTDD necesarios para construir cada una de las clases. Dicho de un modo simple, esto supone que a partir de los requerimientos se escriban pruebas de aceptación automatizadas, de las que deriven pruebas técnicas (de integración o unitarias), que a su vez sean fundamento del código de la aplicación.

En cuanto a los detalles, proponemos, para cada caso de uso o *user story*:

• Definir un protocolo con los mensajes que el sistema debe recibir en el contexto del caso de uso.

• Desarrollar las pruebas de aceptación, sin lógica de presentación (debido a los factores que Meszaros [17] denomina "los problemas de la prueba frágil"), para cada método de la clase de servicios.

• Escribir la clase de servicios que haga que cada prueba de aceptación pase, implementando la interfaz y usando objetos ficticios [17] para representar las clases servidoras del dominio de la aplicación.

Luego, por cada operación de la clase de servicios se deben desarrollar las clases del dominio de la aplicación siguiendo el protocolo de UTDD:

• Definir el protocolo en la clase de dominio.

• Escribir las pruebas técnicas, que serán pruebas unitarias si se basan en operaciones primitivas, o de integración si utilizan servicios de otras clases del sistema.

• Escribir el código necesario para que las pruebas pasen.

Figs. 4 y 5 muestran el procedimiento recomendado por los autores.

Esta propuesta reconoce antecedentes en los trabajos de Larman [18], de Meszaros [17] y de Cockburn [19], entre muchos otros.

En efecto, Cockburn [19] propone definir una arquitectura de al menos cuatro capas, que incluya una capa de servicios entre la capa de presentación y la de dominio. Esa capa es una fachada sin lógica de dominio, que contiene los servicios que expresan el comportamiento de la aplicación. A su vez, esos servicios delegan el comportamiento en clases de la capa de dominio. Algunos autores han recomendado usar esa capa de servicios también para agregar el manejo de transacciones y el alcance de la persistencia [20]. A nosotros nos ha servido para reunir allí los servicios contra los que se ejecutan las pruebas de aceptación.

Por otro lado, Larman [18] propone hacer diagramas de secuencia de sistema, a razón de uno por caso de uso, con los actores del caso de uso enviándoles mensajes al sistema. De ese diagrama, él deriva los contratos de las operaciones de la futura aplicación, agrupándolas por caso de uso. Por eso nosotros proponemos reunir las operaciones del sistema en una o más clases de servicios, que luego delegarán en clases de do-minio.

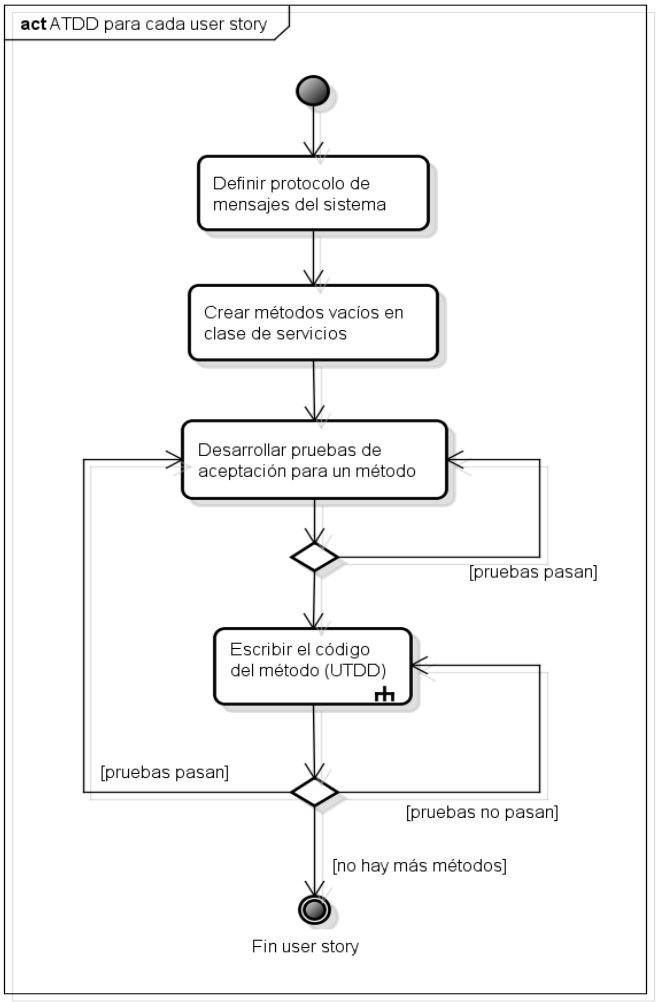

Figura 4. Ciclo de ATDD para un caso de uso o user story.

Finalmente, Meszaros [17] propuso el patrón Test Class per User Story que, como su nombre lo indica, consiste en desarrollar una clase de pruebas por cada *user story*. Nuestra recomendación es construir de esta manera las pruebas de aceptación.

Es necesario aclarar que la recomendación metodológica de desarrollo no es condición necesaria para el *Refactoring asegurado por niveles de pruebas*, sino solamente una sugerencia. Pero es una sugerencia fuerte, que termina respaldando el uso de ATDD.

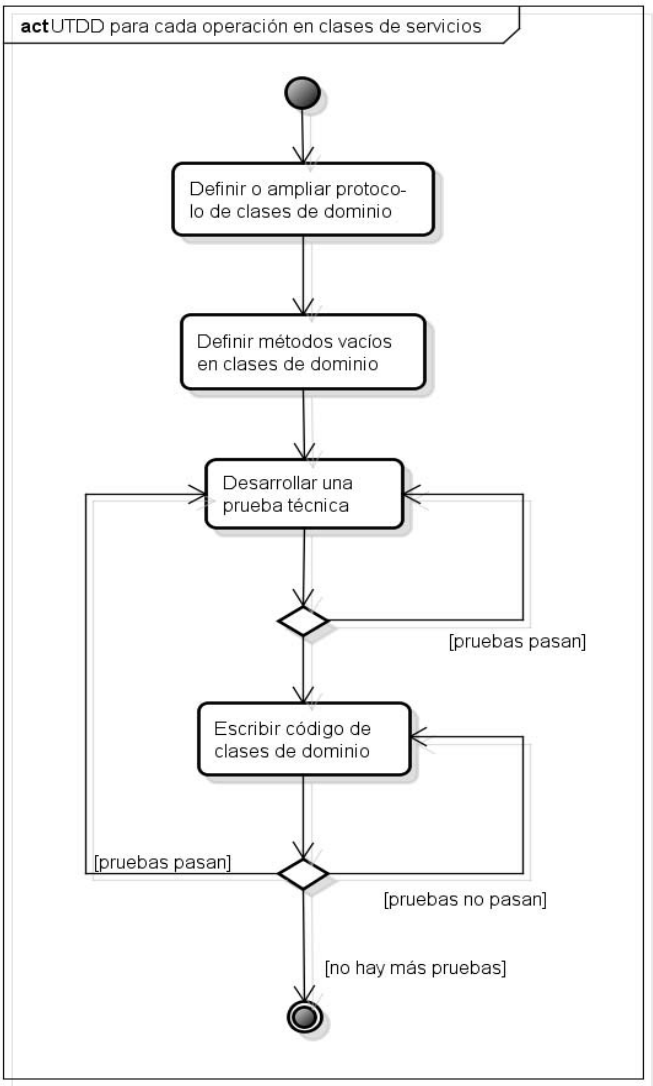

Figura 5. Ciclo de UTDD para cada operación en las clases de servicios.

V. Casos de Estudio.

El primer caso de estudio en el que se utilizó este enfoque metodológico fue una aplicación de Ta-Te-Ti con inteligencia, desarrollada íntegramente con ATDD y con una arquitectura de cuatro capas, en el marco de la tesis de Fontela [15]. Las capturas de pantalla de las Figs. 2 y 3, correspondientes a la herramienta *Multilayer Coverage*, se hicieron sobre esa aplicación. No obstante ser una aplicación sencilla, tanto la metodología seguida como la arquitectura utilizada fueron las de grandes aplicaciones industriales, por lo cual se pudo ver la conveniencia de haber seguido el protocolo de ATDD a la hora de realizar grandes refactorizaciones.

Luego se ha utilizado el enfoque metodológico en proyectos de investigación y tesinas de grado en el ámbito de la Universidad de Buenos Aires.

En uno de los proyectos se está realizando una refactorización a gran escala de un sistema bancario real. El sistema es una aplicación web, que había sido desarrollada en Java 1.4, con EJB y un servidor de aplicaciones comercial, y la refactorización incluyó cambios de arquitectura, de versión de la plataforma, que pasó a Java 1.6, el abandono del servidor de aplicaciones comercial por uno *open source*, y el abandono de los componentes EJB por el uso del framework Spring. Los requerimientos del sistema están documentados en 180 casos de uso, y su implementación antes de la refactorización contenía 595 clases que ocupaban 66.815 líneas de código.

Inicialmente, el sistema no tenía pruebas automatizadas, lo cual hacía imposible cualquier refactorización. Se empezó generando pruebas técnicas en JUnit, y en las situaciones más complejas se introdujeron pruebas de aceptación realizadas con Selenium. Se siguieron las recomendaciones de Feathers [21] sobre la introducción de pruebas, trabajando solamente con las partes a modificar. Sobre las partes que se debían modificar, la cobertura de las pruebas JUnit terminó siendo del 70% y las pruebas de Selenium del 20%, con un nivel de cobertura superpuesta del 93% (es decir, el 93% del código cubierto con pruebas de aceptación también estaba cubierto con pruebas de unidad). A modo de estudio comparativo, se midieron los tiempos de refactorización en los casos de contar con pruebas de aceptación y en los que sólo se contaba con pruebas unitarias. Usando pruebas de aceptación, las refactorizaciones han sido más sencillas de realizar cuando fallaban las pruebas unitarias, encontrándose ahorros de tiempo de hasta el 50% en el caso de refactorizaciones más grandes.

VI. Conclusiones y Trabajos Futuros

Este artículo ha mostrado que ATDD, más allá de las ventajas que ofrece hacer TDD con pruebas de aceptación de usuarios, es un enfoque metodológico que permite realizar refactorizaciones más seguras. De esta manera, ambas prácticas se realimentan de manera positiva y segura. También hemos explicado brevemente la práctica de Refactoring asegurado por niveles de prueba, desarrollada por los autores, y que brinda el marco teórico. Finalmente, hemos presentado una herramienta que ayuda en este proceso, al indicar qué pruebas de distintos niveles de granularidad cubren determinada porción de código a refactorizar.

Este enfoque de presentar a ATDD como un prerrequisito para refactorizaciones más seguras no es muy habitual, y los autores no hemos encontrado ningún planteo metodológico integral que muestre el uso conjunto de ATDD y refactoring con un análisis de cobertura.

Desde ya que el método propuesto no es tan provechoso si la aplicación se ha desarrollado sin seguir el protocolo de ATDD. En estos casos, el mejor remedio debería ser refactorizar probando con las pruebas unitarias, y desarrollar las pruebas de aceptación previo a la refactorización si aquéllas se rompen.

Es de esperar que la profundización de estudios como éste lleve a una mayor adopción de ATDD en el futuro. En efecto, la práctica de ATDD aún no se encuentra tan difundida en la industria, pero nuestro planteo de la necesidad de distintos

niveles de pruebas automatizadas para lograr refactorizaciones más seguras, es un argumento más en su favor.

Otra cuestión relacionada y en la cual es interesante profundizar en trabajos futuros es en la trazabilidad entre pruebas y su relación con ATDD y Refactoring. El artículo de Huffman Hayes y otros [22] ha planteado este tema, y entendemos que brinda otro punto de contacto entre ATDD y Refactoring.

En este sentido, estamos trabajando en otro proyecto cuyo objetivo es comparar ATDD y UTDD en casos de cambios de requerimientos funcionales sobre distintas aplicaciones comerciales, utilizando *Multilayer Coverage* para analizar los casos de cobertura múltiple. En este trabajo se está viendo que *Multilayer Coverage* sirve también para establecer trazabilidad entre pruebas de aceptación, que son especificaciones ejecutables de los requerimientos que se van a modificar, y el código que deberá ser cambiado.

Seguiremos trabajando en el futuro en la ampliación de la herramienta *Multilayer Coverage* para permitir su uso en conjunto con frameworks específicos de ATDD, tales como Fit o FitNesse.

REFERENCIAS

[1] M. Fowler. "Refactoring: Improving the Design of Existing Code", Addison-Wesley, 1999. ISBN 0201485672.

[2] K. Beck. "Test Driven Development (By Example)", Addison-Wesley, 2002. ISBN 0321146530.

[3] A. Elssamadisy, "Patterns of Agile Practice Adoption", Lulu.com, 2007. ISBN 1430314885.

[4] J.U. Pipka, "Refactoring in a 'Test-First' World", Proceedings of Extreme Programming Conference, Chicago, Estados Unidos, 2002.

[5] Martin Lippert y Stephen Roock, "Refactor-ing in Large Software Projects", John Wiley & Sons, 2006. ISBN 0470858923.

[6] K. Beck, "Extreme Programming Explained: Embrace Change", Addison-Wesley Profession-al, ISBN 0201616416.

[7] D. North, "Introducing BDD", disponible en http://dannorth.net/introducing-bdd/, consultado en julio de 2013.

[8] T. Reppert, "Test and Mesure. Don't Just Break Software. Make Software", Better Soft-ware Magazine, July-August 2004, pp. 18-23.

[9] K. Pugh. "Lean-Agile Acceptance Test Driv-en Development: Better Software Through Col-laboration: A Tale of Lean-Agile Acceptance Test Driven Development". Addison Wesley, 2011. ISBN 0321714083.

[10] W. Opdyke. "Refactoring Object-Oriented Frameworks". PhD Thesis. Univ. of Illinois at Urbana-Champaign, 1992.

[11] G. Adzic, "Bridging the Communication Gap. Specification by example and agile ac-ceptance testing", Neuri, 2009. ISBN 0955683610.

[12] R. Mugridge, W. Cunningham, "Fit for Developing Software: Framework for Integrated Tests", Prentice Hall, 2005. ISBN 0321269349.

[13] C. Fontela, A. Garrido, A. Lange, "Hacia un enfoque metodológico de cobertura múltiple para refactorizaciones más seguras", Proceed-ings of the 14th Argentine Symposium on Soft-ware Engineering (ASSE 2013), Córdoba, Ar-gentina, 2013.

[14] E. Gamma *et.al*, "Design Patterns", Addison-Wesley, 1995, ISBN 0-201-63361-2.

[15] C. Fontela, "Cobertura entre pruebas a distintos niveles para refactorizaciones más seguras", Tesis de Maestría en Ingeniería de Software, Universidad Nacional de La Plata, La Plata, Argentina, 2013.

[16] S. Cornett, "Code Coverage Analysis", consultado en julio de 2013 en http://www.bullseye.com/coverage.html .

[17] G. Meszaros, "xUnit Test Patterns: Refac-toring Test Code", Addison-Wesley Professional, 2007. ISBN 0131495054.

[18] C. Larman, "UML y patrones", Pearson Prentice-Hall, 2003. ISBN 8420534382.

[19] Vlissides, Coplien, Kerth (eds.), "Pattern Languages of Program Design 2", Addison-Wesley, 1996. Artículo "Prioritizing Forces in Software Design", de A. Cockburn. ISBN 0201895277.

[20] J. Bazzocco, "Persistencia orientada a objetos", Edulp: Editorial de la Universidad de La Plata, 2012, ISBN 978-950-34-0821-6.

[21] M. Feathers, "Working Effectively with Legacy Code", Prentice Hall, 2005.

[22] J. Huffman Hayes, A. Dekhtyar, D.S. Jan-zen, "Towards Traceable Test-Driven Develop-ment", Proceedings of the 2009 ICSE Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE '09), Vancouver, Canadá.

Carlos Fontela is an associate professor at Fac. de Ingeniería, Universidad de Buenos Aires, Argentina. Fontela has a MS degree in Software Engineering from Universidad Nacional de La Plata, Argentina. He published some books in the fields of UML, and Object Oriented Programming and Design. His current research interests include automated testing and software development processes.

Alejandra Garrido is an assistant professor at Fac. Informática, Univ. Nacional de LaPlata, Argentina, and a researcher at CONICET (Argentina's National Scientific and Technical Research Council). Her current research interests include refactoring and Web engineering. Garrido has MS and PhD degrees in Computer Science from the University of Illinois at Urbana-Champaign.