Universidad Nacional de La Plata

Facultad de Informática

Cobertura entre pruebas a distintos niveles para refactorizaciones más seguras

Autor: Moisés Carlos Fontela

Director: Dra. Alejandra Garrido

Tesis presentada para obtener el grado de

Magíster en Ingeniería de Software

Abril de 2013

A Ana, Joaquín y Clara

Índice de temas

Agradecimientos

En primer lugar, quiero agradecer a mi familia, que con su apoyo y su tiempo a lo largo de estos años, me ha permitido acrecentar mi conocimiento.
Asimismo, mucho de lo que avancé en el paradigma de objetos y la ingeniería de software se lo debo a los docentes – y alumnos – con los que me tocó trabajar, y con los cuales he discutido acaloradamente temas de diseño y metodológicos. Me gustaría mencionar especialmente a Marcio Degiovannini, Diego Fontdevila, Nicolás Páez, Pablo Suárez y Eugenio Yolis, quienes hoy son mis auxiliares docentes en la Facultad de Ingeniería de la UBA, así como también a Andrés Lange, que fue quien desarrolló la herramienta Multilayer Coverage, como parte de su Trabajo Profesional de Ingeniería Informática.
Quiero asimismo recordar a quienes fueron mis docentes en las materias de la Maestría en Ingeniería de Software de la UNLP, y a quienes me fueron orientando para elegir mi tema de tesis: Javier Bazzocco, Alejandro Oliveros, Claudia Pons, Gustavo Rossi, entre otros que también colaboraron en mi formación.
Finalmente, agradezco a Alejandra Garrido, quien me dirigió en esta tesis, y también fue parte de muchos de los grupos que menciono más arriba: como profesora en los cursos de programación y diseño orientados a objetos de la Maestría; como consejera en la elección del tema de tesis; y finalmente orientándome y discutiendo vivamente conmigo algunas opiniones que he vertido en este trabajo.

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 estamos cambiando código que sabemos que funciona por otro que – aunque presumimos que va a ser de mejor calidad – no sabemos si funcionará igual que antes.
Para permitir refactorizaciones más seguras y confiables, una buena táctica es trabajar con pruebas unitarias automatizadas, escritas antes de refactorizar, y correrlas después para asegurarnos que nuestro programa sigue funcionando tal como lo hacía previo al cambio.

Luego de que surgiera el término refactoring, en un trabajo de Opdyke y Johnson [Opdyke 1990] de 1990, la práctica se difundió escasamente hasta su aplicación en el marco de Extreme Programming (XP) en [Beck 1999]. A partir de allí se popularizó y comenzaron a surgir catálogos de problemas (“bad smells”) [Fowler 1999], de refactorizaciones clásicas [Fowler 1999], de refactorizaciones hacia patrones [Kerievsky 2005], de refactorizaciones de modelos [Boger 2003], y hasta de refactorizaciones de pruebas automatizadas [Meszaros 2007, Deursen 2001]. También los entornos de desarrollo incluyeron herramientas para favorecer ciertas refactorizaciones de catálogo.

Sin embargo, no ha habido suficiente investigación sobre el impacto de las refactorizaciones sobre las pruebas automáticas, que en ocasiones resultan afectadas por las propias refactorizaciones, causando que la funcionalidad no pueda ser verificada a posteriori. Esto resulta más evidente cuanto menor sea la granularidad de las pruebas y cuanto mayor sea el alcance de la refactorización. El trabajo de Pipka [Pipka 2002] se plantea el problema, pero sólo explica la cuestión de cambios de diseño estructurales sobre algunas refactorizaciones de catálogo, sin validarlo con una aplicación real ni contra requerimientos. Otros trabajos también han encarado la cuestión, aunque menos explícitamente y sin un enfoque metodológico.

1.1 Objetivos Generales

Esta tesis busca encontrar una práctica metodológica que permita definir distintos niveles de pruebas que operen como garantía de refactorizaciones seguras, independientemente del alcance de las mismas.
Se enmarca en el tema general de refactoring, con elementos de Test Driven Development (TDD), utilizando las prácticas recomendadas en el marco de Behavior Driven Development (BDD) y de Acceptance Test Driven Development (ATDD).

La práctica de refactoring descansa fuertemente en la existencia de pruebas unitarias automatizadas, que funcionan como red de seguridad que garantiza que el comportamiento de la aplicación no varía luego de una refactorización. Sin embargo, este simple enunciado no prevé que hay ocasiones en que las pruebas dejan de funcionar al realizar las refactorizaciones, con lo cual se pierde la sincronización entre código y pruebas, y la cualidad de red de seguridad de estas últimas. Esto es especialmente cierto ante refactorizaciones estructurales y rediseños macro.
Por lo tanto, y dado que el uso de pruebas como red de contención es uno de los supuestos más fuertes de la práctica del refactoring, vamos a desarrollar, como objetivo de esta tesis, una práctica metodológica para permitir definir distintos niveles de pruebas que aseguren distintos tipos de refactorizaciones, validándola con un caso de estudio y apoyándonos en una herramienta automática desarrollada en el marco de este trabajo.

1.2 Objetivos Específicos

Los objetivos específicos de la tesis son:

- Encontrar una práctica metodológica que permita sincronizar pruebas automatizadas en distintos niveles con refactorizaciones a distintos niveles.
- Utilizar pruebas de distinta granularidad que aumenten la redundancia en la cobertura.
- Definir una arquitectura que facilite la aplicación de la práctica metodológica del punto anterior.
- Desarrollar una herramienta que facilite la detección de la cobertura del código por pruebas a distintos niveles.
- Validar la práctica y la herramienta en un caso de estudio.
- Analizar limitaciones metodológicas del enfoque presentado.

1.3 Contribuciones de esta tesis

Las contribuciones de esta tesis son:

- **La formalización del enfoque metodológico que hemos denominado refactoring asegurado por niveles de pruebas:** se trata de un enfoque integral que cubre situaciones de insuficiencia de un nivel de pruebas por pruebas de mayor granularidad.
- **El hallazgo de nuevas ventajas de la técnica de ATDD[1]:** en efecto, la necesidad de pruebas de aceptación para garantizar la corrección de refactorizaciones es una razón más para la práctica de ATDD.
- **La construcción de la herramienta Multilayer Coverage:** se trata de una herramienta que detecta la cobertura de una porción de código por distintos niveles de prueba, especialmente diseñada para utilizar como parte del método propuesto.
- **La validación del enfoque y la herramienta en un caso de estudio:** si bien el caso de estudio presentado en esta tesis es de menor envergadura, cumple con los más exigentes requisitos de arquitectura y diseño de la práctica industrial.
- **Un análisis de la aplicación del enfoque metodológico a problemas afines:** se ha encontrado que existen puntos en común entre la solución al problema planteado en la tesis y los casos de refactoring de código legacy, refactoring de pruebas, refactorizaciones que afectan interfaces publicadas y refactoring de código compartido.
- **Un análisis de las limitaciones de las soluciones alternativas planteadas en trabajos anteriores:** se muestra la superioridad del *refactoring asegurado por niveles de pruebas*, no sólo por sobre el planteo ingenuo de utilizar sólo pruebas unitarias, sino también por encima de otros más elaborados, como los de Pipka [Pipka 2002], Lippert y Roock [Lippert 2006] y otros.

1.4 Publicaciones y trabajos vinculados a esta tesis

Durante el desarrollo de esta tesis se prepararon dos artículos. Uno de ellos es el trabajo titulado “Hacia un enfoque de cobertura múltiple para refactorizaciones más seguras”, que al momento de la presentación de esta tesis está siendo submitido a las 42 JAIIO (Jornadas Argentinas de Informática), al ASSE 2013 - 14th Argentine Symposium on Software Engineering, en la categoría de “full research papers”. El autor de esta tesis es autor de ese artículo, acompañado por Alejandra Garrido y Andrés Lange. La aceptación o no del mismo se conocerá a comienzos de julio de 2013, mientras que las jornadas se desarrollarán en el mes de septiembre en la ciudad de Córdoba.

[1] Acrónimo de “Acceptance Test Driven Development”. Ver capítulo 3.

El segundo artículo se titula “Vinculación entre refactorizaciones seguras y ATDD”, y trata el tema explicado en el capítulo 7 de esta tesis. Lo estaremos enviando a la brevedad a la revista chilena *Ingeniare*.

2 Refactoring: nociones básicas

El refactoring[2] es una técnica de mejoramiento de la calidad del código con vistas a facilitar su mantenimiento, que no altera el comportamiento observable del sistema. En este capítulo presentamos el tema, los tipos de refactoring más comunes, algunas de las herramientas más usuales y los fundamentos de la práctica. También nos vamos a detener en el aseguramiento de la corrección del refactoring, que es el fundamento del tema central de esta tesis.

2.1 Conceptos

2.1.1 Definiciones

Comencemos citando las definiciones de algunos autores:

- William Opdyke, en su tesis [Opdyke 1992] define a las **refactorizaciones** como "transformaciones que, cuando se aplican a código fuente, lo transforman, sin cambiar el comportamiento del sistema".
- También Opdyke, en conjunto con Ralph Johnson, las define como "reestructuraciones de programas que preservan el comportamiento" [Opdyke 1990] o como "transformaciones a un programa que lo reorganizan sin cambiar su comportamiento" [Johnson 1993].
- La definición de Don Roberts [Roberts 1999] se parece mucho a las de Opdyke: dice que son "transformaciones en código fuente que preservan el comportamiento".
- Lippert y Roock [Lippert 2006] hacen énfasis en su objetivo, al definir las refactorizaciones como "cambios a la estructura interna de un software, de tal manera que lo haga más comprensible y modificable sin afectar su comportamiento visible en tiempo de ejecución".
- Martin Fowler [Fowler 1999] pone el acento en el aspecto metodológico, al decir que **refactoring** es "el proceso de cambiar un sistema de software en una forma tal que el cambio no afecte el comportamiento externo del código, pero que mejore la estructura interna".

[2] En el marco de esta tesis, vamos a utilizar el término en inglés ("refactoring") solamente para el nombre de la práctica metodológica. Tanto el sustantivo que describe un cambio particular ("refactorización") como el verbo que describe la acción de realizar el cambio ("refactorizar") se utilizarán en castellano. En inglés se suelen unar el sustantivo "refactoring" y el verbo "to refactor", para indicar cada cambio y la acción de llevarlo a cabo.

- Elssamadisy [Elssamadisy 2007] hace una definición puramente utilitaria, al afirmar que "el refactoring aumenta la flexibilidad y el tiempo de vida del producto, permitiendo e incentivando a los desarrolladores a cambiar el diseño del sistema a medida que lo necesitan".

Por lo tanto, resumiendo, podríamos decir que una **refactorización** es un cambio realizado sobre el código de un sistema de software de modo tal que no se afecte su comportamiento observable, y con el sólo fin de mejorar la legibilidad, la facilidad de comprensión, la extensibilidad u otros atributos de calidad interna con vistas al mantenimiento.

En definitiva, contemplando todas las definiciones, vemos que apuntan a:

- Se trata de una práctica metodológica para la mejora de la calidad del código fuente y del diseño.
- El objetivo principal de la calidad que se busca es la facilidad de mantenimiento.
- Debe preservar el comportamiento observable.

Ahora bien: estas definiciones revelan un obstáculo, y es la necesidad de precisar qué significa que el comportamiento observable no cambie. Opdyke [Opdyke 1992] propuso que el conjunto de entradas y el conjunto de salidas debería ser el mismo antes y después de una refactorización. Pero esta definición tampoco soluciona el problema. Veámoslo con algunos ejemplos de casos límite:

- Duraciones de procesos en sistemas de tiempo real: si luego de un cambio en el código, la ejecución de un proceso cambia su duración, en principio diríamos que no hay un cambio de comportamiento; pero en sistemas de tiempo real parece evidente que las duraciones de los procesos son parte del comportamiento, dado que suelen ser derivados de los requerimientos de la aplicación.
- Uso de memoria en sistemas embebidos: es un caso similar al anterior, atendiendo al uso de memoria en vez del tiempo de ejecución.
- Contenido y navegación en sitios web.
- Reflexión[3] o metaprogramación: un programa que utilice estas técnicas no siempre puede ser cambiado en su código sin que cambie su comportamiento; diríamos que el código es parte del comportamiento.

[3] Llamamos "reflexión" (del inglés "reflection") a la capacidad que tiene un programa para examinarse a sí mismo y opcionalmente modificar su estructura. Está presente en algunos lenguajes, como Smalltalk, Java, C# y otros.

Don Roberts ya se sentía incómodo con la definición basada en la preservación del comportamiento, y propuso [Roberts 1999] una definición más formal, al decir: "una refactorización es un par *R = (pre, T)*, en donde *pre* es la precondición que el programa debe satisfacer y *T* es la transformación a realizar". Notemos que no indica cómo se deben especificar *pre* y *T*. Incluso podría ser que *pre* implicase restricciones de tiempo de ejecución o de uso de memoria.

Por lo tanto, la palabra clave en las definiciones anteriores es "observable". Lo que busca una refactorización es que preserve lo que un usuario espera del sistema, que es lo que habitualmente llamamos requerimientos. Podríamos entonces definir una refactorización considerando esta premisa:

> Una **refactorización** es un cambio realizado sobre el código de un sistema de software, con el sólo fin de mejorar la calidad interna con vistas al mantenimiento, de modo tal que, luego de esa transformación, se sigan cumpliendo los requerimientos de la aplicación.

Y podemos también definir la práctica así:

> **Refactoring** es la práctica metodológica que busca mejorar el diseño de un sistema de software con vistas al mantenimiento, preservando el comportamiento del mismo, tal como lo plantean los requerimientos de la aplicación.

2.1.2 Un primer ejemplo

Para poner en claro lo que acabamos de definir, mostremos un ejemplo. Supongamos que definimos una clase *Fecha*, que representa una fecha en el calendario, de modo tal que las instancias de dicha clase sean capaces de recibir un mensaje de validación, que tenga como resultado un valor booleano indicando si la instancia receptora representa o no una fecha válida.

Una posible implementación en Java del método correspondiente podría ser:

```
package carlosfontela.utilidades;

public class Fecha {
```

```
	private int dia;
	private int mes;
	private int anio;

	public Fecha(int dia, int mes, int anio) {
		super();
		this.dia = dia;
		this.mes = mes;
		this.anio = anio;
	}

	public boolean valida() {
		// descarte global:
		if (dia < 1 || dia > 31)
			return false;
		if (mes < 1 || mes > 12)
			return false;
		// veo si el año es bisiesto:
		boolean bisiesto;
		if ((anio % 4 == 0) && ( !(anio % 100 == 0)) )
			bisiesto = true;
		else if (anio % 400 == 0)
			bisiesto = true;
		else bisiesto = false;
		// veo los dias del mes:
		int diasMes = 31;
		if ( (mes == 4) || (mes == 6) || (mes == 9) || (mes == 11) )
			diasMes = 30;
		if (mes == 2) {
			if (bisiesto)
				diasMes = 29;
			else diasMes = 28;
		}
		// cierre:
		if (dia > diasMes)
			return false;
		else return true;
	}

	... sigue el resto de la clase ...
}
```

Sin embargo, podemos observar que este método realiza varias acciones, violando por lo tanto el principio de cohesión funcional[4] [Martin 2002].
Por lo tanto, podríamos separar las acciones en métodos. De esta manera, crearíamos los métodos *anioEsBisiesto*, que determina si el año de la fecha en cuestión es bisiesto, y *diasMes*, que devuelve la cantidad de días del mes. En ese caso, el código anterior quedaría como sigue:

```
package carlosfontela.utilidades;

public class Fecha {

	private int dia;
	private int mes;
	private int anio;

	public Fecha(int dia, int mes, int anio) {
		super();
		this.dia = dia;
		this.mes = mes;
		this.anio = anio;
	}

	public boolean valida() {
		// descarte global:
		if (dia < 1 || dia > 31)
			return false;
		if (mes < 1 || mes > 12)
			return false;
		// cierre, comparando con los días del mes:
		if ( dia > diasMes () )
			return false;
		else return true;
	}

	private int diasMes() {
		int diasMes = 31;
```

[4] Se denomina cohesión funcional a aquélla que se da cuando el módulo ejecuta una y sólo una tarea, teniendo un único objetivo a cumplir.

```
                if ( (mes == 4) || (mes == 6) || (mes == 9) || (mes == 11) )
                        diasMes = 30;
                if (mes == 2) {
                        if ( anioEsBisiesto () )
                                diasMes = 29;
                        else diasMes = 28;
                }
                return diasMes;
        }

        private boolean anioEsBisiesto() {
                if ((anio % 4 == 0) && ( !(anio % 100 == 0)) )
                        return true;
                else if (anio % 400 == 0)
                        return true;
                else return false;
        }

        ... sigue el resto de la clase ...
}
```

Lo que acabamos de realizar es una refactorización, porque:

- Se modificó el código que resolvía el problema.
- El comportamiento observable (determinar si una fecha es válida) sigue siendo el mismo.
- Se hizo con la finalidad de mejorar la calidad del diseño, cuya legibilidad aumentó y cuya complejidad disminuyó con el cambio.

Incluso se trata de una refactorización de catálogo, aplicada dos veces seguidas: la que Martin Fowler [Fowler 1999] denomina *Extract Method* y que Opdyke [Opdyke 1990] bautizó como *Replacing statement list with function call*.

Más adelante volveremos sobre este ejemplo.

2.1.3 Refactoring y paradigmas

La práctica de refactoring surgió en ambientes de orientación a objetos, y allí ha tenido y tiene su mayor difusión y uso. Opdyke [Opdyke 1990] utilizó C++ en su trabajo fundacional, Roberts [Roberts 1999] lo hizo con Smalltalk y Fowler [Fowler 1999] con Java, por nombrar sólo algunos de los trabajos más destacados.

Sin embargo, su uso se ha extendido hacia otros paradigmas, sobre todo el estructurado, el funcional y el de programación orientada a aspectos.

Alejandra Garrido [Garrido 2000, Garrido 2003 y Garrido 2005] fue de las primeras en trabajar en refactoring en lenguajes estructurados, y particularmente en C, incluso con un primer catálogo para este lenguaje. También Ralph Johnson se ha ocupado del tema. Además se ha desarrollado un *plugin* de Eclipse para Fortran, denominada Photran, y Mariano Méndez [Méndez 2011] ha realizado un primer catálogo para ese lenguaje. Lamentablemente, no hay desarrollos avanzados para Cobol, que es el lenguaje de los sistemas *legacy* por excelencia.

En el paradigma funcional ya hay algunas herramientas para Haskell y Erlang.
Y en cuanto al paradigma de aspectos, hay algunas líneas de investigación abiertas [Rura 2003, Iwamoto 2003], y hasta un catálogo preliminar [Monteiro 2005].

2.1.4 Distintos tipos de refactoring

Si bien el refactoring apareció como una práctica de mejora del código fuente, han surgido otras situaciones en las que se pueden mejorar otros artefactos de software, sin cambiar su comportamiento y con cambios en la implementación. Como era de esperar, a estas prácticas también se las englobó bajo el paraguas del refactoring.

Entre ellas destacan:

- Refactoring de bases de datos relacionales, incluyendo desde cambios de diseño para cumplir con algunas de las formas normales hasta modificaciones a gusto del programador. El tema de refactoring en bases de datos es tan rico, que ha dado lugar a varias publicaciones y a un catálogo [Ambler 2006].
- Refactoring de modelos, sobre todo aplicado a diagramas UML. Si bien resulta problemático por la dificultad de demostrar la preservación del comportamiento, es factible usando las definiciones formales de UML [UML], e interesante cuando una refactorización se ve mejor en modelos que en código. El artículo [Boger 2003] propone una herramienta de refactorización para diagramas de clases, estados y actividades.
- Refactoring de lenguajes de marcado, como HTML y XML [Harold 2008].
- Refactoring de pruebas automatizadas, tema que abordaremos en el apartado de trabajos futuros del capítulo 7.

2.1.5 Catálogos de refactoring

Una vez establecido el refactoring como una práctica deseable, comenzaron a surgir algunos catálogos que explican algunas refactorizaciones típicas y cómo realizarlas. La mayor parte de estos catálogos trabajan con lenguajes orientados a objetos, aunque también hay unos pocos catálogos para refactorizaciones en lenguajes de otros paradigmas, como el estructurado y el funcional.

Ya Opdyke [Opdyke 1992] presentó un catálogo de 23 refactorizaciones primitivas, a las que agregó tres refactorizaciones compuestas (sucesiones de refactorizaciones) a modo de ejemplo, trabajando con C++. Don Roberts [Roberts 1999] amplió las definiciones de las refactorizaciones primitivas agregándoles postcondiciones, y trabajando con un lenguaje diferente, Smalltalk.
Tal vez el catálogo más difundido en ambientes organizacionales sea el libro de Martin Fowler [Fowler 1999], que incluye 72 refactorizaciones típicas, que el mismo autor complementa en su sitio web (http://www.refactoring.com/), agregando algunas más. En este caso, utiliza Java, aunque las mismas refactorizaciones se pueden adaptar sin problemas para lenguajes orientados a objetos con chequeo estático de tipos, tales como C++ o C#.
Siguiendo la línea de Fowler, Joshua Kerievsky [Kerievsky 2005], presenta un catálogo de refactorizaciones hacia patrones de diseño en Java, complementando una idea prevista en trabajos previos, de que el uso de patrones está mejor justificado cuando surge como necesidad durante una refactorización.
Lippert y Roock [Lippert 2006] han construido un catálogo de grandes refactorizaciones, aunque con un grado de detalle menor a los que se pueden encontrar en las publicaciones del estilo de las de Opdyke y de Fowler.

2.1.6 Refactorizaciones compuestas

Existen varios autores [Opdyke 1992, Roberts 1999, Chen 2008] que han postulado la existencia de refactorizaciones simples, fácilmente catalogables, y **refactorizaciones compuestas**, que se pueden realizar siguiendo una serie de pasos ordenados, cada uno de los cuales es una refactorización simple u otra refactorización compuesta que puede descomponerse en refactorizaciones simples.
Ya dijimos que Opdyke [Opdyke 1992] cataloga 23 refactorizaciones primitivas, muy pequeñas, y luego, a modo de ejercicio, muestra tres casos de refactorizaciones más complejas, que se apoyan en las primitivas.

El trabajo [Chen 2008] define siete tipos de refactorizaciones atómicas (*Add Field*, *Delete Field*, *Change Visibility*, *Add Method*, *Delete Method*, *Change Method*, *Add Class*, *Delete Class*), y afirma que el resto de las refactorizaciones de catálogo podrían realizarse con estos componentes básicos.
Algunos autores [Beck 1999, Fowler 1999, Elssamadisy 2007] sostienen que aún las refactorizaciones de gran envergadura pueden tratarse de esta manera, por descomposición. Lamentablemente, esto es cierto en algunos casos, aunque no en todos, como analizaremos en el próximo ítem.

Un aspecto interesante de las refactorizaciones compuestas es su reversibilidad. En efecto, si una refactorización es una transformación que preserva el comportamiento, la inversa de cualquier refactorización es también una refactorización. Esto es obviamente aplicable a casos de refactorizaciones compuestas, y por lo tanto, puede servir para construir herramientas de refactoring que permitan deshacer cambios complejos. Nuevamente, esto, que desde el punto de vista teórico es inobjetable, es más difícil de llevar a la práctica en todos los casos. Pensemos, por ejemplo, en una refactorización que elimine dos clases hermanas (hijas de la misma clase madre), colapsando la jerarquía y haciendo que las anteriores instancias de las clases hijas pasen a ser instancias de la clase madre: la operación de deshacer no resulta tan sencilla si no se almacenó de alguna manera el estado inicial.

2.1.7 Refactorizaciones de gran envergadura

Existen ocasiones en las que los cambios de requerimientos o la necesidad de nuevas funcionalidades nos obliga a realizar previamente grandes refactorizaciones.
Si bien resulta difícil definir qué es "grande", podemos seguir el criterio de [Lippert 2006], que clasifica de **grandes refactorizaciones** a aquellas que cumplen las tres condiciones que siguen:

- Se trata de iniciativas de larga duración (al menos mayor a un día de trabajo).
- Involucran a todo el equipo de desarrollo.
- No pueden ser descompuestas en refactorizaciones básicas seguras[5] y de catálogo.

Algunos de los inconvenientes de estas refactorizaciones, que hacen que se las trate como casos aparte son:

[5] Entendemos por refactorizaciones seguras a aquellas que se realizan de forma tal que todos los estados intermedios de las mismas preservan el comportamiento.

- Las personas que deben trabajar en ellas tienden a perder la traza de la refactorización y se les hace difícil estimar el grado de avance.
- Como realizarlas lleva mucho tiempo, se las suele interrumpir, debiendo garantizar que el sistema quede en un estado válido y de preservación del comportamiento, aunque en estos estados intermedios, la calidad del diseño pueda ser peor que cuando se inició.
- Suele ser difícil prever todos los cambios intermedios, por lo que requieren desarrolladores experimentados y con mucha creatividad para resolver lo inesperado.

Por todas estas razones es que muchos autores desalientan la realización de grandes refactorizaciones, recomendando convertirlas en refactorizaciones compuestas de refactorizaciones simples. Sin embargo, esto no siempre es posible.
Otra manera de evitarlas es adoptar la práctica de realizar refactorizaciones pequeñas en forma permanente durante el desarrollo y luego de cada pequeño cambio, intentando prever modificaciones posteriores. Igual que antes, esto puede funcionar en la mayor parte de los casos, pero no en todos: a veces las grandes refactorizaciones surgen por grandes cambios en una aplicación, que aparecen sin aviso previo.

Un caso especial de refactorización de gran envergadura se da cuando realizamos cambios en repositorios persistentes, sobre todo bases de datos relacionales en sistemas orientados a objetos.
En efecto, cuando cambiamos una base de datos, esto suele traer cambios importantes en las clases que mapean a tablas de la base de datos. Esto ocurre porque la relación entre tablas y clases, aún en casos relativamente sencillos, no es biunívoca, como se puede ver en los casos en que los objetos se crean a partir de *joins* de tablas de la base de datos.
Puede pensarse que un buen diseño en capas debería mitigar este problema, y eso es correcto: una capa de mapeo objeto-relacional que no se pueda sortear, debería limitar los cambios solamente a esta capa. Pero aún así se mantienen problemas de compatibilidad. Incluso los cambios en las clases persistentes pueden provocar cambios necesarios en los repositorios.

Otras refactorizaciones que suelen convertirse en grandes refactorizaciones son las que afectan relaciones complejas entre clases con muchos clientes, que a su vez cambian porque cambiaron sus colaboradores. También se suele llegar a grandes refactorizaciones cuando involucramos grupos de clases, tales como paquetes, subsistemas y capas de una aplicación.

No ha habido tantos catálogos de grandes refactorizaciones como sí los ha habido de las elementales. El libro [Lippert 2006] es una notable excepción.
Lo cierto es que este tipo de refactoring tiene muchos puntos en común con la reingeniería de software, aunque por definición es refactoring, debido a que se busca preservar el comportamiento.

Yendo al aspecto metodológico, una forma de encarar las grandes refactorizaciones es haciendo iteraciones que sólo se realicen para refactorizar, lo cual hace que todo el equipo ponga foco en la refactorización en curso y se evitan varios de los problemas de mantener un proceso de refactoring en paralelo con el desarrollo. Por supuesto, puede ser difícil convencer a un cliente o a un gerente de proyecto de la conveniencia o pertinencia de estas iteraciones.
También es conveniente planificarlas como parte del proyecto, ya que no sería realista considerar que un desarrollador las hace como parte de su rutina diaria, como sí se puede hacer con las refactorizaciones más pequeñas.
Como práctica, es aconsejable comenzar por realizar refactorizaciones pequeñas en varios lugares antes de comenzar la de mayor envergadura, de modo de facilitar el trabajo en una segunda fase.

2.2 Justificación del refactoring

2.2.1 Aplicación de principios de diseño

La justificación más obvia del refactoring como práctica, es la aplicación de un principio de buen diseño allí donde un programa no parece estar respetándolo.
En la jerga de los desarrolladores, se dice que algo en el diseño "huele mal" cuando ciertas cosas en el código violan principios de diseño conocidos, de allí que a estas situaciones se las llame "olores" o "malos olores" ("smells" o "bad smells", en inglés).
Un mal olor no es necesariamente un problema de diseño que amerita una refactorización, aunque sí es un poderoso indicio.

Lo cierto es que los catálogos de refactorizaciones suelen venir acompañados de catálogos de olores, y asocian unos con otros. Fue Kent Beck[6] [Fowler 1999] el primero que sugirió basarse en olores para detectar lugares de deseables refactorizaciones, y define a los olores

[6] En realidad, es una trascripción de Beck realizada por Fowler.

como aquellas "estructuras en el código que sugieren la posibilidad de una refactorización". Los catálogos de olores se han popularizado a partir del libro de Fowler [Fowler 1999].
Así como hay olores de código, el libro [Lippert 2006] introdujo los olores de arquitectura, que requieren refactorizaciones mayores.

En el ejemplo que mostramos antes, de un método que valida una fecha, Martin Fowler diría que hay dos olores en la primera versión, que él denomina *Long Method* y *Comments*, y que hemos solucionado mediante una refactorización.

2.2.2 Entropía y su control

Todo producto de software, si es exitoso, va cambiando con el tiempo: surgen nuevos requerimientos, cambia la manera en que se provee un servicio, se mejoran algunas prestaciones, etc. De hecho, los procesos modernos de desarrollo de software plantean que el estado del software es el de evolución continua, que un sistema nunca termina de construirse, salvo cuando se deja de usar.
El primero que planteó este tema fue Meir Lehman, que esbozó una serie de leyes, conocidas como las "leyes de Lehman de la evolución del software", entre 1974 [Lehman 1974] y 1996 [Lehman 1996]. Estas leyes se aplican a sistemas que resuelven problemas en el mundo real. Nos interesan cuatro de ellas:

- Ley I: "Un programa de tipo E[7] que se usa, debe ser continuamente adaptado, o devendrá progresivamente menos satisfactorio".
- Ley II: "A medida que un programa evoluciona, su complejidad crece a menos que se trabaje para mantenerla o reducirla".
- Ley VI: "El contenido funcional de un programa debe ser continuamente incrementado para mantener la satisfacción de los usuarios a lo largo de su ciclo de vida".
- Ley VII: "Los programas de tipo E serán percibidos como de calidad decreciente a menos que sean rigurosamente mantenidos y adaptados para adecuarse a cambios de entornos operativos".

Por lo tanto, el cambio es constante en las aplicaciones de software, y cada cambio va empeorando la calidad de las mismas.

[7] Esta denominación de Lehman se refiere a programas que resuelven problemas del mundo real [Lehman 1996].

A su vez, cada vez que se produce la necesidad de un cambio, los desarrolladores se encuentran en una situación de tensión entre dos necesidades que generalmente se contraponen: realizar el cambio en el menor tiempo y con el menor costo posible; y realizar el cambio manteniendo una buena calidad del código de la aplicación. Lamentablemente, es muy probable que termine prevaleciendo la primera.
En consecuencia, aun cuando el diseño de una aplicación sea excelente en el momento de su implantación, los cambios futuros tienden a degradarlo, como indican la segunda y la séptima leyes de Lehman. Esta merma de calidad progresiva y casi inevitable del diseño del software es lo que se conoce en ingeniería de software como **entropía**[8] o decadencia del software. La entropía debilita en forma incuestionable la facilidad de mantenimiento de una aplicación, lo cual es especialmente grave debido a que el mayor costo del software se da durante su mantenimiento, no durante el desarrollo de la primera versión.
Un autor histórico de la ingeniería de software, Fred Brooks, nos advertía ya en 1975 [Brooks 1975] que: "Toda reparación tiende a destruir la estructura, a aumentar la entropía y el desorden del sistema. Cada vez es menor el esfuerzo puesto en arreglar las fallas originales del diseño; cada vez gastamos más en corregir fallas introducidas por correcciones anteriores. A medida que pasa el tiempo, el sistema va quedando peor ordenado. Más temprano que tarde, las correcciones dejan de ganar terreno."
Y el especialista en refactoring Don Roberts [Roberts 1999] dice que "cada vez que a un programador se le pide agregar alguna función que no estaba prevista en el diseño original, intentará encontrar un lugar en la estructura del código en el que pueda agregar la característica que simplemente funcione".

Debido a este aumento constante de la entropía de los diseños de software, es que se necesitan refactorizaciones frecuentes. Cuanto más frecuentes, menos va a aumentar la entropía y más sencillo será realizar nuevos cambios y, entre ellos, nuevas refactorizaciones. Si, en cambio, no tenemos el hábito de refactorizar asiduamente, nuestro producto va a ser cada vez más difícil de mantener.
En este contexto, podemos ver al refactoring como una técnica para mantener la calidad interna del software, que apunta a aumentar el tiempo de vida de los productos de software.

[8] Uno de los primeros en usar el término "entropía" fue el propio Lehman [Lehman 1996], asimilando el software a los sistemas termodinámicos, y vinculando la segunda ley con el segundo principio de la Termodinámica.

2.2.3 Refactoring surgido por requerimientos metodológicos

La práctica de refactoring está muy asociada a Extreme Programming (XP) y a otros métodos de desarrollo ágiles. En ellos, se hace especial hincapié en el carácter evolutivo del desarrollo en general, y del diseño en particular. En especial, Kent Beck [Beck 1999] ha hecho énfasis en evitar grandes diseños realizados antes de comenzar a codificar[9].
Dice John Gall [Gall 1977] que "un sistema complejo que funciona, invariablemente ha evolucionado desde un sistema simple que funcionaba".
Precisamente, la práctica de refactoring ayuda en este sentido del diseño evolutivo e incremental. Veamos.

Uno de los objetivos principales de un buen diseño es prever el cambio, y diseñar con los cambios potenciales en mente. Sin embargo, es imposible prever todo lo que se va a necesitar en el futuro y los cambios que vendrán. Por lo tanto, no podremos prever todo en el diseño inicial de una aplicación. La práctica de refactoring permanente nos ayuda a que el diseño vaya evolucionando conforme los cambios van surgiendo.
Por otro lado, aun cuando los cambios fueran previsibles, es costoso introducir desde el primer momento toda la flexibilidad necesaria para afrontarlos. Además, aun los cambios más previsibles pueden no ocurrir, y ese caso habremos introducido una flexibilidad aparentemente útil que luego no resultó necesaria, y que mientras tanto encareció el desarrollo y el mantenimiento.
También los patrones de diseño han surgido para facilitar el cambio. Por eso, un enfoque incremental de introducción de patrones en forma evolutiva sería realizar las refactorizaciones hacia patrones antes de un cambio. De esa manera, se pospone el uso de un patrón hasta que sea realmente necesario.

En definitiva, las buenas prácticas recomendarían esperar que el cambio sea necesario, en ese caso refactorizar para que el diseño se adapte mejor al cambio, y luego recién realizar el cambio en cuestión. De esta manera, el diseño surgirá en forma realmente incremental.
Otra manera de enunciarlo es que, para agregar nueva funcionalidad, primero hay que cambiar el diseño para facilitar el cambio funcional, sin cambiar el comportamiento; y recién luego cambiar el comportamiento.

[9] *Big Design Upfront* es el nombre peyorativo que le ha dado a esta práctica.

2.2.4 Refactoring de código legacy

Legacy code es una expresión inglesa muy utilizada en la Ingeniería de Software, aunque bastante difícil de definir. Habitualmente se refiere a código antiguo, aunque ésta no es la característica que más nos interesa destacar.
La mejor definición de código *legacy* [Brodieand 1995] es que se trata de un sistema que "se resiste significativamente a la evolución y a la modificación para cumplir con nuevos y constantes cambios de requerimientos".
Feathers [Feathers 2005] define código legacy a aquél que se mantiene porque funciona, sin saber por qué funciona, y su principal distinción es no tener pruebas automatizadas.

En efecto, casi todos los procesos de desarrollo asumen que se desarrolla software desde cero, cuando en realidad la mayor parte del esfuerzo de desarrollo, globalmente hablando, se emplea en el mantenimiento de aplicaciones, algunas de ellas muy antiguas.

En estos casos, el refactoring surge como una necesidad para lograr dos objetivos [Feathers 2005]:

- Hacer más sencilla la introducción de cambios en el código *legacy*, sin temor de introducir cambios en forma inesperada, que devienen errores sutiles.
- Hacer el que el código *legacy* pueda utilizarse en contextos diferentes a los que motivaron su primer desarrollo.

Probablemente, el principal problema de contar con código legacy, que no fue desarrollado siguiendo el protocolo de TDD, tiene que ver con que se trata de sistemas muy acoplados, con hay muchas dependencias a romper para poder probar.

2.2.5 Oportunidad del refactoring

Una pregunta que cabe hacerse es en qué momento, respecto del proceso de desarrollo, deben hacerse las refactorizaciones.
Tengamos en cuenta que el motivo principal del refactoring como práctica es la mejora de la calidad del código. Por lo tanto, refactorizamos:

- Antes de modificar código existente, porque el código claro es más sencillo de modificar.
- Después de modificar código existente, para mejorar la calidad de lo que se acaba de implementar, de forma tal que quede fácil de mantener en el futuro.

- Antes del *debugging*, porque el código legible es una premisa fundamental para encontrar problemas.
- Como consecuencia de una revisión por pares de diseño o de código, de la cual pudo haber surgido alguna oportunidad de mejora.
- Antes de optimizar, porque el código más legible y mejor modularizado es más fácil de optimizar.

Lo que nunca se debe hacer es refactorizar e introducir cambios al mismo tiempo, sea por corrección de errores, agregados o modificaciones en la funcionalidad u optimizaciones. Ambas tareas deben estar separadas para garantizar la calidad del resultado.

2.2.6 Algunos desafíos pendientes

El refactoring como práctica metodológica hace ya mucho tiempo que no es una novedad. Los primeros trabajos en mencionar al refactoring por su nombre [Opdyke 1990] datan de principios de la década de 1990, mucho antes de la mención al mismo en el marco de los métodos ágiles [Beck 1999].
Sin embargo, no todo está resuelto.

El principal inconveniente de la práctica de refactoring es que no agrega valor visible al negocio.
Por lo tanto, resulta un poco difícil de convencer a los gerentes de proyecto o a los clientes de incluir horas de refactoring en los proyectos. Y si bien unas pocas horas en un proyecto mediano son tolerables, lo que resulta mucho más complejo es persuadir a gerentes o clientes de la necesidad de hacer grandes refactorizaciones que posterguen el desarrollo de nuevas funcionalidades.
Por eso mismo, es una buena idea hacer las refactorizaciones como parte de los proyectos de cambios de alcance o de mantenimiento correctivo. Incluyendo sus costos en los proyectos, a la vez que van a tener un aprovechamiento inmediato, seguramente van a ser mejor aceptados.
Lo que hay que tener en cuenta es que el refactoring no se hace porque sí, sino para mejorar alguna debilidad del diseño. Por lo tanto, se espera que esta práctica agregue rentabilidad a un proyecto, en vez de restarle.
De todas maneras, faltan estudios completos de la economía del refactoring, comparando costos contra beneficios en diferentes contextos. Está claro que en aplicaciones de corta vida no tiene tanto sentido poner atención en la entropía como en aplicaciones longevas.

El otro desafío del refactoring es el aseguramiento de su resultado, que tratamos en el ítem que sigue, y que tiene derivaciones complejas como las que analizamos en el capítulo 4.

2.3 Aseguramiento del resultado del refactoring

2.3.1 El problema

Uno de los grandes desafíos del refactoring es asegurarse de que los cambios de código introducidos no provoquen cambios de comportamiento. Al fin y al cabo, al refactorizar, estamos cambiando código que funciona por otro, que supuestamente es de mejor calidad, pero no sabemos si funcionará correctamente. Siempre existe el temor de que, por mejorar cierta característica, empeoremos otra, dejando al sistema en un estado peor. De allí el aforismo que indica: "si funciona, no lo arregle".

Existen varios modos de asegurar la preservación del comportamiento luego de una refactorización, entre ellas:

- Métodos analíticos.
- Pruebas.

En los ítems subsiguientes describimos estas formas de aseguramiento de la preservación del comportamiento, que son complementarias en muchos casos.

2.3.2 Métodos analíticos

Los métodos analíticos buscan poder demostrar que las refactorizaciones son correctas, esto es, que preservan el comportamiento, y poder determinar qué refactorizaciones pequeñas son constituyentes de otras mayores, de modo tal de poder hacer refactorizaciones grandes correctas por aplicación sucesiva de refactorizaciones simples correctas.
Opdyke fue el primero en proponer un método analítico, en el trabajo fundacional [Opdyke 1990], y lo hizo basándose en las precondiciones que cada refactorización debe cumplir para que la transformación preserve el comportamiento. Por lo tanto, en los casos de refactorizaciones complejas, él plantea asegurar que en cada paso, la salida de la refactorización deja al sistema en un estado tal que cumpla con la precondición del siguiente paso.
Don Roberts [Roberts 1999] siguió en la línea de Opdyke, aunque ampliándola con el agregado de postcondiciones. En este caso, las postcondiciones sirven para derivar

precondiciones y reducir el análisis de refactorizaciones posteriores. Además, permiten calcular, mediante lógica de primer orden, las dependencias entre refactorizaciones sucesivas. Estos métodos analíticos propuestos tempranamente por Opdyke y Roberts resultan ideales para automatizar las refactorizaciones que se hagan en forma estática.

Ahora bien, todo análisis estático es conservador, ya que establece condiciones para que una refactorización sea correcta en cualquier ejecución posible del programa. En lenguajes de tipos dinámicos, que ofrecen poca información en forma estática, como Smalltalk o Python, esto puede ser muy restrictivo.

En consecuencia, Don Roberts [Roberts 1999] propuso también un método de análisis dinámico para cubrir casos que no se puedan analizar en forma estática o en los cuales sea muy costoso el análisis estático.
El problema con los análisis dinámicos es que, al ser menos conservadores, obligan a asegurar la preservación del comportamiento de alguna manera complementaria. Lo habitual es utilizar pruebas automáticas, como se explica en el ítem que sigue.

Otro enfoque interesante y más moderno es el seguido por Schäfer y de Moor [Schäfer 2010]. Ellos plantean analizar los problemas de preservación de flujo de datos y flujo de control mediante el chequeo de la preservación de dependencias semánticas. Basándose en Java 5, han creado un lenguaje más rico, mediante "extensiones livianas", que permiten hacer refactorizaciones más pequeñas en dicho lenguaje ampliado. Estas *microrefactorizaciones*, como las llaman, no las advierten quienes realizan la refactorización de mayor alcance. Tampoco las extensiones al lenguaje aparecen nunca en el código refactorizado. La implementación de una herramienta de refactoring siguiendo este enfoque resulta más sencilla, más segura e incluso de menor tamaño que sus competidoras más conocidas.

2.3.3 Pruebas de preservación del comportamiento

Las pruebas para asegurar la preservación del comportamiento surgen por varias limitaciones de los métodos anteriores. Como decíamos, los análisis dinámicos no alcanzan siempre para asegurar la corrección del refactoring. Pero también ocurre que hay refactorizaciones complejas que no se pueden automatizar, como por ejemplo algunas refactorizaciones hacia patrones de diseño de las que presenta Kerievsky [Kerievsky 2005].

Intuitivamente, lo que asegura la preservación del comportamiento es hacer pruebas que validen que esto ocurre. En definitiva, habría que ejecutar las mismas pruebas que se usaron para control de calidad, de forma tal de asegurarse que el comportamiento sigue siendo el mismo antes y después de la refactorización, corriendo las mismas pruebas que antes funcionaban sin problemas.
En el caso de validar la corrección de una refactorización con pruebas, decimos que:

Una refactorización es **correcta** si luego de la misma, las pruebas que antes funcionaban siguen funcionando sin problema.

Una cuestión no suficientemente abordada por los trabajos sobre refactoring es la que surge cuando una refactorización rompe pruebas que se usaban para probar su corrección, desde el punto de vista de la preservación del comportamiento.
Si bien Pipka [Pipka 2002] y muchos más han encarado el problema, dista mucho de estar resuelto con un enfoque metodológico completo y seguro. Este asunto está analizado en mayor detalle en el capítulo 4.

Una buena práctica en el marco del refactoring, que si bien no asegura el resultado lo favorece, es hacer pequeñas iteraciones y probar el mantenimiento del comportamiento después de cada una de ellas. Como los entornos de desarrollo hacen algunas tareas en forma automática, a veces no es necesario descomponer en pasos tan pequeños, pero hay muchas situaciones que los entornos de desarrollo no resuelven, y en estos casos la recomendación sigue siendo válida.
Por ejemplo, aun cuando parezca trivial y estuviera soportada por el entorno de desarrollo, la refactorización del método para validar fechas la hicimos de a pasos pequeños:

- Primero extrajimos el método *diasMes* usando la herramienta automática de Eclipse[10].
- En segundo término, eliminamos una variable *diasMes* que nos generó el entorno de desarrollo, reemplazándola por la llamada al método directamente, en forma manual.
- Luego refactorizamos el propio *diasMes*, extrayendo el método *anioEsBisiesto*, también usando las facilidades de Eclipse.
- A continuación, simplificamos el método *anioEsBisiesto*, a mano.
- Finalmente, eliminamos una variable *anioEsBisiesto* que nos generó el entorno de desarrollo, reemplazándola por la llamada al método directamente, también a mano.

[10] Eclipse es el entorno de desarrollo que utilizamos en este trabajo (http://www.eclipse.org/).

En el caso de grandes refactorizaciones no es siempre posible realizarlas en pasos pequeños, y por lo tanto el riesgo es mayor.

2.4 Herramientas de refactoring

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.

2.4.1 Tipos de herramientas

Hay muchas herramientas que pueden ayudar en el proceso de refactoring, entre ellas:

- Analizadores de código.
- Herramientas autónomas de refactoring automático.
- Herramientas de refactoring automático que funcionan como *plugins* de entornos de desarrollo.
- Herramientas de refactoring automático que son parte de los entornos de desarrollo.
- Herramientas de pruebas automáticas.

2.4.2 Herramientas de análisis de código

Hay gran cantidad de herramientas que analizan código.

Algunas de ellas trabajan generando reportes de posibles problemas de diseño. Entre ellas, la más evolucionada es la que es parte de Maven[11].

Otras herramientas generan diagramas de clases o paquetes del sistema, mostrando posibles problemas en forma gráfica. Entre ellas, destacan JDepend, ClassCycle, Dr. Freud, y la primera en detectar posibles problemas de arquitectura, Sotograph, desarrollada por los autores de [Lippert 2006].

2.4.3 Refactoring automatizado

Como dijimos, el análisis estático facilita la automatización de refactorizaciones.

[11] Maven es una herramienta de gestión y construcción de proyectos para Java, de licencia libre y que se desarrolla en el marco de la Apache Foundation (http://maven.apache.org/).

Si las pequeñas refactorizaciones se pueden hacer en forma automática, un desarrollador podría hacer varias en forma sucesiva sin un trabajo excesivo, y no debería ser necesario probar que cada refactorización preserva el comportamiento. En cambio, se podrían hacer una serie de pequeñas refactorizaciones que fuesen fáciles de revertir, y recién luego de una serie, comprobar la preservación del comportamiento. De esta manera, los tiempos de refactoring resultan razonables.
Incluso las refactorizaciones de gran envergadura se pueden ver favorecidas por el refactoring automático. Si bien la gran refactorización como un todo es, por definición, no automatizable, puede que muchas pequeñas partes de la misma sí lo sean, y esto se convierte en una ventaja tremenda al reducir el trabajo manual, aunque sea en esas pequeñas porciones.

Don Roberts, en el mismo trabajo [Roberts 1999] que plantea su método de análisis, propone un método y una herramienta de refactoring automatizado para Smalltalk, denominada *Refactoring Browser*.
En la actualidad hay varios entornos de desarrollo (conocidos con la sigla inglesa IDE) que realizan refactorizaciones totalmente automatizadas, lo cual las hace muy seguras porque el propio IDE garantiza que se preserva el comportamiento. El ejemplo de refactorización de la validación de fechas que realizamos más arriba se hizo ayudado por la herramienta de refactoring que viene con el IDE Eclipse.

Si bien el refactoring nació en ambientes Smalltalk, con el Smalltalk Refactoring Browser para Visual Works, hoy el lenguaje que tiene más y mejores herramientas es Java.
Entre las herramientas autónomas, las primeras fueron JRefactory, JavaRefactor, Jrbx. Como *plugins* destacan XRefactory para Emacs, RefactorIt para varios IDE y Transmogrify para JBuilder. Los entornos de desarrollo más evolucionados en cuanto al refactoring automático incorporado son Eclipse, IntelliJ IDEA, Netbeans y JBuilder.
En ambientes .NET, hay herramientas que cubren varios lenguajes y otras que sólo se enfocan en uno. Entre ellas destacan los *plugins* para Visual Studio C# Refactory, Ref++, Visual Assist X, Refactor! y JustCode!.
En otros lenguajes destacan Rope y Bicycle Repair Man, ambos para Python, y ModelMaker para Delphi.

Hay también investigadores tratando de automatizar refactorizaciones hacia patrones de diseño, como muestra el trabajo [Cinnéide 2000].

A pesar de esta larga lista, los desarrolladores distan de estar conformes con la variedad y calidad de herramientas, salvo en unos pocos lenguajes dentro del paradigma de objetos.

2.4.4 Pruebas automatizadas

Las herramientas de pruebas automáticas exceden en mucho a la problemática del refactoring. De hecho, se las suele asociar más a TDD, aunque preceden en el tiempo a este enfoque metodológico. Como es un tema central del próximo capítulo, lo analizaremos allí.

En resumen, hemos visto qué es el refactoring, sus objetivos, las propuestas que se han realizado para verificar su corrección, y algunas herramientas.
En el próximo capítulo vamos a analizar los temas de pruebas automatizadas, sus tipos y cobertura, de modo tal de poder presentar, en el siguiente capítulo, el método más habitual que se emplea para verificar el refactoring, y algunas de sus limitaciones.

3 Pruebas automatizadas

Las pruebas automatizadas son una práctica de uso habitual en el desarrollo de software desde hace varias décadas. Sin embargo, se ha popularizado en los últimos años a partir de algunas metodologías que las incluyeron como parte necesaria de las mismas, entre las que destaca Test-Driven Development (TDD). En este capítulo se presenta el tema, hacemos clasificaciones de tipos de pruebas y tipos de TDD necesarios para comprender los capítulos subsiguientes, se explica el tema de la cobertura de las pruebas y de las relaciones entre pruebas y arquitectura.
Haremos sólo un análisis introductorio, ya que, si bien el tema es fundamental para esta tesis, no resulta esencial a la misma, y hay un trabajo del autor que ya ha desarrollado en mayor profundidad estos contenidos [Fontela 2011].

3.1 Pruebas automatizadas y TDD

3.1.1 Pruebas automatizadas de programador

Se llama pruebas automatizadas de programador [Elssamadisy 2007] a las pruebas de software que un programador escribe en forma de código y que determinan la calidad de lo que él mismo (o, en algunos casos, otro programador) desarrolló. Son "automatizadas" en el sentido de que, una vez escritas, se las puede correr una y otra vez sin un costo muy alto, solamente haciendo trabajar a la computadora.
De esta manera, la sola corrida del código de pruebas debería indicarnos si lo que estamos probando funciona bien o mal. Para poder garantizar la calidad de una clase, por ejemplo, debería haber pruebas para cada responsabilidad y para cada método no trivial[12].
Además de las ventajas de facilidad de ejecución y velocidad, las pruebas automáticas tienen otro valor fundamental: la independencia del factor humano, con su carga de subjetividad y variabilidad en el tiempo. Es decir, si bien la calidad de las pruebas automáticas es consecuencia de acciones humanas, el resultado de las mismas es inapelable (pasan o no pasan, sin posibilidad de corridas "casi correctas" que nos lleven a quedarnos tranquilos) y siempre analizan de la misma forma el código que se esté probando, sin importar el horario, el día de la semana o el cansancio del programador que las esté corriendo.

[12] Un método trivial, en este contexto, sería aquél que sólo consulta o cambia el valor de un atributo (los llamados "getters y setters" en Java, "accessors" en Smalltalk, o "propiedades" en C#.

Finalmente, las pruebas automatizadas son una especificación del software escrita en forma no ambigua.

Hay dos subclases de pruebas automatizadas del programador, que Elssamadisy [Elssamadisy 2007] llama *Test-First Development* y *Test-Last Development*, refiriéndose a si se escriben antes o después del código que deben probar. Es decir, con *Test-Last Development* escribimos las pruebas después del código que queremos probar, mientras que con *Test-First Development* el código de pruebas precede al código a probar. Es que, si bien las pruebas, como es obvio, se deben correr luego del código a probar, el código de las mismas puede escribirse tanto antes como después.
A priori, *Test-Last Development* parece ser más natural, porque todo programador tiende a encarar su tarea principal en primer lugar y luego verificar su calidad. El inconveniente de esta variante es que las pruebas suelen resultar sesgadas por el código ya escrito, en cuanto a que muchas veces verifican que el programador escribió lo que él quería escribir (lo cual no es poco) y no lo que los requerimientos hubieran indicado. A pesar de estos argumentos en contra, hay varios autores que proponen usar *Test-Last Development* en determinadas circunstancias, entre ellos Dave Astels [Astels 2003].
Sin embargo, muchos autores han propuesto usar *Test-First Development*, para que las pruebas garanticen que el código responde a los requerimientos, no a lo que el programador pretende. Además, al escribir las pruebas antes, tendemos a especificar el comportamiento del código sin restringirnos a una única implementación.

En un contexto de pruebas automatizadas, cuando aparece una falla inesperada en la aplicación, el programador debería escribir la prueba que evidencie el error, reproducirlo y ver que la prueba lo detecta, corregir la falla, y asegurarse de que el resto de la aplicación sigue funcionando. De esta manera, la falla no debería volver a aparecer sin que la prueba introducida para evidenciarla la detecte.

3.1.2 Frameworks para el desarrollo de pruebas de programador

Para facilitar la escritura de pruebas de programador existen frameworks que permiten crear objetos, darles valores iniciales, ejercitar el código a probar e informar los efectos de las pruebas sobre ciertos objetos.
Los frameworks más comunes son los de la familia xUnit, llamados así porque sus nombres suelen ser SUnit, JUnit, CppUnit, NUnit, etc. Todos ellos se basan en un trabajo de Kent Beck

[Beck 1998], de 1998, que dio por resultado el primero de ellos, SUnit, un framework para pruebas automatizadas de programas Smalltalk.

Si bien el nombre xUnit puede hacer pensar en que se utilizan solamente para pruebas unitarias, estas herramientas pueden utilizarse para pruebas de distintos niveles. De hecho, este es un equívoco bastante común que ha llevado a numerosos errores de concepción de lo que debe ser una prueba de programador[13].

Cada prueba escrita con los frameworks de la familia xUnit tiene una estructura secuencial como la que sigue:

- Se establece un estado inicial de los objetos a probar.
- Se le envían ciertos mensajes a los objetos.
- Se verifica que el estado de los objetos receptores haya cambiado conforme a lo que se espera como parte de su comportamiento, y que los objetos devueltos tengan los valores esperados.
- De ser necesario, se restablece el estado inicial de los objetos.

En este trabajo, utilizaremos JUnit[14], que es el más popular de los frameworks de pruebas de programador para programas Java. El framework JUnit fue desarrollado por Kent Beck y Erich Gamma, basándose en SUnit, siendo su primer derivado, aunque su versión actual ha evolucionado bastante, alejándose del framework para Smalltalk.

Una clase de pruebas de JUnit 4 debe cumplir con las siguientes reglas:

- Cada prueba individual debe estar contenida en un método público, sin parámetros ni valor devuelto (*void*), precedido de la anotación *@org.junit.Test*.
- La verificación del estado de los objetos se hace mediante los métodos contenidos en la clase *org.junit.Assert*. Esta clase contiene métodos de clase para realizar afirmaciones, tales como *assertEquals*, *assertSame*, *assertNotEquals*, *assertTrue*, *assertFalse*, *assertNull*, etc.
- Si deseamos afirmar que el código que se está probando arroja una excepción, se puede hacer mediante un parámetro *expected* en la anotación *@org.junit.Test* del método de prueba, asignándole el tipo de excepción esperada.

[13] Para mayor detalle, ver el trabajo del autor [Fontela 2011].

[14] JUnit es un framework de pruebas automatizadas para laa plataforma Java. En este caso vamos a usar la versión 4 del mismo (http://www.junit.org/).

- Se puede definir código de inicialización para ser ejecutado antes de cada método de prueba, lo cual se hace precediéndolos de la anotación *@org.junit.Before*.
- También podemos definir código a ser ejecutado luego de cada prueba, colocándolo en un método con la anotación *@org.junit.After*.

Hay muchas otras características de JUnit que omitimos para no complicar la explicación.

Por ejemplo, el código de prueba completo de una clase *CuentaBancaria* simplificada, podría ser la clase JUnit que sigue:

```
import org.junit.*;
import static org.junit.Assert.*;

public class PruebaCuentaBancaria {

	private final int NUMERO_VALIDO = 1234;
	private final int NUMERO_INVALIDO = -1234;

	@Test
	public void debeCrearUnaCuentaValidaConDatosSuministrados( ) {
		CuentaBancaria cuenta = crearCuentaValida();
		assertEquals ("El número de cuenta se guardó mal",
					NUMERO_VALIDO, cuenta.getNumero() );
		assertEquals ("El nombre del titular se guardó mal",
					"Carlos", cuenta.getTitular() );
		assertEquals ("El saldo no comenzó siendo cero",
					0, cuenta.getSaldo() );
	}

	@Test (expected = IllegalArgumentException.class)
	public void creacionConNumeroInvalidoDebeArrojarExcepcion ( ) {
		CuentaBancaria cuenta =
			new CuentaBancaria (NUMERO_INVALIDO, "Carlos");
	}

	@Test (expected = IllegalArgumentException.class)
	public void creacionConTitularVacioDebeArrojarExcepcion ( ) {
		CuentaBancaria cuenta =
			new CuentaBancaria (NUMERO_VALIDO, "");
	}

	@Test (expected = IllegalArgumentException.class)
```

```
public void creacionConTitularNuloDebeArrojarExcepcion ( ) {
        CuentaBancaria cuenta =
                new CuentaBancaria (NUMERO_VALIDO, null);
}

@Test
public void depositoDebeIncrementarSaldo ( ) {
        int montoDepositado = 100;
        CuentaBancaria cuenta = crearCuentaValida();
        int saldoPrevio = cuenta.getSaldo();
        cuenta.depositar(montoDepositado);
        assertEquals("El depósito funcionó mal",
                saldoPrevio + montoDepositado, cuenta.getSaldo() );
}

@Test (expected = IllegalArgumentException.class)
public void depositoConMontoNegativoDebeArrojarExcepcion ( ) {
        CuentaBancaria cuenta = crearCuentaValida();
        cuenta.depositar(-100);
}

@Test
public void extraccionDebeDisminuirSaldo ( ) {
        CuentaBancaria cuenta = crearCuentaValida();
        int montoDepositado = 100;
        int montoAextraer = 50;
        cuenta.depositar(montoDepositado);
        cuenta.extraer (montoAextraer);
        assertEquals("La extracción funcionó mal",
                montoDepositado - montoAextraer, cuenta.getSaldo() );
}

@Test (expected = IllegalArgumentException.class)
public void extraccionConMontoNegativoDebeArrojarExcepcion ( ) {
        CuentaBancaria cuenta = crearCuentaValida();
        cuenta.extraer(-100);
}

@Test (expected = SaldoInsuficiente.class)
public void extraccionConSaldoInsuficienteDebeArrojarExcepcion( ) {
        CuentaBancaria cuenta = crearCuentaValida();
        cuenta.depositar(50);
        cuenta.extraer(100);
```

```
        }

        private CuentaBancaria crearCuentaValida() {
                return new CuentaBancaria (NUMERO_VALIDO, "Carlos");
        }
}
```

Allí vemos, por ejemplo, que cada prueba va precedida de la anotación *@Test*, y ninguna tiene parámetros ni valor devuelto. Un ejemplo de ello es el método *depositoDebeIncrementarSaldo*.
También vemos que, cuando esperamos que una prueba lance una excepción, usamos el parámetro *expected*, como en el caso del método *extraccionConSaldoInsuficienteDebeArrojarExcepcion*, al que lo anotamos con *@Test(expected = SaldoInsuficiente.class)*.
En cada método de prueba, las afirmaciones las hemos hecho con invocando el método *assertEquals* de la clase *Assert*, pasándole el valor esperado y el obtenido de la ejecución del código que se está probando.

3.1.3 Objetos ficticios[15]

A menudo se presenta el problema de cómo probar las interacciones con clases aún no implementadas. La idea más simple es la de construir objetos ficticios, que devuelvan valores fijos o tengan un comportamiento limitado, y sirvan sólo para probar.
Se han hecho algunos intentos de categorización de distintos tipos de objetos ficticios, que son los que Meszaros [Meszaros 2007] llama *Test Doubles* (o "dobles de prueba"), y que él clasifica así[16]:

- *Dummy Object* (literalmente, "objeto ficticio"): son aquellos que deben generarse para probar una funcionalidad, pero que no se usan en la prueba. Por ejemplo, cuando un método necesita un objeto como parámetro, pero éste no se usa en la prueba.
- *Test Stub* (literalmente, "impulsor de prueba"): son los que reemplazan a objetos reales del sistema, habitualmente para generar entradas de datos o impulsar

[15] Para mayor detalle, ver el trabajo del autor [Fontela 2011].

[16] Si bien Meszaros hace una distinción muy minuciosa entre todos los tipos de dobles, e incluso hay otros autores que han hecho esfuerzos por distinguirlos, no entraremos en estas cuestiones tan detalladas en este trabajo, teniendo en cuenta que muchas personas definen en forma ligeramente diferente cada tipo, haciendo que las distinciones entre un tipo y otro no siempre estén claras.

funcionalidades del objeto que está siendo probado. Por ejemplo, objetos que invocan mensajes sobre el objeto sometido a prueba.

- *Test Spy* (literalmente, "espía")[17]: se usan para verificar los mensajes que envía el objeto que se está probando, una vez corrida la prueba.
- *Mock Object* (literalmente, "objeto de imitación"): son objetos que reemplazan a objetos reales del sistema para observar los mensajes enviados a otros objetos desde el objeto receptor.
- *Fake Object* (literalmente, "objeto falso"): son objetos que reemplazan a otro objeto del sistema con una implementación alternativa. Por ejemplo, el reemplazo de una base de datos en disco por otra en memoria por razones de desempeño.

Notemos que lo que se busca con cada una de estas soluciones es aislar el objeto que se está probando de otros objetos, disminuyendo las dependencias, y limitar la prueba a sólo un objeto o conjunto de objetos.

Muchos de estos dobles no se usan solamente porque haya clases aún no implementadas. A veces usamos dobles porque es muy lento o muy complicado probar usando una base de datos o un sistema externo. Por lo tanto, estas técnicas sirven para probar en cualquier situación que requiera desacoplar objetos.

El término "objeto", tal como lo estamos usando, se debe ver en su acepción más amplia: a veces hay capas enteras de la aplicación que se reemplazan por una capa ficticia.

3.1.4 Tipos de pruebas

Desde hace tiempo se han clasificado distintos tipos de pruebas en la Ingeniería de Software, tales como unitarias, de integración, de sistema, funcionales, de interacción, etc. Sin embargo, ni hay un acuerdo generalizado sobre una taxonomía que defina los tipos de pruebas, ni muchas de estas categorías que recién hemos mencionado como ejemplos son siquiera excluyentes. En 2004, Dale Emery [Emery] planteaba en su blog una recopilación de tipos de prueba muy conocida que incluye diversas categorías. Ambler ha analizado el tema en [Ambler 2011]. Feathers [Feathers-2] también ha propuesto su propia categorización.

[17] Muchos autores consideran al *Spy* y al *Mock* como una misma categoría, a la que denominan genéricamente *Mocks*. Tal vez la única diferencia estriba en una cuestión de implementación: en el caso del *Mock*, se observan los mensajes enviados a medida que van ocurriendo, mientras que en el *Spy* sólo se verifica al final.

Siguiendo a Feathers, podríamos decir que las pruebas se pueden clasificar en base a distintos criterios: por alcance, por la disciplina del proceso de desarrollo en la que se aplican, por su foco, su visibilidad, por los roles de la gente que los elabora y por varias categorías más. Emery plantea 13 dimensiones. Esta multidimensionalidad hace que no se puedan establecer claramente tipos de pruebas excluyentes.

Por todo lo anterior, y para poder hablar el mismo lenguaje en este trabajo, vamos a hacer nuestra propia clasificación.

La clasificación que sigue no pretende ser única ni abarcar todos los tipos de pruebas posibles. Solamente es una categorización con vistas a ser usadas en este trabajo.

- **Pruebas técnicas** o de programador: son aquellas pruebas que verifican el buen funcionamiento de una o varias clases, métodos, paquetes u otros módulos técnicos.
- **Pruebas unitarias**: son aquellas pruebas técnicas que prueban aisladamente sólo una responsabilidad de una clase. Pueden tener llamadas a más de un método, pero solamente para poder verificar la responsabilidad en cuestión. No deben hacer uso de otras clases: si se requiriera, se debe trabajar con objetos ficticios. Siempre deben correr en memoria, sin dependencias de conexiones de red, repositorios persistentes, valores del medio (como la fecha del sistema o sistema operativo en el cual está corriendo) o cualquier otro servicio externo.
- **Pruebas de integración**: son pruebas técnicas que verifican el funcionamiento de un conjunto de objetos. Por lo tanto, suelen involucrar varias clases y sus interacciones, y generalmente se refieren solamente a la lógica de la aplicación, sin cuestiones de interfaz de usuario o acceso a datos persistentes. Por ello, se trata de que no dependan de otras capas de la aplicación, trabajando con objetos ficticios para lograrlo.
- **Pruebas de aceptación**: son aquellas pruebas que verifican un requerimiento del usuario, expresado como caso de uso, *user story*[18] o similar. Podrían depender de servicios externos, conexiones de redes o repositorios persistentes o, para hacerlas más livianas, reemplazar estos mecanismos con objetos ficticios.
- **Pruebas de interacción**: son pruebas de aceptación que se centran en verificar la interacción del usuario con la aplicación. Pueden verificar el comportamiento asociado a las interacciones o reemplazarlo por el uso de objetos ficticios.
- **Pruebas de comportamiento**: son pruebas de aceptación cuyo objetivo es verificar que el comportamiento de la aplicación es el esperado, independientemente de la

[18] El término "user story" lo hemos dejado en inglés, porque su traducción habitual, "historia de usuario" no evoca el mismo significado en castellano que en inglés, mientras que el más correcto de "relato de usuario" no se usa en la práctica.

implementación de la interfaz de usuario. Para simular la interfaz de usuario, si fuese necesario, se pueden usar objetos impulsores.

La figura 3.1 muestra un diagrama con la taxonomía de pruebas que hemos presentado, que va a ser la que utilicemos en este trabajo.

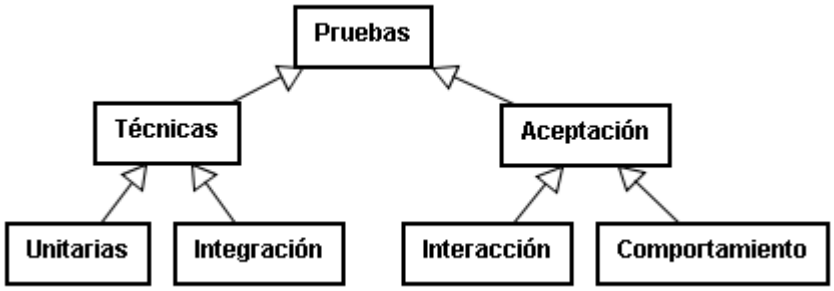


Figura 3.1: taxonomía de pruebas para esta tesis

Por supuesto, todos estos tipos de pruebas se pueden automatizar. Para ello, sería bueno que las pruebas en cuestión fuesen repetibles y consistentes, dando los mismos resultados una y otra vez. Además, conviene que se puedan ejecutar en un tiempo razonable desde el punto de vista de quien debe correrlas. Estas cuestiones son las que llevan a que a menudo se desee aislarlas usando objetos ficticios.

Todos los tipos de pruebas verifican "requerimientos", de una u otra forma. Ahora bien, las pruebas de aceptación verifican requerimientos del usuario, mientras que las de integración y unitarias verifican requerimientos (imperativos) de diseño y funcionamiento de bajo nivel, las primeras a nivel de conjuntos de clases y las segundas a nivel de responsabilidades individuales de las clases. Por eso, las pruebas de aceptación son construidas teniendo en cuenta los deseos de los clientes y usuarios, mientras que pruebas técnicas se construyen para validar diseños técnicos y su funcionamiento, que no afectan en forma directa lo que observa un usuario.
En definitiva, estamos ante el viejo problema de chequear si estamos resolviendo el problema apropiado o estamos resolviendo bien lo que entendimos que era el problema. Las pruebas de aceptación constatan lo primero, y las técnicas lo segundo.

3.1.5 Nociones de TDD[19]

Test-Driven Development[20] (TDD) es una práctica iterativa e incremental de diseño de software orientado a objetos, que fue presentada por Kent Beck y Ward Cunningham como parte de Extreme Programming[21] (XP) en [Beck 1999].
Muchas veces se ha considerado a TDD como sinónimo de *Test-First Development*, aunque no es un método de pruebas, sino de diseño, que además de las prácticas de *Test-First Development* incluye una refactorización al final de cada iteración.

El diagrama de actividades de la figura 3.2 muestra el ciclo de TDD[22].

[19] Para mayor detalle, ver el trabajo del autor [Fontela 2011].

[20] En castellano, "Desarrollo guiado por las pruebas". En adelante figurará como TDD, su sigla en inglés.

[21] En castellano, "Programación extrema". En adelante figurará como XP, su sigla en inglés.

[22] En los tres casos en que el diagrama dice "correr pruebas" se refiere a todas las pruebas generadas hasta el momento. Así, el conjunto de pruebas crece en forma incremental y sirve como pruebas de regresión ante agregados futuros.

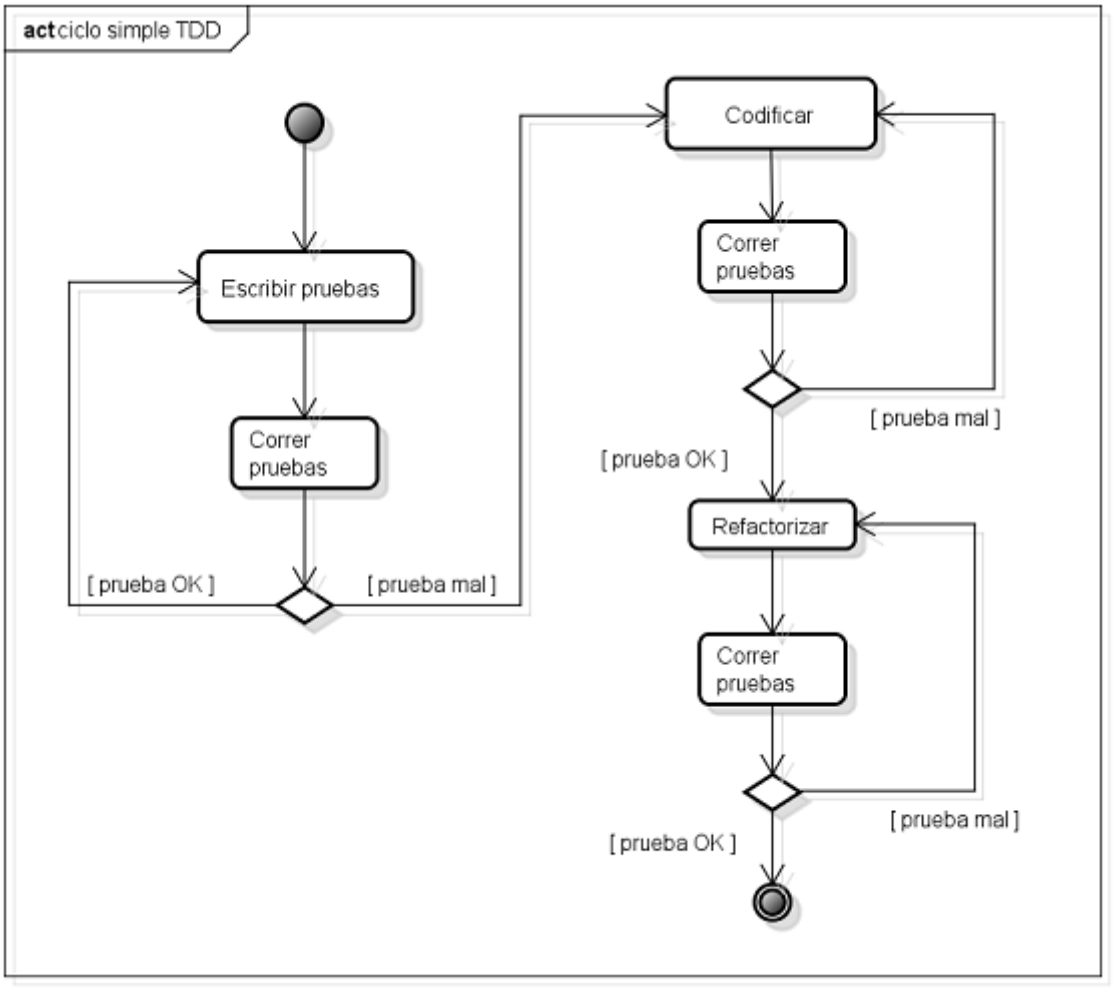


Figura 3.2: ciclo de Test-Driven Development básico

Además de las ventajas ya consideradas de las pruebas automáticas y de *Test-First Development*, la refactorización constante facilita el mantenimiento de la calidad interna a pesar de los cambios que, en caso de no hacerla, la degradarían.

Hay una regla de oro de TDD, que conviene destacar, y es la que dice: "Nunca escribas nueva funcionalidad sin una prueba que falle antes" [Beck 2002]. El corolario obvio es que ninguna funcionalidad futura debería escribirse por adelantado, si no tiene antes el conjunto de pruebas que permita verificar su corrección a posteriori. Si a eso se le suma que sólo se debería escribir de a una prueba por vez, tenemos un desarrollo incremental extremo, definido por pequeños incrementos que se corresponden con funcionalidades muy acotadas.

Notemos que la hemos definido como una práctica de diseño, no de pruebas (o al menos no principalmente), ya que al escribir las pruebas antes del propio código productivo estamos

derivando código a partir de las mismas, que a su vez nos sirven para probar el sistema. En definitiva, las pruebas explicitan los requerimientos del sistema a nivel de código. Beck [Beck 1999] dice que si una prueba resulta difícil de llevar a código, probablemente estemos ante un problema de diseño, y sostiene además que el código altamente cohesivo y escasamente acoplado es fácil de probar en forma automática.
Sin embargo, mucha gente ha destacado su doble finalidad, enfatizando que presta un servicio importante a las pruebas, lo cual no es necesariamente malo, sobre todo teniendo en cuenta que hay muchas ocasiones en que las tareas de pruebas y correcciones implican la mitad del esfuerzo del proyecto de desarrollo.

A pesar de que pareciera estar bien definida, la práctica de TDD fue variando a lo largo del tiempo, y también lo que se pretendió obtener a partir de ella. También fue creciente su independencia respecto de XP, pudiendo usarse en el marco de otra metodología de desarrollo, ágil o no.

Una crítica que se ha hecho a TDD es que pretende especificar con pruebas, y una especificación con pruebas, por definición, no es generativa: podemos hacer un programa que simplemente haga que las pruebas funcionen sin codificar nada del negocio y devolviendo los resultados que éstas esperan.

3.1.6 Tipos de TDD[23]

A medida que se fue ampliando el uso de TDD como práctica metodológica, surgieron distintas variantes, en general asociadas a distintas necesidades de diseño.

Una buena clasificación de tipos de TDD y tipos de pruebas es la que figura en la tabla que sigue[24]:

Técnica de TDD relacionada	**Tipo de prueba**	**Pregunta que responde**
STDD, ATDD, BDD	Aceptación (comportamiento)	¿Funciona el sistema de acuerdo con los requerimientos, independientemente de la interfaz de usuario?

[23] Para mayor detalle, ver el trabajo del autor [Fontela 2011].

[24] Basada en la planteada por Steve Freeman y Nat Pryce en [Freeman 2010].

Técnica de TDD relacionada	Tipo de prueba	Pregunta que responde
TDD de interacción	Aceptación (interacción)	¿Funciona la interfaz de usuario de acuerdo con los requerimientos?
RDD	Integración	¿Cada parte del sistema interactúa correctamente con las demás partes?
UTDD	Unitarias	¿Las clases hacen lo que deben hacer y es conveniente trabajar con ellas?

UTDD[25] es la forma más difundida de TDD, que se basa en pruebas unitarias. Sus mayores pretensiones son que las pruebas sirvan como especificación del uso esperado de las clases y métodos, antes del desarrollo de esas clases y métodos y sin restricciones de implementación.
Sin embargo, tiende a basar todo el desarrollo en la programación de pequeñas unidades, sin una visión de conjunto. Tampoco permite probar interfaces de usuario ni comportamiento esperado por el cliente. Finalmente, es una práctica centrada en la programación, con lo cual no sirve para especialistas del negocio ni testers. Los testers tradicionales tienden a desconfiar de ella por este mismo motivo.
Como UTDD afronta fallas de módulos, pero no las que se producen por la interacción de los módulos, se suele acompañar a UTDD con algunas pruebas de integración, utilizando objetos ficticios cuando es necesario.

Con Responsibility-Driven Development[26] (**RDD**) se encara TDD desde el punto de vista del diseño orientado a objetos, poniendo énfasis en el comportamiento de los objetos más que en el estado, y en garantizar el encapsulamiento de los objetos con ejemplos antes de pasar a la implementación. La idea es poder especificar el comportamiento de los objetos, definiéndolo en términos de cómo envía mensajes a otros objetos. Y para ello hace uso intenso de objetos ficticios, en la forma de *Mock Objects*.

[25] UTDD es una sigla poco común que significa "Unit Test-Driven Development". En castellano se traduciría como "Desarrollo guiado por las pruebas unitarias". En adelante figurará como UTDD.

[26] En realidad, sus impulsores [Freeman 2010] lo llamaron Need-Driven Development (NDD), un nombre que no refleja adecuadamente de qué se trata. Como también se lo ha llamado "diseño guiado por responsabilidades", que es un nombre más apegado a la realidad, en este trabajo lo llamaremos por su sigla en inglés, RDD. Como enfatiza el comportamiento, hay quienes lo confunden con BDD.

Sin embargo, un problema de RDD es que es sensible a las refactorizaciones de los objetos colaboradores. Efectivamente, las refactorizaciones se hacen difíciles cuando los frameworks de *Mock Objects* trabajan buscando métodos por nombre usando reflexión, ya que cuando renombramos métodos, las pruebas dejan de funcionar. Encima, no es que no sea necesario refactorizar con RDD: es tanto o más necesario que en UTDD, ya que el refactoring es lo único que puede asegurar que no habrá duplicación de código cuando lleguemos desde distintos requerimientos y mediante varias cadenas de objetos ficticios a los mismos objetos concretos.
Además, desde el punto de vista del diseño, los *Mocks* tienen el inconveniente que conocen mucho del código que están probando, al punto que contienen información de los mensajes que salen del mismo. Esto ocasiona mayor fragilidad en las pruebas y hace depender el diseño externo de la implementación interna, hasta el punto de que en ocasiones escribimos el propio código probado en la prueba.

Behaviour Driven Development[27] (**BDD**) es una forma de TDD que pretende expresar las especificaciones en términos de comportamiento, de modo tal de lograr un grado mayor de abstracción respecto de UTDD, y probando la interacción de objetos en escenarios. BDD pone un énfasis especial en cuidar los nombres que se utilizan, tanto en las especificaciones como en el código. De esta manera, hablando el lenguaje del cliente, mantenemos alta la abstracción, sin caer en detalles de implementación.

Acceptance Test Driven Development[28] (**ATDD**) 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 escribir el código. De esta manera, las pruebas de aceptación de una *user story* se convierten en las condiciones de satisfacción del mismo, y los criterios de aceptación se convierten en especificaciones ejecutables.
Al igual que BDD, a la cual se parece mucho, se trata de empezar de lo general y las necesidades del usuario, para ir deduciendo comportamientos y generando especificaciones ejecutables. La diferencia estaría en que, mientras que en BDD solemos escribir los *user stories* en forma tradicional, en las herramientas de ATDD se enfatiza que las pruebas de

[27] En castellano, "Desarrollo guiado por el comportamiento". En adelante figurará como BDD, su sigla en inglés.
[28] En castellano, "Desarrollo guiado por las pruebas de aceptación". En adelante figurará como ATDD, su sigla en inglés.

aceptación también las escriban los clientes o especialistas de negocio. 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 es que, tanto BDD como ATDD pusieron el énfasis en que no eran pruebas de pequeñas porciones de código lo que se desarrollaba, sino especificaciones de requerimientos ejecutables. Y así como UTDD pretende ser una técnica de diseño detallado, BDD se presenta como una de diseño basado en dominio, y ATDD una de requerimientos. Ambas técnicas (BDD y ATDD) ponen el foco en que el software se construye para dar valor al negocio, y no debido a cuestiones técnicas, y en esto están muy alineadas con los métodos ágiles.
En definitiva, UTDD y RDD facilitan el buen diseño de clases y sus interacciones, mientras que ATDD y BDD facilitan construir el sistema correcto.
Las mayores coincidencias entre BDD y ATDD están en sus objetivos generales:

- Mejorar las especificaciones
- Facilitar el paso de especificaciones a pruebas
- Mejorar la comunicación entre los distintos perfiles: clientes, usuarios, analistas, desarrolladores y testers
- Mejorar la visibilidad de la satisfacción de requerimientos y del avance
- Disminuir el *gold-plating*[29]
- Usar un lenguaje único, más cerca del consumidor
- Focalizar en la comunicación, no en las pruebas

Story-Test Driven Development[30] (**STDD**) es una forma de TDD 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.
Al fin y al cabo, habitualmente un analista de negocio escribe especificaciones, que le sirven a los desarrolladores y a los testers; los desarrolladores construyen su código y definen pruebas unitarias, para las cuales deben basarse en ejemplos de entradas y salidas; los testers, por su lado, desarrollan casos de prueba, que contienen a su vez ejemplos; en algunas ocasiones, para

[29] Se denomina "gold-plating", en la jerga de administración de proyectos, a incorporar funcionalidades o cualidades a un producto, aunque el cliente no lo necesite ni lo haya solicitado.

[30] STDD es el nombre que le dieron sus impulsores, Kerievsky y Marick. En castellano, "Desarrollo guiado por las historias de prueba". En adelante figurará como STDD, su sigla en inglés. Otros nombres que se han usado son "especificación con ejemplos", "pruebas de cara al negocio", "requerimientos de caja negra", "requerimientos guiados por pruebas" y el más abarcativo de "pruebas de aceptación ágiles".

fijar las especificaciones, los desarrolladores y los testers les solicitan a los analistas que les den ejemplos concretos para aclarar ideas; e incluso hay ocasiones en que los propios analistas proveen escenarios, que no son otra cosa que requerimientos instanciados con ejemplos concretos. STDD pretende usar los mismos ejemplos en todos los casos, ya sea porque se especifica directamente con ejemplos, o porque se acompañan los requerimientos con ejemplos.
Entre sus ventajas, destacan que las pruebas sirven como herramienta de comunicación, evitando "teléfonos descompuestos" y que los requerimientos son más sencillos de acordar con los clientes, al basarse en ejemplos que sirven como pruebas de aceptación.
Siguiendo la premisa principal de TDD, y aplicándosela a STDD, no deberían desarrollarse funcionalidades que no estén acompañadas por una prueba de aceptación. Y si durante una prueba manual, o en producción, surgiera un problema, habría que analizar qué pasó con la prueba de aceptación correspondiente antes de resolverlo.

TDD de interacción (el nombre es mío) es una forma de TDD bastante poco habitual, que basa el desarrollo en pruebas automatizadas de interacción ejecutadas sobre la interfaz de usuario.
Su uso no está muy difundido, y hay muchos autores que no lo recomiendan [Mugridge 2005, North, Adzic 2009, Freeman 2010] por diversas razones. Sin embargo, no hay que perder de vista que el valor para el negocio también se logra a través de la interfaz de usuario, y por eso tiene sentido probar la interacción.
Como además las pruebas de interacción de manera automática existen desde hace décadas, se han hecho intentos de hacer TDD a partir de maquetas de interfaz de usuario y modelos, tales como en los trabajos realizados en el LIFIA[31] [Burella 2010, Robles 2010, Robles 2010-2] y los realizados por Meszaros y otros [Meszaros 2003]. También se han hecho intentos de usar BDD para desarrollar interfaces web, como ocurre con la utilización del patrón denominado *Page Object* [Stewart] en ambientes de la herramienta Selenium, que parte de la idea de que la interfaz de usuario también tiene comportamiento, a veces complejo, y también es susceptible de ser modelada con objetos.

Por todo lo visto, hay muchas clases de TDD. UTDD está típicamente basado en clases, en el chequeo del estado de los objetos luego de un mensaje y se basa en un esquema *bottom-up*. BDD, ATDD y STDD se basan en diseño integral, enfocándose en el chequeo de

[31] Laboratorio de Investigación y Formación en Informática Avanzada, de la Universidad Nacional de La Plata.

comportamiento y en el desarrollo *top-down*. RDD se encuentra a mitad de camino entre ambos enfoques.
Lo que ocurre es que hay dos maneras de ver la calidad: una interna, la que les sirve a los desarrolladores, y otra externa, la que perciben usuarios y clientes. UTDD y RDD apuntan a la calidad interna, mientras que ATDD, BDD y STDD apuntan a la externa.
En definitiva, con objetivos distintos, existe un gradiente de técnicas que van desde un extremo al otro.

3.1.7 Relación entre refactoring y TDD

Las prácticas de refactoring y TDD aparecen relacionadas en dos sentidos:

- El ciclo de TDD incluye una refactorización luego de hacer que las pruebas corran exitosamente, para lograr un mejor diseño que el probablemente ingenuo que pudo haber surgido de una implementación cuyo único fin fuera que las pruebas corrieran sin problemas.
- Una refactorización segura exige la existencia de pruebas automáticas escritas con anterioridad, que permitan verificar que el comportamiento externo del código refactorizado sigue siendo el mismo que antes de comenzar el proceso.

La relación entre ambas prácticas es, por lo dicho, de realimentación positiva. Así como el refactoring precisa de TDD como red de contención, TDD se apoya en el refactoring para eludir diseños triviales.
Así es como a veces ambas prácticas se citan en conjunto, y se confunden sus ventajas. Por ejemplo, cuando se menciona que TDD ayuda a reducir la complejidad del diseño conforme pasa el tiempo, en realidad debería adjudicarse este mérito a las refactorizaciones que acompañan a los ciclos de TDD.
Sin embargo, dejemos establecido que el refactoring no necesita forzosamente del cumplimiento estricto del protocolo de TDD: sólo precisa que haya pruebas (en lo posible automatizadas), y que éstas hayan sido escritas antes de refactorizar. Por lo tanto, si en el desarrollo de una aplicación no se ha empleado TDD, al menos hay que hacer emplear *Test-Last Development*, como lo propone Feathers [Feathers 2005].

En el marco de esta tesis, y de aquí en más, vamos a considerar sinónimos a BDD, ATDD y STDD, refiriéndonos siempre al acrónimo más habitual, ATDD.

3.2 Cobertura de las pruebas

3.2.1 Definición

Llamamos **cobertura** al grado en que los casos de pruebas de un programa llegan a cubrir dicho programa al recorrerlo. Se la suele denominar tanto "cobertura de pruebas" como "cobertura de código"[32].

Se usan como una medida, aunque no la única, de la calidad de las pruebas: a mayor cobertura, las pruebas del programa son más exhaustivas, y por lo tanto existen menos situaciones de posibles errores que no están siendo probadas. No obstante, no hay que confundir con la calidad del programa: la cobertura mide sólo la calidad de las pruebas, indirectamente, y sólo muy subsidiariamente afecta la calidad del programa.

Por lo tanto, el nivel de cobertura no debería guiar el desarrollo. Sólo tiene sentido usarlo para ver, a posteriori, cuán cubiertos estamos ante regresiones. Porque si bien decimos que es una medida de la calidad de las pruebas, ni siquiera la mide en cuanto a su adecuación para guiar el diseño en un escenario de TDD.

Otra buena idea es usar las métricas de cobertura para observar tendencias. Por ejemplo, si una determinada métrica de cobertura disminuye con el tiempo, se puede deber a que la aplicación creció sin que se escribieran las pruebas correspondientes, que algunas pruebas fueron eliminadas, o un poco de cada cosa.

3.2.2 Análisis de cobertura

Según [Cornett], el análisis de cobertura, es el proceso de:

- Hallar áreas de un programa que no están siendo recorridas por un conjunto de casos de prueba.
- Crear casos de prueba adicionales para acrecentar la cobertura.
- Determinar una medida cuantitativa de cobertura de código, que sea una medida indirecta de la calidad de las pruebas.

Adicionalmente, se usa para identificar casos de prueba redundantes, en el sentido de que no aumentan la cobertura por su presencia, ya que existen otros que cubren lo mismo.

Un análisis de cobertura no es un análisis contra requerimientos, como si se tratase de una prueba funcional. Más bien es un análisis estructural, que analiza cómo están hechas las pruebas y su relación con el código que deben probar.

[32] En inglés, "test coverage" o "code coverage".

Además, hay que tener cuidado con los análisis de cobertura. Que una determinada línea de un programa esté siendo cubierta por un conjunto de pruebas no implica que se estén analizando todos los casos posibles. Esto nos lleva al tema de las métricas de cobertura, que es el del próximo ítem.

El análisis de cobertura es importante cuantas más ramificaciones tiene el código y cuanto más comportamiento exhibe. Por lo tanto, no tiene mucho sentido hacerlo en aplicaciones muy simples, o centradas en datos persistentes.

3.2.3 Métricas de cobertura

Se ha definido una gran cantidad de métricas de cobertura. En esta tesis hemos seguido la clasificación de [Cornett], con algunas variantes. Las métricas más típicas según este autor son:

- Cobertura de sentencias: mide el grado en que las pruebas cubren cada una de las líneas ejecutables (que generan código ejecutable). En Java, a veces se cuentan las sentencias de código intermedio (*bytecode*).
- Cobertura de ramas, también llamada cobertura de decisiones: mide si las condiciones booleanas en sentencias de decisión o de ciclos (*if, while*), se evalúan tanto en el caso de ser verdaderas como falsas; no tiene en cuenta casos diferentes si se usan conectivos lógicos *and* u *or*.
- Cobertura de condiciones: mide la cobertura de todas las expresiones booleanas, tanto en el caso de ser verdaderas como en el caso de ser falsas.
- Cobertura de condiciones múltiple: mide la cobertura de todas las posibles combinaciones de condiciones verdaderas y falsas.
- Cobertura combinada de ramas y condiciones: es una combinación de las métricas de ramas con la de condiciones.
- Cobertura de ramas y condiciones modificada: esta es una métrica compleja que impone que se invoquen al menos una vez cada punto de entrada y salida en el programa, que cada condición en una decisión tome al menos una vez cada uno de los valores posibles, lo mismo para cada rama, y que cada condición en una decisión afecte las salidas de las decisiones en forma independiente.

- Cobertura de trayectorias[33]: mide que cada posible recorrido en cada función haya sido ejecutado por las pruebas, entendiendo por recorrido a cada secuencia única de ramas desde la entrada hasta la salida de la función.
- Cobertura de funciones o de invocaciones[34]: mide si se están invocando todas las funciones, procedimientos o métodos.

Existen muchas otras métricas menos usadas, que hemos dejado de lado.

Medir la cobertura de sentencias parece tener mucho sentido. Sin embargo, no es tan así. Por ejemplo, las siguientes líneas en Java:

```
if ( cond )
                    accion1 ( );
accion2 ( );
```

tienen un 100% de cobertura de sentencias aun cuando nunca probemos el caso de que la condición *cond* tome el valor *false*.
Incluso distorsiona los resultados en otros casos de sentencias *if*. Por ejemplo, si tenemos 99 acciones en la rama de la cláusula *then* y 1 en la rama de la cláusula *else*, y sólo escribimos pruebas para recorrer la cláusula *then*, vamos a tener un 99% de cobertura sin haber recorrido una de las ramas ni siquiera una sola vez. Como, en general, existen más problemas de fallas asociadas a la lógica de control que a los cálculos, estos defectos de la métrica se vuelven más serios.
En general, una métrica de cobertura de sentencias tiene mayor sentido cuando esperamos que los problemas sean más de cálculos que de flujo de control.

La métrica de cobertura de ramas da una solución sencilla a los problemas de la cobertura de sentencias. Su principal problema ocurre cuando hay conectivos lógicos y se ignoran ramas por operaciones de cortocircuito. Por ejemplo, en Java:

```
if ( condicion1 && ( condicion2 || accion1 ( ) ) {
 …
}
```

[33] "Path coverage" es el nombre que le da Cornett.

[34] Cornett separa estas dos métricas, con sutiles diferencias entre ambas.

se podría dar el caso de que nunca se ejecute *accion1*, y sin embargo la cobertura de ramas estar dando el 100%. En lenguajes sin optimizaciones de cortocircuito este problema no existe.
Este problema lo subsana parcialmente la métrica de cobertura de condiciones.
Una mejor solución es la métrica de cobertura de condiciones múltiple, ya que comprueba todas las posibles combinaciones, en el sentido todas las celdas de una tabla de decisión. El problema con esta métrica es que, en ocasiones, se requiere gran cantidad de pruebas, algunas muy difíciles de hallar, para lograr una cobertura total.
Otra posibilidad es usar la métrica de cobertura combinada de ramas y condiciones, que es simple de usar.
La mejor métrica para resolver todos estos problemas juntos es la de cobertura de ramas y condiciones modificada. Sin embargo, es difícil de evaluar y complejo determinar cuáles son todos los casos de prueba que garanticen su cobertura. Por eso se usa para programas muy críticos, y en los cuales la falla puede poner en riesgo vidas humanas.

La métrica de cobertura de trayectorias es también muy usual, ya que responde bien a lo que suele hacerse cuando un programador realiza pruebas de escritorio de caja blanca. Sin embargo, la definición estricta de esta métrica implicaría que los ciclos se deberían ejecutar varias veces, incluso una cantidad indefinida en la mayor parte de los casos. Por ello, se han presentado variaciones de esta métrica, algunas de ellas limitando los recorridos de los ciclos a dos casos: ninguna entrada al ciclo y una o más entradas; para el caso de los ciclos del estilo *do-while*, estos dos casos son: una entrada y más de una entrada.
Hay otros problemas asociados a esta métrica. Una de ellos es que el número de trayectorias crece exponencialmente con el número de ramas. Un diseño de código que limite las bifurcaciones, separando en métodos aparte cuando haya anidamientos muy profundos, puede ayudar a acotar este problema.
Pero también existe el inconveniente de que muchas trayectorias son de cumplimiento imposible debido a relaciones entre los datos, y eso declararía coberturas menores a las reales. Este ejemplo simple en Java lo muestra:

```
if ( cond )
                    accion1( );
accion2( );
if ( cond )
                    accion3( );
```

Dado que *cond* es siempre la misma, hay sólo dos trayectorias que se pueden dar aquí. Sin embargo, una herramienta que evalúe la métrica de cobertura de trayectorias sólo va a reportar un 100% de cobertura si se siguieran 4 trayectorias, dos de las cuales son impracticables.

La métrica de cobertura de funciones tiene escasa utilidad, salvo en pruebas preliminares en las que se quiera asegurar que las mismas están cubriendo todas las partes del programa. La presunción básica de esta métrica es que muchos errores se producen en las interfaces entre módulos, y por eso mide qué tanto se están probando esas interfaces.

Como decíamos, hay muchas otras métricas, algunas de las cuales buscan medir la cobertura de programas concurrentes multi-hilos, otras buscan cobertura de los valores de borde en comparaciones, y así sucesivamente.

Una buena elección de métricas de cobertura para medir la calidad de las pruebas es fundamental. No tiene sentido medir todos los indicadores posibles, más aún cuando hay métricas que, de una u otra forma, incluyen a otras. El trabajo de [Cornett] hace un análisis de cuáles son las métricas que resultan incluidas en otras.

3.2.4 Grado de cobertura deseable

Si bien lograr un 100% parece ser a priori una buena meta, no es así teniendo en cuenta los costos frente a los beneficios que podemos obtener. Habitualmente, un objetivo razonable es llegar a un 80% o 90% en métricas tales como las de sentencias, ramas y combinada de ramas y condiciones.
De tener que elegir una sola métrica, esta última es la más recomendable, aunque ello depende de que la herramienta que usemos para medir cobertura nos brinde ese indicador.

No obstante, hay que tener un especial cuidado con estas recomendaciones. Es común que algunos desarrolladores, con el objetivo de mejorar sus indicadores de cobertura, busquen introducir pruebas sencillas, que casi no agregan valor, pero que mejoren su grado de cobertura.
Tampoco es bueno imponer un valor de grado de cobertura único para todos los desarrollos, como una condición de liberación del software. Lo más recomendable es hacer un análisis de costos y beneficios para cada caso.
Finalmente, aun en el marco de un mismo proyecto, suele haber zonas de diferente criticidad. Las zonas de mayor criticidad requerirán un grado de cobertura mayor, tal vez cercana al

100%, mientras que las zonas menos críticas tal vez admitan un grado de cobertura mucho menor.

3.2.5 Herramientas

Así como existen herramientas para realizar pruebas unitarias, también hay herramientas de análisis de cobertura, que automatizan este proceso.
Entre ellas, destacan Cobertura, Emma, JaCoCo, Clover, SimpleCov, y muchas más, habitualmente fáciles de vincular con varios entornos de desarrollo. Incluso hay entornos de desarrollo que traen sus propias herramientas de análisis de cobertura.

3.3 Pruebas y diseño en capas

3.3.1 Aplicaciones en capas

En el diseño moderno de aplicaciones, se suele trabajar con arquitecturas en capas [Shaw 1996].
Con **arquitectura** nos referimos a formas de descomponer, conectar y relacionar partes de un sistema. Tiene que ver con el diseño macro y con la subdivisión de un sistema en subsistemas o paquetes.
El término capa hace referencia a las capas de una torta, como la torta milhojas argentina; una capa se apoya sobre la anterior, y sólo está en contacto con la capa que está apoyada en ella y la que se encuentra debajo.

En los **patrones arquitectónicos basados en capas** cada módulo – denominado **capa** – depende de una única capa inferior y brinda servicios a una o más capas superiores. Por lo tanto, cada capa conoce a la inmediata inferior, de la que depende, pero no puede conocer nada de las capas superiores que utilizan sus servicios, ni de aquellas capas inferiores no adyacentes.

Las ventajas de estos patrones son:

- Abstracción y encapsulamiento: cada capa se refiere a una abstracción diferente y puede ser analizada y comprendida sin saber cómo están implementadas el resto de las capas. Por ejemplo, podemos tener una capa de acceso a datos persistentes y otra de interacción con el usuario en una aplicación, y cada una es independiente de la otra.
- Bajo impacto de cambios: se puede cambiar la implementación de una capa sin afectar mucho a las demás. En el ejemplo anterior, si cambiamos la base de datos por

almacenamiento en archivos XML, sólo cambiaremos la capa de persistencia, y el resto de la aplicación no se verá afectada.

- Bajo acoplamiento: la dependencia entre capas se mantiene al mínimo. Cada capa depende de una sola capa inferior.
- Una misma capa puede dar servicios a varias capas superiores colocadas en paralelo. Una misma capa de acceso a datos persistentes puede servir para que varias capas – o incluso varias aplicaciones – utilicen el almacenamiento de datos en cuestión.

Las desventajas, si bien son pocas, existen de todos modos:

- A veces un cambio en una capa inferior lleva a cambiar capas superiores que dependen de ella, en una especie de efecto dominó.
- Muchas capas pueden dañar el desempeño por la necesidad de llamadas de métodos en cadena.

Dentro de los patrones en capas, tal vez el más conocido es el de tres capas, que implica dividir el sistema en:

- Capa de *interfaz de usuario* o de *presentación*
- *Lógica de la aplicación* o capa de *dominio* o *modelo de negocio*
- Capa de *acceso a datos* o de *persistencia*

Las capas se comunican mediante interfaces expuestas por cada una.
En el modelo de tres capas, la presentación es la capa superior y muestra lo que ocurre en la capa de la lógica de la aplicación (depende de lo que ocurre en ella). Asimismo, esta última capa depende de la capa de persistencia, en la medida que se comunica con ella cada vez que necesita algo y lo obtiene de ella.
En la figura 3.3 se muestra el esquema típico de tres capas en su forma más sencilla.

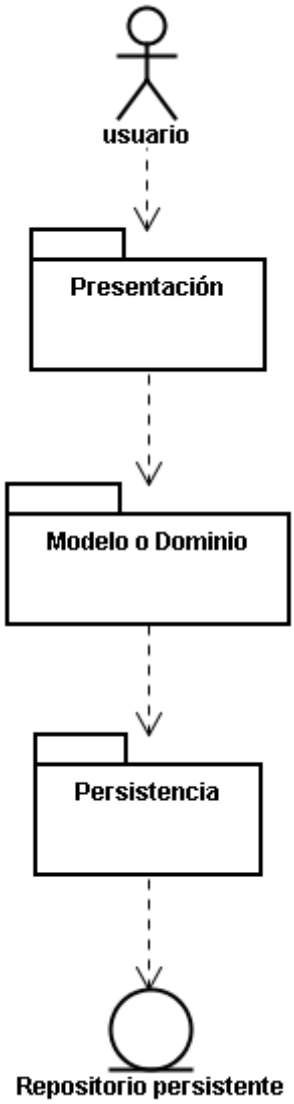


Figura 3.3: arquitectura de tres capas

Las ventajas de este modelo son:

- Posibilitar el cambio de modelo sin modificar la interfaz de usuario o el acceso a los datos persistentes.
- Posibilitar la evolución de la interfaz de usuario sin cambiar el modelo o la base de datos.
- Desarrollar múltiples interfaces de usuario o módulos de acceso a datos para un mismo modelo.
- Posibilidad de utilizar diferentes herramientas y plataformas en cada capa.

- Posibilidad de utilizar diferentes paradigmas en capas distintas (por ejemplo, usar orientación a objetos para el modelo, con una base de datos relacional como mecanismo de persistencia).
- Cada capa puede ser desarrollada por un equipo distinto, con cronogramas relativamente independientes.

Así como se definió un modelo de tres capas, podemos extenderlo para cuatro, cinco o la cantidad de capas que deseemos. Por ejemplo, hay autores que colocan una capa de servicios entre la presentación y la lógica de dominio. En ocasiones necesitamos capas para manejar la provisión de servicios a programas clientes o el consumo de servicios brindados por otras aplicaciones.

Los patrones de arquitecturas en capas, en definitiva, buscan llevar a la práctica el principio de separación de incumbencias[35].
La separación de incumbencias apunta a particionar la aplicación por zonas de cambio. Así, si se espera que la interfaz de usuario cambie por razones diferentes, o más menudo, que la lógica de la aplicación, las clases que implementen la interfaz de usuario deberían estar separadas – tal vez en un paquete diferente – de las clases que implementen la lógica de negocio. De esta manera, los cambios son menos costosos.

3.3.2 Capas y pruebas

Si bien todas las capas de una aplicación se pueden probar, las aplicaciones orientadas a objetos suelen tener el comportamiento principal en la capa de dominio. Por lo tanto, la mayor parte de las pruebas se van a ocupar de la misma, ya que el comportamiento más rico implica mayor complejidad de diseño y mayores probabilidades de problemas. No obstante, todo debería probarse.
Asimismo, el propio diseño de una arquitectura en capas facilita la generación y ejecución de pruebas, ya que en cada capa nos centramos en construir pruebas para un aspecto relevante para esa capa, y de esa manera estamos separando incumbencias.

Ahora bien, podemos llegar a extender esta idea e invertir la causalidad, generando capas con el objetivo principal de facilitar la construcción y ejecución de pruebas.

[35] A veces denominado “separación de intereses”.

Así es como han surgido varias ideas de construcción de una capa intermedia entre la interfaz de usuario y la capa de modelo de negocio.
Por ejemplo, se podrían hacer pruebas de comportamiento en base a casos de uso o *user stories*, trabajando sobre una capa de servicios que haga de Fachada [Gamma 1994], sin comportamiento propio, de los objetos del dominio, y delegando el comportamiento en ellos. Así, las pruebas de comportamiento correrían sobre la capa de servicios, mientras que la capa de dominio se probaría con pruebas unitarias y técnicas de integración.
Hay varios autores que proponen enfoques similares, como Larman [Larman 2003], Freeman y Pryce [Freeman 2010] y Meszaros [Meszaros 2007], aunque cada uno ofrece sus variantes y nombres diferentes para esta arquitectura. Otros, como Brian Marick [Marick 2002], proponen saltear la interfaz de usuario usando un lenguaje de scripting y la biblioteca publicada de la aplicación a probar, en la medida en que exista.
Esta capa de servicios se encuentra justo por debajo de la capa de presentación, con lo cual se podría probar el sistema completo, a excepción de la interfaz de usuario (ver próximo ítem).

Volveremos sobre este tema en capítulos posteriores, ya que lo vamos a necesitar para resolver la problemática central de la tesis.

3.3.3 Un caso especial: la capa de interfaz de usuario

La capa de presentación, o de interfaz de usuario, se ocupa de mostrar a usuarios humanos lo que está ocurriendo en una aplicación y tomar las acciones de los usuarios, validar los datos de entrada y enviarlos a otras partes de la aplicación.
Se suele separar en un módulo aparte para, entre otras cosas, permitir cambios de la interfaz para el mismo modelo de negocio o admitir múltiples tipos de interfaces.

En el ítem anterior dijimos que las pruebas de comportamiento suelen realizarse sobre una capa inmediatamente por debajo de la interfaz de usuario. Entonces, para comprobar la interfaz de usuario necesitamos de pruebas de interacción. Sin embargo, es poco común automatizarlas, en buena medida porque la automatización de las pruebas de interacción trae diversos posibles problemas, que Meszaros [Meszaros 2007] denomina “los problemas de la prueba frágil”:

- Sensibilidad al comportamiento: los cambios de comportamiento provocan cambios importantes en la interfaz de usuario, que hacen que las pruebas de interacción dejen de funcionar.

- Sensibilidad a la interfaz: aún cambios pequeños a la interfaz de usuario suelen provocar que las pruebas automáticas dejen de correr y deban ser cambiadas.
- Sensibilidad a los datos: cuando hay cambios en los datos que se usan para correr la aplicación, los resultados que ésta arroje van a cambiar, lo que hace que haya que generar datos especiales para probar.
- Sensibilidad al contexto: igual que con los datos, las pruebas pueden ser sensibles a cambios en dispositivos externos a la aplicación.

Esta fragilidad, más el problema de la lentitud de las pruebas a través de la interfaz de usuario, son las que más desistimientos han provocado.

Como consecuencia de estos problemas, hay algunos cuidados que debemos tener si queremos automatizar estas pruebas:

- Tratar de no probar en forma conjunta, o como parte del mismo juego de pruebas, el funcionamiento de la aplicación y la lógica de interacción.
- Evitar hacer este tipo de pruebas en forma automática si la lógica de negocio es muy cambiante.
- Dado que estas pruebas son más frágiles que las de comportamiento, hay que volver a generarlas cada tanto.

Resumiendo, hemos presentado las pruebas automatizadas de programador, algunas herramientas y la práctica ágil denominada TDD, con sus variantes. Luego hemos tratado el tema de la cobertura del código y algunas métricas. Para terminar, hemos analizado las arquitecturas en capas y su relación con las pruebas automatizadas.
En el capítulo que sigue nos enfrentaremos al tema de la fragilidad de las refactorizaciones no triviales cuando sólo contamos con pruebas de unidad, y mostraremos algunos esbozos de solución planteados.

4 Fragilidad del refactoring ante cambios de protocolo

Si bien casi todos los autores sostienen la idea de que son las pruebas unitarias automatizadas las que deben avalar la corrección del refactoring, esto no resulta cierto en todos los casos, necesitándose a menudo otras formas de garantizar la preservación del comportamiento. Como este problema se ha advertido ya hace tiempo, han surgido algunas propuestas para resolverlo, algunas bien orientadas y otras no tanto, aunque ninguna de ellas plantea una solución completa al problema. Este capítulo presenta el problema en detalle y analiza las soluciones que se han esbozado.

4.1 El problema

4.1.1 Un ejemplo introductorio

Dijimos en los capítulos anteriores que lo que se suele hacer para asegurar la preservación del comportamiento luego de una refactorización es realizar pruebas que garanticen esa preservación. Tomemos como ejemplo el caso de la refactorización de un método de validación de fechas del capítulo 2.

Recordemos que habíamos cambiado el método *valida* de una clase *Fecha*:

```
public boolean valida() {
        // descarte global:
        if (dia < 1 || dia > 31)
                return false;
        if (mes < 1 || mes > 12)
                return false;
        // veo si el año es bisiesto:
        boolean bisiesto;
        if ((anio % 4 == 0) && ( !(anio % 100 == 0)) )
                bisiesto = true;
        else if (anio % 400 == 0)
                bisiesto = true;
        else bisiesto = false;
        // veo los dias del mes:
        int diasMes = 31;
        if ( (mes == 4) || (mes == 6) || (mes == 9) || (mes == 11) )
                diasMes = 30;
        if (mes == 2) {
```

```
                        if (bisiesto)
                                diasMes = 29;
                        else diasMes = 28;
                }
                // cierre:
                if (dia > diasMes)
                        return false;
                else return true;
        }
```

Por una versión más modular, extrayendo métodos:

```
        public boolean valida() {
                // descarte global:
                if (dia < 1 || dia > 31)
                        return false;
                if (mes < 1 || mes > 12)
                        return false;
                // cierre, comparando con los días del mes:
                if ( dia > diasMes () )
                        return false;
                else return true;
        }

        private int diasMes() {
                int diasMes = 31;
                if ( (mes == 4) || (mes == 6) || (mes == 9) || (mes == 11) )
                        diasMes = 30;
                if (mes == 2) {
                        if ( anioEsBisiesto () )
                                diasMes = 29;
                        else diasMes = 28;
                }
                return diasMes;
        }

        private boolean anioEsBisiesto() {
                if ((anio % 4 == 0) && ( !(anio % 100 == 0)) )
                        return true;
                else if (anio % 400 == 0)
                        return true;
                else return false;
```

```
	}
```

Las pruebas que podríamos utilizar para realizar la refactorización garantizando que se preserva el comportamiento pueden ser:

```
package carlosfontela.utilidades.pruebas;

import org.junit.Test;
import junit.framework.Assert;
import carlosfontela.utilidades.Fecha;

public class PruebaFechaValida {

	private Fecha diaNegativo = new Fecha (-2, 12, 2008);
	private Fecha diaCero = new Fecha (0, 12, 2008);
	private Fecha diaMayorQue31 = new Fecha (32, 12, 2008);
	private Fecha dia31deMesDe30 = new Fecha (31, 11, 2008);
	private Fecha dia29FebreroEnAnioComun = new Fecha (29, 2, 2007);
	private Fecha dia29FebreroEnBisiesto = new Fecha (29, 2, 2008);
	private Fecha dia29FebreroEnBisiesto2000 = new Fecha (29, 2, 2000);
	private Fecha dia29FebreroEnFalsoBisiesto1900 = new Fecha (29, 2, 1900);
	private Fecha mesNegativo = new Fecha (15, -2, 2008);
	private Fecha mesCero = new Fecha (15, 0, 2008);
	private Fecha mesMayorQue12 = new Fecha (15, 13, 2008);
	private Fecha unaFechaValida = new Fecha (15, 4, 2005);

	@Test
	public void diaNegativo () {
		Assert.assertFalse(diaNegativo.valida());
	}

	@Test
	public void diaCero () {
		Assert.assertFalse(diaCero.valida());
	}

	@Test
	public void diaMayorQue31 () {
		Assert.assertFalse(diaMayorQue31.valida());
	}

	@Test
```

```
public void dia31deMesDe30 () {
	Assert.assertFalse(dia31deMesDe30.valida());
}

@Test
public void dia29FebreroEnAnioComun () {
	Assert.assertFalse(dia29FebreroEnAnioComun.valida());
}

@Test
public void dia29FebreroEnBisiesto () {
	Assert.assertTrue(dia29FebreroEnBisiesto.valida());
}

@Test
public void dia29FebreroEnBisiesto2000 () {
	Assert.assertTrue(dia29FebreroEnBisiesto2000.valida());
}

@Test
public void dia29FebreroEnFalsoBisiesto1900 () {
	Assert.assertFalse(dia29FebreroEnFalsoBisiesto1900.valida());
}

@Test
public void mesNegativo () {
	Assert.assertFalse(mesNegativo.valida());
}

@Test
public void mesCero () {
	Assert.assertFalse(mesCero.valida());
}

@Test
public void mesMayorQue12 () {
	Assert.assertFalse(mesMayorQue12.valida());
}

@Test
public void unaFechaValida () {
	Assert.assertTrue(unaFechaValida.valida());
}
```

```
}
```

En este caso, y usando las pruebas recién escritas, pudimos haber realizado la refactorización sin problemas, y garantizar un comportamiento sin cambios, porque habríamos contado con pruebas unitarias del método en cuestión. Corriendo las pruebas luego del cambio, y habiendo verificado que las mismas sigan corriendo satisfactoriamente, podríamos garantizar el éxito de la refactorización.

4.1.2 Pruebas y refactoring

En realidad, las pruebas no brindan una solución formal. Cinnéide [Cinnéide 2000] dice que sólo brindan una solución semi-formal.
Lo que ocurre es que en cualquier desarrollo bien conducido, siempre hay pruebas. Puede que sean automáticas, como ocurre si usamos TDD u otra práctica metodológica que implique automatización de pruebas, o simplemente pruebas que sean corridas en forma manual. Pero pruebas debe haber, ya que son el mecanismo típico de control de calidad de todo proyecto de software. Por lo tanto, utilizar las pruebas para verificar la corrección de una refactorización resulta relativamente económico, ya que no implica el desarrollo de un artefacto nuevo.
Incluso ocurre a menudo que usar las pruebas es el único camino posible, dadas las características de los lenguajes de programación, que hacen tan difíciles las demostraciones de comportamiento formales.

El problema con las pruebas es que nunca son completas, ya que lograr conjuntos completos de pruebas implica cantidades muy grandes de las mismas. Además, las pruebas deben ser modificadas con cada cambio de requerimientos, y si el conjunto de pruebas es grande, el trabajo de modificarlas también lo será. Como vimos en el capítulo 3, se pueden hacer estudios de cobertura de pruebas que analicen qué tanto de la aplicación prueban las mismas.
Incluso podría ocurrir que las pruebas pasen bien en una ejecución y no en otra, lo cual no pasa con el análisis estático de precondiciones. Se podría argumentar que si ello ocurre, las pruebas no estaban bien diseñadas, pero es un problema a tener en cuenta.
Por todo lo anterior, debemos considerar al análisis de corrección de refactorizaciones con pruebas como una aproximación.

Sin embargo, es el método que más se utiliza, por varias razones:

- Cuando encaramos refactorizaciones que no están automatizadas por alguna herramienta que chequee la preservación del comportamiento no hay otra manera.

- Las pruebas suelen estar disponibles en los proyectos bien encauzados, como ya dijimos.
- Como una refactorización es correcta si, luego de la misma, se siguen cumpliendo los requerimientos de la aplicación, si las pruebas son la especificación operacional de los requerimientos, la corrida de las mismas sin errores garantiza la corrección, haciendo las veces de una red de contención ante cambios de comportamiento no deseado.

Por lo tanto, el procedimiento para hacer una refactorización con pruebas, es:

- Si no hay pruebas que verifiquen el comportamiento, escribirlas antes de la refactorización.
- Correr las pruebas para asegurarse de que están funcionando bien.
- Realizar la refactorización.
- Volver a correr las pruebas para asegurarse de que no hubo cambios de comportamiento.

El diagrama de actividades sería el de la figura 4.2.

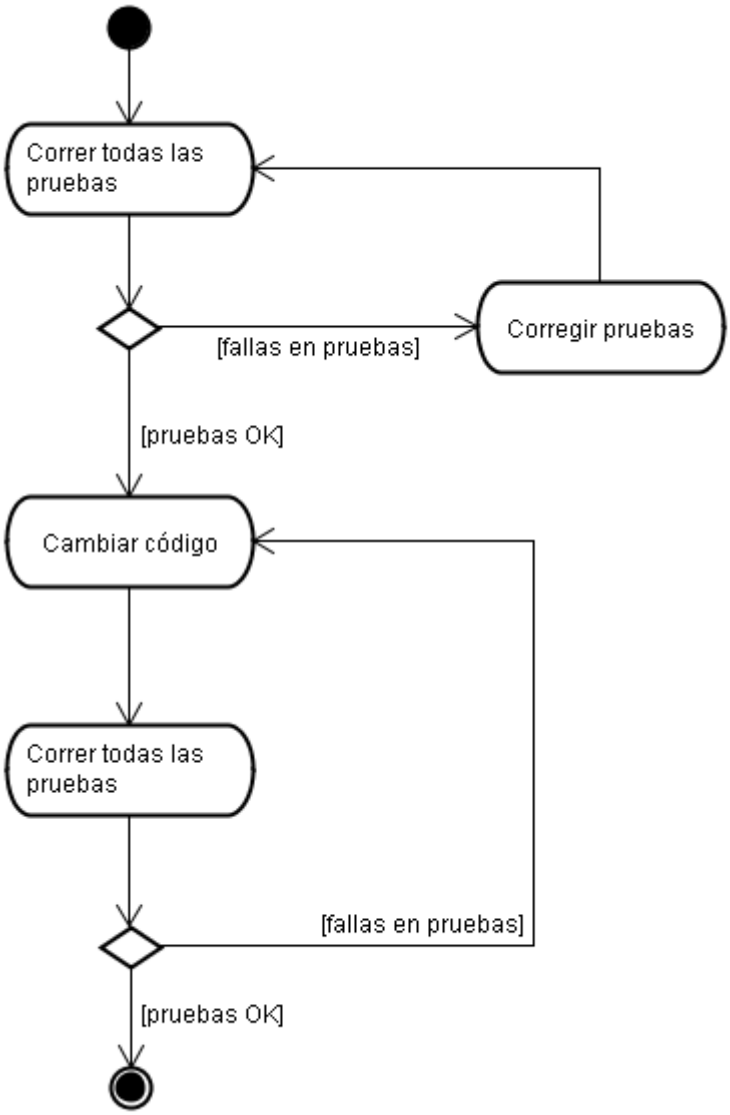


Figura 4.2: diagrama básico de una refactorización

Lo mejor es trabajar con pruebas automatizadas: esto garantiza que siempre se está probando lo mismo y que los resultados no son afectados por condicionantes humanos, como el cansancio o la subjetividad. Y en los lenguajes de tipos estáticos, como Java o C#, prestarle atención a las advertencias del compilador es una excelente idea, que de este modo se convierte en una prueba automática de bajo nivel.

4.1.3 Situaciones de insuficiencia de las pruebas

Ahora bien, si las refactorizaciones cambian protocolos de clases, pueden hacer que las pruebas de verificación del comportamiento ya no sirvan más. Una objeción apresurada

podría ser que no se trata de una refactorización, pues las pruebas fijan cuál es el comportamiento esperado, y las mismas dejaron de funcionar.
Este argumento tiende a ser ingenuo. Como ya vimos, hay distintos niveles de pruebas que definen distintos niveles de requerimientos.

Por ejemplo, si cambiamos el nombre de un método, porque nos parece que el nombre previo no era el adecuado, los requerimientos del usuario se mantienen inalterados, y por lo tanto se trataría de una refactorización, aun cuando la prueba unitaria que verificara ese método dejase de funcionar porque invoca un método que no existe más.
Lo mismo va a ocurrir ante cualquier cambio de protocolo, como por ejemplo:

- Nombres de clases
- Movimiento de una clase entre paquetes
- Nuevos parámetros de métodos
- Añadido o eliminación de nuevos métodos o clases

El problema, entonces, surge porque hay situaciones en las cuales las pruebas dejan de funcionar, aun cuando se esté preservando el comportamiento observable, y por lo tanto ya no hacen las veces de una red de contención del refactoring. Y sin red de contención, refactorizar se torna riesgoso y azaroso.

En lenguajes compilados con chequeo estático de tipos, muchos de estos problemas los detecta el propio compilador. En Java esto aplica incluso para las excepciones que lanzan los métodos. En lenguajes sin chequeo estático, o aún más en los lenguajes interpretados, la falla se va a detectar recién al correr las pruebas.

4.1.4 Situaciones salvables

El caso de nuestro primer ejemplo, en el que refactorizamos un método de validación de fechas, no presentaba problemas, pues el cambio que realizamos era, no sólo muy pequeño, sino que preservaba el protocolo del método en cuestión.
Sin embargo, si ahora decidiésemos cambiar el nombre del método *valida* por *correcta*, ya nos encontraríamos con problemas: los métodos de la clase *PruebaFechaValida* ya no compilarían y nos quedaríamos sin una red de contención. Afortunadamente, en este caso, el renombrado de un método lo hace nuestro entorno de desarrollo en forma automática, actualizando el nombre del método, tanto en la clase *Fecha* como en todas las clases que dependan de la misma, entre ellas *PruebaFechaValida*, con lo cual luego podríamos correr la

prueba modificada sin mayor problema, y tendríamos la corrección de nuestra refactorización asegurada.
Lo mismo ocurriría en un caso de nombre de clase, o de traslado de una clase de un paquete a otro (que en Java es poco más que un cambio de nombre de clase): el entorno de desarrollo hace estas refactorizaciones en forma automática. No es raro que esto sea así, ya que se trata de casos de refactoring catalogados desde los inicios: Opdyke los llamaba, en el catálogo de su trabajo fundacional [Opdyke 1990], *Changing a member function name* y *Changing a class name*.

Ahora bien, ni siquiera estos casos son siempre solucionados por el entorno de desarrollo. Podría ser que el código no estuviese todo disponible en el entorno, por ejemplo porque lo tenemos distribuido, o podría ser que nuestra aplicación tuviese protocolos publicados, y por consiguiente, los nombres a cambiar podrían estar siendo utilizados por otros programas. Volvemos brevemente sobre este tema en el capítulo 7, donde entre los temas relacionados mencionamos el de refactoring de protocolos publicados y el refactoring de código compartido.

4.1.5 Un cambio de protocolos

Por lo que hemos dicho, no siempre verificar la corrección de una refactorización es tan sencillo como lo que hicimos en nuestro ejemplo inicial. Veamos qué ocurre si deseamos introducir un cambio que implica la incorporación de nuevas clases. Tomemos como ejemplo la adopción del patrón de diseño Strategy en un problema de facturación de un supermercado, que realiza distintos descuentos sobre productos dependiendo de si el cliente es un jubilado, un empleado del supermercado, etc.

El código completo del ejemplo se encuentra en el anexo B. Aquí sólo figuran las partes más relevantes del mismo.

Una solución simple (sin el patrón Strategy) a este problema puede verse en el código de la clase *Factura*:

```
package carlosfontela.facturacion;

import java.util.*;

public class Factura {
```

```
private static int ultimoNumero = 0;
private int numero;
private TipoCliente tipoCliente;
private Collection <ItemFactura> items;
private int descuentoEspecialPorcentaje;
private int descuentoEspecialCantidad;

public class ItemFactura {
        … código de la clase interna …
}

public Factura ( ) {
        ultimoNumero++;
        this.numero = ultimoNumero;
        this.items = new ArrayList<ItemFactura> ( );
        this.tipoCliente = TipoCliente.COMUN;
        this.descuentoEspecialPorcentaje = 0;
        this.descuentoEspecialCantidad = 0;
}

public long total ( ) {
        if ( (tipoCliente == TipoCliente.COMUN) && (descuentoEspecialCantidad > 0) )
                return totalDescuentoEspecial (descuentoEspecialPorcentaje,
                                                descuentoEspecialCantidad);
        else if (tipoCliente == TipoCliente.JUBILADO)
                return totalJubilado();
        else if (tipoCliente == TipoCliente.EMPLEADO)
                return totalEmpleado();
        else return totalComun();
}

private long totalComun() {
        long total = 0;
        for (ItemFactura item : this.getItems()) {
                total += item.getCantidad() * item.getPrecio();
        }
        return total;
}

private long totalJubilado() {
        int porcentaje = 20;
        long total = 0;
```

```
                for (ItemFactura item : this.getItems()) {
                        long itemSinDescuento = item.getCantidad() * item.getPrecio();
                        total += (long)(itemSinDescuento * (1-Math.floor(porcentaje / 100.0)));
                }
                return total;
        }

        private long totalEmpleado() {
                ArrayList<Long> codigosConDescuento = new ArrayList<Long>();
                codigosConDescuento.add(new Long(3456));
                codigosConDescuento.add(new Long(7890));
                int porcentaje = 30;

                long total = 0;
                for (ItemFactura item : this.getItems()) {
                        long itemSinDescuento = item.getCantidad() * item.getPrecio();
                        if (codigosConDescuento.contains(item.getCodigo()))
                                total +=
                                        (long)(itemSinDescuento * (1-Math.floor(porcentaje / 100.0)));
                        else
                                total += itemSinDescuento;
                }
                return total;
        }

        private long totalDescuentoEspecial
                        (int descuentoEspecialPorcentaje, int descuentoEspecialCantidad) {
                long total = 0;
                for (ItemFactura item : this.getItems()) {
                        long itemSinDescuento = item.getCantidad() * item.getPrecio();
                        long montoDescontar = 0;
                        if (descuentoEspecialCantidad > 1) {
                        int articulosDescontar = item.getCantidad() % descuentoEspecialCantidad;
                        montoDescontar = (long) (articulosDescontar * item.getPrecio() *
                                        (1-Math.floor(descuentoEspecialPorcentaje / 100.0)));
                        }
                total += itemSinDescuento - montoDescontar;
                }
                return total;
        }

                … otros métodos de la clase Factura …
}
```

El código de pruebas podría ser:

```
package carlosfontela.facturacion.pruebas;

… sentencias import …

public class PruebaTotalFactura {

    private Factura factura;

    @Before
    public void inicializarFactura ( ) {
        factura = new Factura();
        factura.agregarItem(1234, 2, 1120);
        factura.agregarItem(3456, 3, 2150);
        factura.agregarItem(5678, 1, 4320);
        factura.agregarItem(7890, 4, 1030);
    }

    @Test
    public void sinDescuento ( ) {
        factura.setTipoCliente (TipoCliente.COMUN);
        Assert.assertEquals(totalSinDescuentos(), factura.total());
    }

    @Test
    public void descuento20porcientoJubilados ( ) {
        factura.setTipoCliente (TipoCliente.JUBILADO);
        Assert.assertEquals(totalConDescuentoPorcentual(20), factura.total());
    }

    @Test
    public void descuentoEmpleados30porcientoSobreArticulos3456y7890 ( ) {
        factura.setTipoCliente (TipoCliente.EMPLEADO);
        ArrayList<Long> codigosConDescuento = new ArrayList<Long>();
        codigosConDescuento.add(new Long(3456));
        codigosConDescuento.add(new Long(7890));
        Assert.assertEquals( totalConDescuentoPorcentualProductos (30,
                        codigosConDescuento), factura.total() );
    }
```

```
    @Test
    public void descuento20porcientoSegundoArticuloIgual ( ) {
            factura.setTipoCliente (TipoCliente.COMUN);
            factura.setTipoDescuentoEspecial (20, 2);
            Assert.assertEquals(totalConDescuentoPorcentualIguales (20, 2), factura.total());
    }

          … métodos auxiliares varios …

}
```

El código de la clase *Factura* funciona bien, pero necesitaría ser modificado si deseásemos incorporar nuevos descuentos. Esto puede ser evitado utilizando el patrón de diseño Strategy [Gamma 1994]. Recordemos que la premisa del patrón Strategy es que se debe poder variar un algoritmo en tiempo de ejecución y añadir nuevos algoritmos sin modificar la clase básica.

Deberíamos ir a un esquema de clases como el del diagrama de la figura 4.3.

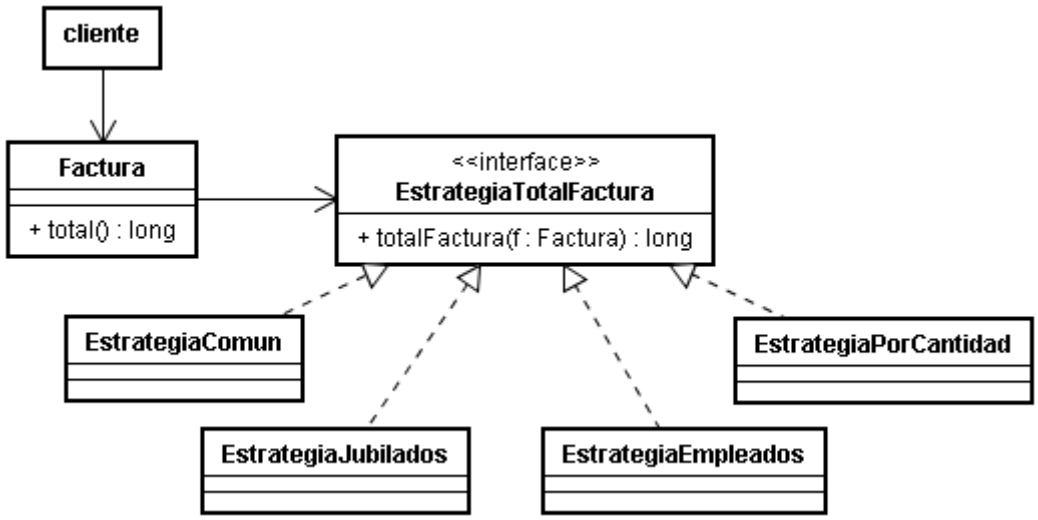


Figura 4.3: propuesta de refactorización usando Strategy

Según Roberts [Roberts 1999], se trata de una refactorización compuesta que se puede descomponer en refactorizaciones simples: *Add Class*, *Add Instance Variable*, *Move Method* y *Move Instance Variable*. Si bien el entorno de desarrollo con el que trabajamos no realiza esas refactorizaciones en forma automática, las hicimos manualmente, moviendo además los métodos privados a las clases de estrategias. Así llegamos a una clase *Factura* como la que sigue, pudiendo comprobarse la preservación del comportamiento con la misma clase de

pruebas, ya que el protocolo que ven los clientes de *Factura* no ha cambiado. A lo sumo, tal vez sería necesario agregar nuevas pruebas para las clases de estrategias, pero ese no es nuestro problema en este contexto.

La clase refactorizada agrega un atributo y cambia el método *total*:

```
package carlosfontela.facturacion;

import java.util.*;

public class Factura {

        … atributos varios …

        private EstrategiaTotalFactura estrategiaTotal;

        public long total ( ) {
                if ( (tipoCliente == TipoCliente.COMUN) && (descuentoEspecialCantidad > 0) )
                        this.estrategiaTotal = new EstrategiaPorCantidad( );
                else if (tipoCliente == TipoCliente.JUBILADO)
                        this.estrategiaTotal = new EstrategiaJubilados( );
                else if (tipoCliente == TipoCliente.EMPLEADO)
                        this.estrategiaTotal = new EstrategiaEmpleados( );
                else this.estrategiaTotal = new EstrategiaComun( );
                return estrategiaTotal.totalFactura(this);
        }

        … otros métodos y clase interna …

}
```

A continuación mostramos la interfaz y una de las clases de estrategias[36], ambas nuevas:

```
package carlosfontela.facturacion;

public interface EstrategiaTotalFactura {
        public long totalFactura (Factura factura);
}
```

[36] El resto de las clases de estrategia se encuentran en el anexo B.

```
package carlosfontela.facturacion;

import carlosfontela.facturacion.Factura.ItemFactura;

public class EstrategiaPorCantidad implements EstrategiaTotalFactura {

	@Override
	public long totalFactura (Factura factura) {
		long total = 0;
		for (ItemFactura item : factura.getItems()) {
			long itemSinDescuento = item.getCantidad() * item.getPrecio();
		long montoDescontar = 0;
			if (factura.getDescuentoEspecialCantidad() > 1) {
				int articulosDescontar = item.getCantidad() %
						factura.getDescuentoEspecialCantidad();
				montoDescontar = (long) (articulosDescontar * item.getPrecio() *
				(1-Math.floor(factura.getDescuentoEspecialPorcentaje() / 100.0)));
			}
			total += itemSinDescuento - montoDescontar;
		}
		return total;
	}

}
```

El inconveniente es que la nueva clase *Factura* sigue necesitando que la modifiquemos si queremos agregar nuevos tipos de descuentos. Esto no está previsto en la refactorización que realiza Roberts [Roberts 1999], pero es esencial al uso del patrón Strategy y condición necesaria para obtener el provecho que se espera.

La solución tradicional a este problema consiste en inyectar directamente la estrategia de descuento a la clase *Factura*. Y no es difícil: bastaría con agregar un método *setEstrategiaTotal*, y eliminar todas las referencias a tipos de cliente que, al menos en este caso, no nos son útiles. La clase *Factura* quedaría como se muestra a continuación:

```
package carlosfontela.facturacion;

import java.util.*;

public class Factura {
```

```
//      ~~private TipoCliente tipoCliente;~~

        private EstrategiaTotalFactura estrategiaTotal;

        … el resto de los atributos y la clase interna quedan igual …

        public Factura ( ) {
                ultimoNumero++;
                this.numero = ultimoNumero;
                this.items = new ArrayList<ItemFactura> ( );
                this.descuentoEspecialPorcentaje = 0;
                this.descuentoEspecialCantidad = 0;
                this.estrategiaTotal = null;
        }

        public EstrategiaTotalFactura getEstrategiaTotal ( ) {
                return estrategiaTotal;
        }

        public void setEstrategiaTotal (EstrategiaTotalFactura estrategiaTotal) {
                this.estrategiaTotal = estrategiaTotal;
        }

        public long total ( ) {
                return estrategiaTotal.totalFactura(this);
        }

        … el resto de los métodos no cambia …

}
```

Ahora sí, si deseamos agregar nuevas estrategias de descuentos, sólo debemos agregar clases que implementen la interfaz *EstrategiaTotalFactura*.

El problema es que ahora las pruebas no compilan[37] más, porque no debemos asignarle a una factura su tipo de cliente al inicializarla, sino directamente un tipo de estrategia de descuento.

[37] Decimos "no compilan" porque estamos trabajando en Java, y en esta plataforma es el compilador el que chequea que la interfaz de un método sea compatible con su invocación. En lenguajes de chequeo dinámico

Si intentamos compilar la clase *PruebaTotalFactura*, nos encontramos con errores de chequeo estático.

Una solución posible es modificar *PruebaTotalFactura* como se muestra a continuación:

```
package carlosfontela.facturacion.pruebas;

… sentencias import …

public class PruebaTotalFactura {

    private Factura factura;

    @Before
    public void inicializarFactura ( ) {
        … este método queda igual …
    }

    @Test
    public void sinDescuento ( ) {
        factura.setEstrategiaTotal ( new EstrategiaComun() );
        Assert.assertEquals(totalSinDescuentos(), factura.total());
    }

    @Test
    public void descuento20porcientoJubilados ( ) {
        factura.setEstrategiaTotal ( new EstrategiaJubilados() );
        Assert.assertEquals(totalConDescuentoPorcentual(20), factura.total());
    }

    @Test
    public void descuentoEmpleados30porcientoSobreArticulos3456y7890 ( ) {
        factura.setEstrategiaTotal ( new EstrategiaEmpleados() );
        ArrayList<Long> codigosConDescuento = new ArrayList<Long>();
        codigosConDescuento.add(new Long(3456));
        codigosConDescuento.add(new Long(7890));
        Assert.assertEquals (totalConDescuentoPorcentualProductos (30,
                                                    codigosConDescuento), factura.total());
```

diríamos que la prueba falla o se rompe. En este trabajo vamos a usar dos términos indistintamente: “no compila” y “se rompe”.

```
    }

    @Test
    public void descuento20porcientoSegundoArticuloIgual ( ) {
            factura.setEstrategiaTotal ( new EstrategiaPorCantidad() );
            factura.setTipoDescuentoEspecial (20, 2);
            Assert.assertEquals(totalConDescuentoPorcentualIguales (20, 2), factura.total());
    }

          … métodos auxiliares varios …

}
```

Pero ahora ya no podemos afirmar que el comportamiento se sigue preservando. Si hemos cambiado la prueba para que compile, está claro que lo hicimos violando un principio fundamental, y no podemos garantizar que el comportamiento sea el mismo. En efecto, la clase de prueba era la red de contención que acreditaba la preservación del comportamiento, en la medida en que la prueba no variase. Pero si la prueba cambia, ya no hay nada que asegure la corrección de la refactorización.

Sin embargo, habría una consideración adicional a realizar: este último cambio que hicimos, de hacer que la estrategia sea inyectada a la clase *Factura*, ¿es una refactorización? O dicho de otra manera, ¿preserva el comportamiento? ¿O es un cambio de requerimientos?
La respuesta a esta pregunta, nuevamente, tiene que ver con el carácter de "observable" del comportamiento, y de que este cambio afecte o no al cliente. Por ejemplo, si antes de este último cambio no se hubiese permitido introducir nuevos tipos de descuentos en forma dinámica, y se realizó el cambio de diseño en respuesta a un pedido de cambio del cliente, podríamos argumentar que el cambio de comportamiento es "observable", y por lo tanto no es una refactorización. Si, en cambio, se trata solamente de una mejora de diseño que prevé futuros cambios, pero que en lo inmediato no debería afectar el comportamiento observable, sí es una refactorización. Este último caso parece ser el más ajustado a nuestro problema.

Más allá de las especulaciones recién hechas, lo que queda claro es que tenemos un problema en cuanto a la manera de verificar la preservación del comportamiento: cada vez que el protocolo de una clase cambia, las pruebas que verifican el comportamiento de esa clase pueden dejar de compilar, y por lo tanto nos quedaríamos sin una manera de chequear la preservación del comportamiento.

Este es el tema que motivó este trabajo, y consecuentemente lo trataremos con mayor profundidad luego.

Una digresión final. Convendría haber colocado parámetros en el constructor de *Factura*, en vez de utilizar un método "setter", para cumplir con el principio de diseño de dejar a los objetos en un estado válido a la salida del constructor, como lo proponen Meyer [Meyer 2000] y Beck [Beck 1996]. Si se hizo así fue para mayor claridad de exposición, aunque un cambio para mejorar este diseño volvería a enfrentarnos con un cambio de protocolo que provocaría una necesidad de modificación de la clase de prueba.

4.1.6 Casos más complejos

Hay casos aún peores que el que enfrentamos en el apartado anterior al introducir el patrón Strategy.

Por ejemplo, la clase *ItemFactura* que usamos como clase interna a *Factura* no parece que deba ser pública. Al fin y al cabo, su mera existencia es una cuestión de implementación, e incluso se trata de una clase sin comportamiento.
Sería mejor, desde el punto de vista del diseño, que *ItemFactura* fuese una clase interna privada, y que se le pueda pedir a la *Factura* un iterador que permita recorrer sus renglones. Esto también va a provocar que las pruebas dejen de compilar, y los cambios sobre las pruebas serían mayores y más difíciles de hacer. En consecuencia, este cambio será más propenso a errores.

El que acabamos de citar es un caso particular de una situación más general, que se da cuando eliminamos una clase sin comportamiento, cuya existencia sólo está motivada para mantener datos relacionados entre sí. Cuando combinamos esta clase con aquélla que tiene el comportamiento correspondiente, nos encontraremos que toda prueba que referencie a la clase eliminada va a fallar.

Cuando una clase está muy acoplada a otras (dicho de otra manera: muchas otras clases dependen de ella, o tiene muchas clases clientes), cualquier cambio que hagamos sobre el protocolo de la misma va a afectar a muchos clientes. Por eso, es muy probable que una refactorización de este tipo provoque la necesidad de refactorizaciones en cascada para adaptar las interacciones entre las clases y lograr que las pruebas corran.

El caso más problemático sería el de una clase de acceso global, y como caso particular aquellas que responden al patrón Singleton [Gamma 1994], que genera por definición una instancia de acceso global.
Pero en todo sistema suele ocurrir que, sin caer en clases globales, existen algunas que son tan utilizadas que se convierten de hecho casi en clases globales.

Otro caso podría ser la refactorización que se realice para crear una clase abstracta como madre de más de una clase preexistente. En este caso tampoco va a alcanzar con las pruebas unitarias de las clases hijas.

Hay muchos otros casos. Por ejemplo, reemplazar el uso de un constructor explícito por un uso del patrón Factory Method [Gamma 1994] nos obligará a adaptar todas las pruebas que utilizasen ese constructor.

Asimismo, las grandes refactorizaciones caen casi siempre en esta categoría.
Supongamos, por ejemplo, que deseamos agregar una capa de servicios a la arquitectura de una aplicación, entre la capa de presentación y la de dominio, de modo tal de facilitar el acceso al sistema desde distintas interfaces de usuarios y otros sistemas. La necesidad de probar la preservación de comportamiento de la aplicación a través de esta capa de servicios hará que prácticamente no tengamos ninguna prueba previa que nos sirva.
Una situación parecida se da cuando, para evitar dependencias cíclicas entre paquetes movemos una o más clases de visibilidad limitada de un paquete a otro. Felizmente, este caso es bastante más sencillo de resolver. No ocurre lo mismo, sin embargo, si hubiera relaciones de herencia entre clases que estuviésemos moviendo de un paquete a otro.

Decimos que el refactoring es **frágil** cuando, por el hecho de que una refactorización modifica el protocolo de una o más clases, las pruebas unitarias se rompen o fallan, dejando a dicha refactorización sin su red de contención natural.

En resumen, encontramos que el refactoring es frágil en determinadas situaciones, entre las cuales hemos destacado:

- Cuando separamos un algoritmo en una clase aparte[38] que luego nos obliga a inyectar la instancia.
- Cuando cambiamos la firma de un constructor debido a la inyección de una dependencia en el mismo.
- Cuando eliminamos una clase[39] o le restringimos su visibilidad.
- Cuando una cambiamos el protocolo de una clase muy acoplada a otras del sistema, provocando refactorizaciones en cascada de más clases y pruebas.
- Cuando modificamos una clase de acceso global[40].
- Cuando creamos una clase abstracta como base de clases preexistentes.
- Cuando reemplazamos el uso de un constructor por el uso de un patrón de creación[41].
- Cuando movemos clases entre paquetes, de manera tal que provoquemos que las relaciones de herencia crucen las fronteras entre paquetes.
- Cuando se hacen grandes refactorizaciones, agregando o suprimiendo capas a un sistema.

Esta lista no pretende ser exhaustiva, pero muestra suficientes ejemplos de situaciones problemáticas que ameritan que las tengamos en cuenta.

4.1.7 Un enfoque erróneo

Según Pipka [Pipka 2002], el enfoque más habitual en la práctica es realizar la refactorización y ver si las pruebas siguen funcionando. Si ninguna prueba se rompe, nos quedamos tranquilos. 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. Nótese que es el procedimiento que usamos en el último ejemplo.

Pipka [Pipka 2002] llama "enfoque rápido y sucio" a esta práctica, y es bastante obvio por qué: se pierde mucho control y seguridad, al punto que la presunta reconstrucción de las pruebas no puede garantizar la preservación del comportamiento.

---

[38] En nuestro caso, lo hicimos con la introducción del patrón Strategy, pero hay muchas más situaciones, como la implementación de un Command o un State [Gamma 1994].

[39] Por ejemplo, porque no tiene comportamiento.

[40] El ejemplo que dimos tiene que ver con el patrón Singleton [Gamma 1994], que provoca un acceso global del resto del sistema.

[41] Los más típicos son Factory Method, Abstract Factory y Builder [Gamma 1994].

Por lo tanto, esta no es una solución para el problema que hemos denominado de fragilidad del refactoring.

4.2 Propuestas de solución existentes

Este problema que hemos planteado no es nuevo, y por lo tanto se han enunciado algunas propuestas de solución, aunque casi no existen publicaciones que las analicen íntegramente.

Uno de los trabajos que más se ha ocupado de este asunto es el de Pipka [Pipka 2002], que enuncia algunas soluciones posibles para evitar el que él mismo denomina enfoque sucio, que mencionamos recién.

4.2.1 Test-First Refactoring

Una primera opción es seguir el protocolo de TDD a fondo. El procedimiento, por lo tanto, sería:

- Adaptar las pruebas antes de refactorizar.
- Comprobar que la nueva versión de las pruebas falla con el código antiguo.
- Refactorizar la aplicación.
- Corremos nuevamente las pruebas en su segunda versión, que ahora deberían funcionar bien.

A este método, Pipka [Pipka 2002] lo llama **Test First Refactoring**. En un trabajo posterior [Pipka-2] propone una ligera variante que llama **Test Driven Refactoring**.

Este enfoque tiene una ventaja innegable: establece el protocolo 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, dejando esta decisión para el momento de la construcción. Claramente, el *Test First Refactoring* es preferible al "enfoque sucio", y en éste se verifican las típicas ventajas de TDD:

- Las pruebas se escriben antes que el código.
- Todo lo que puede fallar estará siendo probado.
- Todas las pruebas se pueden correr en todo momento.

El diagrama de actividades de *Test First Refactoring* sería el de la figura 4.4.

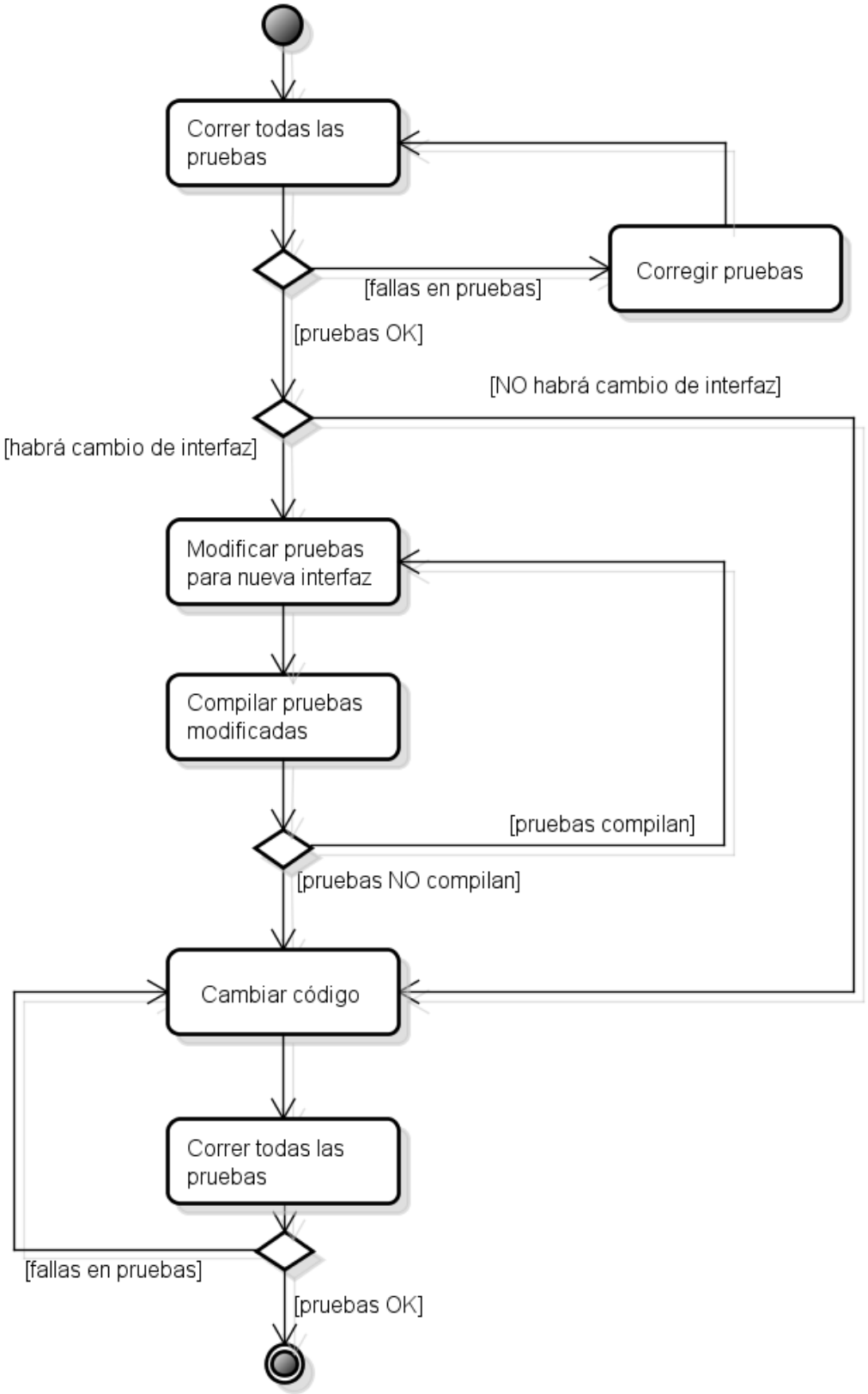


Figura 4.4: procedimiento del Test-First Refactoring

Notemos que este diagrama, si no hubiera cambios de protocolo en las clases a refactorizar, es igual al de refactoring tradicional que habíamos mostrado en el capítulo 2.
Este método sirve incluso cuando no tenemos pruebas automatizadas del código a refactorizar. En esos casos, escribiríamos primero una prueba que pase, usando el código como se encuentre antes de la refactorización, que se presume correcto. Ahora el propio código es quien nos indica la calidad de la prueba, fallando la misma si está mal escrita.

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, ¿cómo podemos garantizar que la nueva versión de la prueba, obviamente diferente de la anterior, comprueba el mismo comportamiento que antes? Una revisión cuidadosa del desarrollador podría servir para quedarse más tranquilo, pero en rigor de verdad todo está quedando librado a su criterio que, como ya dijimos al defender las pruebas automatizadas, puede estar debilitado por peculiaridades propias de los seres humanos, tales como cansancio, falta de atención, apuro, etc. Además, si la refactorización no es una pequeña modificación localizada, sino un cambio más amplio, la probabilidad de error crece sensiblemente.
En su segundo trabajo sobre el tema [Pipka-2], Pipka propone partir de requerimientos, un poco en línea con el que luego llamaremos *refactoring asegurado por pruebas de comportamiento,* aunque plantea dividir los requerimientos en "tareas técnicas que puedan expresarse en pruebas unitarias" en vez de usar pruebas de aceptación.

4.2.2 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. Esto es, en todo programa orientado a objetos, una clase o conjunto de clases brindan servicios a otras clases, a las que habitualmente denominamos *clientes* de aquéllas. 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 ni su protocolo. 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 [Lippert 2006], al decir que, en última instancia, el refactoring debería asegurarse con pruebas de integración. En este trabajo lo denominaremos **refactoring asegurado por los clientes**, y es parte del método que proponemos.

Sin embargo, el método, así enunciado, presenta algunas complicaciones.
Primero, porque si las pruebas unitarias dejan de compilar, es previsible que lo mismo ocurra con las clases clientes, que venían usando el protocolo anterior, ahora modificado. Por lo tanto, las propias clases clientes deben ser cambiadas junto con las pruebas unitarias. No es que esto sea imposible de resolver, pero agrega complejidad a la solución.
En segundo lugar, no siempre va a poder ser aplicado si estamos trabajando con RDD o algún otro enfoque que utilice abundantemente objetos ficticios. 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. Esto es, ¿cómo garantizamos que las pruebas de los clientes cubren el código que está siendo refactorizado? Porque si no es así, es decir, si las pruebas de los clientes no provocan recorridos en el código que se va refactorizar, la eliminación de ciertas pruebas del conjunto nos 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. Sólo con cobertura redundante podríamos estar seguros. Y deberíamos poder determinar que el código a refactorizar sigue siendo recorrido cuando eliminamos del conjunto la o las pruebas problemáticas. Se podría advertir que, si una aplicación se construye siguiendo el protocolo de TDD en forma exhaustiva, todas las clases del sistema estarían cubiertas por las pruebas de sus clientes. Pero aun así no es tan sencillo. Ya volveremos sobre esto en el método propuesto por este trabajo.

4.2.3 Refactoring asegurado por pruebas de comportamiento

Cuando surgieron formas más evolucionadas de TDD, tales como ATDD y BDD, sus impulsores afirmaron en diversos trabajos que las pruebas de comportamiento (en el sentido que les damos en este trabajo, como un tipo particular de pruebas de aceptación) sirven para

garantizar refactorizaciones más seguras. Así lo afirmaron, entre otros, Mugridge y Cunningham [Mugridge 2005].

En efecto, si la condición a verificar de una refactorización para garantizar su corrección es la preservación del comportamiento, qué mejor que mantener pruebas de comportamiento, en el sentido de ATDD o BDD, y correrlas en cada refactorización para asegurarnos que siguen funcionando. En este trabajo denominaremos **refactoring asegurado por pruebas de comportamiento** a ese enfoque.

Como ocurría en el *refactoring asegurado por los clientes* con las pruebas técnicas de integración, las pruebas de comportamiento podrían ser una red de seguridad para las pruebas técnicas. En consecuencia, podríamos proceder de manera similar a la del enfoque anterior:

- Refactorizar y determinar si las pruebas técnicas no se rompen.
- Si las pruebas técnicas se rompen, eliminar provisoriamente las mismas del conjunto de pruebas.
- Correr las pruebas de comportamiento, y asegurarse de que pasan.
- Cambiar las pruebas que antes fallaban, para luego incorporarlas de nuevo en el conjunto de pruebas y asegurarnos finalmente que todo sigue corriendo bien.

Pareciera que esta es una solución impecable, por la propia definición del refactoring. No obstante, también debería provocar algunos reparos.
En efecto, si bien las pruebas de comportamiento garantizan que la refactorización es correcta, lo cierto es que, en aplicaciones medianas, son pruebas muy amplias, que verifican mucho código fuente cada una. Por lo tanto, una buena práctica de desarrollo sería mantener las pruebas de comportamiento, pero desarrollar también pruebas de menor alcance. Y estas pruebas de menor alcance se van a romper.

Además, al igual que en el enfoque de refactoring asegurado por los clientes, está la cuestión de la cobertura. En este caso el dilema sería cómo garantizar que las pruebas de comportamiento recorren también el código que prueban las pruebas técnicas. Encima, al ser pruebas mucho más amplias, es más difícil ver la correlación entre pruebas de comportamiento y pruebas técnicas. También sobre esto volveremos al proponer nuestro método.

En resumen, hemos visto que, cuando una refactorización modifica el protocolo de una o más clases, casi siempre ocurre que las pruebas unitarias se rompen o fallan, dejando a dicha refactorización sin su red de contención natural. Eso es lo que hemos llamado refactoring frágil, y lo hemos ejemplificado en unas cuantas situaciones típicas, desde ya que no exhaustivamente.

Para solucionar este problema, se han propuesto algunas iniciativas, las que hemos denominado:

- Test-First Refactoring
- Refactoring asegurado por los clientes
- Refactoring asegurado por las pruebas de comportamiento

Sin embargo, ninguno de estos enfoques ofrece una respuesta integral. Esa es la razón que motiva esta tesis y cuya solución abordaremos en el próximo capítulo.

5 Refactoring asegurado por niveles de pruebas

El refactoring, como práctica de desarrollo de software, tiene que cumplir con dos objetivos centrales:

- Ser útil (o **efectivo**): debe mejorar la calidad interna con vistas al mantenimiento.
- Ser **correcto**: debe preservar el comportamiento observable.

Entre ambos objetivos, este trabajo se centra en el segundo. Es decir, dejando a un lado la cuestión sobre si una refactorización es o no útil, vamos a desarrollar una propuesta metodológica que garantice la corrección.
El método que vamos a presentar en este capítulo, basado en otras propuestas de solución al problema (las que en el capítulo precedente hemos llamado *refactoring asegurado por los clientes* y *refactoring asegurado por las pruebas de comportamiento*), las supera por su enfoque metodológico detallado.
La propuesta es acompañada por algunas prácticas metodológicas y de diseño de arquitectura que facilitan su aplicación, y que también presentamos en este capítulo.

A nuestro método lo llamaremos **refactoring asegurado por niveles de pruebas**.

5.1 Propuesta de este trabajo

5.1.1 Refactoring asegurado por niveles de pruebas

Refactoring asegurado por niveles de pruebas es una técnica que permite alcanzar un grado de seguridad más alto en cuanto a la corrección del refactoring, utilizando redes de seguridad en forma escalonada mediante pruebas cada vez más abarcativas.

El aspecto central del método 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. 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.

En pocas palabras, el procedimiento es:

- Refactorizar usando las pruebas unitarias como red de apoyo 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.

En cada paso, al excluir 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 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.
Las pruebas de comportamiento son el último nivel a considerar. Esto es así porque, si las pruebas de comportamiento están bien escritas, una falla en las mismas implicaría que no estamos ante una refactorización, pues no se estaría pudiendo garantizar la preservación del comportamiento. Por lo tanto, y en última instancia, las pruebas de comportamiento hacen las veces de invariantes para el refactoring.

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, como mostramos en capítulos anteriores. 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 [Beck 1999, Fowler 1999], que enfatizan que la práctica del refactoring debería ser realizada con herramientas que no entorpezcan el desempeño de las tareas del programador.

La figura 5.1 muestra los tres tipos de pruebas y las partes de la aplicación que prueba cada tipo.

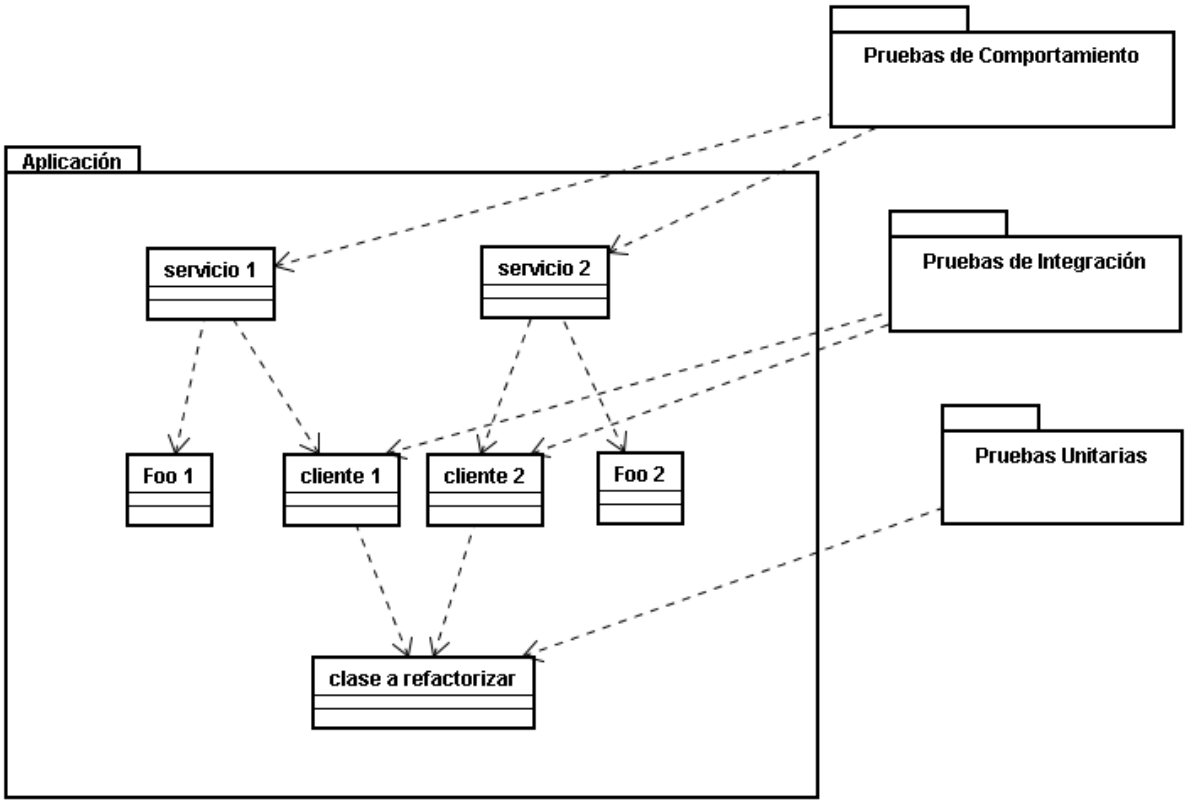


Figura 5.1: aseguramiento escalonado del refactoring con niveles de pruebas

La figura 5.2 muestra el diagrama de actividades simplificado del método.

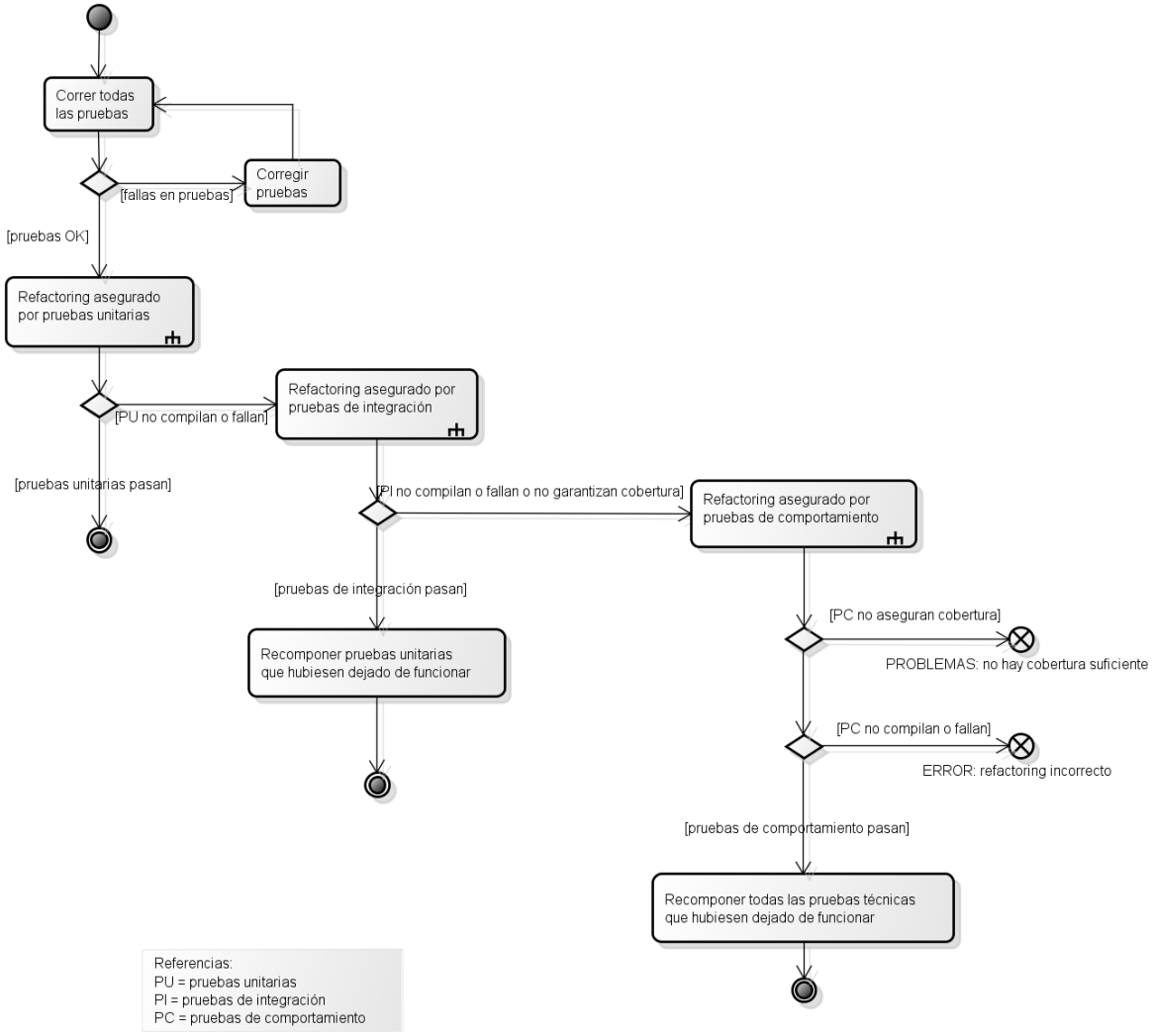


Figura 5.2: refactoring asegurado por niveles de pruebas

El diagrama se tornaría muy complejo si abrimos cada actividad compuesta, por lo cual desglosaremos cada etapa del método más adelante. No obstante, en este diagrama se puede apreciar bien, por sus posiciones desplazadas horizontalmente, los tres niveles del método propuesto.

Lo que sí es 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. Como ya dijimos antes, si una prueba de comportamiento cambia, estamos ante un cambio de requerimientos, no de una refactorización.

Las **pruebas de comportamiento son los invariantes** que garantizan la corrección de una refactorización. Deben mantenerse en todos los hitos de una refactorización que sean considerados estables.

Por lo tanto, si las pruebas de comportamiento fallan, significa que la refactorización no era segura y debe deshacerse (volver al código antes de la refactorización).

Ahora bien, hay algo en lo que no hemos hecho suficiente énfasis desde que comenzamos a presentar el método: la cuestión de la cobertura.
Es que la condición de la cobertura de unas pruebas por otras de mayor granularidad es central. 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 mayor granularidad, si ocurriera, sería un falso positivo.

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**.

Lo que acabamos de definir es lo central del método, que nos indica cómo proceder en diversos tipos de refactorizaciones, de las más simples a las más complejas, incluyendo los problemas de cobertura.
Hay además, un par de recomendaciones que acompañan al método, que si bien no son necesarias, lo facilitan mucho:

- Una recomendación metodológica de desarrollo, que nos indica las prácticas a seguir durante el desarrollo de la aplicación, de modo tal que las refactorizaciones sean sencillas de realizar según el método propuesto.
- Una recomendación de arquitectura, que facilite la construcción de la aplicación siguiendo la propuesta metodológica de desarrollo y luego facilite las refactorizaciones siguiendo el método de este trabajo.

En los próximos ítems analizaremos estas recomendaciones.

5.1.2 Recomendación metodológica de desarrollo

Lo que sigue es la propuesta que define las condiciones planteadas a la metodología de desarrollo de una aplicación para que luego pueda aplicarse fácilmente el refactoring asegurado por niveles de pruebas.

Proponemos que las aplicaciones se hagan siguiendo las prácticas de ATDD en forma completa, incluyendo 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.
Esto es así porque necesitamos la preexistencia de distintos niveles de prueba que se cubran entre sí. De hecho, cualquier refactorización necesita de pruebas automatizadas, pero más aún el método por niveles que proponemos aquí.

Además, proponemos seguir las recomendaciones de Larman [Larman 2003] y el patrón *Test Class per User Story* de Meszaros [Meszaros 2007], en ambos casos con algunas ligeras variantes.
La propuesta de Larman [Larman 2003] es 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. Si bien no haremos los diagramas de secuencia, sí reuniremos las operaciones del sistema en una o más clases de servicios.
El patrón *Test Class per User Story* [Meszaros 2007], 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 comportamiento.

Por lo tanto, el procedimiento para construir la aplicación sería, 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.
- Escribir las pruebas de comportamiento, sin lógica de presentación, para cada método de la clase de servicios.
- Escribir la clase de servicios que haga que la prueba de comportamiento pase, implementando la interfaz y usando objetos ficticios para representar las clases servidoras del dominio de la aplicación.

Luego, por cada clase del dominio de la aplicación se seguirá el protocolo de UTDD:

- Definir el protocolo de la clase.
- Escribir las pruebas técnicas de la clase, que serán pruebas unitarias si se basan en operaciones primitivas, o de integración si utilizan servicios de otras clases del sistema.
- Escribir la clase.

Es necesario volver a aclarar que la recomendación metodológica de desarrollo no es condición necesaria para el refactoring asegurado por niveles de pruebas, sino solamente una sugerencia. De hecho, si no se hubiese seguido, siempre es posible generar las pruebas necesarias antes de comenzar la refactorización.
No obstante, cada día es más habitual recurrir a variantes de ATDD para desarrollar aplicaciones en la industria, con lo cual esta propuesta no parece ser muy difícil de seguir[42].

5.1.3 Recomendación de arquitectura

Al igual que lo que ocurre con la recomendación metodológica de desarrollo, el refactoring asegurado por niveles de pruebas se ve facilitado por la propuesta arquitectónica que planteamos a continuación, aunque la misma no es condición necesaria para el método en cuestión.

En principio, proponemos una arquitectura en capas, que al menos contemple las tres capas tradicionales: presentación, dominio y persistencia.
Además, para separar las clases asociadas a requerimientos, que definen los servicios brindados por la aplicación, introducimos una capa de servicios. Las pruebas de comportamiento se ejecutan contra clases de la capa de servicios, según lo que dijimos para la recomendación metodológica, mientras que las pruebas técnicas (de integración y unitarias) se ejecutan contra clases de la capa de dominio.
Por lo tanto, el diagrama de la arquitectura sería el de la figura 5.3.

[42] Ver apartado “Discusión” en el capítulo 7.

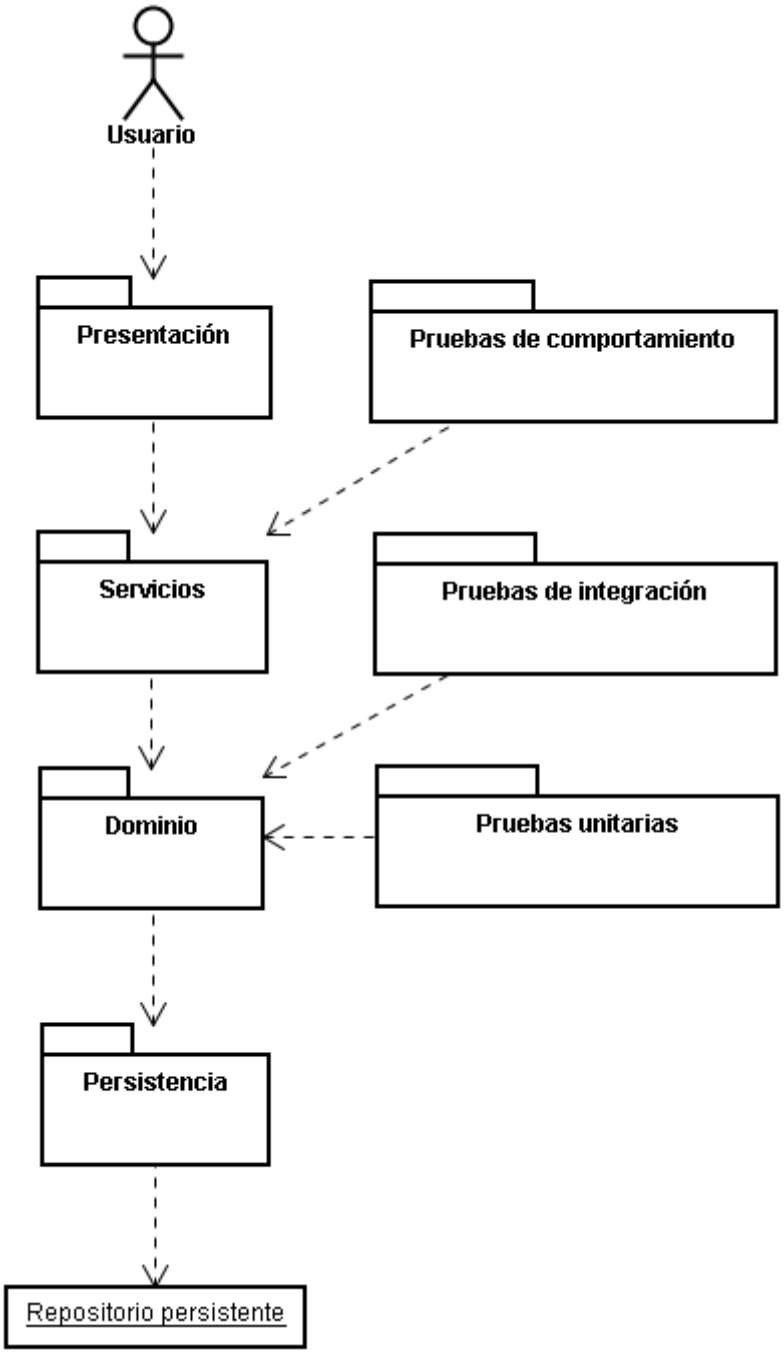


Figura 5.3: arquitectura propuesta para el refactoring asegurado por niveles de pruebas

La capa de servicios, es una implementación del patrón Façade [Gamma 1994], que delega la implementación de las funcionalidades en clases de dominio. En algunos casos puede agregársele el manejo de transacciones y el alcance de la persistencia [Bazzocco 2012].
Lo realmente importante de la capa de servicios es que está escrita con un nivel de abstracción tal que sirva como interfaz (humana o no) de la aplicación. Por eso, las pruebas contra la capa

de servicios son pruebas de aceptación, escritas pensando en el dominio del problema y con el vocabulario de los usuarios, en el sentido del Domain Driven Design [Evans 2003].

Respecto de la organización de las pruebas en paquetes, no hay ninguna recomendación. En el caso de estudio vamos a establecer una regla que definiremos allí, pero no constituye una recomendación de carácter general.
Para el momento de ejecutar las pruebas, y para que éstas no demoren mucho tiempo, se pueden usar repositorios en memoria, al estilo de los Fake Objects [Meszaros 2007].

Esta arquitectura recomendada facilita en mucho el uso del enfoque metodológico de esta tesis. Y aunque no es de cumplimiento forzoso, conviene seguirla, sobre todo teniendo en cuenta que es bastante habitual en las aplicaciones industriales de hoy en día.

5.1.4 El problema de la cobertura

Desde el punto de vista de la cobertura, nuestro enfoque metodológico necesita asegurar cobertura redundante de unas pruebas con otras. De allí que no busquemos un análisis de cobertura en el sentido tradicional. De hecho, lo que nosotros necesitamos es una cobertura mayor al 100% en los trozos de código que se quiere refactorizar.
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 (ver capítulo 6).
En principio, la mejor métrica de cobertura para nuestro enfoque es la de sentencias y de ramas y condiciones.

5.1.5 Distinción entre errores de compilación y fallas

Debemos hacer una aclaración respecto de los términos “compilar” y “fallar” al hablar de las pruebas, ya que estas instancias no se dan necesariamente en todos los lenguajes de la misma manera.

- En lenguajes compilados con tipos de datos fuertes y estáticos (Java, C#, C++, Eiffel), al compilar una prueba, la compilación va a fracasar si se ha cambiado algo en la firma de los métodos que ésta ejercitaba. La falla en las pruebas a posteriori de la compilación va a ser una situación menos frecuente (aunque no imposible), ya que en casi todos los casos va a ser el compilador el que va a advertir el problema antes de poder ejecutarlas. En el caso particular de Java, al ser las excepciones parte de la firma de los métodos, también un cambio en las excepciones que lanzasen esos métodos va a

provocar un error de compilación, con lo cual las fallas de las pruebas son aún más infrecuentes[43].

- En lenguajes compilados con tipos de datos menos estrictos o con tipeo dinámico, como Smalltalk, la compilación no siempre va a detectar todos los cambios en firmas de métodos, con lo cual el problema en muchos casos se va a detectar recién al correr la prueba automática.
- En lenguajes interpretados (Python, Ruby) no hay una instancia de chequeo estática, por lo que la situación en la cual la prueba automática no compila no se va a dar nunca, y sólo nos enteraremos del problema cuando la prueba falle.

5.1.6 Sobre la exclusión de las pruebas de interacción en el método

Una cuestión que hay que aclarar es por qué no usamos pruebas de interacción automatizadas, que corran al nivel de la interfaz de usuario. Hay dos respuestas a este asunto.

En primer lugar, ya expusimos los inconvenientes de automatizar pruebas de interacción, que suelen ser frágiles y lentas.

En segundo, las pruebas de comportamiento, si se construyen a partir de ejemplos en el sentido de ATDD, y están consensuadas con el cliente, son especificaciones ejecutables, y no cambian si no hay un cambio en los requerimientos. Recordemos que, para que esto que decimos sea cierto, deberían expresar el comportamiento del sistema con el vocabulario del dominio.

Por eso es que el límite de nuestras pruebas son las pruebas de comportamiento, definidas en conjunto con usuarios o clientes del sistema, pero que ejercitan una capa de la aplicación que se encuentre inmediatamente por debajo de la lógica de interacción y de la interfaz de usuario. Es la capa que hemos denominado como capa de servicios, que es el nombre más habitual en la literatura [Fowler 2003, Buschmann 1996].

5.1.7 Limitaciones

El enfoque que proponemos en esta tesis para realizar una refactorización segura tiene algunos supuestos que lo hacen impracticable en determinadas situaciones. Por ejemplo:

- No se cuenta con pruebas técnicas automatizadas.

[43] Hay algunas situaciones particulares, sin embargo. Por ejemplo, en Java, existen excepciones que no se chequean en tiempo de compilación, como ocurre con todas las instancias de clases derivadas de *java.lang.RuntimeException*.

- Las pruebas técnicas automatizadas no ofrecen suficiente cobertura para asegurar el comportamiento.
- No se cuenta con pruebas de aceptación automatizadas.
- Las pruebas de aceptación sólo cubren una parte del código de la aplicación.

Estos escenarios recién enumerados no son todos igualmente invalidantes. Como veremos enseguida, todos ellos admiten que se aplique el procedimiento, aunque lleve un poco de trabajo adicional.

En primer lugar, si no se cuenta con pruebas técnicas automatizadas, cualquier refactorización es difícil de realizar de manera segura. La literatura fundacional del refactoring como práctica ágil ni siquiera se plantea esta cuestión.
La mejor solución en este caso es crear pruebas técnicas automatizadas antes de comenzar la refactorización, al menos aquéllas que sirvan para cubrir el código a modificar. Esto será más sencillo en las refactorizaciones pequeñas que en aquellas de gran envergadura. Usando los mismos principios del enfoque que presenta esta tesis, se puede utilizar al propio código, que se presume está funcionando bien, como garantía de la corrección de las pruebas que se van a escribir. Si además se cuenta con pruebas de aceptación, tanto mejor, pues se podrán usar éstas como un reaseguro.
Una solución alternativa es usar pruebas de aceptación directamente. El problema de este enfoque, como ya explicamos más arriba, que no recomendamos a menos que el anterior sea impracticable, es doble. Por un lado, hay refactorizaciones que se pueden descomponer en otras más pequeñas, y no vale la pena usar pruebas de aceptación cuando alcanza con las unitarias o de integración. Por otro, la granularidad de las pruebas de aceptación es mucho mayor y se hace mucho más difícil y lento corregir problemas a medida que avanzamos en la refactorización. Además, si las pruebas de aceptación no se encuentran automatizadas (algo esperable si no hay pruebas técnicas automatizadas), no hay una manera fiable de establecer la cobertura, tema al que pasaremos enseguida.

El segundo de los escenarios problemáticos sería que las pruebas técnicas estuvieran disponibles, pero no cubrieran todo el código a refactorizar.
En realidad, la solución no presenta grandes diferencias con el caso anterior. Bastaría con agregar las pruebas que terminen de cubrir el código en cuestión. Probablemente, una ventaja frente a la situación anterior sea que parte del código ya está cubierto y que el sistema fue realizado con las pruebas automatizadas en mente, lo cual va a facilitar mucho la tarea.

En el tercer caso, nos hemos propuesto analizar qué pasaría si no se contase con pruebas de aceptación automatizadas. Lamentablemente, este escenario no es infrecuente. En efecto, hace ya una década que se insiste en la importancia de automatizar las pruebas técnicas, pero la automatización de las pruebas de aceptación es una práctica más reciente y mucho menos difundida en la industria. 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 aceptación 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[44].
Sin embargo, toda aplicación desarrollada siguiendo una metodología más o menos profesional debería contar con casos de prueba de aceptación, aunque estos no estén automatizados. Estos casos de prueba se podrían correr como pruebas de aceptación. El inconveniente que permanece es que, al no estar automatizados, no es sencillo garantizar la cobertura, ante lo cual tenemos dos opciones: o se corren todos los casos de prueba de la aplicación o se escriben pruebas automatizadas en base a los casos de prueba. Por supuesto, si no hay casos de prueba desarrollados, ninguna de estas soluciones es posible, pero allí el problema excede en mucho al tema de la corrección del refactoring.

Finalmente, la última consideración que hemos hecho es la falta de cobertura de las pruebas de aceptación.
Las soluciones a esta cuestión son muy parecidas a las del caso anterior. Si lo que falta cubrir es poco, lo más sencillo es tratar de escribir pruebas de aceptación automáticas que cubran lo que falta.
Si, en cambio, la cantidad de pruebas de aceptación automáticas a escribir es muy grande, o bien no se cuenta con un cliente con el cual validarlas, se puede trabajar con los casos de prueba de aceptación que se hayan desarrollado durante la construcción de la aplicación, aunque será difícil asegurar la cobertura. Probablemente una refactorización pequeña no amerite este esfuerzo, pero en casos de mayor envergadura no sería un costo tan destacado.

[44] De esta manera, supondría más una verificación de condiciones de aceptación con el cliente, que no eran explícitas en la versión anterior al cambio.

5.2 El método aplicado

5.2.1 Refactorizaciones que no afectan a las pruebas unitarias

El caso más sencillo de refactoring se da cuando las pruebas unitarias alcanzan para verificar su corrección. Por lo mismo, es el primer método que se intenta para cada refactorización. Además, en la literatura sobre refactoring es el caso canónico y el más difundido.

Básicamente, hay que cumplir con los siguientes pasos (el diagrama de actividades de la figura 5.4 los muestra en forma gráfica):

- Cambiar el código para mejorar su calidad.
- Compilar las pruebas unitarias.
- Si las pruebas hubiesen dejado de compilar, debemos cambiar de procedimiento, pasando al refactoring asegurado por pruebas de integración (ver próximo título).
- Correr las pruebas unitarias para verificar que siguen pasando.
- Si las pruebas fallasen, debemos cambiar de procedimiento, pasando al refactoring asegurado por pruebas de integración (ver próximo título).
- Si las pruebas unitarias pasan, la refactorización terminó exitosamente.

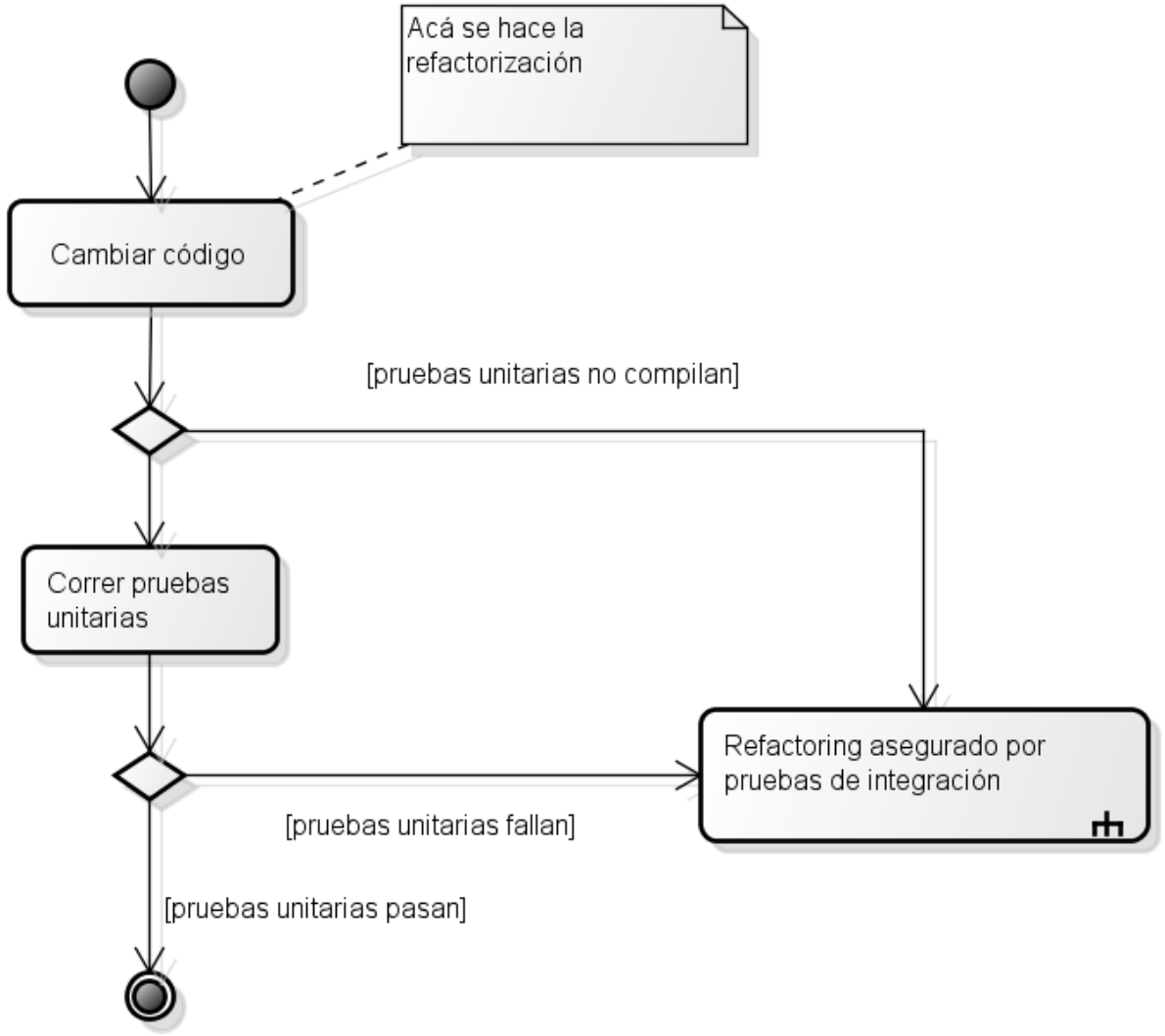


Figura 5.4: refactoring asegurado por pruebas unitarias

5.2.2 Refactorizaciones que rompen las pruebas unitarias, pero no a las de integración

Cuando las pruebas unitarias dejan de funcionar luego de una refactorización, ya sea porque no compilan más o porque fallan, no nos sirven como red de contención que asegure la preservación del comportamiento. Sin embargo, existen otras pruebas técnicas que tal vez sí nos sirvan: las que prueban a los clientes de la clase que estamos modificando, las mismas que hemos denominado pruebas de integración.

Muy probablemente, junto con las pruebas unitarias hayan dejado de compilar o fallen también las clases clientes, por ejemplo por cambios que hayamos introducido en los protocolos de nuestros métodos, que impidan que las clases clientes se sigan comunicando

con dichos métodos usando la firma que usaban hasta el cambio. Sin embargo, dado que vamos a trabajar sobre las pruebas de esos mismos clientes, esto no debería preocuparnos, ya que si logramos que estas pruebas funcionen, será porque los clientes habrán sido corregidos durante la refactorización.

En este caso, los pasos a realizar son (el diagrama de actividades de la figura 5.5 los muestra en forma gráfica):

- Utilizando las clases clientes (podemos observar qué clases del sistema dejaron de compilar o fallaban al hacer la refactorización), observar si las pruebas unitarias de esas clases clientes cubren el código que se quiere refactorizar. Estas pruebas son candidatas a ser las pruebas de integración que nos aseguren la corrección de la refactorización en curso.
- Si no hay pruebas de los clientes que cubran en su totalidad el código a refactorizar, no vamos a poder usarlas como garantía de la preservación del comportamiento. Por lo tanto, abandonamos el procedimiento y pasamos al refactoring asegurado por pruebas de comportamiento (ver próximo título).
- Si hay al menos una prueba o conjunto de pruebas de los clientes 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 las clases clientes y las pruebas de integración.
- Si las pruebas hubiesen dejado de compilar, debemos cambiar de procedimiento, pasando al refactoring asegurado por pruebas de comportamiento (ver próximo título).
- Correr las pruebas de integración para verificar que siguen pasando.
- Si las pruebas fallasen, debemos cambiar de procedimiento, pasando al refactoring asegurado por pruebas de comportamiento (ver próximo título).
- Si las pruebas de integración corren sin fallar, podremos concluir que la refactorización es exitosa, aunque haya pruebas unitarias que sigan fallando, que son las mismas que hicieron que abandonásemos el refactoring asegurado por pruebas unitarias.
- Corregir las pruebas unitarias que habían fallado al comienzo. Como medio de aseguramiento de la preservación del comportamiento de estas pruebas, usar la clase que se ha cambiado, ya que sabemos que funciona bien gracias a las pruebas de integración que así lo garantizan. Ahora sí la refactorización estaría completa y las pruebas todas funcionando nuevamente.

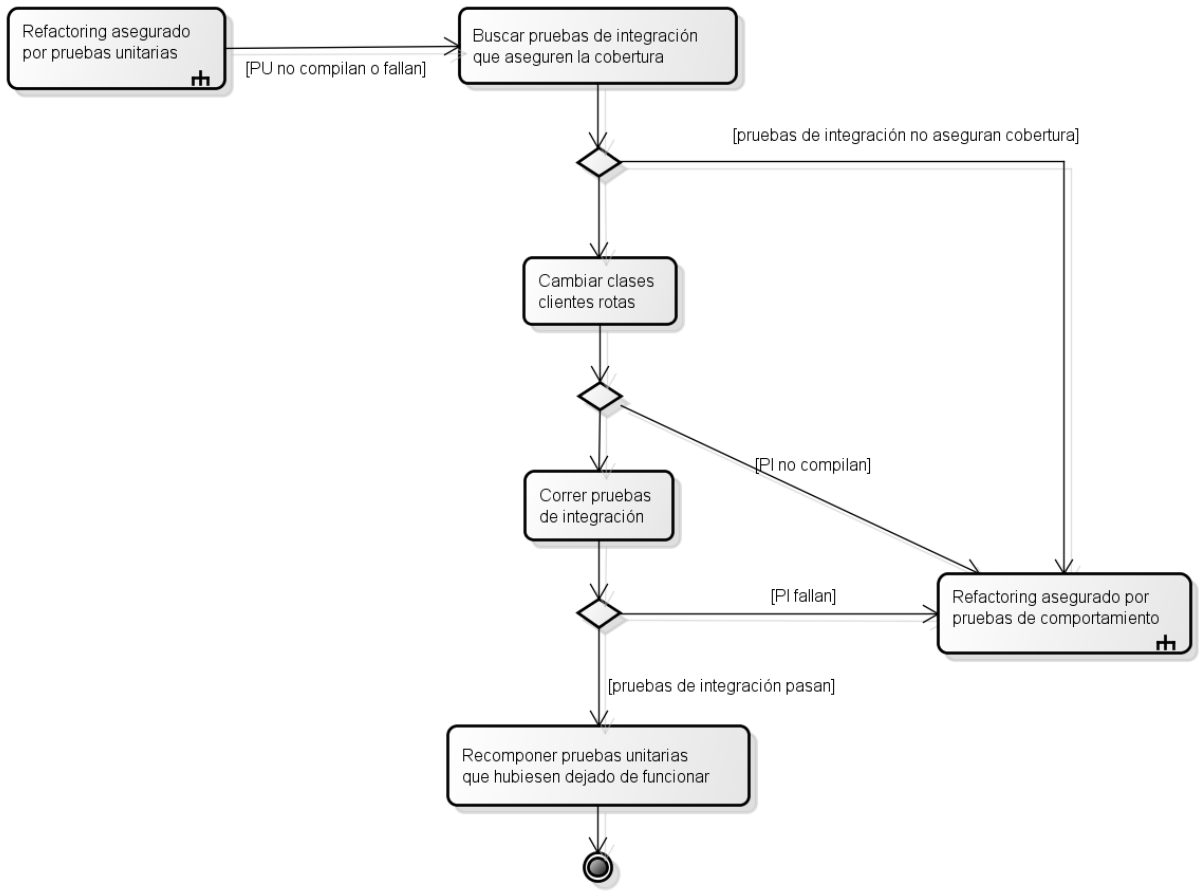


Figura 5.5: refactoring asegurado por pruebas de integración

5.2.3 Refactorizaciones que rompen las pruebas de integración, pero no a las de comportamiento

Cuando las pruebas unitarias dejan de funcionar luego de una refactorización, y tampoco nos sirven 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.

Los pasos a realizar en este caso son (el diagrama de actividades de la figura 5.6 los muestra en forma gráfica):

- Buscar entre las pruebas de comportamiento aquellas que cubran el código que se quiere refactorizar. Nos puede servir como guía usar las pruebas de las clases de servicios que hayan dejado de compilar o hayan fallado luego del cambio.

- 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. En este caso, no vamos a poder asegurar la corrección del refactoring. 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. Esto puede implicar tanto a las clases de dominio como a clases de servicios.
- Compilar todo el sistema, incluyendo las clases de dominio y de servicios, más las 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. De todos modos, podría haber algún caso en que esta falla se deba a un cambio menor en el protocolo de los servicios: aun en ese caso habría que proceder con cuidado y sólo cambiar las pruebas si un usuario o cliente acuerda en ello.
- Correr las pruebas de comportamiento para verificar que siguen pasando.
- Si las pruebas fallasen, la refactorización falló. Estamos sin duda ante un cambio de comportamiento, lo cual viola la definición de refactoring. En este caso, debe deshacerse la refactorización y hay que volver al código como estaba antes de la misma.
- 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 mismas que hicieron que abandonásemos el refactoring asegurado por pruebas de integración.
- 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 estas pruebas, 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.

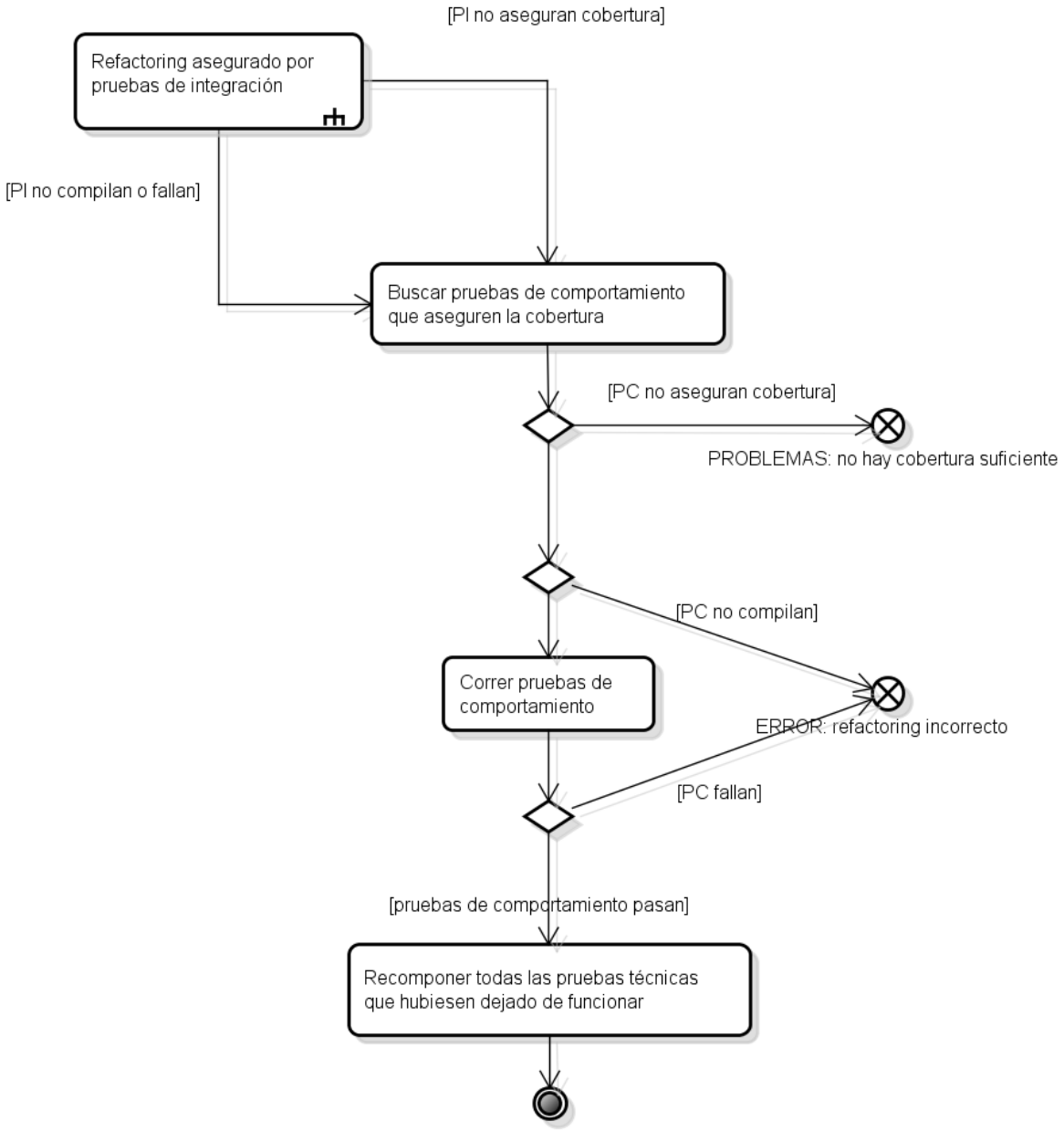


Figura 5.6: refactoring asegurado por pruebas de comportamiento

---

En este capítulo hemos desarrollado el enfoque metodológico que da sustento a esta tesis, el **refactoring asegurado por niveles de pruebas**. El mismo consiste en ir excluyendo pruebas que fallan y cambiándolas por otras, por niveles. Cuando las pruebas unitarias no nos sirven, utilizamos las de integración. Cuando estas últimas dejan de ser útiles, pasamos a las pruebas

de aceptación. Si fallasen las pruebas de aceptación, concluimos que la refactorización no se puede realizar, ya que son precisamente las pruebas de aceptación las que funcionan de invariantes en cuanto al comportamiento observable.
Por lo tanto, si dejan de funcionar las pruebas de aceptación está claro que no estamos ante una refactorización, pues no se cumple con su definición. Tal vez se trate de un cambio funcional, que habremos confundido con una refactorización. Y si bien resulta muy interesante analizar el impacto de cambios funcionales sobre las pruebas, no es el objetivo de esta tesis. En el capítulo 7 volveremos brevemente sobre este tema.
En el próximo capítulo validaremos el método presentado con una prueba de concepto sobre un caso de estudio.

6 Herramienta de cobertura múltiple y su aplicación en un caso de estudio

El objetivo de este capítulo es validar el enfoque metodológico presentado en el capítulo anterior. Lo haremos con un caso de estudio simple, aunque desarrollado con las mejores prácticas de diseño.
También nos servirá para presentar la herramienta *Multilayer Coverage*, desarrollada para esta tesis[45], que indica cuáles pruebas cubren cierta porción de código – el código a refactorizar – en forma simultánea.
Por razones prácticas, se usará una aplicación existente y pequeña, aunque le introdujimos algunos cambios importantes para poder seguir las recomendaciones metodológicas y arquitectónicas planteadas en el capítulo anterior.

6.1 El caso de estudio

6.1.1 Descripción funcional de la aplicación

La aplicación que se tomó como caso de estudio es un juego de "Ta-Te-Ti", también llamado "Tres en línea". Se trata de un juego en el que dos jugadores van ocupando alternativamente las celdas de un tablero de 3 filas y 3 columnas, debiendo conseguir formar una línea de tres celdas ocupadas por el mismo jugador: el jugador que lo logra primero, gana el juego.

La aplicación tomó la interfaz de usuario de otra solución ya desarrollada.

Las sucesivas versiones de la aplicación, antes de comenzar con la validación del enfoque metodológico, fueron:

Versión 1: Ta-Te-Ti para dos jugadores humanos

- Iteración 1.1: permitir que se coloquen cruces (X) y círculos (O) en un tablero de 3 por 3, manteniendo además la noción de que las cruces corresponden a un jugador y los círculos a otro. La salida de esta iteración no da valor al usuario, pues ni siquiera tiene

[45] El desarrollo de Multilayer Coverage fue motivado por esta tesis. El autor de la tesis dirigió su desarrollo, mientras que Andrés Lange la programó, en el marco de su Trabajo Profesional de Ingeniería Informática de la Universidad de Buenos Aires.

interfaz de usuario. Todas las pruebas de comportamiento se realizaron mediante una herramienta de prueba automatizada sobre la capa de servicios.

- Iteración 1.2: chequear los turnos de cada jugador, no permitiendo que se coloquen dos cruces ni dos círculos consecutivos, forzando además el comienzo con una cruz (X). Sigue sin haber interacción con un usuario a través de la interfaz.
- Iteración 1.3: que la aplicación determine el momento en que un usuario logra ganar el juego.
- Iteración 1.4: desarrollo de la comunicación entre la interfaz gráfica[46] y el código ya escrito. Esta es la primera versión con valor para un usuario, pues en la misma se puede ver qué sucede en el juego a través de su interfaz.
- Iteración 1.5: desarrollo de un repositorio persistente, almacenando, para cada jugador, la cantidad de partidas jugadas, ganadas, empatadas y perdidas. Lo realizado en esta iteración no tiene todavía mayor valor para el usuario, que no puede hacer consultas ni reportes. Se agregó solamente para trabajar sobre una aplicación más realista en términos de arquitectura.

Versión 2: permitir que un jugador juegue contra la computadora

- Iteración 2.1: se permite elegir jugar contra la computadora, otorgándole a ésta una inteligencia nula, consistente en colocar su círculo (O) en la primera celda que encuentre libre cada vez. Comienza siempre el jugador humano, que usa la cruz (X). Si bien es una versión operativa, no sirve a los fines prácticos.
- Iteración 2.2: que la computadora gane el juego cuando sólo le falta una jugada para colocar tres círculos en línea, colocando el tercero.
- Iteración 2.3: que la computadora detecte cuando el usuario esté a una jugada de ganar el juego, bloqueando la celda que le falte a éste para lograrlo.

Esta versión 2 es, no sólo operativa, sino que sirve como base para mejoras futuras. Notemos, no obstante, que la lógica que sigue la máquina es aún muy primitiva.

6.1.2 Posibles mejoras funcionales

Como dijimos al presentar la práctica de refactoring, una refactorización se hace para mejorar la calidad del código de una aplicación – existente o en desarrollo – con vistas a

[46] La interfaz de usuario fue tomada de otra solución y se la adaptó para que utilizara el modelo que se desarrolló para esta tesis.

modificaciones futuras. De allí que nos interese analizar qué mejoras se le pueden hacer a la aplicación en futuras versiones.

Hay una serie de mejoras que se le pueden hacer a esta aplicación para que resulte más interesante:

- Que cuando juega un humano contra la computadora, el primer movimiento pueda corresponder también a la máquina.
- Dar mayor inteligencia a la computadora: en la versión actual, sólo hace Ta-Te-Ti cuando puede, y evita el Ta-Te-Ti del oponente, pero en todos los demás casos juega en la primera celda que encuentra disponible.
- Permitir varias partidas sucesivas entre un mismo par de jugadores.
- Permitir consultas y emitir reportes de partidas ganadas, empatadas y perdidas de cada jugador que esté almacenado en el repositorio persistente.

También hay evoluciones posibles a más largo plazo:

- Permitir más de un usuario en forma concurrente.
- Permitir jugar desde dispositivos móviles.
- Usar esta aplicación para derivar un framework para aplicaciones de tablero sin movimiento, empezando por las más parecidas, como el "Cuatro en línea", y siguiendo luego hacia aplicaciones con piezas que permitan movimientos de las mismas (damas y ajedrez, por ejemplo).

6.1.3 Decisiones de lenguaje, plataforma y herramientas

Como lenguaje de programación elegimos Java, en particular la versión 6 de la plataforma Java SE[47]. La elección del lenguaje tuvo que ver con que es un lenguaje orientado a objetos muy difundido y el que cuenta con más herramientas de desarrollo, refactoring y cobertura, lo cual nos permitió concentrarnos en los aspectos centrales de la tesis, sin estar buscando herramientas de escasa difusión.
Lo que hay que tener en cuenta es que Java es un lenguaje con comprobación estática de tipos. No obstante, se podrían haber utilizado lenguajes de comprobación dinámica, como Python y Smalltalk, usando el mismo enfoque metodológico, aunque su uso en el caso de estudio habría sido más laborioso.

[47] http://docs.oracle.com/javase/6/docs/

El framework de pruebas automatizadas por el que se optó fue JUnit, tanto para las pruebas técnicas como para las de aceptación.
En el caso de las pruebas técnicas pesó el hecho de que JUnit es el estándar de facto para Java, aun cuando hay herramientas más sofisticadas, como TestNG[48].
La decisión en cuanto a las pruebas de aceptación es más polémica. En proyectos reales de la industria se hace especial énfasis en que las pruebas de aceptación deben poder ser leídas por usuarios legos, e incluso los clientes deberían colaborar en su elaboración. Por ello es que existen numerosos frameworks que permiten automatizar pruebas de comportamiento en base a documentos legibles por personas no técnicas: en Java destacan FIT[49], FitNesse[50], JBehave[51], Concordion[52], etc. Todas estas herramientas sirven para vincular las pruebas de aceptación de usuarios con código JUnit, y algunas de ellas incluso permiten especificar escenarios en castellano u otros idiomas distintos del inglés. En nuestro caso, dado que sólo estamos pretendiendo validar un método, la elección de la herramienta fue realizada con un criterio más pragmático: contar con un único framework sencillo, liviano, potente y suficientemente difundido entre lectores técnicos. Por eso se utilizó JUnit, sabiendo que si en una situación de un proyecto del mundo real se desea usar un framework específico, se puede seguir el mismo procedimiento, sobre todo teniendo en cuenta que todos ellos terminan mapeando con pruebas JUnit.

El entorno de desarrollo seleccionado fue Eclipse, nuevamente por ser el más difundido entre los desarrolladores de Java, por ser gratuito y de código abierto, y por contar con *plugins* para JUnit y para varias herramientas de cobertura.
El uso de Eclipse nos permitió tomar algunas decisiones que facilitaron el trabajo con el caso de estudio, como definir proyectos distintos para el código de la aplicación, para las pruebas técnicas y para las pruebas de aceptación. A pesar de haber tomado estas decisiones que nos facilitaron la tarea, no mencionaremos estos usos más adelante, ya que el enfoque de esta tesis es independiente del entorno de desarrollo que utilicemos.

[48] http://testng.org/

[49] http://fit.c2.com/

[50] http://fitnesse.org/

[51] http://jbehave.org/

[52] http://www.concordion.org/

Como herramienta de cobertura se usó EclEmma[53], una extensión para Eclipse del motor de cobertura JaCoCo[54]. Se trata de una herramienta gratuita, de código abierto, muy sencilla de utilizar, que brinda informes de cobertura de muy sencilla interpretación[55].
El hecho de ser de código abierto permitió desarrollar la herramienta *Multilayer Coverage* como una extensión a EclEmma, y de ese modo obtuvimos cierto grado de automatización en el análisis de cobertura entre pruebas.

En cuanto al repositorio de objetos persistentes, se recurrió a la solución más sencilla: el uso del mecanismo de serialización nativo en la plataforma Java. Ello se debió a que el enfoque metodológico que presentamos en esta tesis no depende de la táctica de persistencia, con lo cual no tiene sentido desarrollar algo más sofisticado. Incluso se pudo haber hecho con una base de datos en memoria, o siguiendo el patrón Fake Object [Meszaros 2007][56]. Al no usar una base de datos, estamos dejando bien establecido que el enfoque de la tesis termina con el comportamiento, entendiendo también que en las bases de datos bien diseñadas no debería haber comportamiento.
Esto no impide, en un entorno productivo, reemplazar este mecanismo por el uso de un repositorio más sofisticado, desde bases de datos relacionales, bases NoSQL o cualquier otro que se desee.

Más allá de todo lo dicho en este punto, el enfoque metodológico de la tesis es independiente de las tecnologías usadas en este caso de estudio. Por ello, no haremos referencia a las mismas, salvo cuando sea estrictamente necesario, y siempre prescindiendo de ellas al referirnos al enfoque metodológico.

6.1.4 Arquitectura de la aplicación

Como decíamos, la sencillez de la aplicación, que resulta de la simplicidad de sus requerimientos técnicos (aplicación monousuario de escritorio), no impidió que el desarrollo

[53] http://www.eclemma.org/. Debe su nombre a que originalmente corría sobre el motor de cobertura Emma (http://emma.sourceforge.net/).

[54] http://www.eclemma.org/jacoco/

[55] Realiza análisis de cobertura de sentencias a nivel de código intermedio de Java (*bytecode*), además de indicar la cobertura parcial o total de ramas y condiciones.

[56] Éste reemplaza una base de datos relacional por un conjunto de colecciones asociativas, como el *HashMap* de Java.

se hiciese siguiendo las mejores prácticas de diseño para aplicaciones más complejas de envergadura industrial.

El diagrama de paquetes de la aplicación (figura 6.1) muestra la arquitectura de capas utilizada y el muy bajo acoplamiento entre las mismas. En el mismo, cada capa se representó como un paquete de UML.

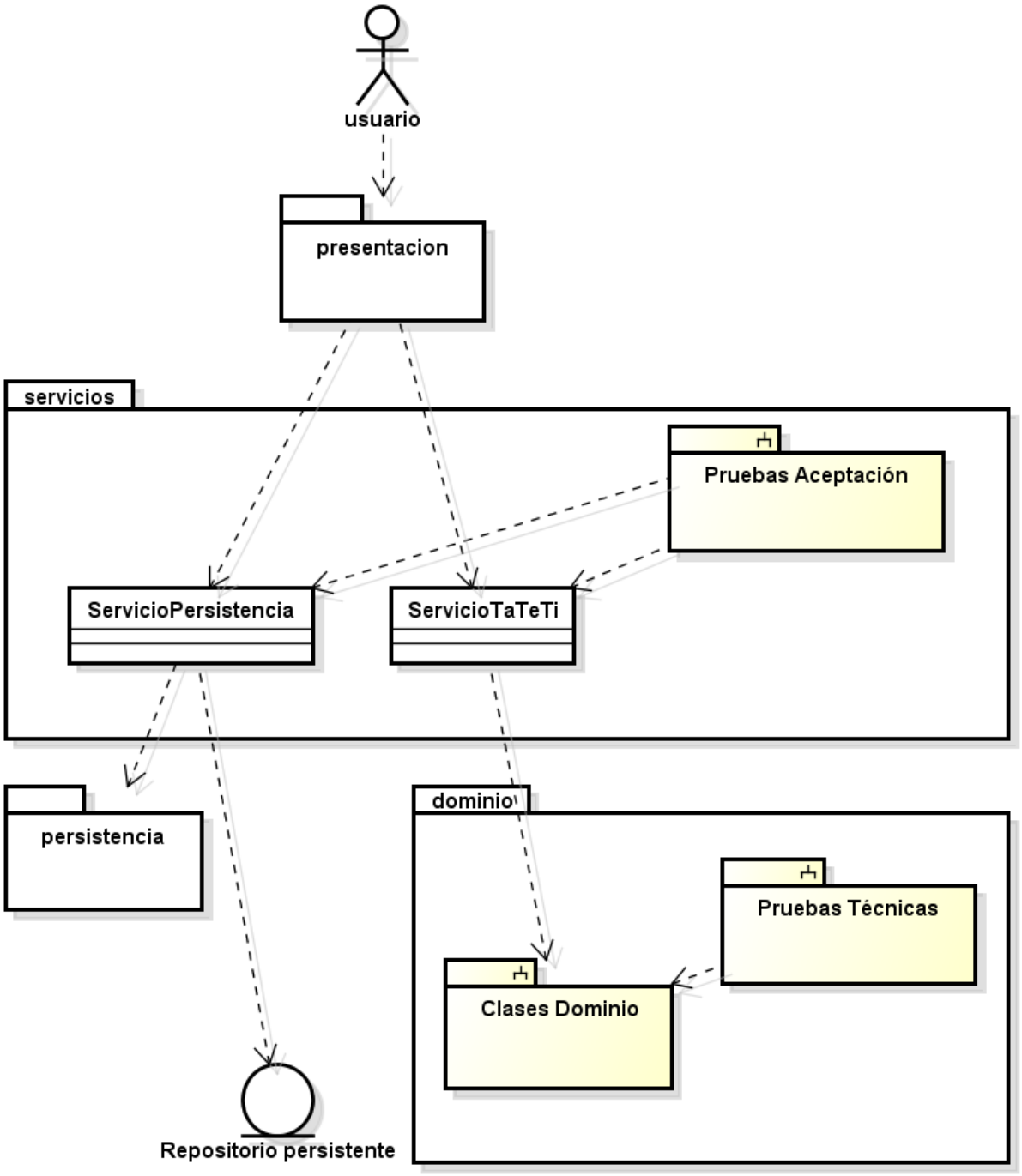


Figura 6.1: arquitectura del caso de estudio

Nótese incluso la ubicación atípica de la capa de persistencia, que ni siquiera interactúa directamente con la capa de dominio ni con el propio repositorio persistente. Sólo contiene clases de apoyo para lograr la persistencia, pero la propia persistencia se maneja desde la capa de servicios.

Esta arquitectura tiene las siguientes ventajas:

- La capa de dominio es totalmente orientada a objetos, basada en objetos de dominio con su comportamiento, y no tiene ningún elemento que dependa de la persistencia ni de la interfaz de usuario. Todo el comportamiento de la aplicación está en la capa de dominio, por lo que las capas de interfaz de usuario y de persistencia son simples auxiliares de aquélla, sin comportamiento.
- La capa de servicios, como se ve en el diagrama, está simplemente compuesta por unas pocas clases sin comportamiento, que sólo hace de Fachada [Gamma 1994] de las capas de persistencia y de dominio, delegando en ellas todo el comportamiento que necesita, en base a solicitudes recibidas desde la capa de presentación.
- La capa de presentación no conoce a los objetos de dominio ni a los que se usan para la persistencia, y sólo se comunica con los servicios usando objetos que tienen sentido en el ámbito de los actores de los casos de uso.
- Las pruebas de aceptación sólo interactúan con clases en la capa de servicios, por lo que están desarrolladas siguiendo la lógica de la capa de presentación, aunque sin utilizar la propia capa de presentación. En nuestra taxonomía de pruebas esto las convierte en lo que hemos llamado pruebas de comportamiento.
- Las pruebas técnicas sólo interactúan con los objetos de la capa de dominio, para determinar y validar el diseño técnico a nivel de clases de dominio.

En nuestro diagrama, las pruebas no figuran en paquetes aparte, porque se siguió la práctica de ubicarlas en el mismo paquete en el cual están las clases que deben probar, de modo tal de que tengan un mayor acceso a ciertas propiedades de las mismas[57].

[57] En Java, las clases y los métodos pueden tener visibilidad de paquete, en cuyo caso son visibles para otras clases del mismo paquete. Por eso, colocando las pruebas técnicas dentro del mismo paquete que las clases de dominio, les damos acceso a algunas características que necesitamos para probarlas, sin necesidad de hacer públicas dischas características. Lo mismo hacemos al colocar a las pruebas de aceptación en el mismo paquete que las clases de la capa de servicios. Para separar en algún lugar ambas cuestiones, trabajamos con proyectos distintos de Eclipse, para clases de la aplicación, pruebas técnicas y pruebas de aceptación. En el diagrama de la figura 6.1, las partes de los proyectos dentro de paquetes se representaron como subsistemas.

6.1.5 Diseño de clases de la capa de dominio

Respecto de la capa de dominio, las clases de la misma se muestran en el diagrama de la figura 6.2.

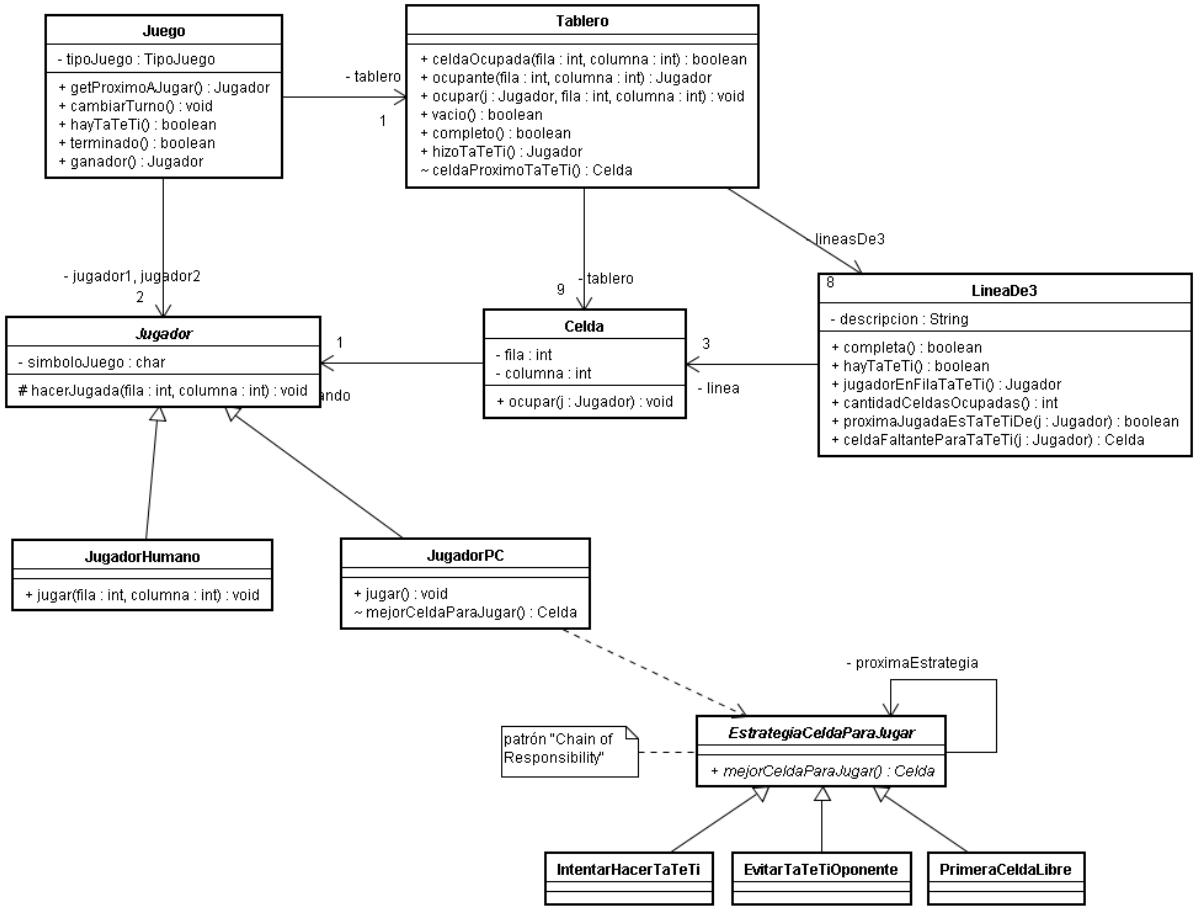


Figura 6.2: dominio del caso de estudio

El diseño de las clases de dominio se hizo de forma tal de que fueran lo más ricas posibles en cuanto a comportamiento. Prácticamente no tienen métodos que cambien el estado de los objetos ni devuelvan propiedades de los mismos, ofreciendo preponderantemente métodos de comportamiento.

Para comprender mejor las responsabilidades de cada clase, se muestran a continuación algunos diagramas de secuencia.

El de la figura 6.3 es el diagrama que muestra la lógica de obtención del ganador de una partida[58].

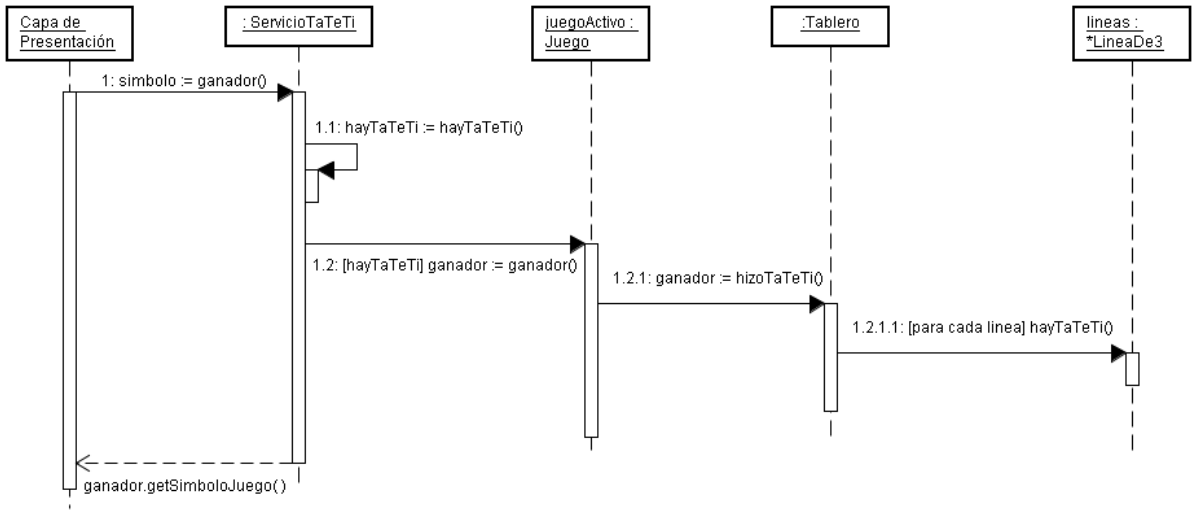
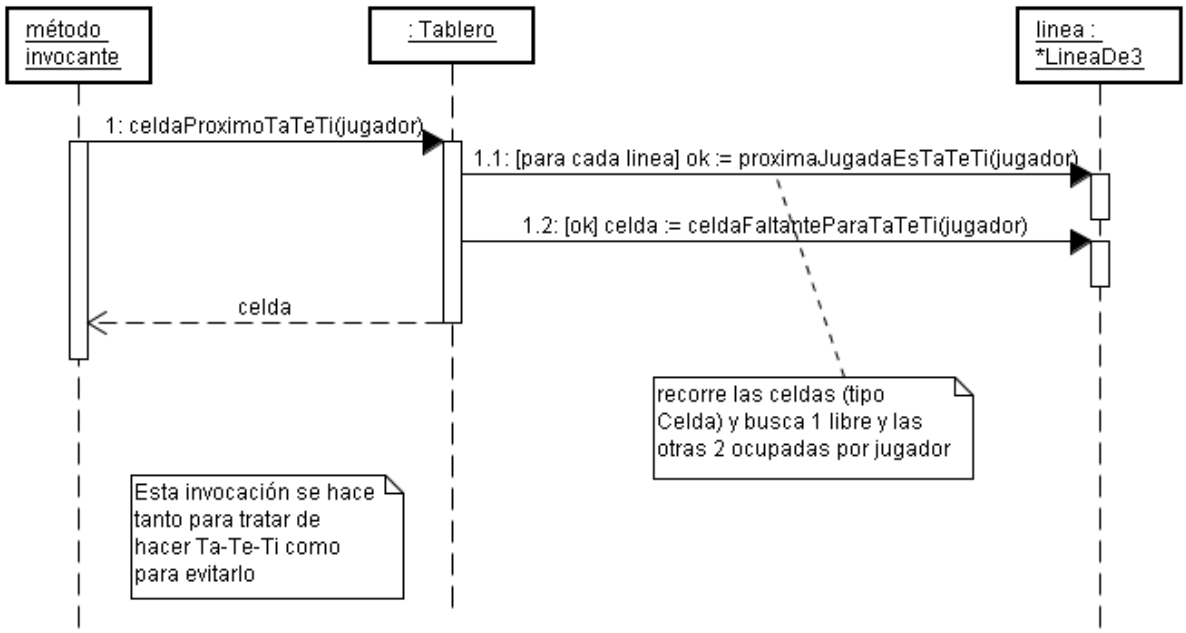


Figura 6.3: secuencia de la obtención del ganador de una partida

El diagrama de secuencia de la figura 6.4 representa el escenario para determinar si hay alguna celda en la que en la próxima jugada se pueda producir Ta-Te-Ti:



[58] Por cuestiones de simplicidad, el diagrama se hizo hasta la determinación de la existencia de tres celdas con el mismo ocupante en la misma línea. A partir de allí, cada instancia de la clase *LineaDe3* delega en sus celdas (instancias de *Celda*) la determinación del ocupante.

Figura 6.4: secuencia de determinación de un posible Ta-Te-Ti en la jugada siguiente

6.1.6 Metodología adoptada en la construcción de la aplicación

El método que se siguió en la construcción de la aplicación fue el protocolo de ATDD, tomando de a una iteración cada vez, construyendo para la misma una clase de pruebas de aceptación, del tipo de las de comportamiento, para comprobar la capa de servicios.
Para hacer funcionar las pruebas de aceptación se fue construyendo la capa de servicios, haciendo que funcione una prueba cada vez. Al construir las clases de la capa de servicios, fue necesario desarrollar o modificar las clases de dominio, las cuales se construyeron siguiendo el protocolo de UTDD[59].

6.2 La herramienta Multilayer Coverage

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 planteamos 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. La misma se realizó como extensión a EclEmma.

Multilayer Coverage permite marcar un método que se quiere refactorizar, y nos muestra cuántas y cuáles son las clases de prueba que cubren ese método. Para ello, 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, como herramienta de apoyo que es, permite determinar cuáles son las pruebas específicas que cubren determinadas líneas. Adicionalmente, la herramienta provee información resumida sobre el grado de cobertura de un trozo de código.

[59] Para las definiciones de ATDD y UTDD ver capítulo 3.

Por ejemplo, la Figura 6.4 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.

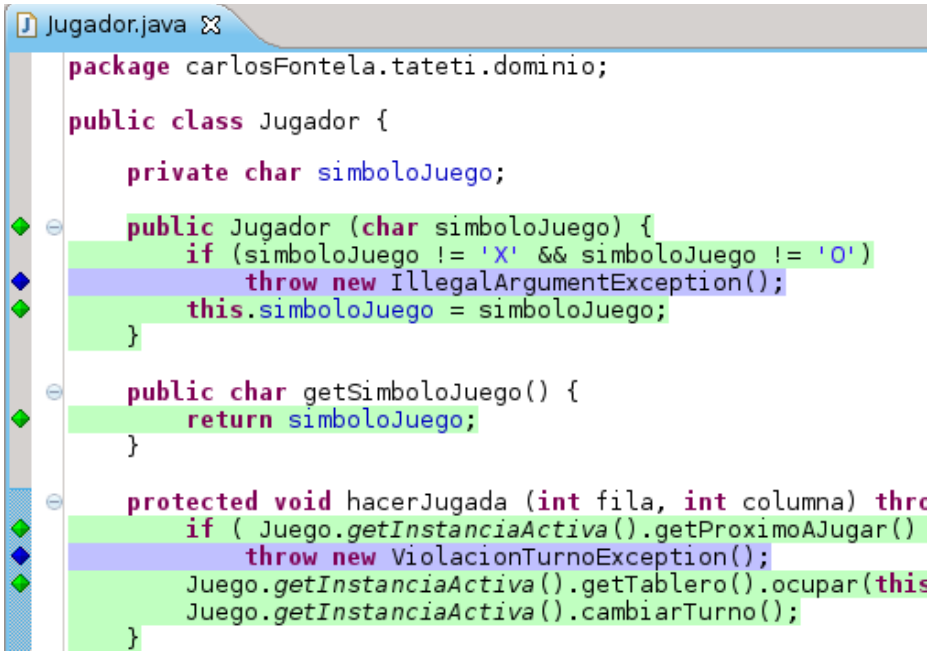


Figura 6.4: Grado de cobertura en Multilayer Coverage

Para comenzar entonces a refactorizar el método *hacerJugada*, necesitaríamos primeramente conocer cuáles son las pruebas que puntualmente cubren las líneas de ese método. Como dijimos anteriormente, Multilayer Coverage permite conocer cuáles son esas pruebas. 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 6.5.

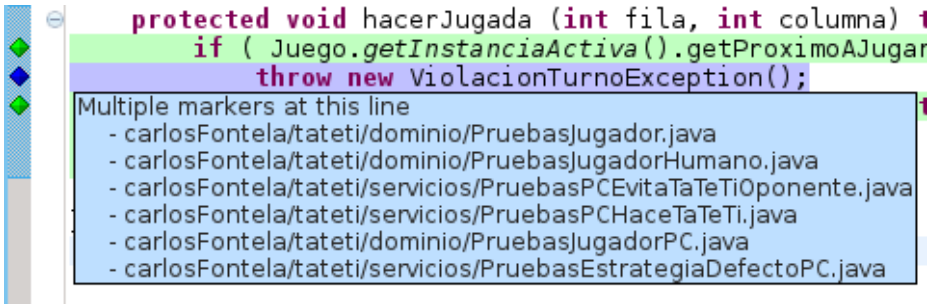


Figura 6.5: Clases que brindan cobertura en Multilayer Coverage

La Figura 6.6 muestra la información de resumen adicional que provee la herramienta. Allí vemos que hay métodos sin cobertura (valor 0.0), otros donde al menos hay una prueba que los recorre (valor 1.0), y así sucesivamente.

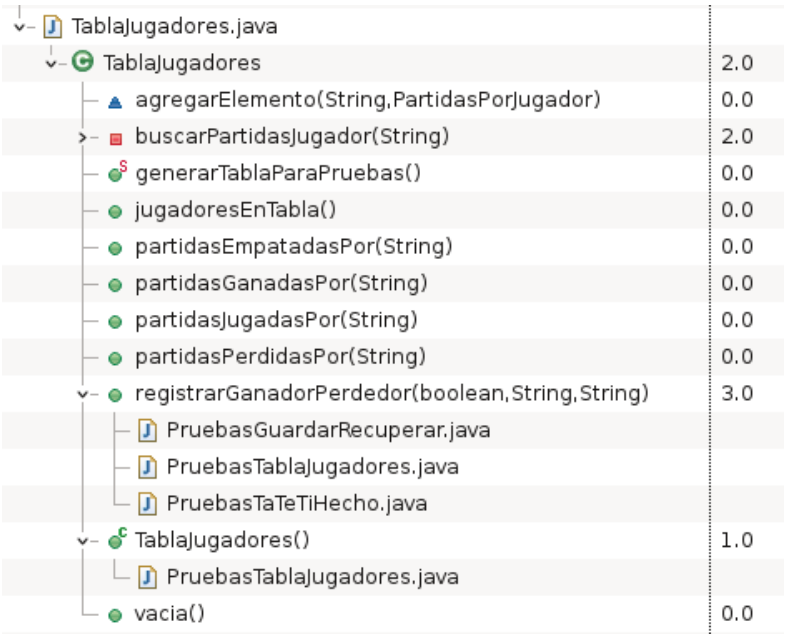


Figura 6.6: información de resumen de cobertura en Multilayer Coverage

En esta etapa, Multilayer Coverage evalúa solamente cobertura de sentencias [Cornett] 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.

6.3 Refactoring asegurado por niveles de pruebas en el caso de estudio

6.3.1 Realización de refactorizaciones automáticas

Estamos denominando *refactoring automático* a aquél que realiza el entorno de desarrollo con un mínimo de participación del desarrollador. En estos casos casi no hay que realizar ninguna comprobación[60]: el entorno se ocupa de buscar todo el código afectado por la refactorización y cambiarlo en forma simultánea. Por ello, el código de todas las clases clientes de la que se está modificando, incluyendo las de pruebas, sigue compilando luego del cambio, y se presume también que las pruebas seguirán corriendo sin inconvenientes.
Por todo esto, en general no es necesario hacer ningún análisis de cobertura. Tampoco diríamos que aplicamos el método presentado en este trabajo, salvo en su forma más simplificada.

Un buen ejemplo de este tipo de refactorización es la que se realiza para cambiar nombres. En nuestro caso, hemos realizado dos sobre la aplicación. La primera consistió en cambiar el nombre de la variable local *mejor*, del método *mejorCeldaParaJugar* de la clase *JugadorPC*, que nos pareció más adecuado llamar *mejorCelda*.

El método era:

```
Celda mejorCeldaParaJugar() {
        Celda mejor = null;
        EstrategiaCeldaParaJugar cadenaDeResponsabilidad =
                new IntentarHacerTaTeTi(
                        new EvitarTaTaTiOponente( new PrimeraCeldaLibre(null) ) );

        do {
                if (cadenaDeResponsabilidad == null)
                        // no debería ocurrir: tablero lleno y juego terminado
                        break;
                mejor = cadenaDeResponsabilidad.mejorParaJugar();
                cadenaDeResponsabilidad =
                        cadenaDeResponsabilidad.getProximaEstrategia();
        }
```

[60] Una limitación a este escenario se da en los casos de interfaces publicadas y refactoring de código compartido, temas relacionados brevemente tratados en el capítulo 7.

```
            while (mejor == null);

            return mejor;
    }
```

Las pruebas unitarias que chequean este método están en la clase *PruebasJugadorPC*. Con EclEmma podemos verificar que las mismas cubren totalmente los usos de la variable local.

El método, luego del cambio, quedó así (en negrita las líneas que cambiaron):

```
    Celda mejorCeldaParaJugar() {
            Celda mejorCelda = null;
            EstrategiaCeldaParaJugar cadenaDeResponsabilidad =
                    new IntentarHacerTaTeTi( new EvitarTaTaTiOponente(
                                    new PrimeraCeldaLibre(null) ) );

            do {
                    if (cadenaDeResponsabilidad == null)
                            // no debería ocurrir: tablero lleno y juego terminado
                            break;
                    mejorCelda = cadenaDeResponsabilidad.mejorCeldaParaJugar();
                    cadenaDeResponsabilidad =
                            cadenaDeResponsabilidad.getProximaEstrategia();
            }
            while (mejorCelda == null);

            return mejorCelda;
    }
```

Al ser una variable local lo que cambió de nombre, el entorno de desarrollo no cambió nada en ninguna clase cliente ni en las pruebas. Luego del cambio se corrieron las pruebas unitarias contenidas en la clase *PruebasJugadorPC*, que pasaron todas correctamente.
Nótese que, al cubrir las pruebas unitarias totalmente al código que se modificó y al pasar estas sin problema luego de la refactorización, no es necesario correr pruebas de integración ni de comportamiento.

El segundo caso que encaramos fue un cambio de protocolo, no ya un nombre local. Nos propusimos cambiar el nombre del método *completa* de la clase *LineaDe3* por el más apropiado *estaCompleta*.

En este caso estamos modificando el nombre de un método con visibilidad de paquete. Por lo tanto, el entorno de desarrollo, al solicitarle el cambio, actualizó todos los llamados a ese método desde clases del mismo paquete. Entre ellas, las pruebas técnicas de la clase *PruebasLineaDe3* (pruebas unitarias), que cubrían totalmente al método en cuestión.
Usando nuestro método, deberíamos probar que las pruebas que nuestra herramienta de cobertura marcó cubriendo el método *completa* siguen funcionando cuando el nombre cambia a *estaCompleta*. En efecto, corridas después de la refactorización, pasaron sin problemas.
No es necesario analizar pruebas de integración ni de comportamiento, ya que con las pruebas de unidad que cubrían el código modificado obtuvimos resultados positivos.

Como se ve, en estos casos en que el entorno de desarrollo hace la refactorización en forma automática, el método propuesto en este trabajo, si bien es aplicable, sólo tiene sentido usarlo para corroborar que, como se esperaba, el entorno de desarrollo funciona bien. Eso mismo pasaría si se hubieran visto afectadas pruebas de integración, o incluso de comportamiento, ya que el entorno de desarrollo hubiese cambiado todas las referencias para que éstas siguieran siendo válidas.

6.3.2 Realización de refactorizaciones no automáticas que se verifican sólo con pruebas de unidad

En este ítem realizaremos algunas refactorizaciones que no se puedan realizar en forma automática, pero que, por referirse sólo a cuestiones de implementación internas de un método o clase, no deberían afectar más que a las pruebas unitarias.

Tomamos como caso el método *estaCompleta* de la clase *LineaDe3*, recientemente renombrado. Este método fue implementado así:

```
boolean estaCompleta ( ) {
        if (linea[0].getJugadorOcupando() == null || linea[1].getJugadorOcupando() == null
                || linea[2].getJugadorOcupando() == null)
            return false;
        else
            return true;
}
```

Esta implementación, preguntando por cada una de las celdas de una línea, funciona bien para el Ta-Te-Ti, pero no es fácilmente extensible para otros juegos similares, como el “Cuatro en

línea". Para que sea más fácil de generalizar luego, podemos cambiar la triple pregunta por un ciclo *for*.
Por ejemplo, esta solución parece más apropiada:

```
boolean estaCompleta ( ) {
        for (Celda celda : linea)
                if (celda.getJugadorOcupando() == null)
                        return false;
        return true;
}
```

Antes de realizar la refactorización debemos analizar el tema de la cobertura. Utilizando nuestra herramienta de cobertura, vemos que el método *estaCompleta* está cubierto sin problemas por las pruebas unitarias de la clase *PruebasLineaDe3*. Además, la cubren indirectamente otras pruebas (de integración y de comportamiento), que por el momento no nos preocupan.

Realizamos entonces la refactorización propuesta y compilamos todo el código. El mismo compila sin problemas, como es esperable, ya que no hemos cambiado nada del protocolo del método. Para verificar la preservación del comportamiento, corremos las pruebas unitarias de la clase *PruebasLineaDe3*, obteniendo resultados positivos.
Dado el resultado obtenido hasta aquí, la refactorización termina en este punto, sin necesidad de verificar pruebas de integración ni de comportamiento[61].

Lo que hicimos recién fue una refactorización sencilla. Pero el mismo procedimiento se puede aplicar a casos un poco más complejos. Por ejemplo, el código del método *proximaJugadaEsTaTeTiDe* en la clase *LineaDe3* es bastante poco elegante:

```
boolean proximaJugadaEsTaTeTiDe (Jugador jugador) {
        if ( cantidadCeldasOcupadas() == 2 ) {
                if ( ( linea[0].getJugadorOcupando() == jugador ) &&
                        ( linea[1].getJugadorOcupando() == jugador ) )
                                return true;
                if ( ( linea[0].getJugadorOcupando() == jugador ) &&
                        ( linea[2].getJugadorOcupando() == jugador ) )
                                return true;
```

[61] Notemos que esto sólo se puede afirmar por el nivel de cobertura existente.

```
                    if ( ( linea[1].getJugadorOcupando() == jugador ) &&
                            ( linea[2].getJugadorOcupando() == jugador ) )
                                return true;
            }
            return false;
    }
```

En primer lugar, el nombre del método es poco feliz, y podría reemplazarse por algo como *proximaJugadaPodriaSerTaTeTiDe*, lo cual se puede lograr con una simple refactorización automática de cambio de nombre.

Pero, más allá del nombre, quedaría mucho más claro si el método se reescribiera así:

```
    boolean proximaJugadaPodriaSerTaTeTiDe (Jugador jugador) {
            if ( cantidadCeldasOcupadas() == 2 && cantidadCeldasOcupadasPor(jugador) == 2)
                    return true;
            else
                    return false;
    }
```

En cuyo caso, habría que agregar un método más:

```
    private int cantidadCeldasOcupadasPor (Jugador jugador) {
            int cantidad = 0;
            for (Celda celda : linea)
                    if (celda.getJugadorOcupando() == jugador)
                            cantidad++;
            return cantidad;
    }
```

Antes de proceder a refactorizar, comprobamos la cobertura, viendo que las pruebas de la clase *PruebasLineaDe3* (unitarias) cubren totalmente el código del método *proximaJugadaPodriaSerTaTeTiDe*. Por lo tanto, refactorizamos, y la sola corrida de las pruebas unitarias nos garantiza la corrección.

Hasta este punto, no hemos realizado ninguna refactorización que precise de pruebas que no sean las de unidad. Por lo tanto, estamos usando el enfoque que se plantea como el habitual en

la literatura de refactoring. Recién a partir del próximo título comenzamos a usar los aspectos novedosos del **refactoring asegurado por niveles de pruebas**.

6.3.3 Realización de una refactorización que necesita de las pruebas de integración para su verificación

En este ítem realizaremos alguna refactorización que no pueda realizarse en forma automática y en la cual las pruebas unitarias van a dejar de funcionar, de forma tal que necesitaremos de las pruebas de integración para asegurar la corrección del refactoring.
Recién a partir de este nivel podemos decir que comenzamos a aplicar el enfoque metodológico de la tesis, saliéndonos del procedimiento tradicional basado en pruebas unitarias automatizadas.

Antes de comenzar, una observación. Este caso de insuficiencia de las pruebas unitarias y en el cual alcanza con comprobar la pruebas de integración, no es tan habitual, siendo que se cae a menudo en el procedimiento más complejo del próximo título, que precisa de las pruebas de comportamiento.

Vamos a tomar el caso del constructor de la clase *LineaDe3*:

```
LineaDe3 (Celda celda0, Celda celda1, Celda celda2, String descripcion) {
        if (celda0 == null || celda1 == null || celda2 == null || descripcion == null)
                throw new IllegalArgumentException();
        this.linea[0] = celda0;
        this.linea[1] = celda1;
        this.linea[2] = celda2;
        this.descripcion = descripcion;
}
```

Si bien la calidad del código no es mala, tiene dos problemas:

- Tiene cuatro parámetros, lo cual en esta situación puede ser demasiado, más aún teniendo en cuenta que los tres primeros se refieren a las celdas que incluye la línea, y el cuarto a la descripción de la misma. Esta cantidad y heterogeneidad de los parámetros hace que el constructor caiga en dos olores de los definidos por Fowler [Fowler 1999]: *Long Parameter List* y *Data Clumps*. Incluso Robert Martin [Martin 2008] menciona este escenario como problemático, al decir que hay que minimizar el número de parámetros, tratando de que a lo sumo sean dos. Ver anexo A.

- No es sencillo generalizar a otros juegos muy similares, como el "Cuatro en línea", ya que, manteniendo esta firma de constructor (más allá del nombre de la clase), debería tomar un parámetro más.

La versión del constructor que se muestra a continuación, en cambio, es menos cuestionable desde esos puntos de vista (los cambios figuran en negrita):

```
LineaDe3 ( Celda[ ] linea, String descripcion ) {
        if ( linea == null || descripcion == null )
                throw new IllegalArgumentException();
        for (Celda celda : linea)
                if ( celda == null )
                        throw new IllegalArgumentException();
        this.linea = linea;
        this.descripcion = descripcion;
}
```

Aquí hemos aplicado una variante de la refactorización que Fowler [Fowler 1999] denomina *Introduce Parameter Object*[62]. Con ello mantenemos una lista de parámetros menor, más consistente, y a su vez, más sencilla de generalizar a otros juegos similares.

Lo primero que hacemos, siguiendo el enfoque que presenta la tesis, es realizar el cambio. Enseguida el IDE nos advierte que las pruebas unitarias de la clase *LineaDe3*, todas ellas en la clase *PruebasLineaDe3* han dejado de compilar, pues todas usan el constructor anterior, cuya firma es diferente antes y después de la refactorización. Por lo tanto, el procedimiento de usar las pruebas unitarias para verificar la corrección del refactoring no es válido, pues las propias pruebas unitarias se han roto. Incluso la clase *Tablero* tiene ahora errores de compilación, así que vamos a tener que cambiarla.

Lo que tenemos que hacer, entonces, es analizar la cobertura con nuestra herramienta Multilayer Coverage. Así es que vemos que las pruebas que cubren el constructor en cuestión son las contenidas en las clases *PruebasLineaDe3* (unitarias) y *PruebasTablero* (de integración según nuestra definición, ya que son pruebas unitarias de la clase *Tablero*, cliente

[62] En realidad, no hemos seguido la refactorización de Fowler al pie de la letra, ya que la misma implicaría crear una clase para el arreglo de celdas, y en este caso hemos simplemente pasado el arreglo de celdas sin introducir un tipo nuevo.

de *LineaDe3*). Ergo, debería bastar con asegurarnos que esas pruebas corren sin problemas luego de la refactorización de *Tablero*.

Ahora sí deberíamos cambiar las clases *LineaDe3* y *Tablero*, para luego probar con las pruebas de integración. La clase *Tablero* utiliza al constructor de la clase *LineaDe3* en su propio constructor. El cambio es sencillo, y se encuentra en el anexo B.

Si introducimos los dos cambios (los constructores de *LineaDe3* y de *Tablero*), vemos que la clase *Tablero* ahora sí compila, y que no ha dejado de compilar ninguna de las clases de la aplicación: siguen compilando todas las clases del dominio y la única clase de pruebas que sigue sin compilar es *PruebasLineaDe3*, que era lo que ya sabíamos.
Siguiendo nuestro método, corremos las pruebas de la clase *PruebasTablero*, y obtenemos una corrida satisfactoria. Esto solo ya nos está diciendo que la refactorización introducida en el constructor de la clase *LineaDe3* es correcta, pues está preservando el comportamiento (recordemos que esto se debe a que las pruebas de *PruebasTablero* aseguran la cobertura total del constructor que estamos cambiando).
Por lo tanto, la refactorización es correcta.

Ahora bien, aún no hemos terminado, pues sigue sin compilar la clase *PruebasLineaDe3*, que es parte de nuestra aplicación, siguiendo el concepto de que las pruebas son parte integral del sistema.
Para que compilen, debemos introducir cambios en la clase *PruebasLineaDe3*, lo cual a priori parecería inseguro, al no contar con una red de contención. En realidad, lo que deberíamos hacer es mantener corriendo las pruebas de la clase *PruebasTablero* (que no deberían dejar de funcionar, pues no estamos afectando en nada lo que prueban), a la vez que logramos, sin modificar la clase *LineaDe3*, que cada una de las pruebas incluidas en *PruebasLineaDe3* van comenzando a funcionar nuevamente.
Si hacemos eso, estaremos usando a la clase de dominio *LineaDe3* (que sabemos que funciona bien, pues ya lo hemos comprobado con las pruebas de integración) como red de contención y como invariante de la clase *PruebasLineaDe3*.

Debido a que el cambio es muy simple, y que consiste en cambiar las llamadas al constructor de *LineaDe3* en todos los casos, el código se encuentra en el anexo B, para facilitar la lectura de la tesis.

Una vez realizados los cambios en las pruebas unitarias, podemos decir que la refactorización es correcta, pues:

- Las pruebas de la clase *PruebasTablero*, que no han sido modificadas, garantizan la cobertura y la preservación del comportamiento en las clases *LineaDe3* (modificada) y *Tablero* (no modificada).
- La propia clase *LineaDe3*, que no se ha modificado luego de la constatación anterior, garantiza la corrección de los cambios de la clase *PruebasLineaDe3*, realizados para que vuelva a quedar operativa.

6.3.4 Realización de una refactorización que necesita de las pruebas de comportamiento para su verificación

En este ítem realizaremos una refactorización no automática y en la cual las pruebas técnicas (unitarias y de integración) van a dejar de funcionar, de forma tal que necesitaremos de las pruebas de aceptación (del tipo de comportamiento) para asegurar la corrección del refactoring.

El caso que vamos a tomar será la eliminación de un cuasi-Singleton. Como explicamos en el anexo A, el patrón de diseño Singleton [Gamma 1994] presenta algunos problemas que lo convierten en un posible mal olor, y por lo tanto ameritaría una refactorización que lo elimine.

Si bien en nuestro caso de estudio no tenemos ningún Singleton clásico, la clase *Juego* es un cuasi-Singleton, ya que hay una sola instancia de dicha clase asociada a cada instancia de la aplicación.
En términos de la implementación, esto se logró haciendo que la clase *ServicioTaTeTi* tuviera una referencia a una instancia de la clase *Juego*, a la cual se accede a través del atributo *instanciaActiva*, y que la clase *Juego* tuviera un método *getInstanciaActiva*, estático, que retorna una referencia a la instancia en cuestión.
En efecto, el siguiente extracto de la clase *ServicioTaTeTi* muestra el manejo de una única instancia de *Juego*:

```
public class ServicioTaTeTi {

        private Juego juegoActivo;
        private int ultimaFilaJugada;
        private int ultimaColumnaJugada;
```

```
Juego getJuegoActivo() {
        return juegoActivo;
}

public TipoJuego getTipoJuego() {
        return juegoActivo.getTipoJuego();
}

public void setTipoJuego (TipoJuego tipoJuego) {
        juegoActivo.setTipoJuego(tipoJuego);
}

public int getUltimaFilaJugada() {
        return ultimaFilaJugada;
}

public int getUltimaColumnaJugada() {
        return ultimaColumnaJugada;
}

public ServicioTaTeTi (TipoJuego tipoJuego) {
        this.juegoActivo = new Juego (tipoJuego);
        this.ultimaColumnaJugada = -1;
        this.ultimaFilaJugada = -1;
}

boolean tableroVacio ( ) {
        return juegoActivo.getTablero().vacio();
}

boolean celdaOcupada (int fila, int columna) {
        return juegoActivo.getTablero().celdaOcupada(fila, columna);
}

void ocuparCelda (char simboloJugador, int fila, int columna)
                                        throws CeldaOcupadaException {
        Jugador jugador = juegoActivo.getJugadorPorSimbolo(simboloJugador);
        Tablero tablero = juegoActivo.getTablero();
        tablero.ocupar(jugador, fila, columna);
}

public boolean juegoTerminado ( ) {
        return juegoActivo.terminado();
```

```
	}

	public char getProximoAJugar ( ) {
		return juegoActivo.getProximoAJugar().getSimboloJuego();
	}

	public boolean hayTaTeTi ( ) {
		return juegoActivo.hayTaTeTi();
	}

	public char ganador ( ) {
		if ( hayTaTeTi() )
			return juegoActivo.ganador().getSimboloJuego();
		else
			throw new IllegalStateException("...");
	}

	boolean tableroCompleto ( ) {
		return juegoActivo.getTablero().completo();
	}

	// sigue la clase...

}
```

Y en la clase *Juego* se controla que haya una única instancia accesible:

```
public class Juego {

	private TipoJuego tipoJuego;
	private Tablero tablero;
	private Jugador jugador1;
	private Jugador jugador2;
	private Jugador proximoAJugar;
	private static Juego instanciaActiva;

	public Juego (TipoJuego tipoJuego) {
		this.tipoJuego = tipoJuego;
		this.tablero = new Tablero();
		this.jugador1 = new JugadorHumano('X');
		if (tipoJuego == TipoJuego.JugadorVsJugador)
			this.jugador2 = new JugadorHumano('O');
```

```
                else
                        this.jugador2 = new JugadorPC ( );
                this.proximoAJugar = jugador1; // para que empiece el jugador 1
                instanciaActiva = this;
        }

        public boolean hayTaTeTi ( ) {
                if ( instanciaActiva.getTablero().hizoTaTeTi() == null)
                        return false;
                else
                        return true;
        }

        public boolean terminado ( ) {
                Tablero tablero = instanciaActiva.getTablero();
                if ( tablero.completo() || (tablero.hizoTaTeTi() != null ) )
                        return true;
                else return false;
        }

        public Jugador ganador ( ) {
                return tablero.hizoTaTeTi();
        }

        static Juego getInstanciaActiva ( ) {
                return instanciaActiva;
        }

        // sigue la clase...

}
```

Por como está implementada la solución, con la limitación de una sola aplicación corriendo a la vez, esto configura el Singleton clásico, ya que hay sólo una instancia de la clase *Juego* por aplicación que esté corriendo. Además, hay algunas características indeseables en esta implementación, que hacen que se parezca a los malos olores de algunas implementaciones de Singleton. Entre ellas destacan:

- La clase *Juego* viola el principio de única responsabilidad [Martin 2002], ya que contiene la lógica de dominio propia del juego de Ta-Te-Ti, a la vez que controla la existencia de una única instancia por aplicación. Habiendo dos responsabilidades,

deberían estar en dos clases distintas: la lógica de la clase *Juego* en la clase *Juego*, y el control de una única instancia activa en la clase *ServicioTaTeTi*[63].

- Si bien lo anterior parecería muy sencillo, pues en los hechos está prácticamente implementado vía la referencia a la instancia activa de *Juego* en la clase *ServicioTaTeTi*, esta manera de implementar *Juego* provocó que algunas de las clases del dominio accedan a la instancia de *Juego* en forma global, utilizando el método *getInstanciaActiva*, lo cual deriva en los problemas clásicos del uso del patrón Singleton (ver anexo A). De hecho, como veremos luego, todas las pruebas de integración van a dejar de compilar al eliminar el método mencionado, lo cual está revelando un grado de acoplamiento enorme en la aplicación.

La refactorización que haremos afectará a cada clase que utilice el método *getInstanciaActiva* de la clase *Juego*, y consistirá en una variante de la que Cunningham [Cunningham c2] denomina Abstract Singleton:

- Agregar un atributo *juego*.
- Agregar un parámetro al constructor de la clase para inicializar el atributo *juego* (inyección de dependencia).
- Modificar los clientes de las clases para que invoquen al nuevo constructor.
- Eliminar todas las llamadas *Juego.getInstanciaActiva( )* y reemplazarlas por el uso del atributo *juego*.
- Verificar si las pruebas técnicas de integración siguen compilando. Si lo hacen, buscar con la herramienta de cobertura cuáles son las que ejercitan el método *getInstanciaActiva*, y correrlas.
- Si las pruebas técnicas de integración han dejado de compilar, verificar lo anterior con las pruebas de comportamiento que cubran la misma funcionalidad.
- Una vez seguido este procedimiento para todas las clases que invoquen el método *getInstanciaActiva* de la clase *Juego*, vamos a eliminar dicho método y volveremos a correr las pruebas de integración o de comportamiento.
- Finalmente, corregiremos las pruebas unitarias o de integración que hubiesen fallado y volveremos a correr el conjunto de todas las pruebas técnicas, unitarias y de integración.

[63] Esto ya está implementado en forma parcial de este modo, ya que desde la clase *ServicioTaTeTi* sólo se puede acceder a una instancia de *Juego*.

Veamos si hay cobertura redundante para el código que vamos a refactorizar. Usando Multilayer Coverage, vemos que las clases de prueba que ejercitan el código en cuestión son:

- *PruebasJuego* (unitarias)
- *PruebasJugadorHumano* (de integración)
- *PruebasJugadorPC* (de integración)
- *PruebasChequeoCeldas* (de comportamiento)
- *PruebasChequeoTurnos* (de comportamiento)
- *PruebasEstrategiaDefectoPC* (de comportamiento)
- *PruebasPCEvitaTaTeTiOponente* (de comportamiento)
- *PruebasPCHaceTaTeTi* (de comportamiento)
- *PruebasTaTeTiHecho* (de comportamiento)
- *PruebasSeteoJuego* (de comportamiento)

Como vemos, hay pruebas de los tres tipos que están involucradas. Por lo tanto, un nivel de pruebas podrá servirnos de cobertura ante la falla de un nivel de menor granularidad.
Una cuestión que nos va a llevar a hacer un análisis posterior es que todas las pruebas de comportamiento recorren el código a refactorizar. Ya volveremos sobre esto.

Analizando el código del dominio de la aplicación (utilizamos la detección de referencias cruzadas que nos da el entorno de desarrollo) vemos que las clases que utilizan el método *getInstanciaActiva* de la clase *Juego* son *Jugador* (abstracta, lo utiliza a través de su método *hacerJugada*) y *EstrategiaCeldaParaJugar* (abstracta, lo utiliza desde su constructor). Esto implica que, si bien las pruebas unitarias de la clase *Juego* (contenidas en la clase *PruebasJuego*) van a dejar de funcionar al eliminar el método *getInstanciaActiva*, tenemos a las pruebas de las clases clientes (de las clases concretas descendientes de *Jugador* y *EstrategiaCeldaParaJugar* en este caso) que actuarán como pruebas técnicas de integración. Y en última instancia, siguiendo nuestro método, podríamos recurrir a las pruebas de comportamiento.
La otra clase que se ve afectada es *ServicioTaTeTi*, de la capa de servicios, y que sólo está cubierta por pruebas de comportamiento.

Dado que estamos ante una refactorización de mayor envergadura, vamos a proceder por partes:

- En primer lugar, realizaremos los cambios en la clase *EstrategiaCeldaParaJugar*, sus clases dependientes, y corremos las pruebas necesarias para asegurar la corrección.
- Luego, cambiaremos la clase *Jugador* y sus clientes, siguiendo con las pruebas respectivas.

- Como tercer paso, haremos las modificaciones necesarias la clase *ServicioTaTeTi*, a la que probaremos con las pruebas de aceptación correspondientes.
- A continuación, eliminamos el atributo estático *instanciaActiva* y el método *getInstanciaActiva* de la clase *Juego* y vemos que las pruebas no se rompan.
- Finalmente, arreglamos las clases de pruebas que se hubiesen roto durante la refactorización.

Lo primero que hacemos, siguiendo el procedimiento, es cambiar la clase *EstrategiaCeldaParaJugar*. Le agregamos un atributo *juego*, y luego cambiamos el constructor, que era:

```
EstrategiaCeldaParaJugar ( ) {
        this.proximaEstrategia = null;
        this.juegoActivo = Juego.getInstanciaActiva();
        Jugador jugador2 = juegoActivo.getJugador2();
        if (jugador2 instanceof JugadorPC)
                this.jugadorPC = (JugadorPC)jugador2;
        else
                throw new IllegalStateException
                        ("error de programación: sólo debería llegar acá si juega la PC");
}
```

Por este nuevo constructor (las líneas que cambian figuran en negrita):

```
EstrategiaCeldaParaJugar ( Juego juego ) {
        this.proximaEstrategia = null;
        this.juegoActivo = juego;
        Jugador jugador2 = juegoActivo.getJugador2();
        if (jugador2 instanceof JugadorPC)
                this.jugadorPC = (JugadorPC)jugador2;
        else
                throw new IllegalStateException
                        ("error de programación: sólo debería llegar acá si juega la PC");
}
```

Como consecuencia de estos cambios, y antes de poder correr ninguna prueba, vemos que han dejado de compilar las tres clases concretas derivadas de *EstrategiaCeldaParaJugar*: *IntentarHacerTaTeTi*, *EvitarTaTeTiOponente* y *PrimeraCeldaLibre*.

La solución a este nuevo problema es sencilla, pues basta con agregar el parámetro *juego* a los constructores de las clases derivadas, y hacer que éstas lo usen para pasarlo como argumento del constructor de la clase madre. Por ejemplo, en el caso de *EvitarTaTeTiOponente* el constructor era:

```
EvitarTaTaTiOponente (EstrategiaCeldaParaJugar proximaEstrategia) {
        super();
        this.setProximaEstrategia (proximaEstrategia);
}
```

Y ahora quedó:

```
EvitarTaTaTiOponente (EstrategiaCeldaParaJugar proximaEstrategia, Juego juego) {
        super (juego);
        this.setProximaEstrategia (proximaEstrategia);
}
```

Esto a su vez afecta a la clase *JugadorPC*, que invoca a los constructores de estas clases y que, al tener el *juego* como uno de los atributos en su ancestro *Jugador*, es muy sencilla de corregir, aunque nos obliga a modificar nuevamente la clase *Jugador* agregando un método *getJuego*, protegido para que lo utilice su clase derivada *JugadorPC*.

Por lo tanto el método *mejorCeldaParaJugar* de *JugadorPC* que era:

```
Celda mejorCeldaParaJugar() {
        Celda mejorCelda = null;
        Juego juego = getJuego();
        EstrategiaCeldaParaJugar cadenaDeResponsabilidad = new IntentarHacerTaTeTi(
                        new EvitarTaTaTiOponente( new PrimeraCeldaLibre(null) ) );

        do {
                if (cadenaDeResponsabilidad == null)
                        // no debería ocurrir: tablero lleno y juego terminado
                        break;
                mejorCelda = cadenaDeResponsabilidad.mejorCeldaParaJugar();
                cadenaDeResponsabilidad =
                        cadenaDeResponsabilidad.getProximaEstrategia();
        }
```

```
                while (mejorCelda == null);

                return mejorCelda;
        }
```

Cambian las líneas tercera a quinta, así:

```
                EstrategiaCeldaParaJugar cadenaDeResponsabilidad =
                                new IntentarHacerTaTeTi( new EvitarTaTaTiOponente(
                                        new PrimeraCeldaLibre(null, juego), juego ), juego );
```

Si bien la legibilidad de estas líneas ha empeorado, en aras de mantener sencilla la refactorización, no la vamos a modificar en este paso.

Si analizamos lo que ocurrió luego de hacer estos cambios, vemos que no sólo dejaron de compilar las pruebas unitarias, sino también las clases de prueba de varios clientes de la clase *EstrategiaCeldaParaJugar*.
Si bien pareciera muy sencillo corregir estas pruebas para que funcionen, ya que en casi todos los casos los problemas provienen de las llamadas a los constructores de las clases derivadas de *EstrategiaCeldaParaJugar*, no deberíamos hacerlo, pues nos quedaríamos sin garantía de corrección de la refactorización. La única garantía que podría subsistir es la de las pruebas de comportamiento. Éste es el escenario que esperábamos que se produjera para poder usar el procedimiento completo de **refactoring asegurado por niveles de pruebas**.

Detengámonos en un detalle. En realidad, las únicas pruebas de comportamiento que deberíamos correr son aquellas que Multilayer Coverage nos haya indicado que cubrían el cambio realizado. Como dijimos antes, Multilayer Coverage marcó a todas las clases de pruebas de comportamiento de la aplicación.
Sin embargo, si nos detenemos solamente el cambio que estuvimos haciendo por el momento, las pruebas de aceptación a correr son las que se encuentran en las clases *PruebasPCHaceTaTeTi*, *PruebasPCEvitaTaTeTiOponente* y *PruebasEstrategiaDefectoPC*, según nos informa Multilayer Coverage. Por ello, nos quedamos con correr esas pruebas para garantizar la preservación del comportamiento. Las corremos, y las pruebas pasan, con lo cual nos quedamos tranquilos respecto de esta primera parte.

Como decíamos antes, la otra clase que utiliza el código a cambiar en la clase *Juego* es la clase *Jugador*. En esta clase procedemos de forma análoga a lo que hicimos con la clase *EstrategiaCeldaParaJugar*, generando el atributo *juego* e incorporando un parámetro al constructor que permita inicializarlo.

También resulta afectado el método *hacerJugada* de *Jugador*, que era:

```
protected void hacerJugada (int fila, int columna)
                throws CeldaOcupadaException, ViolacionTurnoException {
        if ( Juego.getInstanciaActiva().getProximoAJugar() != this )
                throw new ViolacionTurnoException();
        Juego.getInstanciaActiva().getTablero().ocupar(this, fila, columna);
        Juego.getInstanciaActiva().cambiarTurno();
}
```

Y ahora queda así:

```
protected void hacerJugada (int fila, int columna)
                throws CeldaOcupadaException, ViolacionTurnoException {
        if ( juego.getProximoAJugar() != this )
                throw new ViolacionTurnoException();
        juego.getTablero().ocupar(this, fila, columna);
        juego.cambiarTurno();
}
```

Como pasó con *EstrategiaCeldaParaJugar*, al ser la clase *Jugador* una clase abstracta, sus principales clases clientes son sus derivadas: *JugadorHumano* y *JugadorPC*. La modificación es análoga a lo que hicimos antes. Por ejemplo, en el caso de *JugadorPC* el constructor era:

```
public JugadorPC ( ) {
        super ('O');
}
```

Y ahora quedó:

```
public JugadorPC (Juego juego) {
        super ('O', juego);
}
```

Esto a su vez provoca que deje de compilar la clase *Juego*, ya que su constructor era:

```
public Juego (TipoJuego tipoJuego) {
        this.tipoJuego = tipoJuego;
        this.tablero = new Tablero();
        this.jugador1 = new JugadorHumano('X');
        if (tipoJuego == TipoJuego.JugadorVsJugador)
                this.jugador2 = new JugadorHumano('O');
        else
                this.jugador2 = new JugadorPC ( );
        this.proximoAJugar = jugador1; // para que empiece el jugador 1
        instanciaActiva = this;
}
```

Y también tenía un método *setTipoJuego*:

```
public void setTipoJuego (TipoJuego tipoJuego) {
        this.tipoJuego = tipoJuego;
        if (tipoJuego == TipoJuego.JugadorVsJugador)
                this.jugador2 = new JugadorHumano('O');
        else
                this.jugador2 = new JugadorPC ( );
}
```

Este problema se soluciona haciendo que el propio objeto receptor (tipo *Juego*) se pase a sí mismo a los constructores de los tipos de jugadores. Por ejemplo, el método *setTipoJuego* quedará:

```
public void setTipoJuego (TipoJuego tipoJuego) {
        this.tipoJuego = tipoJuego;
        if (tipoJuego == TipoJuego.JugadorVsJugador)
                this.jugador2 = new JugadorHumano('O', this);
        else
                this.jugador2 = new JugadorPC (this);
}
```

Ahora estamos en el mismo punto que en el cambio de las clases *EstrategiaCeldaParaJugar* y sus derivadas: varias pruebas técnicas han dejado de compilar, pero siguen corriendo las

pruebas de comportamiento. Por ello corremos todas las pruebas que Multilayer Coverage nos ha marcado como necesarias para garantizar la preservación del comportamiento.

La descripción que hemos hecho corresponde a los cambios en las clases de dominio. Otra clase que se ve afectada por la refactorización es la clase de servicios *ServicioTaTeTi*. Con ella, procedemos como en los otros dos casos, analizando cobertura con Multilayer Coverage, cambiando lo necesario y corriendo las pruebas de aceptación al final.

Ahora falta el paso decisivo, el más importante: eliminar el atributo estático *instanciaActiva* y el método *getInstanciaActiva* en la clase *Juego*, que son los que motivaron todos los cambios anteriores para evitar el cuasi-Singleton. Si todo estuviese bien, se deberían poder borrar sin afectar el comportamiento del sistema.
Por lo tanto, borramos el atributo y el método, cambiando las apariciones del atributo por la referencia al objeto receptor *this* en todos los métodos en que aquél se estuviera usando.
Al hacerlo, observamos que no aparecen errores de compilación en las clases del dominio, tampoco en las pruebas de comportamiento, aunque sí aparecen algunos problemas adicionales en las pruebas técnicas. Ignorando las pruebas técnicas, que no compilaban desde las primeras modificaciones introducidas, corremos el conjunto de pruebas de comportamiento que indicó Multilayer Coverage al principio, y éstas pasan sin problema, garantizando la corrección de la refactorización.

Ahora bien, la refactorización es correcta, pero necesitamos corregir las pruebas técnicas, entre otras cosas, para futuras refactorizaciones. Todas las clases de pruebas técnicas tienen problemas de compilación a esta altura. No es tan raro si tenemos en cuenta que hemos realizado una refactorización que ha encapsulado un objeto de acceso global.
Aparentemente estaríamos encarando una tarea riesgosa, ya que estaríamos cambiando pruebas que funcionaban antes de la refactorización, y para las cuales pareciera que no tenemos red de contención. Pero no es así: una red de contención obvia la brindan las pruebas de comportamiento; y la otra, tal vez menos evidente, son las propias clases del dominio de la aplicación, que se sabe que son correctas porque lo dicen las pruebas de aceptación, y a las cuales no modificaremos. Así que, lejos de perder una red de contención, contamos con dos redes, una en cada extremo.
Por lo tanto, por cada cambio que hagamos, vamos a chequear que las pruebas de comportamiento sigan funcionando sobre el código básico, que no cambiaremos.

Algunos cambios son muy menores. Por ejemplo, en la clase *PruebasCelda* hay dos líneas que han dejado de compilar. La inicialización de un atributo:

```
private JugadorHumano jugador = new JugadorHumano('X');
```

Y la inicialización de una variable local en el método *pruebaOcupar*:

```
JugadorHumano jugadorPrevio = new JugadorHumano('O');
```

Obviamente, los cambios llevan a que a dichas líneas se les agregue como parámetro una instancia cualquiera de *Juego*, la misma en ambos casos. Aprovechamos para mejorar un poco el diseño y definimos un método de inicialización para los atributos:

```
private JugadorHumano jugador;
private JugadorHumano jugadorPrevio;

@Before
public void inicializacion ( ) {
        Juego juego = new Juego (TipoJuego.JugadorVsJugador);
        jugador = new JugadorHumano('X', juego);
        jugadorPrevio = new JugadorHumano('O', juego);
}
```

Luego de este cambio, la clase *PruebasCelda* compila y corre sin problemas.
Corremos a continuación las pruebas de comportamiento, y vemos también su correcto funcionamiento. Esta corrida de las pruebas de comportamiento no es necesaria, de todas maneras, porque al no haber cambiado en nada el código de las clases del dominio, no debería cambiar el resultado de las mismas. No obstante, las corremos para evitar cualquier problema inadvertido.

La misma situación se da en las clases *PruebasLineaDe3* y *PruebasTablero*.
En ambas agregamos el argumento faltante a las invocaciones de los constructores de las clases *JugadorHumano* y *JugadorPC*, verificamos que las clases de pruebas técnicas compilen, que corran sin problemas, y finalmente corremos las pruebas de comportamiento para asegurarnos de la corrección de los cambios hechos.

En el caso de las clases *PruebasJugadorHumano* y *PruebasJugadorPC*, nos encontramos con que no sólo debemos introducir el cambio en cuestión, sino que también hay referencias al método *getInstanciaActiva* de la clase *Juego*, ahora inexistente luego de la refactorización.
En realidad, el cambio es sencillo: en ambos casos se necesita un objeto de tipo *Juego* en diversas pruebas. Basta entonces con crear ese objeto con anticipación y usarlo como el que antes era la instancia activa.

Por ejemplo, el método *precondicionPruebaJugadas* en *PruebasJugadorPC* era:

```
private void precondicionPruebaJugadas ( ) {
        new Juego (TipoJuego.JugadorVsPC);
        juegoActivo = Juego.getInstanciaActiva();
        jugadorHumano = (JugadorHumano) juegoActivo.getJugador1();
        jugadorPC = (JugadorPC) juegoActivo.getJugador2();
        tablero = juegoActivo.getTablero();
}
```

Y sus dos primeras líneas se verán reducidas a:

```
        juegoActivo = new Juego (TipoJuego.JugadorVsPC);
```

Como siempre, hacemos el cambio, nos aseguramos de que la clase compila, que la prueba corre sin problemas y luego, para evitar errores involuntarios, corremos también las pruebas de comportamiento.

Con la clase *PruebasJugadorHumano* procedemos exactamente igual.

La clase *PruebasJuego* es más sencilla aún, pues sólo presenta accesos al método inexistente *getInstanciaActiva*, que solucionamos de la misma manera que en los dos casos anteriores.

En este punto hemos terminado con la refactorización de eliminación del cuasi-Singleton, que nos obligó a utilizar pruebas de comportamiento al romperse las pruebas unitarias y técnicas de integración.

Hemos visto en este capítulo al enfoque de esta tesis en acción. Se lo ha usado en una aplicación real, que aunque pequeña, cumple los estándares de diseño de las de mayor envergadura. También se ha presentado y puesto a prueba una herramienta de chequeo de cobertura redundante con pruebas de distinta granularidad, que nos ha servido para garantizar, en nuestras refactorizaciones, que podíamos suprimir algunas pruebas que se rompían debido a las mismas.
A partir de aquí, vamos a ir cerrando la tesis con conclusiones y trabajos relacionados.

7 Conclusiones

Este capítulo cierra la tesis con una recapitulación sobre aportes de la misma, una discusión sobre limitaciones y derivaciones del método, y una breve reseña de temas y trabajos relacionados.

7.1 Discusión

7.1.1 Limitaciones de las alternativas al método propuesto

Esta tesis presenta un procedimiento que permite refactorizar código en forma segura aun cuando las pruebas unitarias dejen de funcionar debido a la propia refactorización, a condición de contar con pruebas de mayor granularidad que brinden cobertura sobre el código a refactorizar.

Hay muchos autores (por ejemplo, [Beck 1999, Fowler 1999]) que sostienen que, en la medida en que se hagan refactorizaciones frecuentes, siempre van a ser pequeñas. Sin desmerecer esas opiniones, que tienen su parte de verdad, lo cierto es que no siempre se pueden evitar las grandes refactorizaciones con la práctica del refactoring continuo. Ello se debe principalmente a:

- Si bien todo diseño se hace con el cambio en mente, hay ocasiones en que un proyecto debe cambiar en un sentido no previsto durante su diseño inicial. Por lo tanto, las refactorizaciones pequeñas podrían no haber sido hechas en la dirección del cambio que se necesita.
- Es usual que un equipo deba hacerse cargo de un proyecto en el cual no se vino realizando habitualmente refactorizaciones. En estos casos, muy frecuentes, nos encontramos ya con un diseño entrópico antes de comenzar a trabajar en el mismo, lo cual nos lleva a grandes refactorizaciones y a la necesidad de un enfoque como el presentado en esta tesis.

7.1.2 Argumentos en favor de ATDD

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, utilizando como punto de partida a las pruebas de aceptación. ATDD aún no se encuentra tan difundida en la industria, 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.
No faltan autores que coincidan con este planteo, aunque han expresado sus opiniones de forma muy escueta y sin detalles metodológicos. Entre ellos, destacan Adzic [Adzic 2009], Mugridge y Cunningham [Mugridge 2005] y el sitio web de la herramienta Concordion[64]. Sin embargo, hasta donde conocemos, nadie ha presentado una práctica metodológica que incluya el problema de la cobertura. De allí la relevancia de este trabajo.
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.

Una solución alternativa, cuando no hay pruebas suficientes o cuando no se cuenta con pruebas de mayor granularidad, podría ser el uso de generadores automáticos de pruebas. Existen varias herramientas para ello, y su uso principal son las pruebas de regresión, dado que se basan en el código ya escrito.
Un generador automático, como Randoop [Pacheco 2007], genera pruebas en base al código. Sin embargo, suelen ser pruebas unitarias y muy simples.
También hay otras herramientas, como JCrasher, Eclat, Jartege y Jtest. Algunas de éstas trabajan con contratos (en el sentido del diseño por contrato, de Bertrand Meyer [Meyer 2000]) para inferir las pruebas, otras se basan en anotaciones[65] en el código, algunas en ambas cosas.
Todas estas herramientas pueden servir para generar pruebas antes de un refactoring sobre código que no las tenía, y por lo tanto podrían servir para generar pruebas adicionales, que suplanten pruebas de unidad que se hayan roto en una refactorización. Sin embargo, no suelen ser un sustituto de nuestro enfoque por al menos dos razones:

- Las pruebas generadas por las herramientas existentes en la actualidad son de baja calidad, por lo que requieren una revisión manual por parte del programador.
- Al estar basadas en código, son de una granularidad excesivamente baja, con lo cual de ninguna manera pueden ofrecer una red de contención de un nivel mayor ante pruebas unitarias que se rompen.

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

[65] Usamos el término anotación para indicar la metainformación que algunas plataformas permiten generar para ser procesada por otros programas. Por ejemplo, los *Attribute* de .NET y las *annotations* de Java.

7.2 Trabajos relacionados

Hay algunos temas en los cuales se está trabajando que tienen puntos en común con el que ha motivado esta tesis. A continuación se mencionan y describen brevemente los mismos.

7.2.1 Refactoring de pruebas

Si trabajamos con pruebas automatizadas, como ya dijimos, las pruebas también son código, y por lo tanto son susceptibles de refactorización. Si además suscribimos el argumento de que el código de las pruebas es parte del código del sistema, y que por lo tanto debe cumplir los mismos requisitos de calidad que el resto del código, el refactoring de pruebas pasa a ser importante, ya que ayudaría a controlar la entropía del código de las pruebas.
Así es como hay autores [Meszaros 2007, Deursen 2001] que han tratado este tema y han descubierto algunos olores de pruebas, que suelen ser distintos de los olores del código común. Las refactorizaciones típicas de las pruebas también son distintas de las de código funcional, aunque hay pocos trabajos que las analicen y cataloguen.

Lo interesante del refactoring de pruebas es que lleva a problemas parecidos a los de esta tesis. De la misma manera en que en este trabajo encontramos situaciones en las cuales no teníamos pruebas unitarias que nos sirviesen para verificar la corrección de una refactorización, no suele haber pruebas que garanticen la invariancia de una prueba después de un cambio. Esto es, no hay pruebas de pruebas.
El camino más factible es considerar al código funcional como herramienta de verificación de las pruebas. Sin embargo, es un tema poco desarrollado.

7.2.2 Refactorizaciones que afectan protocolos publicados

Los protocolos publicados son un problema serio para el refactoring que, al igual que lo que ocurre en esta tesis, no cuenta con suficiente cobertura metodológica.

Un protocolo publicado de una aplicación es una especificación de servicios que la aplicación ofrece a otros sistemas, sean estos dentro de la misma compañía o no. Por ejemplo, Google Maps[66] ofrece servicios web que permiten ubicar lugares o determinar caminos entre localizaciones; Despegar[67] permite hacer reservas de viajes y hoteles a través de un protocolo programático.

[66] https://maps.google.com/

[67] http://www.despegar.com.ar/

La manera de usar dichos servicios ha sido dada a conocer a sus potenciales clientes (los servicios han sido "publicados"), y cualquier cambio a los mismos va a afectar a aplicaciones distintas a la que está sufriendo el cambio.
En el caso de las refactorizaciones garantizamos la preservación del comportamiento, pero no la invariancia de los protocolos de las clases, por lo que deberíamos tener mucho cuidado de no afectar protocolos publicados.

Este tema ha sido tratado por varios autores, aunque no siempre en el contexto inmediato del refactoring. Entre los trabajos que sí lo tratan en este contexto destacan el de Dig y Johnson [Dig 2005], el de Diwan y Henkel [Diwan 2005] y el de Lippert y Roock [Lippert 2006]. Incluso los trabajos iniciales de Opdyke [Opdyke 1990, Opdyke 1992] ya estaban motivados por la evolución de frameworks, lo que implícitamente apunta a considerar los protocolos publicados.

La recomendación más generalizada es diseñar con mucho cuidado los protocolos publicados y no modificarlos en el futuro. Por supuesto, como toda solución que implique estabilidad eterna, es inviable.
Otra "solución" es considerar a los protocolos publicados como parte del comportamiento observable, ya que los sistemas externos son también usuarios de la aplicación y detectan a los protocolos como parte del comportamiento. Si así fuera, un cambio de un protocolo publicado no sería una refactorización, y nos habríamos librado de un problema. Pero lo cierto es que hay situaciones en las cuales los cambios de protocolos publicados son una necesidad, más allá de considerarlos o no refactorizaciones.

Yendo a problemas concretos, hay situaciones de inconvenientes bien previsibles. Por ejemplo, si cambiamos el nombre de un método, está claro que los clientes van a dejar de funcionar. Pero hay problemas más sutiles y, por lo tanto, más difíciles de detectar.
Por ejemplo, supongamos que en una aplicación Java agregamos un método abstracto a una clase no final que esté en un protocolo publicado: automáticamente, todas las clases que se hayan creado en otras aplicaciones como derivadas de ésta se convertirán en abstractas. El agregado o eliminación de excepciones de la firma de un método también causa problemas en clases descendientes.
Incluso pueden surgir problemas mucho tiempo después. Siguiendo con el caso de Java, si eliminamos la cláusula *synchronized* en un método publicado, pueden comenzar a tener problemas algunos clientes que utilizasen dicho método en un entorno concurrente. Más aún,

el agregado de la cláusula *synchronized* en un método publicado, que aparentemente no sería un problema, puede provocar *deadlocks* que antes no se producían.

Lo importante es reconocer que el problema existe, y proveer herramientas de migración[68] que mitiguen los problemas derivados del mismo, sea en forma automática o como ayuda durante dicha migración.
En los lenguajes compilados con chequeo de tipos estático, el compilador suele ser de gran ayuda, ya que casi todos los problemas se detectan en el momento de la compilación. Y si bien no tenemos control sobre las aplicaciones que utilizan nuestros protocolos, sí deberíamos tener pruebas que, al compilarlas, nos adviertan de los problemas provocados por los cambios, como para realizar una guía de migración para los clientes.
Algunos trabajos se han realizado para automatizar la migración. Destaca CatchUp [Diwan 2005], una herramienta que graba en un *script* las refactorizaciones a medida que se van realizando, de modo tal que dicho *script* luego pueda servir para refactorizar las aplicaciones clientes.
Otras propuestas son las de las publicaciones [Lippert 2006] y [Diwan 2005]. La propuesta de [Lippert 2006] se basa en un uso amplio de la cláusula *deprecated* de Java, más una serie de etiquetas de refactoring que ellos mismos proponen y que basan en el trabajo de Havenstein y Roock [Havenstein 2002]. También proponen herramientas para migración de bibliotecas.
En los últimos años han aparecido propuestas similares basadas en anotaciones[69], aprovechando su incorporación a .NET y Java en las últimas versiones de esas plataformas.

7.2.3 Refactoring de código compartido

Cuando el código no tiene un dueño (en línea con uno de los principios de XP, de propiedad colectiva de código) y cada desarrollador trabaja con su propia copia, las refactorizaciones hechas por un programador pueden afectar a los cambios que está haciendo otro en el mismo código. En estos casos, confiar en que una determinada refactorización la hace el entorno de desarrollo automáticamente, es erróneo, porque esos cambios sólo afectan la copia local.

[68] Con "migración" nos referimos a los cambios que se deban hacer en una aplicación cliente como consecuencia de un cambio en una interfaz publicada.

[69] Usamos el término anotación para indicar la metainformación que algunas plataformas permiten generar para ser procesada por otros programas. Por ejemplo, los *Attribute* de .NET y las *annotations* de Java.

Es cierto que, cuando se trabaja en equipo, se suele usar alguna herramienta de control de versiones, como CVS[70] o SVN[71], que permiten distinguir divergencias en cambios, e incluso trabajan con ramas que luego se unen.
El problema que tienen estas herramientas es que se basan en texto, sin reconocimiento de la semántica del lenguaje de programación que se utilice[72]. Por ello, sólo marcan conflictos cuando dos programadores cambian una misma línea, sin importar métodos, atributos o clases. Por ejemplo, ante un cambio de nombre de algún elemento semántico (una clase, un método), no tienen forma de saber cómo cambiar el nombre en todos lados, ni forma de reconocerlo si luego viene un cambio de otro autor sobre el mismo.
Otro problema de estas herramientas es que no guardan un historial de refactorizaciones.

Hubo varios intentos de resolver este problema. El trabajo más completo es el de la herramienta MolhadoRef [Dig 2006], un sistema de gestión de la configuración, construida como prototipo, que permite refactorizaciones e intercalado de versiones seguras. MolhadoRef se basa en *operaciones*, entendiendo por tales a cada uno de los cambios hechos sobre el código, entre ellas las refactorizaciones. Una versión se ve como una construcción a partir de una serie de *operaciones* que se hacen sobre el código, las cuales se guardan en forma persistente, quedando un historial de las mismas. Gracias a esto, cada operación se puede deshacer o volver a correr. Para manejar los elementos semánticos, los almacena en el mismo repositorio, en base a un identificador numérico que permanece igual, aun cuando se cambien los nombres. Incluso admite la intervención del programador, para situaciones como la resolución de referencias circulares. La inversión de una refactorización compuesta también es más sencilla, siendo sólo la eliminación de la serie de operaciones almacenadas: esto es más trabajoso, pero también más seguro, que una serie de reversiones sobre cambios en el texto.
En resumen, MolhadoRef es un sistema de administración de configuración, semántico, basado en operaciones y con identificadores persistentes. Se construyó como un *plugin* de Eclipse que trabaja sobre Java, con lo cual conoce la sintaxis y semántica de Java.

Además de MolhadoRef ha habido otras soluciones [Ekman 2004], pero no todas son igual de completas. Algunas de las más conocidas guardan datos en memoria no persistente. Eclipse mismo, a partir de la versión 3.2, permite almacenar refactorizaciones que se hayan hecho,

[70] http://cvs.nongnu.org/

[71] http://subversion.apache.org/

[72] Esto es una consecuencia de haber buscado independencia respecto del lenguaje.

para guardar y volver a correr. Hay herramientas que trabajan con grafos de dependencias o anotaciones semánticas para detectar conflictos de comportamiento [Linand 1996].

Junto con MolhadoRef, se desarrolló la herramienta RefactoringCrawler [Dig 2006-2], que infiere refactorizaciones hechas a mano para incluirlas en el historial de refactorizaciones que luego usa MolhadoRef, y en ese sentido podría proveer una base para resolver cambios en pruebas unitarias derivadas de una refactorización.

7.2.4 Reparación de pruebas que dejan de funcionar por cambios de requerimientos

Cuando nos enfrentamos a un cambio de requerimientos, es esperable que algunas pruebas no sirvan más. En efecto, si las pruebas de aceptación son definiciones operacionales de requerimientos, un cambio de los mismos implica un cambio en aquéllas. Y como las pruebas técnicas no son más que la materialización de los requerimientos en unidades más elementales, es también de esperar que cambien.
Si bien un cambio de requerimientos, por definición, no es una refactorización, los cambios en las pruebas tienen algunos puntos de contacto con el planteo de esta tesis.

Para reparar en forma automática pruebas que se hayan roto por cambios en requerimientos, existe una técnica [Brett 2009] y una herramienta llamada ReAssert, desarrollada para Java, como *plugin* de Eclipse. También se la puede usar en forma asistida cuando fuera necesario. Como ocurre con Multilayer Coverage, requiere un análisis en tiempo de ejecución.
No obstante, en su estado actual, sólo resuelve problemas simples, y pruebas escritas de forma sencilla, por lo que no parece muy adecuada para resolver los problemas que nos hemos planteado en esta tesis.
La única ventaja es que provee posibilidades de extensión para agregar nuevas estrategias de cambio de pruebas.

7.2.5 Trabajos del autor relacionados

El autor de esta tesis está trabajando con alumnos de la carrera de Ingeniería Informática de la Facultad de Ingeniería de la Universidad de Buenos Aires, en algunos temas relacionados con esta tesis. Entre ellos:

- Ha dirigido a un alumno en su Trabajo Profesional de Ingeniería Informática en la construcción de la herramienta Multilayer Coverage, que se utiliza en esta tesis.

- Está dirigiendo a un alumno en su Trabajo Profesional de Ingeniería Informática, referido a la refactorización en gran escala de una aplicación Java que no cuenta con pruebas automatizadas, y para la cual debe generarlas.
- Está dirigiendo a un alumno en su Tesis de Grado de Ingeniería Informática en un trabajo que evalúa la pertinencia de ATDD ante cambios de requerimientos y su comparación con el uso más tradicional de UTDD.
- Está dirigiendo a un alumno en su Tesis de Grado de Ingeniería Informática en un trabajo que evalúa la pertinencia de ATDD en proyectos de integración de sistemas.

Todos estos trabajos pretenden evaluar algunos aspectos de esta tesis y otros que están relacionados con ella. Se están desarrollando aplicaciones web y para dispositivos móviles inteligentes. El propósito es validar a fondo el método, en aplicaciones mayores a la del caso de estudio presentado, con distintas arquitecturas y para diferentes plataformas.

7.3 Recapitulación de los aportes de la tesis

Los aportes principales de la tesis fueron:

- Se presentó un enfoque metodológico completo para cubrir situaciones en que un determinado nivel de pruebas no alcanza para asegurar la corrección de una refactorización. Lo hemos denominado **refactoring asegurado por niveles de prueba**, y tiene como condición necesaria la existencia de pruebas de mayor granularidad, automatizadas, y que cubran al menos el código a cambiar. Ver capítulo 5.
- Se construyó la herramienta Multilayer Coverage, aprovechando las facilidades de extensión de la herramienta de análisis de cobertura EclEmma. La misma provee información sobre cobertura redundante que facilita detectar cuáles son las pruebas que cubren la porción de código a refactorizar, y cómo unas pruebas de un nivel se cubren con otras más abarcativas. Ver capítulo 6.
- Se validó el procedimiento planteado de *refactoring asegurado por niveles de prueba* y la herramienta Multilayer Coverage con un caso de estudio que sigue las mejores prácticas metodológicas y de arquitectura de software. Ver capítulo 6.
- Se encontró una ventaja adicional a las habitualmente citadas del uso de ATDD como práctica metodológica. En efecto, además de la utilidad innegable de contar con pruebas de aceptación escritas antes del desarrollo, para especificar los requerimientos del cliente sin ambigüedades, facilitar la comunicación entre perfiles y permitir regresiones, el uso de las pruebas de aceptación como invariante último del refactoring

es una razón más para la adopción de ATDD en una proporción mayor a la actual. Ver ventajas de ATDD en capítulo 3 y discusión en capítulo 7.

- Se encontraron puntos en común entre la problemática de la tesis y otros problemas de la Ingeniería de Software. Por lo tanto, el procedimiento de *refactoring asegurado por niveles de prueba* podría servir como base para resolver esos problemas. Entre los temas relacionados se encuentran: refactoring de código legacy, refactoring de pruebas, refactorizaciones que afectan protocolos publicados y refactoring de código compartido. Ver trabajos relacionados en capítulo 7.
- Se demostró la ingenuidad del planteo de que las pruebas unitarias brindan una contención eficaz para garantizar la preservación del comportamiento. Asimismo, se expusieron los enfoques que plantearon otros autores [Pipka 2002, Lippert 2006] para resolver el caso anterior, explicando sus cuáles eran sus déficits. El caso de estudio sirvió para demostrar en la práctica la inadecuación de estos planteos, toda vez que mostraron ser insuficientes aun para realizar refactorizaciones simples en una aplicación pequeña. Ver capítulos 4, 5 y 6.

7.4 Direcciones futuras

En futuras versiones, la herramienta Multilayer Coverage se podría mejorar en varios aspectos. Por ejemplo:

- Además de mostrar las pruebas que cubren una determinada porción de código, permitir la navegación hacia las mismas.
- Mejorar la granularidad de la cobertura del lado de las pruebas. Esto es, no mostrar sólo la clase, sino los métodos de prueba que cubren una porción de código a cambiar.
- Mejorar la granularidad de la cobertura del lado del código. La versión actual busca la cobertura de un método. Una versión futura podría analizar la cobertura de una porción de código más pequeña.

Un tema de investigación a explorar es el de refactoring de pruebas, con la ayuda de un enfoque basado en cobertura múltiple.
En efecto, modificar pruebas sin un cambio de requerimientos, implica asegurarse que no se está cambiando el comportamiento del sistema.
En el caso de las pruebas técnicas, se puede analizar el código que cubren y usar el propio código como red de seguridad. Aun si el código debiese ser cambiado, las pruebas de aceptación podrían funcionar como una red de seguridad de último nivel.

También puede ser que se desee refactorizar pruebas de aceptación. Si bien en esta tesis hemos mantenido la premisa de que las pruebas de aceptación son los invariantes de una refactorización, puede ser que, sin cambiar la intención que las mismas expresan, deseemos escribirlas de una manera más clara. En ese caso, las pruebas técnicas y el propio código servirían como redes de contención de la refactorización, para lo cual un análisis de la cobertura sería de gran utilidad.

El tema de reparación de pruebas ante cambios de requerimientos es otro posible tema de investigación.
Por ejemplo, en un proyecto en el que se utilice ATDD para el desarrollo, un cambio de requerimientos debería provocar uno o más cambios en las pruebas de aceptación, antes de encarar el desarrollo del cambio. El inconveniente es que resulta difícil establecer la correlación entre el cambio en las pruebas de aceptación y la necesidad de cambio en pruebas de menor granularidad. En estos casos, la herramienta Multilayer Coverage, acompañada por algún enfoque metodológico similar al de esta tesis, aunque en sentido inverso (de mayor a menor granularidad), sería de gran ayuda. Así se podría saber qué pruebas técnicas cubren el mismo código que las pruebas de aceptación que deban cambiarse, y por lo tanto, analizar cambios en las mismas.

8 Bibliografía y referencias

[Adzic 2009] Gojko Adzic, “Bridging the Communication Gap. Specification by example and agile acceptance testing”, Neuri, 2009.

[Ambler 2006] Scott W. Ambler y Pramodkumar J. Sadalage, “Refactoring Databases: Evolutionary Database Design”, Addison-Wesley Professional, 2006.

[Ambler 2011] Scott Ambler, “The Object Primer: The Application Developer's Guide to Object Orientation and the UML”, Cambridge University Press; 2nd edition, 2011.

[Astels 2003] David Astels, "Test-Driven Development: A Practical Guide", Prentice Hall, 2003.

[Bazzocco 2012] Javier Bazzocco, “Persistencia orientada a objetos”, Universidad Nacional de La Plata, 2012, ISBN 978-950-34-0821-6, disponible como e-book en http://www.editorial.unlp.edu.ar/22_libros_digitales/BAZZOCCO_Persistencia%20Orientada%20a%20Objectos.pdf en diciembre de 2012.

[Beck 1996] Kent Beck, “Smalltalk Best Practice Patterns”,Prentice Hall, 1996.

[Beck 1998] Kent Beck, "Kent Beck's Guide to Better Smalltalk", Cambridge University Press, 1998.

[Beck 1999] Kent Beck, “Extreme Programming Explained: Embrace Change”, Addison-Wesley Professional, 1999.

[Beck 2002] Kent Beck, “Test Driven Development: By Example”, Addison-Wesley Professional, 2002.

[Boger 2003] Marko Boger, Thorsten Sturm, Per Fragemann, “Refactoring Browser for UML”, International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World, Lecture Notes in Computer Science, 2003, Volume 2591/2003, 366-377, DOI: 10.1007/3-540-36557-5_26.

[Brett 2009] Brett Daniel, Vilas Jagannath, Danny Dig, Darko Marinov, “ReAssert: Suggesting Repairs for Broken Unit Tests”, Proceedings of ASE'09, pp 433-444, Auckland, New Zealand, 2009.

[Brodieand 1995] Michael L. Brodieand y Michael Stonebraker, “Migrating Legacy Systems: Gateways, Interfaces and the Incremental Approach”, Morgan Kaufman, 1995.

[Brooks 1975] Frederick P. Brooks, Jr., “The Mythical Man-Month: Essays on Software Engineering”, Addison Wesley, 1975.

[Burella 2010] Juan Burella, Gustavo Rossi, Esteban Robles Luna, Julián Grigera, “Dealing with Navigation and Interaction Requirement Changes in a TDD-Based Web Engineering

Approach", Proceedings of the The 11th International Conference on Agile Software Development, Springer Verlag, LNCS, 2010.
[Buschmann 1996] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, "Pattern-Oriented Software Architecture Volume 1", Wiley, 1996.
[Chen 2008] Jinghong Cox Chen, Hewijin Christine Jiau, Lee Wei Mar, "Previewing the Effects of Refactoring: From Structural Changes to Metrics Changes Induction", VDM Verlag, 2008.
[Cinnéide 2000] Mel O Cinnéide, "Automated Application of Design Patterns: A Refactoring Approach", tesis de PhD, Trinity College Dublin, 2000.
[Cornett] Steve Cornett, "Code Coverage Analysis", consultado en febrero de 2013 en http://www.bullseye.com/coverage.html .
[Cunningham c2] "Singleton Refactorings", Cunningham & Cunningham, Inc., disponible en http://c2.com/cgi/wiki?SingletonRefactorings en diciembre de 2012.
[Deursen 2001] Arie van Deursen, Leon Moonen, Alex van der Bergh, Gerard Kok, "Refactoring Test Code", Proceedings of XP2001: 2nd International Conference on eXtreme Programming and Flexible Proceses in Software Engineering, 92-95, 2001.
[Dig 2005] Danny Dig, Ralph Johnson, "The Role of Refactorings in API Evolution", Proceedings of International Conference on Software Maintenance (ICSM'05), pp. 389-398, Budapest, Hungary, 2005.
[Dig 2006] Danny Dig, Kashif Manzoor, Tien N. Nguyen, Ralph Johnson, "MolhadoRef: a refactoring-aware infrastructure for OO programs", Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange , 2006.
[Dig 2006-2] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson, "Automatic detection of refactorings in evolving components", Proceedings of European Conference on OO Programming (ECOOP'06), 2006.
[Diwan 2005] Amer Diwan y Johannes Henkel, "CatchUp! Capturing and Replaying Refactorings to Support API Evolution", Proceeding of the 27th international conference on Software engineering ACM, 2005.
[Ekman 2004] T. Ekman, U. Asklund, "Refactoring-aware versioning in Eclipse", Electronic Notes in Theoretical Computer Science, Proceedings of the Second Eclipse Technology Exchange: eTX and the Eclipse Phenomenon (eTX 2004), 2004.
[Elssamadisy 2007] Amr Elssamadisy, "Patterns of Agile Practice Adoption – The Technical Cluster", CA Media, InfoQ Enterprise Software Development Series, 2007.
[Emery] Dale H. Emery, "Dimensions of Software Testing", consultado en febrero de 2013 en http://cwd.dhemery.com/2004/04/dimensions/.

[Evans 2003] Eric Evans, "Domain-Driven Design: Tackling Complexity in the Heart of Software", Addison-Wesley Professional, 2003.
[Feathers 2005] Michael Feathers, "Working Effectively with Legacy Code", Prentice Hall, 2005.
[Feathers-2] Michael Feathers, "X Tests are not X Tests", consultado en febrero de 2013 en http://blog.objectmentor.com/articles/2009/04/13/x-tests-are-not-x-tests.
[Fontela 2011] Carlos Fontela, "Estado del arte y tendencias en Test-Driven Development", trabajo de Especialización en Ingeniería de Software, Universidad Nacional de La Plata, 2011.
[Ford 2008] Neal Ford, "The Productive Programmer", O'Reilly Media, 2008.
[Fowler 1999] Martin Fowler, "Refactoring", Addison-Wesley Professional, 1999.
[Fowler 2003] Martin Fowler, "Patterns of Enterprise Application Architecture", Addsion-Wesley, 2003.
[Freeman 2010] Steve Freeman, Nat Pryce, "Growing Object-Oriented Software, Guided by Tests", Addison-Wesley Professional, 2010.
[Gall 1977] John Gall, "Systemantics: How systems work and especially how they fail", Quadrangle, 1977.
[Gamma 1994] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional, 1994.
[Garrido 2000] Alejandra Garrido, "Software refactoring applied to C programming language", PhD Thesis, Universidad de Illinois en Urbana-Champaign, 2000.
[Garrido 2003] Alejandra Garrido y Ralph Johnson, "Refactoring C with conditional compilation, 18th IEEE International Conference on Automated Software Engineering, 2003.
[Garrido 2005] Alejandra Garrido y Ralph Johnson, "Program refactoring in the presence of preprocessor directives", Universidad de Illinois en Urbana-Champaign, 2005.
[Harold 2008] Elliotte Rusty Harold, "Refactoring HTML: Improving the Design of Existing Web Applications", Addison-Wesley Professional, 2008.
[Havenstein 2002] Andreas Havenstein y Stefan Roock, "Refactoring Tags for Automatic Refactoring of Framework", Proceedings of Extreme Programming Conference, 2002.
[Iwamoto 2003] M. Iwamoto y J. Zhao, "Refactoring aspect-oriented programs", 4th AOSD Modeling With UML Workshop, UML, 2003.
[Johnson 1993] Ralph E. Johnson y William F. Opdyke, "Refactoring and Aggregation", Proceedings of International Symposium on Object Techniques for Advanced Software (ISOTAS '93), 1993.
[Kerievsky 2005] Joshua Kerievsky, "Refactoring to Patterns", Addison-Wesley Professional, 2005.

[Larman 2003] Craig Larman, "UML y patrones", Pearson Prentice-Hall, 2003.
[Lehman 1974] Meir M. Lehman, "Programs, Cities, Students, Limits to Growth?", Inaugural Lecture, May 1974, publicado en Programming Methodology, (D Gries ed.), Springer, Verlag, 1978, pp. 42 – 62.
[Lehman 1996] Meir M. Lehman, "Laws of Software Evolution Revisited", Proceedings of the 5th European Workshop on Software Process Technology (EWSPT '96), Springer-Verlag, 1996.
[Linand 1996] Y. Linand, S. Reiss, "Configuration management with logical structures", Proceedings of International Conference on Software Engineering (ICSE'96), 1996.
[Lippert 2006] Martin Lippert y Stephen Roock, "Refactoring in Large Software Projects", John Wiley & Sons, 2006.
[Marick 2002] Brian Marick, "Bypassing the GUI", Software Testing and Quality Engineering Magazine, septiembre-octubre de 2002 (Vol. 4, Issue 5).
[Martin 2002] Robert Martin, "Agile Software Development. Principles, Patterns, and Practices", Prentice-Hall, 2002.
[Martin 2008] Robert C. Martin, "Clean Code. A Handbook of Agile Software Craftmanship", Prentice Hall, 2008.
[Méndez 2011] Mariano Méndez, "Fortran Refactoring for Legacy Systems", tesis de Magíster en Ingeniería de Software, Universidad Nacional de La Plata, 2011.
[Meszaros 2003] Gerard Meszaros, Ralph Bohnet, Jennitta Andrea, "Agile Regression Testing Using Record & Playback", presentado en OOPSLA 2003.
[Meszaros 2007] Gerard Meszaros, "xUnit Test Patterns: Refactoring Test Code", Addison-Wesley Professional, 2007.
[Meyer 2000] Bertrand Meyer, "Object-Oriented Software Construction",Prentice Hall, 2000.
[Monteiro 2005] M.P. Monteiro y J.M. Fernandes, "Towards a catalog of aspect-oriented refactorings", Proceedings of the 4th international conference on Aspect-oriented software development, page122, ACM, 2005.
[Mugridge 2005] Rick Mugridge y Ward Cunningham, "Fit for Developing Software: Framework for Integrated Tests", Prentice Hall, 2005.
[Mugridge 2008] Rick Mugridge, "Managing Agile Project Requirements with Storytest-Driven Development", IEEE software 25, 68-75, 2008.
[North] Dan North, "What´s in a Story", http://blog.dannorth.net/whats-in-a-story/, consultado en febrero de 2013.
[Opdyke 1990] William F. Opdyke, Ralph E. Johnson, "Refactoring: An Aid in Designing Application Frameworks and Evolving Object-Oriented Systems", Proceedings of the

Symposium on Object Oriented Programming Emphasizing Practical Applications (SOOPPA), septiembre de 1990, ACM.
[Opdyke 1992] William Opdyke, "Refactoring Object-Oriented Frameworks", tesis de PhD, Universidad de Illinois en Urbana-Champaign, 1992.
[Pacheco 2007] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, Thomas Ball, "Feedback-directed random test generation", Proceedings of the 29th International Conference on Software Engineering, Minneapolis, 2007 (ICSE 2007), IEEE Computer Society.
[Pipka 2002] Jens Uwe Pipka, "Refactoring in a 'Test-First' World", Proceedings of Extreme Programming Conference, 2002.
[Pipka-2] Jens Uwe Pipka, "Development Upside Down: Following the Test First Trail", Daedalos Consulting GmbH, consultado en febrero de 2013 en http://www.jup-net.de/publications/development_upside_down.pdf .
[Rainsberger] J. B. Rainsberger, "Use Your Singletons Wisely", IBM Developers Work, consultado en febrero de 2013 en http://www.ibm.com/developerworks/library/co-single/index.html en enero de 2013.
[Roberts 1999] Donald Bradley Roberts, "Practical Analysis for Refactoring", tesis de PhD en Computer Science, Universidad de Illinois en Urbana-Champaign, 1999.
[Robles 2010] Esteban Robles Luna, Irene Garrigós, Gustavo Rossi, "Capturing and Validating Personalization Requirements in Web Applications", Proceedings of the 1st Workshop on The Web and Requirements Engineering (WeRE 2010).
[Robles 2010-2] Esteban Robles Luna, Juan Burella, Julián Grigera, Gustavo Rossi, "A Flexible Tool Suite for Change-Aware Test-Driven Development of Web Applications", Proceedings of the ACM/IEEE 32nd International Conference on Software Engineering (ICSE 2010).
[Rura 2003] S. Rura, "Refactoring aspect-oriented software", PhD Thesis, Williams College, 2003.
[Schäfer 2010] Max Schäfer y Oege de Moor, "Specifying and implementing refactorings", Proceedings of the ACM international conference on Object oriented programming systems languages and applications (OOPSLA '10), 2010.
[Shaw 1996] Mary Shaw, David Garlan, "Software Architecture: Perspectives on an Emerging Discipline", Prentice Hall, 1996.
[Stewart] Simon Stewart, "Page Objects", http://code.google.com/p/selenium/wiki/PageObjects, consultado en febrero de 2013.
[UML] "The UML Resource Page", consultado en febrero de 2013 en http://www.uml.org/.

9 Anexos

9.1 Anexo A: Algunos olores usuales que se evidenciaron en el caso de estudio

9.1.1 Problemas del patrón de diseño Singleton y posibles soluciones vía refactorización

El patrón de diseño Singleton [Gamma 1994] se utiliza cuando se desea que una clase tenga una única instancia, dando acceso global a esa instancia única. A pesar de su popularidad y su sencillez de implementación, Singleton ha sido muy criticado, al punto que muchos autores lo consideran un antipatrón, o al menos un mal olor en el código [Ford 2008, Rainsberger].

A modo de ejemplo, a continuación se muestra la implementación de un Singleton en Java:

```
import java.util.*;

public class EmpresaSingleton {

	// el constructor privado garantiza que no se creen instancias adicionales:
	private EmpresaSingleton ( ) {
		empleados = new ArrayList<Empleado> ();
		sectores = new ArrayList<Sector> ();
		clientes = new ArrayList<Empresa> ();
		proveedores = new ArrayList<Empresa> ();
	}

	// "instancia" es la referencia a la única instancia:
	private static EmpresaSingleton instancia = null;

	// el acceso a la instancia será "EmpresaSingleton.getInstancia()":
	public static EmpresaSingleton getInstancia ( ) {
		// se crea la instancia recién cuando se la necesita:
		if (instancia == null)
			instancia = new EmpresaSingleton();
		return instancia;
	}

	// responsabilidades propias de la clase:
```

```
	private String cuit;
	private String nombre;
	private Collection<Empleado> empleados;
	private Collection<Sector> sectores;
	private Collection<Empresa> clientes;
	private Collection<Empresa> proveedores;

	public String getCuit() {
		return cuit;
	}
	public void setCuit(String cuit) {
		this.cuit = cuit;
	}

	public String getNombre() {
		return nombre;
	}
	public void setNombre(String nombre) {
		this.nombre = nombre;
	}

	public boolean estaEmpleado (Empleado e) {
		return (empleados.contains(e));
	}
	public Empleado buscarEmpleadoPorDni (int dni) {
		for (Empleado e : empleados)
			if (e.getDni() == dni)
				return e;
		return null;
	}
	public void agregarEmpleado (Empleado e) {
		empleados.add(e);
	}
	public void eliminarEmpleado (Empleado e) {
		empleados.remove(e);
	}

	// sigue la funcionalidad de la clase...


}
```

Lo cierto es que Singleton provoca algunos problemas en la calidad del código, independientemente de cómo esté implementado:

- Por definición, la propia clase mantiene dos responsabilidades: las que le corresponden como entidad de dominio y las que se ocupan de que sólo exista una instancia de la propia clase. Esto hace que se viole el principio de única responsabilidad [Martin 2002].
- También por definición, limita el número de instancias en la propia clase, lo cual la hace muy difícil de modificar si en algún momento se necesitan más instancias.
- Oculta las dependencias del diseño, ya que al proveer acceso a una instancia global, todos los demás objetos pueden estar dependiendo de la instancia y de la clase, sin que eso sea obvio en el código.
- Debido a que se trata de una clase y una instancia de alcance global, se hace prácticamente imposible de aislar, por ejemplo para realizarle pruebas unitarias.

Por otro lado, la implementación típica en cada lenguaje tiene también algunas desventajas, que en el caso de Java son:

- Se hace imposible extender la clase por herencia, ya que los métodos y atributos estáticos no se heredan.
- Los métodos estáticos y privados complican su uso en inyección de dependencias.
- Si se difiere la inicialización hasta el momento del primer uso (implementación típica), el método de acceso a la instancia debe implementarse con exclusión mutua para evitar problemas de concurrencia que llevarían a que pudiese haber más de una instancia.
- Genera problemas de claridad de código, con indirecciones poco evidentes, como todas las que tienen el esquema: *Clase.getInstance().operacion()*.
- Desde su creación en tiempo de ejecución, un Singleton está siempre vivo, por lo cual el recolector de basura no puede disponer de esa memoria aunque no se esté usando. Por definición, esto es una fuga de memoria o *memory leak*.

Si se desea refactorizar el código de modo tal de eliminar el uso del patrón Singleton se pueden seguir los siguientes pasos, de acuerdo la refactorización que Cunningham [Cunningham c2] denomina Abstract Singleton:

- Crear una interfaz de la clase en cuestión.
- Implementar la clase de instancia única como una implementación particular de la interfaz.

- Pasar la instancia del Singleton a cada objeto mediante su constructor y guardar la referencia en un atributo.
- Separar la responsabilidad de la creación de la instancia en un Factory Method [Gamma 1994] en otra clase y limitar la cardinalidad dentro del Factory Method.
- Tratar de minimizar las llamadas a *getInstance*, idealmente llegando a una sola en todo el código: esa llamada debería estar en algún objeto impulsor de la aplicación. Como corolario, probablemente la aplicación misma deba ser el Singleton.

Una vez realizada una refactorización así, la clase tendrá sus propias responsabilidades separadas de la creación de instancias, pudiendo reutilizarse en más de un contexto, los que necesitan una sola instancia y los que no.

9.1.2 Problemas con las listas de parámetros y una posible solución

Hay dos situaciones en que las listas de parámetros se pueden volver olores:

- Muchos parámetros
- Parámetros muy heterogéneos

Un autor que le dedica atención especial a las listas de parámetros es Martin [Martin 2008]. Según él dice, el número ideal de parámetros es cero, en algunos casos uno, en pocos dos y sólo en casos excepcionales tres.
Hay varias razones para evitar las largas listas de parámetros. Una de ellas es la legibilidad, que empeora cuando hay varios parámetros de los cuales hay que comprender lo que representan. Otro es el mayor acoplamiento que implican la presencia de una mayor cantidad de parámetros en un método. Finalmente, y no menos importante, el número de combinaciones de valores que hay que probar para garantizar la calidad de un método crece más que linealmente con la cantidad de parámetros.
La otra cuestión es la de la heterogeneidad. En este caso, el problema mayor se presenta por la dificultad de lectura de un método cuyos parámetros se encuentran en distintos niveles de abstracción.

En principio, en el paradigma de orientación a objetos se presume que el método correspondiente a un mensaje actúa sobre un objeto receptor de éste. En los casos típicos, el resultado es el cambio de estado del receptor, el cálculo de un valor a partir del mismo o la construcción de otro objeto. No es deseable que se alteren otros objetos ni que se devuelvan valores mediante parámetros.

Hay varias soluciones a estos problemas, que en el catálogo de olores de Fowler [Fowler 1999] estarían enmarcados en *Long Parameter List* y *Data Clumps*. Sin embargo, en el marco de esta tesis nos interesa solamente la solución que el mismo Fowler llama denomina *Introduce Parameter Object*.

En efecto, en el marco del olor *Data Clumps*, Fowler explica que suele haber grupos de objetos que aparecen a menudo a la vez, y en este caso recomienda crear una nueva clase con dichos objetos.

Por ejemplo, en un sistema puede haber varios métodos que cuenten con parámetros *fechaInicial* y *fechaFinal*, que podrían encapsularse en un objeto *Periodo*. Lo mismo ocurriría con parámetros *coordenadaX* y *coordenadaY*, que se podrían englobar en la clase *Punto2D*.

En nuestro caso de estudio había constructores que tomaban tres parámetros *celda1*, *celda2* y *celda3*, y que resolvimos creando un arreglo de objetos *Celda*.

Anexo B: código completo de los ejemplos

9.1.3 Código del ejemplo del capítulo 4: refactorización a Strategy

El código de la clase *Factura* y la enumeración *TipoCliente*, antes de la refactorización, es:

```
package carlosfontela.facturacion;

public enum TipoCliente {
	EMPLEADO,
	JUBILADO,
	COMUN
}
```

```
package carlosfontela.facturacion;

import java.util.*;

public class Factura {

	private static int ultimoNumero = 0;
	private int numero;
	private TipoCliente tipoCliente;
	private Collection <ItemFactura> items;
	private int descuentoEspecialPorcentaje;
```

```
private int descuentoEspecialCantidad;

public class ItemFactura {
        private long codigo;
        private int cantidad;
        private long precio;
        public ItemFactura (long codigo, int cantidad, long precio) {
                this.codigo = codigo;
                this.cantidad = cantidad;
                this.precio = precio;
        }
        public long getCodigo() {
                return codigo;
        }
        public int getCantidad() {
                return cantidad;
        }
        public long getPrecio() {
                return precio;
        }
}

public Factura ( ) {
        ultimoNumero++;
        this.numero = ultimoNumero;
        this.items = new ArrayList<ItemFactura> ( );
        this.tipoCliente = TipoCliente.COMUN;
        this.descuentoEspecialPorcentaje = 0;
        this.descuentoEspecialCantidad = 0;
}

public void agregarItem (long codigo, int cantidad, long precio) {
        items.add(new ItemFactura (codigo, cantidad, precio) );
}

public Collection<ItemFactura> getItems ( ) {
        return this.items;
}

public void setTipoCliente (TipoCliente tipo) {
        this.tipoCliente = tipo;
}
```

```
public void setTipoDescuentoEspecial (int porcentaje, int cantidadIguales) {
        this.descuentoEspecialCantidad = cantidadIguales;
        this.descuentoEspecialPorcentaje = porcentaje;
}

public long total ( ) {
        if ( (tipoCliente == TipoCliente.COMUN) && (descuentoEspecialCantidad > 0) )
                return totalDescuentoEspecial (descuentoEspecialPorcentaje,
                                                descuentoEspecialCantidad);
        else if (tipoCliente == TipoCliente.JUBILADO)
                return totalJubilado();
        else if (tipoCliente == TipoCliente.EMPLEADO)
                return totalEmpleado();
        else return totalComun();
}

private long totalComun() {
        long total = 0;
        for (ItemFactura item : this.getItems()) {
                total += item.getCantidad() * item.getPrecio();
        }
        return total;
}

private long totalJubilado() {
        int porcentaje = 20;
        long total = 0;
        for (ItemFactura item : this.getItems()) {
                long itemSinDescuento = item.getCantidad() * item.getPrecio();
                total += (long)(itemSinDescuento * (1-Math.floor(porcentaje / 100.0)));
        }
        return total;
}

private long totalEmpleado() {
        ArrayList<Long> codigosConDescuento = new ArrayList<Long>();
        codigosConDescuento.add(new Long(3456));
        codigosConDescuento.add(new Long(7890));
        int porcentaje = 30;

        long total = 0;
        for (ItemFactura item : this.getItems()) {
                long itemSinDescuento = item.getCantidad() * item.getPrecio();
```

```
                        if (codigosConDescuento.contains(item.getCodigo()))
                                total +=
                                        (long)(itemSinDescuento * (1-Math.floor(porcentaje / 100.0)));
                        else
                                total += itemSinDescuento;
                }
                return total;
        }

        private long totalDescuentoEspecial
                        (int descuentoEspecialPorcentaje, int descuentoEspecialCantidad) {
                long total = 0;
                for (ItemFactura item : this.getItems()) {
                        long itemSinDescuento = item.getCantidad() * item.getPrecio();
                        long montoDescontar = 0;
                        if (descuentoEspecialCantidad > 1) {
                        int articulosDescontar = item.getCantidad() % descuentoEspecialCantidad;
                        montoDescontar = (long) (articulosDescontar * item.getPrecio() *
                                        (1-Math.floor(descuentoEspecialPorcentaje / 100.0)));
                        }
                total += itemSinDescuento - montoDescontar;
                }
                return total;
        }
}
```

El código completo de las pruebas sería:

```
package carlosfontela.facturacion.pruebas;

import org.junit.*;
import junit.framework.Assert;
import carlosfontela.facturacion.*;
import java.util.*;

public class PruebaTotalFactura {

    private Factura factura;

    /* método que calcula el total de la factura sin ningún descuento */
    private long totalSinDescuentos ( ) {
        long total = 0;
```

```
        for (Factura.ItemFactura item : factura.getItems()) {
                long itemSinDescuento = item.getCantidad() * item.getPrecio();
                total += itemSinDescuento;
        }
        return total;
}

/* método que calcula el total de la factura con X% de descuento */
private long totalConDescuentoPorcentual (int porcentaje) {
        long total = 0;
        for (Factura.ItemFactura item : factura.getItems()) {
                long itemSinDescuento = item.getCantidad() * item.getPrecio();
                total += (long)(itemSinDescuento * (1-Math.floor(porcentaje / 100.0)));
        }
        return total;
}

/* método que calcula el total de la factura con X% de descuento para ciertos productos */
private long totalConDescuentoPorcentualProductos
                                (int porcentaje, List<Long> codigosConDescuento) {
        long total = 0;
        for (Factura.ItemFactura item : factura.getItems()) {
                long itemSinDescuento = item.getCantidad() * item.getPrecio();
                if (codigosConDescuento.contains(item.getCodigo()))
                        total +=
                                        (long)(itemSinDescuento * (1-Math.floor(porcentaje / 100.0)));
                else
                        total += itemSinDescuento;
        }
        return total;
}

/* método que calcula el total de la factura con X% de descuento para Y artículos iguales */
private long totalConDescuentoPorcentualIguales (int porcentaje, int cantidadIguales) {
        long total = 0;
        for (Factura.ItemFactura item : factura.getItems()) {
                long itemSinDescuento = item.getCantidad() * item.getPrecio();
        long montoDescontar = 0;
                if (cantidadIguales > 1) {
                        int articulosDescontar = item.getCantidad() % cantidadIguales;
                        montoDescontar = (long) (articulosDescontar * item.getPrecio() *
                                                                (1-Math.floor(porcentaje / 100.0)));
                }
```

```
                    total += itemSinDescuento - montoDescontar;
            }
            return total;
    }

    @Before
    public void inicializarFactura ( ) {
            factura = new Factura();
            factura.agregarItem(1234, 2, 1120);
            factura.agregarItem(3456, 3, 2150);
            factura.agregarItem(5678, 1, 4320);
            factura.agregarItem(7890, 4, 1030);
    }

    @Test
    public void sinDescuento ( ) {
            factura.setTipoCliente (TipoCliente.COMUN);
            Assert.assertEquals(totalSinDescuentos(), factura.total());
    }

    @Test
    public void descuento20porcientoJubilados ( ) {
            factura.setTipoCliente (TipoCliente.JUBILADO);
            Assert.assertEquals(totalConDescuentoPorcentual(20), factura.total());
    }

    @Test
    public void descuentoEmpleados30porcientoSobreArticulos3456y7890 ( ) {
            factura.setTipoCliente (TipoCliente.EMPLEADO);
            ArrayList<Long> codigosConDescuento = new ArrayList<Long>();
            codigosConDescuento.add(new Long(3456));
            codigosConDescuento.add(new Long(7890));
            Assert.assertEquals(totalConDescuentoPorcentualProductos (30, codigosConDescuento),
factura.total());
    }

    @Test
    public void descuento20porcientoSegundoArticuloIgual ( ) {
            factura.setTipoCliente (TipoCliente.COMUN);
            factura.setTipoDescuentoEspecial (20, 2);
            Assert.assertEquals(totalConDescuentoPorcentualIguales (20, 2), factura.total());
    }
```

```
}
```

La clase *Factura*, luego de la primera refactorización, queda así (en negrita lo que cambió):

```
package carlosfontela.facturacion;

import java.util.*;

public class Factura {

	private static int ultimoNumero = 0;
	private int numero;
	private TipoCliente tipoCliente;
	private Collection <ItemFactura> items;
	private int descuentoEspecialPorcentaje;
	private int descuentoEspecialCantidad;
	private EstrategiaTotalFactura estrategiaTotal;

	public class ItemFactura {
		private long codigo;
		private int cantidad;
		private long precio;
		public ItemFactura (long codigo, int cantidad, long precio) {
			this.codigo = codigo;
			this.cantidad = cantidad;
			this.precio = precio;
		}
		public long getCodigo() {
			return codigo;
		}
		public int getCantidad() {
			return cantidad;
		}
		public long getPrecio() {
			return precio;
		}
	}

	public Factura ( ) {
		ultimoNumero++;
		this.numero = ultimoNumero;
		this.items = new ArrayList<ItemFactura> ( );
```

```
                this.tipoCliente = TipoCliente.COMUN;
                this.descuentoEspecialPorcentaje = 0;
                this.descuentoEspecialCantidad = 0;
                this.estrategiaTotal = null;
        }

        public void agregarItem (long codigo, int cantidad, long precio) {
                items.add(new ItemFactura (codigo, cantidad, precio) );
        }

        public Collection<ItemFactura> getItems ( ) {
                return this.items;
        }

        public void setTipoCliente (TipoCliente tipo) {
                this.tipoCliente = tipo;
        }

        public void setTipoDescuentoEspecial (int porcentaje, int cantidadIguales) {
                this.descuentoEspecialCantidad = cantidadIguales;
                this.descuentoEspecialPorcentaje = porcentaje;
        }

        public int getDescuentoEspecialPorcentaje() {
                return descuentoEspecialPorcentaje;
        }

        public int getDescuentoEspecialCantidad() {
                return descuentoEspecialCantidad;
        }

        public long total ( ) {
                if ( (tipoCliente == TipoCliente.COMUN) && (descuentoEspecialCantidad > 0) )
                        this.estrategiaTotal = new EstrategiaPorCantidad( );
                else if (tipoCliente == TipoCliente.JUBILADO)
                        this.estrategiaTotal = new EstrategiaJubilados( );
                else if (tipoCliente == TipoCliente.EMPLEADO)
                        this.estrategiaTotal = new EstrategiaEmpleados( );
                else this.estrategiaTotal = new EstrategiaComun( );
                return estrategiaTotal.totalFactura(this);
        }

}
```

La nueva interfaz y las clases de estrategias son:

```
package carlosfontela.facturacion;

public interface EstrategiaTotalFactura {
	public long totalFactura (Factura factura);
}
```

```
package carlosfontela.facturacion;

import carlosfontela.facturacion.Factura.ItemFactura;

public class EstrategiaComun implements EstrategiaTotalFactura {

	@Override
	public long totalFactura (Factura factura) {
		long total = 0;
		for (ItemFactura item : factura.getItems()) {
			total += item.getCantidad() * item.getPrecio();
		}
		return total;
	}

}
```

```
package carlosfontela.facturacion;

import carlosfontela.facturacion.Factura.ItemFactura;

public class EstrategiaJubilados implements EstrategiaTotalFactura {

	@Override
	public long totalFactura (Factura factura) {
		int porcentaje = 20;
		long total = 0;
		for (ItemFactura item : factura.getItems()) {
			long itemSinDescuento = item.getCantidad() * item.getPrecio();
			total += (long)(itemSinDescuento * (1-Math.floor(porcentaje / 100.0)));
		}
		return total;
```

```
	}

}
```

```
package carlosfontela.facturacion;

import java.util.ArrayList;

import carlosfontela.facturacion.Factura.ItemFactura;

public class EstrategiaEmpleados implements EstrategiaTotalFactura {

	@Override
	public long totalFactura (Factura factura) {
		ArrayList<Long> codigosConDescuento = new ArrayList<Long>();
		codigosConDescuento.add(new Long(3456));
		codigosConDescuento.add(new Long(7890));
		int porcentaje = 30;

		long total = 0;
		for (ItemFactura item : factura.getItems()) {
			long itemSinDescuento = item.getCantidad() * item.getPrecio();
			if (codigosConDescuento.contains(item.getCodigo()))
				total +=
					(long)(itemSinDescuento * (1-Math.floor(porcentaje / 100.0)));
			else
				total += itemSinDescuento;
		}
		return total;
	}

}
```

```
package carlosfontela.facturacion;

import carlosfontela.facturacion.Factura.ItemFactura;

public class EstrategiaPorCantidad implements EstrategiaTotalFactura {

	@Override
	public long totalFactura (Factura factura) {
		long total = 0;
```

```
                for (ItemFactura item : factura.getItems()) {
                        long itemSinDescuento = item.getCantidad() * item.getPrecio();
                long montoDescontar = 0;
                        if (factura.getDescuentoEspecialCantidad() > 1) {
                                int articulosDescontar = item.getCantidad() %
                                                factura.getDescuentoEspecialCantidad();
                                montoDescontar = (long) (articulosDescontar * item.getPrecio() *
                                (1-Math.floor(factura.getDescuentoEspecialPorcentaje() / 100.0)));
                        }
                        total += itemSinDescuento - montoDescontar;
                }
                return total;
        }

}
```

Al hacer la segunda etapa de la refactorización, llegamos a la clase *Factura* que sigue (en negrita lo que cambió respecto de la anterior):

```
package carlosfontela.facturacion;

import java.util.*;

public class Factura {

        private static int ultimoNumero = 0;
        private int numero;
        private Collection <ItemFactura> items;
        private int descuentoEspecialPorcentaje;
        private int descuentoEspecialCantidad;
        private EstrategiaTotalFactura estrategiaTotal;

        public class ItemFactura {
                private long codigo;
                private int cantidad;
                private long precio;
                public ItemFactura (long codigo, int cantidad, long precio) {
                        this.codigo = codigo;
                        this.cantidad = cantidad;
                        this.precio = precio;
                }
                public long getCodigo() {
```

```
                return codigo;
        }
        public int getCantidad() {
                return cantidad;
        }
        public long getPrecio() {
                return precio;
        }
}

public Factura ( ) {
        ultimoNumero++;
        this.numero = ultimoNumero;
        this.items = new ArrayList<ItemFactura> ( );
        this.descuentoEspecialPorcentaje = 0;
        this.descuentoEspecialCantidad = 0;
        this.estrategiaTotal = null;
}

public void agregarItem (long codigo, int cantidad, long precio) {
        items.add(new ItemFactura (codigo, cantidad, precio) );
}

public Collection<ItemFactura> getItems ( ) {
        return this.items;
}

public EstrategiaTotalFactura getEstrategiaTotal ( ) {
        return estrategiaTotal;
}

public void setEstrategiaTotal (EstrategiaTotalFactura estrategiaTotal) {
        this.estrategiaTotal = estrategiaTotal;
}

public void setTipoDescuentoEspecial (int porcentaje, int cantidadIguales) {
        this.descuentoEspecialCantidad = cantidadIguales;
        this.descuentoEspecialPorcentaje = porcentaje;
}

public int getDescuentoEspecialPorcentaje() {
        return descuentoEspecialPorcentaje;
}
```

```
	public int getDescuentoEspecialCantidad() {
		return descuentoEspecialCantidad;
	}

	public long total ( ) {
		return estrategiaTotal.totalFactura(this);
	}

}
```

Y la clase de pruebas modificada, completa, es:

```
package carlosfontela.facturacion.pruebas;

import org.junit.*;
import junit.framework.Assert;
import carlosfontela.facturacion.*;
import java.util.*;

public class PruebaTotalFactura {

    private Factura factura;

    /* método que calcula el total de la factura sin ningún descuento */
    private long totalSinDescuentos ( ) {
        long total = 0;
        for (Factura.ItemFactura item : factura.getItems()) {
            long itemSinDescuento = item.getCantidad() * item.getPrecio();
            total += itemSinDescuento;
        }
        return total;
    }

    /* método que calcula el total de la factura con X% de descuento */
    private long totalConDescuentoPorcentual (int porcentaje) {
        long total = 0;
        for (Factura.ItemFactura item : factura.getItems()) {
            long itemSinDescuento = item.getCantidad() * item.getPrecio();
            total += (long)(itemSinDescuento * (1-Math.floor(porcentaje / 100.0)));
        }
        return total;
```

```
}

/* método que calcula el total de la factura con X% de descuento para ciertos productos */
private long totalConDescuentoPorcentualProductos
                                (int porcentaje, List<Long> codigosConDescuento) {
        long total = 0;
        for (Factura.ItemFactura item : factura.getItems()) {
                long itemSinDescuento = item.getCantidad() * item.getPrecio();
                if (codigosConDescuento.contains(item.getCodigo()))
                        total +=
                                        (long)(itemSinDescuento * (1-Math.floor(porcentaje / 100.0)));
                else
                        total += itemSinDescuento;
        }
        return total;
}

/* método que calcula el total de la factura con X% de descuento para Y artículos iguales */
private long totalConDescuentoPorcentualIguales (int porcentaje, int cantidadIguales) {
        long total = 0;
        for (Factura.ItemFactura item : factura.getItems()) {
                long itemSinDescuento = item.getCantidad() * item.getPrecio();
        long montoDescontar = 0;
                if (cantidadIguales > 1) {
                        int articulosDescontar = item.getCantidad() % cantidadIguales;
                        montoDescontar = (long) (articulosDescontar * item.getPrecio() *
                                                                (1-Math.floor(porcentaje / 100.0)));
                }
                total += itemSinDescuento - montoDescontar;
        }
        return total;
}

@Before
public void inicializarFactura ( ) {
        factura = new Factura();
        factura.agregarItem(1234, 2, 1120);
        factura.agregarItem(3456, 3, 2150);
        factura.agregarItem(5678, 1, 4320);
        factura.agregarItem(7890, 4, 1030);
}

@Test
```

```
    public void sinDescuento ( ) {
        factura.setEstrategiaTotal ( new EstrategiaComun() );
        Assert.assertEquals(totalSinDescuentos(), factura.total());
    }

    @Test
    public void descuento20porcientoJubilados ( ) {
        factura.setEstrategiaTotal ( new EstrategiaJubilados() );
        Assert.assertEquals(totalConDescuentoPorcentual(20), factura.total());
    }

    @Test
    public void descuentoEmpleados30porcientoSobreArticulos3456y7890 ( ) {
        factura.setEstrategiaTotal ( new EstrategiaEmpleados() );
        ArrayList<Long> codigosConDescuento = new ArrayList<Long>();
        codigosConDescuento.add(new Long(3456));
        codigosConDescuento.add(new Long(7890));
        Assert.assertEquals (totalConDescuentoPorcentualProductos (30,
                                                    codigosConDescuento), factura.total());
    }

    @Test
    public void descuento20porcientoSegundoArticuloIgual ( ) {
        factura.setEstrategiaTotal ( new EstrategiaPorCantidad() );
        factura.setTipoDescuentoEspecial (20, 2);
        Assert.assertEquals(totalConDescuentoPorcentualIguales (20, 2), factura.total());
    }

}
```

9.1.4 Código del caso de verificación del refactoring con pruebas de integración (capítulo 6): cambio del constructor de Tablero

El constructor de la clase *Tablero* usando el constructor de *LineaDe3* antes de la refactorización era:

```
        public Tablero ( ) {
                for (int fila = 0; fila < 3; fila++)
                        for (int columna = 0; columna < 3; columna++)
                                this.tablero [fila][columna] = new Celda(fila, columna, null);
                this.lineasDe3[0] = new LineaDe3 ( this.tablero[0][0],
                        this.tablero[1][0], this.tablero[2][0], "línea vertical izquierda" );
```

```
            this.lineasDe3[1] = new LineaDe3 ( this.tablero[0][1],
                    this.tablero[1][1], this.tablero[2][1], "línea vertical central"  );
            this.lineasDe3[2] = new LineaDe3 ( this.tablero[0][2],
                    this.tablero[1][2], this.tablero[2][2], "línea vertical derecha"  );
            this.lineasDe3[3] = new LineaDe3 ( this.tablero[0][0],
                            this.tablero[0][1], this.tablero[0][2], "línea horizontal arriba" );
            this.lineasDe3[4] = new LineaDe3 ( this.tablero[1][0],
                            this.tablero[1][1], this.tablero[1][2], "línea horizontal central" );
            this.lineasDe3[5] = new LineaDe3 ( this.tablero[2][0],
                            this.tablero[2][1], this.tablero[2][2], "línea horizontal abajo" );
            this.lineasDe3[6] = new LineaDe3 ( this.tablero[0][0],
                            this.tablero[1][1], this.tablero[2][2], "línea diagonal directa" );
            this.lineasDe3[7] = new LineaDe3 ( this.tablero[2][0],
                            this.tablero[1][1], this.tablero[0][2], "línea diagonal inversa" );
    }
```

La modificacción que mejor se adecua al cambio en el constructor de *LineaDe3* propuesto, es:

```
    public Tablero ( ) {
            for (int fila = 0; fila < 3; fila++)
                    for (int columna = 0; columna < 3; columna++)
                            this.tablero [fila][columna] = new Celda(fila, columna, null);

            Celda[ ] linea0 = { this.tablero[0][0], this.tablero[1][0], this.tablero[2][0] };
            Celda[ ] linea1 = { this.tablero[0][1], this.tablero[1][1], this.tablero[2][1] };
            Celda[ ] linea2 = { this.tablero[0][2], this.tablero[1][2], this.tablero[2][2] };
            Celda[ ] linea3 = { this.tablero[0][0], this.tablero[0][1], this.tablero[0][2] };
            Celda[ ] linea4 = { this.tablero[1][0], this.tablero[1][1], this.tablero[1][2] };
            Celda[ ] linea5 = { this.tablero[2][0], this.tablero[2][1], this.tablero[2][2] };
            Celda[ ] linea6 = { this.tablero[0][0], this.tablero[1][1], this.tablero[2][2] };
            Celda[ ] linea7 = { this.tablero[2][0], this.tablero[1][1], this.tablero[0][2] };

            this.lineasDe3[0] = new LineaDe3 ( linea0 , "línea vertical izquierda" );
            this.lineasDe3[1] = new LineaDe3 ( linea1, "línea vertical central"  );
            this.lineasDe3[2] = new LineaDe3 ( linea2, "línea vertical derecha"  );
            this.lineasDe3[3] = new LineaDe3 ( linea3, "línea horizontal arriba" );
            this.lineasDe3[4] = new LineaDe3 ( linea4, "línea horizontal central" );
            this.lineasDe3[5] = new LineaDe3 ( linea5, "línea horizontal abajo" );
            this.lineasDe3[6] = new LineaDe3 ( linea6, "línea diagonal directa" );
            this.lineasDe3[7] = new LineaDe3 ( linea7, "línea diagonal inversa" );
    }
```

9.1.5 Código del caso de verificación del refactoring con pruebas de integración (capítulo 6): cambio de las pruebas unitarias

La clase *PruebasLineaDe3* antes del cambio era:

```
package carlosFontela.tateti.dominio;

import junit.framework.Assert;
import org.junit.Before;
import org.junit.Test;

public class PruebasLineaDe3 {

	private LineaDe3 linea;
	private Celda celda0, celda1, celda2;
	private String descripcion = "línea diagonal inversa";
	private Juego juego = new Juego (TipoJuego.JugadorVsJugador);
	private JugadorHumano jugadorX = new JugadorHumano ('X', juego);
	private JugadorHumano jugadorO = new JugadorHumano ('O', juego);

	@Before
	public void precondicionBasica ( ) {
		celda0 = new Celda (2, 0, jugadorX);
		celda1 = new Celda (1, 1, jugadorX);
		celda2 = new Celda (0, 2, jugadorX);
	}

	@Test (expected = IllegalArgumentException.class)
	public void pruebaConstructorCeldaNull ( ) {
		linea = new LineaDe3 (celda0, null, celda2, descripcion);
	}

	@Test (expected = IllegalArgumentException.class)
	public void pruebaConstructorDescripcionNull ( ) {
		linea = new LineaDe3 (celda0, celda1, celda2, null);
	}

	@Test
	public void pruebaConstructorGetDescripcion ( ) {
		linea = new LineaDe3 (celda0, celda1, celda2, descripcion);
		Assert.assertEquals( descripcion, linea.getDescripcion() );
	}
```

```
@Test
public void deberiaEstarCompleta ( ) {
        linea = new LineaDe3 (celda0, celda1, celda2, descripcion);
        Assert.assertTrue( linea.estaCompleta() );
}

@Test
public void noDeberiaEstarCompleta ( ) {
        linea = new LineaDe3 (celda0, new Celda (1, 1, null), celda2, descripcion);
        Assert.assertFalse( linea.estaCompleta() );
}

@Test
public void noDeberiaDarTaTeTiPorDistintosJugadores ( ) {
        linea = new LineaDe3 (celda0, new Celda (1, 1, jugadorO), celda2, descripcion);
        Assert.assertFalse( linea.hayTaTeTi() );
}

@Test
public void noDeberiaDarTaTeTiPorJugadorNull ( ) {
        linea = new LineaDe3 (celda0, new Celda (1, 1, null), celda2, descripcion);
        Assert.assertFalse( linea.hayTaTeTi() );
}

@Test
public void deberiaDarTaTeTi ( ) {
        linea = new LineaDe3 (celda0, celda1, celda2, descripcion);
        Assert.assertTrue( linea.hayTaTeTi() );
}

@Test
public void jugadorEnFilaTaTeTiCuandoHay ( ) {
        linea = new LineaDe3 (celda0, celda1, celda2, descripcion);
        Assert.assertSame( jugadorX, linea.jugadorEnFilaTaTeTi() );
}

@Test
public void jugadorEnFilaTaTeTiCuandoHayDosDistintos ( ) {
        linea = new LineaDe3 (celda0, new Celda (1, 1, jugadorO), celda2, descripcion);
        Assert.assertNull( linea.jugadorEnFilaTaTeTi() );
}

@Test
```

```
public void jugadorEnFilaTaTeTiCuandoHayCeldaSinJugador ( ) {
        linea = new LineaDe3 (celda0, celda1, new Celda (1, 1, null), descripcion);
        Assert.assertNull( linea.jugadorEnFilaTaTeTi() );
}

@Test
public void celdasOcupadasDebeDarCero ( ) {
        linea = new LineaDe3 ( new Celda (2, 0, null), new Celda (1, 1, null),
                                                new Celda (0, 2, null), descripcion);
        Assert.assertEquals( 0, linea.cantidadCeldasOcupadas() );
}

@Test
public void celdasOcupadasDebeDarUno ( ) {
        linea = new LineaDe3 (new Celda (2, 0, null), celda1,
                                                new Celda (0, 2, null), descripcion);
        Assert.assertEquals( 1, linea.cantidadCeldasOcupadas() );
}

@Test
public void celdasOcupadasDebeDarTres ( ) {
        linea = new LineaDe3 (celda0, celda1, celda2, descripcion);
        Assert.assertEquals( 3, linea.cantidadCeldasOcupadas() );
}

@Test
public void probarTaTeTiProximoPositivo ( ) {
        linea = new LineaDe3 (celda0, celda1, new Celda (0, 2, null), descripcion);
        Assert.assertTrue( linea.proximaJugadaPodriaSerTaTeTiDe(jugadorX) );
        Assert.assertFalse( linea.proximaJugadaPodriaSerTaTeTiDe(jugadorO) );
}

@Test
public void probarTaTeTiProximoNegativoHayUno ( ) {
        linea = new LineaDe3 (new Celda (2, 0, null), celda1,
                                        new Celda (0, 2, null), descripcion);
        Assert.assertFalse( linea.proximaJugadaPodriaSerTaTeTiDe(jugadorX) );
        Assert.assertFalse( linea.proximaJugadaPodriaSerTaTeTiDe(jugadorO) );
}

@Test
public void probarTaTeTiProximoNegativoNoHayNinguno ( ) {
        linea = new LineaDe3 (new Celda (2, 0, null), new Celda (1, 1, null),
```

```
                                                    new Celda (0, 2, null), descripcion);
                Assert.assertFalse( linea.proximaJugadaPodriaSerTaTeTiDe(jugadorX) );
                Assert.assertFalse( linea.proximaJugadaPodriaSerTaTeTiDe(jugadorO) );
        }

        @Test
        public void celdaFaltanteParaTaTeTiEsPrimera ( ) {
                Celda primera = new Celda (0, 2, null);
                linea = new LineaDe3 (primera, celda1, celda2, descripcion);
                Assert.assertSame( primera, linea.celdaFaltanteParaTaTeTi(jugadorX) );
                Assert.assertNotSame( celda1, linea.celdaFaltanteParaTaTeTi(jugadorX) );
                Assert.assertNotSame( celda2, linea.celdaFaltanteParaTaTeTi(jugadorX) );
        }

        @Test
        public void celdaFaltanteParaTaTeTiEsUltima ( ) {
                Celda ultima = new Celda (0, 2, null);
                linea = new LineaDe3 (celda0, celda1, ultima, descripcion);
                Assert.assertSame( ultima, linea.celdaFaltanteParaTaTeTi(jugadorX) );
                Assert.assertNotSame( celda0, linea.celdaFaltanteParaTaTeTi(jugadorX) );
                Assert.assertNotSame( celda1, linea.celdaFaltanteParaTaTeTi(jugadorX) );
        }
}
```

Introduciendo los cambios de a una prueba por vez, vamos chequeando que cada una de ellas empieza a funcionar correctamente. El estado final de la clase *PruebasLineaDe3* es (en negrita los cambios realizados):

```
package carlosFontela.tateti.dominio;

import junit.framework.Assert;
import org.junit.Before;
import org.junit.Test;

public class PruebasLineaDe3 {

        private LineaDe3 linea;
        private Celda celda0, celda1, celda2;
        private String descripcion = "línea diagonal inversa";
        private Juego juego = new Juego (TipoJuego.JugadorVsJugador);
        private JugadorHumano jugadorX = new JugadorHumano ('X', juego);
```

```
	private JugadorHumano jugadorO = new JugadorHumano ('O', juego);

	@Before
	public void precondicionBasica ( ) {
		celda0 = new Celda (2, 0, jugadorX);
		celda1 = new Celda (1, 1, jugadorX);
		celda2 = new Celda (0, 2, jugadorX);
	}

	@Test (expected = IllegalArgumentException.class)
	public void pruebaConstructorCeldaNull ( ) {
		Celda [ ] celdas = { celda0, null, celda2 };
		linea = new LineaDe3 ( celdas, descripcion );
	}

	@Test (expected = IllegalArgumentException.class)
	public void pruebaConstructorDescripcionNull ( ) {
		Celda [ ] celdas = { celda0, celda1, celda2 };
		linea = new LineaDe3 ( celdas, null );
	}

	@Test
	public void pruebaConstructorGetDescripcion ( ) {
		Celda [ ] celdas = { celda0, celda1, celda2 };
		linea = new LineaDe3 ( celdas, descripcion );
		Assert.assertEquals( descripcion, linea.getDescripcion() );
	}

	@Test
	public void deberiaEstarCompleta ( ) {
		Celda [ ] celdas = { celda0, celda1, celda2 };
		linea = new LineaDe3 ( celdas, descripcion );
		Assert.assertTrue( linea.estaCompleta() );
	}

	@Test
	public void noDeberiaEstarCompleta ( ) {
		Celda [ ] celdas = { celda0, new Celda (1, 1, null), celda2 };
		linea = new LineaDe3 ( celdas, descripcion );
		Assert.assertFalse( linea.estaCompleta() );
	}

	@Test
```

```
public void noDeberiaDarTaTeTiPorDistintosJugadores ( ) {
        Celda [ ] celdas = { celda0, new Celda (1, 1, jugadorO), celda2 };
        linea = new LineaDe3 ( celdas, descripcion );
        Assert.assertFalse( linea.hayTaTeTi() );
}

@Test
public void noDeberiaDarTaTeTiPorJugadorNull ( ) {
        Celda [ ] celdas = { celda0, new Celda (1, 1, null), celda2 };
        linea = new LineaDe3 ( celdas, descripcion );
        Assert.assertFalse( linea.hayTaTeTi() );
}

@Test
public void deberiaDarTaTeTi ( ) {
        Celda [ ] celdas = { celda0, celda1, celda2 };
        linea = new LineaDe3 ( celdas, descripcion );
        Assert.assertTrue( linea.hayTaTeTi() );
}

@Test
public void jugadorEnFilaTaTeTiCuandoHay ( ) {
        Celda [ ] celdas = { celda0, celda1, celda2 };
        linea = new LineaDe3 ( celdas, descripcion );
        Assert.assertSame( jugadorX, linea.jugadorEnFilaTaTeTi() );
}

@Test
public void jugadorEnFilaTaTeTiCuandoHayDosDistintos ( ) {
        Celda [ ] celdas = { celda0, new Celda (1, 1, jugadorO), celda2 };
        linea = new LineaDe3 ( celdas, descripcion );
        Assert.assertNull( linea.jugadorEnFilaTaTeTi() );
}

@Test
public void jugadorEnFilaTaTeTiCuandoHayCeldaSinJugador ( ) {
        Celda [ ] celdas = { celda0, celda1, new Celda (1, 1, null) };
        linea = new LineaDe3 ( celdas, descripcion );
        Assert.assertNull( linea.jugadorEnFilaTaTeTi() );
}

@Test
public void celdasOcupadasDebeDarCero ( ) {
```

```
        Celda [ ] celdas = { new Celda (2, 0, null), new Celda (1, 1, null),
                                                    new Celda (0, 2, null) };
        linea = new LineaDe3 ( celdas, descripcion );
        Assert.assertEquals( 0, linea.cantidadCeldasOcupadas() );
    }

    @Test
    public void celdasOcupadasDebeDarUno ( ) {
        Celda [ ] celdas = { new Celda (2, 0, null), celda1, new Celda (0, 2, null) };
        linea = new LineaDe3 ( celdas, descripcion );
        Assert.assertEquals( 1, linea.cantidadCeldasOcupadas() );
    }

    @Test
    public void celdasOcupadasDebeDarTres ( ) {
        Celda [ ] celdas = { celda0, celda1, celda2 };
        linea = new LineaDe3 ( celdas, descripcion );
        Assert.assertEquals( 3, linea.cantidadCeldasOcupadas() );
    }

    @Test
    public void probarTaTeTiProximoPositivo ( ) {
        Celda [ ] celdas = { celda0, celda1, new Celda (0, 2, null) };
        linea = new LineaDe3 ( celdas, descripcion );
        Assert.assertTrue( linea.proximaJugadaPodriaSerTaTeTiDe(jugadorX) );
        Assert.assertFalse( linea.proximaJugadaPodriaSerTaTeTiDe(jugadorO) );
    }

    @Test
    public void probarTaTeTiProximoNegativoHayUno ( ) {
        Celda [ ] celdas = { new Celda (2, 0, null), celda1, new Celda (0, 2, null) };
        linea = new LineaDe3 ( celdas, descripcion );
        Assert.assertFalse( linea.proximaJugadaPodriaSerTaTeTiDe(jugadorX) );
        Assert.assertFalse( linea.proximaJugadaPodriaSerTaTeTiDe(jugadorO) );
    }

    @Test
    public void probarTaTeTiProximoNegativoNoHayNinguno ( ) {
        Celda [ ] celdas = { new Celda (2, 0, null), new Celda (1, 1, null),
                                                    new Celda (0, 2, null) };
        linea = new LineaDe3 ( celdas, descripcion );
        Assert.assertFalse( linea.proximaJugadaPodriaSerTaTeTiDe(jugadorX) );
        Assert.assertFalse( linea.proximaJugadaPodriaSerTaTeTiDe(jugadorO) );
```

```
	}

	@Test
	public void celdaFaltanteParaTaTeTiEsPrimera ( ) {
		Celda primera = new Celda (0, 2, null);
		Celda [ ] celdas = { primera, celda1, celda2 };
		linea = new LineaDe3 ( celdas, descripcion );
		Assert.assertSame( primera, linea.celdaFaltanteParaTaTeTi(jugadorX) );
		Assert.assertNotSame( celda1, linea.celdaFaltanteParaTaTeTi(jugadorX) );
		Assert.assertNotSame( celda2, linea.celdaFaltanteParaTaTeTi(jugadorX) );
	}

	@Test
	public void celdaFaltanteParaTaTeTiEsUltima ( ) {
		Celda ultima = new Celda (0, 2, null);
		Celda [ ] celdas = { celda0, celda1, ultima };
		linea = new LineaDe3 ( celdas, descripcion );
		Assert.assertSame( ultima, linea.celdaFaltanteParaTaTeTi(jugadorX) );
		Assert.assertNotSame( celda0, linea.celdaFaltanteParaTaTeTi(jugadorX) );
		Assert.assertNotSame( celda1, linea.celdaFaltanteParaTaTeTi(jugadorX) );
	}
}
```