Universidad de Buenos Aires
Facultad De Ingeniería

Análisis de impacto de cambios realizados sobre sistemas con pruebas automatizadas con distinta granularidad

Nicolás Dascanio

Director: **Ing. Carlos Fontela**

Febrero 2014

Resumen

Test Driven Development (TDD) o desarrollo guiado por pruebas es una técnica de diseño e implementación de software propuesta en el esquema de Extreme Programming (XP) que ha ido ganando popularidad en los últimos años. En sus comienzos se usaban sólo pruebas unitarias, lo que se conoce como Unit-Test Driven Development (UTDD). Más adelante se incorporaron las pruebas de aceptación como puntapié inicial del ciclo de TDD, lo que se conoce como Acceptance-Test Driven Development (ATDD). Hoy en día es necesario contar con una metodología de desarrollo que esté preparada para realizar cambios en forma segura, ya que sin mantenimiento los sistemas se vuelven obsoletos. En este estudio se muestra que ATDD es un enfoque más adecuado para el desarrollo de software. Esto se debe a que ATDD contiene pruebas de distinta granularidad, lo que permite realizar cambios en forma segura. Se analizarán proyectos con distintos niveles de pruebas, mostrando que ATDD está mejor preparado que UTDD, ya que posee pruebas de mayor granularidad que son mucho más estables que las pruebas unitarias, dando un entorno más seguro para realizar cambios.

Agradecimientos

En primer lugar me gustaría agradecer a las personas de los distintos proyectos que se tomaron el tiempo de completar la encuesta que fue necesaria para completar la investigación. En particular quiero agradecer a Robert Martin y Rod Hilton por la rápida respuesta y por apuntarme en la dirección correcta a la hora de conseguir proyectos.

Por otro lado me gustaría agradecer a mi tutor Carlos Fontela por introducirme en el mundo de TDD y por haberme ayudado durante el transcurso de toda la tesis, brindándome información, consejo y correcciones. Gracias por el tiempo y dedicación y por haber estado cada vez que fue necesario.

Agradezco a mis padres por la educación que me dieron y por haberme dado la oportunidad de estudiar esta carrera. Gracias por el apoyo continuo y por asegurarse que nunca me haya faltado nada.

Por último quiero agradecerle a Gil, por haber estado siempre durante la mayor parte de mi carrera, por haber soportado trabajos prácticos, parciales y finales. Gracias por todo el tiempo y paciencia.

Índice general

Índice de figuras

1. Introducción

En los comienzos de la historia del desarrollo de software, los programas desarrollados eran muy sencillos y resolvían problemas muy puntuales, normalmente relacionados con el ámbito científico. Las limitadas capacidades de las máquinas y la complejidad del lenguaje eran un gran limitante de la magnitud de los problemas a resolver.

Con el paso del tiempo, las máquinas empezaron a ser más potentes, más chicas y más económicas, llegando al punto de ser accesibles para cualquier persona y no sólo para grandes entidades.

> En 1971 Intel lanza al mercado el Intel 4004, el primer microprocesador en un solo chip. Tenía 2300 transistores de 10 μm y ejecutaba 0.07 millones de instrucciones por segundo. En 2004 Intel lanza el procesador Pentium 4E, con 170 millones de transistores de 0.09 μm (90 nm) y capaz de ejecutar 11000 millones de instrucciones por segundo.
>
> Dicho de otra manera, en cerca de 30 años los procesadores se hicieron 160000 veces más rápidos y 100 veces más chicos.

Con el desarrollo de máquinas más potentes, se empezaron a desarrollar programas más grandes y complejos, al mismo tiempo que los lenguajes de programación evolucionaban. Para poder lograr este objetivo, se crearon metodologías que permitiesen ordenar el desarrollo. Una de las primeras y que más éxito tuvo fue el modelo en cascada.

1.1. El modelo de desarrollo en cascada

El modelo en cascada propone distintas etapas bien marcadas para el proceso de desarrollo. Estas etapas están bien definidas, y una etapa debe terminar para que pueda comenzar la siguiente. La figura 1.1 muestra un esquema típico.

Como se ve en la figura, cuando se detecta un error en una etapa, se debe volver a la etapa donde se cometió el mismo. Cuanto más grande la brecha entre que se comete el error y se detecta, más costoso será resolverlo.

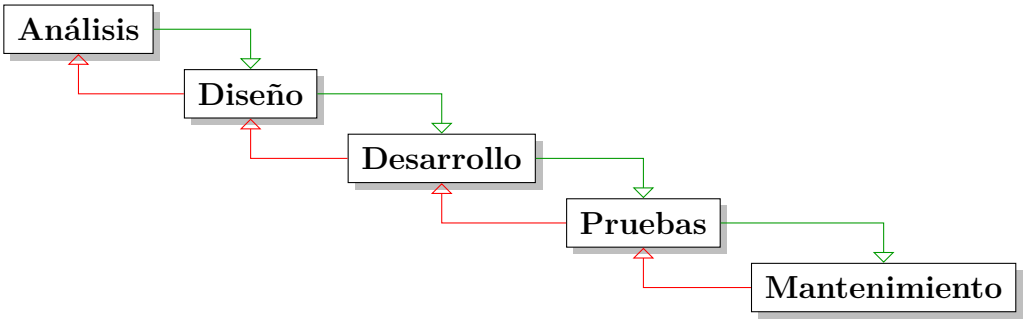

Figura 1.1: Modelo en cascada

Las fases del modelo se pueden resumir de la siguiente manera:

1. **Análisis de requerimientos:** se obtiene información acerca del dominio del problema y los requisitos que debe cumplir el sistema. Hay una gran interacción entre el cliente y analistas de sistemas. En este momento se define el alcance del sistema, es decir, se define lo que el sistema va a hacer y lo que no va a hacer. Una vez terminada esta etapa, no se pueden pedir nuevos requisitos o cambios sobre los existentes.

2. **Diseño:** ya definido completamente el problema, se define la arquitectura a utilizar, módulos que compondrán el sistema, estructura de datos, interfaces, etc.

3. **Desarrollo:** se procede a codificar el sistema partiendo del diseño.

4. **Pruebas:** se prueba que el sistema esté libre de errores, probando características tanto internas como externas. Se realizan pruebas de integración, de sistema y de aceptación.

5. **Mantenimiento:** luego de entregado el sistema, éste sigue sufriendo cambios, ya sea por errores detectados, nuevas funcionalidades o cambios pedidos por el cliente.

Algunas ventajas del modelo en cascada:

- Es ampliamente conocido y fácil de entender, tanto para la gente a cargo del desarrollo del sistema como para el cliente.

- Debido a su sencillez es fácil de implementar.

- En las distintas etapas se necesitan distintos tipos de recursos, por lo que en principio terminada una etapa esos recursos pueden ser liberados y aprovechados para otros proyectos.

Sin embargo, este modelo ampliamente utilizado aún hoy en día (aunque no de forma tan rígida), tiene varias desventajas:

- El modelo requiere en la etapa de análisis mucha precisión en la definición de requerimientos, ya que luego no se pueden hacer cambios. La mayoría de las veces los clientes no saben con tanta claridad lo que buscan.
- Luego de la etapa de análisis, la comunicación e interacción con el cliente es muy escasa. El cliente no ve el producto hasta estar finalizado, durante toda la etapa de desarrollo el cliente no provee una realimentación sobre el producto.
- Hoy en día los negocios son muy cambiantes, en un proyecto que dura 1 o 2 años, muchas cuestiones relacionadas con el negocio pueden cambiar. Este modelo no admite cambios en los requerimientos.
- El cliente recién puede usar el sistema una vez finalizado. Durante todo el transcurso del proyecto el cliente invierte dinero pero no puede recuperar nada.
- Un error de una etapa detectado en etapas siguientes se vuelve más costoso cuanto más alejada esté. En la etapa de pruebas se detectan la mayor cantidad de errores, y esta etapa está recién sobre el final del proyecto.
- Es muy difícil y costoso estimar con precisión al principio del proyecto el costo total del mismo.

Una consecuencia de estas desventajas es que muchas veces el proyecto no termina (no todos los clientes tienen el soporte económico para poder afrontar una inversión tan grande sin poder recuperarla en el corto plazo). También sucede que parte de la funcionalidad pedida (cuando el proyecto está "verde" y todavía no se conoce mucho acerca del mismo) termina no siendo usada o aportando poco valor. Pero sin duda los dos puntos más débiles son la poca flexibilidad al cambio (en un contexto sumamente cambiante) y la poca comunicación con el cliente luego de la etapa de análisis. El cliente habla un idioma y la gente encargada de implementar la solución otro, es muy difícil que entre ambos se entiendan inmediatamente. Necesariamente debe haber un ida y vuelta entre ambos, de modo de detectar problemas (de concepto, comunicación, entendimiento) de forma temprana y evitar que éstos se magnifiquen con el paso del tiempo.

Se han hecho muchos estudios acerca del éxito de los proyectos de software en cuanto a costo y alcance. Si bien hay diversos resultados y críticas a dichos estudios, en general hay un consenso en que más de la mitad de los proyectos de software terminan costando más de lo planeado o no llegan a entregar la funcionalidad

pactada. A la hora de buscar "culpables", hay un estudio en particular que se enfoca en el modelo contractual en el cual se desarrollan estos proyectos (Atkinson y Benefield, 2013). Aquí dicen que una de las grandes fallas de este modelo es que se tiene que especificar toda la funcionalidad de antemano, cuando en realidad el cliente no está en condiciones de hacerlo. En un ambiente sumamente complejo y cambiante, las necesidades no son claras, y mucho menos invariantes.

También es una de las críticas que McCracken y Jackson hacen a este ciclo de vida (McCracken y Jackson, 1982). Ellos dicen que no hay que ignorar el hecho de que el mismo proceso de desarrollo cambia la percepción del usuario acerca de las posibilidades y limitaciones del sistema, así como del entorno del sistema.

1.2. Agilismo

En vista de los problemas mencionados en el desarrollo de software, mucha gente empezó a buscar metodologías alternativas que contemplaran estos problemas. Así, comenzaron a cobrar fuerza distintas propuestas de metodologías iterativas e incrementales.

Kent Beck convoca a un grupo de expertos a una reunión en febrero de 2001 con el objetivo de discutir técnicas y procesos en el desarrollo de software. De esta reunión surgió el manifiesto ágil (Beck et al., 2001) que dice:

> *Estamos descubriendo formas mejores de desarrollar software tanto por nuestra propia experiencia como ayudando a terceros. A través de este trabajo hemos aprendido a valorar:*
>
> **Individuos e interacciones** *sobre procesos y herramientas*
>
> **Software funcionando** *sobre documentación extensiva*
>
> **Colaboración con el cliente** *sobre negociación contractual*
>
> **Respuesta ante el cambio** *sobre seguir un plan*
>
> *Esto es, aunque valoramos los elementos de la derecha, valoramos más los de la izquierda.*

El manifiesto hace un foco muy fuerte en el cliente, en la búsqueda de satisfacer sus necesidades. Propone que el cliente no sea alguien externo que interactúa sólo al principio, sino un integrante más del equipo que acompaña toda la vida del producto. A diferencia de lo que mucha gente cree, el agilismo no propone eliminar la documentación, sino mejorarla, realizar la documentación suficiente y que sea de

utilidad. Por último, en vez de negar la realidad de que estamos ante un contexto sumamente cambiante, lo toma como pilar de su filosofía. El agilismo no sólo está preparado para el cambio, sino que lo impulsa.

Junto con el manifiesto, establecieron los 12 principios del software ágil:

- Nuestra mayor prioridad es satisfacer al cliente mediante la entrega temprana y continua de software con valor.
- Aceptamos que los requisitos cambien, incluso en etapas tardías del desarrollo. Los procesos Ágiles aprovechan el cambio para proporcionar ventaja competitiva al cliente.
- Entregamos software funcional frecuentemente, entre dos semanas y dos meses, con preferencia al periodo de tiempo más corto posible.
- Los responsables de negocio y los desarrolladores trabajamos juntos de forma cotidiana durante todo el proyecto.
- Los proyectos se desarrollan en torno a individuos motivados. Hay que darles el entorno y el apoyo que necesitan, y confiarles la ejecución del trabajo.
- El método más eficiente y efectivo de comunicar información al equipo de desarrollo y entre sus miembros es la conversación cara a cara.
- El software funcionando es la medida principal de progreso.
- Los procesos Ágiles promueven el desarrollo sostenible. Los promotores, desarrolladores y usuarios debemos ser capaces de mantener un ritmo constante de forma indefinida.
- La atención continua a la excelencia técnica y al buen diseño mejora la Agilidad.
- La simplicidad, o el arte de maximizar la cantidad de trabajo no realizado, es esencial.
- Las mejores arquitecturas, requisitos y diseños emergen de equipos autoorganizados.
- A intervalos regulares el equipo reflexiona sobre cómo ser más efectivo para a continuación ajustar y perfeccionar su comportamiento en consecuencia.

De estos principios se desprenden varias cosas importantes además de las ya mencionadas del manifiesto:

- La entrega periódica y en plazos cortos de software funcional permite que el cliente pueda empezar a recuperar la inversión de modo rápido. Permite al cliente ver y sentir cómo está quedando su producto, dando lugar a una realimentación temprana. Como consecuencia se detectan errores o malentendidos de forma temprana, siendo así mucho menos costoso repararlos.
- Se entrega primero lo que es de mayor valor para el cliente, haciendo que pueda recuperar la inversión más rápido.
- Se fomenta la integración del equipo, la comunicación y la motivación. Esto provoca que todos tengan la misma meta, que es sacar adelante el proyecto.

User Stories

El agilismo adopta una filosofía que pone en primer lugar al cliente. Una de las críticas que se le hace al modelo en cascada es la forma de definir requerimientos, ya que es necesario proveer demasiado detalle y tiene poca flexibilidad. Es por eso que se adoptó el uso de User Stories (US) o Historias de Usuario, que son pequeñas oraciones que describen los requerimientos en el lenguaje del cliente. Es una manera rápida de relevar los requerimientos del usuario, sin tener que entrar en detalle de cómo se va a implementar y sin tener que crear una documentación extensa.

La forma más común de escribir US es con el formato:

As a (Como un) *quién*

I Want (Quiero) *qué*

So That (Para que) *por qué*

Si bien es un formato muy extendido, cada caso es único, y hay veces que es conveniente no forzar a que sigan esta convención. Algunos ejemplos de US podrían ser los siguientes:

- Login en el sistema.
- Como bibliotecario, quiero poder buscar libros por fecha de publicación.
- Añadir artículo al carrito de compra.

La razón por la cual son tan cortas, es que las US no pretenden documentar los requerimientos, sino representarlos. Esto implica que al momento de implementar la US se debe profundizar con el cliente qué es lo que se espera.

El uso de US tiene varias ventajas:

- Son muy cortas. Esto hace que sean fáciles de mantener y rápidamente saber de qué se trata .
- Están escritas en el lenguaje del cliente. Este es uno de los puntos más importantes. Si bien no es sencillo escribir buenas US, son muy fáciles de leer, dado que usan el lenguaje del negocio y no un lenguaje técnico que pocos comprenden.
- Dicen *qué* y no *cómo*. Esto permite que se enfoque en lo que realmente se busca. Además, permite una mayor flexibilidad a la hora de implementarlas.
- Al descomponer el sistema en pequeñas US, éstas se pueden estimar (en tiempo y costo) y priorizar, de modo que las más importantes para el cliente se realicen primero.

1.3. Extreme Programming

Extreme Programming (XP) es una metodología de desarrollo creada por Kent Beck en 1996 como respuesta a los problemas mencionados del modelo en cascada. Está centrado en la satisfacción del cliente, la posibilidad del cambio y el trabajo en equipo. Se compone de muchos ciclos cortos, donde en cada ciclo se entrega nueva funcionalidad al cliente en vez de entregarla una sola vez al final; es por este motivo que se la llama una metodología incremental.

XP se basa en los siguientes valores (Beck y Andres, 2004), que tienen en cuenta tanto el valor humano como el comercial:

Simplicidad: se hace solamente lo que se debe hacer, no más. Esto maximiza el valor creado para la inversión hecha hasta la fecha. Se toman pasos pequeños y simples hacia la meta y se arreglan fallas a medida que van ocurriendo.

Comunicación: uno de los pilares fundamentales es la comunicación. Cada persona es parte del equipo y se comunican cara a cara todos los días. Siempre se debe trabajar conjuntamente en todo, desde los requerimientos hasta el código.

Realimentación: se entrega en cada iteración software que funciona. Se hace una demostración temprana del sistema, se escucha atentamente y se hace cualquier cambio necesario. El cliente tiene una realimentación temprana en cuanto a la calidad de sus US, mientras que el equipo de desarrollo tiene una realimentación temprana de la calidad del sistema, tanto interna a través de las pruebas unitarias, como externas, a través de las pruebas de aceptación y comentarios del cliente. El equipo se adapta al proceso y no al revés.

Respeto: todos los miembros del equipo son valiosos y merecen el respeto de todos. Todos aportan valor aunque sólo sea entusiasmo. Los desarrolladores respetan la experiencia de los clientes y viceversa.

Valentía: siempre se dice la verdad del progreso y estimaciones, no se ponen excusas por los fracasos. El cambio no es algo malo, el equipo se adapta a los mismos cuando ocurran.

En estos valores se puede ver el foco en el cliente, la buena comunicación y realimentación. Tanto en los valores como en un conjunto de reglas que siguen XP (Wells, 2009) le dan mucha importancia al aspecto humano, sobre todo a la motivación del equipo, tanto para el cliente al recibir software que funciona de manera temprana y periódica, como a la gente encargada del desarrollo al recibir realimentación y satisfacción del cliente.

2. TDD

Test Driven Development (TDD) o desarrollo guiado por pruebas es una técnica de diseño e implementación de software propuesta en el esquema de XP. A diferencia del resto de las técnicas de desarrollo, en TDD primero se escriben las pruebas y luego el código necesario para que estas prueban pasen.

Cada iteración de XP está compuesta de muchos ciclos TDD, donde cada ciclo se lo suele describir de forma simplificada como "Rojo-Verde-Refactorizar" (Figura 2.1). *Rojo* significa escribir una prueba que falle (normalmente en los frameworks de testing cuando una prueba falla se la marca con color rojo), *Verde* significa escribir el código necesario para que esa prueba deje de fallar (en los frameworks de testing se la suele marcar con color verde) y por último se hace una *refactorización* del código para mejorarlo.

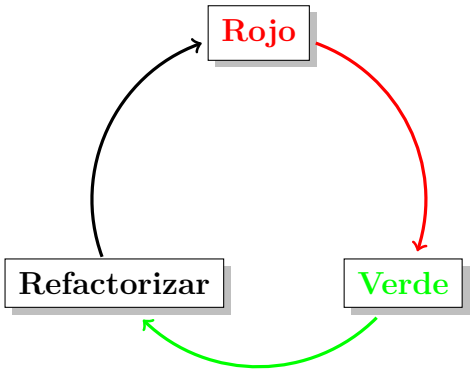

Figura 2.1: Ciclo TDD simplificado

2.1. Ciclo TDD

Una forma más completa de describir el ciclo TDD se puede ver en la figura 2.2. A continuación se explica cada paso del ciclo:

1. El primer paso consiste en escribir una prueba relacionada con lo que se quiere desarrollar. Es el mismo desarrollador que luego va a implementar el código quien escribe la prueba. Siendo que todavía no existe código, ni siquiera la interfaz, esta tarea ya lleva a pensar en el diseño. ¿Qué clase se necesita? ¿Qué métodos se deben llamar? La prueba debe ser pequeña, debe probar solo aquello que se desea probar y no más, y debe estar lo más aislado del resto posible. A veces es difícil probar el Sistema Bajo Prueba (SBP) porque depende de otros componentes que no se pueden (ni convienen) usar en el entorno de pruebas. Esto puede ocurrir porque estos componentes no están disponibles, o tienen efectos secundarios no deseados. En estos casos, se deben usar lo que Meszaros llama "dobles" (Meszaros, 2007), que reemplazan estos componentes por algo ficticio pudiendo controlar su comportamiento.

2. Como se acaba de escribir una prueba antes del código, dicha prueba fallará. Incluso es probable que haya errores de compilación[1] ya que se pueden estar haciendo referencias a clases o métodos que aún no existen. De la misma manera, puede ocurrir que una prueba haga uso de componentes ya implementados y la prueba pase automáticamente.

3. Una vez que se tiene una prueba que falla, se procede a implementar el código que permita hacer pasar esa prueba. Se debe escribir sólo lo necesario para hacer pasar la prueba, no más. No se debe pensar en otras pruebas o futuros requerimientos, ya sean existentes o no. En este momento se debe escribir la cantidad mínima de código que haga que la prueba deje de fallar.

4. Una vez que la prueba pasa, se debe verificar que todas las otras pruebas pasen. Antes del primer paso, todas las pruebas pasaban, y lo mismo debe ocurrir cuando finaliza el ciclo TDD. De esta manera se asegura que el código nuevo satisface la prueba recién agregada y no modificó el comportamiento de lo ya desarrollado.

5. El paso de refactorización es sumamente importante, de hecho Wheeler lo toma como una de las innovaciones más importantes en el mundo del software (Wheeler, 2011) y se debe respetar siempre. Este es el momento de arreglar el código, de hacerlo más prolijo. Algunas veces se harán cambios más grandes, otras veces más chicos. Refactorizar puede ser cambiar el nombre de un método por uno mejor y más descriptivo, eliminar duplicación de código (tanto del sistema como el de las pruebas), separar porciones de código en nuevas funciones, mejorar la legibilidad del código, comentar el código para

[1] En el framework de testing de Ruby se suele distinguir falla cuando una prueba compila pero no pasa y error cuando la prueba no compila, pero en ambos casos el resultado de la prueba es rojo.

una mejor comprensión, entre otros. En definitiva, se mejora la estructura interna del código sin afectar el comportamiento externo.

6. Antes de volver a ejecutar el ciclo TDD, hay que correr todas las pruebas para asegurarse que todas pasan, ya que la refactorización no debería haber cambiado ningún comportamiento externo.

7. Se vuelve a empezar desde el punto 1.

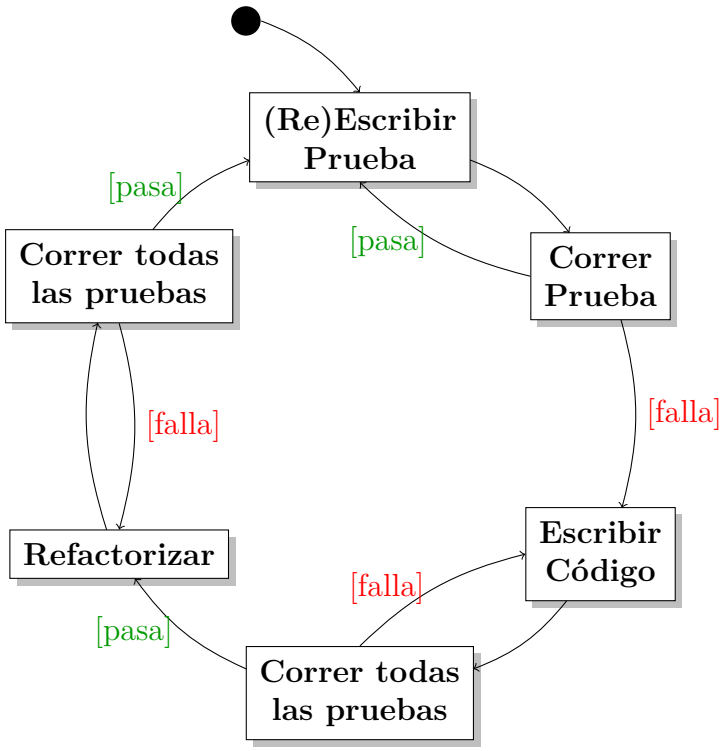

Figura 2.2: Ciclo TDD

La palabra *test* en TDD suele causar confusión, pues se lo suele asociar a una técnica de testing, cuando en realidad es una técnica de diseño. A la hora de realizar las pruebas se toman decisiones de diseño, ya que se define la interfaz de ciertos componentes y su forma de interactuar. Es por eso que las pruebas no sirven simplemente para verificar que el código haga lo que debe hacer, sino que sirve para guiar el desarrollo, para indicar qué se debe hacer. Por otro lado, también indican cuando algo está terminado (algo está terminado cuando las pruebas pasan). Por esta razón, las pruebas son tan importantes como el código mismo.

Robert Martin menciona el caso de un cliente intentando desarrollar con TDD cuya política de la empresa era tratar las pruebas como código que luego se va a tirar(Martin, 2005a). En las pruebas tenían permitido romper todas las reglas, no seguir estándares de código, no seguir buenas reglas de diseño ni eliminar duplicación. El resultado fue que después de un tiempo cuando querían refactorizar el código en producción tenían que cambiar muchísimas pruebas. Terminó siendo tan complejo mantener las pruebas que tuvieron que abandonar TDD.

La regla es entonces mantener el código de las pruebas con la misma calidad que el código en producción. Las pruebas no sirven sólo para el desarrollo del momento, sino para el mantenimiento posterior. Bob Martin llega a decir que las pruebas son tan o más importantes que el código en producción, ya que se puede recrear el código en producción a través de las pruebas pero no al revés.

Una duda frecuente que surge es cada cuánto se deben hacer pruebas. ¿Se deben hacer pruebas para todo? ¿Incluso si es trivial? ¿Y si es algo muy fácil? Siendo que las pruebas dicen qué se debe hacer, no se debería hacer algo que no sea requerido por una prueba, por lo que en principio hay pruebas para todo. Está claro que no tiene mucho sentido hacer pruebas para desarrollar getters y setters[2] triviales, pero a veces no hacer pruebas para hacer un cambio "que es muy sencillo" puede traer problemas.

Otro caso que cuenta Martin es el que él llama "tan solo 10 minutos sin pruebas" (Martin, 2005d). Allí cuenta el caso en el que quiso hacer un cambio en un sistema en el que estaba trabajando y en principio no encontraba una forma incremental de hacerlo. Siendo que no parecía muy complejo y no eran muchos los lugares que tenía que cambiar, decidió tomar el riesgo y hacer los cambios sin pruebas que lo ayudaran. Cuando terminó de hacer todos los cambios, una docena de pruebas fallaban y no lograba encontrar el motivo. Después de varios intentos y frustraciones encontró la manera de hacerlo de forma incremental, deshaciendo todo lo que había hecho y haciéndolo de la forma "TDD". Fue ahí cuando encontró cuál de todos los cambios era el que fallaba y la solución era inmediata. La conclusión de este episodio es que por intentar ahorrarse 10 minutos le costó 2 horas de trabajo.

A la hora de implementar el código de producción, es importante enfocarse en lo que es necesario para poder pasar las pruebas. Siendo que se parte de pruebas para realizar lo que es necesario, y no se implementa más código del necesario, se

[2]Métodos para leer y modificar atributos de una clase.

evita el goldplating[3].

2.1.1. Refactorización

La refactorización de código es una parte tan importante del ciclo (y del desarrollo de software en general) que se han escrito varios libros para tratar este único punto. Martin Fowler define a la refactorización como (Fowler, 1999):

> *El proceso de cambiar un sistema de software de manera tal que no cambia el comportamiento externo del código, sino que mejora su estructura interna. Es una manera disciplinada de ordenar el código para minimizar las posibilidades de introducir errores en el futuro. En esencia, cuando se refactoriza se está mejorando el diseño del código después de haber sido escrito.*

Es importante aclarar que la refactorización se lleva a cabo haciendo pequeñas modificaciones a la vez, con pruebas que las respalden para verificar que no se está cambiando el comportamiento externo. Estos cambios no agregan funcionalidad, sino que mejoran la estructura interna, la hacen más comprensible para un ser humano. Puede ser desde agregar clases para resolver un problema con polimorfismo en vez de un conjunto de `if-else` anidados hasta agregar comentarios que ayuden a entender qué hace una determinada porción de código.

Los efectos que producen la refactorización de código son un mejor diseño, eliminación de código duplicado, separación de incumbencias, código más limpio, entre otros. Hace que el mantenimiento posterior sea más fácil, más rápido y con menor probabilidad de error. También evita el *deterioro* del sistema (esto se explica mejor con las Leyes de Lehman en 4.1).

2.1.2. La importancia del ciclo TDD

Robert Martin hace una analogía muy interesante con los contadores (Martin, 2005e). En el mundo de las finanzas, un simple error en un solo dígito puede costar millones o incluso la quiebra de una organización. Para resolver este problema, los contadores usan una serie de prácticas y disciplinas que reducen la probabilidad de que estos errores no sean detectados. Una de estas prácticas es la contabilidad por partida doble, cada transacción se anota 2 veces en el libro de contabilidad, una vez en la columna *debe* y una en la columna *haber*. Si bien esta técnica no es perfecta y

[3] Cuando se siguen desarrollando mejoras de algo que ya cumple con los requerimientos pensando que va a ser mejor para el cliente aún cuando éste nunca lo pidió. Muchas veces esto provoca el efecto contrario, además de haber perdido tiempo.

se necesitan otros controles, resuelve muchos problemas, y los contadores piensan que vale la pena utilizarla, aún cuando el esfuerzo es exactamente el doble.

TDD actúa de forma muy similar, es una primera línea de defensa contra los errores. Muchas veces se deja de usar TDD ante la presión del tiempo alegando que no hay tiempo para escribir pruebas. Siguiendo el paralelo, ¿dejaría un contador de usar la técnica de contabilidad por partida doble ante la presión del tiempo? O yendo aún más al extremo, ¿qué pasaría si un piloto deja de lado los controles de la lista de verificación ante la presión del tiempo?

2.2. Ventajas y desventajas

A priori esta forma de desarrollar impacta mucho, ya que propone un cambio bastante radical a lo conocido. El simple hecho de hacer pruebas de algo que todavía no está implementado ya de por sí es un cambio radical. Por otro lado, al revés de lo que recomiendan las buenas prácticas del testing tradicional, aquí debe ser la misma persona quien desarrolla las pruebas unitarias y quien implementa el código que hace pasar esas pruebas.

Esta forma de desarrollar trae varias ventajas que se desprenden directamente, y otras más importantes como efecto secundario. Las ventajas de TDD son:

- Se evita el goldplating. De esta manera se utiliza el tiempo para implementar lo que el cliente pidió y no otras cosas que no pidió.
- El código es *testeable*[4] por definición. Cuanto más acoplado sea el código, más difícil es hacerle pruebas. Al ser *testeable*, provoca que se busque y evolucione a un código con bajo acoplamiento y alta cohesión, dos características propias de un correcto diseño de un sistema.
- Reduce mucho el uso del depurador. Siendo que las pruebas están diseñadas para pequeñas porciones de código y se hacen pequeños pasos incrementales, los errores suelen estar bien identificados y aislados. El uso del depurador consume muchísimo tiempo, con lo cual un ahorro en su uso se transforma en un gran ahorro de tiempo.
- Mejora la comunicación en el equipo (siendo el cliente parte del mismo). Dado que el cliente está constantemente interactuando con la gente encargada del desarrollo, en todo momento conoce el estado del proyecto. La realimentación constante también es un excelente canal de comunicación. Una buena comunicación se traduce en un mejor entendimiento y menores confusiones

[4]Que se puede probar. La traducción literal sería “comprobable” o “verificable” aunque no refleja completamente el concepto.

o malentendidos. La brecha entre lo que el cliente quiere decir y lo que el equipo de desarrollo entiende se hace más chica. Esto también genera un ahorro de tiempo, ya que cuanto más temprano se detecta un error, menos cuesta arreglarlo.

- Es una fuente de estimulación para el equipo. Por un lado, el cliente ve cómo el proyecto toma forma, puede empezar a utilizar el sistema desde una etapa temprana. Participa y forma parte en la toma de decisiones, es un integrante más del equipo. Por otro lado, el equipo de desarrollo produce código de alta calidad que le sirve al cliente y recibe su constante realimentación.

- Produce software de alta calidad. Las pruebas están diseñadas para verificar que se cumplan los requerimientos tanto explícitos como implícitos. Si bien una alta cobertura de código no implica que el software sea de alta calidad, la calidad de las pruebas está directamente relacionada con la calidad del producto.

- Las pruebas son la mejor documentación que uno puede tener. Por un lado, siendo que las pruebas deben pasar en todo momento, están siempre actualizadas. Si hay un cambio que hace que una prueba esté desactualizada, ésta debería modificarse para pasar. Por otro lado, cuando uno quiere hacer uso de una biblioteca, antes de leer la extensa documentación técnica busca los ejemplos de uso. Las pruebas son exactamente eso, un ejemplo de cómo se usa un método, clase o módulo dado.

En uno de sus libros Steve McConnell enumera errores clásicos en el desarrollo de software (McConnell, 1996). La naturaleza de TDD evita muchos de ellos, por ejemplo:

- **#1: Motivación debilitada.** Dice que los estudios muestran que la motivación es el factor que más efecto tiene sobre la productividad y la calidad. XP promueve la motivación del equipo y el uso de TDD ayuda a la satisfacción del cliente mediante la entrega temprana de un producto de calidad y al equipo de desarrollo al ver que produce software de utilidad.

- **#7: Roces entre desarrolladores y el cliente.** Puede surgir por diferentes motivos, y el efecto que crea es una comunicación pobre entre ambos, y como efecto secundario confusiones o malentendidos en los requerimientos. La comunicación entre el cliente y el equipo de desarrollo es un pilar fundamental en XP y por la forma en que se lleva a cabo TDD, con el cliente como parte del equipo, esta comunicación se da de forma natural. Hay una realimentación constante por parte del cliente, por lo que los malentendidos en los requerimientos son bajos.

- **#11: Usuario poco involucrado.** Relacionado con lo ya dicho, el cliente interactúa constantemente con el equipo de desarrollo, está tan involucrado como cualquier otro integrante del equipo.

- **#21: Diseño inadecuado.** TDD tiene la característica de producir software de muy alta calidad, con bajo acoplamiento y alta modularidad. Es una herramienta de diseño que soporta el cambio, con lo cual en cada iteración se está diseñando, y está preparada para evolucionar o escalar de ser necesario.

- **#22: Aseguramiento de la calidad recortado.** Cuando en un proyecto hay poco tiempo, se suelen tomar atajos eliminando revisiones de código, refactorizaciones, planificación de pruebas. Esto lleva a software de baja calidad que a la larga termina siendo más costoso de reparar (cuanto más tarde se detecta un error, más costoso es repararlo). En TDD, por su naturaleza, si se hacen esos recortes, se deja de hacer TDD. El aseguramiento de la calidad es constante durante todo el desarrollo, las refactorizaciones son obligatorias, y la planificación de pruebas es el primer paso a tomar.

- **#28: Goldplating en requerimientos.** Muchos proyectos suelen tener más requerimientos que los necesarios desde un comienzo, el alto rendimiento suele ser un ejemplo bastante típico. Cuando el cliente toma la decisión de cuáles son las funcionalidades a implementarse en el próximo ciclo y al tener que desarrollar los criterios de aceptación, allí se da cuenta si un requerimiento es realmente válido y si realmente lo necesita. Ocurre a menudo que al tener que poner el requerimiento en un contexto y pensar un ejemplo de uso se recuestiona la utilidad del mismo.

- **#29: Aumento de funcionalidades o características** Durante el ciclo de vida de un proyecto suele haber cambios en los requerimientos. Si los cambios no son bien manejados, puede provocar fácilmente un aumento de tiempo y costo al proyecto. Si bien no depende únicamente de la metodología usada sino también del manejo y control de cambios por parte del líder de proyecto, una metodología preparada para el cambio ayuda a afrontar estos problemas.

- **#30: Goldplating en el desarrollo.** Como ya se ha mencionado, por la forma en que se desarrolla con TDD se evita el goldplating.

Sin embargo, TDD presenta algunas desventajas:

- Está pensado para construir software desde cero. Como se ha mencionado desde antes, naturalmente crea código *testeable*, altamente desacoplado y cohesivo. En este entorno, es fácil crear nuevas pruebas. Sin embargo no

siempre es fácil utilizarlo para continuar el desarrollo o mantenimiento de un producto de software ya construido con otra metodología. De todas maneras, estos sistemas pueden ser adaptados mediante refactorizaciones, aunque puede ser laborioso al principio (Feathers, 2004).

- Requiere mucha disciplina y un equipo convencido de que es una excelente herramienta. El tiempo es algo que nunca sobra, y si se piensa que las pruebas son una pérdida de tiempo cuando éste escasea, TDD está condenado al fracaso. Siendo que el código en producción es finalmente lo que al cliente realmente le interesa, muchas veces se piensa que las pruebas no son tan importantes (al fin y al cabo, el cliente no utiliza las pruebas sino el código en producción). Esta mirada es muy a corto plazo, y se dejan de lado cuestiones muy importantes, como la calidad del producto entregado y el costo de arreglar los errores en el futuro. Aún así, esto ocurre a menudo, por lo que el primer paso para que TDD sea exitoso es tener un equipo que sea consciente de las ventajas que trae esta herramienta y que no se trata simplemente de tener mucha cobertura de código.

- Es difícil automatizar pruebas para interfaces de usuario. Si bien existen herramientas que permiten automatizar ciertas pruebas, no se pueden automatizar todo tipo de pruebas, como las que aparecen en la figura 2.3 (Hayes, 2000). Por otro lado, las interfaces de usuario suelen ser bastante cambiantes, por lo que las pruebas deberían reescribirse muy seguido y el tiempo perdido empieza a superar al tiempo ahorrado.

- El cliente en el equipo es una pieza fundamental. Muchas veces es difícil conseguir que el cliente dedique una persona o conjunto de personas al equipo del proyecto. Es importante que el cliente entienda que la persona más idónea para tomar decisiones de negocio es él mismo, que conoce del dominio del problema. Las decisiones que no tomen ellos, deberán ser tomadas por alguien que no conoce del tema, con las consecuencias que eso trae.

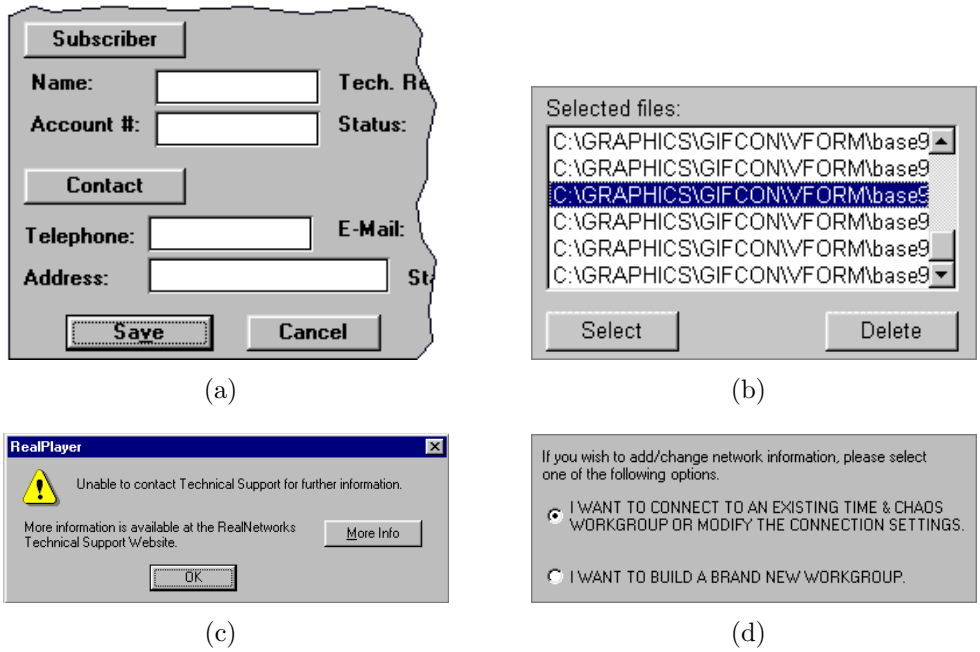

Figura 2.3: Errores en interfaces de usuario extraídas de *The Interface Hall of Shame*. En (a) los campos *Subscriber* y *Contact* son simplemente títulos, aunque mucha gente los confunde con botones. En (b) no hay espacio suficiente para ver el nombre completo. En (c) hay un cartel de error de "No se pudo contactar con el servicio técnico" ofrece el contacto con el servicio técnico. En (d) el uso de mayúsculas va contra las reglas de *netiquette*.

Una arquitectura que evoluciona

Una crítica que se le suele hacer a las metodologías ágiles como XP es que no se piensa en una arquitectura desde el principio. La respuesta a esas críticas es que la arquitectura evoluciona. Eso no significa que evoluciona en algo desorganizado que convenga en el momento, sino que requiere pensar mucho y una gran disciplina.

Muchos son escépticos y piensan que se debe pensar en una arquitectura a priori que sea flexible para solucionar el problema en cuestión. Bob Martin responde (Martin, 2005c)

> *¿Qué clase de arquitectura querés? ¿Una que fue diseñada al principio hace un año y no ha cambiado desde entonces, o una que ha probado su flexibilidad habiendo evolucionado constantemente?*

En el mismo artículo menciona un ejemplo real del proyecto FitNesse[5]. Allí relata cómo TDD juega un rol vital en los pequeños cambios incrementales para poder hacer evolucionar la arquitectura, ya que en todo momento sabe si algún cambio impactó negativamente en lo que ya estaba funcionando.

Burke y Coyner también desalientan los grandes diseños al principio (Burke y Coyner, 2003). En cambio proponen ir desarrollando cada cambio pequeño a la vez e ir evolucionando según sea necesario.

2.3. Variantes de TDD

Como ya se ha mencionado anteriormente, la palabra *test* en TDD trajo confusión, haciendo pensar que estaba más vinculado a la etapa de pruebas que a la de diseño y desarrollo.

A lo largo del tiempo TDD fue evolucionando y derivando en metodologías similares pero con algunas características particulares.

UTDD

Cuando se mencionó el ciclo de vida de TDD, el ciclo se iniciaba escribiendo una prueba. En los comienzos de TDD, las pruebas que se escribían eran unitarias. De hecho, Kent Beck implementó SUnit, un framework para pruebas unitarias automatizadas para Smalltalk y más adelante JUnit junto a Erich Gamma para Java. Estas herramientas son indispensables para poder utilizar TDD, ya que las pruebas deben estar automatizadas.

[5]Framework para pruebas automatizadas de aceptación (`http://fitnesse.org`).

De esta manera, se le ha dado el nombre Unit-Test Driven Development (UTDD) o desarrollo guiado por pruebas unitarias a la práctica que sólo usa pruebas unitarias en el desarrollo. Si bien se le han hecho varias críticas, sigue siendo muy utilizado hoy en día, dado que conserva varias de las ventajas mencionadas, y existe una gran cantidad de frameworks que le dan soporte, más que para las otras variantes que se verán a continuación.

En el ejemplo 2.1 se ve el uso de pruebas unitarias para un juego de blackjack. Aquí se está probando un método que calcula el valor del juego según las cartas que posee el jugador. Como se ve, la clase hereda de `Test::Unit::TestCase`.

```
1  ...
2  class TestJuegoPersona < Test::Unit::TestCase
3    def test_suma_juego_basico
4      # 10 y 6 debe dar 16
5      juego = JuegoPersona.new
6      carta1 = Carta.new Mazo::NUMEROS.first, Mazo::PALOS.
           first, 10
7      carta2 = Carta.new Mazo::NUMEROS.first, Mazo::PALOS.
           first, 6
8      juego.agregar_carta carta1
9      juego.agregar_carta carta2
10
11     assert_equal(16, juego.valor(1), "El valor del juego
           deberia ser 16")
12   end
13
14   def test_suma_con_ases
15     # As y 8 debe dar 19
16     juego = JuegoPersona.new
17     carta1 = Carta.new Mazo::NUMEROS.first, Mazo::PALOS.
           first, 11
18     carta2 = Carta.new Mazo::NUMEROS.first, Mazo::PALOS.
           first, 8
19     juego.agregar_carta carta1
20     juego.agregar_carta carta2
21
22     assert_equal(19, juego.valor(1), "As y 8 debe dar 19")
23   end
24
25   ...
26 end
```

Ejemplo 2.1: Ejemplo simple en Ruby con pruebas unitarias

DDD

Si bien no es una variante de TDD, las ideas de Eric Evans de Domain Driven Design (DDD) o diseño guiado por el dominio fueron de gran influencia para futuras variantes (Evans, 2003). En su libro él menciona que no se trata de una metodología, sino una manera de pensar y un conjunto de prioridades que apuntan a acelerar proyectos de software con dominios complejos.

DDD tiene 2 grandes premisas:

- Para la mayoría de los proyectos de software, el foco principal debería estar en el dominio y la lógica del dominio.
- El diseño de dominios complejos debería estar basado en un modelo.

Otras variantes

El planteo de un diseño guiado por el dominio derivó en un cambio de enfoque y el surgimiento de variantes de TDD donde el cliente tiene un rol fundamental. En el capítulo 3 se analizan las variantes ATDD y BDD.

3. ATDD

Acceptance-Test Driven Development (ATDD) o desarrollo guiado por pruebas de aceptación es muy similar a TDD en cuanto a su estructura, la diferencia radica en que el ciclo parte de pruebas de aceptación. UTDD ayuda al programador a responder *¿El sistema hace esto en forma correcta?*, mientras que ATDD ayuda a responder *¿El sistema hace lo correcto?*.

En el sitio de FitNesse se refleja de forma clara:

> *Unit tests (TDD) is about Building the Code Right*
> *FitNesse (ATDD) is about Building the Right Code*

que traducido sería:

> *TDD se trata de construir el código en forma correcta (bien, limpio, no acoplado, cohesivo), mientras que ATDD se trata de construir el código correcto (el necesario).*

3.1. Pruebas de aceptación

Las pruebas de aceptación son especificaciones de la funcionalidad o comportamiento que debe cumplir la US. Indica cómo debe comportarse el sistema ante diferentes situaciones.

De la misma manera que hay varias maneras de escribir US, también existen varias maneras de escribir pruebas de aceptación. Lo importante es que no sean ambiguas y cubran todos los aspectos que deben cumplir. Las pruebas de aceptación son un indicador de cuándo una US está finalizada: cuando todas las pruebas de una US pasan, esa US está terminada. De esta manera, el esfuerzo para desarrollar una US es el justo, no se hace trabajo de más ni de menos.

Lasse Koskela menciona algunas propiedades de las pruebas de aceptación que son importantes tener en cuenta (Koskela, 2007):

El dueño de las pruebas es el cliente: dado que el sistema está siendo desarrollado para cumplir con los objetivos del cliente, debe ser éste quien especifique los criterios de aceptación.

Las escribe el cliente, junto con los desarrolladores y testers: esta tarea la desarrolla en colaboración con testers y desarrolladores, aportando cada uno su conocimiento en el área. Además provoca un flujo de comunicación, un ida y vuelta de preguntas y respuestas que ayudan a clarificar más la visión del cliente sobre la US.

Indican *qué* y no *cómo*: una característica importante es que indican la fuente de valor para el cliente en vez de cómo ese valor debe ser entregado. Por ejemplo, no es lo mismo pedir "La fecha de cumpleaños debe ser válida" que "Elegir a través de un menú desplegable el año, luego el mes, y según el mes y el año el menú desplegable de días se debe actualizar". El segundo caso tiene detalles innecesarios de la implementación y se pierde cuál es la verdadera fuente de valor.

Escritas en el lenguaje del dominio del problema: esto es fundamental para que el cliente se involucre en la creación de pruebas de aceptación y ayuda mucho a validar si las pruebas son correctas y suficientes. Además, se evita caer en tecnicismos y detalles de implementación.

Deben ser concisas, precisas y sin ambigüedades: cada prueba está hecha para verificar un solo aspecto o escenario de la US. No es necesario poner detalles que se pueden obtener fácilmente después, al fin y al cabo, la conversación con el cliente no termina en este punto, sino que sigue a lo largo de todo el ciclo.

Mike Cohn resalta que el uso de las pruebas debe ser parte del proceso de desarrollo (Cohn, 2004). Por un lado el cliente aporta su conocimiento acerca del dominio del problema, mientras que los testers ayudan con su conocimiento acerca de la elaboración de pruebas. Esto no significa que las pruebas de aceptación que se obtienen al principio de la iteración serán las únicas, sino que con el desarrollo de la US surgirán nuevas pruebas. También hay que tener en cuenta las pruebas unitarias que el desarrollador elabora en cada ciclo UTDD. Por ejemplo, hay ciertos casos donde algunos criterios de aceptación están implícitos, como por ejemplo verificar que la fecha sea válida.

3.2. Ciclo ATDD

En la figura 3.1 se puede ver la forma de un ciclo de ATDD. El ciclo es muy parecido al de la figura 2.2, con la diferencia que en este caso comienza en un nivel más abarcativo.

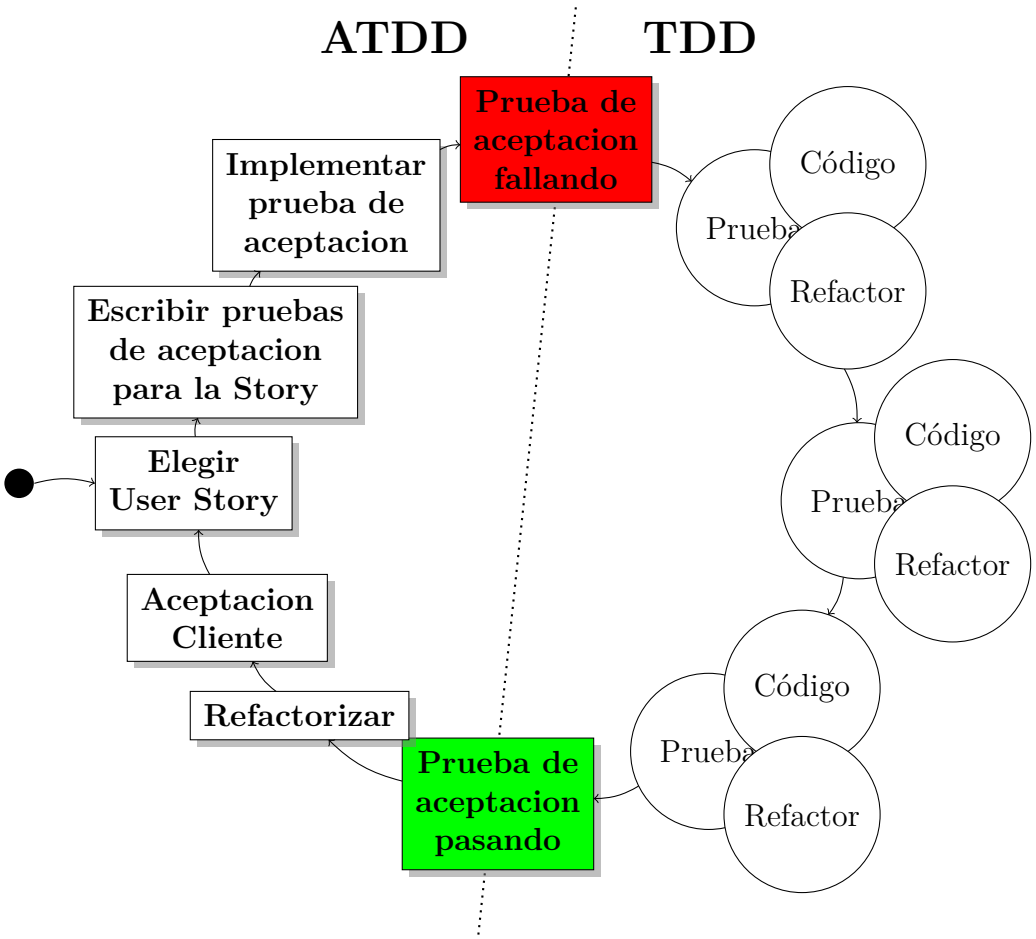

Figura 3.1: Ciclo ATDD

Los pasos del ciclo consisten en:

- Se comienza eligiendo la siguiente US a implementar. Normalmente las US están priorizadas según el valor que le aporta al cliente (él es quien les da la prioridad, ayudado por el equipo de trabajo), por lo cual se intenta elegir la de mayor prioridad.
- El paso siguiente es escribir las pruebas de aceptación para la US. Esta tarea la lleva a cabo el cliente ayudado por analistas, testers e incluso desarrolla-

dores. El objetivo es tener una buena idea de qué es lo que se espera de la US y cuáles son los distintos escenarios que hay que contemplar.

- Una vez que se tienen todos los escenarios, se deben automatizar las pruebas. Existen varias herramientas para la automatización de pruebas de aceptación, y un punto en común que tienen es que están orientadas a un lenguaje que el cliente pueda comprender fácilmente. En la figura 3.2 se puede ver un ejemplo hecho en FitNesse que usa un formato tabular fácil de comprender.
- Con las pruebas de aceptación automatizadas, es hora de implementar el código que las haga pasar. En esta etapa se suelen desarrollar varios ciclos UTDD para poder completar la US, donde en cada uno se implementa una funcionalidad particular necesaria para cumplir con la prueba de aceptación.
- Una vez terminada la implementación y refactorización, el cliente debe constatar que todas las pruebas pasan. Es por eso que es importante que las pruebas estén en un formato de fácil lectura y comprensión. Cuando el cliente ejecuta las pruebas y ve que todas pasan, el ciclo vuelve a comenzar.

Agregar Puestos De Bicicletas
direccion
Santa Fe 4700
Las Heras 3300
Figueroa Alcorta 2300

Obtener Puesto Mas Cercano Segun Ubicacion	
ubicacion	puestoMasCercano?
Cabildo 300	Santa Fe 4700
Las Heras 4250	[Las Heras 3300] expected [Santa Fe 4700]
Las Heras 3600	Las Heras 3300
Santa Fe 3151	Las Heras 3300
Del Libertador 2040	Las Heras 3300
Del Libertador 1600	Figueroa Alcorta 2300
Paseo Colon 850	Figueroa Alcorta 2300

Figura 3.2: Ejecución de pruebas de aceptación automatizadas en FitNesse

Hay un ejemplo de Bob Martin y Bob Koss para mostrar como se desarrolla un ciclo de ATDD(Martin y Martin, 2006). El ejemplo trata de un programa que lleva registro de una liga de bowling. Comienza con algunas US:

- Registro de juegos.
- Determinar posición equipos.
- Determinar ganadores y perdedores de cada partido semanal.
- Llevar el puntaje del juego.

Eligen la US de llevar el puntaje del juego. Este es el comienzo del ciclo ATDD. Como modo de ejemplo (y prueba de aceptación) plantean el siguiente juego:

1 4	4 5	6 /	5 /	/	0 1	7 /	6 /	/	2 / 6
5	**14**	**29**	**49**	**60**	**61**	**77**	**97**	**117**	**133**

En este episodio se puede ver como va evolucionando el sistema, a través de pequeños ciclos UTDD donde se van creando pruebas unitarias para probar cosas puntuales. Las primeras pruebas se resuelven de forma trivial, y a medida que se van creando pruebas más complejas, el código va evolucionando. Por ejemplo, al principio las pruebas son:

```
1 public void testScoreNoThrows()
2 ...
3 public void testAddOneThrow()
4 ...
5 public void testTwoThrowsNoMark()
```

Primeras pruebas sencillas

mientras que las últimas pruebas ya incluyen juegos completos, con *strikes*, *spares* y varios casos límites.

Se puede ver muy bien la separación entre hacer que una prueba pase (por más que no sea código prolijo) y refactorizar el código para eliminar duplicados, usar mejores nombres, mejorar la calidad. También muestran muy bien la evolución de la arquitectura y una de las premisas de XP que es no hacer algo que todavía no se necesita. En este caso, Robert Martin quería usar una lista doblemente encadenada entre `Frames`, pero esperó a que haya alguna prueba que demuestre que realmente lo necesitaban. Cuando finalizaron todas las pruebas, se dio cuenta que no era necesario.

3.3. Ventajas de ATDD

ATDD tiene las ventajas mencionadas en la sección 2.2, donde la mayoría se derivan del hecho de realizar las pruebas primero y luego el código. Cuando se desarrolla con ATDD, surgen algunas ventajas adicionales (Koskela, 2007):

- Da una definición de cuando está terminada una US: la Ley de Parkinson afirma que "el trabajo se expande hasta llenar el tiempo disponible para que se termine". Cuando todas las pruebas pasan, la US está terminada. Las pruebas de aceptación dicen exactamente qué es lo que hay que hacer y cuándo está finalizado el trabajo, sin lugar para las ambigüedades. Como se mencionó anteriormente, esto evita el goldplating. También le sirven al cliente para saber en qué estado está el desarrollo.

- Promueve un trabajo en conjunto. Siendo que todos están trabajando con un mismo objetivo, se promueve un ambiente de trabajo cooperativo. Cuando el testing se hace en una etapa posterior al desarrollo, si un tester encuentra un problema, el desarrollador ya va a estar trabajando en otra cosa, y los objetivos de cada uno entran en conflicto. Con ATDD, no sólo todos trabajan con el mismo objetivo, sino que cada uno aporta su visión y conocimiento al problema.

- Genera confianza y compromiso en el cliente y el equipo de desarrollo. Al ser el cliente quien escribe las pruebas de aceptación junto con el equipo de desarrollo, el cliente obtiene lo que él pide, ve que sus necesidades son cubiertas, y el equipo de desarrollo que su trabajo es valorado por el cliente.

- Especificar con ejemplos es una manera de expresar los requerimientos de una manera comprensible en vez de fórmulas complejas o texto genérico que resulta ambiguo. Las pruebas que son expresadas con ejemplos concretos son más fáciles de leer, de entender y de validar.

- Las pruebas unitarias indican si pequeñas porciones de código funcionan bien de forma aislada, pero no indican si funcionan bien de forma conjunta. Las pruebas de aceptación integran varios componentes, indican si el software desarrollado en conjunto cumple con las expectativas del cliente. ATDD no reemplaza UTDD, sino que trabajan en conjunto. Las pruebas de aceptación son las que indican qué módulos deben hacerse y qué pruebas unitarias son necesarias. Como se ve en la figura 3.1 donde se muestra el ciclo de ATDD, está compuesto por varios ciclos UTDD. Esto trae una ventaja adicional muy grande, que es que las pruebas de aceptación cambian con menos frecuencia que las pruebas unitarias. De la misma manera que se puede refactorizar

código con el respaldo de las pruebas unitarias que indican que no se cambió el comportamiento, se pueden refactorizar, cambiar o corregir pruebas unitarias con el respaldo de las pruebas de aceptación (Fontela et al., 2013).

3.4. BDD

Como ya se ha mencionado, mucha gente piensa que TDD está más relacionado con las actividades de pruebas. El hecho ocurre porque la palabra *prueba* aparece en muchos lados, empezando por el propio nombre. En las primeras versiones de JUnit[6] los métodos de prueba debían empezar con la palabra *test*. También se hablan de *TestCases*, *TestSuites*, etc.

Dan North comenta en un artículo algunos problemas que tenía a la hora de explicar TDD(North, 2006). Las preguntas más frecuentes de sus alumnos eran:

- ¿Por dónde empezar?
- ¿Qué se debe probar? ¿Qué no se debe probar?
- ¿Cuánto se debe probar en una prueba?
- ¿Cómo se deben nombrar las pruebas?
- ¿Cómo entender por qué falla una prueba?

Lo primero que menciona North es que los nombres de los métodos de prueba deberían ser oraciones que expliquen lo que se está probando. Además, el nombre del método debe comenzar con la palabra *should* (debería) para expresar que la clase o método *debería* hacer algo. Luego empezó a usar la palabra *comportamiento* en vez de prueba, para indicar que se está especificando lo que debe hacer el sistema, no que se está verificando que el sistema haga algo.

Dave Astels hace un comentario interesante acerca del lenguaje utilizado (Astels, 2005). Por un lado, menciona que cuando la gente piensa en pruebas, los programadores piensan en "es muy simple, no necesita pruebas" o "esto ya lo hice muchas veces" y los gerentes en "hacemos las pruebas después de que el código esté terminado" o "no tenemos tiempo para hacer pruebas ahora". Si en cambio se habla de especificaciones, el concepto cambia totalmente.

Este cambio de enfoque y lenguaje más apropiado llevó a North a empezar a usar el término Behaviour Driven Development (BDD) o desarrollo guiado por el comportamiento. De la misma manera, dadas las limitaciones de los frameworks de automatización de pruebas existentes, en 2003 decidió implementar JBehave[7].

[6] Framework en Java para pruebas automatizadas (`http://junit.org`).
[7] Framework en Java para pruebas automatizadas para BDD (`http://jbehave.org/`).

Sacó todo lo que hiciera referencia a pruebas y lo reemplazó por vocabulario relacionado con comportamiento. También definió una forma de escribir los criterios de aceptación muy similar a la forma tradicional de escribir US:

Given (dado) un contexto inicial

When (cuando) ocurre un evento

Then (entonces) espero un resultado

La ventaja que tiene el framework de JBehave es que permite al cliente especificar las pruebas de aceptación en el lenguaje del dominio del problema. Un ejemplo de esto se puede ver en el ejemplo 3.1, donde las pruebas las puede escribir el propio cliente sin aprender demasiados tecnicismos. En el ejemplo 3.2 se puede ver la conexión de las pruebas escritas por el cliente con el sistema en desarrollo. Este framework permite al cliente escribir pruebas de forma fácil e intuitiva y al equipo de desarrollo conectarlas de forma fácil al sistema.

```
1  Scenario: croupier planta con 17 o más
2
3  Given una nueva mano
4  When el croupier tiene 10 y 7
5  Then si debería plantar
6
7  Given una nueva mano
8  When el croupier tiene 6 y 1
9  Then si debería plantar
10
11 Scenario:  croupier pide con menos de 17
12
13 Given una nueva mano
14 When el croupier tiene 8 y 8
15 Then no debería plantar
16
17 Given una nueva mano
18 When el croupier tiene 1 y 5
19 Then no debería plantar
```

Ejemplo 3.1: Ejemplo de JBehave donde el cliente escribe las pruebas en el lenguaje del dominio del problema.

```
1  public class ComportamientoSteps {
2      Croupier croupier;
3
```

```
4       @Given("una nueva mano")
5       public void nuevaMano() {
6           croupier = new Croupier();
7       }
8
9       @When("el croupier tiene $carta1 y $carta2")
10      public void elCroupierTiene(int carta1, int carta2) {
11          croupier.agregarCarta(new Carta(carta1));
12          croupier.agregarCarta(new Carta(carta2));
13      }
14
15      @Then("$valor debería plantar")
16      public void croupierDeberiaPlantar(String valor) {
17          boolean debe_pedir = valor.equals("no");
18          assertThat(croupier.debePedirOtraCarta(), equalTo(
                debe_pedir));
19      }
20  }
```

Ejemplo 3.2: Ejemplo JBehave que muestra la facilidad de conectar las pruebas escritas por el cliente con el sistema en desarrollo.

3.5. STDD

Una variante muy similar es la que propone Rick Mugridge que la llama Story-Test Driven Development (STDD)[8] (Mugridge, 2008). En STDD, el cliente escribe un conjunto de ejemplos para clarificar cuál es el sentido de la US, qué valor le aporta. Estos ejemplos luego se traducen a un formato ejecutable, que se usan como punto de partida para el desarrollo.

Lisa Crispin describe esta práctica de forma muy similar como Customer-Test Driven Development (CTDD) (Crispin, 2005), donde cuenta su experiencia con un grupo de trabajo que usaba UTDD y muchas veces lo que desarrollaban era técnicamente correcto y libre de errores pero no satisfacía las necesidades del cliente. CTDD les permitió mejorar la comunicación entre cliente y grupo de desarrollo, usando las pruebas y ejemplos que daba el cliente para describir las US como punto de partida para el desarrollo.

[8] Una posible traducción podría ser "desarrollo guiado por las pruebas de las US".

3.6. Aclaración de términos

En muchos casos, BDD, STDD y ATDD hacen referencia a lo mismo. En algunos casos pueden presentar sutiles diferencias, pero para los efectos de esta tesis, cualquiera de los 3 se denominará de acá en adelante como ATDD. La idea principal e importante es que en todos los casos se parten de pruebas de alto nivel que están vinculadas con el negocio, y luego se desarrolla con pequeños ciclos UTDD, que usa pruebas más específicas que atacan problemas más puntuales.

4. Cambios en software

No importa el tamaño o complejidad del proyecto, en el mundo del software siempre existe la necesidad de hacer modificaciones. Ya sean nuevos requerimientos, errores detectados o cambios en el negocio, hacen que un producto de software necesite de mantenimiento. Si bien hay distintas definiciones, la más aceptada a la hora de hablar de mantenimiento de software es (ISO/IEC/IEEE 24765:2010,):

> *El proceso de modificar un sistema o componente de software después de haber sido entregado para corregir fallas, mejorar rendimiento u otros atributos, o adaptarlo a cambios en el ambiente.*

Una forma de clasificar los cambios en software es (Warren, 1999):

- Mantenimiento: cambios en el sistema debido a cambios en los requerimientos. El mantenimiento puede ser:
  - Para reparar errores, ya sean de codificación, de diseño o de requerimientos. Los errores detectados en los requerimientos son los más costosos de reparar.
  - Para adaptar el sistema a un ambiente distinto. Puede deberse a un cambio de hardware, plataforma, sistema operativo u otros cambios en software utilizado.
  - Para agregar o modificar funcionalidades. Se da cuando cambian los requerimientos en función de un cambio en la organización o en las reglas del negocio.
- Evolución de la arquitectura. Implica grandes cambios en la arquitectura del sistema, por ejemplo la evolución de un sistema centralizado a una arquitectura cliente-servidor.
- Refactorización. No se agrega nueva funcionalidad, sino que se modifica el sistema para que sea más fácil de entender y modificar en el futuro.

4.1. Leyes de Lehman

Cuando se habla de evolución de software es importante analizar el trabajo de Lehman en el área y entender las implicancias de las leyes que formuló. Primero hizo una clasificación de los distintos tipos de programas, siendo el de mayor interés el que él llama *programas tipo E*, que son aquellos que interaccionan y forman parte del mundo real, cambiándolo y obteniendo una realimentación que lo hace evolucionar (Lehman, 1980). Las leyes que enumeró a continuación se basan en este tipo de sistemas.

A través de diferentes artículos Lehman fue enumerando y corrigiendo distintas leyes basadas en la observación de distintos proyectos. A continuación se muestra un resumen (siempre haciendo referencia a los *programas tipo E*) de dichas leyes (Lehman et al., 1997):

1. Cambio continuo: Los sistemas deben ser adaptados continuamente o se volverán progresivamente menos útiles.

2. La complejidad aumenta: a medida que un sistema evoluciona, su complejidad aumenta a menos que se invierta trabajo en mantenerla o reducirla.

3. Autoregulación: El proceso de evolución de un sistema está autorregulado. Los sistemas grandes tienen una dinámica propia que queda establecida por las decisiones tomadas en etapas tempranas del proyecto. Esto determina la tendencia del mantenimiento del sistema y limita la cantidad de cambios que se le pueden aplicar.

   Los atributos de los sistemas, tales como tamaño, tiempo entre entregas y la cantidad de errores documentados son aproximadamente invariantes para cada entrega del sistema.

4. Conservación de la estabilidad organizativa (velocidad de trabajo invariante): durante la vida útil de un sistema, la velocidad de desarrollo es aproximadamente constante e independiente de los recursos asignados al desarrollo. Esta ley sugiere que la mayoría de los sistemas grandes están en un estado "saturado". Esto significa que cambios en los recursos tienen efectos imperceptibles a largo plazo. Esta ley confirma que equipos muy grandes de desarrollo no son tan productivos porque aumentan los tiempos de organización y comunicación.

5. Conservación de la familiaridad: a medida que un sistema evoluciona, el equipo asociado con él (desarrolladores, usuarios, etc.) debe mantener un alto dominio del problema y comportamiento para lograr que la evolución sea

satisfactoria. Un crecimiento excesivo disminuye ese dominio, por lo tanto, en cada entrega se produce un incremento aproximadamente constante.

6. Crecimiento continuo: se deben agregar funcionalidades continuamente para mantener la utilidad del sistema.

7. Disminución de la calidad: la calidad de los sistemas disminuirá con el tiempo a menos que sean mantenidos y adaptados a los cambios del entorno.

8. Realimentación del sistema: para poder sostener la evolución del sistema minimizando los riesgos de decadencia del mismo, debe haber alguna manera de controlar el rendimiento. Esta ley hace referencia a la importancia de incluir mecanismos de realimentación que permitan hacer mediciones de rendimiento, para poder observar y controlar cómo evoluciona el sistema ante los cambios.

Cabe aclarar que estas leyes fueron hechas a través de un estudio empírico de muchos proyectos grandes que Lehman fue observando por varios años. En la época que se hicieron estas observaciones, estaba muy diferenciada la etapa de desarrollo y la de mantenimiento, siendo claramente más importante la primera y dejando en segundo plano la segunda. En un análisis que hace Karch acerca de las leyes de Lehman y el comportamiento observado en la etapa de mantenimiento, dice que XP (junto con otras metodologías ágiles) empieza a cambiar este modo de ver el desarrollo de un sistema, donde las etapas se solapan, con iteraciones donde se agregan y modifican funcionalidades constantemente (Karch, 2011).

4.2. Costos del mantenimiento

Los costos de mantenimiento varían según la organización y el tipo de sistema desarrollado. Para aplicaciones empresariales, se estima que se gasta un poco más del 50 % en mantenimiento que en desarrollo, mientras que para sistemas empotrados[9] de tiempo real el costo de mantenimiento puede llegar a cuadriplicar el de desarrollo (Guimaraes, 1983). Esto implica que un buen diseño e implementación del sistema puede reducir los costos de mantenimiento, haciendo que el costo total del proyecto se reduzca. Stephen Schach muestra que la proporción del costo del mantenimiento ha ido aumentando con el pasar de los años (Schach, 2010). En la figura 4.1 se pueden ver los costos en cada etapa en 2 períodos distintos.

Una de las razones por las cuales los costos de mantenimiento son tan altos es porque es mucho más costoso agregar o arreglar funcionalidad cuando el sistema

[9]Sistema diseñado para realizar unas pocas funciones específicas, frecuentemente en un sistema de tiempo real con hardware dedicado.

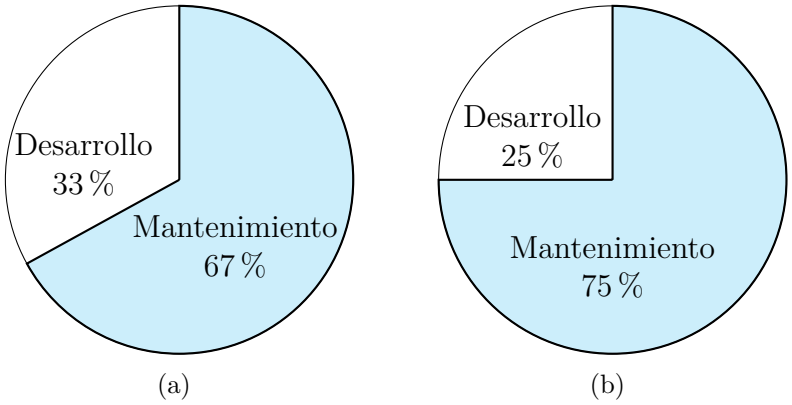

Figura 4.1: Costos aproximados en la etapa de desarrollo y en la etapa de mantenimiento posterior (Schach, 2010). (a) período de 1976 a 1981, (b) período de 1992 a 1998

está en producción que cuando está en la etapa de desarrollo (Sommerville, 2000). Algunos de los factores que explican este comportamiento son los siguientes:

1. Luego de que el sistema fue entregado, es normal que el equipo se divida y se formen nuevos grupos para trabajar en nuevos proyectos. La gente encargada de mantener el sistema no siempre entiende completamente el sistema ni muchas de las decisiones tomadas en el diseño. Esto hace que se dedique mucho esfuerzo en entender el sistema antes de hacer cambios.

2. A veces el proyecto del desarrollo del sistema y del mantenimiento del mismo está divido en 2 contratos separados, incluso algunas veces la responsabilidad de cada parte recae en empresas diferentes. Esto puede provocar que no haya un incentivo para desarrollar un sistema de forma prolija, siendo que luego el equipo que lo desarrolló no tendrá que mantenerlo.

3. Es común que las tareas de mantenimiento sean asignadas a gente con menor experiencia (*junior*), ya sea a modo de entrenamiento o porque se necesitan a los de mayor experiencia (*senior*) para el desarrollo de proyectos más urgentes.

4. Con el paso de los años la estructura de un sistema tiende a degradarse, volviéndose más difícil de entender y cambiar.

En uno de los artículos donde se analizan los efectos que tuvo el famoso Y2K[10], Kappelman menciona alguno de los motivos y consecuencias (Kappelman, 2000). Entre las consecuencias económicas, se estima que se gastaron entre US$ 375.000 millones y US$ 750.000 millones. Cerca del 45 % de las aplicacio-

nes tuvieron que modificarse y un 20 % reescritas completamente.

Este hecho hizo que las organizaciones aprendieran varias lecciones y se replantearan cuestiones como el tiempo dedicado al testing, la buena documentación, diseño modular y entender que un proyecto de software no termina con el desarrollo, sino que continúa con el mantenimiento.

4.3. TDD y mantenimiento

TDD mitiga varios de los problemas mencionados. Por un lado, produce software desacoplado, fácil de cambiar y mantener ya que se cuenta con el soporte de las pruebas para saber si un cambio afecta al resto del sistema. Por otro lado, las pruebas son una excelente documentación para entender el sistema. Si bien es necesario dedicar tiempo para comprenderlo, las pruebas son una excelente guía para saber cómo funciona el sistema y para entender ciertas decisiones de diseño. Por último, con TDD no se degrada la estructura del sistema con el tiempo, sino que permite que evolucione sin perder el uso de las buenas técnicas de desarrollo de software.

Hay un estudio que intenta mostrar en términos económicos los beneficios de desarrollar con TDD en vez de la forma tradicional (Müller y Padberg, 2003). El estudio que hace Müller toma las siguientes hipótesis:

- El desarrollo con TDD es más lento.
- Con TDD se obtiene código de mayor calidad.

Una versión simplificada de lo que proponen se puede ver en la figura 4.2. Si bien el tiempo de desarrollo en TDD es más largo, dentro de este ciclo se encuentran y arreglan más defectos que en el ciclo de desarrollo tradicional. En la forma tradicional, una vez finalizada la etapa de desarrollo agregan una etapa de aseguramiento de la calidad (QA por sus siglas en inglés) para encontrar y reparar todos aquellos errores que se encontraron con TDD y no de la forma tradicional. En el estudio se plantea en qué situaciones es más beneficioso TDD de acuerdo a las velocidades de desarrollo y la calidad del código.

La idea de este estudio muchas veces es extendida entre los que defienden el uso de TDD. Aún suponiendo que la etapa de desarrollo en TDD es mayor que en la forma tradicional (aunque hay quienes sostienen que esto no es cierto), ese esfuerzo extra se ve luego compensado con la ganancia que se obtiene a la hora de mantener el sistema. Siendo que la etapa de mantenimiento puede llegar a ser igual

[10]Del inglés “Year 2000”, fue un caso muy conocido de un error en los sistemas de software que utilizaban 2 dígitos para representar el año en una fecha. La consecuencia de esto era que el día siguiente al 31 de dicimebre de 1999 seria el primero de enero de 1900.

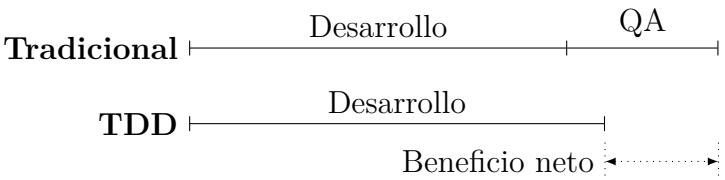

Figura 4.2: Beneficio de desarrollar con TDD en vez de la forma tradicional

o bastante mayor a la de desarrollo, un ahorro en dicha etapa se transforma en un ahorro en el costo total del proyecto. Sommerville hace una conclusión similar al decir que al desarrollar un sistema teniendo en cuenta que habrá que mantenerlo, si bien toma más esfuerzo al principio, luego cuesta menos mantenerlo (Sommerville, 2000).

4.3.1. TDD y leyes de Lehman

En la sección 4.1 se enumeraron las leyes de Lehman. A continuación se muestra como TDD ayuda a mitigar o resolver varios de los problemas que se tratan en dichas leyes:

1. Cambio continuo: TDD está pensado para ser utilizado en un entorno cambiante. La adaptación a los cambios es una de las características distintivas de las metodologías ágiles.

2. La complejidad aumenta: En TDD se hace una refactorización constante, haciendo que el código sea lo más sencillo posible. El trabajo que menciona Lehman para que la complejidad se mantenga está incluido en cada ciclo de trabajo.

3. Conservación de la familiaridad: las pruebas proveen una documentación actualizada en todo momento del sistema, además de ser mucho más fácil de entender que una documentación formal. Esto hace que al crecer un sistema, la familiaridad sea más fácil de recuperar.

4. Disminución de la calidad: está muy relacionado con “la complejidad aumenta”, y TDD ataca este problema de la misma manera.

5. Realimentación del sistema: si bien Lehman hace más referencia al uso de métricas, la realimentación del cliente en todas las iteraciones permite acomodar el rumbo del sistema cada vez que este se desvíe. Como esta realimentación es frecuente, el desvío es mínimo.

4.4. ATDD y mantenimiento

En la sección anterior se habló de cómo TDD (en su definición más general) ayuda a resolver algunos de los problemas relacionados con el mantenimiento. En el caso de ATDD donde el punto de partida son las pruebas de aceptación definidas con el cliente, hay algunas consecuencias más.

4.4.1. Lo que el cliente necesita

Como ya se ha dicho, las pruebas son una excelente documentación. En el caso del código, las pruebas unitarias son el mejor ejemplo de cómo se usa un determinado método o clase. En el caso de las pruebas de aceptación, son el mejor ejemplo de cómo debe comportarse el sistema, qué es lo que se espera de él. Es una forma muy clara de expresar qué es lo que el cliente necesita. A la hora de mantener un sistema nunca hay que perder este enfoque.

En las pruebas de aceptación quedan reflejadas las decisiones tomadas en el pasado. Además, estas pruebas evolucionan, van mejorando continuamente. Estas decisiones están presentes todo el tiempo, cada vez que se ejecutan las pruebas. Es mucho más rápido y claro mirar y ejecutar las pruebas de aceptación que leer documentación extensa.

> En uno de sus artículos Robert Martin dice que las pruebas de aceptación son como guías o soportes (Martin, 2005b). Estas guías permiten que uno permanezca centrado en la idea general de lo que se quiere implementar.
>
> Hace una comparación con una operación en el cerebro que vio por televisión. La operación en cuestión consistía en introducir una especie de alfiler hasta una zona determinada y luego calentar la punta para destruir el grupo de neuronas problemático. La mayor parte del trabajo radicaba en colocar las guías que luego permitirían desplazar la aguja con mínimo margen de error. Este trabajo lleva mucho tiempo y esfuerzo, pero una vez colocadas, el resto del trabajo era muy sencillo.
>
> En el caso de las pruebas de aceptación es muy parecido. Son las guías y el soporte que luego permiten construir el sistema, asegurando que haga lo es necesario, y solamente eso.

La ventaja extra que se obtiene al desarrollar con ATDD es que estas pruebas no sólo están disponibles en todo momento, sino que además se mantienen actualizadas, permitiendo que el cambio sea más sencillo y respetando lo que el cliente necesita.

4.4.2. Cambios más seguros

ATDD presenta otras ventajas con respecto a UTDD:

- Granularidad de sus pruebas.
- Distintos tipos de pruebas.

En cuanto a la granularidad de las pruebas, los distintos niveles ofrecen distinto soporte a la hora de realizar cambios. No siempre es posible realizar cambios sin afectar a las pruebas de menor granularidad, ya que suelen estar más acopladas. En este caso los cambios dejan de ser seguros, pero al contar con distintos niveles de pruebas se vuelve a contar con un marco que permite cambios seguros. El trabajo de Fontela (que se explica en mayor detalle en la sección 7.2) ofrece una práctica metodológica para aplicar refactorizaciones seguras haciendo uso de la distinta granularidad en las pruebas (Fontela, 2013).

El otro punto a favor son los distintos tipos de pruebas. Si bien hay varias similitudes con otros tipos de pruebas (como ser pruebas unitarias o de integración), su concepción es completamente distinta. Las pruebas de aceptación surgen de una charla con el cliente, donde las dos partes están involucradas (cliente y equipo de desarrollo). Para poder arribar a este tipo de pruebas es necesario un entendimiento mutuo del sistema y forman una especie de contrato. El cliente es el dueño de las pruebas de aceptación y todo cambio en las mismas debe estar acordado.

El objetivo de esta tesis es demostrar que ATDD es más seguro ante los cambios por los distintos niveles de granularidad y tipos de pruebas que están presentes. En el capitulo 5 se explica en profundidad este concepto.

5. Propuesta de tesis para el uso de TDD más seguro

Esta tesis busca encontrar cuánto impactan los cambios en sistemas desarrollados con distinta granularidad y de esta manera saber qué tipo de aplicaciones están mejor preparadas. El impacto de los cambios es de particular interés ya que se suele destinar más del 50 % de los recursos (principalmente tiempo y dinero) en el mantenimiento de un sistema a lo largo de su vida útil (según se vio en la sección 4.2).

Un sistema que fue desarrollado siguiendo ATDD cuenta con pruebas de distinta granularidad, brindando mayor soporte y seguridad ante los cambios. Esta seguridad está dada por las pruebas de aceptación que son más abarcativas y menos cambiantes que el resto de las pruebas. Este tipo de sistemas brinda mayor seguridad y confianza ante cambios que un sistema desarrollado con UTDD ya que éste último cuenta con sólo un nivel de pruebas.

5.1. Cambios en sistemas sin pruebas

Un sistema sufre cambios de todo tipo a lo largo de toda su vida útil. Realizar un cambio sobre un sistema sin soporte por ningún tipo de prueba es un salto de fe, ya que no hay forma de saber si ese cambio no alteró de forma negativa el comportamiento del sistema. Este problema se da independientemente de la magnitud y del alcance del cambio.

La falta de pruebas no sólo no permite saber si el cambio en cuestión fue realizado en forma correcta, sino que tampoco hay forma de asegurar que no se alteraron otros componentes o funcionalidades del sistema. Esto se puede volver aún peor si en principio no parece haber ningún error y se detecta mucho más tarde. Hay estudios que muestran que cuanto más tarde se detecta un error, más costoso es repararlo (Boehm, 1981). Kent Beck argumenta que el diagrama que propone Boehm donde el aumento del costo es exponencial no es válido en el esquema de

XP (Beck y Andres, 2004), sino que la curva, a pesar de seguir siendo creciente, se vuelve mucho más suave.

Las implicancias de no saber cómo impacta un cambio (ya sea agregar, modificar o eliminar funcionalidad) son muchas y variadas.

Por un lado, puede afectar de forma sustancial la planificación. Por ejemplo, en el caso de una metodología de desarrollo por ciclos, si no se conocen los errores que introdujo un cambio para la planificación del ciclo, podría no llegar a completarse la funcionalidad acordada para el mismo o podría no cumplirse con el plazo.

Otro efecto que tiene no saber de qué manera impacta un cambio, es que genera poca confianza realizar una modificación: "si funciona, es mejor no cambiarlo". Al no contar con pruebas que respalden el funcionamiento del sistema, el nivel de incertidumbre es alto, por lo tanto el riesgo de hacer un cambio y que éste impacte negativamente en el sistema, es alto. En consecuencia, tiene sentido pensar que los cambios que se realicen sean en su mayoría para reparar errores o agregar funcionalidad. En este contexto de incertidumbre, las refactorizaciones quedan en segundo plano, ya que son riesgosas y "no aportan nada nuevo". Sin embargo esto no es así, ya que al no usar refactorizaciones, el sistema se degrada con el tiempo como lo menciona Lehman (ver sección 4.1).

Muy relacionado con el punto anterior es la motivación y confianza del equipo de desarrollo. Un entorno donde hay mucha incertidumbre es un entorno incómodo, cada cambio que un desarrollador hace puede llegar a romper el sistema y no saberlo. Este contexto es poco motivador e incluso puede tornarse frustrante. Es muy importante la motivación de un equipo.

Alguien que conoce la suma importancia de esto es Google: ofrece a sus empleados todo tipo de comodidades para que sus trabajadores se sientan a gusto (Cook, 2012). El objetivo es sencillo: atraer gente de gran capacidad, retener el talento y motivar a sus empleados. Hay estudios que muestran que los humanos tienen distintos tipos de necesidades que van más allá de lo monetario, como necesidades sociales, satisfacción personal, entre otros (Ivancevich et al., 2008).

Los empleados cuentan con contención de todo tipo:

- Salas de masajes, de descanso, salas de juegos, gimnasio.
- Beneficios para madres y padres con hijos recién nacidos.
- Posibilidad de llevar a sus hijos y mascotas.
- Grupos pequeños para maximizar la creatividad con gran independencia.

Como conclusión, la motivación del personal es un aspecto muy importante y esto no se logra solamente con dinero.

5.1.1. Un ejemplo concreto

Para aclarar cuál es el problema mencionado en sistemas sin pruebas, a continuación se ejemplifica con un caso sencillo.

Una aplicación tiene implementado y hace uso de un algoritmo de ordenamiento, por ejemplo *bubble sort*. En teoría de algoritmos se estudia la complejidad computacional de distintos algoritmos. Sin entrar en detalles, la complejidad computacional sirve para clasificar algoritmos y poder compararlos. De estos estudios se desprende que el algoritmo *quicksort* es más rápido que *bubble sort*, y esto es más notable cuanto más grande es el conjunto a ordenar. Como contrapartida, *bubble sort* es más fácil de entender y programar que *quicksort*.

Si se decide cambiar el algoritmo de ordenamiento por *quicksort*, se corre el riesgo que deje de funcionar el algoritmo de ordenamiento, del cual depende el resto de la aplicación. Esto ocurre porque no hay ninguna prueba que verifique que el sistema sigue funcionando igual que antes. En el caso de introducir un error al hacer la implementación, el problema se puede manifestar inmediatamente, o aún peor, tiempo más adelante por situaciones límites del algoritmo no contempladas. El error puede ser fácilmente identificable o tener un síntoma completamente distinto que hace que sea más difícil de encontrar.

Esta situación planteada no es tan rara como parece. En el ejemplo, el cambio puede haber sido para mejorar la velocidad del algoritmo. Sin embargo, si el sistema sólo utiliza el ordenamiento pocas veces sobre pocos registros, el impacto del cambio es insignificante.

5.2. Cambios en sistemas desarrollados con UTDD

Luego de la sección anterior, queda claro que el desarrollo sin pruebas es demasiado riesgoso. Es por eso que ahora se plantea la situación en donde existen pruebas unitarias. Se plantea primero la situación de sólo pruebas unitarias ya que históricamente los primeros frameworks de pruebas automatizadas fueron pensados para pruebas unitarias.

Como se explicó en la sección 2.1, el ciclo de UTDD comienza con una prueba unitaria que falla y luego se implementa el código que hace que esa prueba pase. Solamente se debería escribir código para hacer pasar una prueba, no más. Esto quiere decir que cada porción de código está cubierta por una prueba. Además, estas pruebas deben ser automatizadas.

Si se cuenta con un buen conjunto de pruebas automatizadas que cubren todo el sistema, es fácil detectar cuando un simple cambio en el código provoca alguna falla en el sistema. Puede detectar fallas directas como indirectas (porciones de código erróneo que impactan en otra parte del sistema).

5.2.1. ¿Pruebas al principio o al final?

Según lo explicado anteriormente, se podría pensar que esta cobertura se da independientemente de si se usa TDD o simplemente se crean las pruebas luego del código.

Sin embargo, esto no es necesariamente así. Por un lado, al desarrollar con TDD todo el código existente es producto de una prueba que necesita ese código. En cambio, al realizar pruebas al final esto no tiene porqué ocurrir. Se podría objetar que se podría conseguir el mismo juego de pruebas independientemente del momento en que se las escriba, pero:

- Seguir el ciclo TDD asegura que el código se derive únicamente de las pruebas, mientras que hacer las pruebas al final no (Beck, 2002).
- En la práctica es frecuente que los sistemas que hacen pruebas al final no cubran todo el código (Beck y Gamma, 2000). La excusa suele ser "no tengo tiempo". Esto se vuelve pronto en un círculo vicioso: a mayor presión, menos pruebas se escriben; al escribir menos pruebas, disminuye la productividad y la estabilidad del sistema; al disminuir la productividad y al ser menos estable el sistema, mayor es la presión.
- En la práctica no siempre se hacen las pruebas de forma minuciosa y rigurosa. Tener 100 % de cobertura en las pruebas no implica que esas pruebas sean buenas (Cornett, 1996). Esta afirmación es muy importante y hay que tenerla siempre presente, ya que se puede tener una falsa sensación de seguridad que puede ser peligrosa. Siempre es mejor tener un buen conjunto de pruebas que no necesariamente den un 100 % de cobertura que un conjunto de pruebas malo que den un mayor porcentaje de cobertura.
- En la práctica, si el que realiza las pruebas al final es el mismo que el que escribió el código, indefectiblemente está sesgado (Beck y Andres, 2004). Este tipo de pruebas suelen probar lo que ya se sabe que anda bien. Beck menciona casos en los que la gente hace pruebas sólo porque están obligados a hacerlas y hay alguien controlando que se hagan. En este contexto tiene sentido pensar que la calidad de las mismas no siempre es la mejor.

Este tema fue de particular interés para un grupo que realizó un estudio que intenta demostrar la efectividad de realizar las pruebas antes que el código en vez

de realizarlas después (Erdogmus et al., 2005). En este estudio hay 2 grupos que Erdogmus llama *Test-First* y *Test-Last* haciendo referencia a los que hacen las pruebas antes del código y a los que hacen las pruebas después del código respectivamente. Allí comenta que en todos los casos, los grupos *Test-First* escribieron muchas más pruebas que los grupos *Test-Last*, habiendo incluso algunos de los *Test-Last* que no llegaron a escribir pruebas.

Como conclusión, si bien se podría pensar que se puede llegar al mismo conjunto de pruebas independientemente del momento en que se escriban, en la práctica es muy difícil que esto ocurra.

Una técnica de diseño

Hay otro punto a tener en cuenta: TDD es una técnica de diseño, no una técnica de testing. Por lo tanto, si se llega al mismo conjunto de pruebas haciendo pruebas al principio o al final no alcanza para decir que el sistema es bueno. Realizar las pruebas al principio implica pensar en un diseño modular, fácil de probar, desacoplado. Además de guiar el diseño, el conjunto de pruebas resultantes sirve para verificar que el resultado obtenido es el esperado, y brinda la posibilidad de realizar refactorizaciones seguras. En cambio, realizar pruebas al final apunta a la verificación y a lo sumo brinda un soporte a las refactorizaciones.

5.2.2. Un ejemplo concreto

En la primera parte de este capítulo se mencionó el caso de realizar un cambio sin el soporte de pruebas y los riesgos implicados.

En este caso se verá qué sucede cuando el desarrollo fue hecho con UTDD. Los pasos a seguir son los 3 pasos de UTDD:

- En caso de ser necesario, se realizan nuevas pruebas. En el ejemplo del algoritmo de ordenamiento, puede bastar con las pruebas que ya se tenían, o pueden ser necesarias nuevas pruebas que contemplen algunos casos especiales a tener en cuenta en el nuevo algoritmo.
- Implementar nuevo algoritmo.
- Ejecutar pruebas. Lo importante de contar con pruebas automatizadas es que rápidamente se puede verificar que no sólo el algoritmo esté bien, sino que el resto del sistema siga funcionando correctamente.

El proceso suele ser iterativo y los pasos se repiten. Por ejemplo, no siempre pasan todas las pruebas de entrada, en ese caso debería corregirse la implementación. También pueden surgir nuevos casos de prueba. Lo importante es que una

vez terminado el ciclo y todas las pruebas pasan, el cambio fue satisfactorio y las pruebas aseguran que todo el sistema se comporta como corresponde.

En el primer ejemplo mencionado donde no existen pruebas, no hay forma de saber cuál fue el impacto del cambio. En el caso de UTDD, se puede saber si la aplicación se comporta de manera diferente a la esperada.

5.2.3. Otro ejemplo

Este ejemplo es muy parecido al anterior. La situación es la siguiente: en un sistema desarrollado con UTDD, se tiene un algoritmo de búsqueda secuencial y se lo quiere cambiar por una búsqueda binaria. Para entrar un poco en contexto, se explicarán brevemente ambos algoritmos.

El caso de la búsqueda secuencial es el más sencillo: se compara elemento por elemento en forma secuencial (empezando por el primero y continuando por el siguiente) hasta encontrar el elemento buscado, o determinar que el elemento no se encuentra en el conjunto dado.

En el caso de la búsqueda binaria es necesario que el arreglo en donde se busca esté ordenado. Estando el arreglo ordenado, se toma el elemento que se encuentra en la mitad del arreglo y se compara con el elemento a buscar. Si el elemento no es el buscado, en el caso de ser mayor, se repite el proceso con la mitad superior, en el caso de ser menor con la mitad inferior. Así sucesivamente hasta encontrar el elemento o determinar que el elemento no existe.

Siguiendo con el caso de desarrollo con UTDD, seguramente se cuente con casos de pruebas unitarias para distintas situaciones:

- Buscar un elemento en un conjunto vacío.
- Buscar un elemento existente en un conjunto con un elemento.
- Buscar un elemento existente en un conjunto con varios elementos.
- Buscar un elemento no existente en un conjunto con varios elementos.
- Otras.

Al igual que en el ejemplo anterior, los pasos a seguir son los mismos y el resultado esperado sería el mismo que antes: si todas las pruebas pasan, no hay un impacto negativo en el sistema.

Sin embargo, puede darse que todas las pruebas unitarias pasen y aún así el sistema no responda de la misma manera que antes. De hecho Donald Knuth dice (Knuth, 1997):

> *Aunque la idea básica de búsqueda binaria es comparativamente sencilla, los detalles pueden ser sorprendentemente complicados.*

Tal es así, que la implementación de la búsqueda binaria utilizada en Java tuvo un error por más de 20 años sin ser detectado (Bloch, 2006). En este caso la falla ocurrió al utilizar la búsqueda binaria sobre arreglos muy grandes (miles de millones de elementos o más). No sólo las pruebas pasaban, sino que el algoritmo fue ampliamente utilizado por millones de aplicaciones y el error pasó inadvertido.

5.2.4. Problemas con cierto tipo de refactorizaciones

Como se mencionó en la sección 2.1, las refactorizaciones son una parte fundamental del ciclo TDD. Fowler da una muy buena definición de refactorización (Fowler, 1999):

> *Refactorización es el proceso de cambiar un sistema de software de tal manera que no se altera su comportamiento externo del código, pero que mejora su estructura interna.*

Hay muchas razones por las cuales es importante realizar refactorizaciones (Fontela, 2013):

- Aplicación de principios de diseño. Martin Fowler habla de situaciones con "malos olores" ("bad smells" o "code smells") que son buenos puntos de partida para refactorizaciones (Fowler, 1999). En su libro menciona muchos tipos de refactorizaciones para las diferentes situaciones.

- Prevenir la degradación del sistema. Lehman menciona que con el tiempo los sistemas se degradan, con lo cual se vuelve más difícil mantenerlos (ver sección 4.1). Brooks está de acuerdo con Lehman y dice que la calidad decae con el tiempo, y que cada vez se gasta más tiempo en corregir errores introducidos en correcciones anteriores (Brooks, 1975). Según él, llega un momento donde "por cada paso que se da hacia adelante, se da uno hacia atrás". Al realizar refactorizaciones frecuentes se puede mantener la calidad interna del código y así evitar que el sistema se vuelva inmanejable. Como menciona Rod Hilton en su tesis, la calidad interna del código es de vital importancia para que un sistema prospere (Hilton, 2009).

- Evolución de la arquitectura. Como ya se ha mencionado, Robert Martin explica que la arquitectura de un sistema no puede ser completamente pensada desde un comienzo, sino que esta evoluciona según los requerimientos

(Martin, 2005c). Es por eso que la refactorización juega un rol clave para poder permitir esta evolución.

- Oportunidad de refactorización. El resultado de hacer refactorizaciones con buena frecuencia es un código prolijo, fácil de entender y mantener. En TDD se propone primero implementar buscando que la prueba pase, sin tener tanto en cuenta la prolijidad, y luego refactorizar para emprolijar el código.

Sin embargo, hay ciertos tipos de refactorizaciones en las que hay que tener especial cuidado.

Un primer ejemplo

Un ejemplo sencillo son las refactorizaciones que involucran cambios de nombre en métodos o clases. Siendo que las pruebas unitarias usan estos mismos métodos y clases, estos renombres implican cambiar las pruebas. Algunos casos pueden ser triviales y otros un poco más complejos, pero en todos los casos se estarían cambiando código y pruebas al mismo tiempo. Así como inicialmente se mencionó que cambiar código sin ninguna prueba de respaldo implicaba un riesgo, este caso es similar.

Un caso claro donde pueden ocurrir problemas al realizar cambios en nombres es cuando se usa reflexión[11]. La forma más fácil de ver esto es con un ejemplo:

En FitNesse se escriben las pruebas de aceptación en un formato de tablas en texto plano. Por ejemplo:

```
|Division                                |
| numerador | denominador | cociente? |
| 10        |  2          | 5.0       |
| 3         |  2          | 1.5       |
| 100       |  4          | 25.0      |
```

Para poder conectar estas pruebas con el sistema para que haga uso del código real, se deben crear clases y métodos respetando ciertas reglas. En el ejemplo mostrado y usando Java, se debe crear una clase llamada `Division` con los métodos `setNumerador`, `setDenominador` y `cociente`.

```
1 class Division {
2   ...
3   public void setNumerador(double numerador) {
4     this.numerador = numerador;
5   }
6
```

[11] Capacidad de un programa de revisar y modificar la estructura o comportamiento de un objeto en tiempo de ejecución.

```
7    public void setDenominador(double denominador) {
8      this.denominador = denominador;
9    }
10
11   public double cociente() {
12     return numerador/denominador;
13   }
14 }
```

Ejemplo 5.1: Clase que conecta la tabla en texto plano con el código real.

Si se cambia alguno de estos nombres, la conexión entre FitNesse y el sistema deja de andar.

Otros ejemplos

Hay refactorizaciones que pueden ser mucho más graves, por ejemplo, el cambio de firma de un método. Si a un método se le agregan o quitan parámetros, las herramientas de refactorización automática no son capaces de manejarlo. Y puede ser incluso peor si se cambian de orden los parámetros: el sistema podría seguir compilando pero el comportamiento es completamente diferente. Esto se puede ver en el ejemplo 5.2. En dicho ejemplo se ve que se cambia el orden de los parámetros del constructor. En el caso de Ruby[12], al ser un lenguaje no tipado este tipo de cambios es muy difícil de detectar sin pruebas.

```
1 class Carta
2     # def initialize numero, palo, valor
3     def initialize palo, numero, valor
4         @numero = numero
5         @palo = palo
6         @valor = valor
7     end
8 end
```

Ejemplo 5.2: Cambio en el orden de los parámetros del constructor de una clase en Ruby.

En estos ejemplos de refactorizaciones, se da que los cambios hechos en código también implican realizar cambios en las pruebas al mismo tiempo. Estos casos son más frecuentes de lo que uno esperaría, y están dados por la naturaleza de las pruebas unitarias, que cambian con frecuencia y están muy ligadas al código.

Cuando se cambia código y pruebas unitarias al mismo tiempo, se vuelve a caer en el mismo problema del principio: ¿cómo se puede asegurar que no se está

[12]Lenguaje de programación orientado a objetos interpretado.

afectando ninguna parte del sistema con el cambio?

5.2.5. Requerimientos de usuarios

Hay otro tipo de problemas que pueden ocurrir y son aquellos que las pruebas unitarias no abarcan. Un ejemplo típico es el rendimiento. Hay cierto tipo de aplicaciones donde el tiempo de respuesta debe ser inferior a una cierta cota, o donde debe ser lo menor posible. Por ejemplo, en lo que respecta a buscadores web, un minuto para devolver el resultado de la búsqueda es inadmisible ya que hoy en día la mayoría tarda pocos segundos.

Estos requisitos de rendimiento están relacionados con el negocio o problema en particular y no son los únicos. Otros ejemplos podrían ser:

- Cambios relacionados con la usabilidad. Por ejemplo: un diario en línea puede requerir que el usuario pueda acceder a cualquier sección del diario siguiendo un único hipervínculo en todo momento y todo lugar.
- Sistemas dependientes de conexiones (Internet, wi-fi, bluetooth, etc.). En estos casos hay que tener en cuenta los tiempos de respuesta, pérdida de conexión, reconexión, seguridad en la transmisión de datos, entre otros.
- Sistemas multiusuarios. El ejemplo típico son los sitios web. Si bien los servicios ofrecidos pueden ser los mismos, la forma de implementar un sitio puede ser completamente diferente si está pensada para unos pocos cientos de usuarios o millones de usuarios.
- Sistemas en tiempo real. En este tipo de aplicaciones hay ciertas restricciones que respetar como ser tiempos máximos de respuesta.

En todos los casos, son situaciones que con las pruebas unitarias no alcanza para indicar el comportamiento esperado. Para estos casos son necesarias pruebas de aceptación, ya que son situaciones pertinentes al negocio. Si un portal web está pensado para que accedan cientos de personas o millones al mismo tiempo, debe estar especificado en forma de pruebas de aceptación, son deseos y necesidades que el cliente expresa.

5.3. Cambios en sistemas desarrollados con ATDD

El ciclo de ATDD comienza con pruebas de aceptación automatizadas definidas por el cliente o en conjunto con el cliente. Estas pruebas son de mayor granularidad que las pruebas unitarias ya que suelen describir el comportamiento de distintas áreas del sistema involucrando a varias partes interactuando entre sí.

Las pruebas de aceptación describen cuáles son las condiciones que debe cumplir el sistema para que una US esté completa. Son una especie de contrato acordado entre el cliente y el equipo que desarrolla el sistema. A diferencia de las pruebas unitarias, el dueño de las pruebas de aceptación es el cliente. Por otro lado, un cambio en los requerimientos se lo debe evaluar, analizar el impacto del mismo y acordar cómo se va a resolver.

Dada la naturaleza de las pruebas de aceptación, éstas son mucho menos cambiantes. Por un lado porque están directamente asociadas con las reglas del negocio, que si bien pueden cambiar, están sujetas a condiciones externas relativamente estables. Esto se debe a que un cambio en los procesos de negocio es costoso y lleva tiempo. Por ejemplo, una fábrica no va a cambiar la forma en que se realiza la producción de un momento a otro. Estos cambios no solo implican adaptar el sistema de software, sino también se debe adaptar la propia compañía.

Por otro, las modificaciones son cuidadosamente planificadas y discutidas al momento de elaborarlas. Surgen de la planificación del próximo ciclo y tienen la visión de cada parte: el cliente aporta su punto de vista y los analistas el suyo, logrando ambos entender qué es lo que se quiere hacer y cómo se validará.

Entonces, en el contexto del desarrollo con ATDD, se tienen pruebas de distinta granularidad: un nivel de pruebas de granularidad fina que aportan las pruebas unitarias y otro nivel de granularidad mayor que aportan las pruebas de aceptación, con la diferencia que estas últimas son mucho menos cambiantes. De esta manera se agrega un nivel más de cobertura ante los cambios, dando el mismo soporte y confianza que las pruebas unitarias le dan a los cambios en código.

5.3.1. Un ejemplo concreto

Siguiendo con el ejemplo de la búsqueda binaria, en el caso de UTDD se vio que hay ciertas situaciones que las pruebas unitarias no abarcan.

En el caso de desarrollar con ATDD, si es importante para el negocio que la búsqueda pueda soportar miles de millones de elementos, estará plasmado en pruebas de aceptación. De este modo, al cambiar de un algoritmo a otro, no sólo se notará si se implementó correctamente, sino que además se notará si afecta los requerimientos pedidos por el usuario.

Por otro lado, las pruebas de aceptación son de mayor granularidad, es decir, son más abarcativas. Hacen uso de distintas partes del sistema para lograr un objetivo.

5.3.2. Refactorizaciones y ATDD

Cuando se hizo referencia a las refactorizaciones y UTDD se mencionó que al cambiar código y pruebas unitarias al mismo tiempo se vuelve al problema inicial

en el cual se hace un cambio y no hay pruebas que le den un sustento para que ese cambio sea seguro.

TDD no es una técnica de *testing*, sino que es una técnica de diseño. Las pruebas guían el diseño y posteriormente el desarrollo. Es natural que las clases ligadas con las pruebas de aceptación sean aquellas relacionadas con el negocio. En principio se trata de utilizar los mismos nombres del negocio en los métodos y nombres de clases. Si bien puede haber cambios, son menos comunes. Por ejemplo, es más difícil que cambie una clase de nombre `CuentaCorriente` que está más relacionada con el negocio a que cambie una clase de más bajo nivel relacionada con detalles de implementación.

Sin embargo el hecho de tener pruebas de diferente granularidad y que algunas pruebas tengan menor frecuencia de cambio, no quiere decir que dicho sistema esté exento de riesgos. Puede suceder que un cambio afecte todos los niveles: código, pruebas unitarias y pruebas de aceptación. La diferencia radica en la frecuencia y en cómo se dan estos cambios.

Siendo que las pruebas de aceptación están ligadas al negocio y son acordadas con el cliente, ya se mencionó que la frecuencia de cambio es menor. Por otro lado, cuando es necesario hacer una modificación a nivel negocio, este cambio es consensuado entre el cliente y el equipo de desarrollo. Ambas partes están involucradas y debe haber un análisis del impacto y costo que significa realizar ese cambio.

5.4. ATDD está mejor preparado que UTDD frente a cambios

A lo largo de este capítulo se analizaron 3 casos y cómo reaccionan ante cambios:

1. Sistemas donde no hay pruebas de ningún tipo.
2. Sistemas donde sólo hay pruebas unitarias (UTDD).
3. Sistemas donde hay pruebas unitarias y de aceptación (ATDD).

El primer caso claramente es el más riesgoso de todos. Cada modificación en el sistema está sujeta al riesgo de no saber si hay una alteración sobre el funcionamiento anterior.

En el segundo caso hay un nivel de pruebas que permiten hacer cierto tipo de modificaciones y seguir "cubierto", en el sentido que si las pruebas pasan el sistema se sigue comportando igual. Sin embargo se planteó el caso que una modificación de código implique cambiar también las pruebas unitarias. Al modificar al mismo tiempo código y pruebas unitarias se pierde la cobertura que se había ganado por desarrollar con UTDD.

Por último, el tercer caso agrega un nivel de pruebas más amplio. Este nivel de pruebas da soporte a cambios en código y pruebas de menor nivel (unitarias). Puede existir la situación en que deban hacerse cambios en los 3 niveles, que en principio representa un problema ya que no se cuenta con un nivel superior que le de soporte a los cambios.

5.4.1. Planteo de tesis

Esta tesis plantea que ATDD está mejor preparado para los cambios que UTDD ya que cuenta con 2 niveles de pruebas, y las pruebas de aceptación son menos cambiantes que el resto de los niveles. Además, siendo que las pruebas de aceptación pertenecen al cliente, cualquier cambio en las mismas debe ser analizado y acordado. Esto provoca una mayor atención a este tipo de cambios.

Se analizarán proyectos desarrollados con ATDD observando los cambios a través de la historia del sistema (analizando el sistema de control de versiones utilizado) y se le prestará especial atención a los casos en que:

1. Cambia solamente código. En este caso, tanto UTDD como ATDD responden de la misma manera.

2. Cambian código y pruebas unitarias. En este caso, si sólo se contase con pruebas unitarias (UTDD) representaría un problema mientras que teniendo pruebas de aceptación (ATDD) estaría cubierto.

3. Cambian código, pruebas unitarias y pruebas de aceptación. En este caso, tanto UTDD como ATDD estarían frente a un potencial problema.

El resultado esperado es que el tercer caso se da con menor frecuencia que el primero y el segundo.

6. Casos de estudio

6.1. Selección proyectos

Para estudiar el comportamiento de los cambios se buscaron sistemas desarrollados con ATDD. Para poder realizar el estudio fue necesario tener acceso a la historia del proyecto y el código (a través de un sistema de control de versiones). Por este motivo se buscó en sistemas de código abierto.

6.1.1. Encuesta

Se envió una encuesta a múltiples proyectos con las siguientes preguntas (se detallan las más importantes):

1. Nombre del proyecto y enlace al repositorio.
2. Lenguaje en el que está desarrollado el proyecto.
3. Metodología utilizada para el desarrollo. Opciones posibles:
   - TDD
   - ATDD
   - UTDD
   - BDD
   - NDD
   - STDD
   - Otro
4. Experiencia previa con xDD (se lo llamó xDD para hacer referencia a cualquier tipo de TDD). Opciones posibles:
   - Éste es mi primer proyecto con xDD.

- Ya he realizado otros proyectos con xDD.

5. ¿Cuánto fue el uso de xDD en el proyecto? Opciones posibles:
   - Casi todo, entre 95 y 100 % fue desarrollado con xDD.
   - Bastante, la mayoría fue desarrollado con xDD.
   - 50 %.
   - Poco o nada.

6.1.2. Resultados de la encuesta

En la tabla 6.1 se puede ver el resultado de la encuesta.

Proyecto	Lenguaje	Metodología	Experiencia	Uso xDD
Allelogram	Java	UTDD	Previa	La mayoría
Metropolia	Java	TDD	Ninguna	La mayoría
Gaphor	Python	TDD	Previa	La mayoría
Specrunner	Java	TDD	Previa	Poco o nada
ProcessPuzzle	Java, JS	TDD	Previa	95-100 %
Call Recording	Scala	TDD	Previa	La mayoría
FitLibrary	Java	ATDD	Previa	95-100 %
htmlparser	Java	TDD	Previa	50 %
NBehave	C#	BDD	Previa	95-100 %
GurkBurk	C#	BDD	Previa	95-100 %
crap4n	C#	BDD	Previa	95-100 %
Jumi	Java	ATDD	Previa	95-100 %
Dimdwarf	Java, Scala	ATDD	Previa	95-100 %

Cuadro 6.1: Resultado encuesta de proyectos

Además de enviar la encuesta a las distintas listas de correo de los proyectos, se consultó a diferentes personas muy reconocidas en el área. Entre ellas, las que contestaron mencionando proyectos están Robert Martin sugiriendo FitNesse y Kent Beck sugiriendo JUnit.

Algunos de los proyectos de la encuesta no tenían enlace a un repositorio público, con lo cual se eliminaron. Éste es el caso de Metropolia y Call Recording. Por otro lado era importante el alto uso de TDD durante todo el proyecto, con lo

cual se tomó sólo los casos donde el uso fue del 95 al 100 %. Por último, para evitar diferencias en los resultados debido a la implementación en distintos lenguajes, se tomaron sólo proyectos en Java. De esta manera, los posibles proyectos a analizar se reducen a Jumi y FitLibrary. En el caso de FitLibrary, es un proyecto bastante pequeño con pocos commits[13] y ningún release, con lo cual se decidió descartarlo.

Se decidió analizar 2 proyectos: Jumi y FitNesse. El otro posible proyecto a analizar es JUnit, pero no están separadas las pruebas unitarias de las de aceptación, con lo cual hace imposible el análisis.

6.2. Descripción del estudio

El estudio realizado sobre los proyectos comprende distintos análisis sobre la historia de los mismos a través del sistema de control de versiones.

En todos los casos se divide el proyecto en distintos períodos, donde cada período corresponde a la actividad entre un release y el siguiente.

Se estudiaron los siguientes aspectos en los proyectos analizados: archivos modificados entre releases, archivos modificados entre releases teniendo en cuenta sólamente modificaciones no triviales, actividad por commit y actividad simultánea por commit que serán explicados a continuación.

6.2.1. Archivos modificados entre releases

Como primer paso se obtiene la cantidad de archivos modificados correspondientes a pruebas de aceptación, pruebas unitarias y código entre releases. El objetivo es conocer como se distribuyen los cambios y si presenta algún patrón.

6.2.2. Modificaciones no triviales entre releases

En el caso anterior se asumió que un archivo fue modificado cuando sufrió cualquier tipo de modificación. A los fines de lo que se quiere analizar, hay ciertas modificaciones que no tienen sentido, como agregar, modificar o eliminar comentarios, espacios o cuestiones de indentación de código.

Es por eso que se repite el mismo análisis, pero teniendo en cuenta que un archivo fue modificado cuando los cambios realizados no son triviales.

6.2.3. Actividad por commit

En este caso, se toma cada *commit* como "unidad de cambio", es decir, todo lo que ocurre dentro de ese *commit* forma parte del mismo cambio. Si bien esto

[13] Se refiere a la idea de consignar un conjunto de cambios "tentativos, o no permanentes".

puede no ser del todo correcto, se lo considera una buena práctica y es común que se hagan de esa manera.

Anteriormente se contó la cantidad de archivos cambiados entre releases, sin tener en cuenta los commits.

Teniendo en cuenta que el foco ahora son los commits como unidad de cambio y no la cantidad de archivos, en este caso se contabilizaron la cantidad de commits donde hay cambios de código, pruebas unitarias o pruebas de aceptación.

6.2.4. Actividad simultánea

En la sección anterior se cambió el enfoque de análisis: en vez de contar cantidad de archivos cambiados, se contó cantidad de commits en donde los archivos cambiaron. Sin embargo todavía no se llega a ver si en un mismo commit cambia solamente un tipo de archivo o todos.

Es por eso que en este caso se calculó la actividad simultánea por commit. Dicho de otra manera, se contabilizó en cuántos commits cambiaron todas las combinaciones posibles de código, pruebas unitarias y pruebas de aceptación:

cod únicamente código.

ut únicamente pruebas unitarias.

at únicamente pruebas de aceptación.

ut_cod pruebas unitarias y código.

at_cod pruebas de aceptación y código.

at_ut pruebas de aceptación y pruebas unitarias.

at_ut_cod pruebas de aceptación, pruebas unitarias y código.

6.3. FitNesse

FitNesse es un framework para pruebas automatizadas de aceptación desarrollado en java. Consiste en una wiki[14] donde se pueden definir de forma rápida y sencilla las pruebas de aceptación, incluso para clientes sin formación técnica.

6.3.1. Archivos modificados entre releases

Se obtuvo la cantidad de archivos modificados correspondientes a pruebas de aceptación, pruebas unitarias y código entre releases.

En la figura 6.1 se puede ver que lo que más cambia son archivos de código, seguido por pruebas unitarias y por último pruebas de aceptación. Esto se repite en cada uno de los períodos.

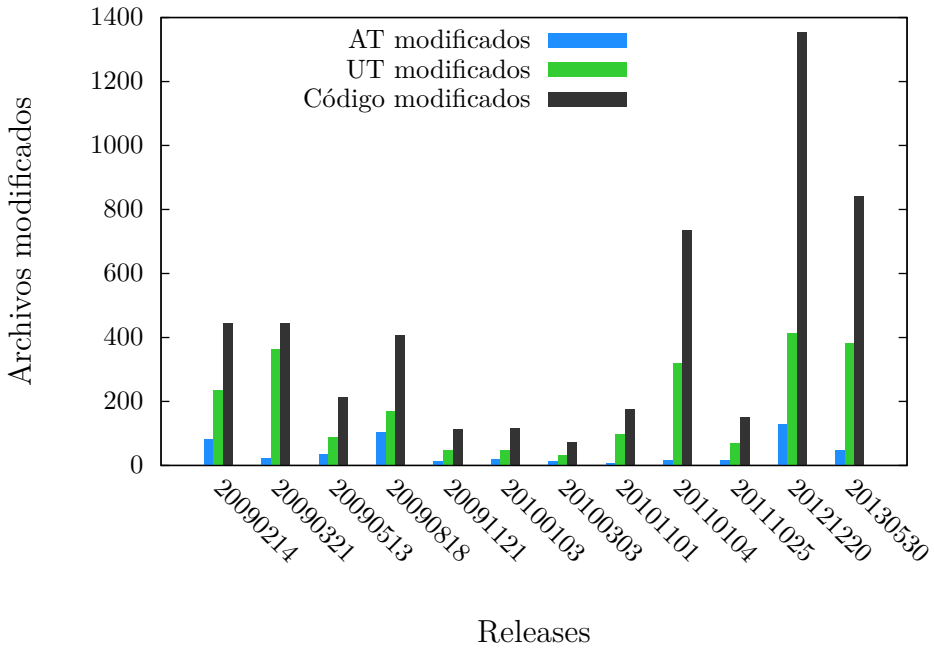

Figura 6.1: Cantidad de archivos modificados entre releases en FitNesse

6.3.2. Modificaciones no triviales entre releases

Se repitió el mismo análisis, pero teniendo en cuenta que un archivo fue modificado cuando los cambios realizados no son triviales.

[14]Sitio web cuyas páginas pueden ser creadas y editadas de forma muy sencilla y además permite la colaboración de múltiples usuarios.

En la figura 6.2 se puede ver el resultado de este análisis. Si bien los números no son los mismos, la forma del gráfico con respecto al anterior se mantiene. El único caso llamativo se da en el primer período. Esto se dio por un cambio de licencia a CPL 1.0. Esta modificación involucró cambiar únicamente un comentario indicando el tipo de licencia en más de 600 archivos.

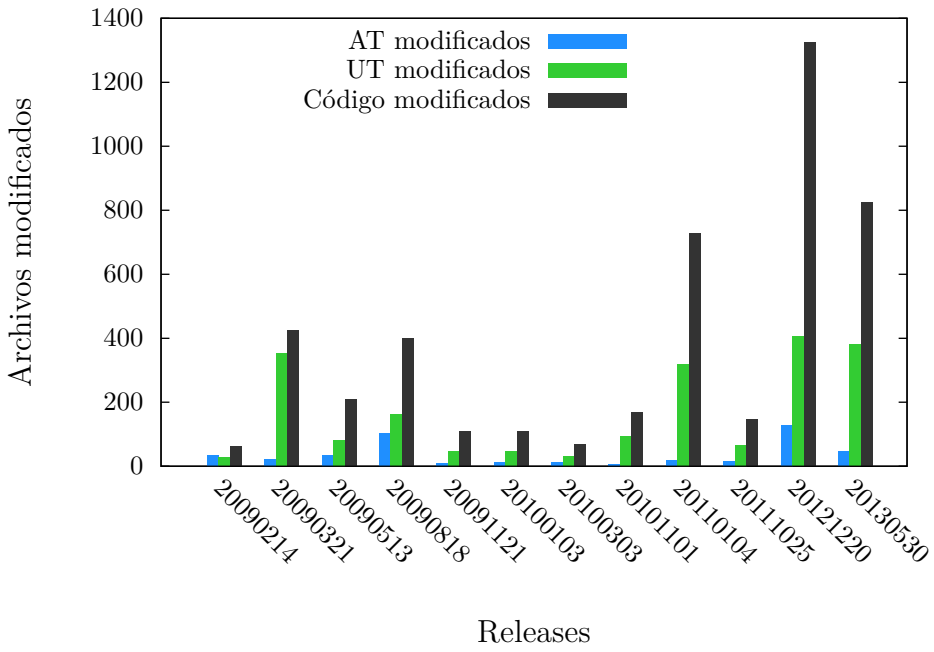

Figura 6.2: Cantidad de archivos con modificaciones no triviales en FitNesse

6.3.3. Actividad por commit

Se contabilizaron cantidad de commits donde hay cambios de código, pruebas unitarias o pruebas de aceptación.

El resultado se puede ver en el cuadro 6.2. La figura 6.3 muestra lo mismo que el cuadro pero expresado en porcentaje.

Como se puede ver, los resultados siguen siendo similares a los obtenidos anteriormente. La mayoría de los commits están dedicados a cambios en código (entre 50 % y 60 %), seguido por cambios en pruebas unitarias (entre 30 % y 45 %) y por último por pruebas de aceptación (menos del 15 %).

codigo	ut	at	Release
33	23	9	20090214
42	29	4	20090321
47	34	13	20090513
104	74	27	20090818
46	28	5	20091121
41	21	9	20100103
39	23	6	20100303
43	31	4	20101101
101	82	9	20110104
62	36	7	20111025
308	148	41	20121220
254	137	26	20130530
1120	666	160	Total

Cuadro 6.2: Actividad por commit en FitNesse

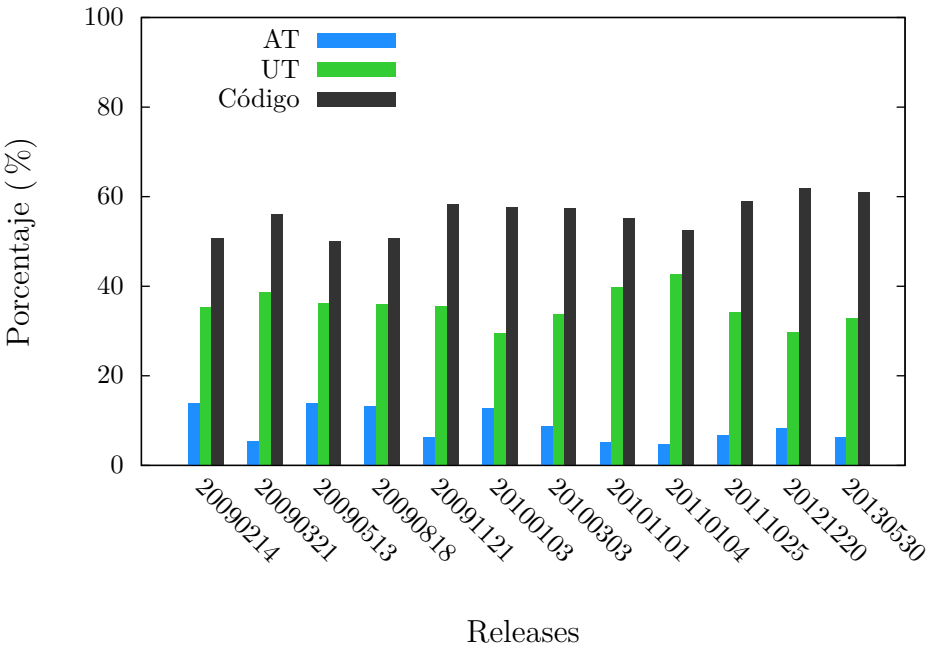

Figura 6.3: Actividad por commit en FitNesse

6.3.4. Actividad simultánea

Se calculó la actividad simultánea por commit. El resultado se puede ver en el cuadro 6.3. Al igual que en los otros casos, el cálculo se hizo tomando el período entre 2 releases consecutivos.

cod	ut	at	ut_cod	at_cod	at_ut	at_ut_cod	Release
10	2	4	18	2	0	3	20090214
16	4	0	22	1	0	3	20090321
14	3	2	22	2	0	9	20090513
32	9	4	49	7	0	16	20090818
17	0	0	24	1	0	4	20091121
17	1	4	19	4	0	1	20100103
18	5	1	16	3	0	2	20100303
17	6	2	24	1	0	1	20101101
20	2	3	75	1	0	5	20110104
27	2	1	29	1	0	5	20111025
171	23	16	112	12	0	13	20121220
143	28	11	97	3	1	11	20130530
502	85	48	507	38	1	73	Total

Cuadro 6.3: Actividad simultánea entre releases en FitNesse

Se puede observar que los cambios más frecuentes se dan en mayor medida en:

- código solamente.
- código y pruebas unitarias al mismo tiempo.
- código, pruebas unitarias y de aceptación al mismo tiempo.

La figura 6.4 muestra estos 3 casos expresado en porcentaje.

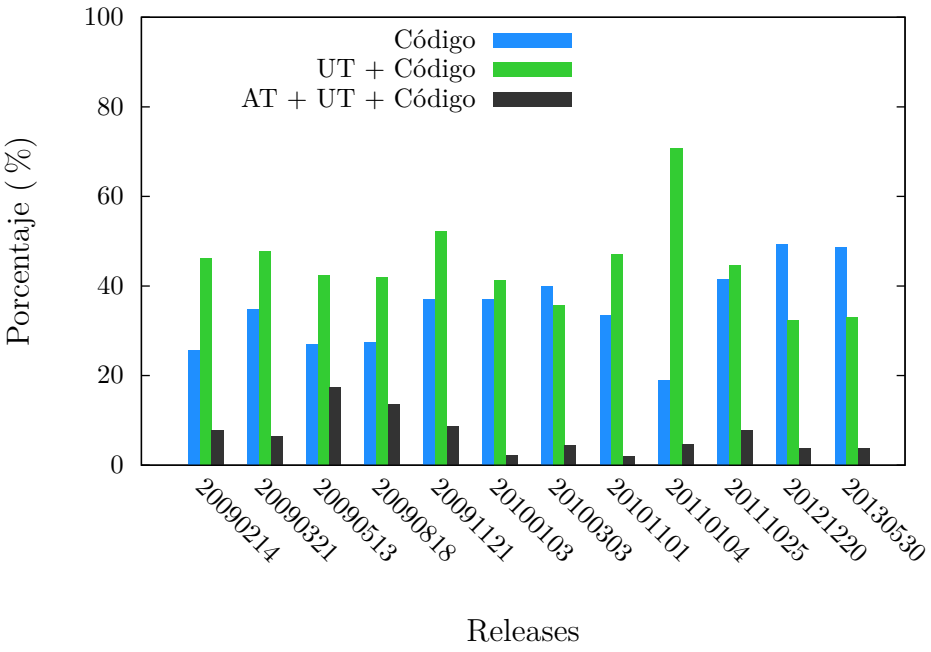

Figura 6.4: Actividad simultánea en FitNesse

6.4. Jumi

Jumi[15] es una herramienta que permite ejecutar pruebas de varios frameworks de pruebas automatizadas (en este momento soporta JUnit y derivados, sbt[16] y Specsy[17]). Provee ciertas facilidades como ejecución de pruebas en paralelo, priorización del orden de ejecución de las pruebas así como también optimizaciones en el tiempo de ejecución.

6.4.1. Archivos modificados entre releases

Se obtuvo la cantidad de archivos modificados correspondientes a pruebas de aceptación, pruebas unitarias y código entre releases.

En la figura 6.5 se puede ver que lo que más cambia son archivos de código, seguido por pruebas unitarias y por último pruebas de aceptación. Esto se repite en cada uno de los períodos y es un resultado análogo que se obtuvo en FitNesse (figura 6.1).

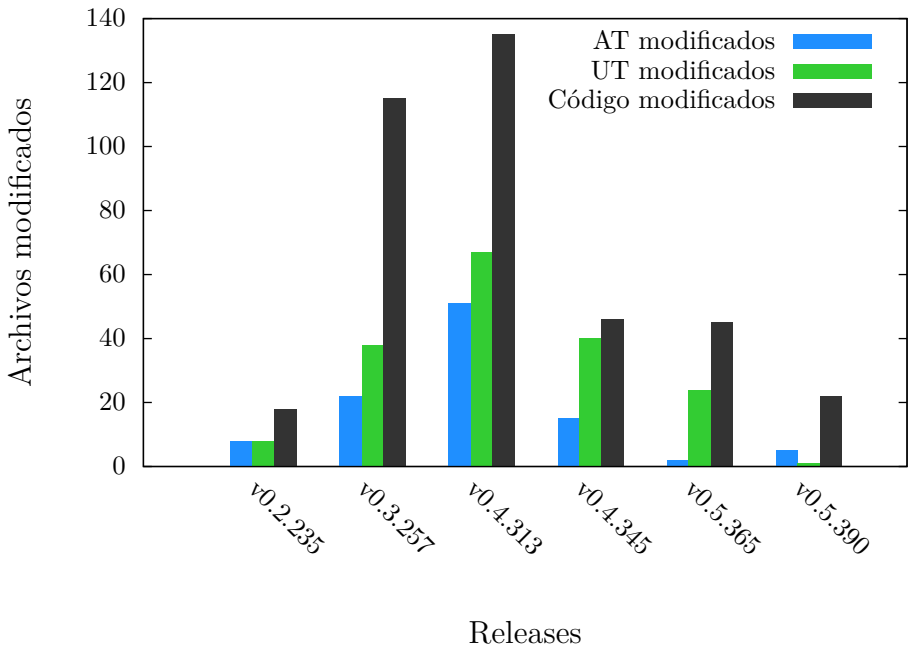

Figura 6.5: Cantidad de archivos modificados entre releases en Jumi

[15]Common test runner para la JVM (http://jumi.fi).

[16]Herramienta para desarrollo de Scala, Java y otros (http://www.scala-sbt.org).

[17]Framework para pruebas automatizadas en Scala, Groovy, Java y otros (http://specsy.org).

6.4.2. Modificaciones no triviales entre releases

Se repitió el mismo análisis, pero teniendo en cuenta que un archivo fue modificado cuando los cambios realizados no son triviales.

En la figura 6.6 se puede ver el resultado de este análisis. Al igual que con FitNesse, los números son ligeramente menores pero la forma del gráfico se mantiene.

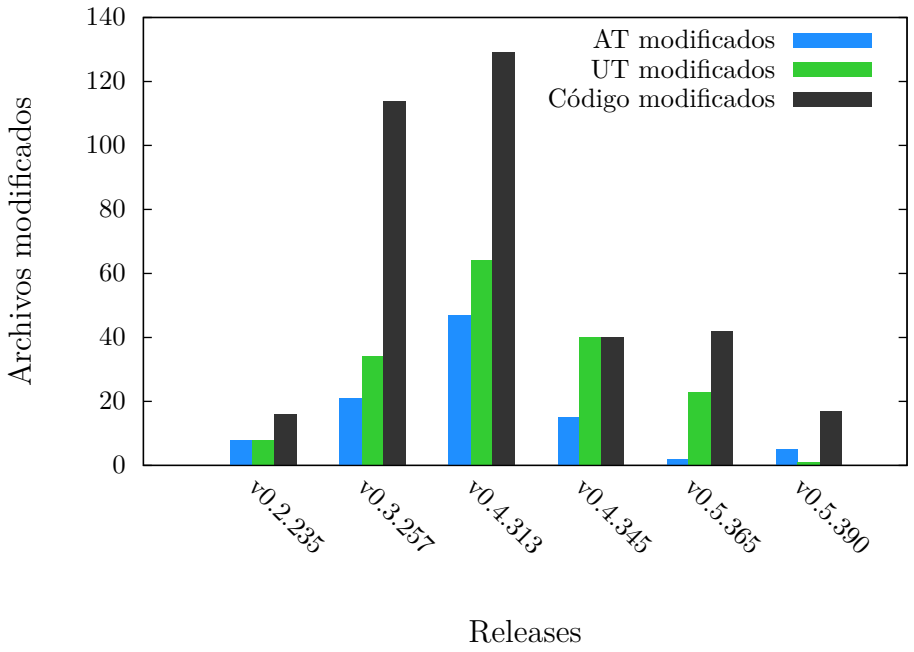

Figura 6.6: Cantidad de archivos con modificaciones no triviales en Jumi

6.4.3. Actividad por commit

Se contabilizó cantidad de commits donde hay cambios de código, pruebas unitarias o pruebas de aceptación.

El resultado se puede ver en el cuadro 6.4. La figura 6.7 muestra lo mismo que el cuadro pero expresado en porcentaje.

A diferencia de lo obtenido anteriormente, los resultados varían un poco según el período en el que se analice. En líneas generales se cumple lo mismo que en FitNesse: en la mayoría de los commits hay cambios en archivos de código (entre 50 % y 60 %), seguido por cambios en pruebas unitarias (entre 30 % y 45 %) y por último por pruebas de aceptación (menos del 20 %).

Hay que tener en cuenta que el primer y último período no son los más representativos ya que cuentan con muy pocos commits.

codigo	ut	at	Release
11	5	6	v0.2.235
28	22	13	v0.3.257
70	55	32	v0.4.313
23	24	4	v0.4.345
20	16	2	v0.5.365
9	1	3	v0.5.390
161	123	60	Total

Cuadro 6.4: Actividad por commit en Jumi

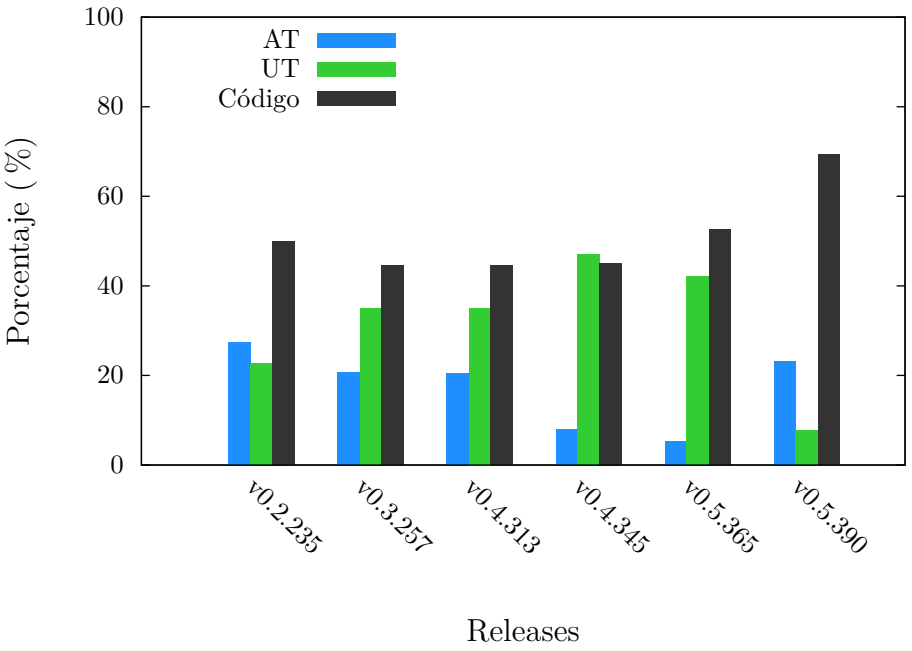

Figura 6.7: Actividad por commit en Jumi

6.4.4. Actividad simultánea

Se calculó la actividad simultánea por commit. El resultado se puede ver en el cuadro 6.5.

cod	ut	at	ut_cod	at_cod	at_ut	at_ut_cod	Release
5	0	4	4	1	0	1	v0.2.235
4	2	3	14	4	0	6	v0.3.257
18	8	11	32	6	1	14	v0.4.313
4	5	0	16	1	1	2	v0.4.345
7	3	1	12	0	0	1	v0.5.365
7	0	2	1	1	0	0	v0.5.390
45	18	21	79	13	2	24	Total

Cuadro 6.5: Actividad simultánea entre tags en Jumi

Se puede observar que los cambios más frecuentes se dan en mayor medida en:

- código y pruebas unitarias al mismo tiempo.
- código solamente.
- código, pruebas unitarias y de aceptación al mismo tiempo.

La figura 6.8 muestra estos 3 casos expresado en porcentaje.

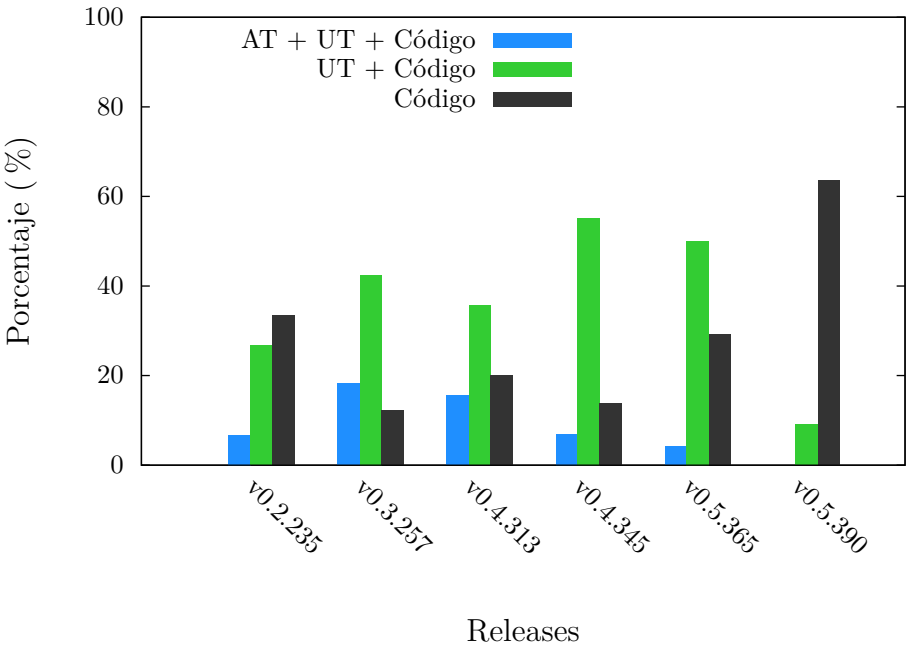

Figura 6.8: Actividad simultánea en Jumi

7. Trabajos relacionados

TDD ha ido ganando popularidad con el correr de los años y varios estudios intentaron evaluar la efectividad con respecto a otras formas de desarrollo. Si bien esta tesis está orientada al impacto de cambios en sistemas desarrollados con UTDD o ATDD, los siguientes estudios han aportado conocimiento e ideas sobre el tema.

7.1. TDD versus non-TDD

Existen varios trabajos que intentan mostrar si es mejor el desarrollo con TDD que sin TDD. Hay varios enfoques distintos para realizar esta comparación. A continuación se muestran algunos de estos trabajos.

7.1.1. TDD en la industria

Müller y Hagner hicieron foco en las ventajas de realizar pruebas al principio comparado con realizarlas al final, abstrayéndose de los principios de TDD o XP (Müller y Hagner, 2002). Los resultados mostraron poca diferencia entre ambos grupos salvo en la reusabilidad de ciertos componentes, que fue mejor en los que realizaban las pruebas al principio.

Boby George y Laurie Williams toman esta investigación y hacen algunas críticas relacionadas con el tamaño de la muestra y la experiencia de la gente involucrada en las pruebas. Hicieron una investigación con proyectos desarrollados con TDD versus proyectos desarrollados con una metodología como cascada, utilizando programadores profesionales en industrias (George y Williams, 2003). El objetivo de ellos fue probar 2 hipótesis:

1. Si se desarrolla siguiendo los principios de TDD se obtiene un sistema con calidad externa superior comparado con el desarrollo tipo cascada.

2. El desarrollo con TDD es más rápido.

El primer punto fue evaluado con pruebas de tipo de caja negra. El grupo que desarrolló con TDD mostró mejores resultados en cuanto a calidad externa. En cuanto al segundo punto, los grupos que desarrollaron con TDD tardaron más tiempo para desarrollar la aplicación. Sin embargo, muchas pruebas de los grupos que las realizaban al final eran de menor calidad. Este punto es retomado en otro estudio donde muestran que en el largo plazo, la alta calidad del código puede significar un menor tiempo de desarrollo (Müller y Padberg, 2003).

Otro caso de estudio de TDD en la industria es el de Laurie Williams en IBM (Williams et al., 2003). En este caso tomaron un proyecto existente que iba a ser migrado a otra plataforma. Los profesionales involucrados en el estudio, si bien todos tenían una formación en ciencias de la computación, ninguno había trabajado con TDD antes. Los resultados encontrados en este trabajo fueron que las pruebas escritas al desarrollar con TDD eran capaces de encontrar más errores que las pruebas escritas anteriormente; dicho de otra manera, encontraron que las pruebas escritas con TDD eran mejores. Por otro lado, la cantidad de defectos encontrados por línea de código se redujeron en un 40 %. Esto indica que con TDD se introdujeron menos errores.

Otro estudio conocido es el de Microsoft llevado a cabo por Bhat y Nagappan en donde tomaron como casos de estudio una parte de Windows y una parte de MSN y seleccionaron proyectos similares para poder comparar (Bhat y Nagappan, 2006). En el primer caso (Windows), encontraron que el proyecto sin TDD tenía 2.6 más defectos cada 1000 líneas de código que el proyecto desarrollado con TDD. Por otro lado, el tiempo total para el desarrollo fue entre 25 y 30 % más en el caso del proyecto desarrollado con TDD. En el segundo caso (MSN) los resultados son similares: el proyecto desarrollado sin TDD contenía 4.5 más defectos pero el desarrollo con TDD tomó 15 % más de tiempo.

7.1.2. TDD usando métricas internas

En los trabajos citados anteriormente, el análisis estaba enfocado en la calidad externa, utilizando principalmente la cantidad de errores como punto de comparación entre un proyecto y otro. Sin embargo hay otros estudios que cambian el foco del análisis en la calidad interna. Estos estudios sostienen que los proyectos con código de alta calidad tienen como consecuencia pocos errores, y es por eso que se concentran en la causa y no en el efecto.

La tesis de doctorado de Janzen evalúa proyectos utilizando métricas internas en proyectos académicos y en la industria (Janzen, 2006). Un punto interesante de las conclusiones es la diferencia encontrada entre desarrolladores con experiencia (en el caso de los proyectos en la industria) y desarrolladores inexpertos (proyectos académicos). En el caso de los desarrolladores con experiencia, el resultado de desarrollar con TDD fue mucho mejor en términos de complejidad de código,

mientras que en el grupo de inexpertos el resultado fue el opuesto.

El trabajo de Rod Hilton busca mejorar algunos aspectos de la tesis de Janzen y también compara sistemas desarrollados con TDD contra sistemas desarrollados sin TDD (Hilton, 2009). Él propone que la calidad interna del código influye mucho en la calidad externa. De hecho afirma que si un sistema carece de calidad interna, a la larga está condenada al fracaso ya que llega un momento que es imposible de manejar.

Hilton tomó múltiples proyectos de código abierto desarrollados con y sin TDD. Luego aplicó distintas métricas que apuntan a medir:

Cohesión Mide cuán relacionadas están las responsabilidades de un módulo. Las métricas que utilizó para medir la cohesión fueron:

- Lack of Cohesion Methods (LCOM)
- Specialization Index (SI)
- Número de parámetros en métodos.
- Número de métodos estáticos

Acoplamiento Indica en qué grado los módulos dependen de otros. Las métricas que utilizó para medir el acoplamiento fueron:

- Afferent Coupling (Ca) y Efferent Coupling (Ce)
- Distancia de la secuencia principal.
- Depth of Inheritance Tree (DIT)

Complejidad La definición de complejidad es variada. El autor tomó la definición de Schach: "la cantidad de esfuerzo que se necesita para entender y modificar el código de forma correcta" (Schach, 2010). Las métricas que utilizó para medir complejidad fueron:

- Tamaño de métodos y clases.
- Complejidad ciclomática.
- Nested Block Depth (NBD)

. Los resultados que obtuvo fueron que los proyectos desarrollados con TDD presentaron mejores resultados en las 3 áreas evaluadas. En el caso de la cohesión, notó una mejora entre el 20 y 25 %; en el caso del acoplamiento la mejora fue del 5 al 10 % y en el caso de la complejidad la mejora fue del 30 %.

7.2. Cobertura entre pruebas a distintos niveles para refactorizaciones más seguras

El trabajo de Carlos Fontela en su tesis de maestría es el más relacionado y el que inspiró esta tesis (Fontela, 2013). El objetivo de Fontela es proponer una práctica metodológica que permita realizar refactorizaciones seguras usando como garantía pruebas de distintos niveles.

La práctica que él propone es la siguiente:

- 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.

El primer paso es el más común, de manual. Se realiza una refactorización con el soporte de las pruebas unitarias. Al no modificarse las pruebas unitarias, es fácil determinar si la refactorización se hizo de forma correcta o no. Está claro que para que esto ocurra se debe contar con buenas pruebas unitarias.

Sin embargo, hay casos en donde las refactorizaciones involucran también un cambio en las pruebas unitarias. Algunos ejemplos podrían ser el cambio de nombre de un método o una clase o el cambio de firma de un método (cantidad o tipo de parámetros). En el caso en que se tenga que cambiar una prueba unitaria se pierde la seguridad que éstas mismas brindaban. De la misma forma que no es conveniente refactorizar código sin una prueba que asegure que ese cambio no alteró el comportamiento del sistema, lo mismo ocurre con las pruebas unitarias. En este caso él propone utilizar las pruebas de integración. Estas pruebas son de mayor granularidad y para poder ser utilizadas deben cubrir por lo menos la porción de código que se quiere refactorizar.

Por último, si en el segundo paso ocurre el mismo problema con las pruebas que en el primer paso, se recurre a las pruebas de comportamiento. Él toma como hipótesis que las pruebas de comportamiento son los invariantes, ya que si éstas deben cambiar, ya no se está en presencia de una refactorización sino en un cambio de requerimientos.

Metodología de desarrollo Como metodología de desarrollo, el autor propone, a modo de recomendación, el uso de ATDD. La elección de ATDD en este contexto

es inteligente, ya que provee de forma natural el marco de pruebas de distinta granularidad que la propuesta necesita.

Al seguir esta metodología de desarrollo, se parte de pruebas de aceptación (o pruebas de comportamiento) y luego se deriva en pruebas de menor granularidad como ser pruebas de integración y pruebas unitarias. Dado que las pruebas nacen a partir de las necesidades de las US, y el código nace para satisfacer estas pruebas, se da de forma natural la cobertura de las pruebas en los distintos niveles de granularidad.

Caso de estudio Esta propuesta metodológica fue puesta a prueba con un caso de estudio. Al seguir esta prueba se puede ver en detalle como funciona cada paso. El primer caso son refactorizaciones que no alteran ningún tipo de pruebas (cambio de nombre en variable local y cambio de algoritmo interno). Estas refactorizaciones quedan cubiertas por las pruebas unitarias y se dan sin inconvenientes. Luego, se da el caso de una refactorización que involucra el cambio de firma de un método que rompe algunas pruebas unitarias, pero no las de integración ni las de comportamiento. En esta oportunidad, dado que se cuenta con pruebas de integración que cubren la porción de código a modificar, se puede realizar la refactorización de forma segura. Por último se da el caso de una refactorización bastante mayor que rompe pruebas unitarias y de integración. En esta situación las pruebas de comportamiento son las que proporcionan una red de contención y permiten hacer las refactorizaciones en el código y luego en las pruebas que fallaban.

8. Conclusiones

A partir de este trabajo se puede concluir que el uso de ATDD es más seguro que UTDD a la hora de realizar cambios debido a los distintos niveles de granularidad de sus pruebas.

Una de las hipótesis tomadas fue que las pruebas de aceptación, debido a su naturaleza, son las más estables, por lo tanto, las menos cambiantes. En los resultados obtenidos en el estudio se puede ver que tanto en cantidad de archivos modificados como en cantidad de commits con modificaciones, los cambios en pruebas de aceptación son raros. También se ha visto que existe la posibilidad de que sea necesario cambiar todos los tipos de pruebas. En este caso, si bien ya no se cuenta con la seguridad de las pruebas, hay que recordar que este tipo de cambios es debido a cambios de requerimientos, los cuales son debidamente analizados y acordados con el cliente.

Por otro lado, en el caso de las pruebas unitarias, la frecuencia de cambio es mucho más alta de lo esperada. Pipka menciona que al realizar refactorizaciones usando TDD, ocurre una especie de paradoja: en muchos casos, la refactorización de código también afecta a las pruebas unitarias (Pipka, 2002). En este caso no se puede verificar si la refactorización del código fue correcta o no. Fontela demuestra en su tesis que los cambios en código y en pruebas unitarias son bastante comunes, haciendo necesarios otros tipos de pruebas para poder realizar refactorizaciones en forma segura (Fontela, 2013).

Por último, se vio qué tipos de cambios son más frecuentes en cada commit. En los resultados obtenidos se puede ver que el código y las pruebas unitarias están muy ligados. Esto confirma el punto citado anteriormente. Es por eso que resulta necesario contar con otros niveles de pruebas para poder asegurar un nivel de seguridad en los cambios.

Como ya se ha visto, el hecho de que una metodología de desarrollo sea segura ante cambios es un hecho sumamente importante. Según diferentes autores, el mantenimiento en el software representa un volumen cada vez mayor. Si bien los números varían entre los autores, todos coinciden en que el mantenimiento representa al menos un 50 % del costo en la vida del sistema (ver sección 4.2). En este contexto, es necesario estar preparado ante el cambio.

ATDD como metodología de desarrollo. Siendo que el mantenimiento representa una parte tan importante en la vida de un sistema de software, es importante contar con una metodología de desarrollo que pueda brindar una base sólida y permita realizar cambios en forma segura. Rod Hilton muestra en su tesis los efectos de TDD en la calidad interna del código (Hilton, 2009). Esto proporciona una base sólida, con software de alta calidad, altamente cohesivo y de bajo acoplamiento.

Por otro lado, ATDD permite realizar cambios en forma segura ya que cuenta con pruebas de distintos tipos de granularidad, que pueden ser aprovechadas para seguir la propuesta metodológica planteada por Fontela para realizar refactorizaciones (Fontela, 2013).

Por último, Müller plantea los distintos escenarios donde TDD puede implicar un menor costo en el total del proyecto debido a la alta calidad del código y la reducción de defectos (Müller y Padberg, 2003).

En resumen, al utilizar ATDD se consigue:

- Código más prolijo, cohesivo y desacoplado, fácil de entender y mantener.
- Realizar cambios en forma segura.
- Reducir el costo total del proyecto al reducir el costo del mantenimiento.

8.1. Trabajos futuros

Una posible mejora al estudio realizado en esta tesis es ampliar la granularidad de los cambios a nivel métodos de prueba en vez de archivos. De esta manera se podría contabilizar con mayor exactitud cuántas pruebas fueron agregadas, modificadas o eliminadas en cada paso. Del mismo modo, se podrían categorizar las pruebas, por ejemplo en niveles como ser “trivial”, “normal” y “complejo”. En este trabajo se dejaron de lado las modificaciones triviales (cambios en comentarios y cuestiones relacionadas con espacios, como por ejemplo cambios en la indentación del código).

También se podrían mejorar los casos de estudio analizando más proyectos. Sería interesante comparar los resultados obtenidos en este trabajo, en el que fueron analizados únicamente proyectos de código abierto, con los obtenidos al analizar proyectos en la industria. Janzen, por ejemplo, encontró una gran diferencia entre proyectos en la industria y proyectos académicos, atribuyendo esta diferencia a la experiencia previa de ambos grupos (Janzen, 2006).

Un enfoque bastante distinto, pero no menos interesante, es el análisis del impacto en los recursos humanos. Siempre que se mencionan las ventajas de TDD se habla de la motivación que adquiere el equipo de desarrollo y la participación de un cliente más involucrado. Estos dos puntos son muy importantes en un proyecto.

Ivancevich menciona que los humanos tienen distintos tipos de necesidades que van más allá de lo monetario, como necesidades sociales, satisfacción personal, entre otros (Ivancevich et al., 2008).

Bibliografía

Astels, D. (2005). A new look at Test-Driven Development. http://techblog.daveastels.com/files/BDD_Intro.pdf. 28

Atkinson, S. y Benefield, G. (2013). Software development: Why the traditional contract model is not fit for purpose. En *Proceedings of the 2013 46th Hawaii International Conference on System Sciences*, HICSS '13, páginas 4842–4851. IEEE Computer Society. 4

Beck, K. (2002). *Test-Driven Development by Example*. Addison-Wesley, Boston, MA, USA. 43

Beck, K. y Andres, C. (2004). *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 2da edición. 7, 41, 43

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mallor, S., Shwaber, K., y Sutherland, J. (2001). The Agile Manifesto. Technical report, The Agile Alliance. 4

Beck, K. y Gamma, E. (2000). Test-infected: Programmers love writing tests. En *More Java Gems*, páginas 357–376. Cambridge University Press. 43

Bhat, T. y Nagappan, N. (2006). Evaluating the efficacy of test-driven development: Industrial case studies. En *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, ISESE '06, páginas 356–363, New York, NY, USA. ACM. 68

Bloch, J. (2006). Extra, extra - read all about it: Nearly all binary searches and mergesorts are broken. http://googleresearch.blogspot.com.ar/2006/06/extra-extra-read-all-about-it-nearly.html. 46

Boehm, B. W. (1981). *Software Engineering Economics*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1ra edición. 40

Brooks, F. P. (1975). *The mythical man-month: essays on software engineering*. Addison-Wesley Pub. Co. 46

Burke, E. M. y Coyner, B. M. (2003). *Java Extreme Programming Cookbook*. O'Reilly Media. 19

Cohn, M. (2004). *User Stories Applied: For Agile Software Development*. Addison-Wesley Professional, Redwood City, CA, USA, 1ra edición. 23

Cook, J. (2012). How google motivates their employees with rewards and perks. http://thinkingleader.hubpages.com/hub/How-Google-Motivates-their-Employees-with-Rewards-and-Perks. 41

Cornett, S. (1996). Code coverage analysis. http://www.bullseye.com/coverage.html. 43

Crispin, L. (2005). Using customer tests to drive development. *Methods & Tools*, 13(2). 30

Erdogmus, H., Morisio, M., y Torchiano, M. (2005). On the effectiveness of the test-first approach to programming. *IEEE Transactions on Software Engineering*, 31(3):226–237. 44

Evans, E. (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 1ra edición. 21

Feathers, M. (2004). *Working effectively with legacy code*. Prentice Hall PTR, Upper Saddle River, NJ, USA. 17

Fontela, M. C. (2013). Cobertura entre pruebas a distintos niveles para refactorizaciones más seguras. Tesis de maestría, Universidad Nacional de La Plata. 39, 46, 70, 72, 73

Fontela, M. C., Garrido, A., y Lange, A. (2013). Hacia un enfoque metodológico de cobertura múltiple para refactorizaciones más seguras. En *Simposio Argentino de Ingeniería de Software ASSE, 42 JAIIO*. 28

Fowler, M. (1999). *Refactoring : improving the design of existing code*. Addison-Wesley Professional, Boston, MA, USA. 13, 46

George, B. y Williams, L. (2003). An initial investigation of test driven development in industry. En *Proceedings of the 2003 ACM symposium on Applied computing*, SAC '03, páginas 1135 – 1139, New York, NY, USA. ACM. 67

Guimaraes, T. (1983). Managing application program maintenance expenditures. *Communications of the ACM*, 26(10):739–746. 34

Hayes, B. C. (2000). The interface hall of shame. http://hallofshame.gp.co.at/. 17

Hilton, R. (2009). Quantitatively evaluating test-driven development by applying object-oriented quality metrics to open source projects. Tesis de maestría, Regis University. 46, 69, 73

ISO/IEC/IEEE 24765:2010. Systems and software engineering – vocabulary. 32

Ivancevich, J. M., Konopaske, R., y Matteson, M. T. (2008). *Organizational behavior and management*. McGraw-Hill/Irwin, Boston, 8va edición. 41, 74

Janzen, D. S. (2006). *An Empirical Evaluation of the Impact of Test-driven Development on Software Quality*. Tesis de doctorado, University of Kansas, Lawrence, KS, USA. 68, 73

Kappelman, L. A. (2000). Some strategic Y2K blessings. *IEEE Software*, 17(2):42–46. 35

Karch, E. (2011). Lehman's laws of software evolution and the staged-model. http://blogs.msdn.com/b/karchworld_identity/archive/2011/04/01/lehman-s-laws-of-software-evolution-and-the-staged-model.aspx. 34

Knuth, D. E. (1997). *The Art of Computer Programming*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 3ra edición. 45

Koskela, L. (2007). *Test driven: practical TDD and acceptance TDD for java developers*, capítulo 9. Manning Publications Co., Greenwich, CT, USA. 22, 27

Lehman, M. M. (1980). Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076. 33

Lehman, M. M., Ramil, J. F., Wernick, P. D., Perry, D. E., y Turski, W. M. (1997). Metrics and laws of software evolution - the nineties view. En *Proceedings of the 4th International Symposium on Software Metrics*, METRICS '97, páginas 20–32, Washington, DC, USA. IEEE Computer Society. 33

Martin, R. (2005a). The bastard child. http://www.butunclebob.com/ArticleS.UncleBob. 12

Martin, R. (2005b). Brain surgery guides. http://butunclebob.com/ArticleS.UncleBob.BrainSurgeryGuides. 38

Martin, R. (2005c). Incremental architecture. http://www.butunclebob.com/ArticleS.UncleBob.IncrementalArchitecture. 19, 47

Martin, R. (2005d). Just 10 minutes without a test. http://www.butunclebob.com/ArticleS.UncleBob.JustTenMinutesWithoutAtest. 12

Martin, R. (2005e). The sensitivity problem. http://www.butunclebob.com/ArticleS.UncleBob.TheSensitivityProblem. 13

Martin, R. C. y Martin, M. (2006). *Agile Principles, Patterns, and Practices in C#*. Prentice Hall PTR, Upper Saddle River, NJ, USA. 26

McConnell, S. (1996). *Rapid Development: Taming Wild Software Schedules*. Microsoft Press, Redmond, WA, USA, 1ra edición. 15

McCracken, D. D. y Jackson, M. A. (1982). Life cycle concept considered harmful. *SIGSOFT Software Engineering Notes*, 7(2):29–32. 4

Meszaros, G. (2007). *XUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, Upper Saddle River, NJ. 10

Mugridge, R. (2008). Managing agile project requirements with storytest-driven development. *IEEE Software*, 25(1):68–75. 30

Müller, M. M. y Hagner, O. (2002). Experiment about test-first programming. *IEE Proceedings Software*, 149(5):131–136. 67

Müller, M. M. y Padberg, F. (2003). About the return on investment of Test-Driven Development. En *In International Workshop on Economics-Driven Software Engineering Research EDSER-5*. 36, 68, 73

North, D. (2006). Introducing BDD. http://dannorth.net/introducing-bdd/. 28

Pipka, J. U. (2002). Refactoring in a "test first"-world. En *Proc. Int'l Conf. eXtreme Programming and Flexible Processes in Software Engineering (XP)*. 72

Schach, S. R. (2010). *Object-Oriented and Classical Software Engineering*. McGraw-Hill, Inc., New York, NY, USA, 8va edición. 34, 35, 69

Sommerville, I. (2000). *Software engineering*, capítulo 27. Addison-Wesley Longman Publishing Co., Inc., 6ta edición. 35, 37

Warren, I. (1999). *The Renaissance of Legacy Systems: Method Support for Software-System Evolution*. Springer. 32

Wells, D. (2009). Extreme programming: a gentle introduction. http://www.extremeprogramming.org. 8

Wheeler, D. A. (2011). The most important software innovations. http://www.dwheeler.com/innovation/innovation.html. 10

Williams, L., Maximilien, E. M., y Vouk, M. (2003). Test-driven development as a defect-reduction practice. En *Proceedings of the 14th International Symposium on Software Reliability Engineering*, ISSRE '03, Washington, DC, USA. IEEE Computer Society. 68

Glosario

commit Se refiere a la idea de consignar un conjunto de cambios "tentativos, o no permanentes". 55, 56, 58, 60, 63, 65, 72, 84, 86, 87

FitNesse Framework para pruebas automatizadas de aceptación (http://fitnesse.org). 19, 22, 25, 47, 48, 54, 55, 57, 62, 63, 83

getters y setters Métodos para leer y modificar atributos de una clase. 12

goldplating Cuando se siguen desarrollando mejoras de algo que ya cumple con los requerimientos pensando que va a ser mejor para el cliente aún cuando éste nunca lo pidió. Muchas veces esto provoca el efecto contrario, además de haber perdido tiempo. 13, 16, 27

JBehave Framework en Java para pruebas automatizadas para BDD (http://jbehave.org/). 28, 29

Jumi Common test runner para la JVM (http://jumi.fi). 62

JUnit Framework en Java para pruebas automatizadas (http://junit.org). 28, 54, 55, 62

reflexión Capacidad de un programa de revisar y modificar la estructura o comportamiento de un objeto en tiempo de ejecución. 47

Ruby Lenguaje de programación orientado a objetos interpretado. 48

sbt Herramienta para desarrollo de Scala, Java y otros (http://www.scala-sbt.org). 62

sistema empotrado Sistema diseñado para realizar unas pocas funciones específicas, frecuentemente en un sistema de tiempo real con hardware dedicado. 34

Specsy Framework para pruebas automatizadas en Scala, Groovy, Java y otros (http://specsy.org). 62

tag Etiqueta que identifica el estado de un proyecto bajo un sistema de control de versiones en un momento determinado, normalmente utilizado para indicar versiones. 84, 86

wiki Sitio web cuyas páginas pueden ser creadas y editadas de forma muy sencilla y además permite la colaboración de múltiples usuarios. 57

Y2K Del inglés "Year 2000", fue un caso muy conocido de un error en los sistemas de software que utilizaban 2 dígitos para representar el año en una fecha. La consecuencia de esto era que el día siguiente al 31 de dicimebre de 1999 seria el primero de enero de 1900. 35

Siglas

ATDD Acceptance-Test Driven Development 22, 24, a, 27, 31, 38–40, 49–53, 67, 70, 72, 73

BDD Behaviour Driven Development 28, 31

Ca Afferent Coupling 69
Ce Efferent Coupling 69
CTDD Customer-Test Driven Development 30

DDD Domain Driven Design 21
DIT Depth of Inheritance Tree 69

LCOM Lack of Cohesion Methods 69

NBD Nested Block Depth 69

SBP Sistema Bajo Prueba 10
SI Specialization Index 69
STDD Story-Test Driven Development 30, 31

TDD Test Driven Development 9–12, 14–17, 19, 21, 22, a, 28, 36–38, 43, 44, 46, 47, 51, 53, 54, 67–69, 72, 73

US User Stories 6, 7, 22–27, 29, 30, 50, 71
UTDD Unit-Test Driven Development 20, 22, 23, 25, a, 27, 30, 31, 39, 40, 42, 44, 45, 50–52, 67, 72

XP Extreme Programming 7–9, 15, 19, a, 34, 41, 67

A. Datos técnicos de los proyectos analizados

A.1. FitNesse

A.1.1. Datos generales

A continuación se muestran algunos datos de referencia del proyecto. En los casos de archivos de código, pruebas unitarias y de aceptación se tomó como referencia el último release (20130530). El repositorio del proyecto se encuentra en https://github.com/unclebob/fitnesse/.

Código En la sección A.1.2 se especifica qué archivos son considerados de código.

- 600 archivos.
- 544 clases, 57 interfaces y 4 con definiciones (enumerados). Hay algunos casos donde se define más de una clase por archivo (por ese motivo la suma no da 600).

Pruebas unitarias En la sección A.1.2 se especifica qué archivos son considerados pruebas unitarias.

- 301 archivos.
- 2200 pruebas.

Pruebas de aceptación En la sección A.1.2 se especifica qué archivos son considerados pruebas de aceptación.

- 401 archivos.
- 53 clases de Java (conexión FitNesse con sistema) y 348 archivos de texto (FitNesse).

Releases Para identificar los releases se analizaron los tags[18] en el sistema de control de versiones. De todos los tags, algunos se tomaron como releases y otros se ignoraron. Estos son los releases que se tomaron:

- 20090112
- 20090214
- 20090321
- 20090513
- 20090818
- 20091121
- 20100103
- 20100303
- 20101101
- 20110104
- 20111025
- 20121220
- 20130530

Los siguientes tags se ignoraron como releases porque no representan un release:

- list
- nonewtmpl

Tiempo entre releases A continuación se detallan el tiempo promedio, mínimo y máximo entre un release y el siguiente:

- Tiempo promedio: 172 días.
- Tiempo mínimo: 32 días.
- Tiempo máximo: 480 días.

Cantidad de commits entre releases A continuación se detallan la cantidad de commits promedio, mínimo y máximo entre un release y el siguiente:

- Cantidad de commits promedio: 190.
- Cantidad de commits mínimo: 48.
- Cantidad de commits máximo: 783. Salvo éste y otro caso, el resto están mucho más cerca de la media.

[18] Etiqueta que identifica el estado de un proyecto bajo un sistema de control de versiones en un momento determinado, normalmente utilizado para indicar versiones.

A.1.2. Identificación de código, pruebas unitarias y pruebas de aceptación

Para identificar qué archivos corresponden a código, pruebas unitarias o pruebas de aceptación se utilizó el siguiente criterio:

Código Todos los archivos java dentro de los directorios (o cualquier subdirectorio de estos):

- `src/fit/`
- `src/fitnesse/`
- `src/fitnesseMain/`
- `src/util/`

exceptuando aquellos archivos que terminen en `Test.java` y los que se encuentren dentro de una carpeta `fixtures`.

Pruebas unitarias Todos los archivos que terminen en `Test.java` dentro del directorio (o cualquier subdirectorio) `src/` exceptuando los siguientes archivos (detallados en el archivo `build.xml` que indica cuales archivos se deben excluir):

- `ShutdownResponderTest.java`
- `QueryTableBaseTest.java`
- `Test.java`
- `SystemUnderTest.java`
- `MySystemUnderTest.java`

Pruebas de aceptación Todos los archivos `content.txt` que se encuentren dentro del directorio (o cualquier subdirectorio) `FitNesseRoot/FitNesse/SuiteAcceptanceTests/` o aquellos archivos java dentro del directorio (o cualquier subdirectorio) `fixtures/`.

A.2. Jumi

A.2.1. Datos generales

A continuación se muestran algunos datos de referencia del proyecto. En los casos de archivos de código, pruebas unitarias y de aceptación se tomó como referencia el último release (v0.5.390). El repositorio del proyecto se encuentra en https://github.com/orfjackal/jumi.

Código En la sección A.2.2 se especifica qué archivos son considerados de código.

- 128 archivos.
- 102 clases, 26 interfaces.

Pruebas unitarias En la sección A.2.2 se especifica qué archivos son considerados pruebas unitarias.

- 55 archivos.
- 362 pruebas.

Pruebas de aceptación En la sección A.2.2 se especifica qué archivos son considerados pruebas de aceptación.

- 48 archivos.
- 91 pruebas.

Releases Para identificar los releases se analizaron los tags en el sistema de control de versiones. De todos los tags, algunos se tomaron como releases y otros se ignoraron. Estos son los releases que se tomaron:

- v0.1.196
- v0.2.235
- v0.3.257
- v0.4.313
- v0.4.345
- v0.5.365
- v0.5.390

Hay ciertos tags que se han ignorado como releases. Los primeros 2 no corresponden a releases. Los otros contienen muy pocos commits entre release y release (alrededor de 10 commits o menos entre release y release). Estos son los tags que se ignoraron como releases:

- v0.1.46
- v0.1.64
- v0.2.241
- v0.4.317
- v0.4.350
- v0.5.376

Tiempo entre releases A continuación se detallan el tiempo promedio, mínimo y máximo entre un release y el siguiente:

- Tiempo promedio: 54 días.
- Tiempo mínimo: 7 días.
- Tiempo máximo: 150 días.

Cantidad de commits entre releases A continuación se detallan la cantidad de commits promedio, mínimo y máximo entre un release y el siguiente:

- Cantidad de commits promedio: 59.
- Cantidad de commits mínimo: 37.
- Cantidad de commits máximo: 133. Éste es el único caso donde la cantidad de commits es tan elevada. En el resto de los casos están cerca de la media.

A.2.2. Identificación de código, pruebas unitarias y pruebas de aceptación

Para identificar qué archivos corresponden a código, pruebas unitarias o pruebas de aceptación se utilizó el siguiente criterio:

Código Todos los archivos java dentro de los directorios (o cualquier subdirectorio de estos):

- `jumi-api/`
- `jumi-core/`
- `jumi-daemon/`
- `jumi-launcher/`

exceptuando aquellos archivos que terminen en `Test.java`.

Pruebas unitarias Todos los archivos que terminen en `Test.java` dentro de los directorios (o cualquier subdirectorio de estos):

- `jumi-api/`
- `jumi-core/`
- `jumi-daemon/`
- `jumi-launcher/`

Pruebas de aceptación Todos los archivos java que se encuentren dentro del directorio (o cualquier subdirectorio) `end-to-end-tests/`.