View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Servicio de Difusión de la Creación Intelectual

Universidad Nacional de La Plata

Facultad de Informática

Trabajo Final Integrador de la

Especialización en Ingeniería de Software

Estado del arte y tendencias en Test-Driven Development

Autor: Moisés Carlos Fontela

Director: Dr. Gustavo Rossi

Estado del arte y tendencias en Test-Driven Development

Autor
Carlos Fontela, cfontela@fi.uba.ar

Palabras clave
TDD, diseño orientado a objetos, pruebas unitarias, pruebas de integración, comportamiento, pruebas de interacción, automatización.

Resumen
Test-Driven Development, o TDD como se lo conoce más a menudo, surgió como una práctica de diseño de software orientado a objetos, basada en derivar el código de pruebas automatizadas escritas antes del mismo. Sin embargo, con el correr de los años, se ha ido ampliando su uso. Se ha utilizado para poner el énfasis en hacer pequeñas pruebas de unidad que garanticen la cohesión de las clases, así como en pruebas de integración que aseguren la calidad del diseño y la separación de incumbencias. Otros han querido enfatizar su adecuación como herramienta de especificación de requerimientos. Y en los últimos años se ha comenzado a avanzar con los conceptos de TDD hacia las pruebas de interacción a través de interfaces de usuario.
Este trabajo pretende hacer una revisión del estado del arte de TDD y evaluar futuras tendencias, que inequívocamente se están dirigiendo a una integración de las distintas clases de TDD.

Índice

Autor 3

Palabras clave 3

Resumen 3

Índice 5

Introducción 7
- TDD como práctica metodológica 7
- La aparición de los frameworks y el corrimiento de TDD a UTDD 9
- Los primeros intentos de pruebas de integración 13
- La visión de las pruebas de cliente de XP 14

El punto de vista de los requerimientos y del comportamiento 14
- ATDD: TDD ampliado 14
- BDD: TDD mejorado 16
- BDD vs. ATDD 18
- STDD: ejemplos como pruebas y pruebas como ejemplos 19
- Limitaciones de la práctica de especificar con ejemplos 21
- Formatos de las especificaciones 22

El foco en el diseño orientado a objetos: NDD 22
- El problema del comportamiento 22
- Una propuesta de solución: NDD 23
- Discusión sobre pros y contras 27

Pruebas de interacción y TDD 28
- Pruebas e interfaz de usuario 28
- La falta de una visión integral 29
- Limitaciones de las pruebas de interacción 29

Tipos de pruebas y automatización 30
- Qué automatizar 30
- Formas de automatización 30

Herramientas de TDD más allá de xUnit 31
- Herramientas que hacen énfasis en la legibilidad 31
- Herramientas para crear dobles 33
- Herramientas que ponen el énfasis en BDD y STDD 33
- Herramientas para pruebas de interacción 34
- Algunas herramientas específicas 34

TDD y enfoques metodológicos 35

Estudios del uso de TDD en la práctica 37
- Validez de los estudios sobre UTDD 37
- Trabajos que fueron analizados 38
- Conclusiones extraídas de la evidencia analizada 38
- Estudios sobre variantes de TDD más allá de UTDD 39

Limitaciones de TDD y prácticas afines 40

Prácticas relacionadas con TDD 41
- Refactoring 41
- Debugging 42
- Integración continua 43

Conclusiones 43

Bibliografía y referencias 45

Introducción

TDD como práctica metodológica

Test-Driven Development[1] (TDD) es una práctica iterativa de diseño de software orientado a objetos, que fue presentada[2] por Kent Beck y Ward Cunningham como parte de Extreme Programming[3] (XP) en [1].

Básicamente, incluye tres sub-prácticas:

- Automatización: las pruebas del programa deben ser hechas en código, y con la sola corrida del código de pruebas debemos saber si lo que estamos probando funciona bien o mal.
- Test-First: las pruebas se escriben antes del propio código a probar.
- Refactorización[4] posterior: para mantener la calidad del diseño, se cambia el diseño sin cambiar la funcionalidad, manteniendo las pruebas como reaseguro.

El siguiente diagrama de actividades muestra el ciclo de TDD[5]:

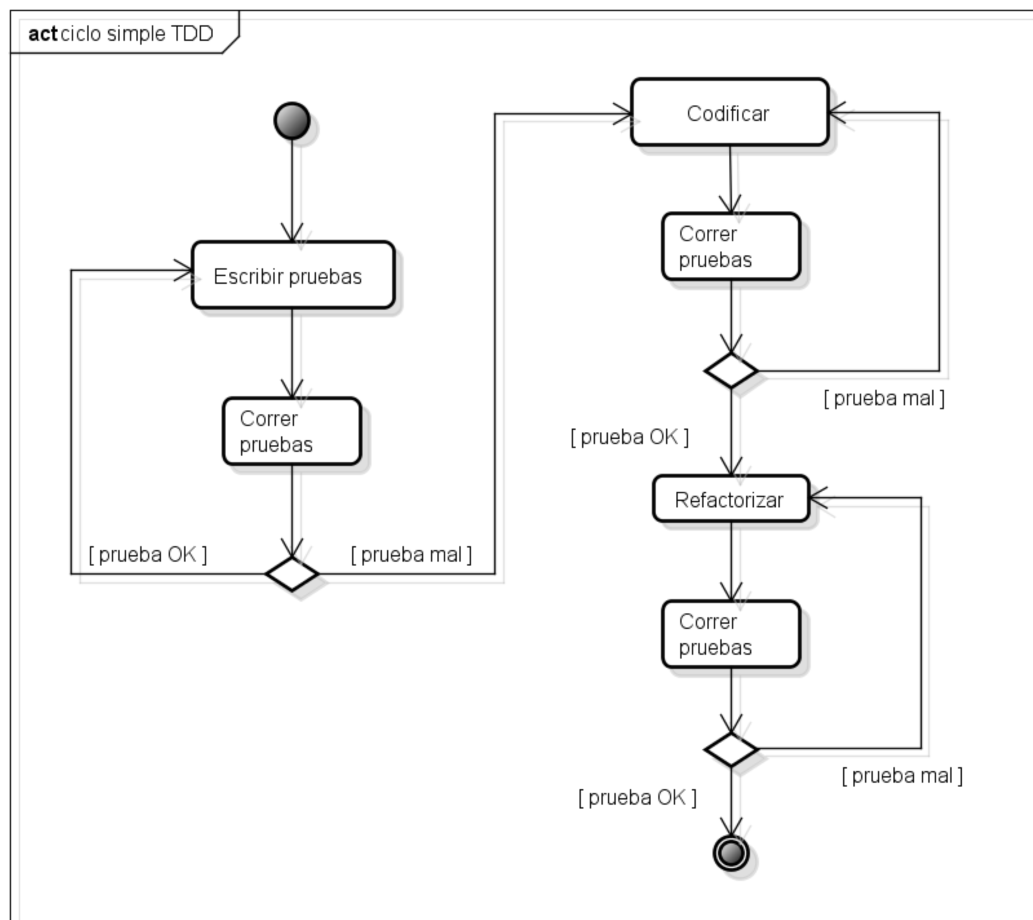
[1] En castellano, "Desarrollo guiado por las pruebas". En adelante figurará como TDD, su sigla en inglés.

[2] De todas maneras, no es una práctica totalmente nueva. Se han encontrado evidencias del uso esporádico de TDD en la década de 1960, aunque no se le hubiese dado un nombre y un alcance metodológico.

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

[4] Hemos usado el término "refactorización" como traducción del término inglés "refactoring", que se analiza luego en la sección "Prácticas relacionadas" de este trabajo.

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



Cada una de estas tres sub-prácticas acarrea ventajas:

- La automatización permite independizarse del factor humano, con su carga de subjetividad y variabilidad en el tiempo.
- La automatización también facilita la repetición de las mismas pruebas, con un costo ínfimo comparado con las pruebas realizadas por una persona. Esto es aplicable a regresiones, debugging y errores provenientes del sistema ya en producción.
- Escribir las pruebas antes del código a probar minimiza el condicionamiento del autor por lo ya construido. También da más confianza al desarrollador sobre el hecho de que el código que escribe siempre funciona.
- Escribir las pruebas antes del código a probar permite especificar el comportamiento sin restringirse a una única implementación.
- La refactorización constante facilita el mantenimiento de un buen diseño a pesar de los cambios que, en caso de no hacerla, lo degradarían.

Hay una regla de oro de TDD, que conviene destacar, y es la que dice: “Nunca escribas nueva funcionalidad sin una prueba que falle antes” [2]. Otra dice: “Si no puedes escribir una prueba para lo que estás por codificar, entonces no deberías estar pensando en codificar” [35]. El corolario obvio es que ninguna funcionalidad futura debería escribirse por adelantado, si no tiene el conjunto de pruebas que permita verificar su corrección. Si a eso se le suma que sólo se debería escribir de a una prueba por vez, tenemos un desarrollo incremental extremo, definido por pequeños incrementos que se corresponden con funcionalidades muy acotadas.

Notemos que la hemos definido como una práctica de diseño, no de pruebas (o al menos no principalmente), ya que al escribir las pruebas antes del propio código productivo estamos derivando código a partir de las mismas, que a su vez nos sirven para probar el sistema. En definitiva, las pruebas explicitan los requerimientos del sistema en forma de diseño. Beck [1] dice que si una prueba resulta difícil de llevar a código, probablemente estemos ante un problema de diseño, y sostiene además que el código altamente cohesivo y escasamente acoplado es fácil de probar en forma automática.
Sin embargo, mucha gente ha destacado su doble finalidad, enfatizando que presta un servicio importante a las pruebas, lo cual no es necesariamente malo, sobre todo teniendo en cuenta que hay muchas ocasiones en que las tareas de pruebas y correcciones implican el 50% del esfuerzo del proyecto de desarrollo.

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

La aparición de los frameworks y el corrimiento de TDD a UTDD[6]

En los primeros años, TDD se vio como sinónimo de pruebas unitarias automatizadas realizadas antes de escribir el código.

Simultáneamente, se escribieron los primeros frameworks de pruebas automatizadas: SUnit para Smalltalk y JUnit para Java. Los mismos fueron las bases para la construcción de las demás herramientas, a las que no sorprendentemente se las llamó genéricamente xUnit.

Hay dos problemas con esta definición simplificada de TDD: el hecho de que se hable solamente de pruebas, y el hecho de que se hable de que las mismas sean unitarias.

El primer problema proviene del propio nombre de TDD, que incluye la palabra “test”. En efecto, mientras por un lado se afirma que es una técnica de diseño y no de pruebas, por otro el nombre invita a pensar otra cosa. Incluso las herramientas que fueron surgiendo a partir de TDD, desarrolladas incluso por los mentores de la práctica, requerían que el código de pruebas tuviese métodos cuyos nombres empezasen con *test* y las clases fueran descendientes de algo como *TestCase* (si bien NUnit y la versión 4 de JUnit mejoraron esto, sigue estando presente la palabra *Test* en las anotaciones que utilizan). Nadie niega que TDD genera un buen conjunto de pruebas de regresión, pero ese no pretende ser su objetivo principal, sino más bien un efecto lateral positivo.

El segundo se produce porque en todos los ejemplos que plantean sus creadores, se trabaja sobre pequeñas porciones de código, e incluso ellos mismos hablan de pruebas unitarias. Asimismo, las herramientas reflejan esto en sus nombres: JUnit, SUnit, NUnit, etc. Y cuando Beck escribe su libro de TDD [2] se basa primordialmente en ejemplos de pruebas unitarias. Pero notemos que el propio esquema de TDD y sus ventajas exceden en mucho las pruebas unitarias.

Con estos equívocos de origen, sin embargo, UTDD se ha aplicado con creciente éxito. Tal vez sus mayores ventajas hayan sido, además de las previstas:

- Las pruebas en código sirven como documentación del uso esperado de las clases y métodos en cuestión.

---

[6] UTDD es una sigla menos habitual, que significa “Unit Test-Driven Development”. En castellano se traduciría como “Desarrollo guiado por las pruebas unitarias”. En adelante figurará como UTDD.

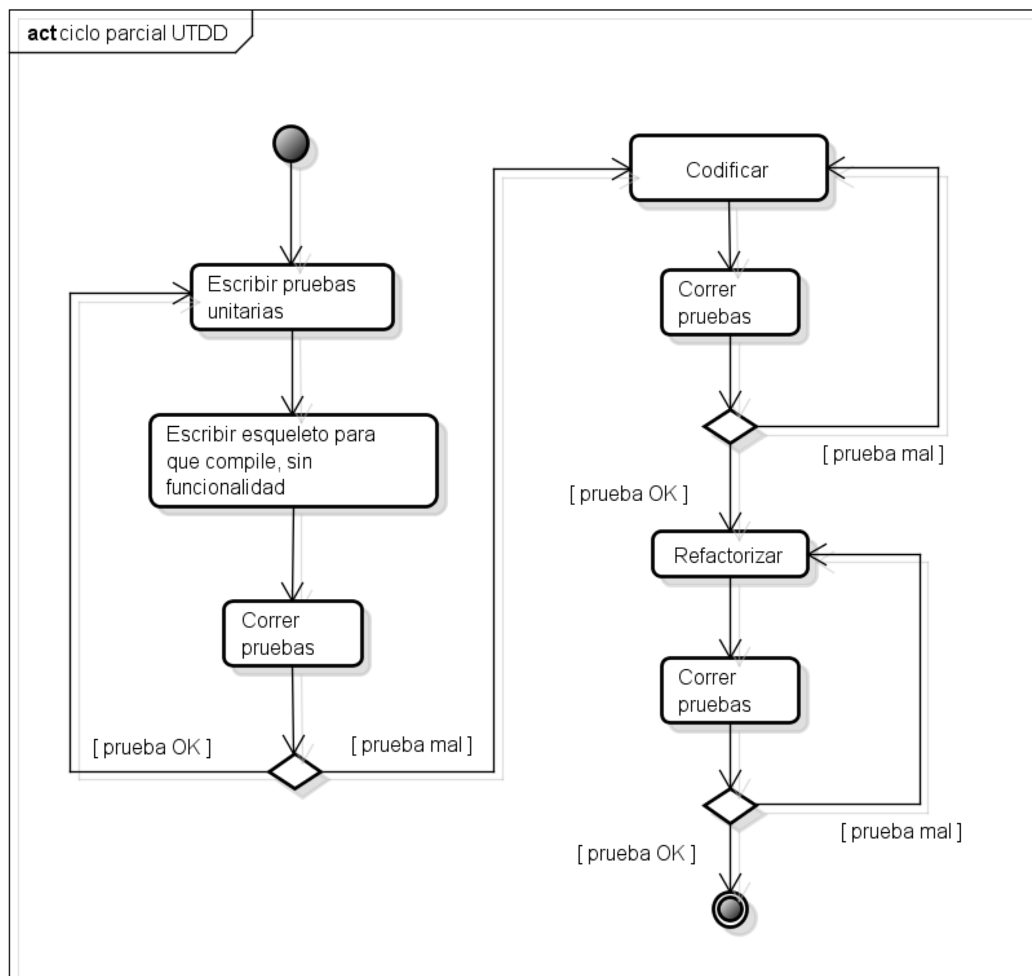
- Las pruebas en código indican con menor ambigüedad lo que las clases y métodos deben hacer.
- Las pruebas escritas con anterioridad ayudan a entender mejor la clase que se está creando, antes de escribirla.
- Las pruebas escritas con anterioridad suelen incluir más casos de pruebas negativas que las que escribimos a posteriori.

Sin embargo, con el tiempo también se evidenciaron algunas contras de la práctica:

- Tiende a basar todo el desarrollo en la programación de pequeñas unidades, sin una visión de conjunto. Hay un antipatrón clásico de UTDD que es el de una clase de prueba por clase productiva y un método de prueba por método productivo.
- Relacionado con lo anterior, muchos han criticado la pretensión de que la arquitectura evolucione sola[7], sin hacer nada de diseño previo.
- No permite probar interfaces de usuario ni comportamiento esperado por el cliente.
- Es una práctica centrada en la programación, con lo cual no sirve para especialistas del negocio ni testers. Los testers tradicionales tienden a desconfiar de ella por este mismo motivo.
- Si bien las refactorizaciones son más sencillas con pruebas automatizadas, los cambios de diseño medianos y grandes suelen exigir cambios en las pruebas unitarias.

A continuación se muestra el diagrama de actividades de un ciclo de desarrollo típico utilizando UTDD:

[7] El libro de XP [1] plantea que al realizar TDD acompañado de refactorización, queda garantizada la evolución incremental de la arquitectura, y que por lo tanto podemos evitar realizar un diseño macro antes de comenzar el desarrollo.



En el diagrama anterior se supone que, al escribir las pruebas unitarias, definimos la interfaz de la o las clases que se van a crear.
Se supone que este ciclo se ejecuta por cada porción de código[8] a desarrollar.

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

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

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

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

[8] La definición más precisa de "porción de código" es motivo de controversia y se ha ido modificando con el tiempo.

```
public class PruebaCuentaBancaria {

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

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

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

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

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

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

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

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

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

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

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

Notemos que, en el marco de la orientación a objetos esta forma de escribir pruebas no es la más ortodoxa, ya que se basa en cambios de estado de los objetos, y no de su comportamiento. Si bien hay muchas ocasiones en que el comportamiento se evidencia mediante cambios de estado en uno o más objetos, no siempre es así. Ya volveremos sobre esto.

Los primeros intentos de pruebas de integración

Decíamos que la primera falla conceptual de hacer UTDD es que se escribían pruebas para pequeñas porciones de código, típicamente clases. Como la filosofía de la orientación a objetos tiende a ver un programa como un conjunto de objetos simples colaborando mediante el envío mensajes, esto ya era problemático.

Como siempre que hablamos de integración, caemos en el problema de qué desarrollar primero y cómo probar las interacciones con módulos aún no implementados. La idea más simple es la de construir módulos ficticios o *stubs*, y fue la primera que se encaró.
Así, se pueden construir objetos ficticios que devuelvan valores fijos o tengan un comportamiento limitado, sólo para probar.

Al avanzar en estos tipos de pruebas se fue notando la necesidad de distintos tipos de objetos ficticios, que son los que Meszaros [14] llama *Test Doubles*, y que clasifica así[9]:

- *Dummy Object* (literalmente, "objeto ficticio"): son aquellos que deben generarse para probar una funcionalidad, pero que no se usan en la prueba. Por ejemplo, cuando un método necesita un objeto como parámetro, pero éste no se usa en la prueba.
- *Test Stub* (literalmente, "muñón"): son los que reemplazan a objetos reales del sistema, generalmente para generar entradas de datos o impulsar funcionalidades del objeto que está siendo probado. Por ejemplo, objetos que invocan mensajes sobre el objeto sometido a prueba.
- *Test Spy* (literalmente, "espía"): se usan para verificar los mensajes que envía el objeto que se está probando, una vez corrida la prueba.
- *Mock Object* (literalmente, "objeto de imitación"): son objetos que reemplazan a objetos reales del sistema para observar los mensajes enviados a otros objetos desde el objeto receptor.

[9] Si bien Meszaros hace una distinción muy minuciosa entre todos los tipos de dobles, e incluso hay otros autores que han hecho esfuerzos por distinguirlos (ver [15]), no entraremos en detalles, teniendo en cuenta que muchas personas definen en forma ligeramente diferente cada tipo, haciendo que las distinciones entre un tipo y otro no siempre estén claras.

- *Fake Object* (literalmente, "objeto falso"): son objetos que reemplazan a otro objeto del sistema con una implementación alternativa. Por ejemplo, un reemplazo de una base de datos en disco por otra en memoria por razones de desempeño.

Notemos que lo que se busca con cada una de estas soluciones es separar el objeto que se está probando de otros objetos, disminuyendo las dependencias. De alguna manera, lo que se está buscando es mantener las pruebas de integración como pruebas unitarias, al limitar la prueba a sólo un objeto en particular.
Por lo tanto, no estamos ante verdaderas pruebas de integración, aunque estemos más cerca. De allí que algunos especialistas recomendaron metodologías para ir reemplazando los dobles por objetos reales a medida que se iban generando sus clases. Y así llegaríamos a la prueba de integración mediante la construcción incremental a partir de pruebas unitarias.

Muchos de estos dobles no se usan solamente porque haya clases aún no implementadas. A veces usamos dobles porque es muy lento o muy complicado probar usando una base de datos o un sistema externo. Por lo tanto, estas técnicas sirven también para probar desacoplando interacciones.

Por supuesto, a medida que surgían estas técnicas, fueron construyéndose frameworks que facilitaban la generación y ejecución de pruebas que las incluyeran.

La visión de las pruebas de cliente de XP

Cuando Kent Beck presentó XP, una serie de buenas prácticas de desarrollo de software que se convertirían en el primer método denominado ágil, planteó que una de las condiciones era la presencia de un representante del cliente junto al equipo de desarrollo.
Son muchas las ventajas que Beck planteaba de la presencia de este cliente ligado al equipo. Entre ellas, no era menor la de que participase escribiendo las pruebas funcionales, que Beck mismo plantea como automatizadas. Lo curioso es que cuando él desarrolla los frameworks de pruebas[10], no prevé estas pruebas de clientes, al poner el foco en pruebas técnicas y, por añadidura, unitarias.
Y si bien podríamos afirmar que los frameworks de pruebas unitarias sirven para hacer pruebas de integración, como explicamos en el ítem anterior, tampoco son éstas las pruebas que puede llegar a redactar un cliente.

El problema con las pruebas técnicas, que de eso se trata cuando hablamos de las que se desarrollan con frameworks de la familia xUnit, sean éstas de unidad o de integración, ponen el foco en el diseño, y no en lo que esperamos de una buena prueba de cliente, que se focalice en los requerimientos. También volveremos sobre esto.

El punto de vista de los requerimientos y del comportamiento

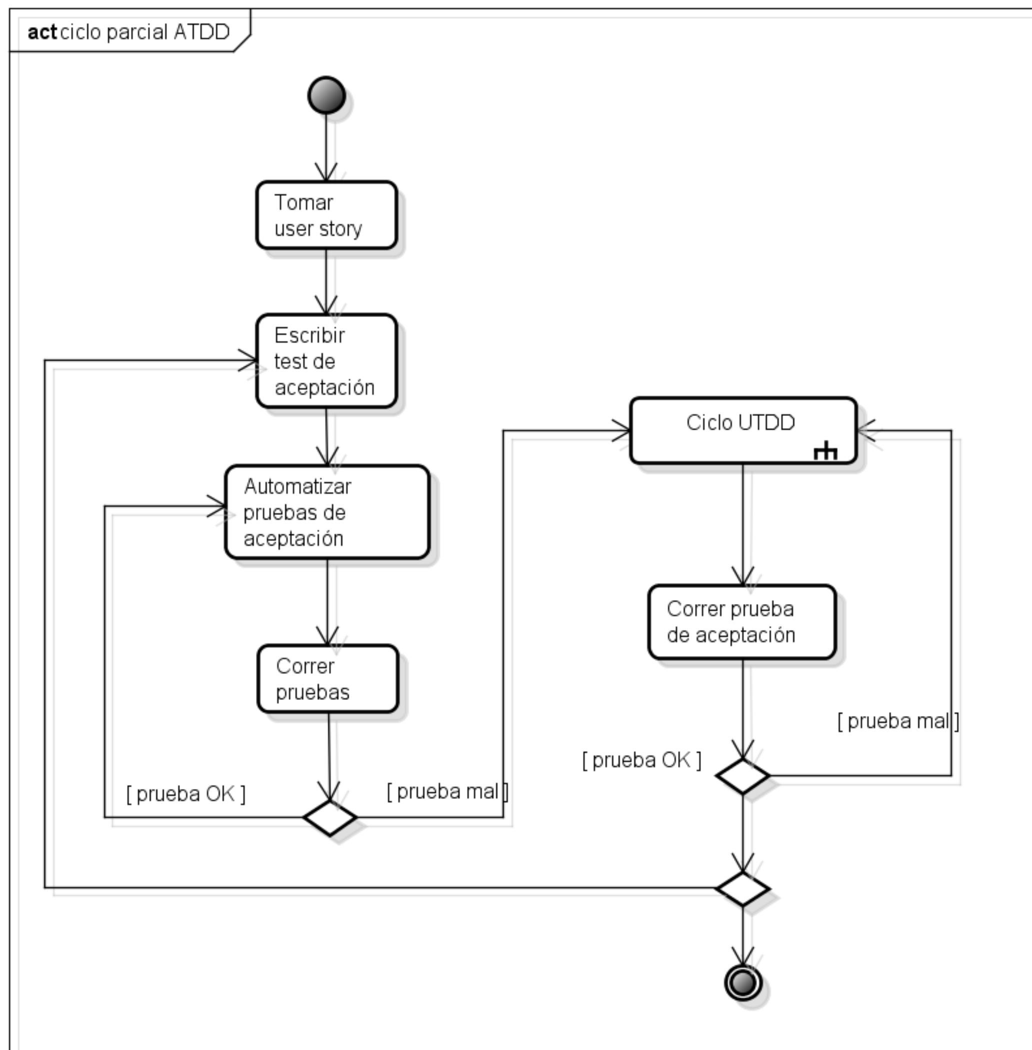
ATDD: TDD ampliado

Acceptance Test Driven Development[11] (ATDD) es una práctica de diseño de software que obtiene el producto a partir de las pruebas de aceptación. Se basa en la misma noción de Test-

[10] Kent Beck creó SUnit, y luego participó, junto a Erich Gamma, en la creación de JUnit.
[11] En castellano, "Desarrollo guiado por las pruebas de aceptación". En adelante figurará como ATDD, su sigla en inglés.

First de TDD, pero en vez de escribir pruebas de unidad, que escriben y usan los programadores, se escriben pruebas de aceptación de usuarios, en conjunto con ellos.
La idea es tomar cada requerimiento, en la forma de una *user story*, construir varias pruebas de aceptación del usuario, y a partir de ellas construir las pruebas automáticas de aceptación, para luego escribir el código.

El diagrama de actividades del procedimiento de ATDD es algo como el que sigue (notemos que debe hacerse todo el ciclo de UTDD por cada prueba de aceptación):



Lo importante del uso de *user stories* es que éstas son requerimientos simples y no representan el cómo, sino solamente quién necesita qué y por qué. Por eso decimos que representan el requerimiento, no que lo especifican.
Las pruebas de aceptación de una *user story* se convierten en las condiciones de satisfacción del mismo.

Así, ATDD fue presentada como una técnica complementaria y más abarcativa de TDD. Incluso la idea de automatización de TDD se puede llevar a ATDD: en ATDD los criterios de aceptación se convierten en especificaciones ejecutables.

La premisa fundamental es que TDD se concibió para diseñar y probar código, pero los clientes no están interesados en el código, sino en que el sistema satisfaga sus necesidades con la mejor calidad posible. ATDD, en cambio, permite conocer mejor cuándo se ha satisfecho un requerimiento, tanto para el desarrollador como para el cliente: simplemente hay que preguntar si todas las pruebas de aceptación de la *user story* están funcionando. Adicionalmente, evita los requerimientos que el cliente da por sobreentendidos y el *gold-plating*[12] de los desarrolladores, y da una visibilidad mayor del avance para ambos.
Incluso el criterio de aceptación no tiene por qué ser exclusivamente funcional. Los atributos de calidad también pueden incluirse en los criterios de aceptación, aunque no provengan de *user stories*, y ser incorporadas las pruebas de aceptación correspondientes.

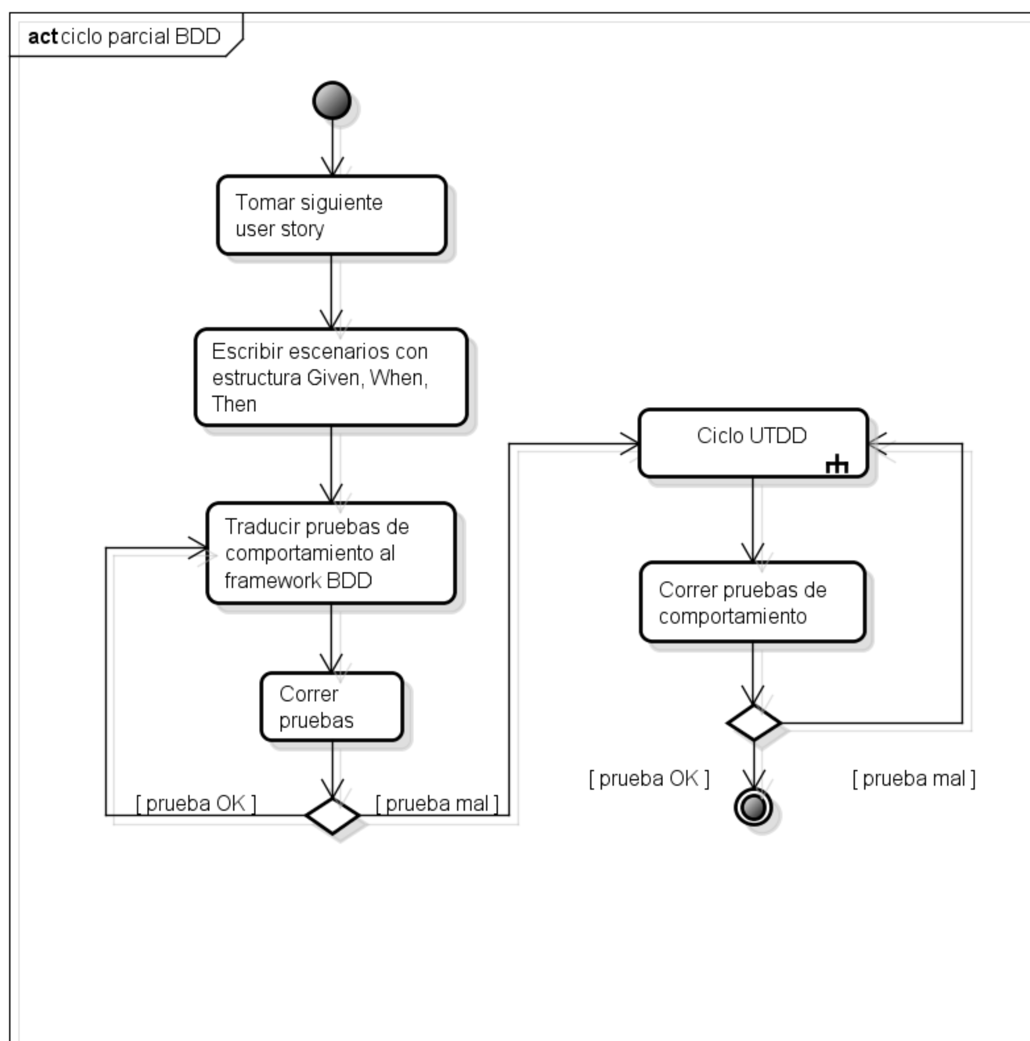
BDD: TDD mejorado

En respuesta a las contras que se encontraron a TDD, Dan North [4 y 5] empezó a hablar de Behavour Driven Development[13] (BDD).
La postura de North, en la que se ve claramente la influencia de Eric Evans y su Domain Driven Design[14] [6] (DDD), es que, en vez de pensar en términos de pruebas, deberíamos pensar en términos de especificaciones o comportamiento. De ese modo, se pueden validar más fácilmente con clientes y especialistas de negocio. Y poner el foco en el comportamiento logra un grado mayor de abstracción, al escribir las pruebas desde el punto de vista del consumidor, y no del productor.
Además, el planteo de North nos recuerda que TDD no es una técnica para probar solamente pequeñas porciones de código, sino la interacción de objetos en escenarios. De hecho, una de las objeciones que hace es que TDD pone el acento en una división estructural de las pruebas, basados en pruebas de clases y métodos, mientras que lo más lógico sería probar porciones de comportamiento esperado del sistema.

Las primeras herramientas que surgieron para realizar BDD, pusieron mucho énfasis en los cambios de nombres, suprimiendo todos los métodos *test* y cambiándolos por *should* o *must*, de modo que parecieran especificaciones escritas en un lenguaje de programación.
Pero también se asoció BDD a mejores prácticas, como la de escribir una clase de prueba por requerimiento (*user story* o caso de uso) y un método por aserción. En la filosofía de BDD, las clases y los métodos se deben escribir con el lenguaje del negocio, haciendo que los nombres de los métodos puedan leerse como oraciones. Los nombres de las pruebas son especialmente útiles cuando éstas fallan.
Esto último no es menor. Todo lo que tenga que ver con el vocabulario ayuda a unificarlo y mejorar la comunicación entre los especialistas de negocio y los desarrolladores. Notemos que esto está en línea con las ideas de DDD. Además, al hablar el lenguaje del cliente ayuda a mantener alta la abstracción, sin caer en detalles de implementación.
Incluso el superar la idea de que se trata de pruebas, hablando más bien de especificaciones, facilita su adopción, ya que pocos administradores de proyecto estaban dispuestos a escribir pruebas antes del código, aunque sí requerimientos. Lo mismo pasaba con los programadores, que muchas veces aducían que ya había testers en su organización, y ellos los estaban reemplazando sin ser reconocidos por ello.

[12] Se denomina "gold-plating", en la jerga de administración de proyectos, a incorporar funcionalidades o cualidades a un producto, aunque el cliente no lo necesite ni lo haya solicitado.
[13] En castellano, "Desarrollo guiado por el comportamiento". En adelante figurará como BDD, su sigla en inglés.
[14] En castellano, "Diseño guiado por el dominio". En adelante figurará como DDD, su sigla en inglés.

El procedimiento de un ciclo de BDD, llevando a un diagrama de actividades lo propuesto por North en sus primeros artículos, es (el parecido con el diagrama de ATDD no es casual, como veremos luego):



BDD, como era de esperarse, sufrió algunas críticas. Entre ellas, se destacan:

- La que dice que BDD es sólo un cambio de nombre a TDD, cuidando un poco el lenguaje. Esto se puede ver en innumerables sitios web y blogs de programadores.
- La que dice que BDD es simplemente TDD bien hecho [7].

Las críticas son válidas, aunque BDD no carece de mérito, al plantear que, como el código que se escribe se hace partiendo de requerimientos, ayuda a entender los requerimientos. Al fin y al cabo, los nombres que les damos a las cosas importan mucho.

Por ejemplo, la plantilla típica de una *user story* es (en su formato habitual en inglés, y en castellano):

```
As a [X]                    Como [X]
I want [Y]                  Deseo [Y]
so that [Z]                 para lograr [Z]
```

Donde Y es una funcionalidad o característica, Z es el beneficio o valor de la funcionalidad y X es la persona o rol que se va a beneficiar con la implementación de la funcionalidad.
Basándose en esta plantilla, Dan North propuso que el comportamiento de una *user story*, y por lo tanto el criterio de aceptación, se podría expresar así:

```
Given [un contexto inicial]
When [ocurre un evento]
Then [asegurar que se obtiene lo siguiente]
```

Y las herramientas de BDD creadas a partir de allí tuvieron en cuenta esto. Por ejemplo, JBehave, la primera herramienta de BDD para Java, trabaja con interfaces o anotaciones *Given* y *Event*, según la versión.

Las primeras herramientas y los artículos iniciales de BDD se basaban en encontrar cómo escribir mejor el código de TDD. En este sentido era cierto que se trataba de practicar TDD de la manera correcta y no mucho más. El foco estaba en ayudar a los desarrolladores a practicar "un buen TDD" [7].
Pero el uso de BDD, más las ideas de ATDD, han llevado a una nueva generación de BDD, tanto como práctica como por las herramientas. El foco está ahora puesto decididamente en una audiencia mayor, que excede a los desarrolladores, e incluye a analistas de negocio, testers, clientes y otros interesados.

BDD vs. ATDD

ATDD y BDD son dos prácticas casi contemporáneas, la primera surgida en 2002 y la segunda empezada a desarrollar en 2003.
Y si bien se lanzaron para resolver cuestiones diferentes (BDD surge como mejora de TDD, mientras que ATDD es una ampliación), llegaron a conclusiones para nada disjuntas. En los dos casos, se trata de actividades que empiezan de lo general y las necesidades del usuario, para ir deduciendo comportamientos y generando especificaciones ejecutables.

En realidad, no está demasiado claro el límite entre ATDD y BDD. En el fondo, es un tema más bien filosófico y de tipos de herramientas.
Mientras que en BDD solemos escribir los *user stories* en forma tradicional, en las herramientas de ATDD se enfatiza que las pruebas de aceptación también las escriban los clientes o especialistas de negocio. En muchos casos se trabaja con formularios en forma de tabla, que suelen ser mejor comprendidos por no informáticos, y de esa manera se consigue una mayor participación de personas no especializadas. Como los frameworks tabulares de ATDD no suelen generar las pruebas en forma automática, debiendo hacer la vinculación entre tablas realizadas con el cliente y el código funcional, mediante código de pruebas (*fixtures*), esto puede verse como una desventaja frente a BDD. Pero lo que ocurre es que las herramientas de BDD suponen, tal vez de manera un poco optimista, que el código escrito puede ser entendido por no informáticos: si así no fuera, la ventaja estaría del lado de las herramientas de ATDD.

Lo importante es que, tanto BDD como ATDD pusieron el énfasis en que no eran pruebas de pequeñas porciones de código lo que se desarrollaba, sino especificaciones de requerimientos ejecutables. Y así como UTDD pretende ser una técnica de diseño detallado, BDD se presenta como una de diseño basado en dominio, y ATDD una de requerimientos. Ambas técnicas ponen el foco en que el software se construye para dar valor al negocio, y no debido a cuestiones técnicas, y en esto están muy alineadas con los métodos ágiles.
En definitiva, UTDD facilita el buen diseño de clases, mientras que ATDD y BDD facilitan construir el sistema correcto.

Las mayores coincidencias entre BDD y ATDD están en sus objetivos generales:

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

La crítica principal de este enfoque de BDD y ATDD viene de un teórico del paradigma de objetos, Bertrand Meyer, impulsor del diseño por contrato y del lenguaje de programación Eiffel. Según él [8], son los contratos los que deben dirigir el diseño y las pruebas. Su postura es que, si bien se pueden derivar pruebas individuales de los contratos, es imposible el camino inverso, ya que miles de pruebas individuales no pueden reemplazar la abstracción de una especificación contractual[15].
La respuesta habitual, como lo expresa Ward Cunningham en [9], es que si bien las especificaciones abstractas son más generales que las pruebas concretas, estas últimas, precisamente por ser concretas, son más fáciles de comprender y acordar con los analistas de negocio y otros interesados. Lasse Koskela [10] amplía esto diciendo que los ejemplos concretos son fáciles de leer, fáciles de entender y fáciles de validar.

De aquí en más, consideraremos como sinónimos a ATDD y BDD.

STDD: ejemplos como pruebas y pruebas como ejemplos

Queriendo hacer confluir las pruebas de cliente de Kent Beck [1] con BDD y ATDD, surgió una técnica habitualmente conocida como Story-Test Driven Development[16] (STDD). El nombre "*story-test*", que Mugridge [24] y Kerievsky eligieron para esta práctica tal vez sea poco habitual, pero es el más difundido.

La idea subyacente es usar ejemplos como parte de las especificaciones, y que los mismos sirvan para probar la aplicación, haciendo que todos los roles se manejen con ejemplos idénticos.
Al fin y al cabo, habitualmente un analista de negocio escribe especificaciones, que le sirven a los desarrolladores y a los testers. Los desarrolladores construyen su código y definen pruebas unitarias, para las cuales deben basarse en ejemplos de entradas y salidas. Los testers, por su lado, desarrollan casos de prueba, que contienen a su vez ejemplos. En algunas ocasiones, para fijar las especificaciones, los desarrolladores y los testers le solicitan a los analistas que les den ejemplos concretos para aclarar ideas. E incluso hay ocasiones en que los propios analistas proveen escenarios, que no son otra cosa que requerimientos instanciados con ejemplos concretos.
En definitiva, hay varias ocasiones en que se plantean ejemplos que, de alguna manera, podrían servir como complementos de requerimientos. De allí que algunos profesionales hayan pensado

[15] Tal vez esto haya impulsado a algunos profesionales a buscar una convergencia entre el Diseño por contrato y TDD, como lo muestra el artículo de Chaplin [32], que introduce la noción de Contractual Test-Driven Development (CTDD).
[16] En castellano, "Desarrollo guiado por las historias de prueba". En adelante figurará como STDD, su sigla en inglés

en especificar directamente con ejemplos, o al menos acompañando los requerimientos con ejemplos.
Estos requerimientos con ejemplos tienen ciertas ventajas, de las cuales las más importantes son:

- Sirven como herramienta de comunicación.
- Se expresan por extensión, en vez de con largas descripciones y reglas en prosa, propensas a interpretaciones diversas.
- Son más sencillos de acordar con los clientes, al ser más concretos.
- Evitan que se escriban ejemplos distintos, una y otra vez, con su potencial divergencia.
- Sirven como pruebas de aceptación.

Notemos que no estamos afirmando nada de la automatización de estos ejemplos. Si bien esto suele hacerse, y en este trabajo vamos a apuntar en ese sentido, la técnica tiene su fundamento principal en la mejora de comunicación usando el lenguaje del negocio, y que ello llegue al diseño y al código.

En el enfoque tradicional, los distintos roles están establecidos de la siguiente manera:

- El analista de negocio hace las veces de “traductor” entre los clientes, por un lado, y los desarrolladores y testers, por el otro. Escucha expectativas, deseos y necesidades, y con ellos elabora especificaciones de requerimientos y los prioriza.
- El tester es quien valida el producto contra las especificaciones. Recibe especificaciones, elabora casos de prueba en base a escenarios y ejecuta casos de prueba. En general no habla directamente con el cliente, sino que usa como interlocutor al analista.
- El desarrollador es el diseñador y constructor del producto. Recibe especificaciones, elabora un diseño y lo plasma en código, para luego ejecutar pruebas unitarias y de integración técnicas. Como ocurre con el tester, tampoco habla directamente con el cliente, sino a través del analista.
- El cliente o usuario, por su lado, sólo habla con el analista.

En el enfoque de STDD, en cambio:

- El analista de negocio debe ser un facilitador de intercambio de conocimiento.
- El desarrollador es uno más de los participantes en la elaboración de los escenarios y ejemplos, que luego le sirven para desarrollar el producto y las pruebas técnicas.
- El tester es otro de los participantes en la elaboración de los escenarios y ejemplos, que luego le sirven para probar la aplicación.
- El cliente tiene un rol más activo, como autor principal de pruebas de aceptación con ejemplos, asistido por el resto de los interesados.

En cuanto a lo metodológico, STDD propone realizar un taller por iteración. Estos sirven para intercambiar ideas entre los participantes, además de establecer un lenguaje común más cercano al del negocio, en el sentido del lenguaje ubicuo de DDD [6]. También en estas sesiones surgen requerimientos implícitos y aquellos que nadie hubiese contemplado, aparte de que se reduce la brecha entre lo que entiende uno y otro participante. El propio formato de taller hace que haya una revisión implícita y que se tenga más presentes a los requerimientos que cuando hay que leerlos de un documento escrito.

Más allá de que pusimos el énfasis en que no necesariamente las especificaciones por extensión deben automatizarse, es interesante poder hacerlo, aprovechando todas las ventajas de TDD, partiendo de genuinos criterios de aceptación.
En este sentido, y siguiendo la premisa principal de TDD, no deberían desarrollarse funcionalidades que no estén acompañadas por una prueba de aceptación. Y si durante una

prueba manual, o en producción, surgiera un problema, habría que analizar qué pasó con la prueba de aceptación correspondiente antes de resolverlo.

STDD ha recibido otras denominaciones:

- Brian Marick acuñó la idea de "especificación con ejemplos"[17] [20], que es otro de los nombres más habituales de la práctica, y también ha calificado a las pruebas como "pruebas de cara al negocio"[18] [21].
- Gerald Weinberg y Donald Gause [19], han hablado de "requerimientos de caja negra"[19].
- Gojko Adzic [13] se refiere a ellos con tres nombres: el mismo de Marick, de "especificación con ejemplos", "requerimientos guiados por pruebas"[20] y el más abarcativo de "pruebas de aceptación ágiles"[21].

Más allá de los nombres, lo substancial es que en todos los casos nos referimos a una técnica de captura de requerimientos. Y eso es especialmente importante ya que, como dice Fred Brooks [12], "la más compleja de las tareas de construir un sistema de software es decidir con precisión lo que se debe construir".

Limitaciones de la práctica de especificar con ejemplos

STDD plantea escribir requerimientos por extensión. Sin embargo, no hay que olvidar que los requerimientos por intensión tienen un nivel de abstracción mayor y expresan cuestiones que no siempre nos acordamos de llevar a los ejemplos. Por lo tanto, así como en otro lado dijimos que toda *user story* debe ir acompañada de pruebas de aceptación, en STDD debemos poner el foco en que los escenarios o pruebas de aceptación deben estar agrupados por *user stories*, que deben desarrollarse si no surgen de los talleres, aunque sólo tengan una o dos oraciones.
A veces ocurre que al plantear los requerimientos como ejemplos se pierde la visión global del proyecto. En estos casos, los especialistas recomiendan, en la medida de lo posible [13], realizar *epics*[22], con pruebas de aceptación globales.

Por otro lado, no todo requerimiento puede llevarse a ejemplos en el sentido de STDD. Por ejemplo, si una aplicación debe generar números al azar, los ejemplos que podamos escribir nunca van a servir como pruebas de aceptación. Tampoco resulta fácil escribir ejemplos para pruebas de estrés o de desempeño.

Otra limitación de la práctica se da en las situaciones en que la interacción es un requerimiento más importante que el comportamiento, como ocurre con las pruebas de usabilidad. Ya volveremos sobre esto.

Si partimos de las características deseables de los requerimientos, podemos analizar qué tan bien se ajusta STDD a las mismas:

- Completo: no se puede lograr, por el mismo hecho de que se trata de ejemplos concretos.
- Consistente: se puede lograr trabajando correctamente en el taller de la iteración.
- Correcto: se puede lograr trabajando correctamente en el taller de la iteración.
- Modificable: ajuste perfecto, las modificaciones se llevan a los ejemplos.

[17] "Specification By Example".
[18] "Business-facing tests".
[19] "Black box requirements"
[20] "Test-Driven Requirements".
[21] "Agile Acceptance Testing"
[22] *Epic* (literalmente, "epopeya") es el término que se usa para referirse a iteraciones que se hacen para lograr un conocimiento más global.

- Conciso: no se puede garantizar, y en principio se contradice con la noción de especificación por extensión.
- Rastreable: se puede lograr entre pruebas y código solamente.
- No ambiguo: ajuste perfecto por la naturaleza de los ejemplos.
- Comprensible: ajuste perfecto por la naturaleza de los ejemplos.
- Validable contra las expectativas de los usuarios: ajuste perfecto.
- Verificable contra las especificaciones: ajuste perfecto.
- Independiente: se puede lograr trabajando correctamente en el taller de la iteración.
- Adecuado nivel de abstracción: no se puede garantizar, y en principio se contradice con la noción de especificación por extensión.

Formatos de las especificaciones

Hay diversos formatos que se han intentado para realizar las especificaciones en el marco de STDD: tablas, texto libre, diagramas y código. Cada uno de ellos tiene ventajas e inconvenientes, defensores y detractores, y hay herramientas para automatizar todas ellas.

Las tablas resultan una notación usual para representar entradas y salidas calculables, aunque son definitivamente inadecuadas para representar flujos de tareas, y requieren bastante trabajo de programación para generar pruebas automáticas.
El texto libre tiene el mismo problema de la dificultad de generación de código de pruebas, a lo que se suma una menor riqueza en la comunicación de entradas y salidas. Por otro lado, tiene la ventaja de que todos los perfiles pueden entenderlo.
En cuanto a los diagramas, a la vez que son muy expresivos para modelar flujos de tareas, no lo son tanto para especificar entradas y salidas. Por otro lado, si bien sirven como vehículo de comunicación en talleres de requerimientos, trabajando a mano alzada, al mismo tiempo requieren un trabajo importante para derivar pruebas automáticas de ellos.
En cuanto a las especificaciones en código, su obvia ventaja radica en la facilidad de generación de pruebas automáticas, a veces incluso a un costo nulo. Sin embargo, son las menos adecuadas como herramientas de comunicación y provocan una resistencia casi instintiva de los roles no técnicos.

Por lo tanto, hay que elegir cuidadosamente el formato que se va a utilizar, teniendo en cuenta que el mismo también va a condicionar la selección de herramientas.

El foco en el diseño orientado a objetos: NDD

El problema del comportamiento

TDD surgió en ambientes de orientación a objetos, inicialmente en proyectos Smalltalk. Y si bien es una práctica independiente del paradigma de programación, su mayor uso permanece en los proyectos de sistemas orientados a objetos.

Ahora bien, desde sus inicios, en la programación orientada a objetos, se ha puesto énfasis en el comportamiento de los objetos más que en el estado.
En ese sentido, los fragmentos que siguen son inadecuados, porque revelan cuestiones del estado de los objetos, violando el encapsulamiento y sugiriendo una forma de representación de los datos internos:

```
// mover un punto 2 píxeles en forma horizontal:
punto.setX ( punto.getX ( ) + 2);
```

```
// depósito de 1000 pesos:
cuentaBancaria.setSaldo ( cuentaBancaria.getSaldo ( ) + 1000 );

// si el cliente y la sucursal están en la misma localidad:
if ( cliente.getDomicilio().getLocalidad().equals(
            sucursal.getDomicilio().getLocalidad() ) ) { ... }
```

Claramente, las líneas que siguen serían más apropiadas, porque expresan lo mismo en términos de comportamiento:

```
punto.moverHorizontalPixeles ( 2 );

cuentaBancaria.depositar ( 1000 );

if ( cliente.estaEnMismaLocalidad (sucursal) ) { ... }
```

Este principio fue enunciado como *Ley de Demeter*[23], y pretende que los objetos revelen lo mínimo posible de su estado interno, dando a conocer solamente su comportamiento.

En el marco de TDD lo primero que hacemos es establecer la interfaz de nuestros objetos y escribir pruebas sobre ellos suponiendo que tienen esa interfaz. Por lo tanto, el encapsulamiento de los objetos se debe garantizar con ejemplos antes de pasar a la implementación.

Los frameworks tradicionales de UTDD suelen verificar el comportamiento de los objetos, verificando el cambio de estado provocado por determinados mensajes recibidos por ellos, o los valores devueltos como consecuencia del envío de otros mensajes. Ahora bien, esta forma de verificación tiene dos problemas.
En primer lugar, no siempre el comportamiento de un objeto puede evaluarse por sus cambios de estado o por los valores que devuelve. Por ejemplo, ¿qué pasa si el propósito de un mensaje es que el objeto receptor imprima algo en una impresora, envíe un mail o almacene ciertos datos en una base de datos? El chequeo de valores devueltos y cambios de estado no nos sirve para evaluar estos comportamientos.
En segundo lugar, para chequear el estado en que queda un objeto luego del envío de un mensaje, hay ocasiones en que necesitamos métodos de consulta que la clase del objeto no tiene, y nos vemos obligados a cambiar la interfaz de la misma solamente para las pruebas. Esto, de nuevo, es contrario al principio del encapsulamiento.

Una propuesta de solución: NDD

Como propuesta de solución para este problema, en OOPSLA 2004, Steve Freeman y otros tres autores presentaron un trabajo [22], en el que plantean que el comportamiento de los objetos debería estar definido en términos de cómo envía mensajes a otros objetos, además de los resultados devueltos en respuesta a mensajes recibidos, esto último propio de UTDD.
En ese trabajo, esta práctica se presentó como Need-Driven Development[24] (NDD). Posteriormente, los mismos autores comenzaron a hablar de lograr un lenguaje específico de dominio con NDD [23], lo cual lo vincula indirectamente con BDD. Curiosamente, el nombre fue abandonado en un libro [18] de dos de los autores del trabajo inicial. El método tiene una variante, habitualmente designada como *Responsibility-Driven Design*, aunque en este caso no se

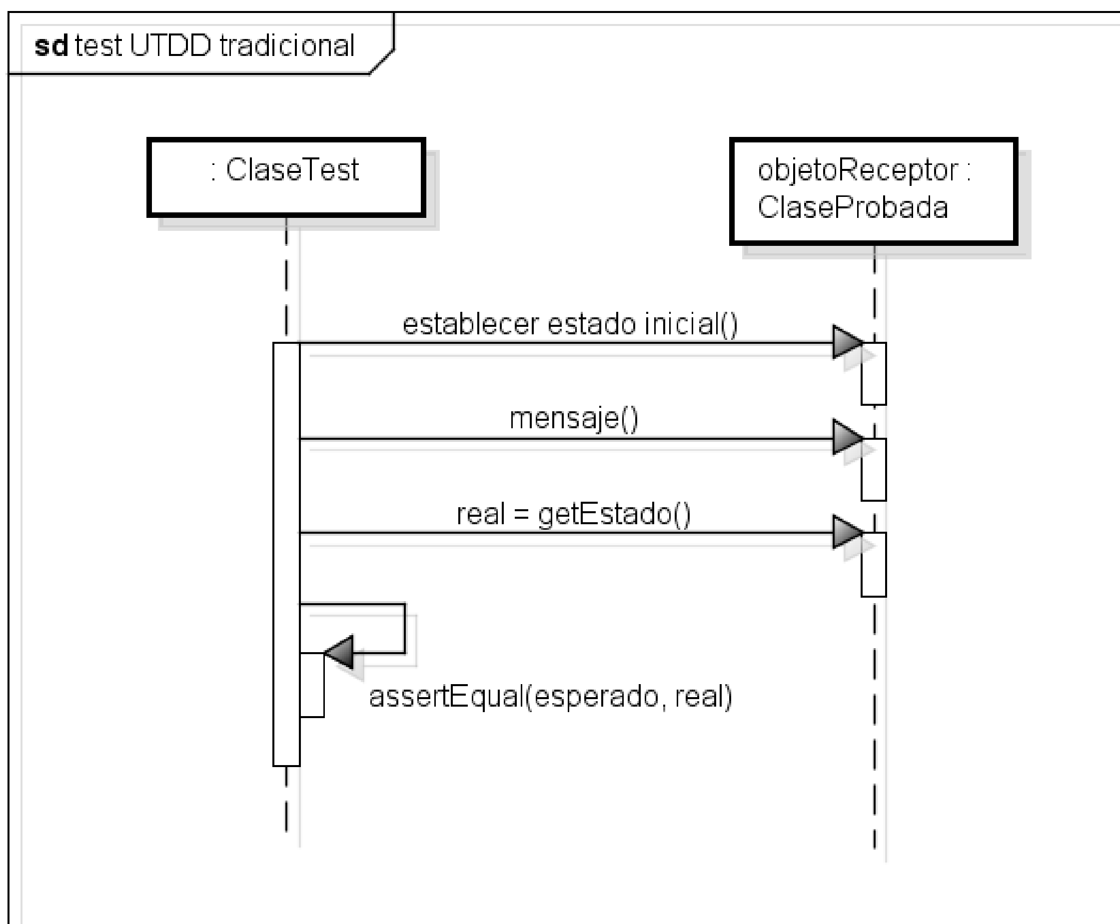
[23] Presentada en OOPSLA 1988, la Ley de Demeter, o *principio de mínimo conocimiento* es una variante del principio de bajo acoplamiento. Ver http://www.ccs.neu.edu/research/demeter/papers/law-of-demeter/oopsla88-law-of-demeter.pdf y http://www.ccs.neu.edu/home/lieber/LoD.html .
[24] En castellano, "Desarrollo guiado por las necesidades". En adelante figurará como NDD, su sigla en inglés.

requiere que las pruebas sean previas al código productivo. Para este trabajo, en aras de tener un nombre para designarla, será NDD.
El propósito fundamental es resolver un buen diseño orientado a objetos y una separación clara de incumbencias, sobre la base de:

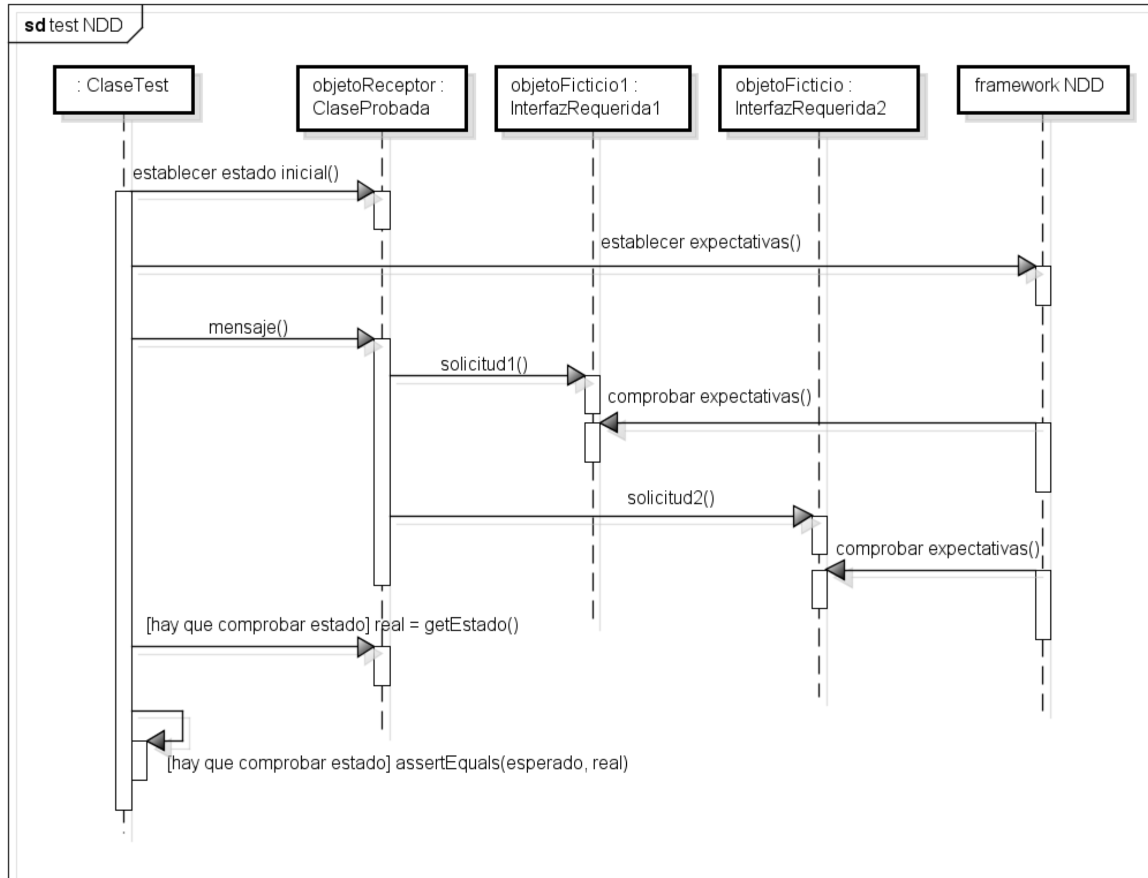
- Mejorar el código con términos de dominio.
- Preservar el encapsulamiento.
- Reducir dependencias.
- Clarificar las interacciones entre clases.

Volviendo a la técnica, la diferencia fundamental con el UTDD tradicional, en el que se basa, es que pretende definir de antemano qué mensajes emitirá el objeto sometido a prueba cuando reciba un mensaje en particular.
Como esto requiere de otros objetos que puedan recibir a su vez los mensajes emitidos por el objeto receptor, pero dichos objetos podrían no tener su clase definida en el momento de escribir la prueba, NDD propone implementarlos como objetos ficticios a partir solamente de la interfaz requerida. En definitiva, UTDD se basa en la interfaz provista por los objetos probados, mientras que NDD necesita también la interfaz requerida para los objetos vecinos, y su comportamiento, para lo cual debe generar algún tipo de objeto ficticio. Los creadores de NDD, que también han desarrollado la herramienta jMock, proponen el uso de *Mock Objects*.

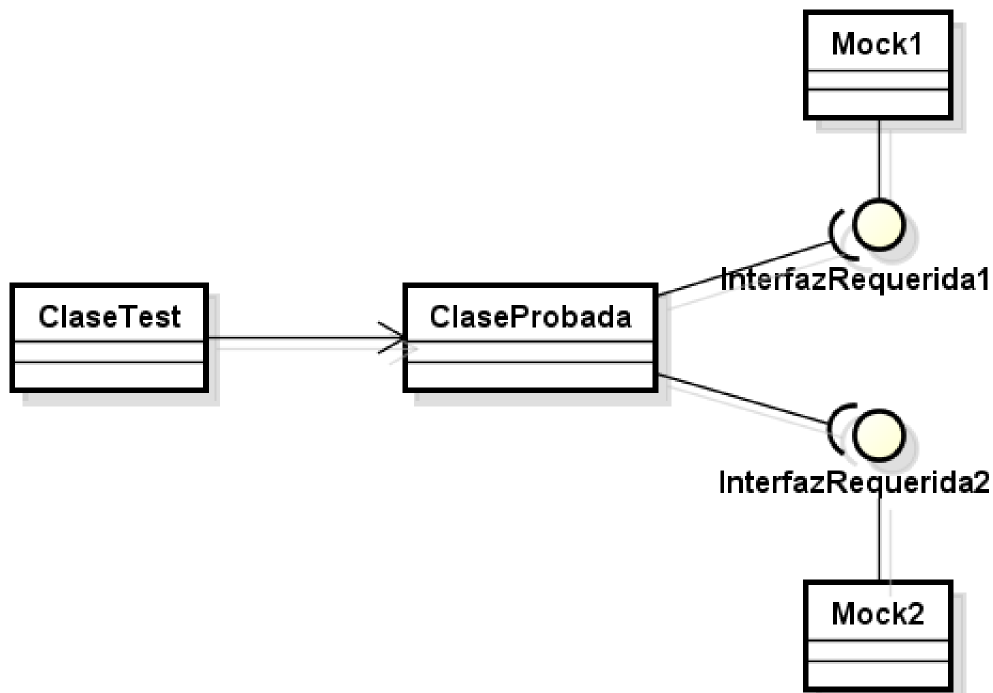
Por lo tanto, mientras el diagrama de secuencia típico de una prueba de UTDD es:



Usando NDD se convertiría en algo parecido a:



El diagrama de clases correspondiente al escenario de NDD podría ser:



Donde *Mock1* y *Mock2* son clases que implementan las interfaces *InterfazRequerida1* e *InterfazRequerida2*, y que genera el framework de *Mock Objects*.

El procedimiento a seguir para la prueba de cada método con NDD sería entonces:

- Escribir las expectativas para describir los servicios que el objeto receptor requiere del resto del sistema.
- Representar los servicios como *Mock Objects*, mediante sus interfaces.
- Crear el resto del contexto para que la prueba corra.
- Definir las postcondiciones que sean necesarias.
- Verificar las expectativas y las postcondiciones.

Como ejemplo sencillo, volvamos al ejemplo de la clase *CuentaBancaria*, y agreguemos la operación de pagar un crédito asociado a la cuenta. El ejemplo lo hemos hecho con jMock 2.

```
import org.junit.*;
import org.junit.runner.RunWith;
import org.jmock.integration.junit4.*;
import org.jmock.*;

@RunWith(JMock.class)
public class PruebaInteraccion {

      private final Mockery contexto = new JUnit4Mockery();
      private final CuentaBancaria cuenta =
              new CuentaBancaria (1234, new Cliente ("Carlos"));


      @Test
      public void deberiaPagarCreditoAsociado ( ) {
              final int MONTO_A_PAGAR = 5000;

              // vinculación de objetos Cuenta y Credito:
              cuenta.setCreditoAsociado(contexto.mock(Credito.class));

              // expectativas:
              contexto.checking(new Expectations() {{

              oneOf(cuenta.getCreditoAsociado()).pagar(MONTO_A_PAGAR);

              }});

              // método que se va a probar:
              cuenta.pagarCredito(MONTO_A_PAGAR);
      }

}
```

En el método *deberiaPagarCreditoAsociado*, estamos especificando que, cuando invoquemos el método *pagarCredito* sobre una cuenta, se debería invocar una sola vez al método pagar sobre el objeto *Credito* asociado, con el mismo monto. Notemos que *Credito* es una interfaz, para la cual todavía no hemos definido ninguna clase que la implemente (es jMock quien, como buen framework de *Mock Objects*, genera una clase ad hoc).

Una variante de NDD es usar *Test Spies* en vez de *Mock Objects*. En este caso, se verifica que el objeto receptor haya enviado determinados mensajes, recién una vez que terminó la ejecución del método que se está probando. La contra que tiene esta variante es que no es tan sencillo

asegurar un orden de invocación o invocaciones condicionadas por otras emisiones de mensajes, ya que el chequeo se hace a la salida, no a medida que las invocaciones se van produciendo. La ventaja es que hace que las pruebas sean menos frágiles, aspecto que analizaremos en el próximo ítem.

Hay algunas recomendaciones a tener en cuenta cuando se utiliza NDD:

- Generar objetos ficticios solamente para las clases que podamos cambiar. Las clases utilitarias o que no están bajo nuestro control deberíamos usarlas como están.
- Sólo generar *Mock Objects* a partir de interfaces y no de clases, porque nos interesa la comunicación entre objetos, no su estado ni cómo implementan el comportamiento[25].
- Generar objetos ficticios sólo para los vecinos inmediatos del objeto receptor.
- Evitar imponer un orden de invocación cuando esto no es necesario. En general, especificar sólo lo que se necesita.
- Evitar el uso de métodos de consulta. Esta recomendación no es de cumplimiento forzoso, pero nos va a conducir a un diseño más basado en el comportamiento.
- Si se usan muchos *Mock Objects* en una misma prueba, es un indicador de que el objeto receptor tiene demasiadas responsabilidades.

Discusión sobre pros y contras

Una consecuencia positiva del uso de esta técnica es que, al trabajar sobre la base de protocolos, además definidos en la forma restringida que necesita cada funcionalidad, se suele llegar a interfaces bien angostas. Y como siempre se crea antes la interfaz que la clase, suelen obtenerse conceptos bien abstractos definidos a nivel de interfaces. Adicionalmente, se cumple a rajatabla con el principio de programar contra interfaces.
La misma razón hace que tendamos a escribir código de pruebas menos acoplado a implementaciones particulares. Y esto, que ya de por sí es una ventaja, nos ayuda a bajar el acoplamiento de las clases de la aplicación.
Por otro lado, es innegable que NDD promueve el ocultamiento de implementación, ya que impone un protocolo para los objetos receptores, pero no dice cómo implementarlo.
También es cierto que permite definir objetos y sus pruebas, no sólo a partir de los cambios de estado que provocan los mensajes enviados, sino asimismo mediante los comportamientos evidenciados por los mensajes salientes.

Sin embargo, no todo son beneficios.
El principal problema que representa NDD es que es sensible a las refactorizaciones de los objetos colaboradores. Efectivamente, las refactorizaciones se hacen difíciles cuando los frameworks de *Mock Objects* trabajan buscando métodos por nombre usando reflexión[26], ya que cuando renombramos métodos, las pruebas dejan de funcionar. Encima, no es que no sea necesario refactorizar con NDD: es tanto o más necesario que en UTDD, ya que es lo único que puede asegurar que no habrá duplicación de código cuando lleguemos desde distintos requerimientos y mediante varias cadenas de objetos ficticios a los mismos objetos concretos.
Adicionalmente, NDD usado por gente inexperta puede empeorar el acoplamiento y la localización de responsabilidades.
Otra consecuencia, tal vez negativa, es que resulta costoso tener que crear *Mock Objects* realistas para todas las interfaces posibles del objeto a desarrollar.

---

[25] Aunque sí debe existir una cabal comprensión del comportamiento esperado del objeto del que sólo se usa su interfaz.
[26] Llamamos "reflexión" (del inglés "reflection") a la capacidad que tiene un programa para observar y opcionalmente modificar su estructura.

Por eso, no habría que exagerar. Es cierto que el comportamiento es importante, y que éste a menudo se evidencia con el envío de mensajes a otros objetos. Sin embargo, muchas veces el envío de un mensaje se hace solamente con el objetivo de cambiar el estado de un objeto u obtener un valor del mismo. En estos casos, el uso del tradicional UTDD debería bastar.

Pruebas de interacción y TDD

Pruebas e interfaz de usuario

Todas las prácticas anteriores (TDD, ATDD, BDD, STDD, NDD) fueron pensadas para especificar, diseñar y probar comportamiento de dominio. No se han planteado automatizar pruebas de interfaz de usuario, de interacción.

Incluso Rick Mugridge y Ward Cunningham [9] dicen repetidas veces que no conviene hacer pruebas automatizadas sobre la interfaz de usuario, porque ésta es muy cambiante y las pruebas se desactualizan con mucha frecuencia. También Dan North [3], Gojko Adzic [13], Steve Freeman y Nat Pryce [18] ponen énfasis en separar el qué del cómo, diciendo que el cómo es menos relevante y más cambiante, y por lo tanto no debe ser parte de TDD, sino un tema a discutir en el momento de la implementación.

Adicionalmente, ocurre que las pruebas de interfaz de usuario suelen ser muy lentas de ejecutarse, lo cual las hace poco convenientes en los términos de las barreras tecnológicas a los métodos ágiles que plantea Kent Beck el presentar XP [1].

Encima, hay cuestiones que sólo las puede probar satisfactoriamente un usuario humano. Por ejemplo, la ubicación de los botones, los colores, la consistencia de la aplicación en su totalidad, no son fáciles de automatizar. Y cada cambio en la interfaz de usuario requiere al menos un recorrido por la misma.

Pero esto está dejando de lado algo fundamental: el valor para el negocio, que se logra indudablemente mediante el comportamiento, se exhibe y se materializa, también indudablemente en una proporción muy mayoritaria de los casos, a través de la interfaz de usuario.

Por añadidura, la mayoría de los usuarios reales sólo consideran que la aplicación les sirve cuando la ven a través de su interfaz de usuario. Ergo, si cada iteración debe terminar con un entregable potencial, sólo debería considerarse terminada si lo que se entrega incluye la interfaz de usuario con su interacción y navegación probadas.

En definitiva, no se puede dejar sin probar la interfaz de usuario.

De todas maneras, no es que no existan herramientas para realizar pruebas de interacción de manera automática. De hecho, hace ya décadas que existen y fue el primer campo de automatización de pruebas.

Casi todas las herramientas permiten grabar una prueba realizada a mano para después ejecutarla en forma repetitiva, sea para chequear la resolución de un problema o para hacer pruebas de regresión. El conflicto de TDD con esta forma de trabajo es más bien filosófico: la propia noción de construir algo para luego grabar su ejecución no se lleva bien con la práctica de Test-First, básica de TDD.

Para resolver esto, se han hecho intentos de construir pruebas a partir de maquetas de interfaz de usuario y modelos, tales como en los trabajos realizados en el LIFIA[27] [26, 27 y 28]. En este sentido también destaca el trabajo de Meszaros y otros [25].

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

La falta de una visión integral

Sin embargo, todavía falta, para considerar a TDD una técnica completa, que las pruebas de interacción sobre la interfaz de usuario puedan integrarse a las pruebas de comportamiento, a nivel de modelo de negocio.
Esto es, si tenemos una aplicación con comportamiento orientado a objetos, habitualmente separamos la lógica de la aplicación, o modelo de dominio, de la interfaz de usuario. El modelo es lo que se suele someter a pruebas automáticas de aceptación.
Pero las pruebas de interacción, automáticas o no, no tienen ninguna relación con el comportamiento en este contexto.

De alguna manera, esta es una consecuencia, que no debería sorprender, de haber separado el modelo de la interfaz con el usuario. Es decir, al desacoplar ambas incumbencias, también desacoplamos las pruebas, y eso dificulta la integración entre ambas.

Sin embargo, hay algo que no se suele tener en cuenta. Y es que la interfaz de usuario también tiene comportamiento, a veces complejo, y también es susceptible de ser modelada con objetos. Por eso mismo es que ha habido intentos de usar BDD para desarrollar interfaces web. El más destacado es la utilización del patrón denominado *Page Object* [11].

De todas maneras, aun cuando no haya una integración entre ambos tipos de pruebas, hacerlas por separado es un gran avance. De hecho, probar la interfaz de usuario sabiendo de antemano que el comportamiento está funcionando correctamente y ha sido probado, es más tranquilizante, ya que seguramente nos vayamos a encontrar con menos problemas en la prueba a través de la interfaz de usuario.

Limitaciones de las pruebas de interacción

Las pruebas de interacción automatizadas tienen mala fama, a tal punto que a este problema se le ha dado un nombre: "El problema de la prueba frágil"[28]. Los problemas que suelen presentar las pruebas automatizadas de interacción generadas por grabación, son:

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

Esta fragilidad, más el tema ya expuesto de la lentitud de las pruebas a través de la interfaz de usuario son las que más quejas han provocado.

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

- Tratar de no probar en forma conjunta, o como parte del mismo juego de pruebas, la lógica de negocio y la lógica de interacción.
- Evitar hacer este tipo de pruebas en forma automática si la lógica de negocio es muy cambiante.

[28] "The Fragile Test Problem", ver [14].

- Dado que estas pruebas son más frágiles que las de comportamiento, hay que volver a generarlas cada tanto (la esperanza de vida de estas pruebas es menor que la de las de BDD o STDD, aunque puede ser similar a las de NDD).

Tipos de pruebas y automatización

Qué automatizar

A la hora de clasificar tipos de pruebas, nos encontramos con tantas clasificaciones como autores. En este trabajo, sin embargo, debemos basarnos en alguna que nos permita evaluar la conveniencia o no de automatizar cada tipo.

Si partimos de la noción de que las pruebas se hacen para controlar la calidad del producto, podemos clasificarlas en más de una dimensión. Por ejemplo, Gerard Meszaros [14] propone una clasificación tridimensional, que podemos simplificar con una enumeración como la que sigue:

- Pruebas de cliente: explicitan la intención del analista de negocio y hacen las veces de especificaciones ejecutables.
- Pruebas de componentes: definen el diseño del sistema y explicitan la intención del arquitecto; a veces se las llama pruebas de integración técnicas.
- Pruebas de unidad: definen el diseño del código y explicitan la intención del desarrollador.
- Pruebas de usabilidad: definen si el sistema es cómodo de usar (y aprender a usar) para sus usuarios.
- Pruebas exploratorias: definen si el sistema es consistente desde el punto de vista del cliente.
- Pruebas de propiedades: especifican atributos de calidad, definiendo si el sistema es seguro, escalable, robusto, etc.

Lo interesante de la clasificación de Meszaros es que él propone que algunas de las pruebas se pueden y deben automatizar, mientras que las demás deben hacerse en forma manual.
En efecto, las de cliente, las de componentes, las de unidad y las de propiedades, se pueden y conviene automatizarlas. Las primeras y las últimas se suelen automatizar usando herramientas específicas, mientras que las de componentes y las unitarias son posibles de automatizar usando herramientas de la familia xUnit y sus derivados. En cambio, las pruebas de usabilidad y las exploratorias, sólo es posible hacerlas en forma manual.

Más allá de estas cuestiones, es deseable realizar algunas pruebas en forma manual, tanto para verificar consistencia de la aplicación, cuestiones de apariencia y, sobre todo, porque solamente las pruebas realizadas por humanos van a poder detectar los errores de las pruebas automáticas mal construidas: al fin y al cabo, las computadoras sólo prueban lo que les decimos que prueben.

Formas de automatización

Ahora bien, tampoco es que toda automatización se haga de la misma manera.

En primer lugar, depende de los medios que utilicemos para la interacción entre las pruebas y la aplicación. Una aplicación construida en capas admite que las pruebas se hagan usando a las distintas capas como interfaz. Simplificando un poco, hay dos maneras principales de interactuar con la aplicación a probar:

- Mediante la interfaz de usuario, simulando un usuario humano
- Mediante una interfaz programática (*API*)

Las pruebas a través de la interfaz de usuario a veces son la única forma de probar, en aquellos casos en que la *API* del programa no prevé acceso a toda la funcionalidad desde afuera.

La otra cuestión es la manera en que materializamos las pruebas:

- Si se construyen escribiendo código (los guiones del patrón *Scripted Test* [14])
- Si se construyen grabando pruebas corridas ad hoc por un tester (según el patrón *Recorded Test* [14])

No hemos hecho un énfasis especial en este trabajo sobre las pruebas grabadas. Sin embargo, en muchas ocasiones es más sencillo grabar pruebas que escribirlas a mano, sobre todo si son pruebas a través de la interfaz de usuario.

Si combinamos ambas clasificaciones podemos obtener cuatro posibilidades.
Las herramientas del tipo xUnit trabajan con pruebas basadas en guiones que se comunican a través de la *API* del sistema.
En el otro extremo, los robots que simulan interacción de usuarios, generan pruebas grabadas que se corren usando la interfaz de usuario.
Pero hay otras dos situaciones intermedias. Por ejemplo, a través de la *API* del sistema se pueden utilizar herramientas que graban interacciones con el sistema para correrlas en forma repetitiva, tal vez a través de una capa de servicios. Y a través de la interfaz de usuario podemos también hacer pruebas basadas en guiones.

Herramientas de TDD más allá de xUnit

Herramientas que hacen énfasis en la legibilidad

Uno de los cualidades primordiales del buen código de pruebas automatizadas es que sea sencillo de mantener, para que los cambios (de diseño, de requerimientos) que provoquen variaciones en las pruebas se puedan hacer de manera sencilla.
Y como ocurre con cualquier código fuente, el fundamento básico de la facilidad de mantenimiento es la legibilidad.

Por ejemplo, la necesidad, tanto en SUnit como en JUnit 3, de que la clase de pruebas extienda a una clase especial tipo *TestCase* limita la construcción de jerarquías de pruebas y empeora la legibilidad. Otro ejemplo es la necesidad de que todos los métodos de prueba tengan un nombre que empiece con la palabra *test*, más allá de los inconvenientes, que ya comentamos, del equívoco entre prueba y especificación.
Estos problemas fueron ya abordados por NUnit, desde sus primeras versiones, al reemplazar la herencia y las convenciones de nombres por anotaciones. JUnit 4 siguió el mismo camino, luego de la incorporación de anotaciones en Java 5.

Por ejemplo, este fragmento de clase de pruebas de JUnit 3:

```
public class PruebaCuentaBancaria extends TestCase {
                              // la clase debe derivar de TestCase

      ...

      // el método debe empezar con la palabra “test”:
      public void testDepositoDebeIncrementarSaldo ( ) {
              int montoDepositado = 100;
              CuentaBancaria cuenta = crearCuentaValida();
              int saldoPrevio = cuenta.getSaldo();
              cuenta.depositar(montoDepositado);
```

```
                assertEquals("El depósito funcionó mal",
                        saldoPrevio + montoDepositado, cuenta.getSaldo() );
        }

        ...
}
```

Se puede escribir en JUnit 4 así:

```
public class PruebaCuentaBancaria {

        ...

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

        ...
}
```

En el mismo camino de la mejora de legibilidad, surgieron herramientas para escribir pruebas más flexibles en base a *matchers*[29]. De ellas, la más conocida es Hamcrest, hoy incorporada a JUnit 4 y jMock 2.
En general, el uso de *matchers* es mejor, en cuanto a su expresividad, que el uso de las aserciones tradicionales.

Por ejemplo, la siguiente aserción típica de JUnit:

```
assertEquals (x, y);
```

puede reemplazarse por esta expresión Hamcrest:

```
assertThat (x, equalTo (y) );
```

A priori, puede parecer una diferencia minúscula, pero hay varios *matchers* estándar que vienen con Hamcrest y que permiten lograr aserciones más legibles para comprobar que una variable está en *null*, o lo contrario, que un arreglo contiene un objeto, que un *Map* contiene una clave o un valor, que un número real contiene un valor cercano a otro, que determinado objeto contiene un valor mayor o menor que otro, comparaciones ignorando mayúsculas o espacios en blanco, etc.
Pero lo interesante de Hamcrest es que permite escribir *matchers* compuestos. Por ejemplo, la siguiente aserción JUnit comprueba que un arreglo contiene una de dos cadenas de caracteres:

```
assertTrue( universidades.contains("UNLP") ||
        universidades.contains("Universidad Nacional de La Plata") );
```

[29] Vamos a usar la palabra inglesa "matcher", sustantivo de "to match", que significa emparejar o "hacer juego con". Una traducción literal, del tipo de "emparejador", parece muy traída de los pelos.

Pero la aserción Hamcrest equivalente es más legible:

```
assertThat(universidades, or( contains("UNLP"),
      contains("Universidad Nacional de La Plata") ));
```

Incluso los programadores pueden definir sus propios *matchers*. Por ejemplo, sería interesante que la siguiente aserción, para comprobar que la raíz cuadrada de un número negativo no tiene resultado, fuera posible:

```
assertThat(Math.sqrt(-1), is(notANumber()));
```

Si bien en JUnit no se puede, con Hamcrest se puede hacer creando una clase de *matchers* llamada *IsNotANumber*.

Así como existe Hamcrest, hay herramientas similares para otros lenguajes y plataformas.

Herramientas para crear dobles

Como decíamos más arriba, al presentar los *Test Doubles*, desde que se necesitó contar con *stubs* y otros objetos ficticios, empezaron a surgir frameworks que facilitasen la generación de los mismos.

Hoy hay muchos frameworks disponibles, para distintas plataformas.
Tal vez el desafío que aún no se ha visto superado es la notación en que se deben escribir las pruebas. La legibilidad de las mismas es motivo de controversia, habiendo quienes afirman que el código escrito con estos frameworks es más legible que en los propios lenguajes de programación y otros que sostienen lo contrario. Lo que es innegable es la diferencia sintáctica y semántica entre ambos, como se puede ver en el siguiente fragmento de código escrito con jMock, obtenido del sitio web de este proyecto, el más popular de la plataforma Java:

```
private void initiallyLoads(Object value) {
      checking(new Expectations() {{
      oneOf (clock).time(); will(returnValue(loadTime));
      oneOf (loader).load(KEY); will(returnValue(value));
      }});
}

private void cacheHasNotExpired() {
      checking(new Expectations() {{
      oneOf (clock).time(); will(returnValue(fetchTime));
      allowing (reloadPolicy).shouldReload(loadTime, fetchTime);
              will(returnValue(false));
      }});
}
```

Puede que el objetivo de este código quede claro o no, dependiendo de cada lector. Lo cierto es que a un programador Java promedio le resultaría difícil decir qué hace, empezando por las llaves dobles.

Herramientas que ponen el énfasis en BDD y STDD

Hay muchos tipos de herramientas que buscan facilitar la tarea de derivar pruebas a partir de requerimientos. Cada una maneja distintos formatos y hace foco en distintos tipos de requerimientos. Las hay tabulares, basadas en texto, basadas en código, en formato Wiki, etc.

Las tabulares, basadas en el patrón *Data Driven Test* [14], han adquirido una mayor difusión. Sin embargo, el problema que tienen es que hay que escribir código que vincule las tablas con el código que se debe probar, por lo que hay que construir al menos tres tipos de artefactos: las tablas, el código de pegamento entre ambos, y el código productivo.

Herramientas para pruebas de interacción

Hay muchas herramientas para realizar pruebas de interacción a través de la interfaz de usuario. Existen herramientas para *Scripted Tests* y para *Recorded Tests* [14]. Las segundas son las más comunes, y existen desde hace décadas. Pero las primeras también son útiles, y se pueden usar para escribir código de pruebas que ejercita la interfaz de usuario.
Por supuesto, hay herramientas mixtas: hay muchas que generan código al grabar pruebas de interacción. Y si el código puede ser generado por un framework, también podría ser escrito por una persona.

El primer uso de las herramientas de pruebas de interacción obtenidas mediante grabación fue su aplicación a pruebas de regresión. Al fin y al cabo, es muy sencillo correr regresiones de esta manera: se corre la aplicación con el robot que ejecuta la prueba. Esto impide la típica pereza del tester que, ante una regresión, prueba sólo lo que cree que puede fallar, típicamente lo que falló la última vez.

Ahora bien, si se pueden escribir pruebas a mano y editar las generadas por una herramienta, se podría hacer TDD con estas pruebas. Por eso es que Koskela [9] considera a los frameworks de prueba grabada como herramientas de ATDD.

Algunas herramientas específicas

Cualquier lista de herramientas que hagamos va a resultar forzosamente incompleta, y se desactualizará día a día. Sin embargo, resulta ilustrativo ver algunas de las más relevantes.

Herramienta	Tipo de TDD	Estrategia de prueba	Formato pruebas	Se accede a través de	Observaciones
Fit	ATDD y STDD	Scripted Test	Tablas	API, con ayuda de código de vinculación desarrollado por programador	Desarrollado por Ward Cunningham. Bueno para resultados calculables, los flujos se pueden construir con la API FitLibrary, aunque complica una herramienta sencilla.
FitNesse	ATDD y STDD	Scripted Test	Tablas en wiki	API con ayuda de código de vinculación desarrollado por programador	Usa Fit, aunque puede trabajar con otros frameworks. Sirve para código Java, Python, C++, .NET y Smalltalk. Tiene su propio control de versiones: No muy fácil de integrar con CVS o SVN

Herramienta	Tipo de TDD	Estrategia de prueba	Formato pruebas	Se accede a través de	Observaciones
JBehave	BDD y STDD	Scripted Test	Texto con plantilla de user story	API	Primera herramienta de BDD para Java. Es una extensión de JUnit. Corre bien en conjunto con IDEs. Muy amigable para desarrolladores. No requiere código para vincular especificaciones con pruebas. Menos amigable para perfiles no técnicos.
RSpec	BDD y STDD	Scripted Test	Texto con plantilla de user story	API	Primera herramienta de BDD para Ruby, muy similar en pros y contras a JBehave
Concordion	BDD y STDD	Scripted Test	Texto libre en HTML	API, mediante código de vinculación entre pruebas y especificación	Es una extensión de JUnit. Corre bien en conjunto con IDEs. Muy amigable para desarrolladores. Relativamente amigable para perfiles no técnicos. Hay versiones para .NET, Ruby y Python.
TextTest	ATDD y STDD	Scripted Test	Texto en una interfaz gráfica	GUI o API	Especial para probar workflows y regresiones. Para cualquier lenguaje.
easyb	BDD	Scripted Test	Texto con plantilla de user story	API	Basado en Groovy
Cucumber	BDD	Scripted Test	Texto con plantilla de user story	API	Basado en Ruby y RSpec, tiene una extensión para Java llamada Cuke4Duke
Selenium	Interacción	Recorded Test	Editable en varios lenguajes de programación	Interfaz web	Para aplicaciones web
CubicTest	Interacción	Recorded Test	Editable en formato gráfico	Interfaz web	Usa Selenium
StoryTestIQ	Interacción	Recorded Test	Editable en formato wiki	Interfaz web	Usa Selenium
HttpUnit	Interacción	Scripted Test	Texto	Interfaz web	

TDD y enfoques metodológicos

TDD nació con un enfoque metodológico que era el de XP, por otro lado bastante laxo. Sin embargo, esto ha ido evolucionando, y en esa evolución mucho tuvo que ver la historia de las herramientas y técnicas particulares.

UTDD supone un trabajo en forma *bottom-up* por su propia filosofía de diseñar pequeñas porciones de código. Por supuesto, esto puede ser problemático si, como ocurre muy a menudo y es lo más recomendable, se definen los requerimientos sin detalles de implementación. Por otro lado, el enfoque *top-down*, de ir del diseño de la interacción al del comportamiento y de allí al de las clases tiene reminiscencias muy antiguas en la historia del desarrollo de software, no es siempre conveniente, sobre todo por la necesidad de gran cantidad de objetos ficticios.

Ahora bien, ¿cuál es el enfoque correcto? La verdad es que hay muchas visiones. Si bien no es el foco de este trabajo, vamos a detenernos brevemente.

Fowler [15] propone hacer lo que él llama "diseño de afuera hacia adentro". En esta metodología, se parte de la interfaz de usuario y se pasa a la capa media, y así sucesivamente. El problema de esta solución es que requiere muchos objetos ficticios aún no desarrollados. NDD lo resuelve mediante *Mock Objects*, pero no necesariamente parte de la interfaz de usuario.
El enfoque de BDD es similar, aunque no idéntico: partir del qué, y de a un requerimiento por vez. No obstante, se propone explícitamente no hacer pruebas de interfaz de usuario [9].
NDD [18] parte de *user stories*, haciendo ATDD con varios ciclos de UTDD dentro, usando objetos ficticios para asegurar que las pruebas sean siempre unitarias. También se recomienda no incluir la interfaz de usuario en este proceso [18].

La ventaja de cualquiera de los enfoques *top-down* es que podemos atacar de a un requerimiento por vez, lo cual está muy alineado con las ideas de los métodos ágiles, que buscan cerrar cuanto antes pequeñas funcionalidades, en forma incremental.
Una buena forma de acompañar esta forma de trabajo es usar el patrón que Meszaros [14] llama *Testcase Class Per User Story*, en el cual se van haciendo casos de prueba para un requerimiento por vez. Lo extraño es que el propio Meszaros sugiere que el patrón *Testcase Class per Fixture* [14] está más alineado con BDD, confundiendo la idea de BDD de enfocarse en el comportamiento con la noción de que conviene que un conjunto de pruebas comiencen todas con el mismo estado del sistema. Para usar lo mejor de los dos enfoques podemos realizar un diseño del tipo del patrón *Service Facade* [17].

En el LIFIA [26, 27 y 28] se está trabajando sobre WebTDD, un enfoque metodológico basado en los principios ágiles y en su preferencia por los ciclos de desarrollo cortos. Lo interesante del planteo es que se parte de pruebas de interacción. A su vez, utilizan un lenguaje específico de dominio llamado WebSpec, que mediante diagramas, sirve para capturar requerimientos de navegación, de interfaz de usuario y de interacción. WebTDD se basa en prototipos con maquetas de interacción para acordar con los usuarios y resulta más formal y prescriptivo en cuanto a la interacción que las *user stories*, ya que permite definir precondiciones e invariantes en las interacciones.
Si bien la idea de WebTDD es ir de un ciclo tradicional de TDD para la interfaz de usuario, utilizando objetos ficticios para los objetos de dominio, que luego se implementan con UTDD, los trabajos desarrollados hasta hoy se centran más bien en cuestiones de diseño y desarrollo de la interfaz de usuario.

Todas estas maneras de organizar las pruebas pueden resultar confusas si no se analiza el tipo de TDD a emplear.
Tal vez los enfoques mixtos sean los que proveen más valor, empezando con BDD sin interfaz de usuario, siguiendo con NDD a la manera del libro de Freeman y Pryce [18] y de allí pasando a un UTDD clásico para reemplazar los *Mock Objects*. El desarrollo de la interfaz de usuario y de la persistencia quedaría para después, con sus pruebas. Por supuesto, de adoptar un enfoque ágil, por cada requerimiento se debe hacer un ciclo completo.
Meszaros [14] presenta un patrón al que llama *Layer Test* y que consiste en hacer pruebas en forma separada para cada capa, haciendo de vez en cuando alguna que cubra todas las capas, y utilizando objetos ficticios para simular las otras capas.

Muchos trabajos se han hecho para guiar el desarrollo desde la interfaz de usuario. Ello es razonable en aplicaciones de escasa complejidad, o aquellas que son simples puentes entre la

interfaz de usuario y un repositorio persistente. Pero en una aplicación orientada a objetos el comportamiento principal está en el modelo de negocio, no en la interfaz de usuario. Tal vez sería interesante tratar de encontrar un mapeo entre objetos de negocio e interfaz de usuario, y a partir de allí, llegar a escribir pruebas de interacción automatizadas.
De esa manera, coexistirían dos tipos de prueba: las de comportamiento, basadas en BDD (a su vez basadas en UTDD y NDD para guiar el diseño micro y las interacciones), más estables e independientes de la interfaz; y las de interacción, basadas fuertemente en la interfaz de usuario.
Incluso el modelo se podría replicar también para pruebas funcionales que chequeen interfaces con sistemas externos o la persistencia.

Estudios del uso de TDD en la práctica

Validez de los estudios sobre UTDD

Como hemos dicho anteriormente, TDD fue entendida y utilizada en primer término en su forma de UTDD. Por ello, ya existe mucha evidencia empírica disponible sobre el uso de UTDD en el mundo real. Tal vez, entonces, podríamos considerar que no es más válida la afirmación de Kent Beck, realizada en 2002 [2], de que no se había demostrado nunca la diferencia en calidad, productividad o deleite a favor de TDD respecto de sus alternativas.
Lamentablemente, no todo es tan así. En primer lugar, porque no todos los estudios que se han hecho comparan lo mismo ni se detienen a medir todos los aspectos. Por ejemplo, se esperaría que el tiempo de desarrollo, siendo como es un factor crítico en la mayor parte de los proyectos, y tal vez más aún en proyectos ágiles, fuese menor usando TDD que si no seguimos la práctica. Pero los estudios no nos ayudan a determinarlo con claridad, ya que no todos miden los tiempos de proyecto de la misma manera ni incluyen el mismo alcance para el ciclo de vida. Hay estudios que excluyen el tiempo de corrección de los defectos del sistema en producción, probablemente por la dificultad de poder medirlo en un experimento acotado en el tiempo; pero precisamente allí se puede esperar un menor retrabajo gracias a TDD. Por la misma razón, el mantenimiento evolutivo, probablemente favorecido por el uso de TDD, tampoco se mide, lo cual sólo tendría sentido para aplicaciones efímeras.
En segundo lugar, hay estudios que han comparado el mismo desarrollo, con y sin TDD, pero esto resulta poco común: la mayor parte de los mismos se enfoca en comparar desarrollos diferentes, lo cual hace dudosa la validez de los resultados obtenidos.
Un tercer inconveniente es que a veces se comparan proyectos disímiles, como uno de juegos sobre dispositivos móviles con otro de control industrial. Estos estudios los hemos desechado.
Finalmente, son muy pocos los experimentos en los que se mantuvo a la gente bajo análisis en el desconocimiento de que su trabajo estaba siendo analizado. Y es sabido que el revelar a las personas que están siendo observadas suele generar resultados sesgados.

Otro inconveniente es que en todos los casos de estudio se analizan proyectos desarrollados desde cero con TDD, lo cual está muy en línea con la práctica, pero es poco realista en contextos industriales. De hecho, muy probablemente TDD no sea una buena opción para desarrollar cambios sobre funcionalidades ya existentes de grandes aplicaciones que se hicieron sin pruebas automatizadas. Pero hay zonas grises que tampoco han sido analizadas, por ejemplo, qué tan bueno es TDD para agregar nuevas funcionalidades a aplicaciones que no han seguido a TDD en su desarrollo.

Por lo tanto, podemos seguir sosteniendo que no contamos con suficiente evidencia empírica que cubra todos los tipos de proyectos, tecnologías, tipos de clientes y lenguajes. No obstante, estudios hay, y en ello nos detendremos en este punto. Al fin y al cabo, siempre es difícil realizar experimentos del mundo real en ingeniería de software, y por ello muchas personas los

descalifican con expresiones tales como "investigación en pequeño", agregando que no es válida la extrapolación a grandes proyectos. Pero algo se ha hecho, y sería desatinado descartarlo sin más.

Trabajos que fueron analizados

Los trabajos más representativos que hemos encontrado han sido:

- El realizado en una universidad sueca [29], que expone el resultado de una investigación industrial que analiza el uso de TDD en 48 trabajos que examinan casos concretos del empleo de la práctica.
- Un estudio de 4 grupos de trabajo en la industria, en las empresas IBM y Microsoft, de los cuales no todos siguen necesariamente otras prácticas ágiles. Uno de los equipos analizados está disperso en varios países [30].
- Un experimento llevado a cabo en una empresa desarrolladora de software en España, en la que todos los desarrolladores eran graduados universitarios en computación, tenían amplio manejo de programación y modelado de software más 5 años de experiencia con Java, que fue la plataforma elegida [31].
- El examen de un caso de desarrollo de aplicaciones móviles con Java ME [33].
- Una comparación entre dos equipos, uno utilizando una metodología en cascada y otro con XP y TDD, trabajando ambos sobre el mismo alcance de un pequeño desarrollo en una fábrica de software china que trabaja para mercados del primer mundo [34].
- Y hay unos 20 papers adicionales que hemos consultado, de menor representatividad que los anteriores.

Conclusiones extraídas de la evidencia analizada

Los principales resultados cuantitativos permiten concluir:

- En cuanto a la calidad externa, medida a través de la densidad de defectos encontrados en las pruebas funcionales que se realizan antes de pasar a producción y de cualquier necesidad de mantenimiento, se ha reportado una mejora de entre el 40% y el 90%. También se ha medido la cantidad de pruebas funcionales que pasan, observándose una mejora del orden del 18% según [30].
- En cuanto al tiempo de desarrollo, se ha reportado un mayor tiempo, del entre 15% y el 35%, aunque no siempre se aclara si fue medido antes de pasar a producción y si se tuvo en cuenta el retrabajo: o bien está claramente explicitado que estas actividades no están incluidas, o no se dice nada de ellas, lo que lleva a pensar que no se las tuvo en cuenta. Habría que analizar, como dijimos antes, los tiempos de corrección de errores y de mantenimiento post-producción, para tener un indicador más completo. Y es de suponer que la comparación no sería igual de desfavorable en estos casos, ya que los tiempos deberían descender por la mejora en la calidad del diseño. Por supuesto, esto dependerá de la longevidad y complejidad de la aplicación.
- En los estudios que han pretendido que se escribiera la misma cantidad de pruebas, con o sin TDD, ésta muestra mejores resultados en tiempo y calidad. Si bien son pocos los estudios que han medido esto, es algo bastante razonable de esperar, ya que TDD es una técnica de diseño, y plantear un diseño tradicional anticipado, y luego escribir las pruebas, llevaría a hacer el trabajo de diseño dos veces: antes y después de la codificación.
- En cuanto al conocido problema de la introducción de errores en situaciones de cambios funcionales y de corrección de errores, se reporta un descenso en la cantidad de errores introducidos.

- Incluso los tiempos de corrección de errores y de cambios muestran descensos interesantes.
- Entre los trabajos consultados, sólo [34] ha medido diferencias en el tamaño del código, y ha encontrado una mejora de reducción de tamaño usando TDD.

Hay también aspectos cualitativos que se analizan en los trabajos analizados. Entre ellos destacan:

- El deseo que muchos desarrolladores han manifestado por utilizar TDD en mayor medida a la que lo hacen en sus proyectos.
- La presión interna de la organización pone en riesgo el uso correcto de TDD, tanto por no comprender la importancia de escribir la cantidad de pruebas necesarias como por la tendencia a abandonar la práctica cuando los cronogramas se vuelven más ajustados.
- Cuando se presentan muchas trabas, humanas, metodológicas o tecnológicas, para hacer TDD, los desarrolladores suelen dejar de lado la práctica en poco tiempo.
- La satisfacción laboral suele ser mayor para los desarrolladores que con TDD pueden ver que las pruebas corren satisfactoriamente. Esto debería redundar en mayor productividad, aunque no se ha medido esta correlación.
- En proyectos chicos y de aplicaciones efímeras, o cuando los desarrolladores no se enfrentan habitualmente al mantenimiento o de errores de producción, no valoran el aumento de la calidad interna que da TDD y sólo se fijan en el tiempo perdido por tener que escribir más código. Quizá sólo valoren el hecho de que mejora la calidad del producto antes de pasar a producción y el tener que hacer menos retrabajo, pero esto no surge de los estudios en forma categórica. La importancia de explicar claramente las ventajas de TDD es central en estos casos, y puede ser reforzada por la evidencia empírica [36] de que el mantenimiento (correctivo o evolutivo) lleva a cometer 40 veces más errores que el desarrollo de cero.

Asimismo, hay cuestiones que, o bien no han sido medidas, o bien no se encontrado evidencia concluyente. Por ejemplo:

- Respecto de la calidad de los casos de prueba producidos por los desarrolladores, no hay suficientes estudios sobre este tema, que es tan crítico para realizar bien TDD. Sí se menciona en más de un estudio que probablemente los desarrolladores no estén del todo preparados para escribir buenos casos de prueba, pero sin evidencias que sostengan esta afirmación. Si así fuera, sería interesante saberlo, pues se podría mitigar con mayor capacitación para los desarrolladores en este tema.
- La flexibilidad, entendida como el esfuerzo que se necesita para adaptar la aplicación en respuesta a un cambio de requisitos, no ha sido medida en todos los estudios y no se observa una correlación clara en los que se intentó medirla, aunque sería interesante analizarlo, pues una de las premisas teóricas de TDD es que la práctica debería colaborar en este sentido.
- No se ha encontrado una correlación clara entre la mayor adherencia al protocolo de TDD y la calidad del producto.
- Si bien casi todos los estudios han medido la calidad externa, basándose en la densidad de defectos, no se nota el mismo esfuerzo para medir la calidad interna o calidad del diseño, que es uno de los objetivos declarados más importantes de TDD.

Estudios sobre variantes de TDD más allá de UTDD

Si bien hay bastantes experimentos realizados con UTDD, otras variantes de TDD no han tenido tanta suerte.

El mejor estudio que hemos consultado es el trabajo sobre STDD de Park y Maurer [37], quienes afirman que a la fecha de esa publicación (septiembre de 2009) no había ningún estudio sistemático publicado sobre STDD. Este trabajo investiga qué ventajas y desventajas se encuentran a STDD, sobre la base de 49 trabajos parciales publicados con anterioridad.

Las observaciones más relevantes recogidas de este trabajo son:

- Respecto del costo, se encuentran algunas evidencias a favor de que STDD ayuda a reducirlo, aunque también hay estudios que dicen que el costo de escribir y mantener las pruebas no compensa sus beneficios. Lo que sí es una constante es que con STDD se realizan pruebas de regresión a un costo menor.
- Respecto de los tiempos, casi toda la evidencia está a favor de STDD, lo cual parece ser mérito de la facilidad de adaptarse a los cambios de requerimientos, la verificación de los mismos en forma continua, la mayor facilidad para estimarlos y la facilidad que tiene el cliente de poder medir el progreso del proyecto.
- En cuanto a la comunicación, que la mayor parte de los trabajos muestran como la mayor virtud de STDD, se hace hincapié en que ofrece una mejor comunicación entre todos los interesados, dando mayor confianza en cuanto a progreso y entregables.
- En cuanto a las pruebas, se ha observado una mayor conciencia sobre la necesidad de pruebas en todo el equipo. Como contra, se observa que los desarrolladores con escasa formación tienden a concentrarse en hacer pasar pruebas y no tanto en colaborar con el cliente para escribir especificaciones que le brinden real valor.
- Muchos trabajos manifiestan la dificultad de hacer pruebas de regresión sobre las propias pruebas de aceptación, lo que hace que los cambios de requerimientos sean riesgosos.
- Siendo Fit y Selenium casi las únicas herramientas utilizadas, con un predominio de la primera sobre la segunda, hay bastantes cuestionamientos a las mismas, entre ellas la falta de habilidades para refactorizar pruebas de STDD.
- Es complicado organizar las especificaciones de forma tal de poder ver el todo. sería bueno contar con herramientas que permitiesen agrupar y buscar pruebas por *user story*, por iteración, por conjunto de funcionalidades, etc.
- Muchos de los trabajos consultados hacen foco en el diseño de código, lo cual resulta sorprendente si se piensa que STDD no es una práctica para desarrolladores, o al menos no para ellos solos.

Limitaciones de TDD y prácticas afines

Como cualquier otra práctica metodológica, TDD tiene sus límites. Por eso, a continuación enumeramos algunas situaciones en las que TDD no es una buena idea:

- Para el desarrollo de mantenimiento evolutivo de software ya construido, en el que no se haya utilizado TDD en sus versiones previas. Lo que ocurre es que, al no haber pruebas de regresión automatizadas previas, se hace difícil ver materializadas las ventajas de TDD en las refactorizaciones y los cambios de requerimientos. TDD fue pensada como una práctica destinada a construir nuevas aplicaciones desde cero, y todos sus impulsores destacan esto.
- Para el diseño y pruebas de interfaces de usuario, aun cuando hay herramientas y se han hecho algunos intentos, como dijimos más arriba. Por supuesto, no es imposible, pero es dudoso que sea una opción mejor que otras prácticas metodológicas.
- Para operaciones contra bases de datos. Si bien es posible y técnicamente sencillo utilizar TDD en estos casos, consume mucho tiempo de ejecución y es necesario escribir mucho código de inicialización, además de trabajar mucho para crear la infraestructura de

pruebas. Tal vez la existencia de mejores herramientas en el futuro podría relativizar esta afirmación.

- Para infraestructura de servicios web. Aun cuando se puede argumentar que es muy sencillo escribir pruebas funcionales de servicios web, hay cuestiones, tales como el establecimiento de valores en variables de entorno, el manejo de identificación de clientes por su IP o algún otro identificador de red, la manejo de permisos, etc., que ya no son tan sencillas y son parte de lo que debería probarse del funcionamiento de los servicios web.

Hay otras cuestiones adicionales.
En general se asocia TDD con evitar el desarrollo de un diseño completo antes de comenzar a trabajar (BDUF[30], lo llama Kent Beck [1]), esto podría ser una limitante en caso de proyectos grandes o complejos, que sí necesiten un diseño previo, aunque sea a nivel macro. Abandonar el dogmatismo extremo es un buen remedio en estos casos.
Existen muchas más herramientas para UTDD que para otras formas de TDD, además de que están más desarrolladas y difundidas. Esto se puede ver como una limitación del modelo, cuando no lo es: mejores herramientas cambiarían esta percepción.
El uso de UTDD escribiendo pruebas para absolutamente todo el código no es una buena práctica, aun cuando haya herramientas que miden la cobertura de las pruebas y fomenten que se llegue al 100% de cobertura. El problema con cubrir todo el código es que el costo de UTDD se haría innecesariamente elevado. Quizá deberíamos recordar mejor la máxima de Kent Beck [1], de probar "aquello que se supone que puede fallar". Como corolario, escribir pruebas de métodos que asignan o consultan valores de atributos ("getters" y "setters"), no es una buena idea. Tampoco lo es pretender cubrir cada una de las ramas más insignificantes de nuestra aplicación.
La cuestión de que TDD se basa en pruebas automatizadas puede llevar a un exceso de confianza que resulte perjudicial. Lo que hay que hacer es no olvidar que, aunque automáticas, las pruebas las escribimos humanos, y por lo tanto las propias pruebas pueden tener errores. Obviamente, en la medida de que revisemos las pruebas de aceptación con clientes y usuarios, al menos éstas serán menos propensas a errores.

Prácticas relacionadas con TDD

Refactoring

Refactoring o refactorización es una técnica habitualmente asociada a XP que consiste en mejorar la calidad del código después de escrito, sin modificar su comportamiento observable. Es, en definitiva, una técnica de mejora del diseño del software.
Uno de los objetivos de la refactorización es la mejora del diseño después de la introducción de un cambio, con vistas a que las sucesivas modificaciones hechas a una aplicación no degraden progresivamente la calidad de su diseño.
La primera publicación difundida en que usó el término fue el artículo de Opdyke y Johnson [38].

La refactorización aparece relacionada con TDD en dos sentidos:

- El ciclo de TDD incluye una refactorización luego de hacer que las pruebas corran exitosamente, para lograr un mejor diseño que el probablemente ingenuo que pudo haber surgido de una implementación cuyo único fin sea que las pruebas pasen.

[30] De "Big Design Up Front" o "gran diseño al comienzo".

- La refactorización segura exige la existencia de pruebas automáticas escritas con anterioridad, que permitan verificar que el comportamiento externo del código refactorizado sigue siendo el mismo que antes de comenzar el proceso.

La relación entre ambas prácticas es, por lo dicho, de realimentación positiva. Así como la refactorización precisa de TDD como red de contención, TDD se apoya en la refactorización para eludir diseños triviales.
Así es como a veces ambas prácticas se citan en conjunto, y se confunden sus ventajas. Por ejemplo, cuando se menciona que TDD ayuda a reducir la complejidad del diseño conforme pasa el tiempo, en realidad debería adjudicarse este mérito a la refactorización que acompaña a TDD.
Sin embargo, dejemos establecido que la refactorización no necesita forzosamente del cumplimiento estricto del protocolo de TDD: sólo que haya pruebas (en lo posible automatizadas), y que éstas hayan sido escritas antes de refactorizar.

Ahora bien, cuando hablamos de mejorar el diseño sin cambiar el comportamiento observable, hay un problema. ¿Cómo logramos que corran las pruebas automáticas si la interfaz de las clases cambia al cambiar el diseño? Y en ese caso, ¿qué otra posibilidad tenemos que nos garantice el mantenimiento del comportamiento luego de una refactorización?
Bueno, ese es un problema, y la solución no es tan sencilla. Cuanto mayor sea el nivel de abstracción de las pruebas, o lo que es lo mismo, cuanto más cerca están las pruebas de ser requerimientos, más fácil va a ser introducir un cambio de diseño sin necesidad de cambiar las pruebas.
En este sentido, si queremos facilitar la refactorización, tendríamos que contar con pruebas a distintos niveles, en línea con lo que sugieren BDD, ATDD y STDD. Las pruebas de más alto nivel no deberían cambiar nunca si no es en correspondencia con cambios de requerimientos. Las pruebas de nivel medio podrían no cambiar, si trabajamos con un buen diseño orientado a objetos, que trabaje contra interfaces y no contra implementaciones. Y en algún punto, probablemente en las pruebas unitarias, debamos admitir cambios a las mismas.

El tema podría ir más allá. Si las pruebas deben estar materializadas en código, las mismas también pueden degradarse, con lo cual sería deseable automatizarlas. Pero, ¿cuál sería la red de contención de las pruebas ante una refactorización? De nuevo, las pruebas a distintos niveles de abstracción podrían ayudar. Como corolario, las pruebas más cercanas a los requerimientos no pueden ser refactorizadas en forma segura con la sola presencia de otras pruebas, que no existen, sino en base a acuerdos con los usuarios, clientes y otros interesados.

Debugging

Debugging o depuración es la técnica que se utiliza para encontrar las causas de los errores y corregirlos. Entendemos por errores a los comportamientos inesperados del sistema, encontrados en pruebas funcionales o con el sistema en producción.

La relación entre TDD y depuración viene del lado de la comodidad y la conveniencia: es más sencillo encontrar un error si existían pruebas automatizadas anteriores que si no las había. De hecho, si se siguió TDD en forma estricta, la única fuente de errores en el sistema sería la ausencia de algunas pruebas o la incorrección de algunas de las existentes.
Por eso existe la máxima de que, cuando encontremos un error en la aplicación, debemos escribir una prueba que evidencia ese error, y recién a partir de allí trabajar para corregirlo. Como decíamos, dado que el error podría provenir de un desliz en una prueba existente, es probable que la corrección del mismo lleve a que alguna prueba, de las antiguas, falle. Esto no es algo

malo, sino precisamente el efecto deseado: la corrección del error va a permitir que se corrija la prueba mal escrita.
Esto último nos lleva a un corolario innegable y al que no hemos prestado demasiada atención en este trabajo: el uso de TDD, aun en la más amplia de sus acepciones, no nos evita las pruebas funcionales realizadas por humanos. Al fin y al cabo, toda la seguridad que nos da TDD viene de las pruebas escritas en código que, como cualquier otra obra humana, puede contener fallas.

Integración continua

La integración continua es una práctica introducida por Beck en XP [1] y luego muy difundida a partir del trabajo de Fowler [39]. Consiste en automatizar y realizar, con una frecuencia al menos diaria, las tareas de compilación, corrida de las pruebas automatizadas y despliegue de la aplicación, todo en un proceso único, emitiendo reportes ante cada problema encontrado. Existen muchas herramientas, tanto libres como propietarias, más o menos sofisticadas, que facilitan esta práctica.

La relación entre TDD e integración continua viene de la propia definición de la segunda: para poder realizarla debimos haber desarrollado pruebas automatizadas previas a la integración.

Conclusiones

Siguiendo a Freeman y Pryce [18], podemos afirmar que "TDD es una técnica que combina pruebas, especificación y diseño en una única actividad holística", y que tiene entre sus objetivos iniciales el acortamiento de los ciclos de desarrollo.
Las ventajas que ofrece TDD, de nuevo según Freeman y Pryce [18], son:

- Clarifica los criterios de aceptación de cada requerimiento a implementar.
- Al hacer los componentes fáciles de probar, tiende a bajar el acoplamiento entre los mismos.
- Nos da una especificación ejecutable de lo que el código hace.
- Nos da una serie de pruebas de regresión completa.
- Facilita la detección de errores mientras el contexto está fresco en nuestras mentes.
- Nos dice cuándo debemos dejar de programar, disminuyendo el *gold-plating* y características innecesarias.

Ahora bien, hay muchas clases de TDD. UTDD está típicamente basado en diseño de clases, en el chequeo del estado de los objetos luego de un mensaje y se basa en un esquema *bottom-up*. BDD y STDD se basan en diseño integral, enfocándose en el chequeo de comportamiento y de en el desarrollo *top-down*. NDD se encuentra a mitad de camino de ambos enfoques.
Lo que ocurre es que hay dos maneras de ver la calidad: una interna, la que les sirve a los desarrolladores, y otra externa, la que perciben usuarios y clientes. UTDD y NDD apuntan a la calidad interna, mientras que BDD y STDD apuntan a la externa.
En definitiva, con objetivos distintos, existe un gradiente de técnicas que van desde un extremo al otro. Lo interesante sería poder converger hacia una especie de TDD integral, ya que existen muchos puntos en común entre las distintas clases. Al fin y al cabo, partiendo de UTDD, se ha abierto un abanico de facilidades y metodologías para pruebas de integración, para mejorar la especificación de requerimientos, para mejorar la claridad y la facilidad de mantenimiento y hasta para especificar y probar interfaces de usuario.

Una buena clasificación de tipos de pruebas y tipos de TDD más usuales es la que figura en la tabla que sigue[31]:

[31] Basada en la planteada por Steve Freeman y Nat Pryce en [18].

Tipo de prueba	Técnica de TDD relacionada	Pregunta que responde	Observaciones
Aceptación	STDD, ATDD, BDD	¿Funciona el sistema de acuerdo a los requerimientos?	No debe pecar por defecto ni por exceso respecto de los requerimientos
Integración	NDD	¿Cada parte del sistema interactúa correctamente con las demás partes?	Tiene que ver con el bajo acoplamiento y las interfaces entre módulos
Unitarias	UTDD	¿Las clases hacen lo que deben hacer y es conveniente trabajar con ellas?	Tiene que ver con la cohesión y el encapsulamiento

Bibliografía y referencias

[1] Kent Beck, “Extreme Programming Explained: Embrace Change”, Addison-Wesley Professional, 1999.
[2] Kent Beck, “Test Driven Development: By Example”, Addison-Wesley Professional, 2002.
[3] Dan North, “What´s in a Story”, http://blog.dannorth.net/whats-in-a-story/, como estaba en junio de 2011.
[4] Dan North, “Introducing BDD”, http://blog.dannorth.net/introducing-bdd/, como estaba en junio de 2011.
[5] Dan North, “Introducing BDD”, revista Better Software, marzo de 2006, citada en [4].
[6] Eric Evans, “Domain-Driven Design: Tackling Complexity in the Heart of Software”, Addison-Wesley Professional, 2003.
[7] Andrew Glover, “Is BDD TDD Done Right?”, http://thediscoblog.com/2007/08/28/is-bdd-tdd-done-right/, como estaba en junio de 2011.
[8] Bertrand Meyer, “Test or spec? Test and spec? Test from spec!”, http://www.eiffel.com/general/column/2004/september.html, como estaba en junio de 2011.
[9] Rick Mugridge y Ward Cunningham, “Fit for Developing Software: Framework for Integrated Tests”, Prentice Hall, 2005.
[10] Lasse Koskela, “Test Driven: TDD and Acceptance TDD for Java Developers”, Manning Publications, 2007.
[11] Simon Stewart (última actualización), “Page Objects”,
http://code.google.com/p/selenium/wiki/PageObjects, como estaba en junio de 2011.
[12] Frederick P. Brooks Jr., “The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition (2nd Edition)”, Addison-Wesley Professional, 1995.
[13] Gojko Adzic, “Bridging the Communication Gap. Specification by example and agile acceptance testing”, Neuri, 2009.
[14] Gerard Meszaros, “xUnit Test Patterns”, Addison-Wesley Professional, 2007. Estaba disponible también en http://xunitpatterns.com/ en junio de 2011.
[15] Martin Fowler, “Mocks Aren´t Stubs”,
http://martinfowler.com/articles/mocksArentStubs.html , como estaba en junio de 2011.
[16] Steve Freeman, Tim Mackinnon, Nat Pryce, Joe Walnes, “Mock Roles, Not Objects”, paper presentado en OOPSLA 2004, en Vancouver, Canadá.
[17] Deepak Alur, Dan Malks, John Crupi, “Core J2EE Patterns, Second Edition: Best Practices and Design Strategies, Prentice Hall, 2003.
[18] Steve Freeman, Nat Pryce, “Growing Object-Oriented Software, Guided by Tests”, Addison-Wesley Professional, 2010.
[19] Donald C. Gause, Gerald M. Weinberg, “Exploring Requirements: Quality Before Design”, Dorset House, 1989.
[20] Brian Marick, “Boundary Objects”,http://www.exampler.com/testing-com/writings/marick-boundary.pdf, como estaba en junio de 2011.
[21] Brian Marick, “Exploration Through Example”, http://www.exampler.com/old-blog/2003/08/21/, como estaba en junio de 2011.
[22] Steve Freeman, Nat Pryce, Tim Mackinnon, Joe Walnes, “Mock Roles, not Objects”, presentado en OOPSLA 2004, estaba también en http://www.jmock.org/oopsla2006.pdf en junio de 2011.
[23] Steve Freeman, Nat Pryce, “Evolving an Embedded Domain-Specific Language in Java”, presentado en OOPSLA 2006, estaba también en http://www.jmock.org/oopsla2006.pdf en junio de 2011.

[24] Rick Mugridge, "Managing Agile Project Requirements with Storytest-Driven Development", IEEE software 25, 68-75 (2008).
[25] Gerard Maszaros, Ralph Bohnet, Jennitta Andrea, "Agile Regression Testing Using Record & Playback", presentado en OOPSLA 2003.
[26] Juan Burella, Gustavo Rossi, Esteban Robles Luna, Julián Grigera, "Dealing with Navigation and Interaction Requirement Changes in a TDD-Based Web Engineering Approach", Proceedings of the The 11th International Conference on Agile Software Development, Springer Verlag, LNCS, 2010.
[27] Esteban Robles Luna, Irene Garrigós, Gustavo Rossi, "Capturing and Validating Personalization Requirements in Web Applications", Proceedings of the 1st Workshop on The Web and Requirements Engineering (WeRE 2010).
[28] Esteban Robles Luna, Juan Burella, Julián Grigera, Gustavo Rossi, "A Flexible Tool Suite for Change-Aware Test-Driven Development of Web Applications", Proceedings of the ACM/IEEE 32nd International Conference on Software Engineering (ICSE 2010).
[29] Adnan Causevic, Daniel Sundmark, Sasikumar Punnekkat, "Factors Limiting Industrial Adoption of Test Driven Development", International Conference on Software Testing, Verification and Validation, Berlin, Germany, estaba también en http://www.mrtc.mdh.se/index.php?choice=publications&id=2367 en junio de 2011.
[30] Nachiappan Naagappan, Michael Maximilien, Thirumalesh Bhat, Laurie Williams, "Realizing Quality Improvement Through Test Driven Development: Results and Experiences of Four Industrial Teams", Journal Empirical Software Engineering, Volume 13, Number 3, June, 2008, 289-302, Springer Science + Business Media, LLC 2008, estaba también en http://research.microsoft.com/en-us/groups/ese/nagappan_tdd.pdf en junio de 2011.
[31] Gerardo Canfora, Aniello Cimitile, Félix García, Mario Piattini, Corrado Aaron Visaggio, "Evaluating Advantages of Test Driven Deleopment: a Controlled Experiment with Professionals", ISESE 06 Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering.
[32] Dave Chaplin, "Contractual Test-Driven Development (DBC with TDD)", estaba en http://www.byte-vision.com/CtddArticle.aspx en junio de 2011.
[33] Pekka Abrahamsson, Antti Hanhineva, Juho Jäälinoja, "Improving Business Agility Through Techical Solutions: A Case Study on Test-Driven Development in Mobile Software Development", IFIP International Federation for Information Processing, 2005, Volume 180/2005, 227-243, DOI: 10.1007/0-387-25590-7_14, estaba también en http://www.agile-itea.org/public/deliverables/ITEA-AGILE-D2.7_v1.0.pdf en junio de 2011.
[34] Lei Zhang, Shunsuke Akifuji, Katsumi Kawai, Tsuyoshi Morioka, "Comparison Between Test Driven Development and Waterfall Development in a Small-Scale Project", Extreme Programming and Agile Processes in Software Engineering, 7th International Conference, XP 2006, Oulu, Finlandia, junio de 2006.
[35] Dave Chaplin, "Test First Programming", Tech Zone, 2001.
[36] Watts Humphrey, "Managing the Software Process", Addison Wesley, 1989.
[37] Shelly Park, Frank Maurer, "A Literature Review on Story Test Driven Development", Department of Computer Science, University of Calgary, Lecture Notes in Business Information Processing, 2010, Volume 48, Part 2, 208-213, DOI: 10.1007/978-3-642-13054-0_20, Springer Verlag, estaba también en http://ase.cpsc.ucalgary.ca/uploads/Publications/ParkMaurerXP2010.pdf en junio de 2011.
[38] William F. Opdyke, Ralph E. Johnson, "Refactoring: An Aid in Designing Application Frameworks and Evolving Object-Oriented Systems", Proceedings of the Symposium on Object Oriented Programming Emphasizing Practical Applications (SOOPPA), septiembre de 1990, ACM.
[39] Martin Fowler, "Continuous Integration", http://martinfowler.com/articles/continuousIntegration.html, como estaba en junio de 2011.