



# Máster en Cloud Apps

## Desarrollo y despliegue de aplicaciones en la nube


Curso académico 2022/2023

Trabajo de Fin de Máster

## ¿Está muerto TDD?

Autor: Daniel Sánchez González  
Tutor: Luis Fernández Muñoz



	Creative Commons Attribution-ShareAlike license <a href="https://creativecommons.org/licenses/by-sa/4.0/">https://creativecommons.org/licenses/by-sa/4.0/</a>
Introducción y objetivos.....	5
Resumen.....	6
TDD .....	7
Un poco de historia .....	7
Definición .....	7

¿Qué es? .....	8
Proceso .....	8
Testing .....	9
Tests unitarios .....	10
Proceso .....	10
Notas y excepciones .....	11
Recomendaciones .....	11
Estrategias de implementación .....	11
Patrones xUnit .....	12
Otros tipos de tests y consideraciones .....	13
Diseño .....	13
Historia y metáforas .....	14
Diseño incremental .....	14
Refactoring .....	15
Patrones de refactoring .....	15
Patrones .....	16
Lista de patrones .....	16
Adaptabilidad, aplicabilidad, métricas .....	17
Adaptabilidad .....	17
Aplicabilidad .....	17
Métricas .....	17
Escuela de Londres .....	18
Proceso del estilo de Londres .....	18
Londres VS Chicago .....	19
Testing .....	19
Diseño .....	20
Debate .....	22
Definición de TDD .....	22
Definiciones de facto y estándares .....	22
Polarización .....	24
Bienestar .....	24
Evangelización .....	25
Fundamentación .....	26
Testing .....	27
Test unitario .....	27
Grado de aislamiento .....	28
Otros aspectos del testing .....	29

TDD vs ATDD.....	29
Pirámide de tests.....	30
Departamento de QA .....	31
Foco en los tests .....	31
Documentación del sistema .....	32
Cobertura de pruebas .....	32
Impacto en el desarrollo .....	33
Diseño / arquitectura .....	34
Diseño/arquitectura previos VS emergente .....	34
Diseño/arquitectura incremental.....	34
Métodos y principios .....	35
Calidad VS daño.....	37
Testeabilidad y diseño .....	39
Adaptabilidad, métricas .....	39
Adaptabilidad .....	40
Ajuste en la cobertura y tipo de pruebas.....	40
Métricas.....	41
Escuela de Londres .....	41
Aprendizaje de TDD .....	43
Adquisición y evaluación del conocimiento .....	43
Evaluación del grado de madurez .....	43
Katas y coding dojos.....	44
Conclusiones y trabajo a futuro .....	45
Conclusiones.....	45
Definición de TDD.....	45
Testing .....	46
Diseño / Arquitectura.....	48
Adaptabilidad, aplicabilidad y métricas .....	49
Escuela de Londres.....	51
Trabajo futuro .....	51
Bibliografía y referencias.....	53

## Introducción y objetivos

Test-Driven Development sigue estando en el foco desde su aparición, generando mucha controversia entre defensores y detractores sobre la bondad de este método que no deja impassible a quien lo practica.

Durante su trayectoria en el contexto de las metodologías ágiles, algunos de sus seguidores predicaban que TDD hacía que la arquitectura del sistema emergiera, mientras que los detractores afirmaban que conducía a proyectos fracasados.

Este trabajo parte del que posiblemente sea el debate más polémico sobre este asunto: “Is TDD dead?”. El debate se originó en el 2014, cuando David Heinemeier Hansson, creador del framework de Ruby on Rails, hizo esta afirmación en la key note de la RailsConf, idea que reforzó después en una entrada en su blog que llevaba el título “TDD is dead. Long live testing”.

Unos días después, Martin Fowler contactó con él e intercambiaron opiniones sobre la charla, algo que también hizo con Kent Beck. Este último sugirió continuar con el debate y se celebraron cinco sesiones de Hangout en las que se discutió de diferentes asuntos relacionados con el TDD.

Con este punto de partida y tomando como referencia otros debates y ponencias similares, así como artículos y bibliografía, este trabajo pretende hacer una reflexión sobre este asunto, tratando de recoger los diferentes puntos de vista. Además, se hace una revisión de la definición de TDD, su uso, limitaciones y aportaciones, recogiendo una serie de conclusiones y trabajos futuros. Más allá del mero análisis, el trabajo pretende resaltar la importancia de las disciplinas de diseño y pruebas mediante el estudio de los debates.

El presente trabajo se sitúa en el contexto del “Máster en Cloud Apps. Desarrollo y despliegue de aplicaciones en la nube”, dentro del apartado de XP del módulo de calidad del software.

## Resumen

El hito fundacional de TDD es la publicación del libro "Test-Driven Development by Example" por Kent Beck en el 2002. Esta se produjo entre la primera edición de su libro "Extreme Programming Explained: Embrace Change", en el 2000, y la segunda edición, en el 2004. Ambos acontecimientos, el nacimiento de TDD y el de XP, tuvieron un gran impacto en la comunidad de desarrollo, que dura hasta nuestros días.

Actualmente TDD aparece relacionado con estilos arquitectónicos como el Hexagonal y los enfoques ubicuos con micro-servicios. A pesar del tiempo que ha transcurrido desde sus orígenes, la polémica sigue viva y candente entre sus defensores y detractores, aunque no siempre se encuentren argumentos fundamentados en ambos bandos.

Este trabajo pretende ahondar en estos debates, tratando de aportar luz sobre los mismos y el propio método. En el primer capítulo se establece un breve contexto histórico para proseguir con un resumen estructurado y objetivo del proceso de TDD partiendo del libro de Kent Beck "Test Driven Development: By Example". Este capítulo se desglosa en varios bloques que tienen que ver con el proceso de TDD, el enfoque de pruebas, el diseño, la adaptabilidad y aplicabilidad del proceso y un apartado dedicado al enfoque de Londres.

El segundo capítulo desgrana los argumentos de los distintos debates, apoyados con otros argumentos que se pueden encontrar en bibliografía relacionada o blogs. De nuevo se vuelven a revisar los grandes apartados, esta vez en clave de argumentos y contraargumentos, además de tratar de señalar algunas contradicciones que aparecen tanto en el libro como en los vídeos.

El apartado de conclusiones revisa las discusiones y ofrece la reflexión que persigue el trabajo, tratando de aportar un nuevo enfoque a las discusiones.

# TDD

## Un poco de historia

Se puede considerar que el hito fundacional de TDD es la publicación del libro "Test-Driven Development by Example" por Kent Beck en el 2002 [0]. Esta se produjo entre la primera edición de su libro "Extreme Programming Explained: Embrace Change", en el 2000, y la segunda edición, en el 2004. Ambos acontecimientos, el nacimiento de TDD y el de XP, tuvieron un gran impacto en la comunidad de desarrollo, que dura hasta nuestros días.

Durante los años previos a este momento, la trayectoria de Kent Beck había estado relacionada, entre otras cosas, con patrones, buenas prácticas y la automatización de pruebas unitarias en el lenguaje Smalltalk, publicando varios libros sobre el tema. Esta actividad contribuyó decisivamente al nacimiento de JUnit.

Según Martin Fowler, Kent Beck y Erich Gamma coincidieron en el vuelo de Zúrich a Atlanta para asistir a la conferencia OOPSLA de 1997 y lo aprovecharon para hacer pair programming, creando la primera versión de JUnit. Para Fowler, JUnit despegó como un cohete y fue esencial para apoyar el creciente movimiento de XP y TDD. [1][2]

Cabe destacar que en la década de los 90 y principios del milenio se produjeron otros acontecimientos relacionados con TDD que fueron también relevantes para el mismo:

- 1993: Bill Opdyke habla en un foro para el personal de AT&T Bell Labs y NCR sobre refactoring. Ese mismo año se publica el libro de Martin Fowler sobre el tema, en el que contribuyen John Brant, Opdyke, Don Roberts y el propio Kent Beck. [3]
- 1994: Se publica "Design Patterns: Elements of Reusable Object-Oriented Software", por Erich Gamma, Richard Helm, Ralph Johnson, and John M Vlissides.
- 2001: Nace el manifiesto ágil
- 2004: Se acuña el acrónimo SOLID cuando Michael Feathers sugiere a Robert C. Martin nombrar su trabajo de principios de diseño con ese término [4]
- 2005: Comienzan a extenderse las code katas, propulsadas por la idea de Dave Thomas [5]
- 2006: Dan North publica "Introducing BDD" como una alternativa al TDD mal entendido [6]

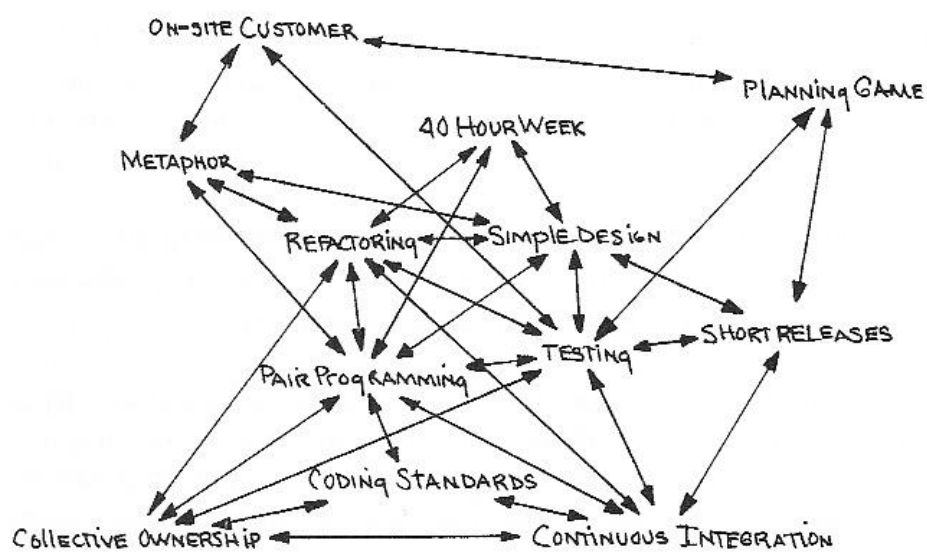
La primera década del siglo XXI acaba con la publicación del libro "Growing Object-Oriented Software, Guided by Tests" en el 2009, que se podría considerar como el momento clave de la escuela TDD de Londres, aunque ese estilo llevaba unos años propulsándose desde el London's Extreme Tuesday Club (XTC), donde los expertos del lugar practicaban con XP y TDD. [7][8]

## Definición

Esta definición parte del libro de Kent Beck, del que se han extraído sus comentarios para organizar y explicar, de la manera más literal posible, su definición de TDD

TDD forma parte de las prácticas fundamentales de XP, recogida en el libro de "Extreme Programming" como Test-First Programming. Kent Beck afirma que TDD y XP se complementan mutuamente de la siguiente manera:

- El diseño simple, codificando solo lo necesario y eliminando la duplicación. Ofrece el diseño necesario para tener la arquitectura perfecta para el sistema actual, que hace que escribir los tests sea más sencillo
- La regla de eliminar la duplicación es otra forma de decir "refactoring"
- Continuous delivery: TDD mejora el tiempo medio entre fallos (MTBF) del sistema, con lo que se puede añadir código en producción más frecuentemente sin interrumpir el sistema
- Integración continua: Las pruebas son un recurso excelente que permite integrar más a menudo. El ciclo puede ser de 15 a 30 minutos en lugar de las 1 a 2 horas a las que se suele aspirar. Esto puede ser parte de la clave para tener equipos más grandes de programadores en la misma base de código.
- La programación por pares potencia a TDD, ofreciendo una visión fresca en los momentos de mayor cansancio



¿Qué es?

Es un conjunto de técnicas que cualquier programador puede seguir y que fomentan diseños sencillos y pruebas que inspiran confianza, lo que define una forma predecible de desarrollar. Pretende romper con la vieja fórmula por fases, que es una danza complicada de análisis, diseño lógico, físico, implementación, pruebas, revisión, integración y despliegue. También quiere acabar con la manera de trabajar en la que se hacen los tests tras programar todo el código.

Pero TDD no es una técnica de pruebas, sino de análisis y diseño, en realidad es una técnica para estructurar todas las actividades de desarrollo. Su objetivo es elaborar código limpio que funcione y hacer cambios con confianza, dejando la búsqueda de la perfección en un lugar secundario

Proceso

TDD se basa en dos reglas básicas que dan lugar al ciclo, un mantra ejecutado en tres fases:

Reglas básicas	Fases del ciclo	
Escribir nuevo código sólo si una prueba automatizada ha fallado	Rojo	Se trata de escribir una pequeña prueba que no funcione, y quizás ni siquiera compile



	Verde	En esta fase hay que hacer que la prueba funcione rápidamente, cometiendo los pecados que sean necesarios en el proceso
Eliminar la duplicación	Refactorizar	Es el momento de eliminar toda la duplicación creada para lograr que la prueba inicial funcionara

## Beneficios

Según su autor, este proceso de desarrollo tiene una larga lista de beneficios, entre los que se encuentran:

- Ayuda a los equipos a construir de forma productiva sistemas poco acoplados, altamente cohesionados, con bajas tasas de defectos y perfiles de mantenimiento de bajo coste. Parte del efecto se debe a la reducción de defectos de manera temprana. La reducción de defectos tiene muchos efectos psicológicos y sociales secundarios. Los programadores se relajan, los equipos desarrollan confianza y los clientes esperan con impaciencia las nuevas versiones.
- Reduce drásticamente la densidad de defectos del código y deja muy claro el objeto de trabajo para todos los implicados. Escribir sólo el código que exigen las pruebas fallidas también tiene implicaciones sociales. El control de calidad puede pasar de ser reactivo a proactivo, los jefes de proyecto podrán hacer estimaciones lo bastante precisas como para implicar a los clientes reales en el desarrollo diario, los temas de las conversaciones técnicas se aclaran lo suficiente como para que los ingenieros de software puedan trabajar en colaboración minuto a minuto, se puede disponer de software con nuevas funciones cada día, lo que dará lugar a nuevas relaciones comerciales con los clientes.
- La calidad del código se mantiene con el tiempo. A medida que las pruebas se acumulan y mejoran, se gana confianza en el comportamiento del sistema. A medida que se refina el diseño, más y más cambios son posibles.
- Acorta el bucle de retroalimentación en las decisiones de diseño. El bucle de las decisiones de diseño va desde la idea de diseño hasta el primer ejemplo, una prueba que encarna esa idea. En lugar de diseñar y luego esperar semanas o meses, la retroalimentación llega en segundos o minutos.
- Da la oportunidad de aprender todas las lecciones que el código tiene que enseñar.
- Mejora la vida de los usuarios de su software
- Permite el apoyo entre compañeros
- Sienta bien escribirlo
- Evita tener que preocuparse por un largo rastro de errores.

## Testing

TDD tiene un fuerte enfoque en las pruebas. Probar significa "evaluar" y también "procedimiento que conduce a la aceptación o al rechazo". Los tests dan un feedback rápido ante experimentos de refactorización, en lugar de tener que pensar si será la estrategia adecuada. Los desarrolladores deben

escribir sus propias pruebas, porque no pueden esperar 20 veces al día a que otra persona la escriba por ellos y el entorno debe proporcionar una respuesta rápida a los pequeños cambios.

Kent Beck utiliza la metáfora del pozo y la polea: la programación sería la manivela para sacar el agua. Las pruebas son el mecanismo dentado que ayuda a elevar un cubo más pesado con menos esfuerzo. Cuando el cubo es menos pesado, hará falta un mecanismo con menos dientes.

Los tests automáticos ofrecen la oportunidad de elegir el nivel de miedo. Se trata de responder a la pregunta: ¿cuál es el conjunto de pruebas que, una vez superadas, demostrarán la presencia de código del que estamos seguros de que realizará las operaciones correctamente?.

Un efecto colateral de TDD, es que cambia el rol de los profesionales de testing de la supervisión a la mejora de la comunicación entre stakeholders y desarrolladores.

## Tests unitarios

Los tests de TDD son unitarios, pero no casan muy bien en la definición aceptada de unit tests, tal y como afirma Kent Beck. Él propone transformar una preocupación por algún aspecto del código en un test, usando criterios que tengan sentido en el dominio del negocio, no en el de los objetos (del lenguaje de programación). El test debe comprobar comportamiento, no implementación. De alguna manera, se trata de explicar el sistema mediante tests. Esto, además, permite difundir el uso de TDD y sus ventajas en la organización.

Los tests deben trabajar de manera aislada, sin afectarse unos a otros, y tienen que ejecutarse de manera rápida y frecuente. Los tests siguen un patrón común AAA (identificado por Bill Wake en 2001): Arrange-Act-Assert [9]. El paso de preparación suele ser el mismo entre tests. Hay dos restricciones que entran en conflicto, rendimiento, por lo que se necesitaría crear los objetos iniciales una sola vez, y aislamiento, con lo que unos tests no afectarían a otros en su ejecución. Se debe evitar que unos tests rompan otros o que tengan que ser ejecutados en cierto orden. El rendimiento es la razón habitual para que las pruebas compartan datos. Hay que trabajar, y a veces mucho, para dividir el problema en pequeñas dimensiones, de modo que configurar el entorno para cada prueba sea fácil y rápido. Aislar las pruebas anima a componer soluciones a partir de muchos objetos muy cohesionados y poco acoplados.

## Proceso

Kent Beck propone las siguientes pautas para escribir los tests:

- **Test List:** Antes de comenzar, escribir la lista de todos los tests que habrá que escribir. En primer lugar, poner en la lista ejemplos de cada operación que se necesita implementar. A continuación, para aquellas operaciones que aún no existen, poner la versión nula de esa operación en la lista. Por último, enumerar todas las refactorizaciones que se sospecha que habrá que hacer para tener un código limpio al final de la sesión. Tener una lista cerca con los objetivos alcanzables en unas pocas horas y otra lista, con un alcance semanal o mensual, colgada en la pared. Cuando surja algo nuevo, decidir rápidamente si pertenece a la lista de "ahora" o a la de "más tarde", o si realmente es necesario hacerlo.

Escribir los tests en masa no funciona, porque puede requerir rehacer la signatura de los métodos, lo cual requerirá trabajo adicional. Además, si varios tests se rompen, se estará lejos de obtener

un nuevo verde. La forma pura de TDD hace que nunca se esté a más de un cambio de obtener el verde.

- Starter Test: El primer test tiene que ser sencillo y con pocas operaciones. Si ya se han realizado sistemas similares, coger uno con algunas operaciones más (una o dos). Coger un test que empiece en un nivel superior, más parecido a un test de aplicación
- One Step Test: El orden de implementación de los tests importa. Hay que buscar una prueba que enseñe algo y que se vea factible de resolver. Si esa prueba funciona y falla la siguiente, se retrocede dos pasos.
- Cantidad de tests para tener feedback: No escribir tests innecesarios. TDD es pragmático
- Se pueden borrar tests que estén cubiertos en otros sitios o que se repiten, salvo que se reduzca la confianza al hacerlo o que el segundo test describa un escenario diferente al primero.

### *Notas y excepciones*

Hay que considerar que, si se añade código nuevo en el refactoring, no se añaden tests, dado que se ha llegado a la refactorización con el código ejercitado.

En algún momento en el libro, durante un error en un refactoring, Kent Beck aboga por continuar, añadiendo otro test para aislar el problema. (Capítulo 14 - Change). También comenta que si se programa con alguien que no ve el objetivo o la lógica se empieza a complicar, se puede escribir una prueba separada. Por último, añade un código para depurar sin pruebas asociadas, dado que es un código de depuración con escasa probabilidad de fallar.

### *Recomendaciones*

Estas son algunas de las recomendaciones que se hacen a la hora de escribir las pruebas:

- Assert First: Escribir las aserciones primero. Un sistema se empieza a construir con historias sobre cómo será el sistema terminado. La funcionalidad se empieza a escribir mediante las pruebas que se quieren pasar cuando el código esté listo. Las pruebas se empiezan a elaborar con las afirmaciones que se pasarán cuando el test esté funcionando. Cuando se escribe una prueba, se resuelven varios problemas a la vez: ¿A dónde pertenece la funcionalidad?, ¿Cómo deben llamarse los nombres?, ¿Cómo vas a comprobar si la respuesta es correcta?, ¿Cuál es la respuesta correcta?, ¿Qué otras pruebas sugieren esta prueba?

Escribiendo primero la aserción, se resuelven los problemas "¿Cuál es la respuesta correcta?" y "¿Cómo voy a comprobarlo?"

- Test Data: Usar datos que sean fáciles de comprender y con significado. Usar el conjunto de datos necesario (el mínimo) para pasar el test. No usar la misma constante para dos cosas.
- Evident Data: Presentar la intención de los datos incluyendo los resultados esperados y reales de la prueba e intentar que la relación sea evidente. Se escriben pruebas para un lector, no sólo para el ordenador. Evident Data es una excepción a la regla de números mágicos.

### *Estrategias de implementación*

A la hora de abordar la escritura de los tests, conviene tener en cuenta estas estrategias de implementación:

- Child Test: Dividir un test grande en otros más pequeños para después volver al test mayor
- Mock Object: Evita acceso a recursos costosos (BD, por ejemplo). También facilita la lectura del test.
- Self Shunt: Chequea en el test que se ha llamado al objeto esperado desde el SUT, usando el propio test, que implementa una interfaz. Mejora la legibilidad del test. El siguiente ejemplo correspondería a un test para una clase Scanner que lee un código de barras y muestra la información del producto escaneado en una pantalla LCD:

```

// act like a display
public class ScannerTest extends TestCase implements Display {
    public ScannerTest(String name) {
        super(name);
    }

    public void testScan() {
        // pass self as a display
        Scanner scanner = new Scanner(this);
        // scan calls displayItem on its display
        scanner.scan();
        assertEquals(new Item("Cornflakes"), lastItem);
    }

    // impl. of Display.displayItem ()
    void displayItem(Item item) {
        lastItem = item;
    }

    private Item lastItem;
}

```

- Log String: Almacenar un log en un string para comprobar si las secuencias de mensajes son invocadas de forma correcta. Especialmente útiles cuando se implementa el patrón Observer y se espera que las notificaciones lleguen en cierto orden. Funciona bien coordinado con Self Shunt.
- Crash Test Dummy: Simular excepciones para testear el código de error
- Broken Test: Dejar una sesión de programación con el último test fallando. De esa manera, al día siguiente se recordará más fácilmente por dónde se dejó la sesión
- Clean Check-in: Dejar una sesión de programación en un equipo con todos los tests corriendo. De esa manera suite de integración funcionará correctamente al empezar.

## Patrones xUnit

xUnit se ha traducido a más de 30 lenguajes de programación en el momento de la edición del libro de TDD. Es la pieza clave que permite escribir los test, por lo que tiene una importancia fundamental. En palabras de Martin Fowler, "nunca en los anales de la ingeniería de software tantos debieron tanto a tan pocas líneas de código".

Se deben considerar las siguientes prácticas en su uso:

- Assertion: Expresión booleana que automatiza la evaluación de si el código funciona. Hay que ser específico en el resultado final. El valor esperado debe estar al inicio de la expresión. Acceder a los objetos como cajas negras y no a sus atributos, puede resultar complicado. Si se accede a los atributos directamente, esto sería caja blanca y podría apuntar a un mal diseño.
- Fixture: Se llama fixture al banco de pruebas que se utiliza en electrónica para probar un componente. Esta técnica consistiría en transformar las variables locales en cada test en variables de instancia, que serán inicializadas en el método de setUp. Normalmente, cada fixture se usará para probar un conjunto de tests relacionados. No hay una relación directa entre las clases de test y del

modelo. En algunas ocasiones, pocas, un fixture sirve para probar varias clases. En otras ocasiones, varios fixtures son necesarios para probar una sola clase. En general, se tiende a tener un número similar de clases de test que del modelo, pero no porque haya una relación uno a uno. Todos los métodos de tests que comparten un mismo fixture, deben estar en la misma clase y a la inversa.

- External Fixture: Implementar `tearDown` para liberar los recursos externos en el fixture.
- Test Method: Comenzar los métodos de test con la palabra "test". El nombre del método debe sugerir el test que se implementa. Los métodos de tests deben ser legibles y con un código directo. El objetivo es escribir el menor test que represente el progreso real hacia la meta. Se puede comentar con líneas que describan el test a implementar y que sirvan de documentación
- Exception Test: Para testear las excepciones, capturarlas e ignorarlas y fallar si no se lanzan
- All Tests: Hacer una suite con todas las suites, una por paquete y una que agregue todos los paquetes

## Otros tipos de tests y consideraciones

Existen otras consideraciones y tipos de tests más allá de las pruebas unitarias utilizadas en el ciclo de TDD. A continuación, se enumeran estas reflexiones:

- Regression Test: Escribir tests lo más sencillo posibles para los defectos. Esto da una oportunidad de aprender sobre la forma de testear que se ha seguido, ya que hay defectos que se escaparon. También se podrían escribir a alto nivel, dando la oportunidad a los usuarios del sistema para clarificar lo que esperaban. Si hay que refactorizar el sistema para aislar el defecto, esto puede indicar un mal diseño.
- Los tests de implementación, que no comprueban comportamiento, pueden servir para experimentar alternativas. En caso de que no funcionen, hay que borrarlos
- Another Test: Añadir tests para discusiones técnicas que surjan, de forma que se queden registradas y no distraigan de la implementación en curso
- Los tests de rendimiento, estrés y usabilidad no pueden ser sustituidos por TDD
- Los test a nivel de aplicación (ATDD) tienen problemas técnicos y sociales. El problema técnico con ATDD es tener que crear el fixture para una funcionalidad que todavía no existe. El social es que escribir tests es una nueva responsabilidad para los usuarios, lo que requiere esfuerzos coordinados en el ciclo de desarrollo para escribir una prueba con antelación. Con TDD, la técnica está bajo el control del desarrollador. Un último problema es la longitud del ciclo entre test y feedback. Si un cliente escribe un test y 10 días después funciona, se permanecerá en el rojo la mayor parte del tiempo.
- Learning Test: Usar tests para software producido externamente la primera vez que se vaya a utilizar una nueva funcionalidad de un paquete. Este test permite evaluar si un nuevo método de API se comporta como esperamos. También permite evaluar si ha habido cambios en sucesivas versiones del paquete. Respecto a este último punto, en otra parte del código comenta que no hay que testear código de otros, a no ser que se desconfíe, en cuyo caso se escribirá un código para documentar un error del código de terceros.

## Diseño

Según Kent Beck, debemos diseñar de forma orgánica, con código en ejecución que proporcione retroalimentación entre las decisiones. Nuestros diseños deben constar de muchos componentes muy

cohesionados y poco acoplados para facilitar las pruebas. Se trata de obtener código limpio que funciona. Primero se hace que funcione y luego se hace limpio. Es lo contrario que el desarrollo dirigido por la arquitectura, en el que se trata de resolver primero que el código sea limpio y luego se integra en el diseño mientras se hace que funcione.

No hay una fórmula mágica que inspire las ideas de diseño. TDD no puede garantizar que tendremos la inspiración adecuada en el momento justo. En cualquier caso, la confianza que ofrecen los tests y un código cuidadosamente producido, nos prepara para su aplicación cuando lleguen.

Es mejor preocuparse por el código que por el diseño a largo plazo. Al refactorizar y desarrollar solo lo necesario, el código es mejor y se satisface el principio open / closed. El sistema implementado en un momento determinado expresa únicamente lo que se necesita en ese momento, sin ninguna suposición futura.

## Historia y metáforas

El proceso de diseño funciona como una narración. Cuando escribimos una prueba, imaginamos la interfaz perfecta para nuestra operación. Nos estamos contando una historia sobre cómo se verá la operación desde fuera. Hay que pensar en cómo debería aparecer en el código la operación que se tiene en mente. Se inventa la interfaz que se desee y se incluye en la historia todos los elementos que se imaginen necesarios para calcular las respuestas correctas.

El uso de la metáfora también es útil para la definición del sistema y ayuda a poner una mirada nueva que permita obtener las mejores ideas de diseño. Hay que pensar en las posibles metáforas del sistema y escribir el test en función de ella.

## Diseño incremental

A lo largo del libro, se insiste continuamente en los pequeños pasos y en el diseño incremental. Si un problema es demasiado complicado, hay que meter pasos más sencillos mediante nuevos tests.

Si se han utilizado pruebas que represente un caso simple de todo el cálculo del sistema, puede que parezca que está escrito de arriba abajo. También podría parecer escrito de abajo a arriba, porque se empieza con piezas pequeñas y se van agregando más y más. En realidad, ni el enfoque descendente ni el ascendente describen el proceso de forma útil. En primer lugar, una metáfora vertical es una visualización simplista de cómo los programas cambian con el tiempo. El crecimiento implica una especie de bucle de retroalimentación, en el que el entorno afecta al programa y viceversa. En segundo lugar, si necesitamos una dirección en nuestra metáfora, la descripción de known-to-unknown es útil. Known-to-unknown implica que tenemos algunos conocimientos y experiencia en los que basarnos, y que esperamos aprender en el curso del desarrollo. Si juntamos ambas cosas, tenemos programas que crecen de lo conocido a lo desconocido.

El paso al verde, haciendo que la prueba funcione, permite ir descubriendo la implementación que se quiere realizar poco a poco. Las tácticas para lograr este descubrimiento son las siguientes:

- Fake It ('Til You Make It): Hacer una primera implementación utilizando constantes. Luego, transformarla gradualmente en una expresión usando variables. Es mejor que se ejecute el test de manera incorrecta que no lo haga, dado que enseña algo sobre el sistema, tiene un efecto psicológico y limita el alcance. Después se elimina la duplicación

- **Triangulate:** Abstraer solo cuando haya dos o más ejemplos. Una vez se triangula, se puede borrar una de las aserciones redundantes. Utilizar solo cuando haya dudas sobre la abstracción
- **Obvious Implementation:** Solo cuando se está seguro de qué implementar. Si se utiliza continuamente, puede tratarse de autoexigencia: hacer dos cosas a la vez, código limpio y que funciona, puede ser demasiado. Se trata de una velocidad superior de implementación
- **One to Many:** Si hay operaciones con colecciones, primero se implementa una solución sin ellas y luego se añade. Así, si se quiere escribir una operación para sumar un array de números, primero se implementa el método con un solo valor y luego se añade el array. Una vez se tiene el caso más simple, se puede cambiar la implementación sin afectar al test y viceversa. Esto último sería un ejemplo de cambio aislado

## Refactoring

En cuanto al refactoring, el autor considera que permite cambiar el diseño del sistema de forma radical y en pequeños pasos. Se trata de refactorizar para eliminar la duplicación. La dependencia es el problema clave en el desarrollo de software a todas las escalas y la duplicación el síntoma.

Eliminar la duplicación en los programas elimina la dependencia. Por eso aparece como la segunda regla en TDD. Al eliminar la duplicación antes de pasar a la siguiente prueba, maximizamos nuestra oportunidad de ser capaces de conseguir que la siguiente prueba se ejecute con un solo cambio.

## Patrones de refactoring

Estos son los patrones de refactoring que nombra en el libro:

- **Reconciliar diferencias:** Unificar dos partes del código similares de manera gradual, hasta que sean idénticas
- **Aislar cambios:** Transformar un método u objeto con varias partes aislando el trozo a cambiar. Algunas posibilidades para lograrlo: Extraer método, extraer objeto o Extract Method, Extract Object o Method Object
- **Migrar datos:** Moverse de una representación duplicando los datos temporalmente. Por ejemplo, cambiar una variable a un nuevo formato
- **Extraer método:** Hacer un método largo y complicado más fácil de entender, sacando una parte de este en otro método separado. Los candidatos suelen ser cuerpos de un bucle o ramas de condicionales. Útil para entender código complicado
- **Método en línea:** Reemplazar la invocación de un método por el método en sí para simplificar un flujo de control que se ha vuelto demasiado complicado.
- **Extraer interfaz:** Permite introducir una segunda implementación de una operación creando una interfaz que contiene la operación compartida
- **Mover método:** Mover un método a la clase que le corresponde e invocarlo. Si el objeto original tiene variables inicializadas, es mejor descartar esta refactorización
- **Method Object:** Consiste en representar un método complicado con varios parámetros y variables locales en un objeto externo. El objeto tendrá, como variables de instancia, las variables locales del método. También tendrá un método run(), cuyo cuerpo será el mismo que el del método original.
- **Añadir parámetro:** Añade un parámetro a un método



- **Parámetro de método a parámetro de constructor:** Mueve el parámetro de un método o métodos al constructor. De esta manera, se simplifica la API de los métodos

## Patrones

En cuanto a los patrones de diseño, Kent Beck considera que el libro "Design Patterns" tiene un sesgo, dirigiendo el diseño como una actividad por fases, en lugar de como refactorización. El diseño en TDD requiere un cambio en la forma de mirar a los patrones. Estos son una forma de interiorizar y mecanizar soluciones, lo que deja más espacio para otros aspectos del diseño y análisis. TDD puede verse como un método de implementación de patrones de diseño, haciendo que se piense en lo que debe hacer el sistema y dejando que el diseño aflore más tarde

### *Lista de patrones*

Los patrones de diseño que se presentan en el libro de TDD no pretenden hacerlos comprensivos, es suficiente con verlos mediante ejemplos.

- **Command:** Representar una invocación de computación como un objeto que es invocado, en lugar de como un mensaje. Hay que rellenar sus parámetros e invocar un método genérico, del estilo `run()`.
- **Value Object:** Objetos ampliamente compartidos, para los que la identidad no es importante. Su estado nunca cambia, una vez creados. Una operación sobre el objeto siempre devolverá un objeto nuevo. Tiene que implementar `equals`
- **Null Object:** Objeto que representa un caso especial
- **Template Method:** Representa la parte invariante de una secuencia de computación, mientras que las variaciones se encuentran en las implementaciones. Secuencias clásicas son las del tipo: entrada-proceso-salida, envío de mensaje / recepción de respuesta, lectura de comando y retorno de resultado. Hay que determinar si se da una implementación por defecto. Es mejor detectar este tipo de patrones mediante la experiencia, en lugar de intentar diseñarlo así desde el principio. Cuando se identifiquen dos variantes de la misma secuencia en dos subclases, se extraen las partes diferentes y, lo que queda, es el `template method`, que se puede mover a una clase superior y eliminar la duplicación.
- **Pluggable Object:** Representa variaciones mediante la invocación de otro objeto con dos o más implementaciones de la misma interfaz.
- **Pluggable Selector:** Almacena el nombre del método y, dinámicamente mediante reflexión, invoca a dicho método.
- **Factory Method:** Crea el objeto en un método, en lugar de usando un constructor. El aspecto negativo es su indirección, por lo que debe usarse solo cuando se necesite esa flexibilidad.
- **Imposter:** Introduce una variación mediante un objeto con el mismo protocolo, pero diferente implementación.
- **Composite:** Objeto cuyo comportamiento es la composición de una lista de otros objetos.
- **Collecting Parameter:** Se añade un parámetro a un método que recopilará el resultado de la operación, dispersa por varios objetos
- **Singleton:** No usar variables globales. Tomarse el tiempo necesario para pensar en el diseño en su lugar



## Adaptabilidad, aplicabilidad, métricas

### Adaptabilidad

La clave de la adaptación del proceso de TDD es el tamaño de los pasos. A continuación, se dan unas cuantas ideas al respecto:

- Pequeños pasos: Cuando haya bloqueo trabajando en grandes ideas de diseño, identificar algo más pequeño y trabajar en ello
- Tamaño de los pasos: Al principio, cuando se refactorice, se deben hacer pequeños pasos. La refactorización manual es propensa al error, y cuantos más errores se cometan y detecten más tarde, menos probabilidades habrá de refactorizar. Una vez que se hayan hecho varias refactorizaciones a mano en pequeños pasos, experimentar omitiendo algunos de los pasos. La refactorización automatizada acelera enormemente la refactorización.
- Los (pequeños) pasos se pueden adaptar y hacer más grandes, según se considere (como si se cambiara de marchas)
- La habilidad de controlar la distancia entre tests permite acelerar o decelerar el proceso.
- A medida que se avanza hacia la periferia del sistema, hacia partes que no cambian tan a menudo, las pruebas pueden ser más puntuales y el diseño peor, sin que interfiera en la confianza generada.

### Aplicabilidad

- Aplicabilidad en sistemas grandes: Su experiencia fue de 4 años, 40 personas / año, 250000 líneas de código funcional y otras tantas de tests en Smalltalk. Había 4000 tests, que se ejecutaban en 20 minutos. La suite completa se ejecutaba varias veces en el día.
- Legacy code: En legacy code, el código no es muy testeable. Hay que limitar el alcance de los cambios, dejando las partes que no demandan cambios tal cual están. Se puede obtener feedback de otras formas, en lugar de con tests, como trabajando con un compañero cuidadosamente o usando tests de sistemas que, aunque no son adecuados, dan mayor confianza. En cualquier caso, hay que escribir el test que se desee o necesite antes de hacer el refactoring.
- Posibles limitaciones de TDD:
  - GUI.
  - Objetos distribuidos automáticamente.
  - Desarrollo del esquema de BD
  - Código de terceros o código generado por herramientas externas.
  - Desarrollo de un compilador / interprete de BNF hasta una implementación de calidad de producción

### Métricas

- Es probable que se termine con aproximadamente el mismo número de líneas de código de prueba que de código modelo al implementar con TDD. Para que TDD tenga sentido desde el punto de vista económico, hay que ser capaz de escribir el doble de líneas al día que antes, o escribir la mitad de las líneas para la misma funcionalidad. Hay que medir y ver qué efecto tiene TDD en la práctica y asegurarse de tener en cuenta el tiempo de depuración, integración y explicación en las métricas.

- Métricas de uso de Junit: Número de ejecuciones en distintos intervalos de tiempo, desde unos pocos segundos, hasta 10 minutos o más. Esta métrica permite ver que la mayoría de las veces se ejecutan mucho los tests en cortos periodos de tiempo y pocas veces se espera más de 10 minutos para ejecutar.
- Métricas de código: Hay un número similar de líneas y de métodos en los tests y en el código. El uso de fixtures en los tests no altera mucho la relación anterior. La complejidad ciclomática en los tests es baja porque no hay ramificaciones. En el código se mantiene baja debido a que se ha sustituido el control de flujo por el polimorfismo. El número de líneas por método en los tests es superior que en código si no se aplica fixtures.
- Métrica de cobertura: No es una medida suficiente de calidad de los tests. En cualquier caso, con TDD supone un 100% de sentencia
- Métricas de defect insertion: Se cambia una línea de código para ver si el test falla. Jester reportó solo 1 línea de código en el hashCode de una clase.
- Si se considera como medida de calidad el número de pruebas que chequean diferentes aspectos de un programa dividido por el número de aspectos que necesitan pruebas (complejidad de la lógica), una forma de mejorar la cobertura sería aumentar los tests para probar el código. La otra sería refactorizar a partir de un conjunto mínimo de tests para hacer el código más testeable y eficiente, reduciendo los casos de pruebas.
- Determinar la calidad de los tests: Atributos que sugieren que el código de test tiene un mal diseño: código de setup largo (puede que por objetos grandes que necesitan ser divididos), duplicación de setup (demasiados objetos interrelacionados), tests con ejecuciones largas (suites de más de 10 minutos terminan recortadas), tests frágiles (tests que fallan de manera inesperada apuntan a que una parte de la aplicación está afectando a otra)

## Escuela de Londres

Tras la definición del modelo de TDD por Kent Beck en su libro, surgieron otros enfoques, como el de la escuela de Londres, que nació durante los años de práctica en la comunidad de Extreme Programming en dicha ciudad. Aunque el movimiento comienza poco después de la publicación del libro de TDD de Kent Beck, culmina con la edición del libro "Growing Object-Oriented Software, Guided By Tests", de Steve Freeman y Nat Pryce en el año 2009.

Debido a sus enfoques, ambas escuelas reciben distintos nombres, que se listan a continuación:

London	Chicago / Detroit
Mockist	Classicist / classical / traditional
Outside-in	Inside-out
Top-down	Bottom-up
Interaction testing / behavioral	State based testing
White-box testing	Black-box testing

## Proceso del estilo de Londres

Normalmente se empieza con una prueba de aceptación que verifica si la función en su conjunto funciona. La prueba de aceptación también sirve de guía para la implementación. Con una prueba de aceptación fallida, que informa de por qué la función aún no está completa, se empiezan a escribir pruebas unitarias. La primera clase a probar es la clase que gestiona una petición

externa (un controlador, un receptor de cola, un controlador de eventos, el punto de entrada de un componente, etc.).

Se hacen algunas suposiciones sobre qué tipo de colaboradores necesitará la clase bajo prueba. A continuación, se escriben pruebas que verifican la colaboración entre la clase bajo prueba y sus colaboradores.

Los colaboradores se identifican en función de todo lo que la clase sometida a prueba necesita hacer cuando se invoca su método público. Los nombres de los colaboradores y los métodos deben proceder del lenguaje del dominio (sustantivos y verbos).

Una vez probada una clase, se elige el primer colaborador (que se creó sin implementación) y se prueba su comportamiento, siguiendo el mismo enfoque que se utilizó para la clase anterior.

### Londres VS Chicago

El libro de Kent Beck se puede interpretar como un enfoque claro en el modelo de objetos de dominio (aunque, como se citó en el apartado de diseño, Kent Beck prefiere nombrarlo como un proceso de known-to-unknown).

Por su parte, el libro de Freeman podría verse con un enfoque en el proceso de desarrollo de una aplicación, más allá del modelo de objetos de dominio, dado que contempla cuestiones como la arquitectura hexagonal. Además, plantea un doble ciclo de tests, donde el ciclo interno coincide con el TDD de Kent Beck y el externo corresponde a una prueba de aceptación

### Testing

El enfoque de Londres utiliza pruebas de caja blanca mediante el uso intensivo de mocks. Desde este enfoque prima la comunicación entre objetos, en lugar de la comprobación de su estado. A menudo está ligado a que una "unidad" debe ser una clase o una función. Así que, por definición, una "prueba unitaria" no debería probar más de una clase y probablemente tampoco más de una llamada a una función. Mediante este enfoque, se puede ahorrar el trabajo con los objetos reales (creación y mantenimiento) sustituyéndolos por dobles de test. El inconveniente es que las pruebas están acopladas al código de producción: cualquier reelaboración/refactorización requiere reescribir las pruebas. Además, un mal diseño puede llevar a una explosión de mocks.

En el caso de la escuela de Chicago, el tipo de pruebas es de caja negra, con lo que se comprueba el estado y el comportamiento del SUT. En este caso, se evita el uso de mocks, con lo que el concepto de unidad aislada puede incluir varios métodos o clases.

Dado que esta escuela empieza en los niveles más internos de la arquitectura, esto tiende a producir pruebas desacopladas de la implementación, lo que permite una refactorización agresiva, además de una cobertura más cohesiva.

Entre los inconvenientes está que con el enfoque clasicista las pruebas tienden a crecer con configuraciones y aserciones. Además, la complejidad reside en los dobles de prueba. Las pruebas dobles en memoria para sustituir dependencias pueden ser muy complejas.

En la escuela de Londres, el diseño comienza en la fase de rojo, mientras se escriben los tests, y se refina durante la fase de refactorización. Cuando se escribe una prueba unitaria que falla, hay que decidir cuáles van a ser los colaboradores de la clase bajo prueba y cómo se va a comunicar esta clase con ellos (API pública) para cumplir el requisito de negocio (prueba).

Las pruebas de aceptación dirigen la arquitectura, dado que se empieza con la capa más externa del sistema, y las necesidades de esta capa guían el resto de la implementación y la arquitectura hasta las clases más profundas del dominio. Todo el sistema se construye de forma que sirva a las necesidades de los clientes y, por tanto, a su propósito.

Este método de diseño adopta el punto de vista del cliente. Se diseña su API pública (la interfaz externa que la clase publica para que sea consumida por un tercero), lo que permite ocultar los detalles de implementación detrás de esta API pública cuando llegue el momento de implementar esta clase. Primero se piensa en qué tipo de comportamiento tendría que proporcionar esta clase para cumplir los requisitos de negocio (pruebas).

En esta escuela no se expone ningún estado sólo con fines de prueba, al contrario de la escuela de Chicago. También, las refactorizaciones son más pequeñas que esta última.

En el caso de Chicago, el diseño es de abajo a arriba, desarrollando primero las unidades que más tarde necesitará el sistema en general. Esto produce una alta cohesión: a medida que se escriben progresivamente pruebas muy específicas a pruebas más generalizadas, el código de producción resultante se vuelve altamente cohesivo, lo que favorece la calidad del código.

La escuela de Chicago facilita la exploración, cuando se conocen la entrada y la salida deseada, pero no se tiene clara la implementación.

El enfoque clasicista el diseño ocurre en el estado de refactorización y retrasa las decisiones de diseño lo máximo posible. Primero se hace funcionar (de rojo a verde) y se piensa en cómo mejorarlo, haciendo la mayor parte del diseño en el paso de refactorización.

Es un enfoque que prima el modelo del dominio, que se considera en muchas ocasiones la parte más valiosa del software. Esto permite una validación temprana de los requisitos de negocio, lo que da la oportunidad de descubrir incoherencias, aclarar los requisitos con más detalles y validar las suposiciones. Obtener un feedback rápido sobre áreas importantes del sistema es una estrategia de mitigación de riesgos.

La desventaja del enfoque clasicista es que se aplaza la API y, por tanto, se puede producir una lógica muy alejada de las necesidades reales del usuario. Otro efecto colateral de esto es que cuando se intenta conectar el backend, puede no satisfacer exactamente todas las necesidades, con lo que habrá que escribir algún código adicional para conectar ambos lados, cambiar el backend o comprometer la usabilidad definida para reutilizar el código.

Las interfaces de usuario tienen fama de ser volátiles, mientras que las reglas de negocio tienden a cambiar mucho menos, una vez definidas. Empezar primero con las reglas de negocio permite a los desarrolladores de backend avanzar mientras los desarrolladores de front-end iteran sobre el diseño de la interfaz de usuario. Por el contrario, si el front-end o las API se hacen después del modelo de dominio (backend), sólo se obtendrá feedback cuando toda la función esté hecha.

Una manera de entregar software de forma incremental es haciendo vertical slicing. Esto puede resultar más difícil cuando se parte del modelo de dominio, ya que no se está escribiendo código para satisfacer una necesidad externa específica. El vertical slicing debe estar guiado por requisitos externos.

Otra de las consecuencias de usar el enfoque inside-out, es que la refactorización suele ser mayor que en la escuela londinense. Durante la fase de refactorización, el SUT puede crecer hasta abarcar varias clases. A medida que las clases emergen, pueden romper pruebas no relacionadas, ya que las pruebas utilizan su implementación real en lugar de un mock.

Por último, los profesionales inexpertos se saltan a menudo el paso de refactorización (mejora del diseño), lo que lleva a un ciclo que se parece más a rojo – verde – rojo – verde – ... – rojo – verde - refactorización masiva.

## Debate

En este apartado se exponen ideas antagónicas alrededor de TDD. Los puntos de discusión surgen principalmente de la serie de debates “Is TDD dead?” entre David Heinemeier Hansson (DHH a partir de ahora), Kent Beck y Martin Fowler, a los que se han añadido otras fuentes para ampliarlo.

### Definición de TDD

La definición inicial de TDD ha sido modificada y complementada a lo largo de los años por parte de la comunidad. Además, el debate de si TDD es bueno o no, se ha polarizado con el paso del tiempo.

### Definiciones de facto y estándares

En el debate entre Jim Coplien y Uncle Bob [23], este último explica sus tres leyes de TDD. Uncle Bob definió estas leyes en el 2005, tres años después de la aparición del libro de Kent Beck.

Martin Fowler también dio una definición en su blog en el año 2005 [24]. A continuación, vemos las diferencias:

Kent Beck	Martin Fowler	Uncle Bob
Escribir nuevo código sólo si una prueba automatizada ha fallado	Escribe una prueba para la siguiente funcionalidad que quieras añadir	No se puede escribir código de producción a menos que sea para hacer pasar una prueba unitaria fallida
	Escribe el código funcional hasta que pase la prueba	No se pueden escribir más pruebas unitarias que las que sean suficientes para fallar, y los fallos de compilación son fallos. Así que no se puede escribir mucho de la prueba unitaria antes de escribir código de producción
Eliminar la duplicación	Refactoriza tanto el código nuevo como el antiguo para que esté bien estructurado	No se puede escribir más código de producción que el que sea suficiente para superar la prueba unitaria fallida

Las leyes de Uncle Bob se encierran en un ciclo que puede durar 30 segundos, en el que se escriben pruebas unitarias y código de producción simultáneamente, anticipando las pruebas entre 30 segundos y un minuto respecto al código de producción. Llama la atención que la propuesta pasa por escribir el código de producción y la prueba casi concurrentemente. Tampoco deja claro si en el paso tres se puede refactorizar.

Por su parte, Martin Fowler, subraya la importancia del refactoring, incidiendo en que se puede refactorizar todo el código y no sólo las últimas líneas añadidas.

DHH sostiene en sus intervenciones que la definición de TDD se ha ido deformando, poniendo cada uno su sello, por lo que habría que hacer un reset y comenzar de nuevo, con la experiencia acumulada y, tal vez en unos 10 años, volver a llegar al mismo sitio.

En su charla, Ian Cooper [25] comenta que se está haciendo TDD mal y necesita revisarse la idea original de Kent Beck. El libro es un referente y es necesario leerlo, dado que encierra mucha sabiduría y Kent llevaba ya años practicando TDD cuando lo escribió. En él, afirma Ian Cooper, se resuelven muchos problemas con los que se había encontrado al empezar a practicar TDD. Al practicar TDD y leer el libro, comenzó a comprender lo que Kent Beck quería decir.

Habla de las ideas que subyacen en los pasos red-green. El sentido del "rojo" es demostrar que el test falla en ausencia de la solución. En palabras de Bertrand Meyer: "El principio de test-first, al igual que la observación de Dijkstra sobre el papel de las pruebas, está relacionado con el concepto de falsabilidad citado al principio de este capítulo. Del mismo modo que un principio interesante debe ser falsable, una función de software interesante debe tener una prueba asociada cuyo fallo demuestre que el producto no cumple la función. (Un caso de prueba con éxito, o cualquier número de ellos, no demuestran nada, del mismo modo que ningún conjunto de ejemplos exitosos puede demostrar la validez de una teoría o un principio). Escribir la prueba ayuda a aclarar de qué trata la función." [63]

El sentido del "verde" es entender cómo solucionar el problema. Se trata de entender el comportamiento del sistema y de dotarlo de ese comportamiento. Kent Beck - dice el ponente - opina que no se pueden hacer dos cosas al mismo tiempo: entender la solución al problema y hacer ingeniería con el código. Si se intentan hacer ambas cosas, entonces se aplicará sobreingeniería, más allá de los requisitos, o se llegará a una parálisis por análisis.

Textualmente, Kent Beck comenta en su libro: "Las distintas fases tienen propósitos diferentes. Exigen distintos estilos de solución, distintos puntos de vista estéticos. Las tres primeras fases (escribir el test, hacer que compile, ejecutar para comprobar que falla) tienen que ser rápidas, para que lleguemos a un estado conocido con la nueva funcionalidad. Podemos cometer cualquier cantidad de pecados para llegar allí, porque la velocidad triunfa sobre el diseño sólo por ese breve momento". [0]

Ian Cooper también comenta que el objetivo de la fase de refactor, tal y como la concibió Kent Beck, es eliminar la duplicidad. Cita a Martin Fowler y la primera edición del libro de refactoring: "¿Qué es la refactorización? La refactorización es el proceso de cambiar un sistema de software de tal manera que no altere el comportamiento externo del código, pero mejore su estructura interna. Es una forma disciplinada de limpiar el código que minimiza las posibilidades de introducir errores. En esencia, cuando se refactoriza se mejora el diseño del código después de haberlo escrito." [3]

Comenta que Joshua Kerievsky, autor del libro Refactoring to patterns, afirma que en el refactoring es cuando se puede identificar si los patrones son los adecuados o no, dado que se tiene una idea clara de la solución. Los patrones se aplican mal, dice Kerievsky, así que el momento ideal para usar patrones es en el refactoring, porque ya se sabe la solución, aunque sea mala. Ian Cooper afirma que en el libro original, Kent Beck ya identifica unos patrones que se podrían usar para refactorizar. Similar a la línea de Kent Beck, en la que afirma que el libro clásico de patrones "parece tener un sutil sesgo hacia el diseño como fase. Desde luego, no hace ningún guiño a la refactorización como actividad de diseño." y ofrece una visión más práctica y superficial, Ian Cooper apoya esta idea a través de Joshua Kerievsky.

Cooper concluye esta reelaboración de las leyes básicas de TDD diciendo que en la fase de refactor no se añaden nuevos tests. Los tests son una red de seguridad que permiten refactorizar, garantizando que el comportamiento no cambie durante el refactoring. Incide en que si se crea una nueva clase no se añaden tests, dado que la regresión está garantizada por las pruebas, siempre que no sean clases públicas (que expongan una interfaz o API hacia el exterior).

Ciertamente Kent Beck afirma en el capítulo 10 del libro “Preferiríamos no escribir una prueba cuando tenemos una barra roja.”, aunque acaba apostillando “A veces sigo adelante y escribo una prueba en rojo, pero no mientras los niños están despiertos.”

Otro de los aspectos que reelabora y matiza Cooper es la adaptación y velocidad de TDD, que Kent Beck expresa con el símil de las marchas de un coche. Menciona “Gears”, lo que parece ser un método con esa forma de desarrollar a diferentes velocidades usando TDD. A veces hay que seguir los pasos de TDD de manera estricta, porque la solución no está clara, y otras se puede ir directo al diseño y la solución sin pasar por refactoring. Si en algún momento es necesario apoyarse en tests adicionales para descubrir una solución, es conveniente eliminar los tests cuando se complete dicha solución, ya que están revelando detalles de la implementación que no es conveniente tener.

En una referencia a “Gears” [26] puede leerse “La primera marcha es para arrancar y construir contexto; la segunda es para mejorar el diseño y aplicar patrones avanzados; la tercera se utiliza para construir funcionalidad, siguiendo patrones y prácticas arquitectónicas existentes del sistema. La marcha atrás se utiliza para volver a la verde después de un refactorizado o para ayudar a adoptar un nuevo enfoque para encontrar la siguiente prueba.

El núcleo de TDD es un ciclo Rojo-Verde-Refactor con una colección de prácticas y principios. TDD Gears es un modelo para explicar cómo encaja todo. Hay tres marchas de avance y una marcha atrás, porque a veces es necesario probar un enfoque diferente).”

Más allá de esta definición de “Gears”, no parece haber una referencia oficial a este supuesto método o modelo. Kent Beck no menciona expresamente un método de adaptación, tan solo la metáfora del cambio de marchas de manera dispersa por el libro, por lo que este parece ser uno de esos añadidos de la comunidad a TDD.

Otra de las incorporaciones a la definición inicial son las katas. Emily Bache las comenta en su charla como una forma de acelerar el aprendizaje de TDD [27]. Parece que la idea de las katas fue introducida por Dave Thomas sobre el año 2003. [28][29][30]

## Polarización

En esta sección se presentan algunos temas del debate en los que las posturas parecen basarse en ideas preconcebidas y sesgos, lo que contribuye a la polarización de las posturas.

### *Bienestar*

Según Kent Beck, la forma de trabajar de TDD reduce la ansiedad que se produce cuando uno se enfrenta a grandes problemas. La compara con la cultura anterior que llegó a vivir, en la que no había TDD o tests automatizados y, por tanto, no se obtenía feedback del desarrollo.

Contrasta la forma de trabajar en la que tiene que leer logs para depurar y entender el sistema, del que no hay una especificación clara, con otra en la que puede acotar una funcionalidad y seguir una secuencia de tests para llegar a un resultado de manera inductiva. Esta última es la manera en que se siente más cómodo y fluye hacia la solución, mientras que la otra le produce mayor estrés.

Por su parte, DHH comenta que con el ciclo de red-green-refactor, que es más un mandato que una opción, no obtiene las cotas de bienestar que se supone que ofrece TDD. Una de las dificultades que



encuentra es a la hora de escribir la prueba primero, lo que encuentra anti-natural (escribir la hipótesis primero para luego rellenarla).

Afirma que el ciclo rojo-verde-refactor de TDD es adictivo y hay gente enganchada a ese método. Cree que se debe a los sentimientos que se producen al liberar mucha dopamina y la confianza que genera, lo que acaba provocando el abuso del método y la deformación de la arquitectura.

Kent Beck responde a esto diciendo que es el peor narcotraficante del planeta.

### *Evangelización*

Para Kent Beck, algunos de los problemas principales que resuelve TDD y por lo que no renuncia a abandonarlo son la confianza que genera, tener pequeños incrementos de trabajo a la vista, la guía que suponen los tests para el desarrollo, especialmente cuando no se ve claro y que permite atacar a los problemas poco a poco. Alude a que TDD aporta medición del progreso, la seguridad / confianza y la productividad y que el desarrollo de software ha evolucionado en los últimos 10 / 15 años.

DHH opina en cambio que TDD se ha vendido bien, pero hay que ver las partes oscuras. El surgimiento de XP, TDD, Ruby y Rails trajo un cambio de paradigma a la comunidad que ahora se da por supuesto, aunque todavía queda mucho que hacer. Ensalza el regression test y self-testing, afirmando que en esto hay más consenso, porque ofrece confianza, independientemente de si la prueba se escribe primero o después, el tiempo que lleva la ejecución, etc.

Sin embargo, comenta cómo la industria empuja a usar TDD, pero hay mucha gente a la que no le va bien y que no se atreve a comentarlo abiertamente o se siente culpable. En el sector hubo personas que no querían matizar que TDD es un buen enfoque para gente experimentada, y lanzaron un discurso generalista en el que, si no se usaba TDD no se era un buen desarrollador. TDD puede ser bueno para algunas cosas, no para aplicaciones web, pero tal vez para otras sí. El problema es que hay que abrir ese debate y parece que hay gente que no está dispuesta.

Emily Bache comenta en su charla la forma en la que Uncle Bob hace ponencias, un tanto afectado y apuntando con el dedo a la audiencia, preguntando si la gente es realmente profesional y escribe tests. Esto hace que el discurso suene a imposición (aunque ella opina que está encarnando a un personaje y de ahí las puestas en escena mediáticas).

Comenta que tal vez ese discurso está dirigido a los novatos, pero no a gente como DHH, que tal vez lo ha personalizado erróneamente. Opina que en la difusión de TDD hay muchos consejos para iniciados, pero no para expertos. Cree que la transmisión debe ser una cuestión de actitud, en la que hay que mostrar a la gente lo grande que puede llegar a ser la práctica de TDD para invitarla a participar.

Martin Fowler es proclive a presentar las técnicas con las razones que subyacen sobre cuándo usarlas y cuándo no. Habla de la victoria que supuso la etiqueta "ágil", pero no su contenido. Pasa lo mismo con TDD, se habla de ello, pero no se hace, lo que califica como "dispersión semántica". Lo que ocurre con TDD es similar a la aplicación que se hacía de agile 10 años atrás, como algo que había que justificar de cara al cliente. Ahora es algo que se impone como un proceso, y los problemas que eso conlleva son otros.

Hace referencia a Dave Thomas (PragDave), que 2014 escribió "Agile is dead (long live agility)" [31] [32]. En ese artículo, Dave Thomas narra cómo se subvirtió la idea original de agile.

Jim Coplien dice que hay gente que hace lo correcto en los proyectos y a eso le llama TDD. Luego, otra gente ve tutoriales de TDD en los que se afirma que "la arquitectura sólo viene de las pruebas". Uncle Bob habla de ciertos sesgos erróneos que tuvo la comunidad ágil alrededor del año 99, que consideraba que la arquitectura era irrelevante, dado que emergería de las pruebas, las historias de usuario y las iteraciones. También afirma que Kent Beck siempre ha hablado de metáforas.

En la línea de Jim Coplien, DHH opina que la comunidad mete muchas cosas en el mismo saco, lo llaman TDD y consiste en: self testing code, red-green refactor, mocks, etc. Las buenas ideas sencillas se acaban corrompiendo añadiéndoles cosas. A TDD o agile les ha pasado eso y actualmente cada persona entiende una cosa distinta.

Él experimenta lo mismo con Rails y ve que la gente ahora hace cosas que no se esperaba y se pregunta por qué lo hacen así, siendo un camino arduo y difícil. Él se basa en los principios fundamentales, pero luego la comunidad va mutándolo hacia otra cosa deformada. Opina que es necesario un reinicio. Utiliza la metáfora de morir como un héroe o vivir mucho tiempo como un villano para referirse a TDD. Como con el lenguaje PHP, en el que, si no tienes cuidado, acabas haciendo mal código, con TDD pasa lo mismo. Aplicarlo a toda una web MVC es más difícil que escribir PHP limpio.

Sostiene que la comunidad de TDD es muy "moralistas" sobre el uso de la técnica para llegar al resultado de self testing code al que, en realidad, todo el mundo quiere llegar. Según opina David, la comunidad de TDD defiende que el principal retorno es el diseño obtenido y la idea de que un diseño solo se puede mejorar introduciendo más tests. En realidad, él ha visto justo lo contrario, diseños llenos de mock en aras de la testeabilidad. Opina que hay cosas más importantes que si el sistema está probado y es si el sistema está claro y es comprensible.

Habla de la "fe basada en TDD". Es la creencia de que TDD, aunque no se tenga evidencia cierta, conducirá al diseño correcto. En su caso, ese bucle llevó a sentirse mal con él mismo cuando no pudo alcanzar el objetivo. Acaba afirmando que, para él, TDD como principio de programación basado en la fe, está muerto. Considera que, para un gran número de casos, simplemente no hay forma de que TDD lleve a un mejor diseño. Más bien, conducirá al lugar equivocado.

Hay una gran dificultad en tener una base de código open source compartida con ejemplos reales, y eso estaría bien. Eso provoca debates filosóficos sin contexto y presentaciones poco fundadas, lo que lleva a mayores desencuentros. Es difícil hacer científica la evaluación de una técnica, principio o proceso, así que el debate se queda circunscrito a las opiniones y experiencias de cada uno.

Kent Beck afirma que hay demasiadas variables para poder verificar experimentos, pero que una mentalidad científica ayuda. No demuestra nada, pero que se pueden recopilar datos y compararlos. Él no reclama ninguna especie de ley universal y pone como ejemplo las ciencias políticas, que asemeja con las ciencias de la computación. Comenta algunos de los problemas principales que resuelve TDD y por lo que no renuncia a abandonarlo: gestión del miedo, desarrollo incremental y guiado por las pruebas y el uso de "divide y vencerás".

Cuando XP empezó a captar la atención de la gente, James Rumbaugh le advirtió que en 10 años no sería capaz de reconocer lo que la gente haría con ello, porque ya estaría fuera de su control. Afirma que su advertencia fue correcta y no la escuchó. De haberlo hecho, tal vez habría creado algo muy distinto.

James Rumbaugh es uno de los padres de RUP. En la revisión del libro "Kent Beck's Guide to Better Smalltalk: A Sorted Collection", James Rumbaugh dice esto: "Kent Beck puede condensar más experiencia práctica en una máxima concisa que la mayoría de los escritores en una página entera." [33].

Cabe destacar el énfasis que pone Kent Beck en la sencillez y la metáfora, tal y como se puede observar en la amplia bibliografía del libro XP, en el que reseña libros en el apartado de filosofía que hablan sobre la estética, la simplicidad y la metáfora.

Martin Fowler afirma que el éxito hace que mute la definición, por una cuestión evolutiva y de adaptación. Se trata de llegar a un acuerdo. Dice que Chris Morris [34] se hace la pregunta de si se trata de un mal uso de TDD o si es inherente a él. Es complicado determinar cuál es el original y cuáles las variaciones (difusión semántica). Lo único que puede hacerse es seguir debatiendo cuáles son las bases, las buenas lecciones y reenfocar el asunto. No estamos en un mundo científico y no se pueden hacer experimentos replicables. Este problema va a suceder siempre.

Concluye diciendo que hay puntos de consenso, como los tests de regresión y el código auto-testeado y también dice que TDD puede ser útil en algunos contextos. Hay diferencias en cuanto a los contextos en los que se puede aplicar, y es algo difícil de determinar, dada la cantidad de variaciones. También comenta la importancia de profundizar en estos temas, dado que no se puede traer cualquier técnica al equipo, porque hace falta usarla y excederse en el uso para encontrar lo que funciona. Solo desarrollando y con la experiencia, se puede alcanzar esto, dado que no es una ciencia exacta.

## Testing

En este capítulo se abordarán los distintos puntos de vista sobre las pruebas en el contexto de TDD.

### Test unitario

Ian Cooper comenta que lo que dice Kent Beck es que los tests prueban comportamiento, no detalles de implementación. Es incorrecto escribir un test para probar un nuevo método de clase, dado que eso es un detalle de implementación. Lo correcto es enfocarse en los nuevos requisitos recibidos y reflejar lo que se quiere que haga el software: "Quiero añadir una cantidad a la cuenta bancaria de este cliente".

Los tests se tienen que centrar en la API pública y su contrato. Con esto no se refiere a una API en REST, sino a la parte pública de un módulo que se ofrece al mundo. Eso es estable y los detalles de implementación son los que pueden variar. Pone como ejemplo que se puede cambiar la forma en la que se implementó un agregado (en terminología DDD) por completo, dado que los tests garantizarán que el comportamiento siga siendo el mismo. El estilo Given-When-Then orienta a probar las historias.

El SUT no es una clase, tal y como se piensa erróneamente a veces. El SUT es un módulo, que se prueba como una caja negra. Al tomar el SUT como una clase concreta, hay que hacer mocks de manera intensiva, lo que lleva a conocer los detalles de implementación de una clase.

Como cita Vladimir Khorikov en su libro "Unit Testing principles, practices and patterns": "Por otra parte, las pruebas de caja blanca suelen ser frágiles, ya que tienden a acoplarse estrechamente a la implementación del código sometido a prueba. Estas pruebas producen muchos falsos positivos [...]".

Además, a menudo no puede establecerse una correspondencia con un comportamiento significativo del negocio, lo cual es una señal de que son frágiles y no añaden mucho valor”. [35]

Algunas veces viene bien hacer tests para entender los detalles de la implementación y poder refactorizar. Luego hay que eliminar esos tests.

Dice que Dan North, experto en agile y creador del BDD (alrededor del 2006) [36], afirmaba que la gente se confundía al ver la palabra Tests en TDD, por lo que acuñó BDD (Behaviour Driven Development), para que la gente lo tuviera más claro.

La gente malinterpretó lo que Kent quiso decir, porque hablaba con la terminología clásica. Kent Beck hablaba de "developer tests". Muestra cómo Kent Beck habla también sobre comportamiento del sistema. El comportamiento es el requisito en TDD, tal y como los describió Kent Beck originalmente: “¿Qué comportamiento necesitaremos para producir el informe revisado? Dicho de otro modo, ¿qué conjunto de pruebas, una vez superadas, demostrará la presencia de código que estamos seguros de que calculará el informe correctamente? Necesitamos poder sumar importes en dos monedas diferentes y convertir el resultado dado un conjunto de tipos de cambio. Tenemos que ser capaces de multiplicar una cantidad (precio por acción) por un número (número de acciones) y recibir una cantidad.”

Contrastando con este enfoque expuesto, en el libro de TDD aparecen varios casos que resultan llamativos. En el capítulo 2, añade una entrada a la lista de test llamada "Dollar side effects?". La descripción, más que un comportamiento del negocio parece una historia técnica. También se puede apreciar cómo en el capítulo 3 arranca comentando la cuestión de la igualdad, la implementación del hash y el concepto de Value Object, muy alejado del comportamiento que se espera de un Dollar desde el punto de vista del negocio. También, en el capítulo 13 añade una entrada en la lista de tests para devolver un objeto Money como resultado de sumar  $5 + 5$ .

Para DHH hay una máxima en TDD que dice que el código que es difícil de probar de forma aislada está mal diseñado, con lo que se fuerza un diseño lleno de indirecciones y sobrecargado con el objetivo de hacer el código más testeable y conseguir tests más rápidos. Considera que la propuesta de Jim Weirich, para aplicar una arquitectura hexagonal en Rails, es errónea [58]. Todo se basa en los tests, que sean rápidos y que no interacciones con la base de datos, en lugar de centrarse en el buen diseño y la claridad del sistema. Para DHH, los controladores deberían probarse con tests de integración, no unitarios, pero la pirámide de testing prescribe el nivel unitario. En cuanto a los tests de vista, deberían de tratarse con test de sistemas (end-to-end).

Propone probar el modelo más intrincado mediante tests unitarios, pero sin tratar de separarlos de la base de datos, a través de fixtures. Los controllers se deberían probar con tests de integración, sin sacar la lógica de sesión, chequeo de permisos o vistas auxiliares. Los tests de vista deberían probarse con tests de sistema / navegador. Si se tiene un modelo simple y una vista compleja, se deben intensificar los tests de sistema y hacer más ligeros los de modelo.

También comenta que la noción de que no se puede escribir una sola línea de código de producción antes de tener un test, implica que cada línea de código debe ser probada, lo que considera sobretesting.

#### *Grado de aislamiento*

En cuanto a ese tema del grado de aislamiento, Ian Cooper sostiene que todos los tests deben correr en una única test suite sin que unos impacten a los otros. Cuando Kent Beck habla de nivel de

aislamiento en los test, se refiere a esto, pero la gente cree erróneamente que se refiere a aislar la clase bajo test, por lo que hacen múltiples mocks para lograr ese nivel de aislamiento.

El error es que la gente utiliza definiciones del testing clásico, pero el unit testing consistiría en probar un módulo. Los tests de integración probarían la interacción entre módulos y los de sistemas, todos los módulos juntos. La gente, en cambio, tendió a interpretar que la unidad era la clase y, por tanto, evitaba ciertas interacciones en los tests (BD, sistema de ficheros, etc). Las razones de Kent Beck para evitar estas interacciones son debido a que los tests comparten elementos, lo que hace que no se ejecuten de forma aislada o que sean más lentos. Se podría utilizar una BD en memoria y, si no hay efectos colaterales, sería perfectamente válido usar la base de datos o el sistema de ficheros en un unit test.

Martin Fowler sostiene que TDD no indica el grado de aislamiento de las pruebas respecto a los sistemas externos. Hubo debate en la comunidad de XP sobre este tema, entre defensores y detractores a la hora de aislar los tests y de definir cómo eran los tests unitarios, lo que produjo múltiples definiciones de Unit Tests. Él opina que hay distintas escuelas de pensamiento.

DHH se pregunta cuál es la definición de test unitario y el grado de aislamiento adecuado: no debe tener colaboradores, acceder a la base de datos o a ficheros y tiene que ser rápido. ¿Tiene sentido que un test unitario no tenga colaboradores? El TDD que suele ver está lleno de mocks. Según ese enfoque, se debe tomar cada parte de un MVC y trocearlo lo suficiente hasta que cada unidad pueda ser aislada de esa forma tan estricta. Él ha visto una serie de aplicaciones que utilizan el patrón Command para extraer la acción y envuelven el repositorio y el negocio para poder testear.

## Otros aspectos del testing

Durante los debates de “Is TDD dead?” aparecen otros aspectos del testing, como las pruebas de aceptación, la pirámide de tests o la labor del departamento de QA

## *TDD vs ATDD*

DHH arremete contra Cucumber y la enorme infraestructura de código que es necesario desplegar para intentar involucrar a la gente de negocio, lo que considera una quimera. Opina que BDD, aplicando el enfoque outside-in, conduce a un uso excesivo de mocks, aunque se decanta más por el objetivo de BDD.

Ian Cooper habla sobre ATDD (Fitness, Cucumber, etc) y lo costosos que son esos tests, que se pasan la mayor parte del tiempo en rojo. Los stakeholders no suelen mirar esa pipeline y los desarrolladores nunca tienen claro si el test falló o si todavía no estaba implementado. Muchas veces se establecen dinámicas acusatorias sobre quién ha roto la pipeline y hay un gran trabajo para rescatarla, hasta que finalmente se ignora o abandona.

Kent Beck expresa una idea similar sobre ATDD en el capítulo 32 del libro. Curiosamente, se desdice con esta introducción para el libro "ATDD by example" de Markus Gärtner [37]: " Uno de los puntos débiles del TDD tal y como se describió originalmente es que puede convertirse en una técnica de programador utilizada para satisfacer las necesidades de un programador. Algunos programadores adoptan una visión más amplia del TDD, cambiando fácilmente entre niveles de abstracción para sus pruebas. Sin embargo, con ATDD no hay ambigüedad: se trata de una técnica para mejorar la comunicación con personas para las que los lenguajes de programación son extraños. La calidad de nuestras relaciones, y la comunicación que subyace a esas relaciones, fomenta el desarrollo eficaz de

software. ATDD puede utilizarse para dar un paso en la dirección de una comunicación más clara, y ATDD por ejemplo es una introducción completa y accesible. ")

James Shore, creador de FIT (Framework for Integrated Test), escribió en el 2010 una entrada en su blog en la que expresa lo siguiente: "Estos dos problemas -que los clientes no participan, lo que elimina el propósito de las pruebas de aceptación, y que crean una importante carga de mantenimiento- significan que las pruebas de aceptación no merecen la pena. Ya no las utilizo ni las recomiendo. " [38]

Para Ian Cooper lo que Gherkin resolvía en realidad era la forma incorrecta de hacer los unit tests que tenía la comunidad, por lo que hay que rescatar la idea original de Dan North: solventar el problema con TDD, porque se entendía mal.

### *Pirámide de tests*

Para Ian Coope, las pruebas unitarias se enfocan en módulos aislados, los test de integración la interacción entre ellos y los de sistema, el conjunto al completo. Explica la inversión de la pirámide de tests que sucede en las organizaciones (test de UI manuales, frágiles, lentos, que produce fricciones en el equipo).

Opina que en la arquitectura hexagonal, el lugar ideal donde apuntar los tests unitarios serían los puertos, que encapsulan la lógica y actúan como puertas o fachadas. Los tests de integración probarían los adapters, que contiene el código mínimo para interactuar con el sistema exterior, de modo que satisfaga la interfaz. No se prueba lógica de negocio, dado que esto se prueba con los tests unitarios. [64] [65]

Emily Bache habla de la pirámide de testing ágil que introdujo Mike Cohn [39]. Muestra unas cifras de alguien de ThoughtWorks que afirmaba que el 5% debía ser UI tests, el 25% service tests y el 70% unit tests. Cita unas palabras de DHH de su post "Test-induced design damage" del 2014: "Si tienes una capa de modelo muy simple, pero tu UI es compleja, entonces deberías realizar muchas pruebas de sistema y pocas pruebas de modelo." [40]

Ella argumenta que los unit tests se ejecutan rápido y encuentran problemas de regresión en puntos específicos del código. También tienen un efecto en el diseño, debido a la posibilidad de refactorización. Pero, cuando se habla de self testing code, se busca algo más general. Cita a Martin Fowler, que habla de esto en el hangout y en este artículo: "El código es auto-comprobable cuando se puede ejecutar una serie de pruebas automatizadas en la base de código y confiar en que, si se superan las pruebas, el código está libre de defectos sustanciales." [41]. Fowler no comenta nada de qué tipo de tests tienen que ser. Dice que todos los tests de la pirámide son necesarios, aunque los de UI e integración son más costosos de escribir y más frágiles.

Emily introduce un tipo de test como posible solución a esta discusión: Approval testing, una técnica útil para test de requisitos y para código legacy. Este tipo de tests prepara una entrada, obtiene la salida y si se juzga correcta, esta será la aserción para los tests. Comenta que los unit tests no detectarían ciertos problemas de requisitos, dado que se centran en otro tipo de aserciones más específicas.

Habla de la herramienta <http://texttest.org>, que permite marcar diferencias entre salidas aprobadas y chequeadas previamente de manera textual, que son fáciles de actualizar. En la web de Approval Testing dice: "Las aserciones de los unit tests pueden ser difíciles de utilizar. Los approval tests



simplifican esto tomando una instantánea de los resultados, y confirmando que no han cambiado." [42]

#### *Departamento de QA*

DHH habla del departamento de QA y de cómo se ha reducido al tener gente que hace sus propios tests. Dice que TDD da una falsa confianza que hace pensar que no hacen falta QAs, aunque el viejo modelo de lanzar el software a QA y olvidarse de él está desfasado. Habla de la importancia de que alguien externo al equipo haga las pruebas, ya que solo escribiendo tests para una pipeline sigue habiendo bugs. La nueva generación, ajena al debate de los 90s, no es capaz de explicar por qué suceden esos bugs. Es bueno los desarrolladores sepan hacer tests y se responsabilicen (mejor que en el pasado), pero es un entendimiento superficial de qué es la calidad. Es el usuario final el que se encuentra los problemas.

Kent Beck comenta que Facebook se deshizo de los QA y ahora tiene unos pocos. Para él las pruebas son responsabilidad de los desarrolladores. Cita la "época oscura" de irresponsabilidad del software, con ciclos largos y mal feedback. Los QAs pueden tener sentido en sistemas complejos con diseños difíciles o deficientes y que necesitan ayuda externa pero, en general, prefiere la responsabilidad de los desarrolladores. Habla de aprender de los tests no escritos, en los que hay que tirar de logs o estadísticas, para descubrir el problema. Esto nos recuerda que no debemos ser muy arrogantes y creer que no necesitamos pruebas para desarrollar.

Martin Fowler comenta que en Thoughtworks tienen QAs en casi todos los proyectos. Dice que la visión interdepartamental entre desarrolladores y QA de los 90s ha cambiado a algo más colaborativo. Apunta a la automatización de QA para ir más rápido (y que la relación vaya mejor). En un artículo del 2018 publicado en la web de Martin Fowler, Ham Vöcke comenta que el enfoque de un equipo de QA centralizado es un anti-patrón que no debería tener cabida en el mundo DevOps. [43]

#### *Foco en los tests*

Martin Fowler habla de los días en los que trabajó con Kent Beck en el proyecto C3 con XP [44], donde hicieron un uso intensivo del testing desde el principio y, aunque no fue un enfoque test first, las historias no se consideraban hechas hasta que el código no tenía tests funcionando. Distingue entre TDD y self testing code. Self testing code consiste en lanzar, con un comando simple, muchos tests de forma rápida. Permite refactorizar y añadir nueva funcionalidad a producción, etc.

En cambio, TDD es una técnica muy particular que, si es bien practicada, da como resultado self testing code. Es uno, y el más importante, beneficio. Si alguien consigue al final de una iteración código y tests (self testing code), aunque sea por otro camino, es un resultado válido. Opina que hay consenso en la importancia que se da a los tests de regresión y el código auto-testeado, aunque quedan cosas por hacer, dado que la testeabilidad todavía no está en el centro del desarrollo.

DHH comenta que también aprecia el aporte que ofrece TDD al auto-test y las pruebas de regresión, aunque esto se puede conseguir igualmente sin TDD.

Emily Bache habla de las bondades de TDD, al poner la automatización de los tests en el centro, lo cual cambia profundamente la forma de encarar un problema de codificación. Esto hace que se pruebe cada línea de código, aislando una pieza en particular para probarla. Los tests son ligeros y, si uno falla, se detecta rápido el problema gracias a la regresión que suponen.

## *Documentación del sistema*

DHH comenta que la documentación del proyecto ha sido sustituida por los tests como la cima de la cadena de autoridad. Son los tests los que mandan si algo está bien o mal y no el código. Además, se invierte más tiempo en el código de tests que en el de producción.

Kent Beck ha tenido la experiencia de tirar el código y quedarse con los tests para hacer el código desde cero. Suele acudir a los tests a consultar cosas. Habría que preguntarse en qué casos se podría prescindir del código y en cuáles de los tests

Martin Fowler dice que lo fundamental es la doble comprobación entre código y tests. Si hay un fallo, es un desajuste entre ambos. Poner mucha energía en los tests (en el entorno), es un error, ya que los tests tienen que proporcionar valor al usuario.

## *Cobertura de pruebas*

DHH comenta que la cobertura dependerá del tipo de proyecto, pero que TDD parece forzar siempre a altas cotas de cobertura. Apunta al coste de usar TDD, en concreto con testing excesivo. Hay más tests que líneas de código y, eso que es visto como bueno, dificulta cambiar el comportamiento del código en cuanto a nueva funcionalidad. Afirma que podría haber código con más o menos cobertura y pone el ejemplo de Active Records en Ruby On Rails, en el que existen validaciones en las que se indica si un valor tiene que estar presente y es correcto.

Comenta la diferencia entre un equipo de consultoría que se hace cargo del código de un tercero y tiene que hacer muchas pruebas y un equipo de un producto estable, que conoce bien el producto y que tiene un 60% / 80% de cobertura, lo cual es suficiente. Sería bueno añadir este tipo de matices para poder relajar la afirmación de que cada línea de producción tenga una línea de test.

Kent Beck pone como ejemplo JUnit en el que había una cobertura del 100%. Si hay un conjunto de pruebas sin cobertura delta, se podrían eliminar sin perder la capacidad de ejercitar partes del sistema, a no ser que tengan algún propósito de comunicación. A las pruebas que se escriben al principio y luego se pueden eliminar, las llama andamiaje (scaffolding test). Estas consisten en escribir algún tipo de prueba de sistema, conseguir que funcionen y, cuando no sean necesarias, eliminarlas. Tener exactamente el mismo comportamiento cubierto de múltiples maneras diferentes es acoplamiento, porque cambiar cualquiera de ellas implica cambiar las otras.

Habla de Christopher Glaeser [45], un gurú de los compiladores que tenía más líneas de test que de código, lo que lo hacía sólido. Comenta que, si el acoplamiento es del estilo de un compilador, en el que una modificación puede afectar al rendimiento, la ratio es alta. En sistemas sencillos, bajamente acoplados y fáciles de probar, la ratio debería ser menor.

Hacer pruebas primero no es necesario en todos los casos, pero que cuando lo haces, tienes más opciones de trabajo. Hace falta concretar la audiencia y necesidades para formular un acuerdo.

En una respuesta de Kent Beck a la pregunta: “¿Qué nivel de granularidad deben tener las pruebas unitarias?” que hizo un usuario de StackOverflow, respondió así: “Me pagan por código que funciona, no por pruebas, así que mi filosofía es probar lo menos posible para alcanzar un determinado nivel de confianza (sospecho que este nivel de confianza es alto en comparación con los estándares de la industria, pero eso podría ser sólo arrogancia). Si no suelo cometer un tipo de error (como establecer las variables incorrectas en un constructor), no lo pruebo. Sí que tiendo a dar sentido a los errores de las pruebas, así que tengo mucho cuidado cuando tengo lógica con condicionales complicados.



Cuando codifico en equipo, modifico mi estrategia para probar cuidadosamente el código en el que, colectivamente, tendemos a equivocarnos.

Cada persona tendrá sus propias estrategias de pruebas basadas en esta filosofía, pero me parece razonable dado el estado inmaduro de la comprensión de cómo las pruebas pueden encajar mejor en el bucle interno de la codificación. Dentro de diez o veinte años probablemente tendremos una teoría más universal sobre qué pruebas escribir, qué pruebas no escribir y cómo distinguirlas. Mientras tanto, lo mejor es experimentar.” [62]

Martin Fowler comenta que en ThoughtWorks puede haber exceso de testing, pero no le preocupa, siempre que no exceda el coste. Siempre se pregunta: "si rompo esta línea de código, ¿dejan de pasar las pruebas?". En caso afirmativo, el test estará bien. La otra comprobación es revisar si es útil el test (ej: no tiene sentido probar los getters). Si se usa un framework, no probará ciertas cosas, ya que delegará en el framework (ej: si uso Rails). Los tests son parte del aprendizaje del sistema. Con pocos tests se tiene inseguridad ante cambios. Muchos tests provocan que haya más cambios en pruebas que en código. Ese aprendizaje viene de la experiencia.

### Impacto en el desarrollo

Kent Beck arranca el tercer vídeo con una disertación de corte general sobre el balance para determinar cuánto testing es necesario, la frecuencia del feedback, el grado de fiabilidad que se pretende y el coste que tiene todo eso, así como del factor tiempo en la vida del software. Si el software tiene una vida corta, igual no merece la pena la inversión.

En un artículo de Mike Bland, citado por Martin Fowler [46], el autor comenta que los tests unitarios tienen costes, como el de la formación inicial hasta alcanzar las destrezas necesarias o el de lidiar con tests indeterministas o lentos, que se vuelven contraproducentes, conduciendo al equipo a un rechazo de la actividad de pruebas. El articulista narra un ejemplo en el que un equipo de Google aumentó su cobertura de pruebas, alcanzando un alto grado de productividad general y estableciendo una cultura de pruebas.

Ian Cooper comenta que se encontraba proyectos en los que el código de tests era tres veces más que el de producción y se preguntaba si era normal. La gerencia se oponía a que hicieran tests y el equipo era un 20% o 25% más lento que antes, dado que escribían tests. Después expone varias alternativas a esto, que parece desechar.

En el 2009-2010 aparecen metodologías que abogaban por no hacer TDD, porque hacía más lento el desarrollo. La primera que cita es "Programmer Anarchy", creada por Fred George. La idea principal es dejar solo desarrolladores en el equipo y hacer piezas de código muy pequeñas (tomados como micro-servicios). Es más rentable reescribir cierta parte de código que tener tests, que son más costosos. [47] [48] [49]

Habla de Lean y de "Spike and Stabilize", modelos asociados a startups de Silicon Valley en las que la filosofía era entregar software rápido a los clientes, dado que lo importante era la idea de negocio incipiente y la solución ofrecida podía ser incorrecta. Si era necesario, se tiraba a la papelera y se volvía a escribir. En ese contexto, TDD se veía útil cuando la organización crecía y había que escalar los procedimientos. También cita los duct-tape programmer (cinta de embalar), aquellos que sacan código rápido y son bien vistos por la gerencia, pero su código es de baja calidad y difícil de mantener. Para DHH, la alta cobertura de TDD impacta claramente en el desarrollo. Según un estudio realizado en Microsoft, en el que compararon el impacto de los procesos de desarrollo con y sin TDD en cuanto a calidad y tiempo total de desarrollo, la calidad del código desarrollado con TDD aumentó entre 2,6

y 4,2 veces en comparación con el código desarrollado sin TDD. Por otra parte, los directores de proyecto estimaron que TDD aumentó el tiempo total de desarrollo entre un 15 % y un 35 %. La cobertura de bloques de cobertura de bloques fue del 79-88 % a nivel de pruebas unitarias en los proyectos que empleaban TDD. [67][68]

DHH habla de la dificultad que tiene para escribir primero la prueba y comenta que se puede llegar a resultados equivalentes haciendo la prueba antes o después. Puede que haya personas a las que les resulte más fácil hacer primero la prueba, pero parece que TDD impone ese método y no da alternativas.

## Diseño / arquitectura

### Diseño/arquitectura previos VS emergente

Kent Beck comenta que TDD puede ayudar a desbloquear situaciones en las que el diseño no se ve claro.

Jim Coplien dice que tiene la experiencia de ver en varios clientes cómo se utiliza TDD sin arquitectura ni framework, lo que coincide con la posición original de Kent Beck: se usa TDD para dirigir la arquitectura. Coplien afirma que en el libro de XP dice: “Sí, hagan algo de arquitectura por adelantado, pero no os dejéis la piel”

TDD dirige a una arquitectura bottom-up procedimental, porque lo que se testean son unidades. El problema es que hay un desajuste entre los objetos de dominio y los tests, orientados a procedimientos, lo que no asegura la misma funcionalidad tras el refactoring. Sugiere partir de un lugar desde el que se capture esa otra dimensión estructural.

Para Coplien, es necesario anticipar el conocimiento del dominio para definirlo y modelarlo en los objetos adecuados. También se deben tomar decisiones previas de arquitectura para facilitar el trabajo posterior: UI, roles, interfaces que documentan la estructura del dominio.

Uncle Bob afirma, en su artículo “TDD Harms Architecture” [66], que coincide con Coplien en que la arquitectura no emerge de TDD. La idea de que el diseño de alto nivel y la arquitectura emerjan es absurda. Uncle Bob afirma que antes de comenzar a codificar en un proyecto software, se necesita una visión arquitectónica. TDD no puede ofrecer esa visión. Eso no significa que el diseño no emerja de TDD, lo hace, pero no al nivel más alto. El diseño que emerge de TDD está uno o dos peldaños por encima del código, íntimamente conectado al código y al ciclo rojo-verde-refactor.

Para Uncle Bob se itera para ir refinando la arquitectura y, por el camino, se desechan opciones. Es mejor no crear abstracciones que luego no se utilizarán y dejar que las pruebas conduzcan la arquitectura más tarde, cuando sea necesario.

Ambos, Coplien y Bob, coinciden con que la arquitectura debe evolucionar.

### Diseño/arquitectura incremental

Kent Beck comenta que, si no se sabe implementar algo, al menos se puede idear un test para luego probar una solución. De esta manera se pueden descartar ideas malas con un coste bajo, al definir la prueba primero. Pone a Junit como ejemplo en el que se hicieron pruebas muy estrictas y se diseñó

poco a poco. El interfaz de las clases del framework es muy claro, y este es el efecto que consigue TDD.

Hay momentos en el desarrollo de software en los que hay cosas difíciles de implementar. Una estrategia suele ser retirarse, hacer el problema más simple y resolver ese problema menor. Hacer un problema sencillo puede ser complejo, por lo que hay que iterar recursivamente. Ahí es donde encuentra el feedback de TDD de ayuda. TDD como feedback de desarrollo es una herramienta poderosa.

Cuando se plantea un problema, hay que pensar en cómo obtener feedback, confianza, dividir el problema en trozos. Hay momentos en los que no es así, con entradas y salidas poco claras y hay que buscar otras formas de feedback. Pero, en general, se tiende a buscar situaciones en las que se pueda aislar un problema, con entradas y salidas claras y restringidas, volviendo a un bucle de feedback a partir de ahí.

Él siempre trata de trocear el problema en abstracciones a las que poder aplicar el flujo de TDD. Siempre trata de buscar la confianza, significancia (en cuanto al valor aportado), colaboración técnica y valor en el largo recorrido. Alude a explorar nuevas vías para obtener esos grandes objetivos y las habilidades para aplicar TDD y lograrlos. Hace referencia a mejorar en el diseño, a experimentar y ver los límites y los acuerdos. TDD no está muerto, pero como Fénix, tendrá que resurgir de sus cenizas. Se termina preguntando qué habilidades se necesitan para lograr esos efectos a largo plazo. Martin Fowler comenta que la fase de refactor depende de las destrezas, que a veces las personas se atascan en eso y que es algo que se va adquiriendo. Lo interesante es ver cómo se va limpiando el código y haciendo más claro.

Ian Cooper, como se comentó en apartados anteriores, se basa en la idea de Kerievsky de que los patrones no se aplican correctamente. En palabras de Kerievsky: “Quizá sea imposible evitar equivocarse en el camino del aprendizaje de patrones. De hecho, la mayoría de nosotros aprendemos cometiendo errores. En más de una ocasión he sido ingenuo con los patrones.

La verdadera sencillez de los patrones viene de usarlos sabiamente. La refactorización nos ayuda a hacerlo centrando nuestra atención en eliminar la duplicación, simplificar el código y hacer que el código comunique su intención. Cuando los patrones evolucionan en un sistema por medio de la refactorización, hay menos posibilidades de sobreingeniería con patrones. Cuanto mejor sepas refactorizar, más posibilidades tendrás de encontrar la alegría de los patrones.”

Emily Bache comenta que TDD fuerza a hacer pequeños diseños, no solo de código sino también de tests.

Para Sandro Mancuso, reputado propulsor del movimiento software craftsmanship y del enfoque de Londres, “tener pasos de refactorización no es suficiente para llamar a TDD una herramienta de diseño”. TDD es un flujo de trabajo de software que proporciona muchos beneficios, incluido el recordatorio constante de hacer el código mejor. Lo que quiera que signifique hacer el código mejor no es parte de TDD. Según Mancuso, en el enfoque de Chicago no se hacen consideraciones de diseño iniciales, sino que el diseño emerge completamente desde el código. Este diseño lo considera “micro-diseño”, contrastándolo con el macro-diseño, que trata de cómo se va a modelar nuestro dominio a más alto nivel. TDD no ayuda con el macro-diseño. [69]

## Métodos y principios

Para Jim Coplien, una prueba unitaria se enfoca en la API de un procedimiento y sus argumentos, lo cual es un procedimiento heurístico de búsqueda de bugs que puede ser infructuoso. Aboga más por

el diseño por contrato, en el que se indican las precondiciones y postcondiciones e invariantes. Tiene la ventaja de centrarse en la vista externa del código, haciéndolo de manera más eficaz que la prueba y obteniendo una cobertura mayor, al cubrir toda la gama de argumentos en lugar de dispersar al azar algunos valores.

Afirma que Bertrand Meyer lo ha llevado esto más lejos con CDD, Desarrollo Dirigido por Contrato. Esta técnica consiste en alimentar los contratos con valores aleatorios de forma que se comprueben las postcondiciones. El problema de TDD, como la mayoría de la gente lo practica, a nivel de clase, es que de esa forma es muy difícil rastrear el negocio desde la API a través de toda la jerarquía. Con aserciones, se tiene el acoplamiento adecuado y esencial entre la semántica de la interfaz y del código en sí, mientras que con los tests unitarios el acople es más difuso y difícil de manejar.

Para Uncle Bob, TDD hace lo mismo, especificando un conjunto de comprobaciones entrantes sobre los argumentos y salientes sobre los valores devueltos. Afirma que se puede pasar de uno a otro. Para Bob, las pruebas unitarias dependen del código de producción, pero no a la inversa. En cambio, en CDD, los contratos están dispersos por el código, lo que es molesto.

Tal y como se afirma en el libro "The pragmatic programmer", ambos métodos son compatibles: "Bertrand Meyer (Object-Oriented Software Construction) desarrolló el concepto de Diseño por Contrato para el lenguaje Eiffel (basado en parte en trabajos anteriores de Dijkstra, Floyd, Hoare, Wirth y otros). Se trata de una técnica sencilla pero potente que se centra en documentar (y acordar) los derechos y responsabilidades de los módulos de software para garantizar la corrección del programa. ¿Qué es un programa correcto? Uno que no hace ni más ni menos de lo que dice hacer. Documentar y verificar esa afirmación es el núcleo del Diseño por Contrato (DBC, por sus siglas en inglés).

¿Es necesario el diseño por contrato en un mundo en el que los desarrolladores practican pruebas unitarias, desarrollo basado en pruebas (TDD), pruebas basadas en propiedades o programación defensiva?

La respuesta corta es "sí". El DBC y las pruebas son enfoques diferentes del tema más amplio de la corrección de los programas. Ambos tienen valor y se pueden utilizar en distintas situaciones. El DBC ofrece varias ventajas sobre los enfoques de comprobación específicos:

- DBC no requiere ninguna configuración o mocking
- DBC define los parámetros de éxito o fallo en todos los casos, mientras que las pruebas sólo pueden centrarse en un caso específico a la vez.
- DBC es más eficiente (y DRY-er) que la programación defensiva, donde todo el mundo tiene que validar los datos en caso de que nadie más lo haga.
- TDD y otras pruebas sólo tienen lugar en "tiempo de prueba" dentro del ciclo de compilación. Pero DBC y las aserciones son permanentes: durante el diseño, el desarrollo, el despliegue y el mantenimiento.
- TDD no se centra en la comprobación de invariantes internas dentro del código bajo prueba, es más un estilo de caja negra para comprobar la interfaz pública.

TDD es una gran técnica, pero como ocurre con muchas técnicas, puede invitarte a concentrarte en el "camino feliz", y no en el mundo real lleno de malos datos, malos actores, malas versiones y malas especificaciones." [50]

Por último, Uncle Bob cita Responsibility Driven Design (RDD), de Rebecca Wirfs-Brock. [51] [52] [53]. Rebecca Wirfs-Brock inventa RDD en 1990, cuando trabajaba en Tektronix. Se trata de una

técnica de modelado orientado a objetos, basada en las experiencias de una serie de diseñadores de Smalltalk.

RDD es una técnica de diseño que hace hincapié en el modelado del comportamiento mediante objetos y colaboraciones. En un modelo basado en responsabilidades, los objetos desempeñan funciones específicas y ocupan posiciones bien conocidas en la arquitectura de la aplicación. Cada objeto es responsable de una parte específica del trabajo. Colaboran de formas claramente definidas, contratándose mutuamente para alcanzar los objetivos de la aplicación. Al crear una "comunidad de objetos", asignando responsabilidades específicas a cada uno, se construye un modelo colaborativo de la aplicación.

En cuanto a la evaluación del resultado, Martin Fowler comenta que es difícil determinar si un código es bueno o malo. Requiere tiempo de trabajo del equipo y cada uno establece un criterio de qué es aceptable y qué no lo es.

Kent Beck afirma que hay demasiadas variables para poder verificar experimentos, pero que una mentalidad científica ayuda. Dice que no demuestra nada, pero que se pueden recopilar datos y compararlos. No reclama ninguna especie de ley universal. Pone como ejemplo las ciencias políticas, y las asemeja con las ciencias de la computación.

Para Sandro Mancuso “TDD no es una herramienta de diseño. Es un flujo de trabajo de desarrollo de software que tiene indicaciones para mejorar el código en su ciclo de vida. En estas indicaciones (escribir pruebas y refactorizar), los desarrolladores necesitan conocer algunas pautas de diseño (4 Reglas de Diseño Simple, Domain Driven Design, SOLID, Patrones, Ley de Demeter, Tell, Don't Ask, POLA/S, Diseño por Contrato, Feature Envy, cohesión, acoplamiento, Principio de Abstracción Equilibrada, etc) para mejorar su código. Sólo decir refactorización no es suficiente para llamar a TDD una herramienta de diseño.” [69]

## Calidad VS daño

Kent Beck comenta que no se puede asociar TDD a las decisiones de diseño. Para él, el quid de la cuestión es entender las decisiones de diseño que se toman en cada contexto y cómo afectan. Más allá de si es TDD o no lo es, se trata de cómo se toman esas decisiones, cómo se busca el feedback del sistema, cómo ganar la confianza necesaria, etc.

TDD ejerce presión en la cuestión de la evolución del diseño. Es la testeabilidad como acicate del diseño. En cualquier caso, se trata de una cuestión de consenso entre el beneficio que aporta el grado de aislamiento y la inversión que supone. Es conveniente buscar formas de probar. A veces hay que recurrir a los resultados intermedios para comprobar cosas. Un testeo deficiente puede ser síntoma de un diseño pobre.

DHH comenta que la arquitectura está dirigida por mocks para evitar colaborar con las capas externas y lentas del sistema, como la BD. Este abuso de mocks y búsqueda de patrones de arquitectura como el Hexagonal para facilitar las pruebas, dañan la arquitectura. Es lo que llama daño inducido por pruebas.

Reconoce que ese flujo de trabajo sí puede ser aportar beneficios cuando hay una entrada y una salida deseada muy claras y no se tiene demasiada dependencia de un contexto. Pero, en su caso, la mayor parte del tiempo no es así, lo que fuerza a la generación de mocks para poder independizar el código del contexto y lograr flujo de trabajo. Esto provoca soluciones forzadas en aras de la testeabilidad. Se pregunta si esta forma de trabajar es aplicable a MVC.

La comunidad de TDD cree que el principal retorno de TDD es el diseño obtenido y la idea de que un diseño solo se puede mejorar introduciendo más tests. En realidad, él ha visto justo lo contrario, diseños llenos de mock en aras de la testeabilidad. Hay cosas más importantes que si el sistema está probado y es si el sistema está claro y es comprensible.

Pone como ejemplo Active Records, un patrón que se utilizaba en Rails. En el libro "Patterns of Enterprise Application Architecture" de Martin Fowler [54], aparece esta definición de "Active Record": "Objeto que envuelve una fila de una tabla o vista de base de datos, encapsula el acceso a la base de datos y añade lógica de dominio sobre esos datos.

Un objeto contiene tanto datos como comportamiento. Muchos de estos datos son persistentes y deben almacenarse en una base de datos. Active Record utiliza el enfoque más obvio, poniendo la lógica de acceso a los datos en el objeto de dominio. De esta forma, todo el mundo sabe cómo leer y escribir sus datos desde y hacia la base de datos. “

Parece que este patrón está muy extendido en la comunidad de Ruby on Rails [55]. En el libro "SQL Antipatterns" aparece citado como un anti-patrón [56]. Existen varios problemas con este patrón, entre ellos, el acoplamiento del modelo de negocio con la base de datos y la dificultad de testear, dado que es necesaria una base de datos para probar. Las operaciones de CRUD tienen que estar en el objeto del dominio, lo que en principio no sería su responsabilidad. Además, esta fórmula fuerza a tener un objeto por tabla, utilizando la agregación para ampliarlo a múltiples tablas.

DHH muestra un fragmento [57] de código con el que trata de demostrar el impacto negativo que puede tener TDD. Opina que se añaden indirecciones innecesarias para poder probar el código, añadiendo capas de complejidad. Se basa en la charla de Jim Weirich "Decoupling from Rails" [58]. Esta misma idea es la que la desarrolla en su post test-induced design damage.

El aumento de las indirecciones hace más difícil hacer cualquier cambio. Pone la arquitectura hexagonal como un ejemplo de cómo se lleva esto al límite, utilizando indirecciones para poder llamar a una aplicación Web desde consola, cuando este caso no se va a dar la mayoría de las veces. Además, tratar de aislar y abstraer subsistemas tan diferentes como bases de datos o servicios web es una especie de quimera. Se suele elegir este enfoque por testeabilidad, aislamiento, posibilidad de usar mocks y velocidad, en lugar de por las necesidades del negocio.

Al tratar de conseguir bajo acoplamiento a toda costa, se acaba dañando la cohesión. Pone como ejemplo un Controller de un MVC sencillo que hace el envío de un mail directamente, o usa el modelo, o el código de autenticación en aras de la concreción. Comenta que hay una creencia sin evidencia alguna de que con TDD se llega a mejores diseños. Él considera que, para un gran número de casos, simplemente no hay forma de que TDD lleve a un mejor diseño. Más bien, conducirá al lugar equivocado.

Emily Bache opina que hacer el test primero, fuerza a diseñar la API y que el momento menos costoso para cambiar la API es antes de que exista la implementación. Esto también permite ver qué colaboradores, argumentos para el método, clases y dobles de test que las reemplacen serán necesarios.

La ponente transformó uno de los ejercicios de un estudio de Luca Minudel [61] en una code kata (TierPressure-Kata) [60]. La kata consiste en escribir los unit tests de la clase alarma, que tiene una dependencia que es difícil de instanciar en los tests unitarios. La idea es introducir un interfaz, el "Sensor", de forma que se pueda reemplazar el test con un test double. Se hace que la alarma dependa de una abstracción, un interfaz, en lugar de una clase concreta, lo que la hace más testeable. Se ha

aplicado el principio de inversión de dependencias en la alarma. Dice que se relacionan estrechamente la testeabilidad y el principio de inversión de dependencia. Concluye que TDD orienta a un mejor uso de principios de diseño.

Después rescata el ejemplo de DHH en el que habla de design damage. En el ejemplo, un controller hace uso de una clase encargada de guardar en BD y mostrar un mensaje en la UI. La ponente comenta que hay acoplamiento entre la UI y la BD en el controller. DHH hace una demostración de cómo habría que refactorizar el controller para poder aislarlo de la BD y UI, pero el refactoring añade cuatro clases más y muchas indirecciones, con lo que el código se vuelve mucho más complejo. Trata de contraargumentar a DHH diciendo que, en una respuesta a su post, alguien afirmaba que se podía probar de una manera mucho más sencilla. (El enlace de la persona que respondía, ya no está accesible: <http://patmaddox.com/2014/05/15/poof-and-then-rails-was-gone/>). La motivación sería dividir las responsabilidades y escalar la aplicación para desarrollar un sistema más grande. Dice que el ejemplo de DHH está sesgado.

Concluye que TDD afecta al diseño, empujándote a no hacer clases muy acopladas mediante el principio de Inyección de Dependencias. Pero opina que no te convierte en un experto diseñador, con lo que hay que saber cuándo este tipo de enfoques es adecuado. Hay una parte de verdad en lo que afirma DHH, y es que se puede hacer sobrediseño y añadir muchas indirecciones. Termina con que el diseño es difícil, se haga TDD o no.

Cita la frase "Todos los problemas en ciencias de la computación pueden ser resueltos agregando otro nivel de indirección, salvo el problema de demasiados niveles de indirección". Parece que la frase es de David J. Wheeler, y aparece citada en "The C++ Programming Language Fourth Edition Bjarne Stroustrup - 2013". El corolario "... salvo el problema de demasiados niveles de indirección", parece que es de Kathleen Henney, según una entrada en Twitter en la que el autor se la atribuye [59]

Martin Fowler sostiene que lo que se persigue es el nivel de aislamiento, porque posibilita probar más fácilmente.

### Testeabilidad y diseño

Kent Beck lanza la idea de al hacer las partes intermedias del sistema más testeables, se puede llegar a mejores ideas de diseño y a una mayor comprensión del sistema. En cualquier caso, se trata de compromisos entre el diseño/testeabilidad y el coste que conlleva.

TDD, y la testeabilidad como principio, presionan al diseño. Hay que determinar la granularidad en la cobertura de tests, llegando a un consenso entre las posiciones extremas de probar muchas decisiones de implementación con un solo test o solo una decisión por cada test. En ese continuo, se debe ser consciente de la dimensión del sistema y adaptar el estilo al coste/beneficio real.

Los tests ayudan al progreso allí donde el código no está claro. Un testeo deficiente puede ser síntoma de un diseño pobre.

DHH no ve que sea una ecuación "más testeable = mejor diseño". Lo tilda como epifanía.

### Adaptabilidad, métricas



## Adaptabilidad

Martin Fowler coincide con David y Kent en que TDD no funciona bien en cierto tipo de problemas. Comenta que actualmente (2014), la mayor parte de lo que hace no es con TDD. TDD puede no ser para determinados contextos, aunque sí es útil tener pequeños pasos y tests de regresión, pero no es aplicable en ciertos casos. Pone el ejemplo de su sitio web, en el que no puede aplicar TDD. Para ciertas cosas, como un presentador de slides, sí lo ha utilizado.

También dependen de la personalidad de la gente. Para las personas con menos experiencia, TDD les fuerza a separar en pequeñas partes. Es bueno para situaciones como la de separar el interfaz de la implementación, que es una de las partes más difíciles del software cuando se trabaja con pequeñas partes. TDD no garantiza buenos resultados si no se tiene buena experiencia diseñando, haciendo módulos, etc. Lo típico de la gente que empieza con TDD, es que no refactoriza demasiado.

Cuando se evalúa TDD, no se puede comparar un desarrollador experimentado con uno que no lo es. Lo que habría que comparar es qué haría un inexperto sin la ayuda de TDD. Habla de la dificultad que tiene esta comparación, dado que no se puede medir. Para la gente inexperta, TDD aporta el self testing code. Aunque el diseño no sea perfecto, al menos el código tiene pruebas y permite hacer refactoring de él. La gente más experta, tiene más capacidad de decidir cuál es la mejor herramienta para cada situación y tiene mejores decisiones con o sin TDD.

Hay consenso en que TDD puede ser útil en algunos contextos. Hay diferencias en cuanto a los contextos en los que se puede aplicar, y es algo difícil de determinar, dada la cantidad de variaciones.

DHH no aprecia que ofrezca mejores diseños, por ejemplo, en MVC. Sí puede ser útil cuando están claras las entradas y salidas de algo que hay que construir (un convertidor de markdown, por ejemplo), entonces puede valer como una herramienta más de las muchas que se usan, pero no para hacer aplicaciones web. TDD puede ser bueno para algunas cosas, no para aplicaciones web, pero tal vez para otras sí.

Kent Beck comenta que hay momentos en el desarrollo de software en los que hay cosas difíciles de implementar. Una estrategia suele ser retirarse, hacer el problema más simple y resolver ese problema menor. Hacer un problema sencillo puede ser complejo, por lo que hay que iterar recursivamente. Ahí es donde encuentra el feedback de TDD de ayuda. TDD como feedback de desarrollo es una herramienta poderosa. Hay momentos en los que no es así, con entradas y salidas poco claras y hay que buscar otras formas de feedback. Pero, en general, tiende a buscar situaciones en las que pueda aislar un problema, con entradas y salidas claras y restringidas, y vuelve a un bucle de feedback a partir de ahí.

Cuando se plantea un problema, piensa en cómo obtener feedback, confianza, dividir el problema en trozos, etc y que tal vez no todo el mundo se lo plantee de la misma forma, así que el punto de vista de David le ayuda a plantearse eso.

## Ajuste en la cobertura y tipo de pruebas

Kent Beck comenta que TDD, y la testeabilidad como principio, presionan al diseño. Hay que determinar la granularidad en la cobertura de tests, llegando a un consenso entre las posiciones extremas de probar muchas decisiones de implementación con un solo test o solo una decisión por cada test. En ese continuo, se debe ser consciente de la dimensión del sistema y adaptar el estilo al coste/beneficio real.



En cualquier caso, se trata de una cuestión de consenso entre el beneficio que aporta el grado de aislamiento y la inversión que supone. Diserta sobre el balance para determinar cuánto testing es necesario, la frecuencia del feedback, el grado de fiabilidad que se pretende y el coste que tiene todo eso, así como del factor tiempo en la vida del software. Si el software tiene una vida corta, igual no merece la pena la inversión.

DHH menciona el artículo de James Coplien, "Por qué la mayoría de las pruebas unitarias son un desperdicio". Opina que dividir las funciones para apoyar el proceso de prueba, destruye la arquitectura y la comprensión del código. TDD se centra en la unidad. La unidad es la pieza sagrada, es donde se pueden controlar todas las variables. Pero esas pocas partes examinadas tal vez no constituyan el nivel adecuado. Tal vez las pruebas no deberían centrarse en la unidad la mayor parte del tiempo. [70] [71]

## Métricas

DHH dice que las métricas de velocidad y cobertura no dan pistas del diseño y se prioriza antes que la entrega de valor o la claridad del sistema.

Emily Bache muestra una herramienta de coding dojo (<https://cyber-dojo.org/creator/home>) en la que se resume el número de tests que pasan tras cada ejecución y se hace una comparativa entre un grupo de iniciados y el mismo grupo varias katas después. Justifica que los ciclos se vuelven más estables y con una pauta: pasan en verde, se hace refactoring, vuelven al rojo, etc. Mientras en las primeras ejecuciones, los tests se tiran más tiempo en rojo.

## Escuela de Londres

Ian Cooper expone que, al refactorizar, notaba cómo muchos tests se rompían, especialmente los que hacían uso intensivo de mocks, porque los mocks decían exactamente qué debía pasar. Se suponía que los tests permitían la refactorización. No solo no estaba sucediendo, sino que además se convertían en un obstáculo. Eso contradecía lo que afirmaba Martin Fowler sobre el refactoring, afirmando que permitía cambiar los detalles de implementación sin romper los tests. Los test que abusan de mocks son difíciles de leer. Incluso puede que el test no tenga ningún SUT, sino que todo sea mock y el test no haga nada.

Al tomar el SUT como una clase concreta, hay que hacer mocks de manera intensiva, lo que lleva a conocer los detalles de implementación de una clase. Hay que escribir tests para el contrato estable que ofrece la API. La clave para llegar a ello es el refactoring.

Algunas veces viene bien hacer tests para entender los detalles de la implementación y poder refactorizar. Luego hay que eliminar esos tests.

Enfocarse en los métodos para hacer las pruebas y describir las interacciones, dificulta la comprensión de los tests. El comportamiento es más sencillo de entender y las aserciones se hacen en el sistema. Si hay un rojo, se localizará el impacto debido a un cambio de comportamiento.

Cuando se expone la implementación, es difícil de cambiar, especialmente con el uso de mocks. Si se sabe demasiado de la implementación y sus detalles, se acopla el test al código y no se puede cambiar uno sin cambiar el otro. Este sería el problema principal de la escuela de Londres. Los mocks son útiles cuando los recursos son caros de crear, pero el problema es que se utilizan para aislar clases y acoplan el código al test.

Para Sandro Mancuso [72] los practicantes de TDD, incluidos los que no les gustan los mocks, están de acuerdo en usarlos en los límites de lo que estén probando. El problema es que rara vez se ponen de acuerdo sobre dónde deben trazarse los límites mientras prueban su código.

Para las personas acostumbradas a Outside-In TDD, los mocks son una herramienta de diseño, no una herramienta de pruebas. TDD se hace mucho más fácil cuando se dibujan los límites de los módulos. Una vez que se tiene un plan de diseño de alto nivel, el esfuerzo en TDD se vuelve más centrado y eficiente, casi mecánico. Pero, ¿qué hacer cuando los límites no están claros? A veces simplemente no se puede ver la solución y es necesario explorar. En casos así, se puede pasar directamente al TDD clasicista y trabaja en pasos de pequeños hasta que surja una solución.

Si se puede visualizar claramente el diseño de los módulos y el tipo de asociación, se hacen pruebas que conduzcan el diseño y se usarán mocks para diseñar la interacción entre los diferentes módulos. Si no se ve la solución, se puede pasar al modo de exploración, trabajando en pequeños incrementos, creando un pequeño desorden, y usando la refactorización para decidir cómo organizar mejor el código. Aunque pueda parecer una gran idea trabajar siempre en pequeños incrementos y refactorizar, Sandro Mancuso lo encuentra extremadamente lento e ineficiente, de ahí la razón por la que mezcla diferentes estilos de TDD.

Martin Fowler opina que TDD no indica el grado de aislamiento de las pruebas respecto a los sistemas externos. Hubo debate en la comunidad de XP sobre este tema, entre defensores y detractores a la hora de aislar los tests y de definir cómo eran los tests unitarios, lo que produjo múltiples definiciones de Unit Tests. Él opina que hay distintas escuelas de pensamiento.

En cuanto a Kent Beck, trata de evitar el uso de mocks, a veces mediante la refactorización. Hay gente que abusa de los mocks y luego no puede refactorizar, porque los tests están completamente acoplados a la implementación.

Como se ha citado anteriormente, para Kent Beck ni el enfoque descendente ni el ascendente describen el proceso de TDD. Para él la metáfora adecuada sería known-to-unknown, que implica tener algunos conocimientos y experiencia a priori, que se completarían durante el desarrollo. Esta definición no se ajustaría completamente a la llamada escuela de Chicago, que sí impone un criterio bottom-up o inside-out y que se ha asociado a la definición inicial de Beck [20].

Aunque Kent Beck defiende esta metáfora, lo cierto es que en el capítulo 17 dice: “A medida que se avanza hacia la periferia del sistema, hacia las partes que no cambian a menudo, las pruebas pueden ser más puntuales y el diseño más feo sin interferir en la confianza.”, lo que se podría asociar claramente con un enfoque inside-out. También afirma en el capítulo 29 que no existe una relación directa entre las clases de prueba y las clases del modelo, lo que podría indicar también que su enfoque de TDD es orientado al modelo.

Incluso parece desdecirse de todo lo anterior cuando en el capítulo 26, hablando del starter test, dice: “Encuentro a menudo que mi starter test está en un nivel superior, más como una prueba de aplicación, que las siguientes pruebas. Un ejemplo que suelo probar es un simple servidor basado en sockets. La primera prueba se parece a esto:

```
StartServer
Socket= new Socket
Message= "hello"
Socket.write(message)
AssertEquals(message, socket.read)
```

El resto de las pruebas se escriben en el servidor solo [...]"

Este enfoque podría interpretarse como una prueba de aceptación.

## Aprendizaje de TDD

### Adquisición y evaluación del conocimiento

Martin Fowler comenta que cualquier técnica requiere de un proceso de aprendizaje y evaluación del que TDD no está exenta. Descubrir que tiene aspectos de valor (como el self testing code) o que puede ser apropiado para un equipo concreto, forma parte del proceso. Para las personas con menos experiencia, TDD les fuerza a separar en pequeñas partes. Para la gente inexperta, TDD aporta el self testing code. Aunque el diseño no sea perfecto, al menos el código tiene pruebas y permite hacer refactoring de él.

Dice que es proclive a presentar técnicas con las razones sobre cuándo usarlas y cuándo no. Comenta que es importante que se repitan los principios básicos a las nuevas incorporaciones y que eso no se puede olvidar. También comenta la importancia de profundizar en estos temas, dado que no se puede traer cualquier técnica al equipo, porque hace falta usarla y excederse en el uso para encontrar lo que funciona. Solo desarrollando y con la experiencia, se puede alcanzar esto, dado que no es una ciencia exacta.

DHH coincide con Martin Fowler sobre lo de utilizar algo en exceso para descubrir los límites y aprender de ello.

### Evaluación del grado de madurez

Jim Coplien está en absoluto desacuerdo con la afirmación de Uncle Bob de que la práctica de TDD es un requisito imprescindible para considerar a un desarrollador como profesional.

Uncle Bob afirma que la industria ha carecido de un estándar de profesionalidad. Hoy día es una irresponsabilidad tener código sin pruebas unitarias y, una de las mejores formas de asegurarse es practicando TDD.

Martin Fowler comenta que es difícil determinar si un código es bueno o malo. Requiere tiempo de trabajo del equipo y cada uno establece un criterio de qué es aceptable y qué no lo es. TDD no garantiza buenos resultados si no se tiene buena experiencia diseñando, haciendo módulos, etc. Lo típico de la gente que empieza con TDD, es que no refactoriza demasiado.

Cuando se evalúa TDD, no se puede comparar un desarrollador experimentado con uno que no lo es. Lo que habría que comparar es qué haría un experto sin la ayuda de TDD. Habla de la dificultad que tiene esta comparación, dado que no se puede medir. La gente más experta, tiene más capacidad de decidir cuál es la mejor herramienta para cada situación y tiene mejores decisiones con o sin TDD. DHH comenta que es difícil hacer científica la evaluación de una técnica, principio o proceso. Dice que el debate se queda circunscrito a las opiniones y experiencias de cada uno.

Kent Beck comenta que siempre trata de trocear el problema en abstracciones a las que poder aplicar el flujo de TDD. Siempre trata de buscar la confianza, significancia (en cuanto al valor aportado), colaboración técnica y valor en el largo recorrido. Alude a explorar nuevas vías para obtener esos grandes objetivos y las habilidades para aplicar TDD y lograrlos. Hace referencia a mejorar en el

diseño, a experimentar y ver los límites y los acuerdos. TDD no está muerto, pero como Fénix, tendrá que resurgir de sus cenizas. Se termina preguntando qué habilidades se necesitan para lograr esos efectos a largo plazo.

Emily Bache habla del estudio de Luca Minudel [61] y lo aporta como una evidencia. El estudio indica que la gente que es buena en TDD, escribe código que sigue más de cerca los principios orientados a objetos, como el de Inversión de Dependencias (DIP) o el de responsabilidad única (SR). Se realizó proponiendo una serie de ejercicios con equipos.

## Katas y coding dojos

Emily Bache habla sobre su libro de Coding dojo para aprender TDD [73]. Resume que la idea es aprender TDD mediante katas. Una kata es un ejercicio de desarrollo de software en el que la atención se centra en aprender nuevas habilidades mediante la práctica y repetición de tipos de ejercicios. El coding dojo pretende fomentar este aprendizaje mediante el divertimento. El dojo es el lugar en el que se reúnen los desarrolladores para practicar.

Dan North tiene una charla en 2012 en la que habla de Deliberate learning [74] y critica las katas, porque se trata de optimizar la ejecución de una secuencia de movimientos que luego se repiten, pero no es código real, en el que nunca se hace dos veces lo mismo. Según Dan North, al inicio de los proyectos hay un grado de ignorancia alto sobre diferentes aspectos, que impiden que el proyecto se entregue con éxito. Es necesario hacer un esfuerzo continuo para conocer y reducir esos factores de ignorancia. Este esfuerzo expreso de conocer los impedimentos e incertidumbres en los proyectos dista mucho del gesto repetitivo de las katas.

Emily opina que la kata es para enfrentarse a los problemas y que los movimientos sean más fluidos. Lo compara con los músicos que ensayan escalas o que practican una parte de una pieza lentamente y luego repiten el conjunto. El coding dojo constituye una red de seguridad y motivación para fomentar el aprendizaje.

Muestra una herramienta de coding dojo [75], en la que se resume el número de tests que pasan tras cada ejecución y se hace una comparativa entre un grupo de iniciados y el mismo grupo varias katas después. Justifica que los ciclos se vuelven más estables y con una pauta: pasan en verde, se hace refactoring, vuelven al rojo, etc. Mientras en las primeras ejecuciones, los tests se tiran más tiempo en rojo.

Resume la diferencia entre práctica involuntaria (o incidental practice), cuando se repite algo que ya se ha hecho previamente, y práctica intencionada (deliberate practice), cuando se trata de adquirir algo nuevo fuera de la zona de confort. Recuerda unas palabras de Kent Beck: No soy un gran programador, soy un buen programador con grandes hábitos.

La frase de Kent Beck se encuentra en el libro de refactoring: 'Me recuerda a una frase con la que Kent Beck se refería a él mismo: "No soy un gran desarrollador, soy un buen desarrollador con grandes hábitos"' (Refactoring: Improving the Design of Existing Code, By Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts).

Por último, opina que las katas ofrecen una sensación de bienestar, al estar los problemas acotados, cosa que no siempre es posible con código en producción.

## Conclusiones y trabajo a futuro

### Conclusiones

#### Definición de TDD

En una de las charlas del debate, Kent Beck narra cómo pudo acceder desde muy temprana edad a libros de programación que había en su casa gracias a su padre, lo que le puso en contacto con conceptos fundamentales del desarrollo desde su infancia. Su trayectoria profesional, en la que cabe destacar sus inicios en Smalltalk rodeado de importantes profesionales, no pasa desapercibida. Su contribución al software es innegable, especialmente en lo que se refiere a la disciplina del testing. Con un conocimiento profundo del desarrollo software y una erudición amplia y heterogénea, tal y como demuestra en la bibliografía del libro *Extreme Programming*, en la que se mezclan libros de software con otros de filosofía, comunicación humana o historia, sus hitos en software han sido tan relevantes como discutidos, incluso en la actualidad.

Parte de la polémica quizás se deba a su peculiar forma de difundir sus propuestas mediante el uso constante de la metáfora, una estrategia que utilizó ampliamente con XP y que no parece casual. Este tipo de comunicación tiene un gran efecto en la audiencia, envolviendo el contenido en un discurso estético, novedoso y prometedor. Hay un esfuerzo intencionado en evitar la concreción, puede que con el propósito de cambiar esquemas obsoletos por una forma revolucionaria de hacer las cosas, puede que por una intención editorial.

Este mismo efecto es el que se encuentra en el libro de TDD, que carece de concreción de manera reiterada. Durante su lectura, surgen preguntas de todo tipo: ¿Cómo se adapta el proceso dependiendo del tipo y tamaño del proyecto?, ¿Cuándo y por qué motivos se puede introducir una excepción al proceso como las que él aplica?, ¿En qué consiste exactamente la lista de tests? ¿Se trata de requisitos, tareas pendientes, tests funcionales y no funcionales? ¿Cómo se escoge de manera adecuada la secuencia de tests? ¿Es un enfoque bottom-up? ¿Emerge el diseño y la arquitectura del sistema? ¿Es necesario tener conocimientos previos avanzados de diseño para practicarlo?

Esta falta de rigor y concreción ha hecho que la comunidad haya tenido que interpretar las palabras de Kent Beck para rellenar los huecos, tal y como hace el propio Ian Cooper en su charla. Tal vez la comunidad ha estado influida por el sesgo de la autoridad experta (“estar dispuesto a seguir las sugerencias de alguien que es una autoridad legítima” [76]) y el argumento basado en el respeto (“En vez de razones, empléense autoridades según la medida de los conocimientos del adversario” [77]). De esta forma, Kent Beck se ha convertido en una especie de maestro cuyas enseñanzas gnósticas han sido descifradas y completadas por la comunidad, que ha predicado en el mundo con versiones apócrifas del escrito original.

Además, el debate ha estado dominado por posiciones radicales, posiblemente guiadas por un “falso consenso”, un efecto por el que las personas creen que sus opiniones son las que asume la mayoría. Este atrincheramiento en posturas antagónicas se ha visto reforzado por el paradigma de la desconfirmación de creencias, rechazando la información incongruente y buscando nueva información que reafirme las creencias propias, ante el esfuerzo que supone examinarlas para cambiarlas. [76]

Lo cierto es que no hay una definición formal y estándar de TDD, sino variantes surgidas a partir de la original de Beck. Hasta el propio autor se salta sus reglas en el libro sin justificación alguna, siguiendo el atajo mental de su larga experiencia acumulada, lo que se podría llamar “intuición”. Martin Fowler denomina a toda esta amalgama de variaciones como “dispersión semántica”. Todo el

mundo habla de TDD, pero hay que llenarlo de contenido. Fowler se presenta como el más moderado defensor de TDD, tratando de llegar a un consenso en el debate y exponiendo unos argumentos serenos, como el de que hay que razonar sobre las bondades del método en lugar de imponerlo.

Tampoco queda claro cuál es la mejor vía de aprendizaje de TDD. Las katas, otro añadido a posteriori, no parecen resolver el problema. Suelen plantear un pequeño reto a resolver en unas pocas horas, pero estos desafíos no pueden asemejarse a los proyectos reales o a problemas de diseño de alto nivel. Además, la forma de plantearlas puede conducir a distintos resultados o a fijar una conclusión en la premisa del ejercicio, como parece hacer Emily Bache en el ejemplo de la alarma, algo tendencioso. Además, utiliza un argumento falaz para justificar que un equipo progresa en su aprendizaje: la regularidad del ciclo rojo-verde-refactor.

La propuesta de aprendizaje de TDD que hacen Kent Beck y Fowler pasaría por la experiencia como fórmula. Se trataría de examinar los límites, tal y como comenta Kent Beck, para ver las mejoras. Tal vez sea necesario el reseteo que reclama DHH para volver a definir qué es eso de TDD.

## Testing

Como se citaba anteriormente, Kent Beck ha hecho grandes aportaciones a la disciplina de testing, bien por su contribución, bien por amplificar prácticas que existían en el momento en el que definió TDD. Durante los debates hay consenso en que TDD aporta el auto-testing y la regresión, aunque DHH opina que con otros enfoques esto también es posible. El feedback que se obtiene con TDD a partir de las pruebas es muy potente y los tests constituyen una red de seguridad para el refactoring. Los primeros beneficiados serían los más inexpertos, según Fowler.

Al poner los tests por delante, como ciudadanos de primera clase, se incide en que probar es una parte fundamental del desarrollo. Así, el testing ha pasado de ser una actividad despreciable a convertirse en una habilidad que todo desarrollador debe tener. Dentro de ese cambio, xUnit ha sido la palanca que ha permitido el movimiento, aunque solo se trata de una herramienta. Lo más importante, quizás, es la visión de Kent Beck sobre los tests unitarios. Estos deben ser de caja negra y, por tanto, probar la API pública (interfaz) de una clase. Además, no deben acoplarse al código de producción, por ello hay que evitar los mocks en la medida de lo posible (que serían tests de caja blanca).

Los tests deben correr de manera aislada, para lo que es fundamental hacer un buen diseño de fixtures. Además, es importante revisar y mantener las pruebas, eliminando aquellas que se vuelvan innecesarias o utilizándolas como los andamios del sistema incipiente, que luego podrán ser eliminadas a medida que se avance hacia la solución final. Por tanto, los tests constituyen una gran documentación y debe haber un buen diseño de casos de pruebas.

En cualquier caso, nada de lo comentado anteriormente es exclusivo de TDD y en un enfoque test-last sería perfectamente válido todo lo anterior. Kent Beck fue variando su idea del testing a lo largo del tiempo, desde los tests obligatorios en un enfoque test-last (en el proyecto de C3, tal y comentaba Martin Fowler), pasando por el enfoque test-first en el libro de XP, hasta culminar en la idea de TDD. El acento está en el enfoque de pruebas tempranas que prácticamente se solapa con el propio desarrollo, en lugar de considerar la actividad como algo separado y a posteriori.

El debate test-first / test-last podría recordar a aquel otro de big endian / little endian. ¿Distan mucho los resultados de uno y otro enfoque? Tal vez al poner el test en primer lugar aparecen cuestiones de diseño e incluso psicológicas, que pueden poner en aprieto a quien siga este enfoque. Comenzar el diseño desde la prueba requiere destreza y, tal vez, se estén obviando esos conocimientos como parece

hacer Kent Beck, que nos presenta un método aparentemente sencillo, pasando por alto su bagaje profesional.

Al poner la prueba primero, ¿qué SUT hay que definir y con qué criterio?. En ese punto hace falta un mínimo de diseño. Tal vez por eso, Uncle Bob sugiere ir desarrollando el test y su código casi en paralelo. Según afirma Uncle Bob en su blog, en el año 99 hizo pairing con Kent Beck y vio cómo “escribía una línea de una prueba que fallaba y luego escribía la línea correspondiente del código de producción para hacerla pasar. A veces era algo más de una línea; pero la escala que utilizaba se acercaba mucho a la de línea por línea.” [78]. Todo esto no hace sino añadir más confusión a la falta de indefinición comentada anteriormente.

Igualmente, ambigua es la elección del primer test, mediante una especie de proceso intuitivo. El problema es que la elección del primer test puede hacer variar el resultado final. Parece que la recomendación de Kent Beck es hacer una especie de test de todo el sistema (un scaffolding o andamio) para trazar un boceto inicial del diseño y, a partir de ahí, continuar trabajando. Este test se desechará cuando el sistema avance.

Otro punto difuso es el tratamiento que se hace de las historias de usuario. Parece que en la lista de tests hay que añadir pruebas descritas desde el punto de vista del negocio y del usuario, lo cual resulta raro, puesto que estamos en la fase de diseño de bajo nivel y codificación y, en este punto, las pruebas se orientan a la verificación de las partes y no a la validación de los requisitos o del sistema.

Kent Beck apuesta por evitar las pruebas de aceptación (ATDD) debido a su alto coste y al tiempo que tardan en dar feedback. La alternativa es aumentar la cobertura de pruebas unitarias y mejorar el diseño, reduciendo la tasa de error. Aunque la pirámide de tests indique que el porcentaje de pruebas unitarias debe ser mayor que el de pruebas end-to-end, como afirma Bertrand Meyer en “Agile! The Good, the Hype and the Ugly”, se necesita una visión del sistema de más alto nivel, más allá de esa otra limitada, por lo que un tipo de pruebas no se puede sustituir completamente por el otro, aunque se trate de reducir su uso a la mínima expresión.

En cuanto a la alta tasa de cobertura, puede ser visto como ventaja o como inconveniente. Un mal diseño de tests conducirá a un número alto de pruebas rotas en la fase de refactoring. En cualquier caso, no se trata de lograr la cobertura de líneas, sino verificar el comportamiento del módulo que se está probando. Por lo tanto, TDD no implicaría una cobertura del 100% (a pesar de que Kent Beck utiliza esa métrica como medida de su éxito en el libro), pero, al poner la actividad de pruebas en primer lugar, la tasa será alta.

El debate de si TDD empuja a que las pruebas sean rápidas y aisladas del acceso a base de datos o si da igual, se podría dar por concluido, dado que las pruebas con base de datos en memoria son relativamente rápidas a día de hoy y, tal y como dice Martin Fowler, TDD no prescribe nada al respecto.

En cualquier caso, la idea de que un diseño que no se pueda testear puede ser un indicador de mal diseño, no implica que haya que hacer todas las partes del sistema testeable. Habrá situaciones en la que no se pueda probar o que sea demasiado costoso. En esos casos habrá que tomar una decisión de si merece la pena asumir el coste. Lo que debería evitarse, y aquí es donde es importante tener conocimientos amplios de diseño, es modificar el diseño en aras de la testeabilidad. Se trataría, más bien, de buscar la alternativa para probar o descartarla por el coste.



TDD tiene un enfoque “divide y vencerás”, lo que para Martin Fowler es una ayuda a la gente menos acostumbrada a trocear los problemas. La idea de aislar los problemas y mejorar la solución mediante el refactoring ayuda a simplificar y permite elaborar diseños más complejos cuando la solución se complica. El par pruebas-refactorización permite generar código más limpio de forma segura, transformando las soluciones radicalmente y de manera incremental.

Al priorizar el test e incidir en el refactoring, TDD recuerda continuamente la importancia del diseño, del que no podemos sustraernos. Pero de nuevo aparece la falta de concreción y contradicciones en este asunto. ¿TDD hace que emerjan el diseño de alto nivel o la arquitectura del sistema? ¿Es TDD una herramienta para diseñar o es necesario acudir a principios de diseño externos a este método?

A lo largo del libro, Kent Beck se empeña sistemáticamente en simplificar la disciplina de diseño y obviarla, a pesar de que él tiene un gran dominio, dada su experiencia. El diseño, según nos explica, parece reducirse a una serie de metáforas e historias con las que se van imaginando las interfaces públicas de las clases que consumirán los tests. No se busca anticipar patrones de diseño de forma conceptual, sino que se espera hasta la fase de refactoring y, de forma muy superflua, se deja que emerjan por sí mismos, casi sin buscarlos expresamente. Afirma que el libro de “Design Patterns” tiene un sesgo hacia el diseño conceptual, en lugar de hacia el refactoring y, acto seguido, simplifica los patrones de diseño de una manera naif. Sostiene que es mejor detectar los patrones desde la experiencia del refactoring que intentar diseñar así desde el principio y acaba rematando con la afirmación de que TDD puede verse como un método de implementación de patrones de diseño, lo que equivaldría a decir que se puede escribir el mítico libro de GoF por medio de TDD. El colofón lo tienen frases de este tipo: “Es mejor pensar en lo que se quiere que haga el sistema y dejar que el diseño se resuelva por sí mismo más adelante”.

Kent Beck parece pecar de ingenuo con esta visión en la que cualquiera, practicando TDD, podrá hacer emerger un diseño de manera incremental, hasta levantar el edificio del sistema al completo. En palabras de Bertrand Meyer: “Un proceso de software definido como la ejecución repetida de los pasos básicos de TDD -escribir una prueba, arreglar el código para que pase la prueba, refactorizar si es necesario- no puede tomarse en serio. Con un enfoque así, uno se limita a una visión de túnel, centrada en la última prueba. Un proceso eficaz requiere una perspectiva de alto nivel, que tenga en cuenta todo el sistema.” [63]. Uncle Bob acusa a la comunidad practicante de TDD de haber malinterpretado las palabras de Kent Beck, que siempre fueron en forma de metáfora. Lo cierto es que no queda claro cuál era su intención comunicativa, viendo sus afirmaciones en el libro.

Durante el debate, Martin Fowler llega a afirmar que TDD no garantiza buenos resultados si no se tiene experiencia diseñando. Además, Sandro Mancuso afirma en uno de sus artículos que tener una fase de refactorización no lo convierte en una herramienta de diseño [69]. La que queda más clara es la afirmación rotunda del propio Uncle Bob en una entrada en su blog, sosteniendo que la arquitectura no emerge. Defiende que es necesario hacer un diseño por adelantado, que sería el macro-diseño y TDD solo se centraría en el micro-diseño. Algo parecido comenta Sandro Mancuso cuando dice que TDD no hace consideraciones de diseño inicial, sino que este emerge del código, pero TDD no ayuda con el macro-diseño.

Además, en el debate, Kent Beck afirma que TDD empuja al diseño, pero el desarrollador es el responsable de tomar las decisiones adecuadas. Esta no es una tarea fácil, Martin Fowler reconoce que la fase de refactor depende de las destrezas y, por ejemplo, separar el interfaz de la implementación es una de las partes más difíciles del software cuando se trabaja con pequeñas partes.



Durante el debate aparecen contradicciones entre la conveniencia de aislar el código para poder probarlo y la afirmación de que TDD no prescribe ningún grado de aislamiento. Llama la atención cómo Emily Bache saca un estudio que afirma que la gente que es buena practicando TDD, sigue más de cerca la Inversión de Dependencia y el Principio de Responsabilidad Única, entre otros. Luego expone una kata en la que justifica, de manera forzada, el uso de una interfaz en aras de la testeabilidad del código, para luego acabar afirmando que las indirecciones añaden complejidad.

Ordenando toda esta amalgama de indefiniciones y contradicciones, se podría afirmar que la actividad de TDD se centra en las fases de codificación, pruebas y diseño de la solución final en el detalle, pero las fases de definición arquitectónica y diseño de alto nivel del sistema tienen que quedar resueltas previamente de alguna manera. Durante la codificación, en esas fases de rojo-verde-refactor, TDD tampoco facilita el diseño, tan solo recuerda que es necesario probar y pensar en la solución, utilizando alguna técnica de diseño. Es responsabilidad del desarrollador tener un conocimiento amplio y profundo de las disciplinas de diseño y pruebas, puesto que de él dependen las decisiones, ya que TDD no prescribe la granularidad de las pruebas o la facilidad / dificultad del testeo. En cualquier caso, se trataría de un método para gente que sabe diseñar bien, dado tienen la capacidad de decidir cuánto grado de aislamiento y qué tipo de pruebas serán necesarias.

Así expresado, TDD no sería más que lo que empuja al diseño continuamente, un acicate, el hilo rojo anudado en el dedo, que nos recuerda que las pruebas son una disciplina fundamental y el diseño algo complejo que hay que revisar continuamente. Todos los participantes del debate dejan claro la dificultad que este encierra y lo complejo que es establecer unos criterios claros de cuándo un diseño es bueno o malo. El diseño es clave en el éxito de un proyecto y es necesario adentrarse en los principios y dominarlos para conseguirlo. Tal vez Kent Beck ha evitado zambullirse en este asunto a lo largo del libro, dada su dificultad, pasando de puntillas por el tema con una simplificación que permita justificar el método.

#### Adaptabilidad, aplicabilidad y métricas

Las reglas que aparecen en el libro de TDD son demasiado simples o difusas y no contemplan de una manera clara cuáles son las excepciones al método y cómo adaptarlo a distintos contextos. Parece que la forma de adaptar TDD es eligiendo el tamaño de los tests. Con tests más grandes, esto es, que cubran más partes del sistema, los diseños serán más rápidos y superfluos. Cuando hay complejidad, habrá que hacer tests de grano fino y dar pequeños pasos. Gears puede que haya sido un intento de formalizar este asunto, aunque no parece haber tenido gran impacto o recorrido, dada la falta de referencias.

Otro asunto es el aumento de casos de pruebas que supone TDD. Sandro Mancuso opina que el TDD de Chicago es lento e ineficiente, con lo que da a entender que sí supone carga adicional al desarrollo. Tal vez, como afirma Kent Beck, las pruebas de grano más fino y la mayor cobertura estaría en las partes internas del sistema y, a medida que se avance hacia los límites externos del sistema, se relajarían los criterios.

Otro de los puntos sin clarificar es en qué contextos es bueno usar TDD y en cuáles no. Durante el debate se citan ejemplos en los que parece funcionar bien, como un convertidor de Markdown o un presentador de slides embebido en una web. Estos ejemplos coinciden con la definición de Kent Beck, cuando habla de situaciones en las que las entradas y las salidas son claras y el problema está aislado. No son grandes sistemas, sino pequeñas soluciones cuya funcionalidad está muy acotada. Otra de las propuestas es su uso para spikes, dado el carácter exploratorio que confiere TDD.

En el libro de TDD aparece la pregunta de si TDD sería aplicable a proyectos con GUI, seguramente por la época en la que se escribió el libro. Actualmente, debido al avance de la tecnología y los frameworks de pruebas, se ha abierto el espectro de proyectos en los que se podría aplicar TDD o, por decirlo de una manera más general, en los que se podrían aplicar pruebas unitarias. Existen experiencias y bibliografía sobre desarrollo de front-end con React usando TDD, por ejemplo. El avance de Javascript y Typescript y de los frameworks de pruebas asociados (con Jest, Cypress, etc) han facilitado este enfoque, además de que los proyectos de front siguen un ciclo de desarrollo completo y ya no son meras interfaces.

Por otra parte, durante el debate, Kent Beck llega a afirmar que no funciona con ciertos tipos de problemas y Martin Fowler confiesa que la mayoría de las cosas que hace no son con TDD. En legacy code con poca cobertura no es aplicable, dado que no hay un arnés de pruebas previo, con lo que no se puede hacer un refactoring con confianza. Sí se podría aplicar si se consigue aislar una modificación como una pieza de código nueva, que se implemente siguiendo el ciclo de TDD.

Tampoco parece recomendable usarlo en proyectos con un alcance largo, especialmente sin una arquitectura o diseño inicial, a no ser que se trate de un proyecto con carácter exploratorio que luego se desechará. Tal y como se comentó anteriormente, la arquitectura y diseño de alto nivel tienen que quedar suficientemente cerrados de manera anticipada para permitir que TDD se centre en el diseño e implementación en detalle de los problemas ya acotados y de alcance más limitado. En cualquier caso, en el arranque de un proyecto con TDD, conviene contar con gente experta en el diseño y disciplina de pruebas, ya que en otro caso el proyecto podría desembocar en el fracaso.

Llama la atención la frecuencia con la que TDD aparece relacionado con la Arquitectura Hexagonal. En el debate, por ejemplo, DHH ataca frecuentemente a este estilo arquitectónico. Además, actualmente se pueden encontrar bastantes charlas sobre TDD y Hexagonal [79], además de alguna bibliografía [80]. ¿Hay relación entre TDD y hexagonal? ¿Por qué aparece juntos frecuentemente?

El enfoque hexagonal parece tener cada vez mayor auge desde su aparición en el 2005, tal vez por la tendencia actual a buscar soluciones ubicuas y con micro-servicios. Da la sensación de que los proyectos parten con un stack predefinido con DDD (para modelar el dominio), Hexagonal (para independizar el dominio del resto del sistema), microservicios y TDD. Independientemente de si es acertada o no, la Arquitectura Hexagonal incide en el aislamiento del dominio del resto del sistema mediante los puertos y adaptadores, lo que facilita las pruebas debido a esas direcciones. Esto casaría bien con la idea citada anteriormente de buscar el aislamiento para hacer pruebas en TDD. De esta manera, se podría comenzar haciendo TDD para el dominio usando pruebas unitarias y, al llegar a los límites del dominio (los adaptadores), se harían pruebas de integración mediante el uso de mocks.

A pesar de que el enfoque Hexagonal aumenta el grado de indirección de manera notable, parece que la facilidad de pruebas que promete hace que el sector se decante por este tipo de soluciones combinadas. Si cada servicio está bien definido, la aplicación de TDD se reducirá a ese contexto acotado, con entradas y salidas bien definidas, reduciendo el riesgo de error a cada uno de los servicios aislados en contenedores, diseñados previamente. La elección de este stack solucionaría los problemas de qué arquitectura elegir, cómo diseñarla y cuánto grado de indirección es necesario para probarla. Esta colaboración ha supuesto un nuevo balón de oxígeno para TDD, que toma impulso una vez más, cobrando de nuevo actualidad, tal y como lo hizo con al calor del movimiento ágil.

En cuanto a las métricas, durante el debate se repite la idea de la dificultad para diseñar experimentos de software que sean medibles y repetibles, dada la cantidad de variables que intervienen. Esta falta de evidencia científica o estadística, muestra a su vez lo complejo que es el diseño de software. La conclusión es que no hay una manera clara de demostrar si TDD funciona o no, con lo que todo queda

circunscrito a la opinión y la experiencia concreta. Así, las métricas de cobertura y velocidad o número de ejecuciones en verde no dan pista de si el sistema es el correcto. Tampoco hay una forma de evaluar el grado de madurez de los profesionales usando TDD.

## Escuela de Londres

No queda claro en ningún sitio si el enfoque de TDD de Kent Beck se corresponde con el de Chicago. Incluso lo deja ambiguo, con su definición del “from unknown to known”, desmarcándose de ambos enfoques. En cualquier caso, se podría afirmar que su propuesta pasa por empezar en la lógica de negocio, idea que también llega a apoyar en algún momento de manera implícita, cuando habla de relajar los tests al llegar a las partes más externas del sistema. El modelo sería la parte más importante del sistema y de mayor independencia respecto de la tecnología o de agentes externos, por lo que sería más sencillo comenzar por ahí.

Aunque el dominio fuese el punto inicial, Kent Beck parece trazar una línea vertical con un primer test, que correspondería al andamio sobre el que va a ir levantando la solución. Este test traza un boceto completo del sistema, que se podrá descartar a medida que se avance en la estructura. El alzamiento de la estructura no tiene por qué ser completamente vertical, sino más bien se trataría de una especie de armazón, similar al de la Torre Eiffel, avanzando distintas partes que luego se irán ensamblando hacia arriba.

En cuanto al enfoque de Londres, la mayor crítica que se le hace es que los test de caja blanca son frágiles y están acoplados al código de producción. Sandro Mancuso sostiene que los mocks son una herramienta de diseño. Tal vez el debate se debería centrar en cómo se diseñan los límites del sistema, en lugar de si es conveniente usar o no los mocks, algo inevitable en algunos casos. Aunque la corriente de Londres esté más orientada al cliente y la interfaz que este percibe, útil por tanto para diseñar la parte externa del sistema, los tests de aceptación suelen tener un coste alto, con lo que en este enfoque se debe prestar sumo cuidado a la estrategia de pruebas.

## Trabajo futuro

Este trabajo abre una serie de interrogantes que podrían dar lugar a futuros trabajos de investigación para tratar de despejar esas incógnitas. A continuación, se enumeran algunas ideas:

1. Definición formal del proceso y guía de adaptación: Sería interesante trabajar en una definición formal de TDD en la que se especificase claramente cómo funciona el proceso en aspectos como la gestión de la lista de tests y su propósito, estrategias de elección de tests, proceso de diseño desde que se toma un test de la lista hasta que se refactoriza la solución, granularidad del proceso y ajustes y contextos en los que es aplicable y cómo adaptarlo.

Tal vez fuese interesante definir un marco de diseño de métricas que apoyase al proceso, esto permitiría tener una idea del grado de calidad del resultado y seguimiento del mismo. Si los tests automáticos ofrecen feedback, las métricas sistematizadas con herramientas podrían ofrecer una mayor solidez al proceso. El objetivo de este punto sería elaborar el método de obtención de métricas y no tanto las métricas específicas.

Como parte de estas métricas, sería interesante considerar especialmente aquellas que tienen que ver con el nivel de adherencia del proceso, para evaluar cuánto se sigue el método definido. Así, se podrían considerar métricas relativas al seguimiento del ciclo rojo-verde-refactor, a las ejecuciones de los tests y su frecuencia, etc.

2. Trayectoria curricular: Derivado del apartado anterior, se podría elaborar una trayectoria curricular en la que se incluyesen las disciplinas de diseño y testing de manera gradual mediante fases, lo que permitiría establecer una manera más clara de adquirir el conocimiento necesario para el ejercicio de este método. Además, podría servir de base para una evaluación de grado de profesionalidad, según las fases definidas en la trayectoria.
3. Base de código y estudios comparativos: Un último punto sería la generación de un repositorio de código que facilitase los estudios comparativos. Se trataría de ejercicios prácticos para construir sistemas reales y los problemas que esto conlleva. Estos sistemas quedarían registrados en repositorios de código abierto para su evaluación, lo que permitiría revisar los resultados y bondades del proceso y compararlo con otros métodos de trabajo.

## Bibliografía y referencias

- [0] Kent Beck, Test Driven Development: By Example. Addison-Wesley Professional, 2002
- [1] James E. McDonough, Automated Unit Testing with ABAP: A Practical Approach. Apress ,2021
- [2] Martin Fowler, XUnit. <https://martinfowler.com/bliki/Xunit.html>, 2006
- [3] Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts, Refactoring: Improving the Design of Existing Code. Addison Wesley, 1993
- [4] Robert C. Martin, Clean Architecture: A Craftsman's Guide to Software Structure and Design. Addison-Wesley, 2017
- [5] Robert C. Martin, What's all this Nonsense about Katas. <https://sites.google.com/site/unclebobconsultingllc/home/articles/what-s-all-this-nonsense-about-katas>, 2009
- [6] Dan North, Introducing BDD. <https://dannorth.net/introducing-bdd/>, 2006
- [7] Michael Feathers and Steve Freeman, Design, <https://www.infoq.com/interviews/feathers-free-man-design/>, InfoQ, 2010
- [8] Maciej Falski, Detroit and London Schools of Test-Driven Development. <https://blog.devgenius.io/detroit-and-london-schools-of-test-driven-development-3d2f8dca71e5>, 2020
- [9] Bill Wake, 3A – Arrange, Act, Assert. <https://xp123.com/articles/3a-arrange-act-assert/>, 2011

### Londo VS Chicago

- [10] Jim Newbery, Are you Detroit- or London-school?. <https://tinnedfruit.com/list/20181004>, 2018
- [11] Tomasz Jędrzejewski, London and Detroit schools of unit tests. <https://zone84.tech/architecture/london-and-detroit-schools-of-unit-tests/>, 2022
- [12] Hit Subscribe, London TDD Vs. Detroit TDD: You're Missing the Point. <https://blog.ncrunch.net/post/london-tdd-vs-detroit-tdd.aspx>, 2018
- [13] Geoff, Chicago and London Style TDD. <https://team-agile.com/2021/02/06/chicago-and-london-style-tdd/>, 2021
- [14] Sandro Mancuso, Does TDD really lead to good design? <https://www.javacodegeeks.com/2015/05/does-tdd-really-lead-to-good-design.html>, 2015
- [15] <https://www.adictosaltrabajo.com/2016/01/29/tdd-outside-in-vs-inside-out/>
- [16] Doug Klugh, London VS Chicago. <https://devlead.io/DevTips/LondonVsChicago>, 2019
- [17] Adrian Booth, Test Driven Development Wars: Detroit vs London, Classicist vs Mockist. <https://medium.com/@adrianbooth/test-driven-development-wars-detroit-vs-london-classicist-vs-mockist-9956c78ae95f>, 2018
- [18] Nikos Voulgaris, Comparing TDD flavours. <https://nvoulgaris.com/comparing-tdd-flavours/>, 2019
- [19] ChemaClass, London vs Chicago, It's an integration, not a choice. <https://chemaclass.com/blog/london-vs-chicago/>, 2021
- [20] Nico Paez, Estilos de TDD: London vs. Chicago. <https://blog.nicopaez.com/2020/11/15/estilos-de-tdd-london-vs-chicago/>, 2020
- [21] Sandro Mancuso, A Case for Outside-In Development. <https://www.codurance.com/publications/2017/10/23/outside-in-design>, 2017
- [22] Mladen Despotovic, Outside-In or Inside-Out? London or Chicago School? — Part 1: Greenfield Projects. <https://itnext.io/outside-in-or-inside-out-london-or-chicago-school-part-1-greenfield-projects-d324390a0dbd>, 2020

### Debate

- [23] Jim Coplien and Robert C. Martin, Debate TDD, CDD and Professionalism. <https://www.infoq.com/interviews/coplien-martin-tdd/>, 2008

- [24] Martin Fowler, Test Driven Development. <https://martinfowler.com/bliki/TestDrivenDevelopment.html>, 2005
- [25] Ian Cooper, TDD, Where Did It All Go Wrong. <https://www.youtube.com/watch?v=EZ05e7EMOLM>, 2017
- [26] TDD Buddy, Gears. <http://www.tddbuddy.com/reference/TDD%20Gears.pdf>
- [27] Emily Bache, Is TDD dead? Of course not! But what's all the fuzz about then?. <https://www.youtube.com/watch?v=PCEHRFKZSk&list=PLOGzxujsqdGDpW8mHsQwBBYVYR2-9GX7u&index=10>, 2014
- [28] Robert C. Martin, What's all this Nonsense about Katas?. <https://sites.google.com/site/unclebobconsultingllc/home/articles/what-s-all-this-nonsense-about-katas>, 2009
- [29] Dave Thomas, CodeKata: How It Started. <http://codekata.com/kata/codekata-how-it-started/>, 2013
- [30] Robert C. Martin, The programming dojo. <http://www.butunclebob.com/ArticleS.UncleBob.TheProgrammingDojo>, 2005
- [31] Dave Thomas, Agile is Dead (Long Live Agility). <https://pragdave.me/thoughts/active/2014-03-04-time-to-kill-agile.html>, 2014
- [32] Dave Thomas, Agile is Dead, GOTO 2015. <https://www.youtube.com/watch?v=a-BOSpxYJ9M>, 2015
- [33] Kent Beck, Guide to Better Smalltalk: A Sorted Collection. SIGS, 1998 (Ver: <https://www.amazon.com/Kent-Becks-Guide-Better-Smalltalk/dp/0521644372>)
- [34] Chris Morris, <https://wiki.c2.com/?ChrisMorris>, 2014
- [35] Vladimir Khorikov, Unit Testing Principles, Practices, and Patterns. Manning, 2020
- [36] Dan North, Introducing BDD. <https://dannorth.net/introducing-bdd/>, 2006
- [37] Markus Gärtner, ATDD by Example: A Practical Guide to Acceptance Test-Driven Development. Addison Wesley, 2012
- [38] James Shore, The problems with acceptance testing. <https://www.jamesshore.com/v2/blog/2010/the-problems-with-acceptance-testing>, 2010
- [39] Mike Cohn, The forgotten layer of the test automation pyramid. <https://www.mountaingoatsoftware.com/blog/the-forgotten-layer-of-the-test-automation-pyramid>, 2009
- [40] David Heinemeier Hansson, Test-induced design damage. <https://dhh.dk/2014/test-induced-design-damage.html>, 2014
- [41] Martin Fowler, Self-testing code. <https://martinfowler.com/bliki/SelfTestingCode.html>, 2014
- [42] Approval tests. <https://approvaltests.com/>
- [43] Ham Vocke, The Practical Test Pyramid, <https://martinfowler.com/articles/practical-test-pyramid.html#End-to-endTest>, 2018
- [44] Martin Fowler, C3. <https://www.martinfowler.com/bliki/C3.html>, 2004
- [45] David Thompson, Comp.benchmarks. <https://pages.cs.wisc.edu/~thomas/comp.benchmarks.FAQ.html>, 1997
- [46] Mike Bland, Goto Fail, Heartbleed, and Unit Testing Culture. <https://martinfowler.com/articles/testing-culture.html>, 2014
- [47] Martin Jee, What is Programmer Anarchy and does it have a future?. <https://martinjeeblog.com/2012/11/20/what-is-programmer-anarchy-and-does-it-have-a-future/>, 2012
- [48] Fred George, Programmer Anarchy. <https://www.infoq.com/news/2012/02/programmer-anarchy/>, 2012
- [49] Fred George, Programmer Anarchy. <https://www.youtube.com/watch?v=uk-CF7kLdA>, 2012
- [50] Dave Thomas and Andy Hunt, The Pragmatic Programmer-your journey to mastery. Addison Wesley, 19
- [51] Rebecca Wirfs-Brock, Responsibility-Driven Design. <https://www.wirfs-brock.com/Design.html>
- [52] Rebecca Wirfs-Brock, Responsibility-Driven Design. <https://www.wirfs-brock.com/PDFs/Responsibility-Driven.pdf>

- [53] Responsibility-Driven Design. <https://www2.cs.arizona.edu/~mercer/Presentations/OOPD/12-RDD-Jukebox.pdf>, 2014
- [54] Martin Fowler, Patterns of Enterprise Application Architecture. Addison-Wesley, 2002
- [55] Ruby on Rails, Active Record Basics. [https://guides.rubyonrails.org/active\\_record\\_basics.html](https://guides.rubyonrails.org/active_record_basics.html)
- [56] Bill Karwin, SQL Antipatterns: Avoiding the Pitfalls of Database Programming, The Pragmatic Programmers, LLC, 2017
- [57] David Heinemeier Hansson, Test induced design damage. <https://gist.github.com/dhh/4849a20d2ba89b34b201>, 2014
- [58] Jim Weirich, Decoupling from Rails. <https://www.youtube.com/watch?v=tg5RFeSfBM4>, 2013
- [59] Kevlin Henney, "All problems in CS ...". <https://twitter.com/drunkcod/status/242591688797855745>, 2012
- [60] Emily Bache, TirePressure Kata. <https://github.com/emilybache/TirePressure-Kata>
- [61] Luca Minudel, TDD with Mock Objects: Design Principles and Emerging Properties. <https://github.com/lucaminudel/TDDwithMockObjectsAndDesignPrinciples>
- [62] Kent Beck, Stack Overflow answer. <https://stackoverflow.com/questions/153234/how-deep-are-your-unit-tests/153565#153565>, 2008
- [63] Bertrand Meyer, Agile! The Good, the Hype and the Ugly. Springer, 2014
- [64] Alan Mellor, Test-Driven Development with Java. Create higher-quality software by writing tests first with SOLID and hexagonal architecture. Packt. 2023
- [65] Valentina Cupac, TDD and Hexagonal Architecture in Microservices. <https://www.youtube.com/watch?v=-620eHMEH-w>, 2023
- [66] Robert C. Martin, TDD Harms Architecture. <https://blog.cleancoder.com/uncle-bob/2017/03/03/TDD-Harms-Architecture.html>, 2017
- [67] Siniaalto, M., & Abrahamsson P. , A Comparative Case Study on the Impact of Test-Driven Development on Program Design and Test Coverage. ESEM, First International Symposium on IEEE. 2007
- [68] Harender Singh Dhanoya, Estudio del TDD aplicado en el desarrollo del Mastermind. <https://github.com/MasterCloudApps-Projects/TDD-Mastermind#331-experimentos-en-ibm-y-microsoft>, 2022
- [69] Sandro Mancuso, Does TDD lead to a good design?. <https://www.codurance.com/publications/2015/05/12/does-tdd-lead-to-good-design>, 2015
- [70] David Heinemeier Hansson, RailsConf 2014 - Keynote: Writing Software. <https://www.youtube.com/watch?v=9LfmrkyP81M>, 2014
- [71] James O Coplien, Why Most Unit Testing is Waste. <https://rbc-us.com/documents/Why-Most-Unit-Testing-is-Waste.pdf>, 2014
- [72] Sandro Mancuso, Mocking as a Design Tool. <https://www.codurance.com/publications/2018/10/18/mocking-as-a-design-tool>, 2018
- [73] Emily Bache, The Coding Dojo Handbook. Emily Bache, 2013
- [74] Dan North, Deliberate Learning. <https://www.youtube.com/watch?v=SPj-23z-hQA>, 2012
- [75] Cyber- dojo. <https://cyber-doj.org/creator/home>

## Conclusions

- [76] Elena Gaviria Stewart, Mercedes López Sáez, Isabel Cuadrado Guiraldo, Introducción a la Psicología Social. Ed. Sanz y Torres, 2019
- [77] Arthur Schopenhauer, El arte de tener razón. Alianza Editorial, 2020
- [78] Robert C. Martin, The Cycles of TDD. <https://blog.cleancoder.com/uncle-bob/2014/12/17/TheCyclesOfTDD.html>, 2014
- [79] Valentina Cupac, TDD and Hexagonal Architecture in Microservices. <https://www.youtube.com/watch?v=-620eHMEH-w>, 2023
- [80] Alan Mellor, Test-Driven Development with Java: Create higher-quality software by writing tests first with SOLID and hexagonal architecture. Packt, 2023

