

TEST DRIVEN DEVELOPMENT

Algunos conceptos previos necesarios para entrar a TDD

Abstracción

Es la habilidad de manejar un concepto ignorando detalles irrelevantes. La abstracción nos permite manejar un objeto complejo de forma simple.

Interfaz e implementación

La interfaz es todo lo que está permitido ver y hacer con un objeto.

La implementación son los detalles que hacen a la interfaz pero que no son relevantes para el mundo exterior.

Clases

Son abstracciones de datos y comportamiento, que nos permiten reducir los niveles de complejidad.

Si la clase no está haciendo nuestra vida más sencilla, no está haciendo bien su trabajo.

Self Testing Code

El código “auto testeado” es código que nos permite corroborar su funcionamiento correcto mediante un set de tests automatizados que ejecutan contra el código fuente de nuestra app.

Self Testing Code

El código “auto testeado” nos permite trabajar con CONFIANZA en nuestro sistema sabiendo que cualquier potencial fallo, introducido en un cambio, puede ser detectado mediante la ejecución de pruebas automatizadas.

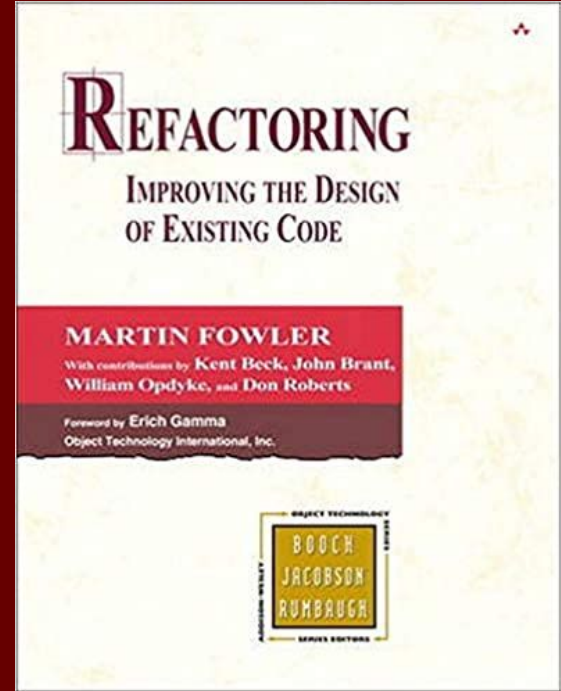
Refactoring

Es una técnica cuyo objetivo es el de mejorar la estructura interna de nuestro código sin modificar el comportamiento visible del mismo.

Refactoring

En otras palabras, Refactoring NO agrega funcionalidad, si no que mejora nuestro sistema haciendo nuestro código más entendible, mantenible y modificable.

Refactoring



Es importante la calidad en software?

- 1) Qué entendemos por calidad en software?**
- 2) Es algo a lo que debemos apuntar? Tiene valor?**

Qué entendemos por calidad? En SW

- 1) Usabilidad
- 2) Confiabilidad
- 3) Tolerancia a fallos
- 4) Performance
- 5) Modificabilidad
- 6) etc

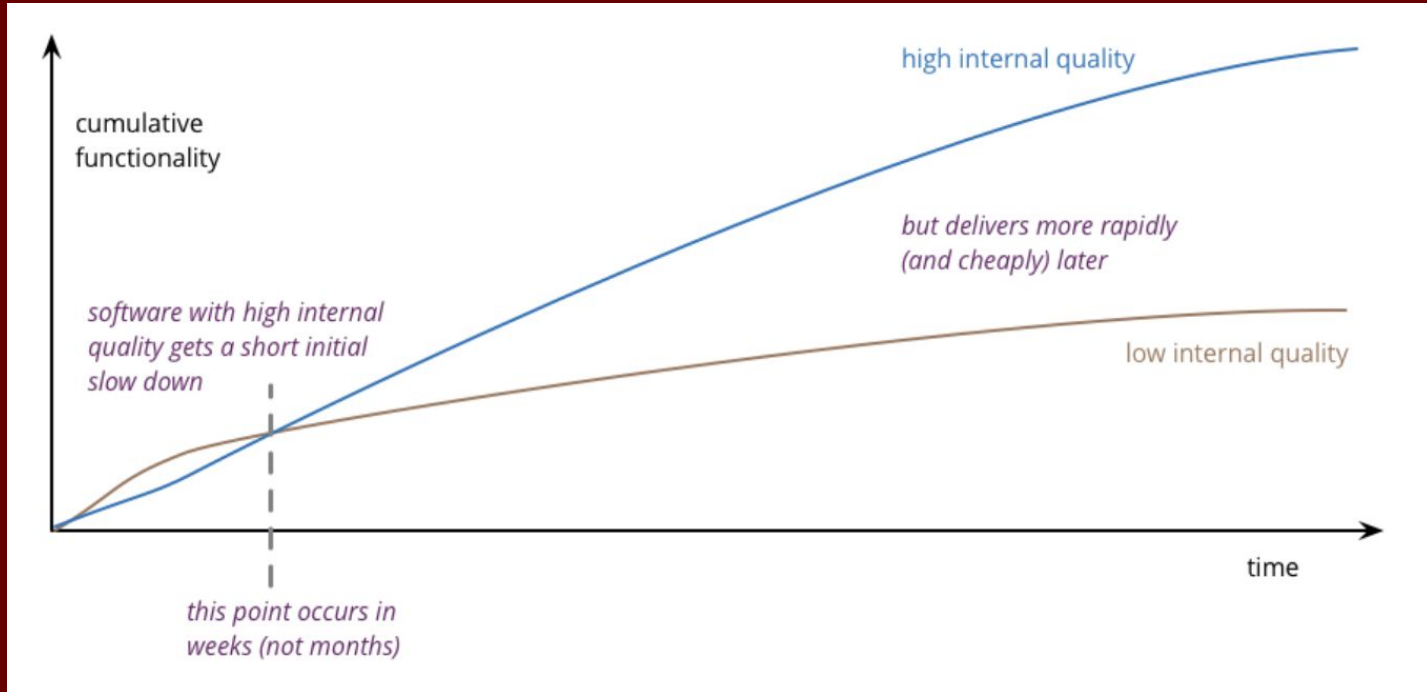
Qué entendemos por calidad? En SW

Básicamente, atributos visibles y no visibles (o externos e internos)

Qué sucede con los atributos internos?

Objetivamente... son importantes? Si el usuario no lo percibe, importan?

Qué sucede con los atributos internos?



Por qué testear? Es importante?

Como programadores debemos entender que el software se degrada. Desde el momento en el que ponemos el sistema en funcionamiento el mismo se enfrenta a fuerzas que “lo desgastan” y lo hacen cambiar.

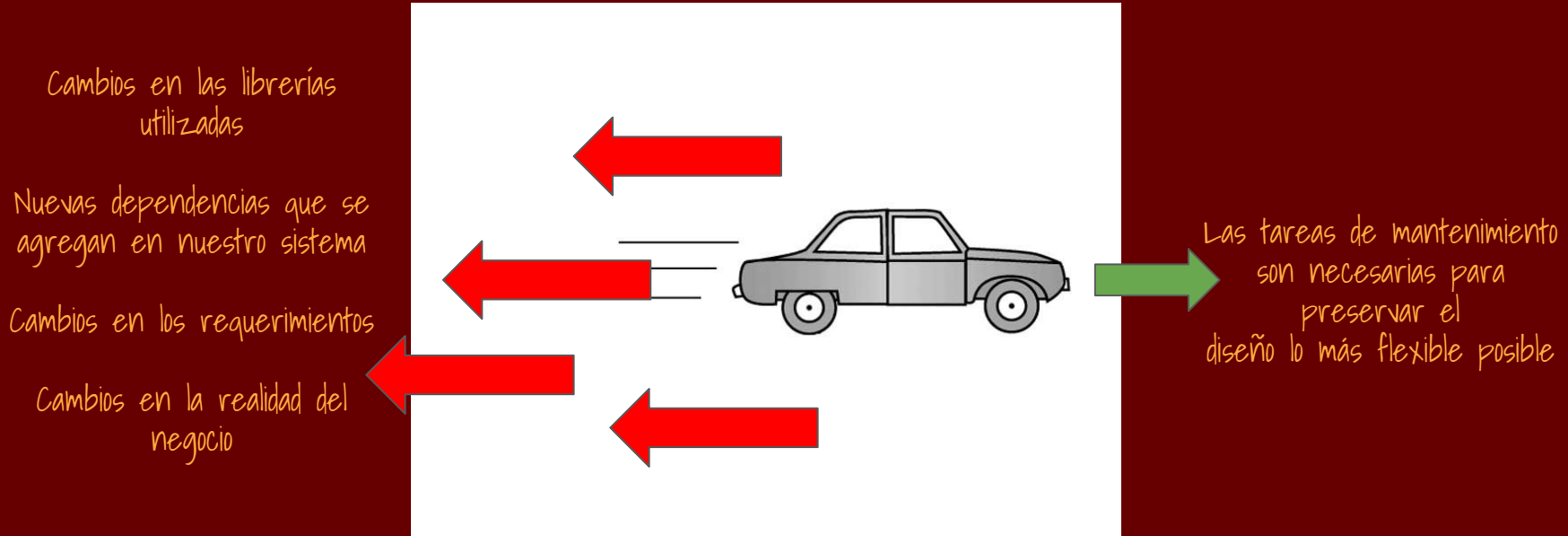
Por qué testear? Es importante?

Si el software nunca cambiase sencillamente casi no tendría valor escribir pruebas. Pero tampoco sería software. Sería hardware.

Por qué testear? Es importante?

Así como llevamos nuestro auto al mecánico para cambiar alguna pieza que funciona incorrectamente o para realizar mantenimiento, lo mismo sucede con el software

Cómo es que el software se degrada



Por qué testear? Es importante?

**Tener un buen set de test que nos respalde
nos permitirá trabajar con CONFIANZA
cuando tengamos que tomar las decisiones
adecuadas en pos del mantenimiento de
nuestro sistema.**

No me resulta natural escribir un
test antes que escribir el código

Voy más lento con TDD que sin
TDD

POR QUÉ DAMOS TDD?

No conozco a nadie en mi trabajo
que use TDD

Google+

David Heinemeier Hansson
@dhh

para conversar sobre

0:09 / 30:25

David

TW Hangouts | Is TDD dead?



The image is a screenshot of a Google+ Hangout video player. The main video frame shows a man with short brown hair and a beard, wearing a grey t-shirt and white earbuds, speaking. In the top left corner of the video frame is the Google+ logo. In the bottom left corner of the video frame is a small circular profile picture of the same man. A white banner with a blue border is positioned over the bottom of the video frame, displaying the name 'David Heinemeier Hansson' and the handle '@dhh'. Below the video frame, there is a black subtitle bar with the text 'para conversar sobre' in white. To the right of the subtitle bar is a small, colorful icon consisting of many small circles. Below the subtitle bar is a video progress bar with a play button, a volume icon, and the text '0:09 / 30:25'. To the right of the progress bar is a small video thumbnail showing the same man, and to the right of that is the name 'David'. The entire video player is set against a dark red background.

A MUY ALTO NIVEL...

Es una metodología de desarrollo y diseño de software en el que los requerimientos se convierten en casos de prueba verificables.

A MUY ALTO NIVEL...

Se escribe el mínimo código necesario para que las pruebas pasen y NO MÁS. Luego se itera para mejorar la estructura de dicho código.

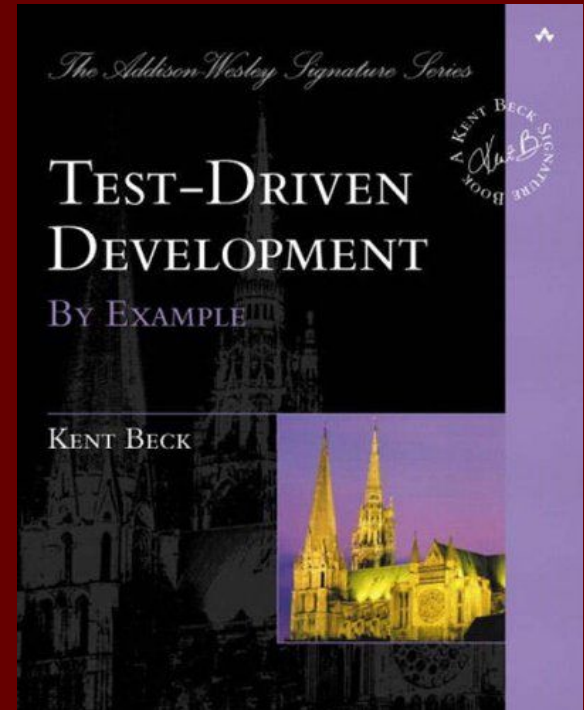
Test Driven Development trae como consecuencia que el código que escribimos es *Self Testing Code*, pero no es la única forma de conseguirlo.

Hacer TDD implica escribir pruebas unitarias

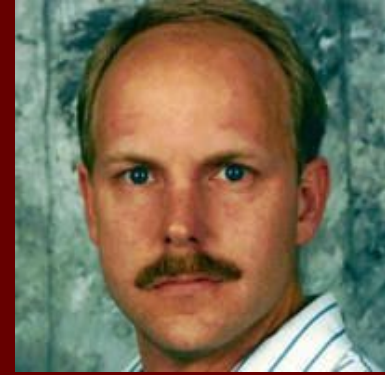


**hacer pruebas unitarias no implica que
estemos haciendo TDD**

KENT BECK



BENEFICIOS? por Kent Beck



- **TDD lleva a producir diseños más sencillos**
 - **TDD inspira confianza (esto es una consecuencia evidente de tener pruebas)**

Qué es un diseño sencillo

- Todos sus tests pasan
- No contiene código duplicado (ojo, peligroso)
- Revela su intención claramente
- Mínimo número de clases y métodos




CÓMO FUNCIONA...

Escribimos
una prueba

La prueba verifica una funcionalidad

Se escribe
el código 

Solamente nos importa que la prueba pase, el código puede estar desprolijo e incluso la implementación no ser la mejor

Corremos las
pruebas y
pasan 

El código que agregamos arregló la prueba, además verificamos que no rompimos funcionalidad existente

Corremos las
pruebas y
esta falla 

Esto nos permite saber que no escribimos la prueba mal y que funciona sin importar el código que agreguemos luego

Refactoreamos
el código 

Mejoramos la estructura interna de lo que escribimos. Lo hacemos más entendible y mantenible. Las pruebas NO DEBEN ROMPERSE

CÓMO FUNCIONA...

Escribimos
una prueba

Corremos las
pruebas y
esta falla




Esto nos permite saber que no escribimos la prueba mal y que funciona sin importar el código que agreguemos luego

La prueba verifica una funcionalidad

Se escribe
el código 

Solamente nos importa que la prueba pase, el código puede estar desprolijo e incluso la implementación no ser la mejor

Corremos las
pruebas y
pasan 

El código que agregamos arregló la prueba, además verificamos que no rompimos funcionalidad existente

Refactoreamos
el código



Mejoramos la estructura interna de lo que escribimos. Lo hacemos más entendible y mantenible. Las pruebas NO DEBEN ROMPERSE



Incluso en las veces en las que no sé cómo implementar una funcionalidad, si puedo pensar una prueba que verifique la misma

Escribir el mínimo código necesario para que nuestros tests pasen produce diseños e implementaciones sencillas. Esto se relaciona con otros principios conocidos.

KISS (Keep it simple, stupid), establece que los sistemas funcionan mejor si se mantienen sencillos en lugar de complicarlos innecesariamente.

YAGNI (You aren't gonna need it), establece que el programador no debe agregar funcionalidad hasta que sea necesaria y apunta a buscar la solución más simple que pueda funcionar.

Escribir primero el test nos obliga a pensar primero la *interfaz* necesaria que debe tener nuestro código para que el test pase, en lugar de pensar en términos de la *implementación*.

**Pensar en términos de cómo vamos a usar
nuestra clase/módulo/código nos facilita
separar *interfaz* de *implementación*.**

DEJAR DE LADO EL EGO...

“Es mejor programar algo sencillo y quedar como un tonto que querer parecer inteligente y programar algo innecesariamente complicado”


Qué tanto UML?

Nunca perder de vista que UML es una herramienta para comunicar, no es el diseño final del sistema.

ESTRUCTURA DE NUESTROS TESTS

1. **SETUP** (preparar el contexto)
2. **EXERCISE** (ejecutar la prueba)
3. **VERIFY** (verificar si el test pasa o no)
4. **CLEANUP o TEARDOWN** (limpiar el contexto)

Mentalidad TDD

Pensar en términos de qué tests necesito tener pasando  para verificar que tengo implementado el comportamiento que especifican los requerimientos.

***Sugerencia:* Escribir una lista de los requerimientos que necesito y los tests que los verifican**

Mentalidad TDD

No adelantarme a pensar en *cómo* lo voy a implementar, si no en *qué* interfaz necesito para escribir mi test. Pensar en una interfaz ideal, soñada 🦄🌈

Mentalidad TDD

Escribir tests para todos los comportamientos posibles. No quedarme solamente con los *happy path*. Considerar los escenarios con errores, en que las cosas pueden salir mal.

Mentalidad TDD

Si encontramos un bug, o un escenario no contemplado, escribir un test que confirme su presencia (es decir, un test que inicialmente falle).

Luego corregir el defecto y verificar que el test ahora pasa.

Buenas prácticas

- Los tests son aislados, la ejecución de uno no afecta la de otro.
- Los tests se corren de forma aleatoria, el resultado no puede depender del orden.
- Los tests también son parte del código fuente.

Referencias

- <https://martinfowler.com/articles/is-quality-worth-cost.html>
- <https://martinfowler.com/articles/designDead.html>
- **Code Complete, Steve McConnell**