

Taller de Test Driven Development

Pablo Orduña (aka NcTrun)



Con la colaboración de Iker Larizgoitia



This work is licensed under the Creative Commons Attribution-ShareAlike License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/2.0/> or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA



Introducción

Test Driven ...

Continuous Integration

Conclusiones

Referencias

Página www

Página de Abertura



Página 1 de 23

Regresar

Full Screen

Cerrar

Abandonar

1. Introducción

1.1. ¿De qué va este taller?

- Es un taller práctico
 - Aunque hay “un poquillo” de chapa al principio ;-)
- Es un taller de Test Driven Development
 - Pero enmarcado dentro de una muy breve introducción de Metodologías Ágiles de Desarrollo
 - No es “un taller de JUnit”
- Pero, sobre todo, es un taller informal
 - Es la primera vez que doy algo de esto
 - Tengo *muy poca* experiencia en el tema
 - Me encantaría oír “pero”s a lo que voy a explicar
- Conclusión → *por favor, participad* ;-)



Introducción

Test Driven ...

Continuous Integration

Conclusiones

Referencias

Página www

Página de Abertura

◀ ▶

◀ ▶

Página 2 de 23

Regresar

Full Screen

Cerrar

Abandonar



1.2. Metodologías de desarrollo Ágil

1.2.1. Metodologías de desarrollo tradicionales

- Tradicionalmente:
 - Se compara el desarrollo de software con la construcción de edificios
 - * Las fases de codificación y pruebas parecen algo “mecánico”
 - Nos centramos únicamente en arquitecturas, requisitos, diseño. . .
 - * Programar → tarea “de bajo nivel”
 - * Diseño → tarea que no necesita experiencia en programación
 - Modelo en cascada
 - * Pocas iteraciones, y muy grandes, que cada una incluye:
 1. Análisis
 2. Diseño
 3. Implementación
 4. Pruebas
 - Producir cambios en las fases de implementación y pruebas sugiere un mal diseño o planificación
 - * Se barajan pequeñas tasas de cambio *aceptables*
 - lo que esté fuera por tanto será ¿inaceptable?
 - * Nos frustramos cuando tenemos que cambiar el diseño de nuestras aplicaciones
 - Pensamos que se ha hecho mal el diseño (lo hayamos hecho nosotros u otros)
 - Nos da miedo modificar código que hace meses que no tocamos
 - Pensamos que hemos hecho mal en no haber documentado al detalle este código que ahora nos da miedo modificar

Introducción

Test Driven . . .

Continuous Integration

Conclusiones

Referencias

Página www

Página de Abertura



Página 3 de 23

Regresar

Full Screen

Cerrar

Abandonar



- Una pesada integración sugiere un mal diseño
 - * Damos por hecho que sin un diseño detallado de todo, la integración será un poco caos
 - * A veces se convierte en un proceso pesado, impredecible, que no sabemos qué impacto tendrá
- Y sin embargo. . . **funciona**:
 - En gran cantidad de proyectos en los que esta forma de trabajar funcionará:
 - * Dependiendo de si la tecnología es bien conocida por el equipo
 - * Dependiendo de cuánto de cambiante sea la tecnología
 - * Dependiendo del tamaño del equipo
 - * Dependiendo de la formación del equipo
 - * Dependiendo de si tenemos competencia directa o no
 - * Dependiendo de la dedicación del equipo (más proyectos?)
 - * Dependiendo de muchos más factores. . .

[Introducción](#)

[Test Driven . . .](#)

[Continuous Integration](#)

[Conclusiones](#)

[Referencias](#)

[Página www](#)

[Página de Abertura](#)



[Página 4 de 23](#)

[Regresar](#)

[Full Screen](#)

[Cerrar](#)

[Abandonar](#)



1.2.2. Metodologías de desarrollo ágil

- Pero no todo son clavos para nuestro martillo:
 - Existen situaciones que pueden dar problemas:
 - * Competidores que desarrollan el mismo software para los mismos clientes → tendremos que **adaptarnos** rápidamente para ofrecer las mismas características. . .
 - Una metodología en la que tenemos que terminar la larga iteración actual para obtener estos nuevos requisitos, y en el que no ofreceremos resultados hasta terminar la siguiente larga iteración puede no ser adecuado
 - * Si utilizamos tecnologías que cambian mucho, o en el que salen nuevas cada poco tiempo, y que afectaría muy positivamente a nuestro proyecto si nos **adaptásemos** a ellas, o que incluso no **adaptarnos** nos penalizaría. . .
 - Una metodología que critica el cambio o propone esperar hasta la siguiente iteración para adaptarlas puede no ser adecuado
 - * Si el proyecto tiene un único cliente y entiende el riesgo que hay en no ofrecer resultados hasta pasados años. . .
 - Una metodología que únicamente confía en las grandes iteraciones puede no ser adecuado, frente a una que proponga muchas pequeñas iteraciones que vayan poco a poco **adaptándose** a los requisitos del cliente, y aceptando los cambios que vaya proponiendo
 - En general, en proyectos *exploratorios*, las metodologías de desarrollo *rigurosas* pueden presentar graves problemas
 - * **Proyectos exploratorios** → *Exploratory projects are frontier (research-like), mission critical, time driven, and constantly changing* (Jim Highsmith, *Agile Software Development Ecosystems*, 2002).
 - * Las metodologías *rigurosas* pueden intentar adaptarse a estas situaciones proponiendo diseños más generales
 - Suficientemente general para cambios que no conocemos en la fase de diseño
 - Terminamos generando mucho más software del que necesitamos → necesitando más tiempo para desarrollar cosas que nunca se usarán (*bloatware*)
 - Si las iteraciones se mantienen grandes, hay situaciones ante las que no podrán responder

Introducción

Test Driven . . .

Continuous Integration

Conclusiones

Referencias

Página [www](#)

Página de Abertura

◀ ▶

◀ ▶

Página 5 de 23

Regresar

Full Screen

Cerrar

Abandonar



- Dando la vuelta a la tortilla:
 - Si el cambio no es tan malo en muchas situaciones → ¿Por qué no aprovecharnos de él?
 - * Si fuésemos capaces de adaptarnos rápidamente a nuevos cambios según llegan nuevas oportunidades...
 - ¿Podrán nuestros competidores **adaptarse** a nuestros cambios?
 - ¿Por qué no ofrecemos versiones periódicas a nuestros clientes y nos adaptamos a los cambios que ellos vayan deseando? ¿No se ajustará mejor el resultado final a lo deseado por el cliente?
Explicando claramente cuál es el impacto de los cambios que el cliente desea
 - Metodologías de Desarrollo Ágil
 - * Conjunto de metodologías que buscan otras formas de desarrollar software:
 - Proponen iteraciones cortas (de una a cuatro semanas dependiendo de la metodología)
 - Proponen diferentes técnicas dependiendo de la tecnología
 - * Sus valores de fondo se transmiten a través del Agile Manifesto <http://agilemanifesto.org/>:
 - Firmado en 2001 por sus fundadores
 - * Encontramos varias formas:
 - Extreme Programming (XP)
 - Scrum
 - Adaptive Software Development
 - Feature Driven Development
 - ...
- Pero sin pasarnos:
 - No hay una bala de plata o martillo de oro que nos vaya a servir en todas las ocasiones

Introducción

Test Driven ...

Continuous Integration

Conclusiones

Referencias

Página www

Página de Abertura



Página 6 de 23

Regresar

Full Screen

Cerrar

Abandonar

- La solución que tomemos dependerá de muchos factores:
 - * Lo que funcionó en una empresa no tiene por qué funcionar en otra
 - * Lo que funcionó en un grupo no tiene por qué funcionar en otro
- Y recordemos que yo ni tengo experiencia ni nada para juzgar nada de esto 0:-D



Introducción

Test Driven ...

Continuous Integration

Conclusiones

Referencias

Página www

Página de Abertura



Página 7 de 23

Regresar

Full Screen

Cerrar

Abandonar



- Espera... ¿de qué decías que iba este taller?
 - Quiero que el taller sea práctico → no quiero centrarme en metodologías de desarrollo ágil, sino en dos prácticas (Test Driven Development y Continuous Integration) de una de las metodologías de desarrollo ágil más popular (Extreme Programming)
 - Sin embargo, me parece imprescindible explicar -aunque haya sido brevemente- qué valores y qué principios están detrás de estas prácticas



Introducción

Test Driven ...

Continuous Integration

Conclusiones

Referencias

Página www

Página de Abertura



Página 8 de 23

Regresar

Full Screen

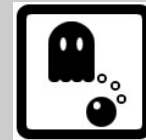
Cerrar

Abandonar

2. Test Driven Development

2.1. ¿Qué es?

- Test Driven Development es:
 - Como hemos explicado, una práctica de al menos una metodología de desarrollo ágil, Extreme Programming
 - Consiste en desarrollar guiado por pruebas
 - * Bajo iteraciones cortas tal que:
 1. Creamos un test que exige una nueva funcionalidad o característica
 2. Ejecutamos los tests y vemos que ese test falla (luz roja)
 3. Añadimos código que haga que funcione
 4. Ejecutamos los tests y vemos que los tests funcionan (luz verde)
 5. Refactorizamos nuestro código
- Ejemplo rápido: `codigo/01_Acumulador`.



[Introducción](#)

[Test Driven ...](#)

[Continuous Integration](#)

[Conclusiones](#)

[Referencias](#)

[Página www](#)

[Página de Abertura](#)



[Página 9 de 23](#)

[Regresar](#)

[Full Screen](#)

[Cerrar](#)

[Abandonar](#)



[Introducción](#)

[Test Driven ...](#)

[Continuous Integration](#)

[Conclusiones](#)

[Referencias](#)

[Página www](#)

[Página de Abertura](#)



[Página 10 de 23](#)

[Regresar](#)

[Full Screen](#)

[Cerrar](#)

[Abandonar](#)

1

ROJO

Escribir un pequeño test que no funcione, incluso igual que ni compile



2

VERDE

Hacer que el test funcione , cometiendo los pecados que hagan falta

3

REFACTORIZAR

Eliminar duplicados y malas artes que se cometieron para que funcionara el test





2.1.1. Motivación

- ¿Hacer pruebas antes del código? ¿Para qué?
 - Diseño desacoplado → difícil de diseñar código acoplado si primero estás haciendo el test de cada unidad
 - Confiamos más en nosotros mismos → vemos que nuestro código hace lo que decimos que haga
 - Podemos refactorizar mejor → nos atrevemos a probar diferentes diseños sin miedo a romper algo
 - * Si rompiésemos algo, algún test nos avisaría
 - * Unido a gestores de control de versiones, no dudamos
 - * Incluso podemos hacerlo pasados meses desde que hicimos el código!
 - En todo momento podemos tener una idea del estado del proyecto
- ¿Cuánto hacer?
 - ¿Cuánto es demasiado testing? Pregunta con trampa → ¿Cuánto es **demasiado** buen diseño?
 - * Obviamente podemos estar gastando *demasiado dinero* o *demasiado tiempo* en testing
 - * Hay formas de detectar que hemos hecho suficientes pruebas para una función concreta
 - * Pero eso no significa que estemos *probando demasiado*
- ¿Y de qué me sirve maximizar la cantidad de código testeado?
 - *¿Funcionará mi código con esta nueva versión de este runtime? ¿Y en esta nueva plataforma? ¿Y con esta nueva versión de la librería?* → lanza tus tests y compruébalo
 - Documentación del código → los tests definen **qué** hace el código funcional (no **cómo**). Manteniéndolos actualizados, son una documentación muy valiosa de cada función.

Introducción

Test Driven ...

Continuous Integration

Conclusiones

Referencias

Página www

Página de Abertura



Página 11 de 23

Regresar

Full Screen

Cerrar

Abandonar



– Cazar fallos → cuesta más arreglar un fallo de código que hicimos ayer que arreglar un fallo de código que acabas de hacer.

- Refactorizando código

– Refactorizar → Mejorar el diseño de código existente

* ¿Pero eso no lo hace el IDE?

· Los entornos de desarrollo cuentan con funcionalidades que permiten realizar refactorizaciones automáticamente

Extraer método, clase, renombrar método/clase/atributo/variable, mover estructuras o funcionalidades en una jerarquía de clases. . .

· Sin embargo la refactorización va más allá → no siempre automatizable

* ¿Qué tiene que ver el tener tests para refactorizar código?

· Si tienes tests que cubren todo tu código → refactorizas más fácil y rápidamente

· Si no tienes tests que cubren todo tu código → riesgo

¿Cuántas veces hemos oído *si funciona, no lo toques?*

– Comparación: `codigo/02_RefactoringInicial` y `codigo/03_RefactoringFinal`.

Introducción

Test Driven . . .

Continuous Integration

Conclusiones

Referencias

Página [www](#)

Página de Abertura

◀▶

◀▶

Página 12 de 23

Regresar

Full Screen

Cerrar

Abandonar



2.2. Introducción a JUnit

- xUnit
 - Muchas plataformas de pruebas unitarias utilizan una arquitectura denominada *xUnit*
 - * Primera implementación → SUnit (Smalltalk), por Kent Beck (cocreador de Extreme Programming)
 - Esta arquitectura informal define una serie de métodos y estructuras que cada implementación de cada lenguaje adapta al lenguaje
 - * JUnit para Java
 - * csUnit y NUnit para .NET
 - * PyUnit para Python
 - * [Y muchos más...](#)
 - Relación con Test Driven Development:
 - * Test Driven Development utiliza plataformas de pruebas unitarias
 - * Algunas de estas plataformas son compatibles con xUnit
 - * Test Driven Development podría utilizar otras plataformas → *aplicar TDD no exige utilizar xUnit*
 - * Y se puede utilizar xUnit sin utilizar sólo para hacer pruebas → *utilizar xUnit no es aplicar TDD*
- JUnit
 - Conceptos básicos:
 - * *Test Fixtures* → Precondiciones que un test necesita
 - Tendremos un método *setUp* que inicialice estas precondiciones
 - En JUnit 4 → basta con anotarlo con Before

Introducción

Test Driven ...

Continuous Integration

Conclusiones

Referencias

Página www

Página de Abertura

◀◀ ▶▶

◀ ▶

Página 13 de 23

Regresar

Full Screen

Cerrar

Abandonar



- Tendremos un método *tearDown* que limpie lo que se ha creado en los tests
En JUnit 4 → basta con anotarlo con *After*
 - * *Test Suites* → Conjunto de tests que tienen el mismo *Test Fixture* y comparten una lógica común
 - Tendremos una clase que tendrá su *Test Fixture* y sus tests
En JUnit anterior a 4, esta clase heredaba de *TestCase*
 - * *Tests* → Los tests en sí, lanzan los métodos que se están comprobando, y se comprueban los resultados
 - Anotados con *@Test* en JUnit 4, normalmente empiezan por *test...*
 - Se comprueba con diferentes asertos (métodos estáticos de la clase *Assert* en JUnit 4)
`assertEquals(expected, actual), assertTrue(condition), (expected=EjemploException.class).`
- Ejemplo: 01_Acumulador
 - Ejercicio: 04_Calculadora

[Introducción](#)

[Test Driven ...](#)

[Continuous Integration](#)

[Conclusiones](#)

[Referencias](#)

[Página www](#)

[Página de Abertura](#)

◀▶

◀▶

Página 14 de 23

[Regresar](#)

[Full Screen](#)

[Cerrar](#)

[Abandonar](#)

2.3. Práctica

- Practicando un poco más: 05_BinarySearch
 - Recordemos: luz roja, luz verde, refactorizar
 - No tenemos por qué hacer la mejor solución a la primera :-)
- Y otro poco más:
 - Utilizando librerías externas: 06_LibreriaExterna
 - Tenemos una librería eterna llamada *LibreriaPuertoSerie*:
 - * Queremos testear la funcionalidad de la clase *UsandoLibreriaExterna*
 - * Esta clase depende de *LibreriaPuertoSerie*
 - * ¿Cómo testeamos esta funcionalidad?
 - * Solución en 07_LibreriaExternaSolucion
- Problemas normales a la hora de testear:
 - Código *viejo*:
 - * Al testear código que no está desarrollado con tests en mente, es común encontrar código *muy acoplado*, que es difícil de testear
 - * Casi sin darnos cuenta, vemos enseguida resultados al comparar cómo nos queda ahora el código y cómo nos quedaba antes
 - Capas que puede costar testear:
 - * GUIs...
 - * Objetos remotos...
 - * Bases de datos...
 - * Threading...



Introducción

Test Driven ...

Continuous Integration

Conclusiones

Referencias

Página www

Página de Abertura



Página 15 de 23

Regresar

Full Screen

Cerrar

Abandonar

- Ejemplo: aplicación que tiene un interfaz de usuario y finalmente utiliza bases de datos y hardware
 - * Podríamos por ejemplo hacer tests unitarios de cada parte del proyecto
 - Wrapeando la parte de más bajo nivel (que tenga relación directa con hardware)
 - Diseñando todo el interfaz de usuario encima de un Service Layer, y testeando todos los módulos desde Service Layer abajo
 - * Luego, hacemos tests de integración
 - No comprueban todo, sino la comunicación entre los interfaces de los diferentes objetos
 - Pueden hablar desde arriba del todo (ServiceLayer) hasta abajo del todo (wrapeando una vez más el hardware)
 - * Si el GUI se apoya sobre un Service Layer testado, no debería costar mucho probar “manualmente” el GUI, que es lo único que no se ha comprobado



[Introducción](#)

[Test Driven . . .](#)

[Continuous Integration](#)

[Conclusiones](#)

[Referencias](#)

[Página www](#)

[Página de Abertura](#)



[Página 16 de 23](#)

[Regresar](#)

[Full Screen](#)

[Cerrar](#)

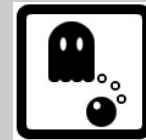
[Abandonar](#)

3. Continuous Integration

3.1. ¿Qué es?

- ¿Qué es Continuous Integration?
 - Otra práctica de Extreme Programming
 - Esencia:
 - * Trasladar el proceso de integración de software al día a día antes que relegarlo a una etapa final del proyecto
 - * Corregir problemas cuanto antes:
 - IDEs → Compilación continua → estamos cómodos
 - Sin embargo, con integración → tendemos a dejarla para el final del proyecto con todo el riesgo que ello conlleva.

Cuando los fallos que se intenten reparar van a ser difíciles de reparar con la seguridad de no estar rompiendo otras partes del proyecto.
 - * Objetivo → conseguir que todo el código (o casi todo) esté integrado en todo momento
 - ¿Cómo?
 - * Contar con un sistema de control de versiones que sea utilizado a diario por todos los desarrolladores
 - * Este sistema de control de versiones debería tener absolutamente todo lo necesario para construir el proyecto
 - Código
 - Scripts de compilación
 - Tests
 - Librerías externas
 - * Contar con scripts de compilación continuamente actualizados



[Introducción](#)

[Test Driven ...](#)

[Continuous Integration](#)

[Conclusiones](#)

[Referencias](#)

[Página www](#)

[Página de Abertura](#)



[Página 17 de 23](#)

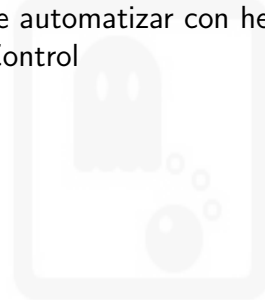
[Regresar](#)

[Full Screen](#)

[Cerrar](#)

[Abandonar](#)

- Compile todo el proyecto automáticamente
 - Despliegue todo el proyecto automáticamente
 - Cualquiera podrá bajarse el proyecto del repositorio en un momento dado y compilar y probar todo el software
- * Para que sea más eficaz → tests
- Automatizados
 - Lanzados por los scripts de compilación
- * Llevar a cabo el proceso de descarga, compilación, despliegue y pruebas a menudo
- Idealmente → por cada versión nueva, llevar a cabo el proceso
 - Este proceso se puede automatizar con herramientas de Integración Continua
Ejemplo: CruiseControl



[Introducción](#)

[Test Driven ...](#)

[Continuous Integration](#)

[Conclusiones](#)

[Referencias](#)

[Página www](#)

[Página de Abertura](#)



[Página 18 de 23](#)

[Regresar](#)

[Full Screen](#)

[Cerrar](#)

[Abandonar](#)



3.2. CruiseControl

- ¿Qué es CruiseControl?
 - Herramienta de Integración Continua :-)
 - Idea → lanza todo el proyecto cada cierto tiempo
 - Muy configurable:
 - * Los tests unitarios se suelen ejecutar en poco tiempo, mientras que los tests de integración (que utilizan BD reales, redes reales...), pueden tardar incluso un par de horas.
 - * CruiseControl permite automatizar que se lancen los tests unitarios cada pocos minutos, mientras que los tests de integración los podría lanzar una vez al día
 - * Informa a determinados usuarios de los resultados
 - Eliges a qué usuarios informar de qué resultados
 - Eliges cómo notificarlo (e-mail, chat a través de jabber, X10 para luces...)
- Ejemplo práctico

[Introducción](#)

[Test Driven ...](#)

[Continuous Integration](#)

[Conclusiones](#)

[Referencias](#)

[Página www](#)

[Página de Abertura](#)

◀▶

◀▶

Página 19 de 23

[Regresar](#)

[Full Screen](#)

[Cerrar](#)

[Abandonar](#)

4. Conclusiones

Vosotros diréis :-D



- Introducción
- Test Driven ...
- Continuous Integration
- Conclusiones**
- Referencias

Página [www](#)

Página de Abertura



Página 20 de 23

Regresar

Full Screen

Cerrar

Abandonar



Introducción

Test Driven ...

Continuous Integration

Conclusiones

Referencias

Página www

Página de Abertura



Página 21 de 23

Regresar

Full Screen

Cerrar

Abandonar

5. Referencias

5.1. Agile Software Development

- Agile Software Development Ecosystems. Jim Highsmith. Addison Wesley.
- Extreme Programming Explained: Embrace Change, 2nd Edition. Kent Beck. Addison Wesley.
- <http://www.extremeprogramming.org/>

5.2. Test Driven Development

- Test Driven Development by Example. Kent Beck. Addison Wesley.
- Refactoring: Improving the Design of Existing Code. Martin Fowler. Addison Wesley.
- “Mocks aren’t stubs”. Martin Fowler. <http://www.martinfowler.com/articles/mocksArentStubs.html> 2007.
- “Mock Roles, Not Objects”. Steve Freeman et al. <http://www.mockobjects.com/files/mockrolesnotobjects.pdf> 2004.
- <http://www.junit.org/>

5.3. Continuous Integration

- “Continuous Integration”. Martin Fowler. <http://www.martinfowler.com/articles/continuousIntegration.html> 2006.
- “Pragmatic Project Automation: How to Build, Deploy, and Monitor Java Applications”. Mike Clark. The Pragmatic Programmers. 2004.

- Continuous Integration: Improving Software Quality and Reducing Risk. Paul M. Duvall. Addison Wesley.
- <http://cruisecontrol.sourceforge.net/>



Introducción

Test Driven ...

Continuous Integration

Conclusiones

Referencias

Página www

Página de Abertura



Página 22 de 23

Regresar

Full Screen

Cerrar

Abandonar

Este documento está escrito por [Pablo Orduña](#) con la colaboración de [Iker Larizgoitia](#) para el “Taller de Test Driven Development” del grupo de software libre de la Universidad de Deusto, el [e-ghost](#). Puede encontrarse junto con los ejemplos y las fuentes \LaTeX en la misma web.

Para cualquier corrección, sugerencia, o comentario en general, no dudes en ponerte en contacto con nosotros en:

Pablo → [pablo @ ordunya . com](mailto:pablo@ordunya.com)

Iker → [ilarizgo @ tecnologico . deusto . es](mailto:ilarizgo@tecnologico.deusto.es)



[Introducción](#)

[Test Driven ...](#)

[Continuous Integration](#)

[Conclusiones](#)

[Referencias](#)

[Página www](#)

[Página de Abertura](#)



[Página 23 de 23](#)

[Regresar](#)

[Full Screen](#)

[Cerrar](#)

[Abandonar](#)