



TDD Avanzado

Hernán Wilkinson

Twitter: @HernanWilkinson

Blog: objectmodels.blogspot.com

www.10pines.com



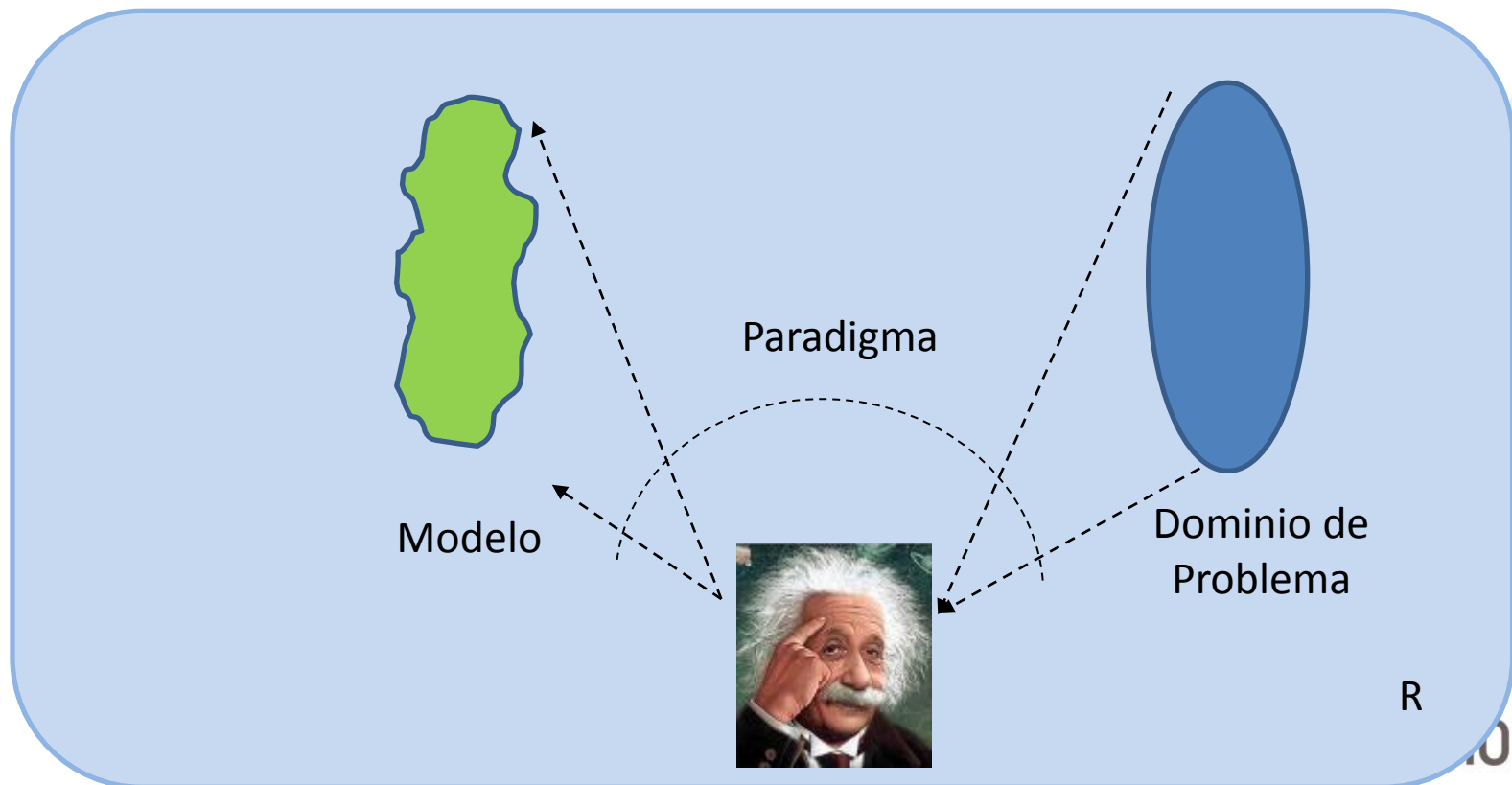
10 Pines
www.10pines.com

agile software development & services

Software

Modelo Computable de un Dominio de Problema de la Realidad

Modelo Computable de un Dominio de Problema de la Realidad



Desarrollo de Software

► Proceso de aprendizaje:

- Es iterativo
- Es incremental
- El conocimiento se genera a partir de hechos concretos
- El conocimiento generado debe ser organizado
- Feedback inmediato favorece el aprendizaje

Características

- El **Cambio** es una característica esencial del software
- El software cambia porque:
 - Cambia el dominio de problema
 - Cambia el soporte de ejecución
 - Cambia nuestra manera de entender el dominio de problema

Modelo

➤ **Buen Modelo:**

- Eje Funcional: Qué tan buena es la representación del dominio
- Eje Descriptivo: Qué tan bien está descripto el modelo, qué tan “entendible es”
- Eje Implementativo: Cómo “ejecuta” en el ambiente técnico

Buen Modelo – Eje Funcional

- Un modelo es bueno cuando puede representar correctamente toda observación de aquello que modela
 - Si aparece algo nuevo en el dominio, debe aparecer algo nuevo en el modelo (no modificarlo)
 - Si se modifica algo del dominio, solo se debe modificar su representación en el modelo
 - Relación 1:1 dominio-modelo (isomorfismo)
 - Es la parte “**observacional**” del desarrollo

Buen Modelo – Eje Descriptivo

- Un modelo es bueno cuando se lo puede “entender” y por lo tanto “cambiar”
 - Importantísimo usar buenos nombres
 - Importantísimo usar mismo lenguaje que el del dominio de problema
 - El código debe ser “lindo”
 - Es la parte “**artística**” del desarrollo

<http://www.10pines.com/content/art-naming>

<http://www.10pines.com/content/about-names-when-designing-objects>

Buen Modelo – Eje Implementativo

- Un modelo es bueno cuando ejecuta en el tiempo esperado usando los recursos definidos como necesarios
 - Performance
 - Espacio
 - Escalabilidad
 - Todo lo relacionado con Requerimientos No Funcionales
 - Es la parte “**detallista**” del desarrollo

Conclusiones

- Debemos hacer buenos modelos
- Debemos usar herramientas que favorezcan el aprendizaje
- Debemos favorecer técnicas que favorezcan el aprendizaje



¿Qué es TDD?

¿Qué es TDD?

- Técnica de Aprendizaje
 - Iterativa e Incremental
 - Basada en Feedback Inmediato
- Como side-effect:
 - Recuerda todo lo aprendido
 - Y permite asegurarnos de no haber “desaprendido”
- Incluye análisis, diseño, programación y testing

¿Cómo se hace TDD?

1) Escribir un test

- Debe ser el más sencillo que se nos ocurra
- Debe fallar al correrlo

¿Cómo se hace TDD?

1) Escribir un test

- Debe ser el más sencillo que se nos ocurra
- Debe fallar al correrlo

2) Correr todos los tests

- Implementar la solución más simple que haga pasar el/los test/s
- GOTO 2 hasta que “todos los tests” pasen

¿Cómo se hace TDD?

1) Escribir un test

- Debe ser el más sencillo que se nos ocurra
- Debe fallar al correrlo

2) Correr todos los tests

- Implementar la solución más simple que haga pasar el/los test/s
- GOTO 2 hasta que “todos los tests” pasen

3) Reflexiono - ¿Se puede mejorar el código?

- Sí -> Refactorizar. GOTO 2
- No -> GOTO 1

¿Cómo se hace TDD?

¿POR QUÉ?

1) Escribir un test

- Debe ser **el más sencillo** que se nos ocurra
- Debe fallar al correrlo

2) Correr todos los tests

- Implementar la solución más simple que haga pasar el/los test/s
- GOTO 2 hasta que “todos los tests” pasen

3) Reflexiono - ¿Se puede mejorar el código?

- Sí -> Refactorizar. GOTO 2
- No -> GOTO 1

¿Cómo se hace TDD?

1) Escribir un test

- Debe ser el **caso de prueba** más sencillo que se nos ocurra
- **Debe fallar** al correrlo

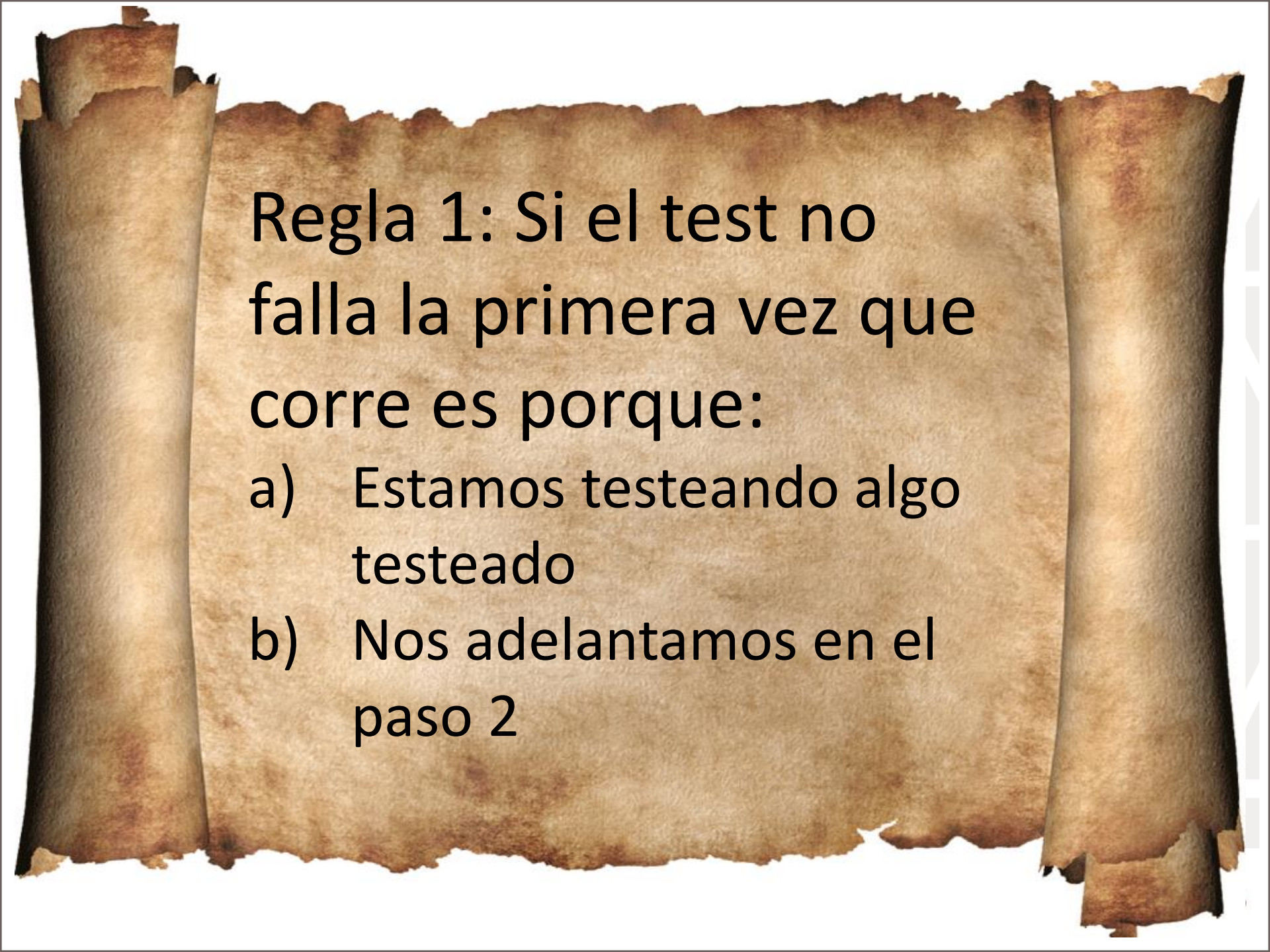
¿POR QUÉ?

2) Correr todos los tests

- Implementar la solución más simple que haga pasar el/los test/s
- GOTO 2 hasta que “todos los tests” pasen

3) Reflexiono - ¿Se puede mejorar el código?

- Sí -> Refactorizar. GOTO 2
- No -> GOTO 1

A scroll of aged parchment with a rough, torn edge. The parchment is a light tan color with some darker staining and texture. The text is written in a simple, black, sans-serif font. The scroll is unrolled, showing a central rectangular area of text.

Regla 1: Si el test no
falla la primera vez que
corre es porque:

- a) Estamos testeando algo
testeado
- b) Nos adelantamos en el
paso 2

¿Cómo se hace TDD?

1) Escribir un test

- Debe ser el más sencillo que se nos ocurra
- Debe fallar al correrlo

2) Correr todos los tests

¿POR QUÉ?

- Implementar **la solución más simple** que haga pasar el/los test/s

- GOTO 2 hasta que “todos los tests” pasen

3) Reflexiono - ¿Se puede mejorar el código?


- Sí -> Refactorizar. GOTO 2
- No -> GOTO 1

¿Cómo se hace TDD?

1) Escribir un test

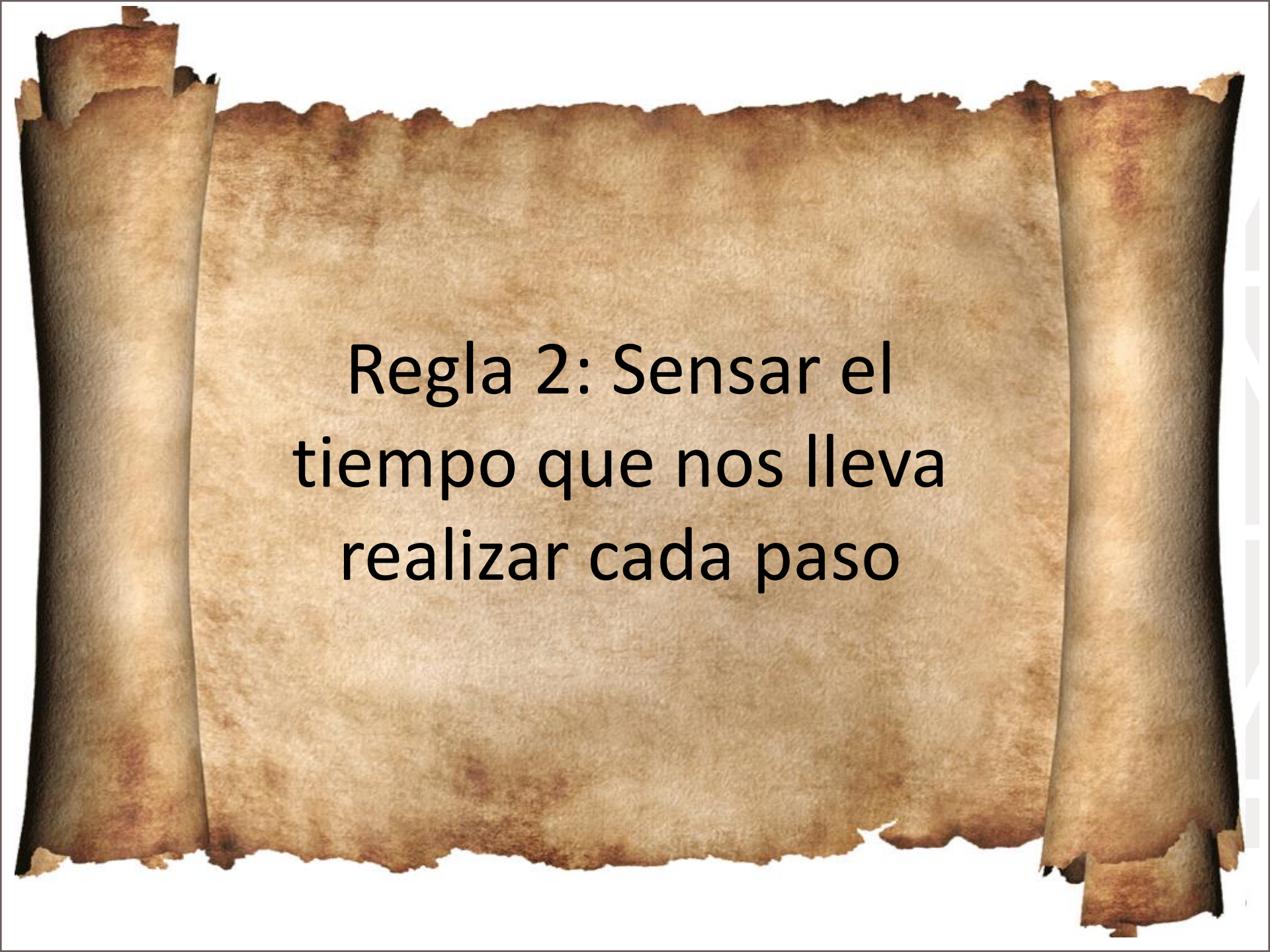
- Debe ser el más sencillo que se nos ocurra
- Debe fallar al correrlo

2) Correr todos los tests

- Implementar la solución más simple que haga pasar el/los test/s
- GOTO 2  que "todos los tests" pasen

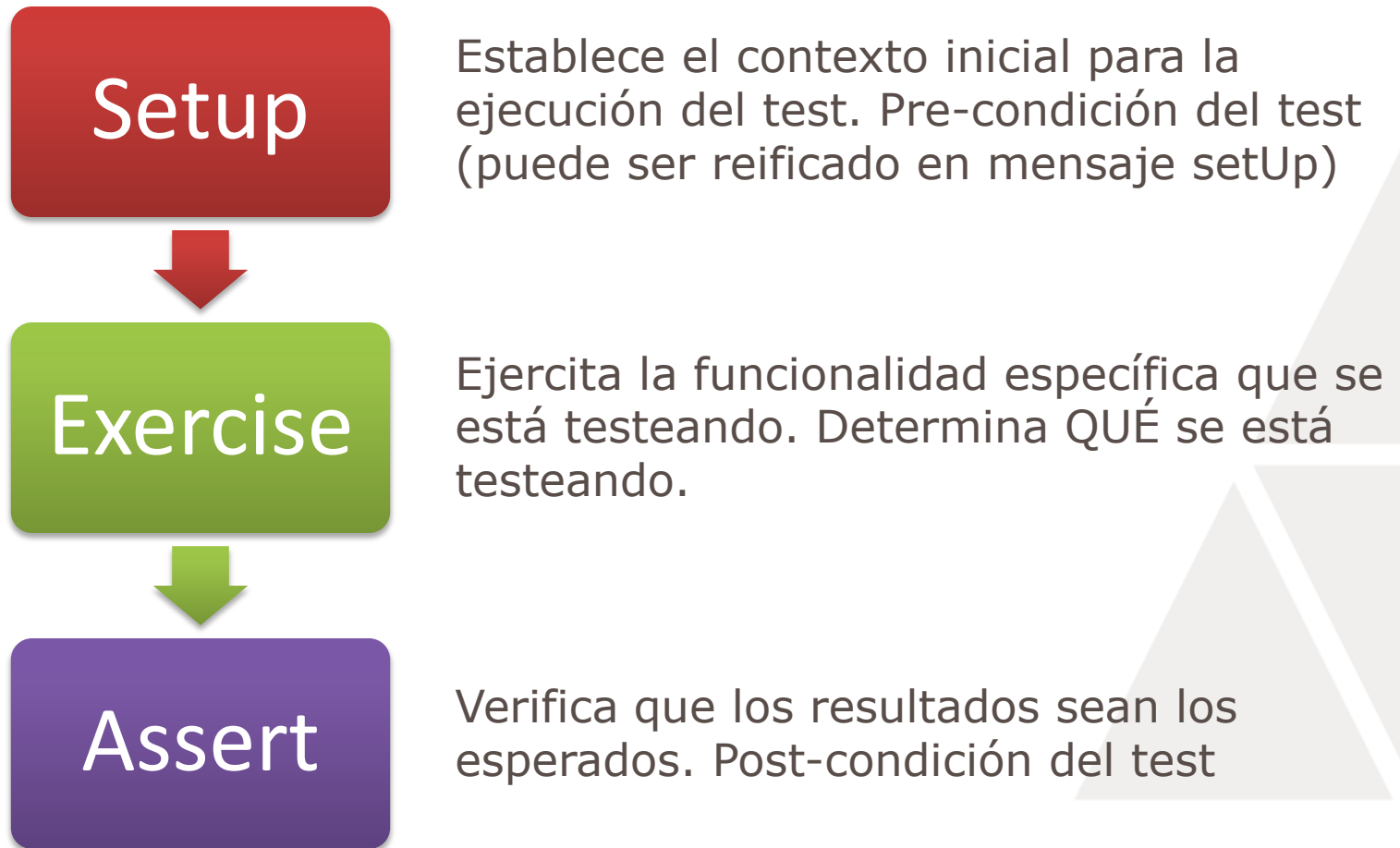
3) **Reflexiono** - ¿Se puede mejorar el código?

- Sí -> Refactorizar. GOTO 2
- No -> GOTO 1

A scroll of aged parchment with a rough, torn edge. The parchment is a light tan color with some darker staining. The text is written in a simple, black, sans-serif font. The scroll is unrolled, showing a wide central area with the text and two narrow vertical strips on the left and right sides.

Regla 2: Sensar el
tiempo que nos lleva
realizar cada paso

Estructura de los tests





Desarrollo Iterativo/Incremental

Las reglas de Oro

- Escribir el test primero
- El test debe ser el más sencillo posible y debe fallar
- Hacer la mínima implementación necesaria para que el test pase
- Refactorizar!

Ejemplo

- ▶ Algoritmo para calcular los factores primos de un número entero. Debe devolver una lista con ellos
 - 1 -> {} (no tiene factores primos)
 - 2 -> { 2 }
 - 3 -> { 3 }
 - 4 -> { 2,2 }
 - 5 -> { 5 }
 - 6 -> { 2,3 }
 - 7 -> { 7 }
 - 8 -> { 2,2,2 }
 - 9 -> { 3,3 }
 - 10 -> { 2,5 }
 - Etc

Conclusiones

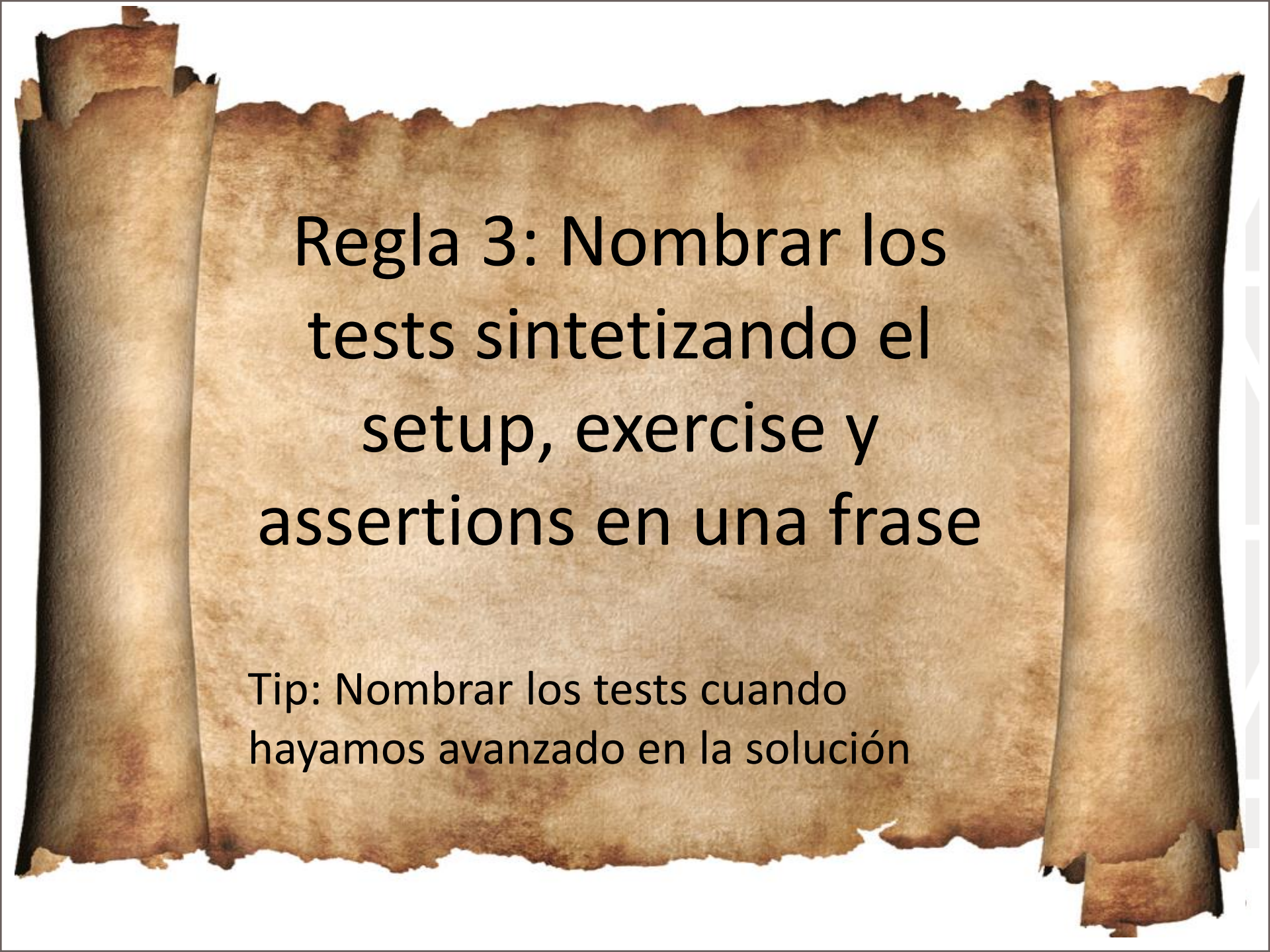
- Cuanto más específicos se hacen los test, más genérico es el código
- Debo aprender a encontrar los patrones en el código
 - El código nos habla
 - Muchos if seguidos → implican while
 - No importa como encontremos el patrón, si el patrón es el mismo la solución va a ser la misma

Conclusiones

- Un test por caso
 - Organizar bien los test
- TDD no implica buen diseño!
- Los buenos diseños los hacen los buenos diseñadores
- Pensar en objetos para tener buenos diseños

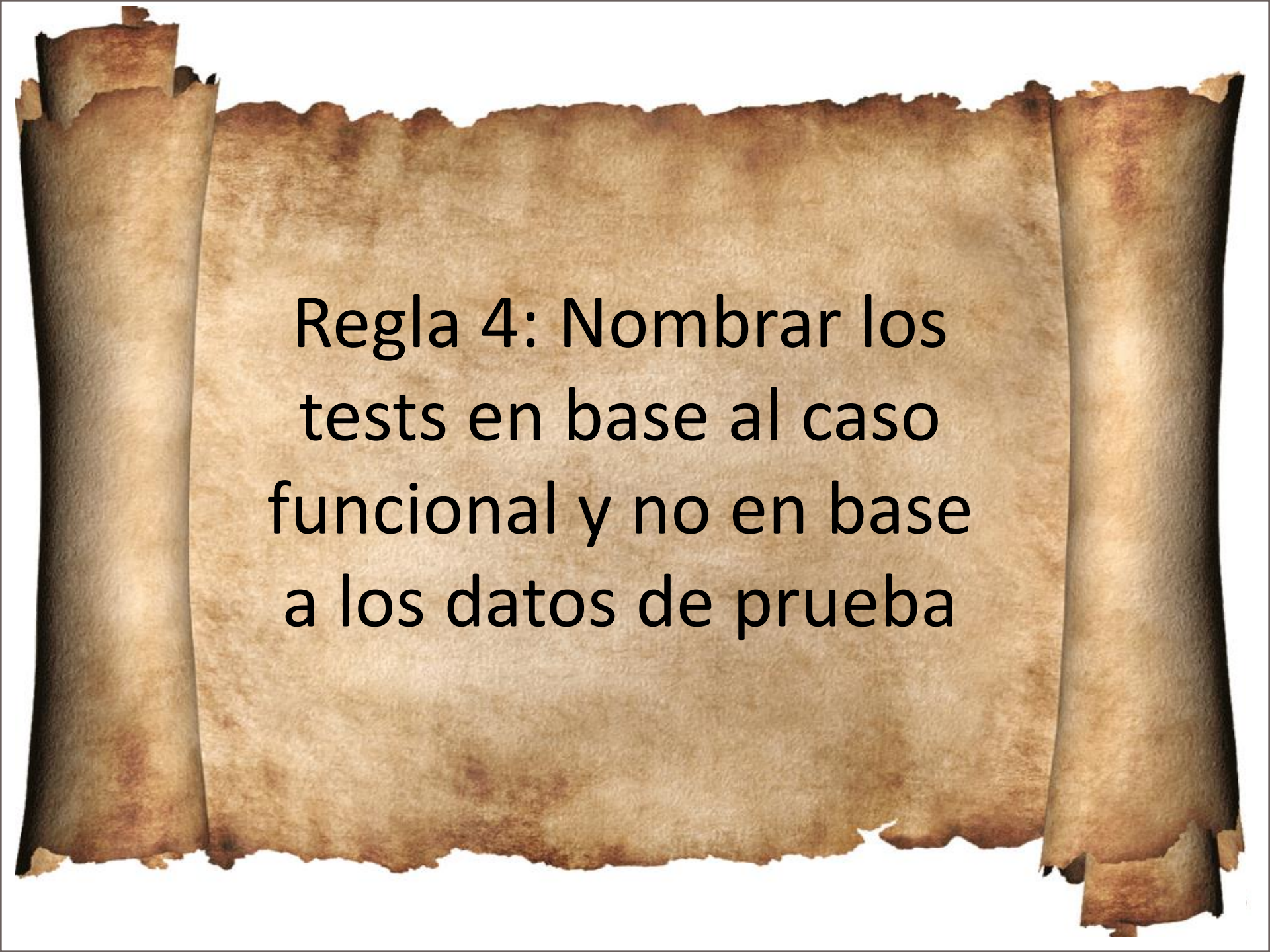
Cómo sacar Código Repetido

- Copiar el código repetido a “un lugar”
- Parametrizar lo que cambia
 - Si lo que cambia es código usar closure
- Ponerle Nombre!



Regla 3: Nombrar los tests sintetizando el setup, exercise y assertions en una frase

Tip: Nombrar los tests cuando hayamos avanzado en la solución

A scroll of aged parchment with a rough, torn edge. The parchment is a light tan color with some darker staining and texture. The text is written in a simple, black, sans-serif font. The scroll is unrolled, showing a central rectangular area of text.

Regla 4: Nombrar los
tests en base al caso
funcional y no en base
a los datos de prueba



Tip: Dato de Prueba \neq Caso de Prueba

- **Dato de Prueba:** Ejemplos concretos que “definen” un caso de prueba
- **Caso de Prueba:** Generalización que incluye los datos de prueba

Ejercicio

- Hacer de manera iterativa-incremental la conversión de números enteros a números romanos
 - $1 \rightarrow I$
 - $5 \rightarrow V, 10 \rightarrow X$
 - $50 \rightarrow L, 100 \rightarrow C$
 - $500 \rightarrow D, 1000 \rightarrow M$

¿Qué no es TDD?

- No es Testing (únicamente)
- No es Unit Testing
- No es “no reflexionar” – “no pensar”
- No es “no refactorizar”

¿Cuándo no hago TDD?

- Cuando no tengo feedback inmediato
 - Escribiendo/Modificando código antes de escribir un test
 - Escribiendo muchos tests antes de escribir el código

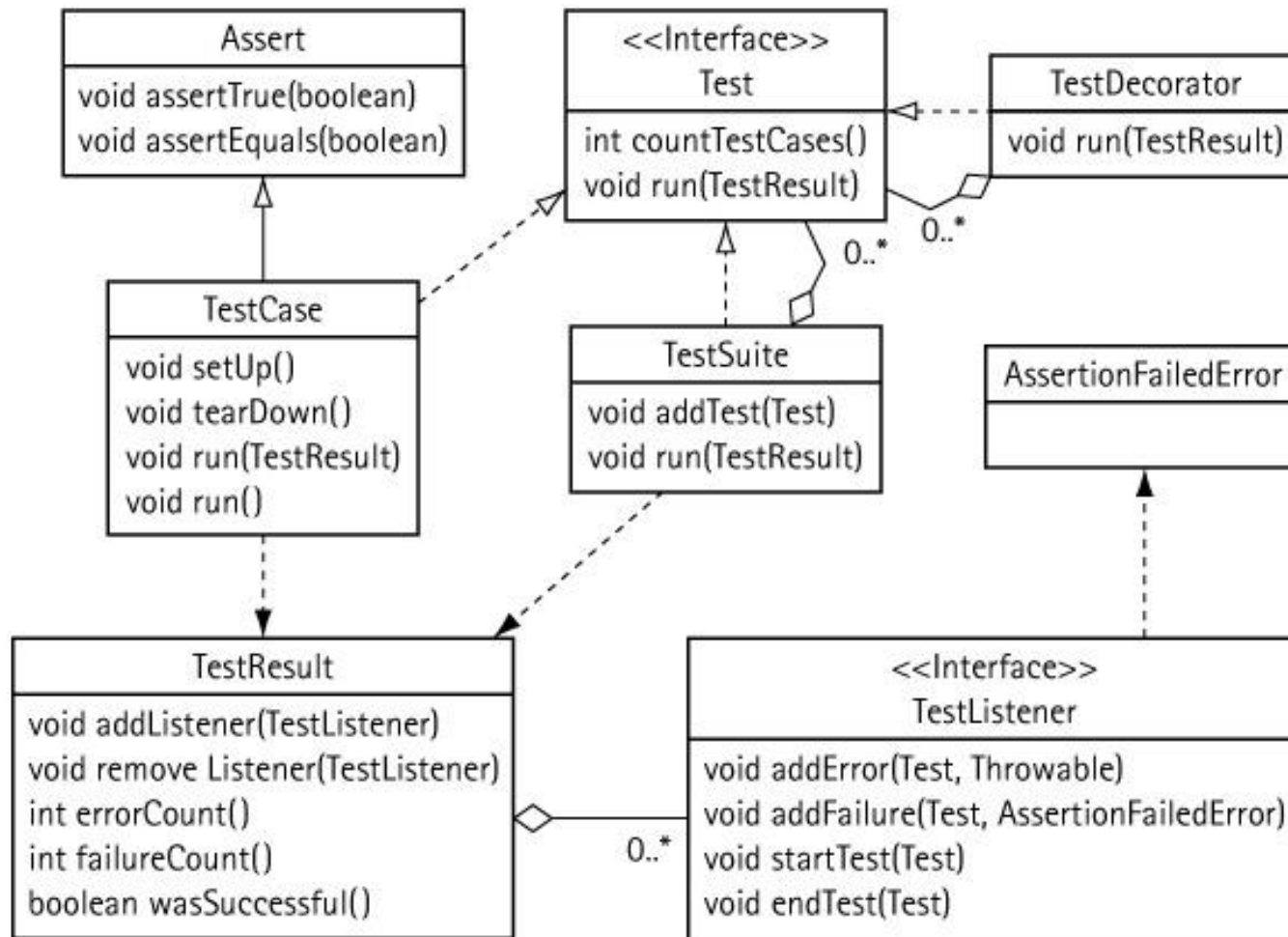
¿Cuándo no hago TDD?

- Cuando no desarrollo de manera iterativa-incremental
 - Escribiendo la “solución completa” de entrada
 - Haciendo up-front design
 - Escribiendo los tests luego de tener el código – Haciendo Testing



Modelo de xUnit

Diseño



Consecuencias

- Una instancia de la clase de test por test
 - Isolation entre tests de la misma clase
- Las instancias quedan vivas mientras quede vivo el TestResult
- Si las instancias referencias muchos objetos/recursos puede ser un problema

Anatomía de los Tests

```
public class MoneyTest {
```

```
    @Test
```

```
    public void testAddingMoneyShouldAddItsAmounts() {
```

```
        //set up
```

```
        Money pesos12 = new Money(12, "ARS");
```

```
        Money pesos14 = new Money(14, "ARS");
```

```
        Money expected = new Money(26, "ARS");
```

} SetUp / Arrange

```
        //exercise
```

```
        Money result = pesos12.add(pesos14);
```

} Excercise / Act

```
        //verify
```

```
        assertEquals(expected, result);
```

} Verify / Assert

```
    }
```

```
}
```

SetUp

- ¿Qué hacer cuando hay código repetido de set up en los distintos tests?
 - Usar el setup del framework (setUp, @Before, TestInitialize, etc)
 - Recordar que hay una instancia por test! → Creación de Objetos innecesarios
 - Solo poner en setUp lo que es común a todos los tests
 - Usar distintos métodos de setUp si hay conjuntos distintos de objetos comunes entre tests

TearDown

- Generalmente no se usa debido al GarbageCollector
- Usar sólo para relesear recursos tomados durante el setUp
 - Pero recordar que no se deberían usar recursos externos en los tests!
 - Conclusión: Usar TearDown es un smell de mal test

BeforeClass / AfterClass

- Ejecuta antes de todo un suite y después de todo un suite
- Tienen la misma consecuencia que setUp/tearDown pero para toda una Suite

Fixtures

- Conjunto de objetos de prueba
- Tipos de Fixtures:
 - Transient: Se crean en cada test. No hay relación entre distintos tests
 - Persistent Fresh: Los objetos creados quedan persistidos entre los tests, pero se crea uno nuevo por test
 - Persistent Shared: Idem Persistent Fresh pero los objetos se comparten entre tests

Fixtures

- Favorecer Fixtures Transient
 - Evitan acoplamiento e impacto entre tests
 - Permiten paralelizar ejecución de tests
 - No implican orden de ejecución de tests
- Evitar Fixtures Persistent y más aún los shared
 - Ejecución más lenta de tests
 - Los shared no permiten paralelizar tests e imponen orden de ejecución
 - Los shared son motivo de tests erráticos

Características deseables de los tests

- Deben correr rápido
- Deben ser "chicos"
- Deben ser "entendibles"
- Debe haber un test por caso
- Deben estar en control de todo
- Misma cantidad de líneas de código que las del sistema
- Los nombres deben ser declarativos y resumir el given/when/then

Técnicas de Testing

- Testear un solo “caso” por test
 - Esto no implica tener un solo assert
- Empezar siempre por el test más sencillo
- Empezar por la asserción primero, ayuda a entender qué se quiere hacer

Técnicas de Testing

- Siempre debe haber algún assert en el test (o fail, etc), de lo contrario no es un test
- Recordar de testear casos “negativos” no solo “positivos”
- Testear el “paso del tiempo”
- Recordar que el test debe estar en “control de todo”

Técnicas de Testing

- Testeo de colecciones:
 - Assertar sobre el size
 - Assertar que estén únicamente los objetos que deben estar

Técnicas de Testing

➤ Testeo de excepciones:

```
try {
```

...

Un solo envío de mensaje

fail();

Siempre! Para evitar falsos positivos

```
}
```

```
catch (ExpectedException e) {
```

assercciones sobre e

assercciones sobre invariantes

```
}
```

Para asegurarnos que realmente no se hizo nada

Tipos de Tests

➤ Tests insoportables

- Tardan mucho!
- Seguramente usan algún recurso lento (base de datos, conexión con otro sistema, etc.)

➤ Tests frágiles

- Tests que se rompen cuando se modifica la implementación interna de un objeto
- Son tests de caja blanca

➤ Tests Erráticos

- A veces andan, a veces no...
- Hay dependencia de fixtures entre tests o usan recursos externos

Bad Smells

- Un nuevo test funciona de entrada la primera vez que lo corro

- Significa:
 1. El caso ya está testeado → Borrar el test
 2. No estoy haciendo un desarrollo interactivo incremental → Controlar la ansiedad, hacer “baby steps”

Bad Smells

- Tardo mucho para hacer funcionar un test
- Significa:
 1. El test era muy complejo, no estoy haciendo desarrollo incremental → Borrar el test y hacerlo más simple
 2. El modelo es muy complejo → Debo refactorizar

Bad Smells

- Tardo mucho en escribir un test
 - Por que el set up es muy largo
- Significa:
 1. No estoy “reflexionando”→ Debo pensar como simplificar la creación de objetos de prueba

Bad Smells

- Tardo mucho en escribir un test
 - Por que las “acción” es muy larga

- Significa:
 1. El test es muy complejo, no estoy haciendo el test más sencillo → Borrar el test y hacerlo más simple. Particionar el test en varios casos
 2. El modelo es muy complejo → Debo refactorizar

Bad Smells

- Tardo mucho en escribir un test
 - Por que son muchas aserciones

- Significa:
 1. El test es muy complejo, estoy testeando varios casos juntos → Particionar el test en varios casos
 2. No estoy “reflexionando” sobre qué significa cada aserción → Debo refactorizar

Bad Smells

- Me lleva mucho tiempo decidir que test escribir

- Significa:
 1. Les estoy dando mucha importancia pensando cual es el más simple → Actuar!! Hay que tener feedback para aprender y mejorar!!
 2. El dominio de problema es muy complejo → Aceptarlo
 3. Terminé → Ir a tomar una cerveza!



Bad Smells

- Los tests tardan mucho en correr
- Significa:
 1. Les estoy dando mucha importancia pensando cual es el más simple → Actuar!! Hay que tener feedback para aprender y mejorar!!
 2. El dominio de problema es muy complejo → Aceptarlo
 3. Terminé → Ir a tomar una cerveza!

Bad Smells

- Tardo mucho en hacer refactorings
- Significa:
 - ¡Aprender refactorings automatizados!
 - Si están programando con editores de texto ¡pasar a un IDE!
 - Aprender como hacer cambios chicos por medio de refactorings encadenados
 - El diseño del sistema es muy malo, esta muy acoplado y no queda otra que tomarse tiempo para arreglarlo ☹



Mantenimiento

Hacer TDD en código existente

➤ Problemas

- Tengo que arreglar un error ...
- ... pero no puedo modificar código sin escribir un test primero...
- ... pero no puedo escribir el test porque hay muchas dependencias que me lo impiden...
- ... y debo romper esas dependencias para poder tester ...
- ... pero no puedo modificar código sin escribir un test primero...
- ...

Hacer TDD en código existente

➤ Problemas

- Tengo que agregar funcionalidad ...
- ... pero no puedo modificar código sin escribir un test primero...
- ... pero no puedo escribir el test porque el código es muy complicado, no lo entiendo...
- ... y debo modificarlo para entenderlo ...
- ... pero no puedo modificar código sin escribir un test primero...
- ...

Hacer TDD en código existente

➤ Problema: Mal diseño

- Código no entendible (malos nombres, mala distribución de responsabilidades, poca cohesión, etc).
- Mucho acoplamiento (referencia a recursos externos, mala descomposición funcional, etc)

➤ Solución

- Refactoring, objetos simuladores, romper dependencias

Ejemplo

➤ Customer Import

- El sistema importa desde archivo clientes y sus direcciones.
- Formato de archivo:

Cientes

C,Pepe,Sanchez,D,22333444
A,San Martin,3322,Olivos,1636,BsAs
A,Maipu,888,Florida,1122,Buenos Aires
C,Juan,Perez,C,23-25666777-9
A,Alem,1122,CABA,1001,CABA

} Direcciónes del Cliente

Ejemplo

➤ Customer Import

- Hay un main que prueba que funciona!
- Pero... ¿Funciona bien?

Mantenimiento de código

- Reglas de mantenimiento
 - Identificar los puntos de cambio
 - Identificar los puntos de testeo
 - Romper dependencias ← REFACTORING! Y otras técnicas
 - Escribir tests
 - Hacer cambios por medio de TDD

Refactorings

➤ Rename

- Renombra el elemento seleccionado y todas las referencias

➤ Extract method

- Crea un nuevo método con el código seleccionado y lo reemplaza por un envío de mensaje a ese método
- Algunas herramientas reemplazan todas las repeticiones de la selección

Refactorings

➤ Move method

- Mueve el método seleccionado a una clase que debe estar visible desde el contexto en que se mueve. Modifica todas las referencias al método movido

➤ Convert local to field - Introduce Field

- Convierte una variable local en variable de instancia
- Puede mover la inicialización de la variable

Refactorings

- Extract to Local - Introduce Variable
 - Extrae el código seleccionado en una variable local inicializada con dicho código
- Introduce parameter
 - Reemplaza el código seleccionado con una referencia a un parámetro, modifica todos los senders agregando dicho parámetro que será el código reemplazado

Refactorings

- Change Method signature - Change Signature
 - Permite modificar los elementos que definen un método como tipo de retorno, parametros, etc.
 - Modifica todos los senders para quedar de acuerdo a la nueva definición
 - Cuando se agregan parámetros se puede indicar con qué código usado en los senders

Refactorings

- Generalize Declared Type - Use base type where possible
 - Permite reemplazar el tipo de una variable por algún supertipo

Ejercicio – 1ra Parte

➤ Customer Import

- Pero... ¿Funciona bien?
- Le han pedido que también se puede importar usando sockets

Refactorings

➤ Inline

- Copia el código representado por el método o variable a los lugares donde se referencia dicho método o variable

➤ Extract to Local + Extract Method + Inline

- Se los suele utilizar para extraer código que se encuentra separado

Ejercicio – 2da Parte

➤ Customer Import

- Hay veces que no se importa nada y no se sabe por que...
- Lo único que se sabe es que se genera excepciones como `NullPointerException` e `IndexOutOfBoundsException`
- Solucionarlo!

Refactorings

➤ Encapsulate Field

- Referencias a variables de instancia se reemplazan por getter y setters

➤ Encapsulate Field + Move

- Permite mover variables de instancia a otras clases

Refactorings

➤ Extract Interface

- Permite crear una interface a partir de parte del protocolo definido en una clase

➤ Extract Interface + Generalize Declared Type

- Sirve para crear jerarquías polimórficas

Ejercicio – 3ra Parte

➤ Customer Import

- Los tests no cumplen con la regla de que deben ser rápidos...
- Están acoplados con la base de datos!
- Hacer que los test corran más rápido desacoplándolos de la base...
- ... pero recordar que a veces hay que testear con la base... cuando se está en el ambiente de integración

Ejercicio – 4ta Parte

- Dejo de ser Customer Import...
- Ahora es ERPSystem!!
 - Nuestros queridos vendedores no se cansan de darnos trabajo! Vendieron el sistema como ERP por lo que hay que agregar proveedores! (Supplier)
 - Se debe poder importar Supplier pero no hay mucho tiempo porque el sistema ya está vendido...
 - ... nos prometieron tiempo para poder mejorarlo después

Supplier

@Entity

@Table(name = "SUPPLIERS")

public class Supplier {

 @Id

 @GeneratedValue

 private long id;

 @NotEmpty

 private String name;

 @Pattern(regexp="D|C")

 private String identificationType;

 @NotEmpty

 private String identificationNumber;

 @OneToMany(cascade = CascadeType.ALL)

 private Set<Address> addresses;

 @OneToMany(cascade = CascadeType.ALL)

 private Set<Customer> customers;

Ejemplo

➤ Supplier Import



Refactorings

- Extract Parameter Object - Extract Class from Parameters
 - Permite crear una abstracción nueva a partir de un conjunto de parámetros de un método
 - Reemplaza todos los senders por la instanciación de la nueva abstracción

Refactorings

- Extract Parameter Object + Move
 - Permite crear nuevas abstracciones a partir de código existente
- Extract Parameter Object + Generalize Declared Type
 - Permite usar la misma abstracción extrayéndola de métodos distintos

Refactorings

- Extract Superclass

- Idem Extract Interface pero con clase

- Extract Class

- Permite encapsular variables de instancia en una clase

Refactorings

➤ Pull Up

- Mueve variables de instancia y/o métodos a la superclase o declara el método como abstracto en la superclase

➤ Push Down

- Mueve un conjunto de variables de instancia y/o métodos a las subclases

Ejercicio – 5ta Parte

- Tenemos tiempo para sacar la “deuda técnica”
 - Nos dimos cuenta que Customer y Supplier tienen mucho en común por lo que creamos la abstracción Party
 - También vimos que identificationType e identificationNumber van juntos por todos lados... deberíamos crear la abstracción PartyIdentification



Mutation Testing

¡Siempre Recordar!



- Dijkstra: "Un test solo verifica que lo que se testea funciona o no"
- No es completo ni formal
- Para mejorar el testing:
 - Coverage
 - Mutation Testing

Frameworks

- ▶ Java: PIT Mutation Testing
 - <http://pitest.org/>
- ▶ C#: Ninja Turtles
 - <http://www.mutation-testing.net/>
- ▶ Ruby: Mutant
 - <https://github.com/mbj/mutant>
- ▶ Smalltalk: Mutalk
 - <https://code.google.com/p/mutalk/>



Conclusiones

Recordar

- Lo más importante: Feedback inmediato para poder reflexionar
 - No preocuparme por nombres de tests al principio
 - No preocuparme por organización de los tests al principio
 - Recordar que es iterativo incremental

Recordar

- Usar la computadora como soporte/medio de aprendizaje
 - No correr el sistema en nuestra cabeza!
- Hacerle caso a nuestro instinto
 - El código nos habla
 - No ser derrotista
- Es test no es el fin, es un medio
- Un programador no es buen programador si no es buen tester

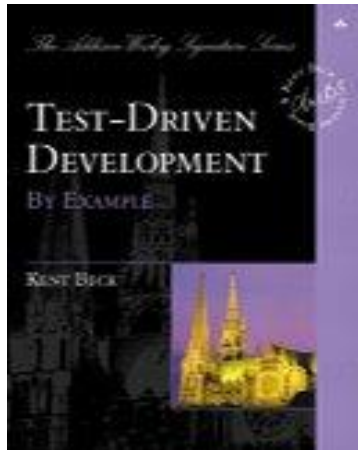
Recordar

- TDD ayuda a generar diseños menos acoplados
- TDD NO IMPLICA BUEN DISEÑO!!
- TDD nos da confianza para realizar cambios cuando el modelo así lo requiera
- Debido al punto anterior, el sistema no se convierte en Legacy



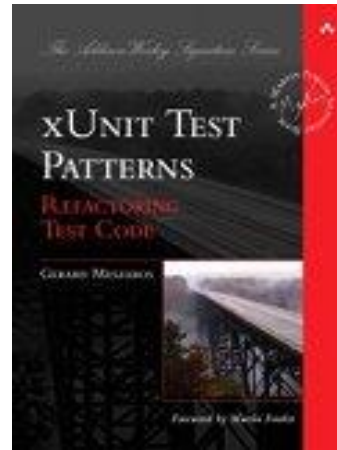
Bibliografía y referencias

Bibliografía recomendada



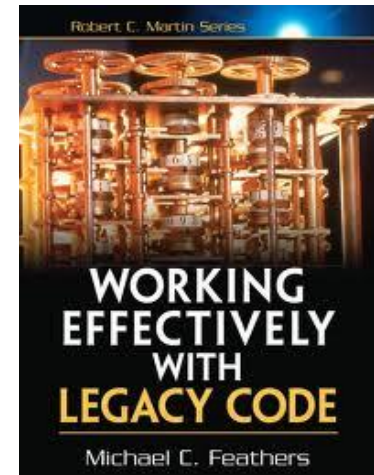
Test Driven
Development By
Example.

Kent Beck, 2002.



xUnit Test Patterns:
Refactoring Test Code

Gerard Meszaros, 2007.



Working Effectively with
Legacy Code

Michael Feathers

Bibliografía adicional

- ▶ S. Freeman, N. Pryce – *Growing Object Oriented Software with Tests*, 2009
- ▶ K.Beck, M.Fowler – *Planning Extreme Programming*. 2000
- ▶ L.Crispin, J.Gregory - *Agile Testing: A Practical Guide for Testers and Agile Teams*. 2009
- ▶ L.Koskela - *Test Driven: TDD and Acceptance TDD for Java Developers*. 2007
- ▶ T.Mackinnon, S.Freeman, P.Crai - *Endo-Testing: Unit Testing with Mock Objects*. 2000
<http://www.mockobjects.com/files/endotesting.pdf>
- ▶ M.Fowler - *Mocks Aren't Stubs*.
<http://martinfowler.com/articles/mocksArentStubs.html>