

A photograph of a steep, green mountain slope. A narrow, winding path or stream bed cuts through the grass and rocks. The sky is clear and blue.

Test Driven Development

¿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?

1) Escribir un test

¿POR QUÉ?

- 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 más simple 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

¿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 → **¿POR QUÉ?** que “todos los tests” pasen

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

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

¿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

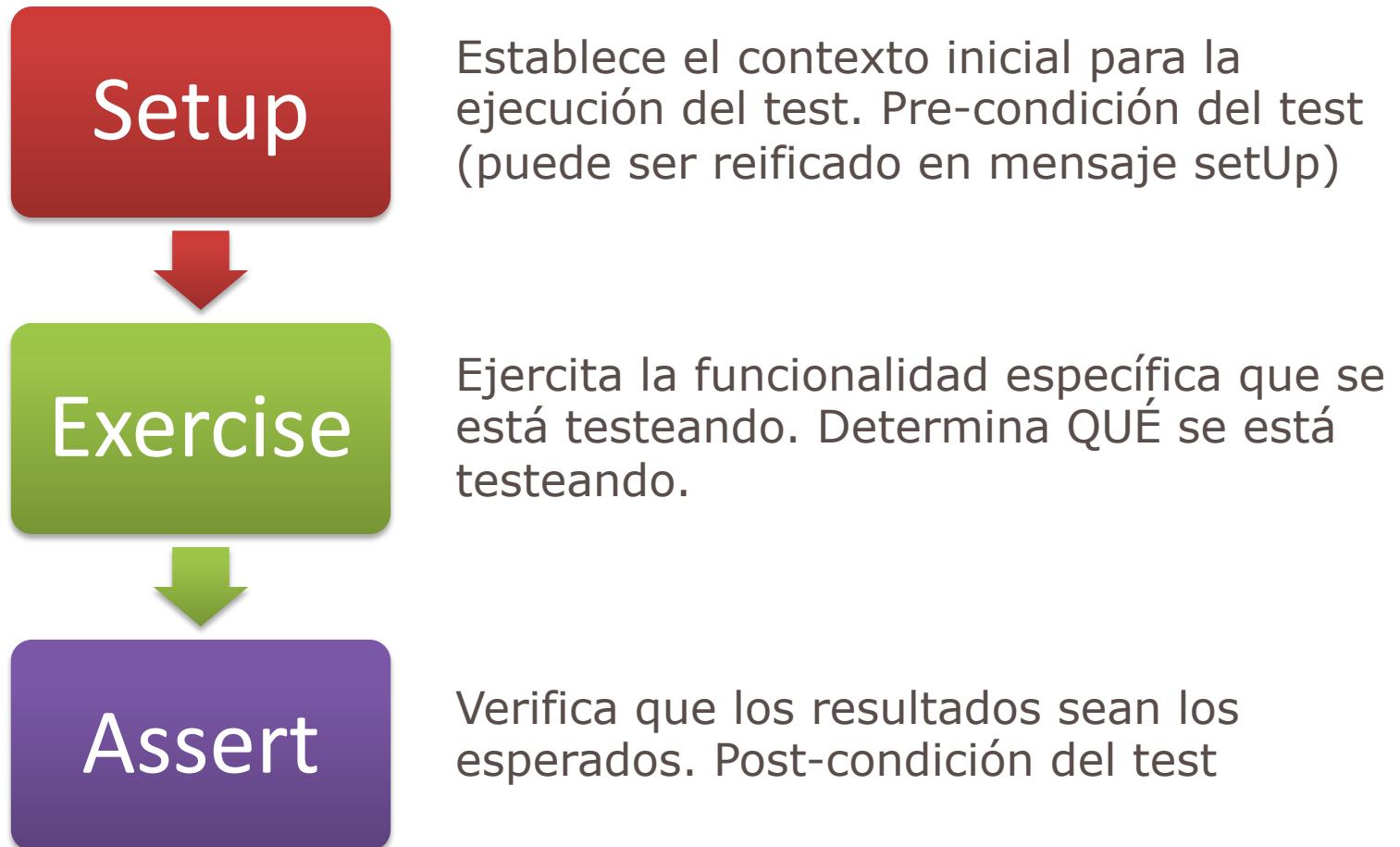
¿Cuándo no hago TDD?

- 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

Framework de Testing: xUnit

- Implementado por Kent Beck
- Framework de caja blanca pero simple
- Los casos de test se implementan en métodos, ej: testSimpleAdd
- Los casos de test se agrupan en clases (suites “naturales”), ej: MoneyTest
- El Test Runner usa reflexión para saber que tests correr
- Primera implementación: SUnit
- Nombre es “misleading”

Estructura de los tests



En Smalltalk: SUnit

```
TestCase subclass: #MoneyTest  
instanceVariableNames: ""  
classVariableNames: ""  
poolDictionaries: ""  
category: 'Money-Tests'
```

Las clases de test
subclasifican TestCase

Los test deben empezar con
#test

testSimpleAdd

```
| pesos12 pesos14 pesos26 sum |
```

"set up"

```
pesos12 := Money for: 12 currency: 'ARS'.  
pesos14 := Money for: 14 currency: 'ARS'.  
pesos26 := Money for: 26 currency: 'ARS'.
```

"exercise"

```
sum := pesos12 + pesos14.
```

```
self assert: pesos26 = sum.|
```

TestCase implementa los
mensajes para hacer
aserciones

Runner en el mismo ambiente de desarrollo

Ejemplo

- ▶ Modelar un Calendario de días feriados al que se le pueda preguntar si una fecha es feriado o no
- ▶ Se pueda indicar qué días son feriados de la siguiente manera:
 - Por medio de un día de la semana, ej. Domingo
 - Por medio de un día de un mes, ej. 25 de Diciembre
 - Por medio de un día particular, ej. 20/4/2012



Conclusiones - Diseño

- Es difícil poner nombres
 - Se debe a que no conocemos el dominio aún!
 - No perder mucho tiempo en eso al principio
 - Poner “Nombres sin Significado” en vez de “Malos Nombres”

Conclusiones - Diseño

➤ Recordar que es una aprendizaje continuo

- Con el tiempo tuvimos más conocimiento del dominio
- Pudimos elegir mejores nombres
- Refactorizamos y mejoramos el modelo (proceso incremental)

Conclusiones - Diseño

- Implementar la solución más sencilla, acorde al problema presentado
- No caer en “análisis parálisis”

Conclusiones - Técnica

- Escribir el assert primero
- Técnica de testing 1: Assertar por casos negativos, no solo positivos
- Escribir un test por caso!
(no implica un assert por test)

Conclusiones - Técnica

- Implementar la funcionalidad mínima para hacer pasar el test:
 - Nos hace pensar en todos los otros casos a tener en cuenta
 - Nos asegura que vamos a cubrir todos los casos y estamos haciendo un desarrollo incremental

Conclusiones - Impacto

- TDD nos permite tener un modelo inicial rápidamente
- ¡Está funcionando! → Fuerte efecto Psicológico

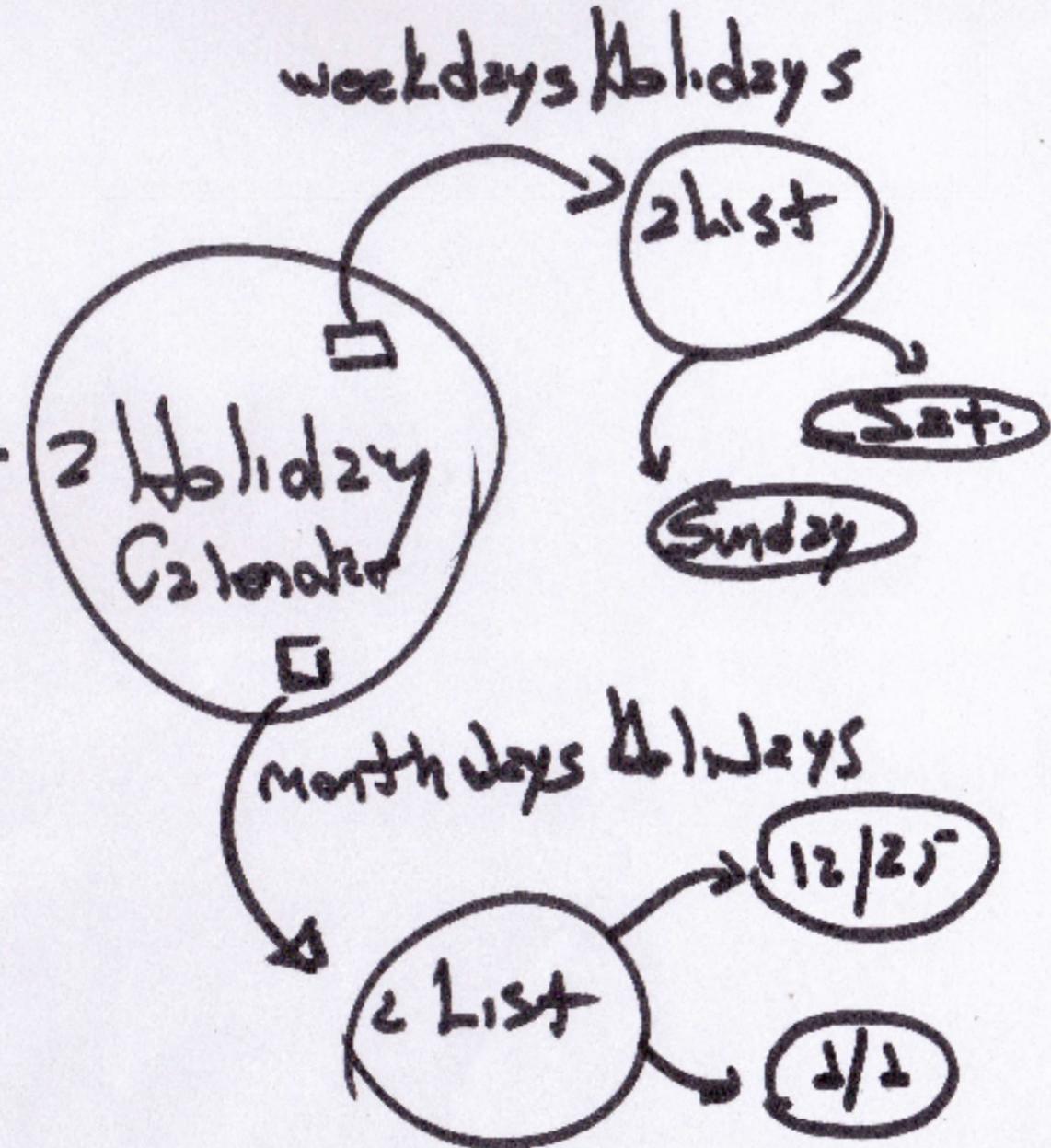
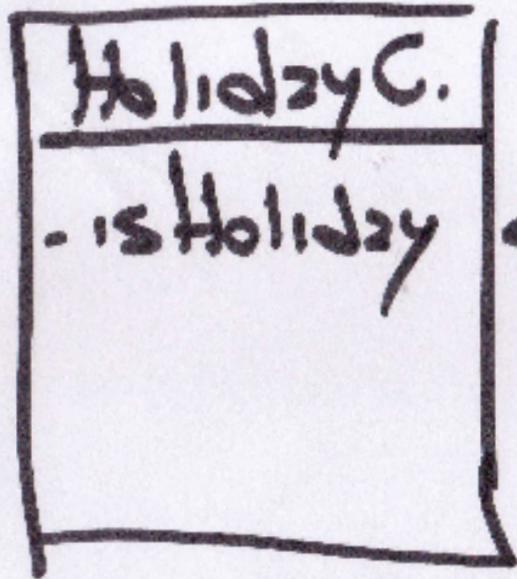
Conclusiones - Impacto

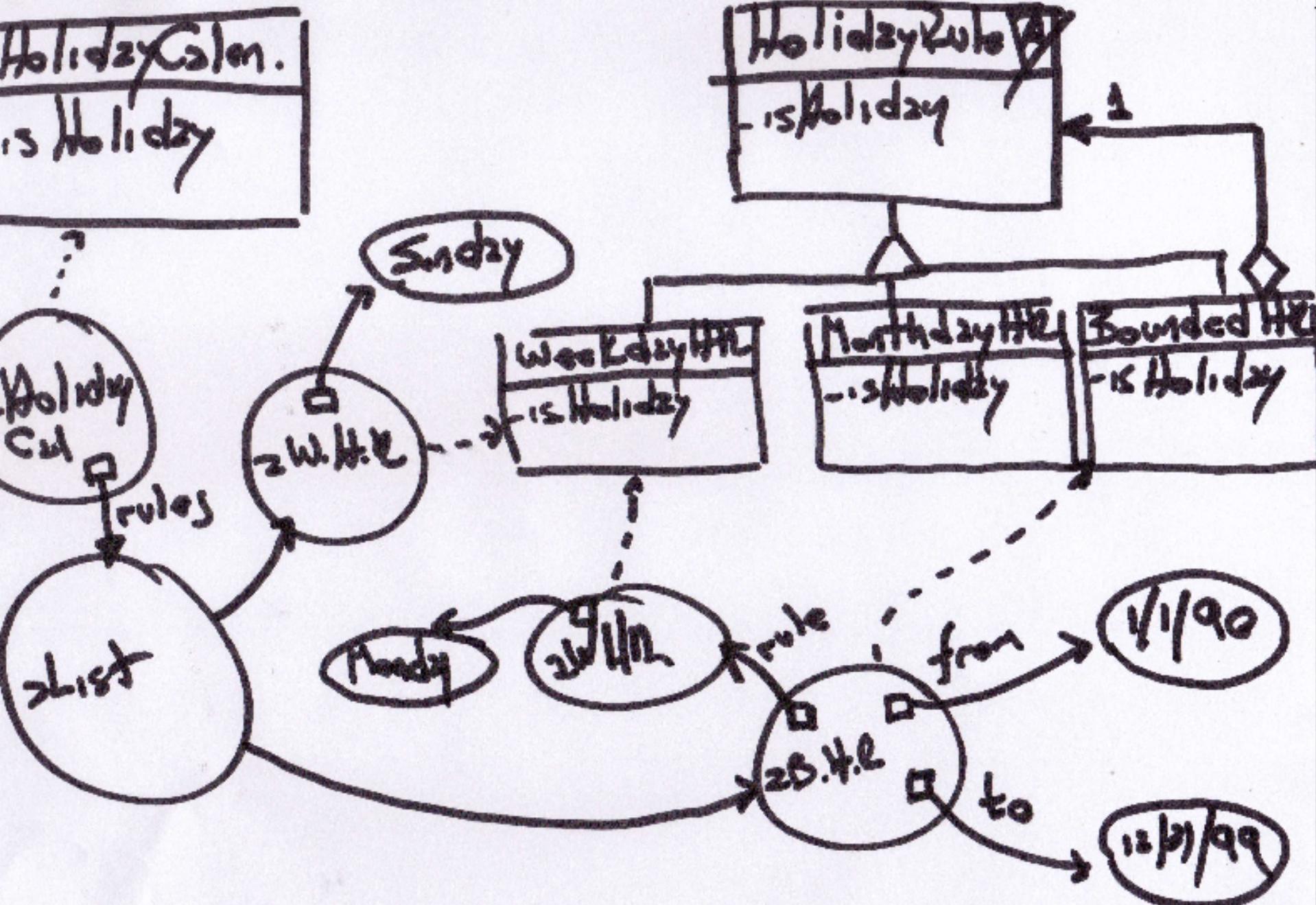
- Obtuvimos feedback inmediato sobre un error!
- Permite asegurarnos que aquello que aprendimos sigue funcionando con lo nuevo que aprendemos

Nuevo Requerimiento!

- Los feriados son válidos en un intervalo de tiempo! Ejemplos:
 - A partir del 2/8/2002 el 24/3 es feriado
 - Del 1/1/1990 al 31/12/1999 fueron feriado los Lunes

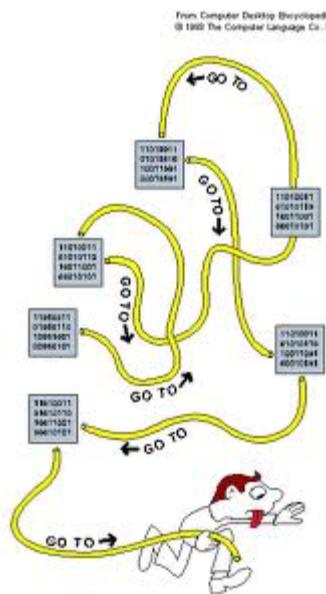






Conclusiones - Diseño

- El código nos habla!
- La solución anterior funcionaba pero no escalaba
- Test difíciles de escribir son un indicio de mal diseño!



Conclusiones - Diseño

- TDD no implica buen diseño
- Los buenos diseños son hechos por buenos diseñadores!



Conclusiones - Diseño

- La Nueva Solución implica un “Salto Conceptual” muy grande
 - Es lo que queremos lograr en esta materia



Conclusiones - Impacto

- Pudimos implementar un nuevo requerimiento asegurando calidad en lo existente
- Se hizo un cambio de diseño muy grande, con un impacto muy fuerte...
 - ... pero no tuvimos miedo en hacerlo
 - ... pudimos asegurar que el cambio no “rompiera nada”





Recapitulando

¿Por qué TDD sirve?

- Porque se basa en resolver casos concretos → Misma técnica que usamos para aprender
- Porque “recuerda” lo que aprendimos → Nos permite asegurar que cuando aprendemos algo nuevo, lo viejo sigue siendo válido

¿Por qué TDD sirve?

- Porque si nos dimos cuenta que habíamos aprendido algo incorrecto, nos indica rápidamente dónde tenemos que corregir esos errores
- Porque usa la computadora como herramienta de aprendizaje, no le exige a nuestra cabeza estar recordando todos los detalles

Conclusiones sobre la técnica...

- Es más que simplemente testing...
- Herramienta de Aprendizaje
 - Ejemplos concretos
 - Es cómo empezar por la UI con la ventaja de que son tests
 - Aprendizaje potenciado por la “maniabilidad” de la computadora
 - Feedback inmediato
- El test es un medio para lograr un fin, el fin es tener código confiable y “lindo”
- ¡Un programador no es bueno si no sabe testear!

Ventajas

- Aprendizaje Concreto
- Solo se escribe código testeado
- Ayuda a ir de lo Concreto a lo Genérico
- Refactorización constante = Cálidad de Código y Diseño

Ventajas

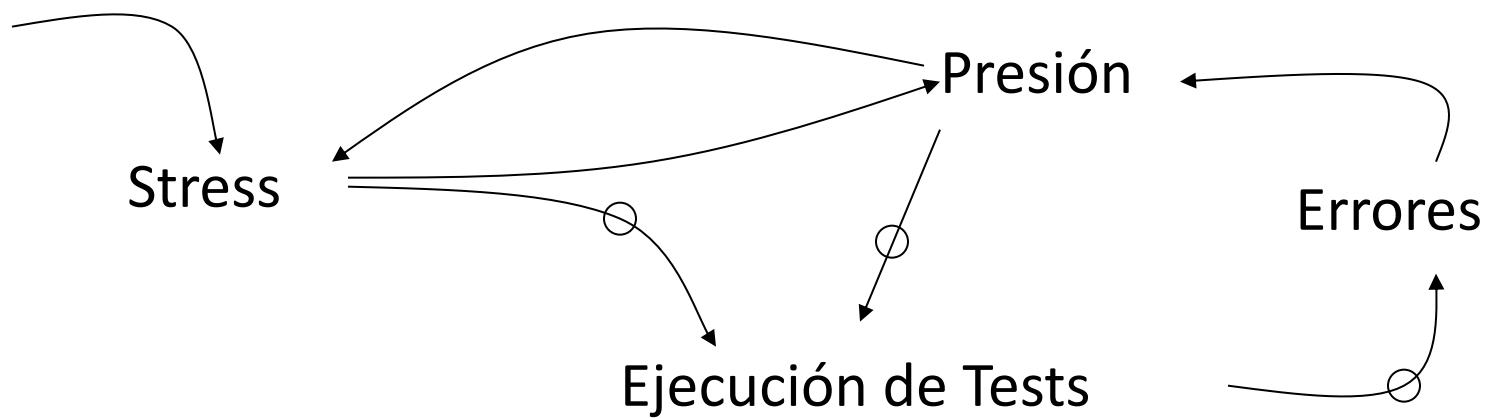
- Seguridad Frente al Cambio
 - Evita el efecto “sistema legacy”
 - Rompe el miedo del “sistema está en producción”, “no lo arregles si no está roto”
- Ayuda a Generar Mejores Diseños
 - Menor Acoplamiento
 - Mejor Distribución de Responsabilidades

Desventajas

- No es fácil empezar a usarlo
- Hay que romper una “barrera” de muchos años, de estar acostumbrado a otra técnica de desarrollo, hay que llegar a estar “Test Infected”
- Cambio Cultural fuerte
- Tiempo mayor de desarrollo, pero el balance al final es mejor puesto que disminuyen retrabajos, errores, etc.

La Espiral de la Muerte

► “No hay tiempo para testear”



¡Siempre Recordar!



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

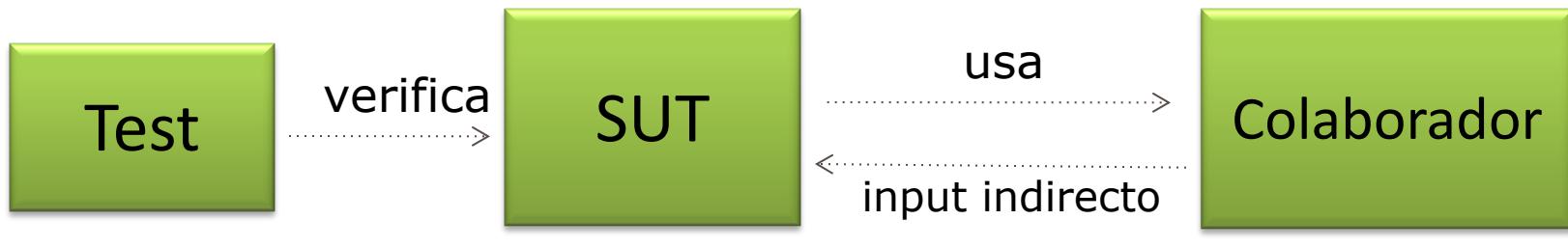
Estadísticas de Calidad

- Misma cantidad de líneas de código que las del sistema
- Correr todos los tests no debe llevar mucho tiempo, si no uno los deja de correr
- Mal uso:
 - Mucho código de preparación de test (setup)
 - Duplicación de código de preparación
 - Tests que tardan mucho
 - Tests frágiles
 - Tests Erráticos



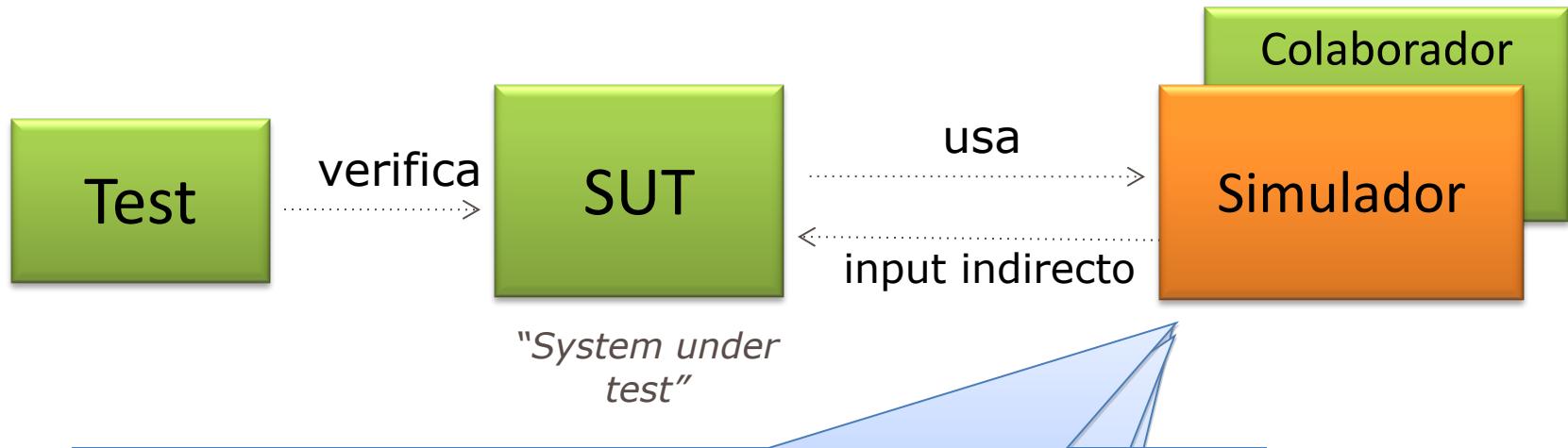
Objetos Simuladores

El problema:



- "System under test"
- a) no está disponible en el ambiente de test (ej: mail server)
 - b) hace más lenta la ejecución del test o el setup de los datos (ej: base de datos)
 - c) ejecutan en forma asíncrona (ej: colas de mensajes)
 - d) no devuelven los resultados necesarios para nuestro test (ej: simulación de errores)

La solución: Objectos Simuladores

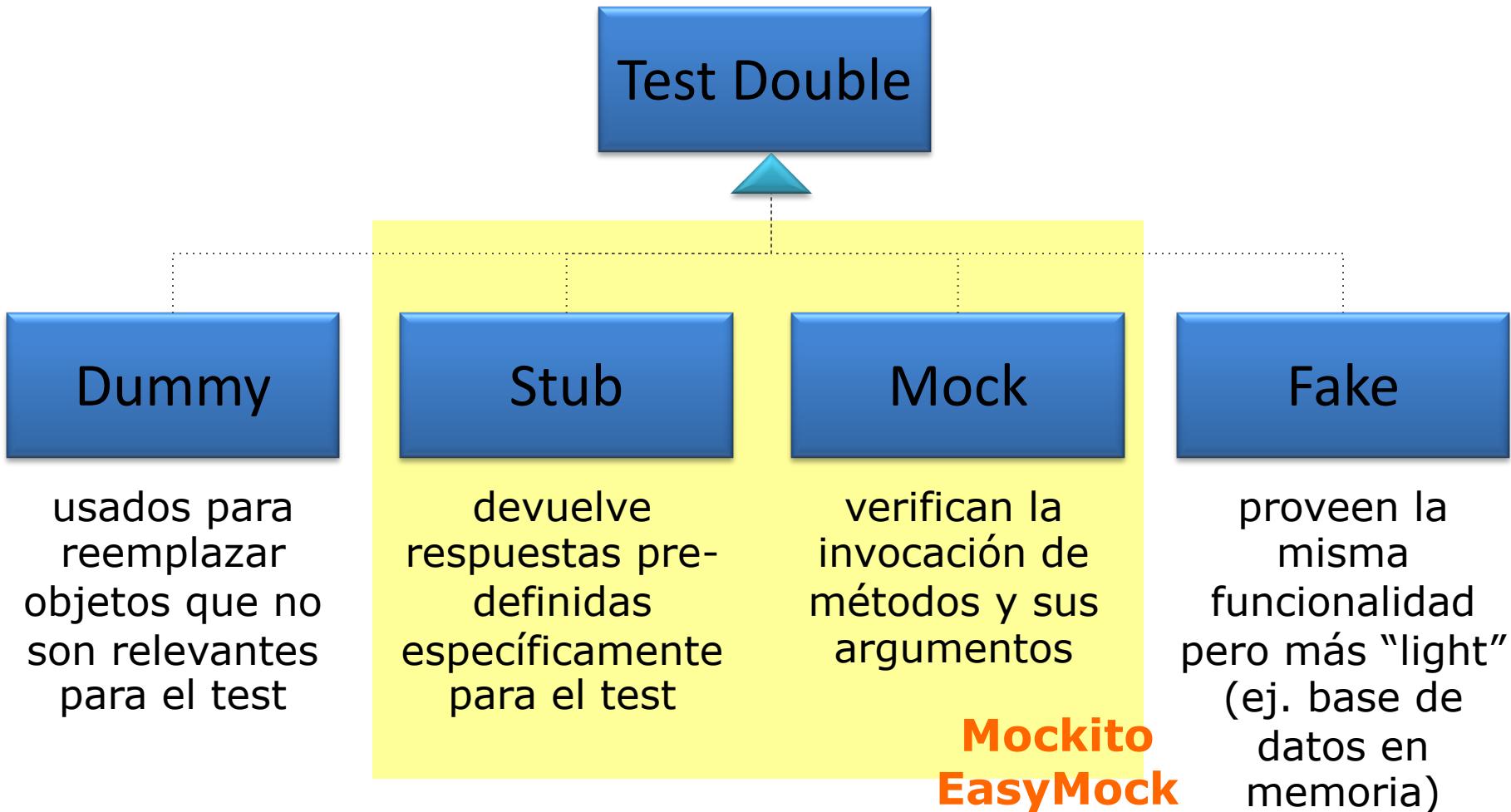


Es una implementación “de mentira” de la interfaz del colaborador

Reemplaza al objeto verdadero durante la ejecución del test

Pueden producir cualquier valor que necesite el test

Familia de Objectos Simuladores



Testear usando simuladores

1. Crear una instancia simulada de la clase real
2. Configurar el comportamiento del objeto simulador
3. Configurar al SUT para que use el objeto simulador en lugar del objeto real
4. Realizar la ejercitación de la funcionalidad a testear
5. Verificar que el SUT se comportó de la manera esperada

Ejemplo con Mocks

```
public class ProductRemovalTest {  
    private ProductSystem productSystem;  
    private AuditService auditService;  
  
    @Before  
    public void setupProductSystem() {  
        productSystem = new DefaultProductSystem();  
  
        auditService = mock(AuditService.class);  
        productSystem.setAuditService(auditService);  
    }  
  
    @Test  
    public void shouldCreateAuditRecordWhenProductRemoved() {  
        //setup  
        Product milk = milkProduct();  
        productSystem.add(milk);  
  
        //exercise  
        productSystem.remove(milk);  
  
        //verify  
        verify(auditService).audit(ProductManagerImpl.REMOVE_PRODUCT_ACTION,  
            milk.getId());  
    }  
    ...  
}
```

The diagram illustrates the workflow of a JUnit test using mocks. It consists of four yellow callout boxes with arrows pointing to specific lines of code:

- A yellow box labeled "Crear mock" points to the line `auditService = mock(AuditService.class);`.
- A yellow box labeled "Injectar el mock" points to the line `productSystem.setAuditService(auditService);`.
- A yellow box labeled "Invocar la funcionalidad a testear" points to the line `productSystem.remove(milk);`.
- A yellow box labeled "Verificar que se cumplieron las expectativas" points to the line `verify(auditService).audit(ProductManagerImpl.REMOVE_PRODUCT_ACTION, milk.getId());`.

Ejemplo con Stubs dinámicos

```
public class TicketServiceTest {  
    @Test  
    public void shouldNotSaleWithStolenCreditCard() {  
        MerchantProcessor merchantProcessor = new MerchantProcessorStub (  
            creditCard -> throw new RuntimeException("Stolen credit card"));  
  
        Cashier cashier = new Cashier (createValidCard(),  
            createStolenCreditCard(), emptySalesBook(), merchantProcessor );  
  
        try {  
            cashier.checkOut();  
            fail();  
        } catch (RuntimeException ex) {  
            assertEquals("Stolen credit card", ex.getMessage());  
            assertTrue(cashier.salesBook().isEmpty());  
        }  
  
    }  
    ...  
}
```

Crear stub

Configurar el stub para que devuelva los valores deseados

Injectar el stub

Invocar la funcionalidad a testear

Verificar los resultados

Ejemplo con Stubs dinámicos

```
public class TicketServiceTest {  
    @Test  
    public void shouldNotSaleWithStolenCreditCard() {  
        MerchantProcessor merchantProcessor = mock(MerchantProcessor.class);  
        when(merchantProcessor.debit(anyNumber(), anyCreditCard())).  
            thenThrow(RuntimeException("Stolen credit card"));  
  
        Cashier cashier = new Cashier(createValidCard(),  
            createStolenCreditCard(), emptySalesBook(), merchantProcessor);  
  
        try {  
            cashier.checkOut();  
            fail();  
        } catch (RuntimeException ex) {  
            assertEquals("Stolen credit card", ex.getMessage());  
            assertTrue(cashier.salesBook().isEmpty());  
        }  
    }  
    ...  
}
```

The diagram illustrates the five steps of a dynamic stub test:

- Crear stub**: Points to the line `MerchantProcessor merchantProcessor = mock(MerchantProcessor.class);`.
- Configurar el stub para que devuelva los valores deseados**: Points to the line `when(merchantProcessor.debit(anyNumber(), anyCreditCard())).thenThrow(RuntimeException("Stolen credit card"));`.
- Injectar el stub**: Points to the line `Cashier cashier = new Cashier(createValidCard(), createStolenCreditCard(), emptySalesBook(), merchantProcessor);`.
- Invocar la funcionalidad a testear**: Points to the line `cashier.checkOut();`.
- Verificar los resultados**: Points to the line `fail();`.

Conclusiones sobre Mocks

- Evitarlos lo más posible!
 - Pensar por qué
- Genera tests frágiles (tests de caja blanca)
- Es mejor tener factories de objetos que son utilizados en los tests
 - Pensar por qué

Objetos Simuladores - Conclusiones

- Usarlos para que el test esté “en control de todo”
- Usarlos para representar todo aquello que sea externo al modelo
- Utilizarlos para poder concentrarse en testear de manera desacoplada el modelo



Técnicas de Testing

Técnicas de Testing

- Usar una nomenclatura clara para los nombres de los tests
 - Ej: testWhen...Then...Should...
Ejemplo:
`whenCustomerHasInvalidIdThenAddingItShouldRaiseException`
- 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

- Usar setUp y tearDown con cuidado, puesto que genera acoplamiento
- 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”
- Agrupar assertions repetidas para generar semántica

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 {  
    ...  
    fail();  
} catch (ExpectedException e) {  
    asseraciones sobre e  
    asseraciones sobre invariantes}  
}
```

➤ Reificar en cuando hay closures (Ej. Smalltalk)