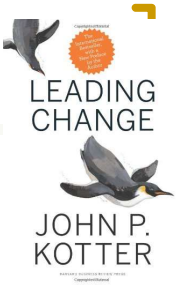
Refactoring – Agile Test Driven Development

Dra. Alejandra Garrido
Objetos 2 – Fac. De Informática – U.N.L.P.
alejandra.garrido@lifia.info.unlp.edu.ar

Costo del mantenimiento

- Mantenimiento
  - correctivo, evolutivo, adaptativo, perfectivo, preventivo.
- Costo de Mantenimiento:
  - **Entender código existente:** 50% del tiempo de mantenimiento
- La incapacidad de cambiar el software de manera rápida y segura implica que se pierden oportunidades de negocio

Situación actual



- Por qué se pierden oportunidades de negocio?

2012

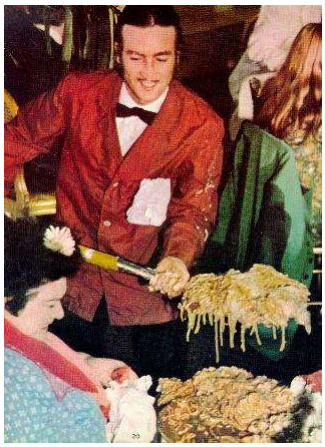
  - "The rate of **change** in business is growing **exponentially**"
  - "Exponential change means exponential smaller reaction time"

Leyes de Lehman

- Continuing Change
  - Los sistemas deben adaptarse continuamente o se vuelven progresivamente menos satisfactorios
- Increasing Complexity
  - A medida que un sistema evoluciona su complejidad se incrementa a menos que se trabaje para evitarlo
- Continuing Growth
  - la funcionalidad de un sistema debe ser incrementada continuamente para mantener la satifacción del cliente
- Declining Quality
  - La calidad de un sistema va a ir declinando a menos que se haga un mantenimiento riguroso

Big Ball of Mud

- Querriamos tener arquitecturas de software elegantes, diseños que usen patrones y código flexible y reusable.
- En realidad tenemos toneladas de “spaghetti code”, con poca estructura, atado con alambre y duct tape.
- Es una pesadilla, pero sin embargo subsiste. ¿Por qué?



- “Big Ball of Mud”. Brian Foote and Joe Yoder. Pattern Languages of Programs 4. Addison-Wesley 2000.

BBoM modernos

```
if (evt1.AbsoluteTime < evt2.AbsoluteTime) {
        return -1;
    } else if (evt1.AbsoluteTime > evt2.AbsoluteTime) {
        return 1;
    } else {
        // a iguar valor de AbsoluteTime, los channelEvent tienen prioridad
        if(evt1.MidiEvent is ChannelEvent && evt2.MidiEvent is MetaEvent) {
            return -1;
        } else if(evt1.MidiEvent is MetaEvent && evt2.MidiEvent is ChannelEvent){
            return 1;
        //      si ambos son channelEvent, dar prioridad a NoteOn == 0 sobre NoteOn > 0
        } else if(evt1.MidiEvent is ChannelEvent && evt2.MidiEvent is ChannelEvent) {

            chanEvt1 = (ChannelEvent) evt1.MidiEvent;
            chanEvt2 = (ChannelEvent) evt2.MidiEvent;

            // si ambos son NoteOn
            if(     chanEvt1.EventType == ChannelEventType.NoteOn
                    && chanEvt2.EventType == ChannelEventType.NoteOn){

                //      chanEvt1 en NoteOn(0) y el 2 es NoteOn(>0)
                if(chanEvt1.Arg1 == 0 && chanEvt2.Arg1 > 0) {
                    return -1;
                //      chanEvt1 en NoteOn(0) y el 2 es NoteOn(>0)
                } else if(chanEvt2.Arg1 == 0 && chanEvt1.Arg1 > 0) {
                    return 1;
                } else {
                    return 0;
```

Cómo escribir código inmantenible?

```
for(j=0; j<array_len; j+ =8)
{
    total += array[j+0 ];
    total += array[j+1 ];
    total += array[j+2 ]; /* Main body of
    total += array[j+3]; * loop is unrolled
    total += array[j+4]; * for greater speed.
    total += array[j+5]; */
    total += array[j+6 ];
    total += array[j+7 ];
    }
```


BBoM en Smalltalk

```
m1: anObject
| a |
a := OrderedCollection new.
anObject do: [:x| x \\ 2 = 1 = true ifTrue: [a
add: x]].
^a
```

```
do: var
    ^self perform: (var instVarAt: 2)
```

¿Qué hacemos con el BBofM?

- BBofM existen porque funcionan, y han probado funcionar mejor que otras propuestas
- La arquitectura casual es natural en las primeras etapas del desarrollo
- Debemos aspirar a mejorar, reconociendo las fuerzas que llevan al deterioro de la arquitectura y aprendiendo a reconocer las oportunidades para mejorarla

"Architectural insight is not the product of master plans, but of hard won experience"

Throwaway Code

- Cuando estamos costruyendo un sistema solemos empezar por un prototipo
- Codificamos rápido para probar una idea, un concepto, con la intención de que después se haga bien
- Se hace lo más simple, expeditivo y descartable posible
- Pero el código queda instalado

Piecemeal Growth

- Por más que hayamos comenzado con un diseño de arquitectura elegante, ocurren:
  - aparición de nuevos requerimientos
  - cambios en el entorno / tecnología
  - bug fixing
  - cambios, cambios, cambios
- Y se agrega código como un “Piecemeal growth” continuo que corroe las mejoras arquitecturas

Diseñar es difícil!

- Los elementos distintivos de la arquitectura de un sistema no surgen hasta *después* de tener código que funciona
- No se trata sólo de agregar, sino de *adaptar*, *transformar, mejorar*
- Construir el sistema perfecto es imposible
- Los errores y el cambio son inevitables
- Hay que aprender del **feedback**

La iteración es fundamental

- “Reusable software is the result of many design iterations. Some of these iterations occur after the software has been reused”
- Los cambios de una iteración a la siguiente pueden involucrar únicamente cambios estructurales entre componentes existentes que no cambian la funcionalidad

(Bill Opdyke. 1992)

Refactoring

- "Refactoring Object-Oriented Frameworks".
  - Bill Opdyke, PhD Thesis. Univ. of Illinois at Urbana-Champaign (UIUC). 1992. Director: Ralph Johnson.
- Refactoring es una transformación que preserva el comportamiento, pero mejora el diseño

Refactoring como un proceso

- Es el proceso a través del cual se cambia un sistema de software
  - para ***mejorar*** la organización, legibilidad, adaptabilidad y mantenibilidad del código luego que ha sido escrito
  - que ***NO altera*** el comportamiento externo del sistema

Características del Refactoring

- Implica
  - Eliminar duplicaciones
  - Simplificar lógicas complejas
  - Clarificar códigos
- A través de cambios *pequeños*
  - Hacer muchos cambios pequeños es más fácil y más seguro que un gran cambio
  - Cada pequeño cambio pone en evidencia otros cambios necesarios
- Testear después de cada cambio

Un mal diseño no es grave

- no afecta al compilador, este no sabe si el código es claro o es “imposible”

- Hasta que hay que hacer cambios!!!
  - participan desarrolladores, quienes se preocupan o son afectados
  - no es fácil descubrir donde cambiar
  - es probable que se introduzcan errores

Importancia del refactoring

- Nuestra única defensa contra el deterioro del software.
- Facilita la incorporación de código
- Permite *agregar patrones* después de haber escrito el programa
- Permite preocuparse por la generalidad mañana.
- Es decir, permite ser ágil en el desarrollo

Surgen las metodologías ágiles

- Las metodologías ágiles o "lightweight" son:
  - adaptativas (como opuesto a predictivas)
  - orientadas a la gente (y no al proceso)
- Reconocen la gran diferencia entre el diseño y la construcción en la ingeniería "civil", y el diseño y la construcción en software.

Características principales

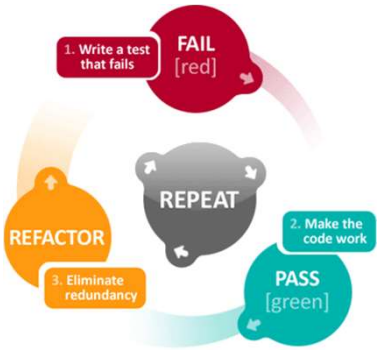
Agile Approach
Mismo grupo de personas para todo el desarrollo que trabajan en un mismo espacio
Comunicación de calidad
Desarrollo iterativo e incremental
Producto funcionando en cada "build"
Se valora el feedback
Cambios bienvenidos "changes embraced"

Por dónde empezar

- Si se debe ir tomando de a un requerimiento o pocos por vez para tener un producto funcionando al final de cada iteración, no cuento con un diseño completo para empezar a desarrollar
- Qué cosa guía el desarrollo? Por dónde empezar si no es de un diseño completo?

Test Driven Development

- Empezar por el test!



Test Driven Development (TDD)

- Combina:
  - *Test First Development*: escribir el test antes del código que haga pasar el test
  - *Refactoring*
- Objetivo:
  - pensar en el diseño y qué se espera de cada requerimiento antes de escribir código
  - escribir código limpio que funcione (como técnica de programación)

¿Por qué no dejar testing para el final?

- Para conocer cuál es el final
  ¿De qué otra manera podemos saber que terminamos?
- Para mantener bajo control un proyecto con restricciones de tiempo ajustadas (permite estimar)
- Para poder refactorizar rápido

¿Qué logramos con TDD?

- Diseño simple
- Saber cuándo terminamos
- Confianza para el desarrollador
- Coraje para refactorizar
- Documentación práctica que evoluciona naturalmente
- Incrementar la calidad del software

Incrementar la calidad del software

- Mejorar la calidad del software, en dos aspectos:
  - que el software esté *construido correctamente*
  - que el software construido sea el *correcto*

Filosofía de TDD

- Vuelco completo al desarrollo de software tradicional. En vez de escribir el código primero y luego los tests, se escriben los tests primero antes que el código.
- Se escriben tests funcionales para capturar use cases que se validan automáticamente
- Se escriben test de unidad para enfocarse en pequeñas partes a la vez y aislar los errores

Filosofía de TDD (cont.)

- No agregar funcionalidad hasta que no haya un test que no pasa porque esa funcionalidad no existe.
- Una vez escrito el test, se codifica lo necesario para que todo el test suite pase.
- Pequeños pasos: un test, un poco de código
- Una vez que los tests pasan, se refactoriza para asegurar que se mantenga una buena calidad en el código.

Algunas reglas de TDD

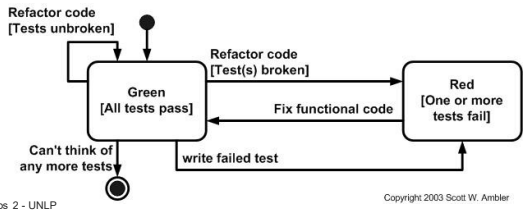
- Diseñar incrementalmente:
  - teniendo código que funciona como feedback para ayudar en las decisiones entre iteraciones.
- Los programadores escriben sus propios tests:
  - no es efectivo tener que esperar a otro que los escriba por ellos.
- El diseño debe consistir de componentes altamente cohesivos y desacoplados entre si:
  - mejora evolución y mantenimiento del sistema.

Granularidad

- Test de aceptación
  - Por cada funcionalidad esperada.
  - Escritos desde la perspectiva del cliente
- Test de unidad
  - aislar cada unidad de un programa y mostrar que funciona correctamente.
  - Escritos desde la perspectiva del programador
- Test de integración

Automatización de TDD

- TDD asume la presencia de un framework de unit-testing (gratuito: xUnit family o comercial).
- Sin herramientas que automaticen el testing, TDD es prácticamente imposible.
- El ambiente de desarrollo debe proveer respuesta rápida ante cada cambio (build en 10 minutos).



Framework Xunit

- La primera herramienta de testing automático fue Sunit, escrito por Kent Beck para Smalltalk
- Hoy en día existe para muchos lenguajes de programación

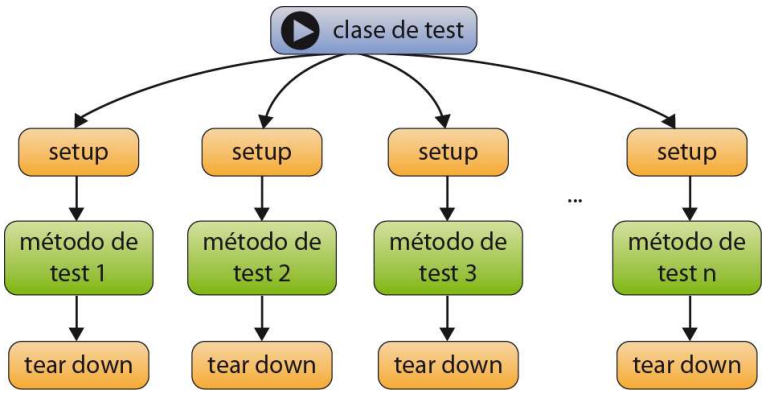
Test de unidad (Xunit)

- Testeo de la *mínima unidad de ejecución*.
- En OOP, la mínima unidad es un método.
- **Objetivo**: aislar cada parte de un programa y mostrar que funciona correctamente.
- Cada test confirma que un método produce el output esperado ante un input conocido.
- Es como un contrato escrito de lo que esa unidad tiene que satisfacer.

Partes de un test de unidad

- Fase 1: Fixture set up:
  Preparar todo lo necesario para testear el comportamiento del SUT
- Fase 2: Exercise:
  Interactuar con el SUT para ejercitar el comportamiento que se intenta verificar
- Fase 3: Check:
  Comprobar si los resultados obtenidos son los esperados, es decir si el test tuvo éxito o falló
- Fase 4: Tear down
  Limpiar los objetos creados para y durante la ejecución del test

Tests con xUnit



Ejemplo

```
Integer>>factorial
   self < 0
      ifTrue: [^self error: 'Function out of range'].
   ^self = 1
      ifTrue: [1]
      ifFalse: [self * (self - 1) factorial]
```

Subclase de TestCase

- Subclase de TestCase: #TestInteger
- Variables de instancia : ‘zero small big neg'

setUp y tearDown

setUp

```
zero := 0.
small := 2.
big := 10.
neg := -1
```

tearDown

Mars Global Surveyor

Casos de testing

testFactorial

```
self assert: (small factorial = 2).
self assert: (big factorial =  3628800).
self should: [neg factorial] raise: Error
self assert: (zero factorial = 1)
```

Integer>>factorial

```
self < 0
   ifTrue: [^self error: 'Function out of range'].
self = 1
   ifTrue: [1]
   ifFalse: [self * (self - 1) factorial]
```

Consideraciones

- should: o assert: ?
- ¿Qué valores testear?
- ¿Cuántos aspectos testear por test?

should: o assert: ?

- should: aBlock
- should: aBlock raise: anException <- más usado así

- assert: aBoolean
- assert: actual equals: expected

¿Qué valores testear?

- Escribir casos de testing es deseable pero es costoso
- Testear todos los valores no es práctico
- Se busca encontrar casos importantes
- Lo importante es conocer la 'cobertura' de los casos de testing
  - Particiones Equivalentes
  - Valores de Borde

Particiones Equivalentes

- Tratar conjuntos de datos como el mismo (si un test pasa, otros similares pasarán)
- Para rangos, elegir un test en el rango, un test en cada extremo.
- `Debe aceptar años entre 1-2050.`
  - `Casos de testing: 0, 1876 , 2076.`
- Para conjuntos, elegir uno en el conjunto, uno fuera del conjunto.
- `Passwords deben ser de 6-8 caracteres de largo:`
  - `Casos de testing: ab, abcdefg, abcdegujswidn`

Valores de Borde

- La mayoría de los errores ocurren en los bordes o límites entre conjuntos
- Debe aceptar años entre 1-2050.
  - Casos de testing: 0, 1, 2050 , 2051.
- Passwords deben ser de 6-8 caracteres de largo:
  - Casos de testing: abcde, abcdef, abcdefgh, abcdefghi
- División por cero es un borde “computacional”.
- Los “Valores de Borde” complementa “Particiones Equivalentes”.

Testeo todos los aspectos / valores en 1 método

```
testFactorial

   self assert: (small factorial = 2).
   self assert: (big factorial =  3628800).
   self should: [neg factorial] raise: Error
   self assert: (zero factorial = 1)
```

O testeo 1 aspecto por vez

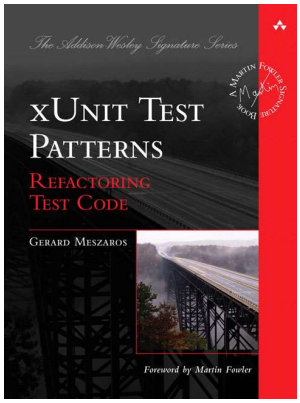
```
testSmallFactorial
   self assert: (small factorial = 2)

testBigFactorial
   self assert: (big factorial =  3628800)

testNegNumberFactorial
   self should: [neg factorial] raise: Error

testZeroFactorial
    self assert: (zero factorial = 1)
```


XUnit Test Patterns



Tamaño de los métodos de testing

- Postura más purista: verificar una sola condición por cada test.
- Ventaja para detectar errores: cuando un test falla se puede saber con precisión qué está mal con el SUT.
- Un test que verifica una única condición ejecuta un solo camino en el código del SUT
- Debemos aislar cada camino de ejecución y escribir un método de test que verifiquen las condiciones necesarias para testear ese camino → costoso
- Los test con varias condiciones surgen para evitar las repetidas configuraciones del estado inicial de un test.

Tamaño de las clases de testing

- Una subclase de TestCase por cada clase
  - *TestInteger*
- Una clase testcase por cada característica (feature)
  - *TestIntegerFactorial*
- Una clase testcase por cada fixture
  - *TestLargeIntegerFactorial*

Qué se hace en Fixture setup?

- La lógica del *fixture setup* incluye:
  - El código para instanciar el SUT
  - El código para poner el SUT en el estado apropiado
  - El código para crear e inicializar todo aquello de lo que el SUT depende o que le va a ser pasado como argumento


```
FlighStateTestCase>>testStatusInitial
    "in-line setup"
    departureAirport := Airport newIn: 'Calgary' name: 'YYC'.
    destinationAirport := Airport newIn: 'Toronto' name: 'YYZ'.
    flight := Flight newNumber: '0572'
        from: departureAirport  to: destinationAirport.

    "exercise SUT and verify outcome"
    self assert: (flight getStatus = 'PROPOSED')
}
```

```
FlighStateTestCase>>testStatusCancelled
    "in-line setup"
    departureAirport := Airport newIn: 'Calgary' name: 'YYC'.
    destinationAirport := Airport newIn: 'Toronto' name: 'YYZ'.
    flight := Flight newNumber: '0572'
                from: departureAirport  to: destinationAirport.
    flight cancel.
    "exercise SUT and verify outcome"
    self assert: (flight getStatus = 'CANCELLED').
  }
"idem para scheduled"
```


Fixture setup patterns

- In-line Setup
  (inlined en el método de test)
- Delegated Setup
  (extrayendo el inlined setup)
- Implicit Setup
  (en el método setUp)
- Hybrid Setup
  (combinación de los tres anteriores)

Mantener los test independientes

- Mientras mayor sea la dependencia entre los test, menos exacta será información acerca de un fallo en particular.
- Si tenemos tests que dependen de la ejecución de otros, los cambios introducidos en estos últimos afectarán el comportamiento de los primeros.
- Al hacer que los tests sean independientes de la ejecución de otros, los fallos indicarán información mucho más útil y los test serán más confiables.

Aislar el SUT

- Las distintas funcionalidades del SUT en muchos casos dependen entre sí o de componentes ajenos al SUT.
- Cuando se producen cambios en los componentes de los que depende el test, es posible que este último empiece a fallar.
- Al testear funcionalidades del SUT es preferible no depender de componentes del sistema ajenos al test.

Mock objects

- **Mock objects** son “simuladores” que imitan el comportamiento de otros objetos de manera controlada.
- Por ejemplo, un reloj alarma que debe hacer sonar una campana a determinada hora.
  Para testear el test debería esperar la hora de alarma → se usa un mock object que provee la hora de alarma.

Cuando usar Mock objects

- Cuando el objeto real es un objeto complejo que:
  - retorna resultados no-deterministicos (ej., la hora actual o la temperatura actual).
  - tiene estados que son dificiles de reproducir (ej., un error de network);
  - es lento (ej, necesita inicializar una transaccion a la base de datos);
  - todavia no existe;
  - tiene dependencias con otros objetos y necesita ser aislado para testearlo como unidad.

Cuándo/Cómo/Por qué testear

- "Test with a purpose" (Kent Beck)
- Saber por qué se testea algo y a qué nivel debe testearse.
- El objetivo de testear es encontrar bugs
- Se puede aplicar a cualquier artefacto del desarrollo
- Se debe testear temprano y frecuentemente
- Testear tanto como sea el **riesgo** del artefacto
- Un test vale más que la opinión de muchos

Bibliografia

- "Big Ball of Mud". Brian Foote and Joe Yoder. Pattern Languages of Programs 4. Addison-Wesley 2000.
- "Refactoring". Martin Fowler. 1999
- "Test Driven Development: by Example". Kent Beck. Addison Wesley. 2002
- Kent Beck. "Simple Smalltalk Testing: with Patterns" http://swing.fit.cvut.cz/projects/stx/doc/online/english/tools/misc/testfram.htm
- "xUnit Test Patterns: Refactoring Test Code". Gerard Meszaros. Addison Wesley.2007

Videos interesantes

- TDD: The Bad Parts —

https://www.youtube.com/watch?v=xPL84vvLwXA

- Is TDD dead?

https://www.youtube.com/watch?v=z9quxZsLcfo&list=PLJb2p0qX8R_qSRhs14CiwKuDuzERXSU8m