Refactoring – 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

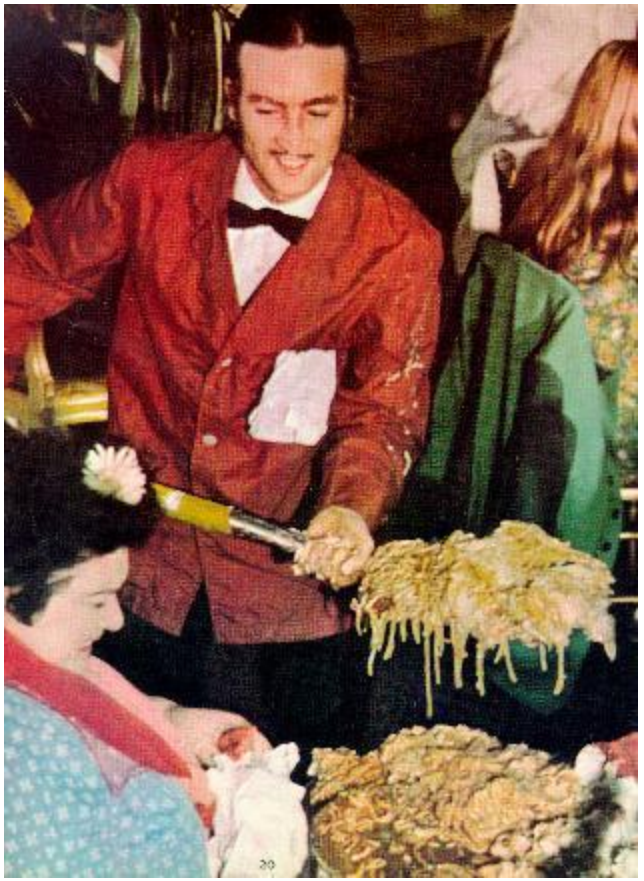
- Mantenimiento
  - correctivo, evolutivo, adaptativo, perfectivo, preventivo.
- Costo de Mantenimiento:
  - **Entender código existente:** 50% del tiempo de mantenimiento
- La incapacidad de cambiar el software de manera rápida y segura implica que se pierden oportunidades de negocio

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.

Patrones del Big Ball of Mud

- Comienza siendo “Throwaway code” que se instala y persiste, simplemente porque funciona
- Cambios de requerimientos y agregado de funcionalidad como un “Piecemeal growth” continuo que corroe las mejoras arquitecturas
- Dejamos un Façade alrededor de lo que no queremos mostrar o tocar, “Sweeping it under the rug”

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
```

```
selectOddNumbersFrom: aCollection
    ^aCollection select: [:each| each isOdd]
```

```
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 de 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”

Piecemeal Growth

- 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.*
- Construir el sistema perfecto es imposible
- Los errores y el cambio son inevitables
- Hay que aprender del **feedback**

Diseñar es difícil

- “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 as a transformation that preserves behavior

1992

Changing OO systems by Refactoring

- Restructurings in a class hierarchy
  E.g. “Create an abstract superclass”
  - “Creating Abstract Superclasses by Refactoring”. Opdyke & Johnson. ACM Conf. Computer Science. 1993

- Restructurings between components
  E.g. “Converting inheritance into aggregation”
  - “Refactoring and Aggregation”. Johnson & Opdyke. ISOTAS 1993.

Changing Smalltalk code automatically

- First refactoring tool: Refactoring Browser
- For Smalltalk code.
- In UIUC by John Brant & Don Roberts

1997

Refactoring by Fowler

- *Refactoring* (noun): “A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior”.

- *Refactor* (verb): “To restructure software by applying a series of refactorings without changing its observable behavior”

1999

Again: qué es Refactoring?

- 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

Proceso de 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

Por ejemplo…

```
CheckingAccount>>withdraw: amount
    (balance – amount > 0)
        ifTrue: [balance := balance – amount.
                transaction := WithdrawalTransaction
                        on: Date today
                        at: Time now
                        for: amount.
                transactions add: transaction].
```

CarefreeChecking: “allows overdraft; overdraft fee: 34; monthly fee”

PillarChecking: “balance > 5000; no monthly fee; interest bearing”

Catálogo de refactoring

- Para programas orientados a objetos
- Ej. “Pull Up Method”
- Pero también va más allá de clases y jerarquías
- Ej. Refactorings para simplificar expresiones condicionales : “Consolidate conditional expression”

```
double disabilityAmount() {
    if (_seniority < 2) return 0;
    if (_monthsDisabled > 12) return 0;
    if (_isPartTime) return 0;
    // compute the disability amount
```

```
double disabilityAmount() {
    if (isNotEligibleForDisability)
            return 0;
    // compute the disability amount
```

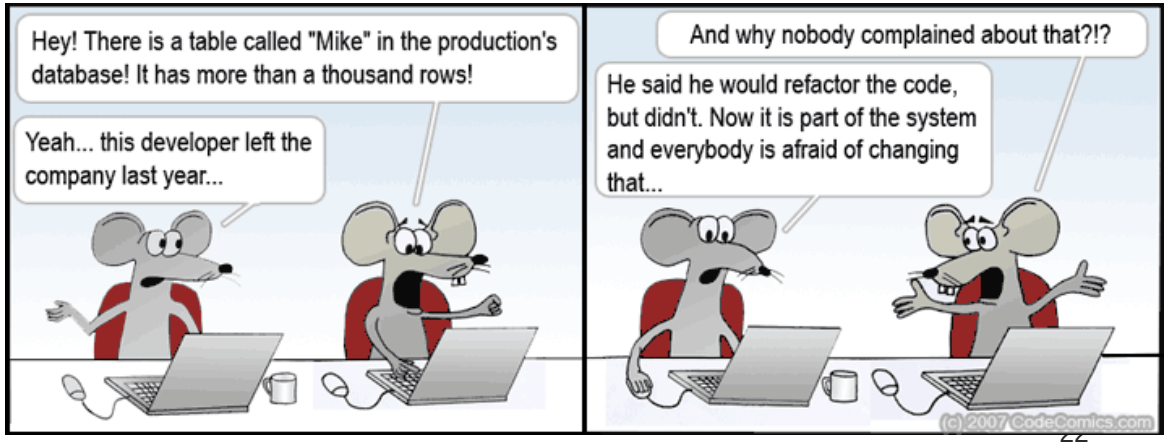
Cuando y donde aplicar refactoring?

- **Bad smells:** “structures in code that suggest the possibility of refactoring”.
- E.g.
  - Duplicated code
  - Long method
  - Data class
  - Long parameter list

1999

entonces ☞ Refactoring

- Se toma un código ***"con mal olor"*** producto de mal diseño
  - Codigo duplicado, no claro, complicado
- y se lo trabaja para obtener un buen diseño
- Cómo?
  - Moviendo un atributo de una clase a otra
  - Extrayendo código de un método en otro método
  - Moviendo código en la jerarquía
  - Etc etc etc …



¿Por qué refactoring es importante?

- Ganar en la comprensión del código
  - hay código que no se entiende, aunque le pongamos comentarios (desodorante)
  - facilita la detección de bugs
  - mejoras continuas al código lo hacen mas facil de trabajar

Automatización del refactoring

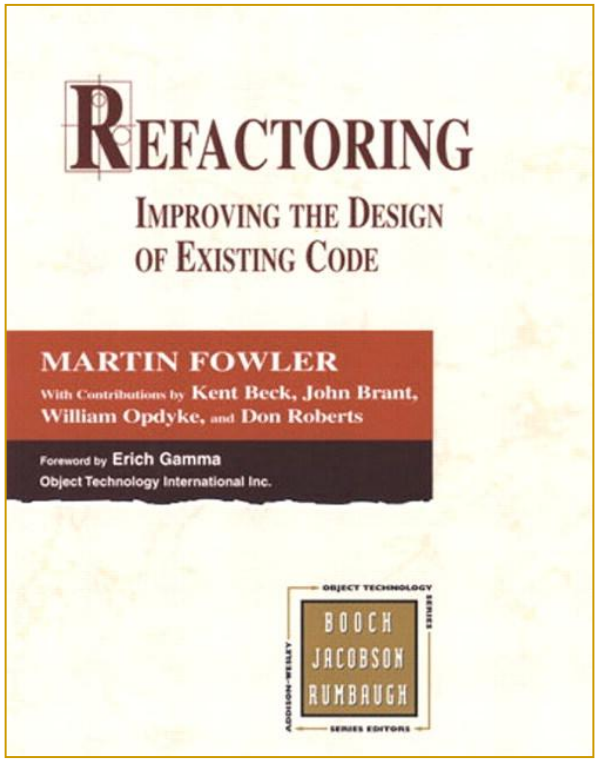
- Refactorizar a mano es demasiado costoso: lleva tiempo y puede introducir errores
- Surgen las herramientas de refactoring
- Características de las herramientas:
  - potentes para realizar refactorings útiles
  - restrictivas para preservar comportamiento del programa (uso de *precondiciones*)
  - interactivas, de manera que el chequeo de precondiciones no debe ser extenso

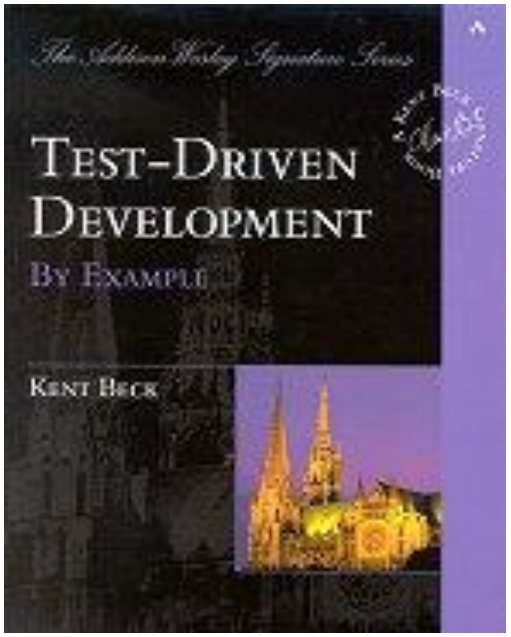
Cuando refactorizar

- Cuando un nuevo requerimiento se vea dificil de implementar
- Cuando una nueva funcionalidad agregada produjo código duplicado, inflexible, spaguetti.
- Cuando no puedes soportar mirar tu código

¿Por qué refactoring es importante?

- Nuestra única defensa contra el deterioro del software.
- Facilitar la incorporación de código
- Permite *agregar patrones* después de haber escrito el programa; permite *transformar un programa en framework*.
- Permite preocuparse por la generalidad mañana; hoy solo hay que hacerlo andar
  "*Make it work. Make it right. Make it fast*". Kent Beck.
- "*Necessary for beautiful software*". Ralph Johnson


REFACTORING
IMPROVING THE DESIGN OF EXISTING CODE
MARTIN FOWLER
With Contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts
Foreword by Erich Gamma
Object Technology International Inc.
OBJECT TECHNOLOGY SERIES
BOOCH
JACOBSON
RUMBAUGH
ADDISON-WESLEY
SERIES EDITORS


TEST-DRIVEN DEVELOPMENT
BY EXAMPLE
KENT BECK

1999

2002

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)

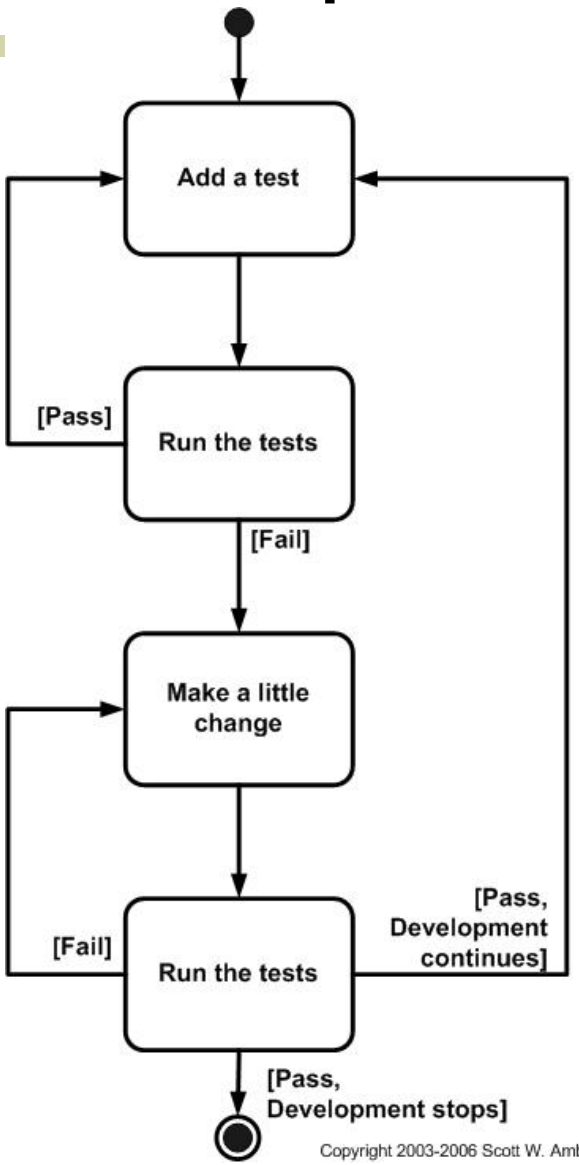
¿Por qué no dejar testing para el final?

- 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

- Para poder refactorizar rápido

Test First Development (TFD)



¿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

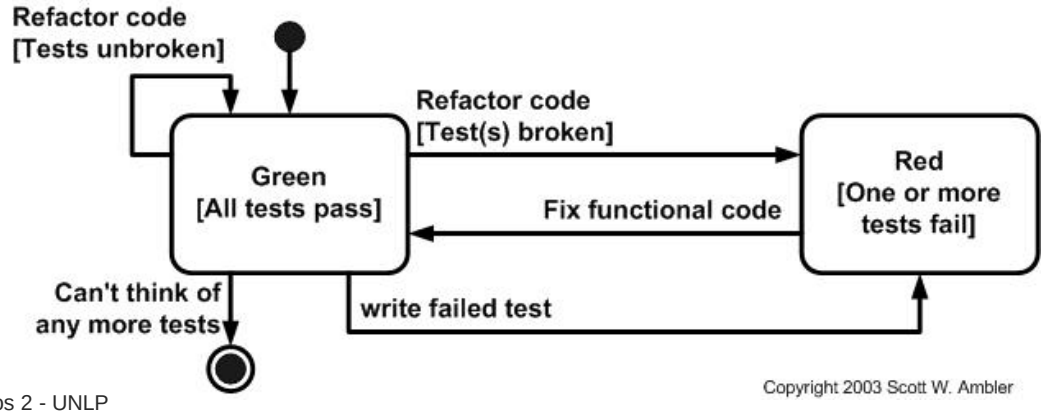
- Diseñar incrementalmente:
  - teniendo código que funciona como feedback para ayudar en las decisiones entre iteraciones.
- Los programadores escriben sus propios tests:
  - no es efectivo tener que esperar a otro que los escriba por ellos.
- El diseño debe consistir de componentes altamente cohesivos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

Granularidad

- Test funcional (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

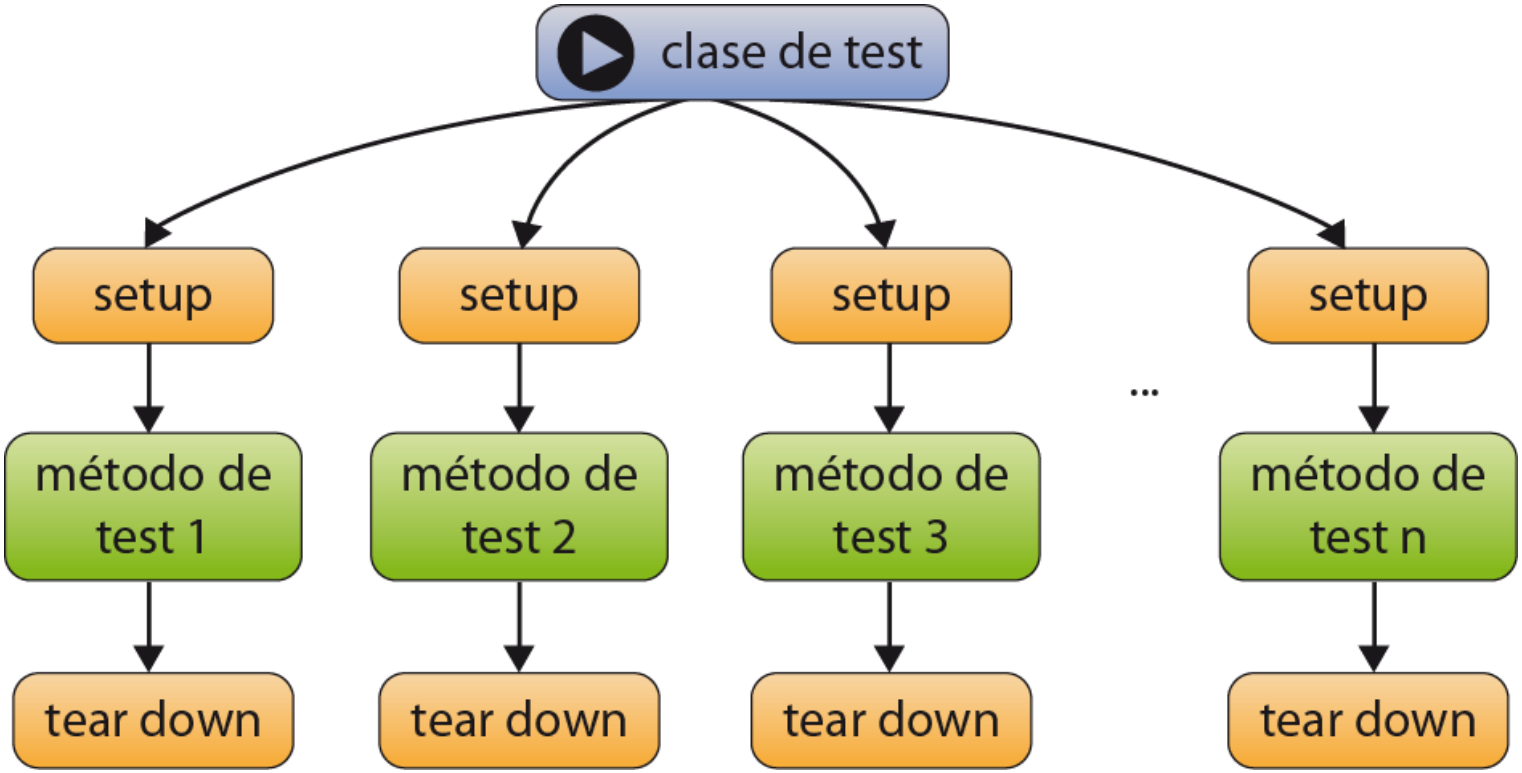
- TDD asume la presencia de un framework de unit-testing (gratuito: xUnit family o comercial).
- Sin herramientas que automaticen el testing, TDD es prácticamente imposible.
- El ambiente de desarrollo debe proveer respuesta rápida ante cada cambio (build en 10 minutos).



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

Tests con xUnit



Cuándo/Cómo/Por qué testear

- “Test with a purpose” (Kent Beck)
- Saber por que se testea algo y a que 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

- “Refactoring”. Martin Fowler. 1999
- “Test Driven Development: by Example”. Kent Beck. Addison Wesley. 2002
- “xUnit Test Patterns: Refactoring Test Code”. Gerard Meszaros. Addison Wesley.2007
- Kent Beck. “Simple Smalltalk Testing: with Patterns” http://www.xprogramming.com/testfram.htm

  http://home.arcor.de/fbdiplom/lit_pdf/bec00_beck_testing_framework.pdf