



# Introducción a Refactoring

Dra. Alejandra Garrido

Objetos 2 – Fac. De Informática – U.N.L.P.

[alejandra.garrido@lifa.info.unlp.edu.ar](mailto:alejandra.garrido@lifa.info.unlp.edu.ar)

# [Cambios



# [Leyes de Lehman]

## ■ Continuing Change

- Los sistemas deben adaptarse continuamente o se vuelven progresivamente menos satisfactorios

## ■ Continuing Growth

- la funcionalidad de un sistema debe ser incrementada continuamente para mantener la satisfacción del cliente

## ■ Increasing Complexity

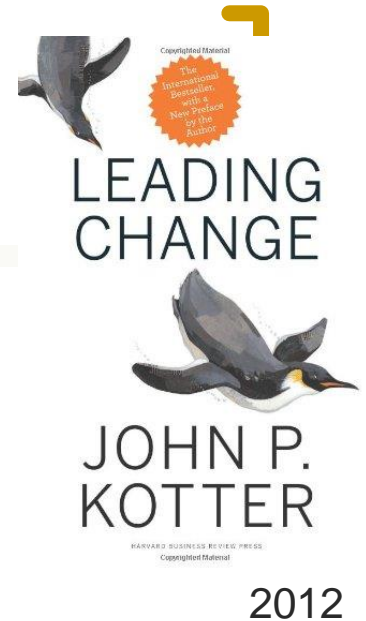
- A medida que un sistema evoluciona su complejidad se incrementa a menos que se trabaje para evitarlo

## ■ Declining Quality

- La calidad de un sistema va a ir declinando a menos que se haga un mantenimiento riguroso

# [Prepararse para el cambio

- Por qué se pierden oportunidades de negocio?
  - El ritmo del cambio en los negocios está creciendo **exponencialmente**
  - Cambio exponencial implica tiempo de reacción exponencialmente menor

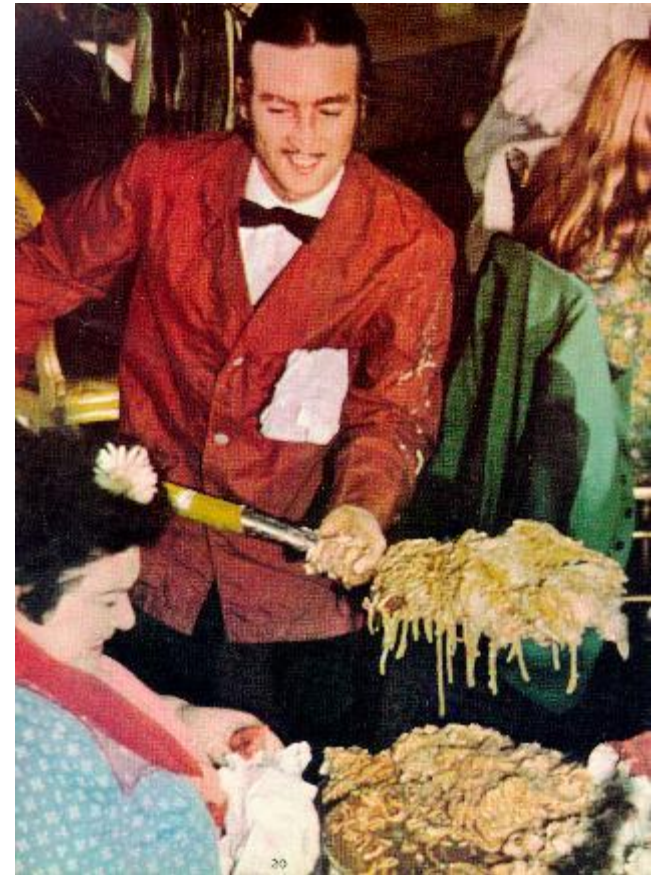


# [ Costo del mantenimiento ]

- 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

# [ 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 igual 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 construyendo 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

## ■ Cuándo

- Una vez que tengo código que funciona y **pasa los tests**
- A medida que voy desarrollando:
  - cuando encuentro código difícil de entender (ugly code)
  - cuando tengo que hacer un cambio y necesito reorganizar primero
- Antes de llegar a



## ■ Testear después de cada cambio

# [Conduciendo]

“Driving is not about getting the car going in the right direction. Driving is about constantly paying attention, making a little correction this way, a little correction that way”.

Kent Beck's mom

“Stay aware. Adapt. Change.”

Kent Beck.



# [ ¿Cómo manejar el cambio? ]

- Un mal diseño no es grave, hasta que hay que hacer cambios
- Tener en cuenta todas las posibles alternativas como para que no haya nada que cambiar. Funciona?
- No podemos prevenir los cambios
- El problema no es el cambio sino nuestra incapacidad de manejarlo



# [ Manejar el cambio es difícil ]

- Participan desarrolladores, quienes se preocupan o son afectados



- No es fácil descubrir donde cambiar
- Es probable que se **introduzcan errores**

# [Cómo ayuda el refactoring?]

- Introduce mecanismos que solucionan problemas de diseño
- 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

# [Importancia del refactoring]

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



# [ Más información ]

- “Big Ball of Mud”. Brian Foote and Joe Yoder. Pattern Languages of Programs 4. Addison-Wesley 2000.
- Big ball of mud @ Google Talks 2007:  
[https://www.youtube.com/watch?v=LH\\_e8NfNV-c&t=75s](https://www.youtube.com/watch?v=LH_e8NfNV-c&t=75s)
- “Refactoring”. Martin Fowler. Addison-Wesley 1999.  
(original con ejemplos en Java)
- “Refactoring. 2nd edition”. Martin Fowler. Addison-Wesley 2018. (ejemplos en JavaScript)
- Martin Fowler @ OOP2014 “Workflows of Refactoring”:  
<https://www.youtube.com/watch?v=vqEg37e4Mkw>