# Refactoring


Alejandra Garrido
Objetos 2
Facultad de
Informática - UNLP

# En la clase pasada…

- 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**

# Nos ponemos ágiles

- http://agilemanifesto.org/
- Dos prácticas ágiles esenciales:
  - Refactoring
  - Test first



# 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

Beautifying code

# Surge el refactoring en la OO

- 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.

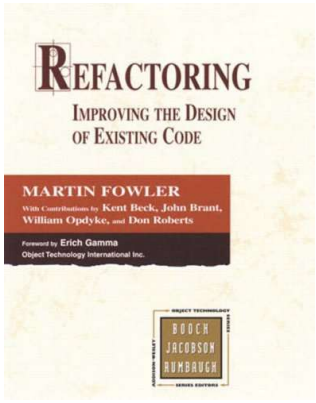




# Create CLEAN Code

- CLEAN:
  - Cohesive,
  - Loosely coupled,
  - Encapsulated,
  - Assertive,
  - Non-redundant.
- Pero además: legible

# Refactoring by Martin Fowler

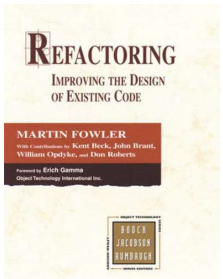




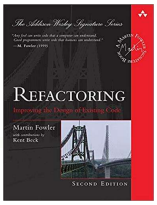



# Refactoring

- Es el proceso a través del cual se cambia un sistema de software mejorando la organización, legibilidad, adaptabilidad y mantenibilidad del código luego que ha sido escrito….
  - que **NO altera** el comportamiento externo del sistema,
  - que ***mejora*** su estructura interna



# Refactoring by Fowler




- *Refactoring* (sustantivo): cada uno de los cambios catalogados
  - "A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior".
  - Con un nombre específico y una receta (mecánica)
- *Refactor* (verbo): el proceso de aplicar refactorings
  - "To restructure software by applying a series of refactorings without changing its observable behavior"

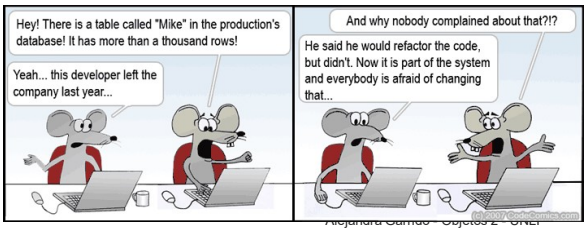



# ¿Por qué refactoring es importante?

- Ganar en la comprensión del código
- Reducir el costo de mantenimiento debido a los cambios inevitables que sufrirá el sistema
  (por ejemplo, código duplicado que haya que cambiar)
- Facilitar la detección de bugs
- La clave: poder agregar funcionalidad más rápido después de refactorizar

## entonces ☞ Refactoring

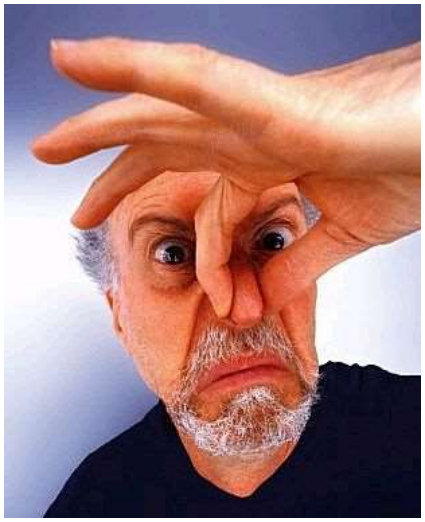
- Se toma un código ***"con mal olor"*** producto de mal diseño
  - Codigo duplicado, no claro, complicado
- y se lo trabaja para obtener un buen diseño
- Cómo?
  - Moviendo un atributo de una clase a otra
  - Extrayendo código de un método en otro método
  - Moviendo código en la jerarquía
  - Etc etc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






## BAD SMELLS!! (in code)

- Indicios de problemas que requieren la aplicación de refactorings



## Algunos bad smells

- Duplicate Code
- Large Class
- Long Method
- Data Class
- Feature Envy
- Long Parameter List
- Switch Statements



## Code smell: Código duplicado

- El mismo código, o código muy similar, aparece en muchos lugares.
- Problemas:
  - Un bug fix en un clone no es fácilmente propagado a los demás clones
  - Hace el código más largo de lo que necesita ser
  - Es difícil de cambiar, dificil de mantener

## Code smell: Método largo

- Un método tiene muchas líneas de código
- Cúanto es muchas LOCs?
  - Más de 20? 30?
  - También depende del lenguaje
- Problemas:
  - Cuanto más largo es un método, más difícil es entenderlo, cambiarlo y reusarlo



## Code smell: Envidia de atributo

- Un método en una clase usa principalmente los datos y métodos de otra clase para realizar su trabajo (se muestra “envidiosa” de las capacidades de otra clase)
- Problema:
  - Indica un problema de diseño
  - Idealmente se prefiere que los datos y las acciones sobre los datos vivan en la misma clase
  - “Feature Envy” indica que el método fue ubicado en la clase incorrecta

## Code smell: Clase grande

- Una clase intenta hacer demasiado trabajo
- Tiene muchas variables de instancia
- Tiene muchos métodos
- Problema:
  - Indica un problema de diseño.
  - Algunos métodos puede pertener a otra clase
  - Generalmente tiene código duplicado
- Ejemplos (Pharo): BlockNode, CharacterScanner, Paragraph



## Code smell: Condicionales

- Cuando sentencias condicionales contienen lógica para diferentes tipos de objetos
- Cuando todos los objetos son instancias de la misma clase, eso indica que se necesitan crear subclases.
- Problema: la misma estructura condicional aparece en muchos lugares

# Code smell: Clase de datos

- Una clase que solo tiene variables y getters/setters para esas variables
- Actúa únicamente como contenedor de datos
- Problemas:
  - En general sucede que otras clases tienen métodos con “envidia de atributo”
  - Esto indica que esos métodos deberían estar en la “data class”
  - Suele indicar que el diseño es procedural



# Veamos un ejemplo….

- Imprimir los puntajes de cada set de un jugador en cada partido de tenis de una fecha específica.

Puntajes para los partidos de la fecha 15/5/2017

Partido:

Puntaje del jugador: Rafael Nadal: 6; 5; 7; Puntos del partido: 36

Puntaje del jugador: Roger Federer: 4; 7; 6; Puntos del partido: 34

Partido:

.....

# class ClubTenis (1)

```
mostrarPuntajesJugadoresEnFecha: aDate
        | partidosDeLaFecha result |
        result := WriteStream on: String new.
        result nextPutAll: 'Puntajes para los partidos de la fecha' , aDate asString; cr.
        partidosDeLaFecha := coleccionPartidos select: [ :p | p fecha = aDate ].
        partidosDeLaFecha
                do: [ :partido |
                        | j1 j2 totalGames |
                        result
                                nextPutAll: 'Partido:';
                                cr.
                        j1 := partido jugador1.
                        totalGames := 0.
                        result
                                nextPutAll: 'Puntaje del jugador: ';
                                nextPutAll: j1 getNombreJugador;
                                nextPutAll: ': '.
```



# class ClubTenis (2)

```
(partido puntosDelJugador: j1)
        do: [ :gamesDelSet |
                result
                        nextPutAll: gamesDelSet asString, ';'.
                totalGames := totalGames + gamesDelSet ].
result nextPutAll: ' Puntos del partido: '.
j1 zona = 'A'
        ifTrue: [ result nextPutAll: (totalGames * 2) asString ].
j1 zona = 'B'
        ifTrue: [ result nextPutAll: totalGames asString ].
j1 zona = 'C'
        ifTrue: [ partido ganador = j1
                        ifTrue: [ result nextPutAll: totalGames asString ]
                        ifFalse: [ result nextPutAll: 0 asString ] ].
```

# class ClubTenis (3)

```
j2 := partido jugador2.
totalGames := 0.
result
        nextPutAll: 'Puntaje del jugador: ';
        nextPutAll: j2 getNombreJugador;
        cr.
(partido puntosDelJugador: j2)
        do: [ :gamesDelSet |
                result
                        nextPutAll: gamesDelSet asString, ';'.
                totalGames := totalGames + gamesDelSet ].
result nextPutAll: '. Puntos del partido: '.
j2 zona = 'A'
        ifTrue: [ result nextPutAll: (totalGames * 2) asString ].
j2 zona = 'B'
        ifTrue: [ result nextPutAll: totalGames asString ].
j2 zona = 'C'
        ifTrue: [ partido ganador = j2
                        ifTrue: [ result nextPutAll: totalGames asString ]
                        ifFalse: [ result nextPutAll: 0 asString ] ] ].
^result contents
```



# Cambios pedidos …

- Cambiará la manera de calcular los puntos
- Pueden cambiar las zonas

## Ejemplo del club de tenis

- Cuáles son los malos olores?
- Por dónde empezamos?



## Extract Method

- Motivación :
  - Métodos largos
  - Métodos muy comentados
  - Incrementar reuso
  - Incrementar legibilidad

# Extract Method

- Mecánica:
  - Crear un nuevo método cuyo nombre explique su propósito
  - Copiar el código a extraer al nuevo método
  - Revisar las variables locales del original
  - Si alguna se usa sólo en el código extraído, mover su declaración
  - Revisar si alguna variable local es modificada por el código extraido. Si es solo una, tratar como query y asignar. Si hay más de una no se puede extraer.
  - Pasar como parámetro las variables que el método nuevo lee.
  - Compilar
  - Reemplazar código en método original por llamada
  - Compilar



# Extract method: Extrayendo

```
mostrarPuntajesJugadoresEnFecha: aDate
        | partidosDeLaFecha result |
        result := WriteStream on: String new.
        result nextPutAll: 'Puntajes para los partidos de la fecha' , aDate asString; cr.
        partidosDeLaFecha := coleccionPartidos select: [ :p | p fecha = aDate ].
        partidosDeLaFecha
                do: [ :partido |
                        self mostrarPartido: partido en: result].
        ^result contents
```

## A tener en cuenta…

- Testear siempre después de hacer un cambio
  - Sí se cometió un error es más fácil corregirlo
- Definir buenos nombres



## Vale la pena?

- Todo buen código debería comunicar con claridad lo que hace
- Nombres de variables adecuados aumentan la claridad
- Sólo los buenos programadores escriben código legible por otras personas

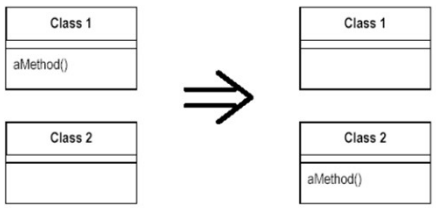
# En la clase ClubTenis…

- A quien pertenece realmente el codigo de
- **mostrarPartido: partido en: result ?**



# Move Method

- Motivación:
  - Un método esta usando o usará muchos servicios que están definidos en una clase diferente a la suya (Feature envy)
- Solucion:
  - Mover el método a la clase donde están los servicios que usa.
  - Convertir el método original en un simple delegación o eliminarlo



## Move Method: Mecánica

- Revisar otros atributos y métodos en la clase original (puede que también haya que moverlos)
- Chequear subclases y superclases de la clase original por si hay otras declaraciones del método (puede que no se pueda mover)
- Declarar el método en la clase destino
- Copiar y ajustar el código (ajustando las referencias desde el objeto origen al destino); chequear manejo de excepciones
- Convertir el método original en una delegación
- Compilar y testear
- Decidir si eliminar el método original → eliminar las referencias
- Compilar y testear



## En la clase Partido…

El código de

- **mostrarEn: result**

sigue siendo bastante largo, porque tiene código duplicado

→ más Extract Method!

## Seguimos teniendo el switch

- ¿Cómo eliminar el switch?
- ➔ Replace Conditional with Polymorphism

- ¿Tiene sentido hacer subclases de Partido? ¿Corresponde a Partido este cálculo? No, de Jugador



## Replace Conditional with Polymorphism

- Crear la jerarquía.
- Por cada variante, crear un método en cada subclase que redefina el de la superclase.
- Copiar al método de cada subclase la parte del condicional correspondiente.
- Compilar y testear.
- Borrar de la superclase la sección (branch) del condicional que se copió.
- Compilar y testear.
- Repetir para todos los branchs del condicional.
- Hacer que el método de la superclase sea abstracto.

## En JugadorZonaA…

```
>>mostrarPuntosDePartido: unPartido en: unStream
        | totalGames |
        self mostrarNombreEn: unStream.
        totalGames := 0.
        (unPartido puntosDelJugador: self)
                do: [ :gamesDelSet |
                        unStream nextPutAll: gamesDelSet asString, ';'.
                        totalGames := totalGames + gamesDelSet ].
        unSream nextPutAll: ' Puntos del partido: '.
        unStream nextPutAll: (totalGames * 2) asString.

>>mostrarNombreEn: unStream
        unStream
                nextPutAll: 'Puntaje del jugador: ';
                nextPutAll: self nombre;
                nextPutAll: ': '.
```



## Método repetido en c/ subclase

- `>>mostrarNombreEn: unStream`
- Cómo eliminamos esta duplicación?
- → Pull Up Method
  - Si los métodos en subclases son iguales → subir directamente
  - Si los métodos en subclases no son iguales → parametrizar primero

# Pull Up Method

1. Asegurarse que los métodos sean idénticos. Si no, parametrizar
2. Si el selector del método es diferente en cada subclase, renombrar
3. Si el método llama a otro que no está en la superclase, declararlo como abstracto en la superclase
4. Si el método llama a un atributo declarado en las subclases, usar "*Pull Up Field*" o "*Self Encapsulate Field*" y declarar los getters abstractos en la superclase
5. Crear un nuevo método en la superclase, copiar el cuerpo de uno de los métodos a él, ajustar, compilar
6. Borrar el método de una de las subclases
7. Compilar y testear
8. Repetir desde 6 hasta que no quede en ninguna subclase



# Redundancia de variables temporales

```
>>mostrarPuntosDePartido: unPartido en: unStream
        | totalGames |
        self mostrarNombreEn: unStream.
        totalGames := 0.
        (unPartido puntosDelJugador: self)
                do: [ :gamesDelSet |
                        unStream nextPutAll: gamesDelSet asString, ';'.
                        totalGames := totalGames + gamesDelSet ].
        unSream nextPutAll: ' Puntos del partido: '.
        unStream nextPutAll: (totalGames * 2) asString.
```

## Replace Temp with Query

- Motivación: usar este refactoring:
  - Para evitar métodos largos. Las temporales, al ser locales, fomentan métodos largos
  - Para poder usar una expresión desde otros métodos
  - Antes de un Extract Method, para evitar parámetros innecesarios
- Solución:
  - Extraer la expresión en un método
  - Remplazar TODAS las referencias a la var. temporal por la expresión
  - El nuevo método luego puede ser usado en otros métodos



## Replace Temp With Query

- Mecánica:
  - Encontrar las vars. temporales con una sola asignación (si no, Split Temporary Variable)
  - Extraer el lado derecho de la asignación (tener cuidado con los efectos colaterales; si no, Separate Query From Modifier)
  - Remplazar todas las referencia de la var. temporal por el nuevo método
  - Eliminar la declaración de la var. temporal y las asignaciones
  - Compilar y testear

# Después de Replace Temp with Query

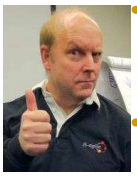
```
>>mostrarPuntosDePartido: unPartido en: unStream
        self mostrarNombreEn: unStream.
        (unPartido puntosDelJugador: self)
                do: [ :gamesDelSet |
                        unStream nextPutAll: gamesDelSet asString, ';'].
        unSream nextPutAll: ' Puntos del partido: '.
        unStream nextPutAll: (self totalGamesEn: unPartido * 2) asString.
```



# Sobre la performance

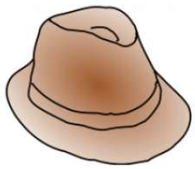
- La mejor manera de optimizar un programa, primero es escribir un programa bien factorizado y luego optimizarlo, previo profiling …

# The 2 hats



## Adding Function

Se exploran ideas, se corrigen bugs



## Refactoring

Solo puedo refactorizar con tests en verde

Puedo cambiar de sombrero frecuentemente
Pero solo puedo usar 1 sombrero por vez



# Malos olores

- Código duplicado
  - Extract Method
  - Pull Up Method
  - Form Template Method
- Métodos largos
  - Extract Method
  - Decompose Conditional
  - Replace Temp with Query
- Clases grandes
  - Extract Class
  - Extract Subclass
- Muchos parámetros
  - Replace Parameter with Method
  - Preserve Whole Object
  - Introduce Parameter Object

# Malos olores (2)

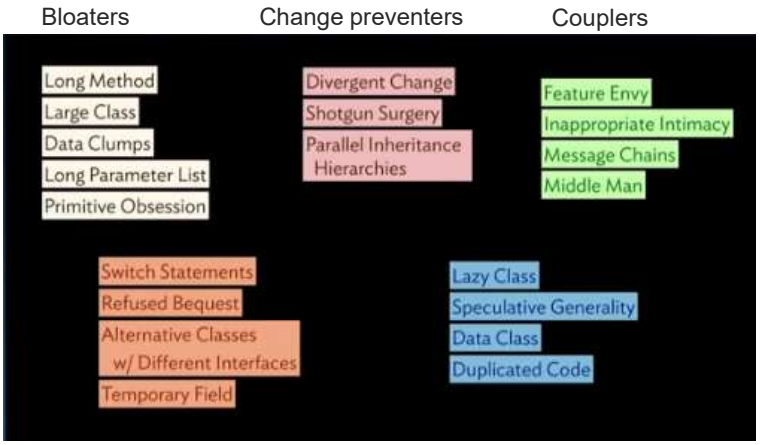
- Cambios divergentes
  - Extract Class
- “Shotgun surgery”
  - Move Method/Field
- Envidia de atributo (Feature Envy)
  - Move Method
- Sentencias Switch
  - Replace Conditional with Polymorphism
- Lazy class
  - Inline Class
- Generalidad especulativa
  - Collapse Hierarchy
  - Inline Class
- Cadena de mensajes
  - Hide Delegate
  - Extract Method



# Malos olores (3)

- Middle man
  - Remove Middle man
- Inappropriate Intimacy
  - Move Method/Field
- Data Class
  - Move Method
- Legado rechazado
  - Push Down Method/Field
- Comentarios
  - Extract Method
  - Rename Method

# Categorización de bad smells






# Catálogo de Fowler

- Refactoring manual
- Formato:
  - Nombre
  - Motivación
  - Mecánica
  - Ejemplo
- Por qué necesitamos aprenderlo?

# Organización catálogo Fowler

- Composición de métodos
- Mover aspectos entre objetos
- Organización de datos
- Simplificación de expresiones condicionales
- Simplificación en la invocación de métodos
- Manipulación de la generalización
- Big refactorings



# Composición de métodos

- Permiten “distribuir” el código adecuadamente.
- Métodos largos son problemáticos
- Contienen:
  - mucha información
  - lógica compleja

- Extract Method
- Inline Method
- Replace Temp with Query
- Split Temporary Variable
- Replace Method with Method Object
- Substitute Algorithm

Nota: los subrayados fueron vistos en clase

## Mover aspectos entre objetos

- Ayudan a mejorar la asignación de responsabilidades

- Move Method
- Move Field
- Extract class
- Inline Class
- Remove Middle Man
- Hide Delegate



## Manipulación de la generalización

- Ayudan a mejorar las jerarquías de clases

- Push Up / Down Field
- Push Up / Down Method
- Pull Up Constructor Body
- Extract Subclass / Superclass
- Collapse Hierarchy
- Replace Inheritance with Delegation
- Replace Delegation with Inheritance

# Organización de datos

- Facilitan la organización de atributos

- Self Encapsulate Field
- Encapsulate Field / Collection
- Replace Data Value with Object
- Replace Array with Object
- Replace Magic Number with Symbolic Constant



# Simplificación de expresiones condicionales

- Ayudan a simplificar los condicionales

- Decompose Conditional
- Consolidate Conditional Expression
- Consolidate Duplicate Conditional Fragments
- Replace Conditional with Polimorfism

Nota: Decompose Conditional en StandardRoom https://www.refactoring.com/

# Simplificación de invocación de métodos

- Sirven para mejorar la interfaz de una clase

- Rename Method
- Preserve Whole Object
- Introduce Parameter Object
- Parameterize Method

Nota: “Preserve Whole Object” es recomendable para pasar el contexto a un Strategy o State



# Cuando aplicar refactoring

- En el contexto de TDD: red-green-refactor
- Cuando se descubre código con mal olor, aprovechando la oportunidad
  - dejarlo al menos un poco mejor, dependiendo del tiempo que lleve y de lo que esté haciendo
- Cuando no puedo entender el código
  - aprovechar el momento en que lo logro entender
- Cuando encuentro una mejor manera de codificar algo

# 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




# El refactoring automático nace con Smalltalk

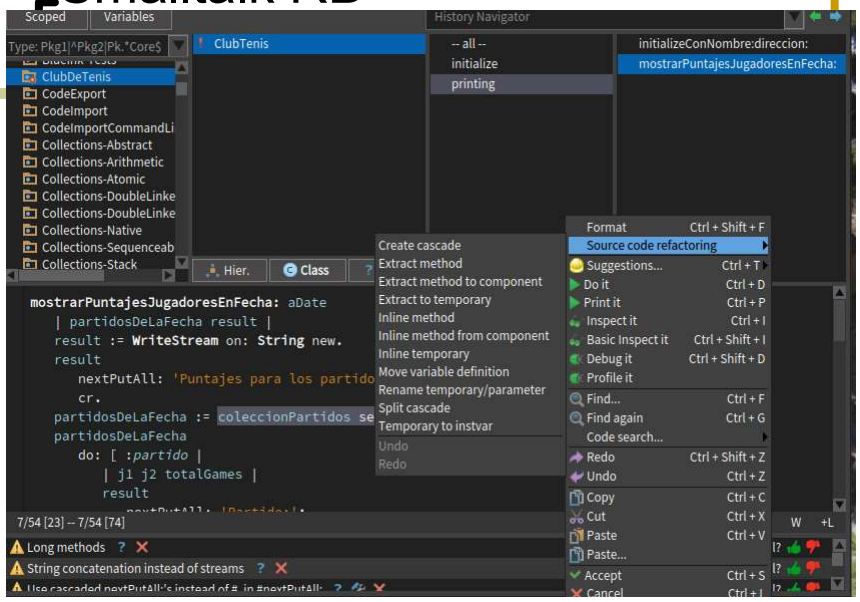
- Primera herramienta de refactoring: Refactoring Browser (RB) (en UIUC by John Brant & Don Roberts del grupo de Ralph Johnson)
- Practicamente todos los lenguajes tienen herramienta de refactoring hoy en día, y copian la misma arquitectura / técnica del RB
- Más adelante: herramienta Code Critic que detecta code smells
- Smalltalk 1ro en:
  - XUnit
  - Refactoring

# Las herramientas…

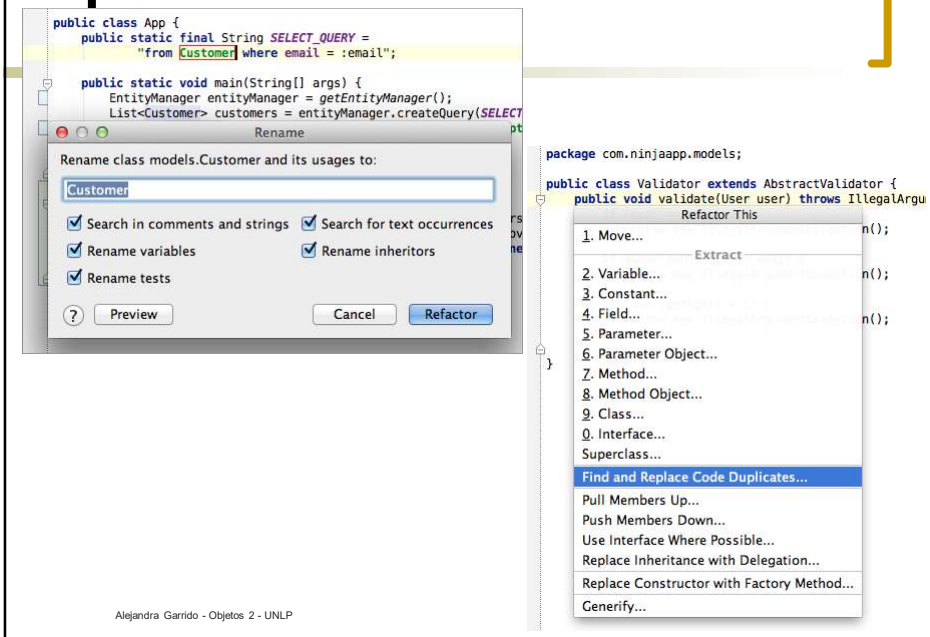
- Solo chequean lo que sea posible desde el árbol de sintaxis y la tabla de símbolos
- Pueden ser demasiado **conservativas** (no realizan un refactoring si no pueden asegurar preservación de comportamiento) o **asumir** buenas técnicas de programación



# Smalltalk RB

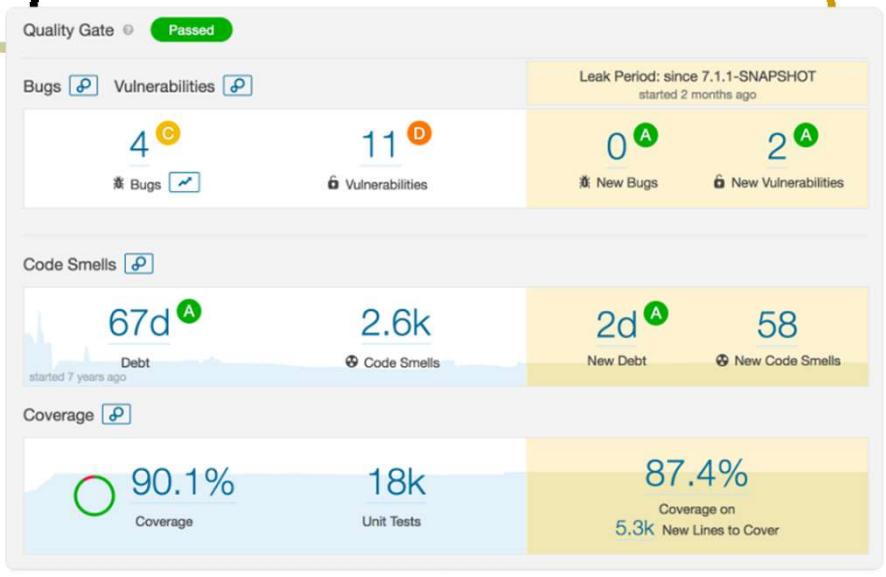


# IntelliJ IDEA






# Sonar qube



# Referencias

- “Refactoring. Improving the Design of Existing Code”. Martin Fowler. Addison Wesley. 1999.
- Refactoring. https://blog.bryanbibat.net/2009/08/25/refactoring/
- “Technical Excellence”. David Bernstein. https://www.linkedin.com/pulse/technical-excellence-david-bernstein



# Videos interesantes

- Code refactoring: https://www.youtube.com/watch?v=vhYK3pDUijk
- Martin Fowler @ OOP2014 "Workflows of Refactoring“: https://www.youtube.com/watch?v=vqEg37e4Mkw
- Code Refactoring: Learn Code Smells And Level Up Your Game!: https://www.youtube.com/watch?v=D4auWwMsEnY