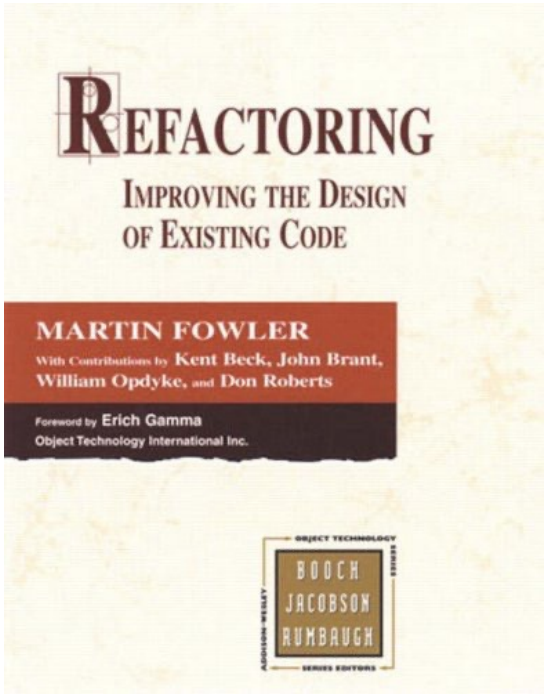
Refactoring. Catálogo



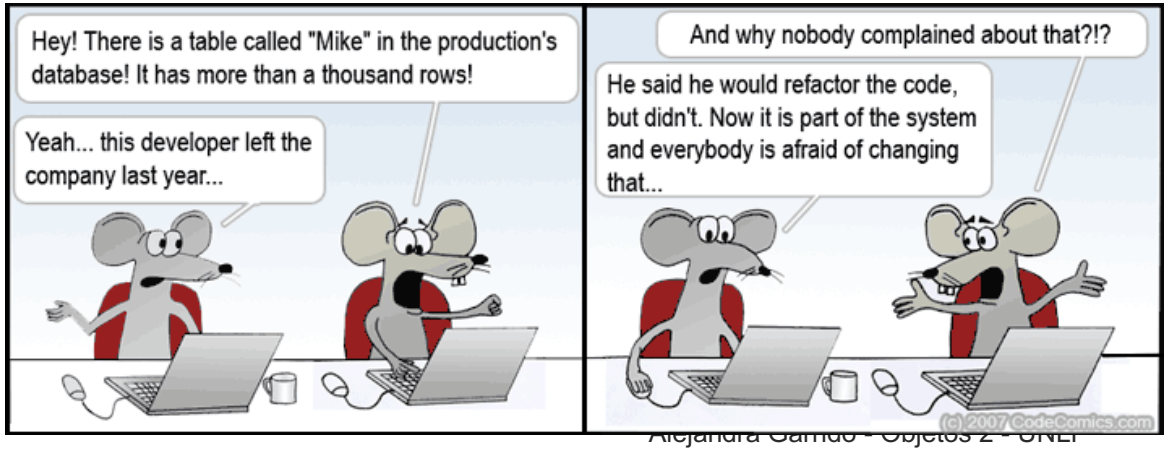
Alejandra Garrido
Objetos 2
Facultad de
Informática - UNLP

Importancia del refactoring

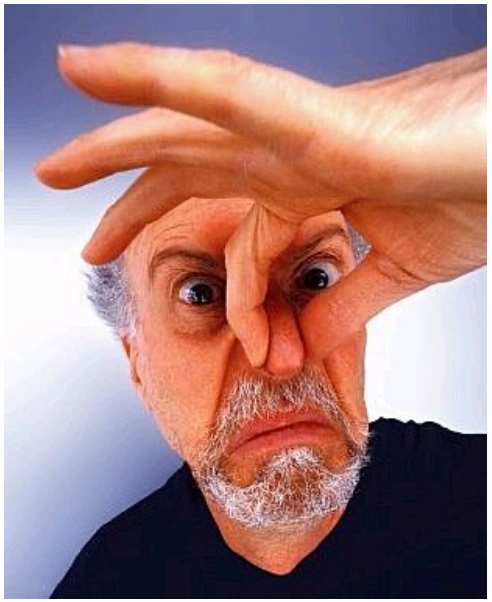
- Nuestra única defensa contra el deterioro del software.
- Facilita la incorporación de código
- Permite preocuparse por la generalidad mañana.
- Es decir, permite ser ágil en el desarrollo

Proceso de refactoring

- Se toma un código que ***"huele mal"*** producto de mal diseño
  - Codigo duplicado, ilegible, complicado
- y se lo trabaja para obtener un buen diseño
- Cómo?
  - Moviendo atributos / métodos de una clase a otra
  - Extrayendo código de un método en otro método
  - Moviendo código en la jerarquía
  - Etc etc etc …



BAD SMELLS!



- Indicios de problemas que requieren la aplicación de refactorings

- Posteriormente llamados CODE SMELLS

Algunos code 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:
  - Hace el código más largo de lo que necesita ser
  - Es difícil de cambiar, dificil de mantener
  - Un bug fix en un clone no es fácilmente propagado a los demás clones

Code smell: Clase grande

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

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 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

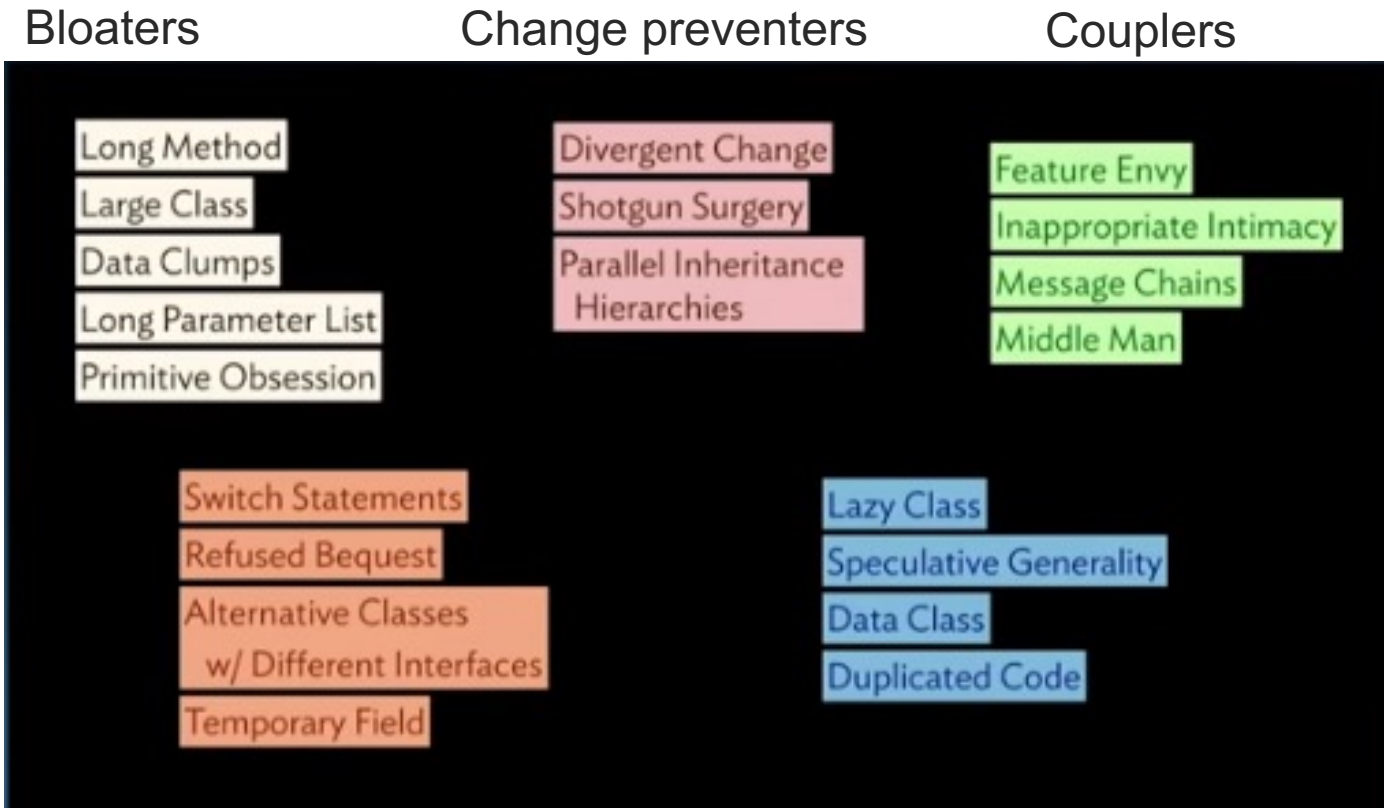
Code smell: Condicionales

- 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: Long Parameter List

- Un método con una larga lista de parámetros es más dificil de entender
- También es difícil obtener todos los parámetros para pasarlos en la llamada entonces el método es más difícil de reusar
- La excepción es cuando no quiero crear una dependencia entre el objetos llamador y el llamado

Categorización de CODE SMELLS



https://www.youtube.com/watch?v=D4auWwMsEnY

Catálogo de refactorings

- Refactoring manual: viene con recetas!
- 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

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

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/

Decompose Conditional

Se tiene una expresión condicional compleja

Extraer métodos de la condición, la parte “then” y la parte “else”

```
if (date.before(SUMMER_START) || date.after(SUMMER_END))
        charge = quantity * winterRate + winterServiceCharge;
else
        charge = quantity * summerRate;
```

```
if (notSummer(date))
        charge = winterCharge();
else
        charge = summerCharge();
```

Simplificación de invocación de métodos

- Sirven para mejorar la interfaz de una clase

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

Code smells –Catálogo Fowler (1)

- 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

Code smells –Catálogo Fowler (2)

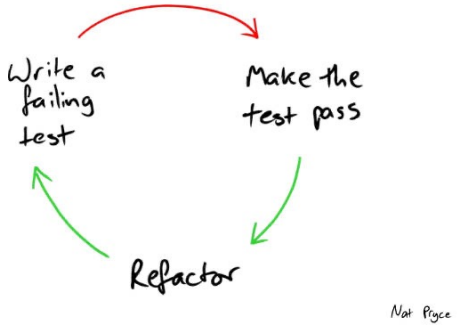
- Cambios divergentes (Divergent Change)
  - Extract Class
- “Shotgun surgery”
  - Move Method/Field
- Envidia de atributo (Feature Envy)
  - Move Method
- Data Class
  - Move Method
- Sentencias Switch
  - Replace Conditional with Polymorphism
- Generalidad especulativa
  - Collapse Hierarchy
  - Inline Class
  - Remove Parameter

Code smells –Catálogo Fowler (3)

- Cadena de mensajes

  (banco cuentaNro: unNro) movimientos first fecha
  - Hide Delegate
  - Extract Method & Move Method
- Middle man
  - Remove Middle man
- Inappropriate Intimacy
  - Move Method/Field
- Legado rechazado (Refused bequest)
  - Push Down Method/Field
- Comentarios
  - Extract Method
  - Rename Method

Cuando aplicar refactoring

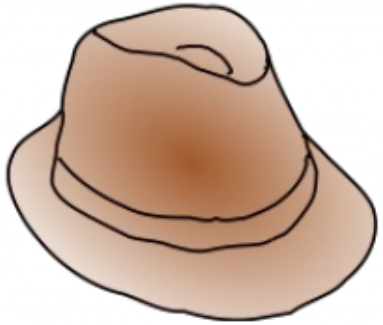


- En el contexto de TDD
- Cuando se descubre código con mal olor, aprovechando la oportunidad
  - dejarlo al menos un poco mejor, dependiendo del tiempo que lleve y de lo que esté haciendo
- Cuando no puedo entender el código
  - aprovechar el momento en que lo logro entender
- Cuando encuentro una mejor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

The 2 hats

Adding Function

Se exploran ideas, se corrigen bugs



Refactoring

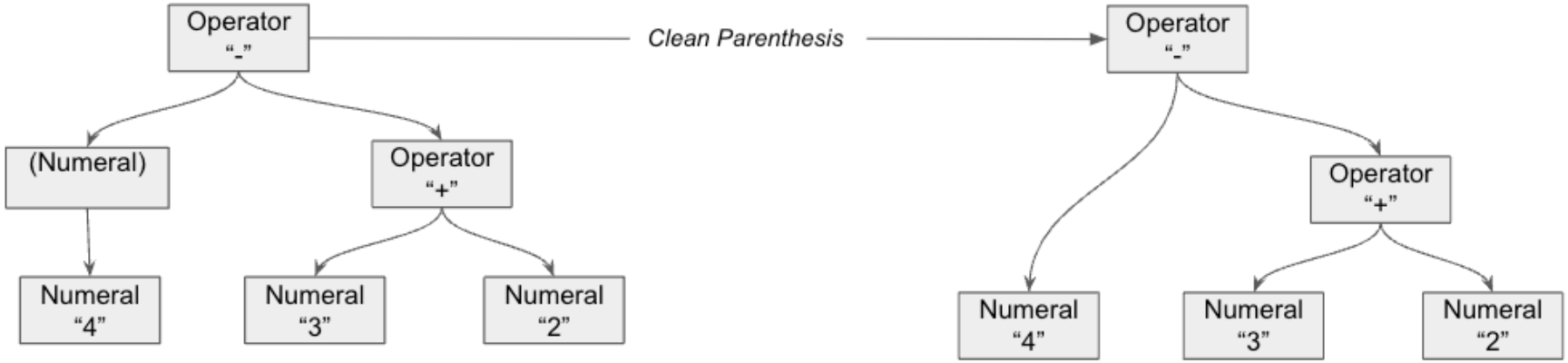
Solo puedo refactorizar con tests en verde

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

Automatización del refactoring

- Refactorizar a mano es demasiado costoso: lleva tiempo y puede introducir errores
- 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
  - AST+symbol table → AST rewriting

Ejemplo de AST rewriting



Pretty print ((4) - (3 + 2))

Pretty print (4 - (3 + 2))

¿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

Referencias y videos

- “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
- “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

Más referencias y videos

- Catálogo de refactoring: https://refactoring.com
- Categorías de code smells: https://sourcemaking.com/refactoring/smells
- Code Refactoring: Learn Code Smells And Level Up Your Game!: https://www.youtube.com/watch?v=D4auWwMsEnY
- “Object Oriented Metrics in Practice”. Lanza & Marinescu. Springer 2006