



Pontificia Universidad Católica de Chile  
Escuela de Ingeniería  
Departamento de Ciencia de la Computación

# **Clase 6**

# **Code Smells + Refactoring**

## **IIC2113 – Diseño Detallado de Software**

Rodrigo Saffie

rasaffie@uc.cl

12 de octubre de 2018

# Notas I1

- 65 alumnos:
  - Promedio: 4.73
  - Mínimo: 2.70
  - Máximo: 6.30
- 9 rojos (13.8%)
  - Principalmente por patrones de diseño

# Tarea 2

- Objetivos:
  - Detectar y aplicar patrones de diseño para resolver problemas computacionales.
  - Aplicar sintaxis **UML 2.0** para definir la estructura de código de un sistema de *software*.

# Actividad 5

- Patrones de diseño
  - *Chain of Responsibility*
  - *Interpreter*
  - *Mediator*
  - *State*
  - *Strategy*
  - *Template Method*
  - *Visitor*

# Patrones de diseño

- Revisamos 23 patrones de OOP agrupados en 3 categorías
- También existe el concepto de [anti-patrones](#):
  - *Big ball of mud*
  - *God object*
  - *Spaghetti code*
  - *Magic numbers/strings*

# Actividad Opcional

- Respondieron 26 alumnos (37.6%)
  - ¿Cuál es la diferencia entre una librería y un *framework*?
  - ¿En qué se diferencia un *framework* a un patrón de diseño?
  - Angular vs ReactJS

# ¿Qué es un *Code Smell*?

- “A code smell is a surface indication that usually corresponds to a deeper problem in the system.” [[Martin Fowler](#), 2006]
- “Smells are certain structures in the code that indicate violation of fundamental design principles and negatively impact design quality.” [Girish Suryanarayana]
- El término fue acuñado por [Kent Beck](#) y se hizo famoso con el libro [Refactoring: Improving the Design of Existing Code](#) de Martin Fowler.

# *Code Smell*: Características

- **No son bugs:**

- el programa funciona correctamente, pero su débil diseño dificulta el desarrollo e incrementa la posibilidad de generar *bugs*.
- Es un indicador superficial rápido de detectar
- No siempre indican un problema en el código
- Se pueden generar por un mal diseño del programa, pero también por presiones en cumplir plazos al desarrollar



# *Code Smell*: Clasificaciones

Existen 5 clasificaciones:

- *Bloaters*
- *Object-Orientation Abusers*
- *Change Preventers*
- *Dispensables*
- *Couplers*

# ¿Qué es *refactoring*?

- Es un proceso sistemático para mejorar código sin modificar su funcionalidad.
- El objetivo es reducir la deuda técnica, para así mejorar la mantenibilidad y facilitar la extensibilidad.
- Normalmente la presencia de un *code smell* motiva su uso para generar *clean code*.

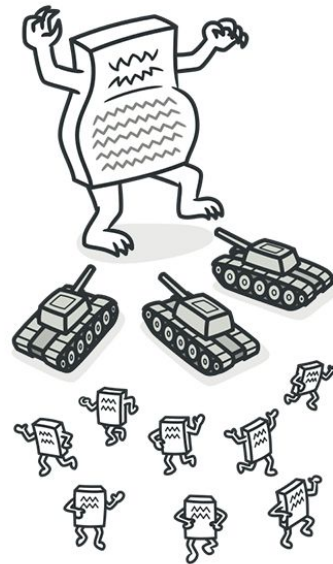
# Referencias

- [\*Refactoring Guru\*](#)
- [\*reek\*](#)

# Code Smells

## ***Bloaters***

- Representan código, métodos y clases que han crecido a tal punto que es difícil trabajar con ellos.



# *Bloaters*

## *Long Method*

### **Síntomas**

- Un método contiene excesivas líneas de código (más de 10 puede ser un problema)

### **Razones del problema**

- La misma persona que escribe el código lo mantiene
- “Es más fácil” agregar líneas a un método existente que crear nuevos

### **Beneficios de solucionarlo**

- Código más fácil de entender y mantener
- Más fácil detectar código duplicado

# *Bloaters*

## *Large Class*

### **Síntomas**

- Una clase contiene excesivas líneas de código, atributos y métodos

### **Razones del problema**

- Las clases crecen en conjunto con la aplicación con baja cohesión
- “Es más fácil” agregar líneas a un método existente que crear nuevos

### **Beneficios de solucionarlo**

- Código más fácil de entender y mantener
- Más fácil detectar código duplicado

# *Bloaters*

## *Primitive Obsession*

### **Síntomas**

- Uso de variables primitivas en vez de objeto pequeños
- Uso de constantes para referenciar índices en arreglos

### **Razones del problema**

- Se agregan atributos a una clase a medida que se necesitan, sin considerar que se pueden agrupar en un objeto
- Mala elección de tipos primitivos para representar una estructura de datos

### **Beneficios de solucionarlo**

- Código más fácil de entender, organizar y extender
- Más fácil detectar código duplicado

# *Bloaters*

## *Long Parameter List*

### **Síntomas**

- Más de 3 o 4 parámetros por método. Es difícil entender listas extensas de parámetros

### **Razones del problema**

- Se agrupa funcionalidad en un solo método
- Se pasan parámetros a un método directamente, y no a través de objetos que contienen la información o llamar a otros métodos para obtenerlos

### **Beneficios de solucionarlo**

- Código más fácil de entender, organizar y extender
- Más fácil detectar código duplicado



# *Bloaters*

## *Data Clumps*

### **Síntomas**

- Diferentes partes de una aplicación contienen una definición recurrente de variables

### **Razones del problema**

- Una pobre estructura de la aplicación, o *copy-paste programming*
- No se agrupan valores inseparables en un objeto

### **Beneficios de solucionarlo**

- Código más fácil de entender y organizar
- Menos código

# Code Smells

## ***Object-Orientation Abusers***

- Estos *code smells* se generan por una incompleta o incorrecta utilización de los principios OOP.



# *Object-Orientation Abusers*

## *Switch Statements*

### **Síntomas**

- Se tiene un operador *switch* complejo, o una gran secuencia de *if*

### **Razones del problema**

- Mal uso de polimorfismo

### **Beneficios de solucionarlo**

- Código más organizado

# *Object-Orientation Abusers*

## *Temporary Field*

### **Síntomas**

- Se tienen atributos de una clase que se les asigna un valor bajo ciertas circunstancias. El resto del tiempo se encuentran vacíos.

### **Razones del problema**

- En vez de pasar datos como parámetros, se crean atributos para las clases

### **Beneficios de solucionarlo**

- Código más claro y organizado

# *Object-Orientation Abusers*

## *Refused Bequest*

### **Síntomas**

- Una subclase utiliza solamente algunas de las propiedades y métodos heredados de sus padres.

### **Razones del problema**

- Se quiso reutilizar código entre una super-clase y una clase, pero son completamente distintas

### **Beneficios de solucionarlo**

- Código más claro y organizado

# *Object-Orientation Abusers*

## *Alternative Classes with Different Interfaces*

### **Síntomas**

- Dos clases realizan funcionalidades idénticas, pero con nombres distintos (clases y métodos)

### **Razones del problema**

- El programador que creo una clase probablemente no sabía que otra clase que ya existía tenía funcionalidad equivalente

### **Beneficios de solucionarlo**

- Elimina código duplicado
- Código más claro y organizado

# Actividad 6



Pontificia Universidad Católica de Chile  
Escuela de Ingeniería  
Departamento de Ciencia de la Computación

# **Clase 6**

# **Code Smells + Refactoring**

## **IIC2113 – Diseño Detallado de Software**

Rodrigo Saffie

rasaffie@uc.cl

12 de octubre de 2018