IIC2113 - Diseño Detallado de Software

CLASE 3 - BUENAS PRÁCTICAS DE DESARROLLO

Pontificia Universidad Católica de Chile
2020-2

Índice

Retomando

Empezamos viendo lo que significa diseñar software. Exploramos lo que era diseñar a priori con los diagramas UML. Luego comenzamos a ver lo que es un buen diseño con los principios fundamentales, SOLID y vimos cómo medirlo.

Exploramos uno de los *code smell* más comunes que son los bloaters.

04. Code Smells

Object-Orientation Abusers

Implementación incompleta o incorrecta de los principios de programación orientados a objetos.

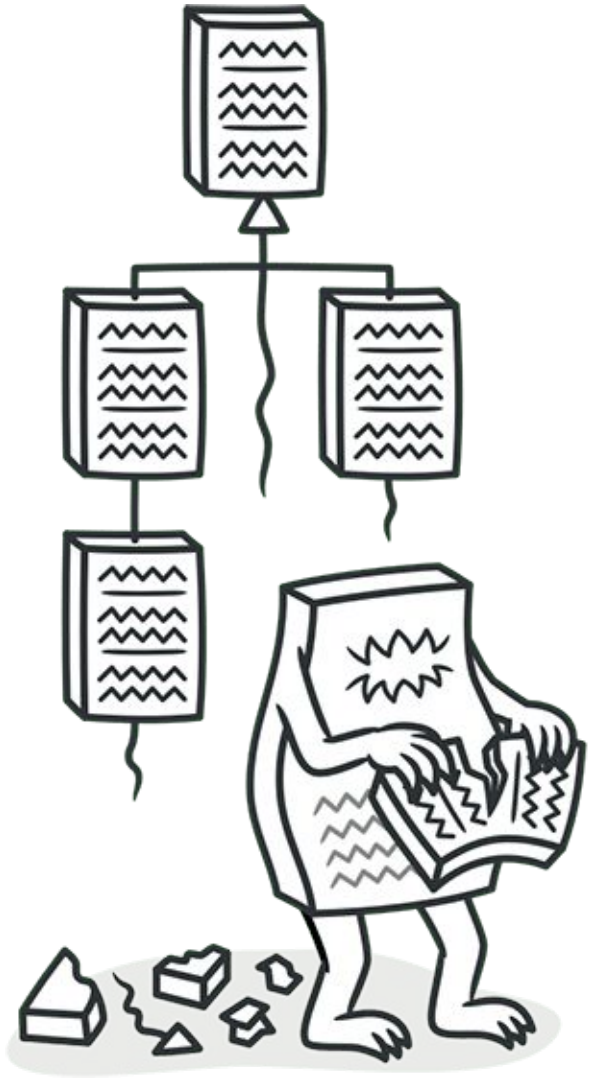


Ilustración extraída de https://refactoring.guru/

Object-Orientation Abusers - Switch Statements

- **Síntomas**: Se tiene un operador switch complejo, o una secuencia de if-else.
- **Razones del problema**:
  - Mal uso de polimorfismo.
  - Poca organización del código.
- **Soluciones**:
  - Aislar en un solo switch/if-else en la clase correcta.
  - Si el switch/if-else depende de un “tipo”, reemplazarlo por una clase o objeto más complejo.
  - Usar polimorfismo, pero solo en caso de que aplique.
  - Si los switch/if-else tienen una acción asociada, usar directamente la acción.

Object-Orientation Abusers - Temporary Field

- **Síntomas**: Se tienen atributos de una clase que solo tienen valor bajo ciertas circunstancias, estando el resto del tiempo sin definir o nulos.
- **Razones del problema**:
  - Se usan atributos de clase en vez de pasar los datos como parámetros.
- **Soluciones**:
  - Quitar los atributos temporales en una clase relacionada (que si los use).

Object-Orientation Abusers - Refused Bequest

- **Síntomas**: Una subclase utiliza algunas propiedades y métodos de sus padres.
- **Razones del problema**:
  - Se quiso reutilizar código entre una superclase y una clase, pero son completamente distintas.
- **Soluciones**:
  - Reemplazar la herencia por delegación o asociación.
  - Extraer los métodos comunes del padre y el hijo, para pasarlo a otra clase de la cual ambas clases hereden.

Object-Orientation Abusers - Alternative Classes with Different Interfaces

- **Síntomas**: Dos clases realizan funcionalidades idénticas, pero con nombres de métodos distintos.
- **Razones del problema**:
  - Desconocimiento por parte del programador sobre la existencia de la otra clase.
- **Soluciones**:
  - Cambiar los nombres de los métodos, ajustar los parámetros y variables para poder reusar la misma clase y borrar el clon.
  - Si solo una parte está duplicada, crear una superclase para hacer herencia.

Change Preventers

Reflejan un posible problema si un cambio en una parte del código implica varios cambios en otras secciones, aumentando el costo de desarrollar.

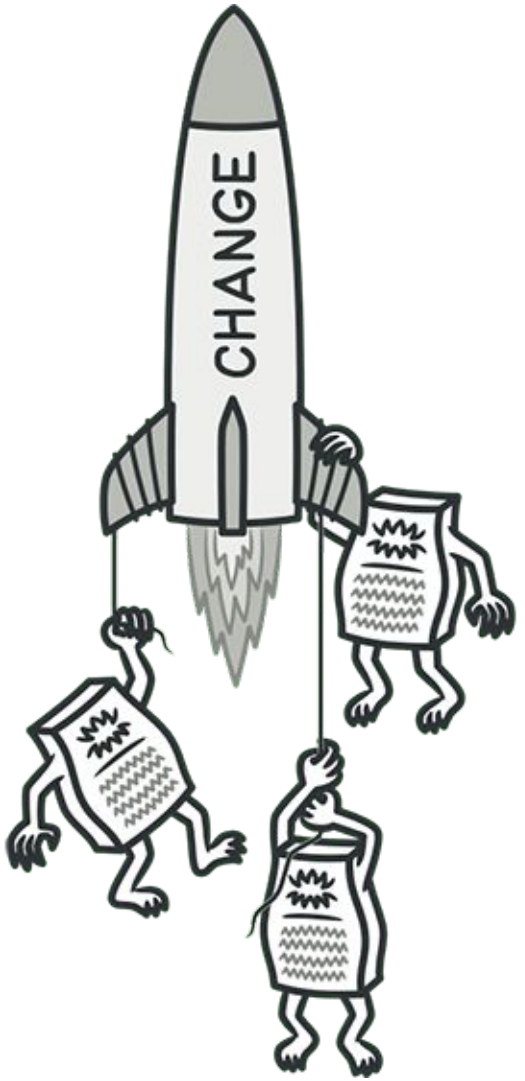


Ilustración extraída de https://refactoring.guru/

Change Preventers - Divergent Change

- **Síntomas**: Cambios en la clase desencadenan cambios en métodos que no tienen relación entre sí.
- **Razones del problema**:
  - Estructura poco pensada.
  - *Copypasta programming*
- **Soluciones**:
  - Extraer el comportamiento de las clases si es algo estándar.
  - Combinar clases a través de herencia.
  - Muchas veces se solucionan de forma simple re-ordenando el código dentro de la misma clase.

Change Preventers - Shotgun Surgery

- **Síntomas**: Cambios en la clase desencadenan cambios en otras clases.
- **Razones del problema**:
  - Responsabilidad fue dividida entre varias clases.
- **Soluciones**:
  - Agrupar la responsabilidad en una sola clase: una ya existente o crear una nueva.
  - Quitar las clases redundantes.

Change Preventers - Parallel Inheritance Hierarchies

- **Síntomas**: Hay exceso de dependencia en el comportamiento: Añadir una **subclase A** a una **clase B**, implica añadir una **subclase C** a una **clase D**.
- **Razones del problema**:
  - Cuando las jerarquías son pequeñas, las modificaciones necesarias se tienen bajo control. Pero a medida que el árbol de jerarquías crece, se hace cada vez más difícil realizar cambios.
- **Soluciones**:
  - Combinar las clases en la jerarquía de ser posible.

Dispensables

Fragmentos de código que son innecesarios e inútiles, además su ausencia haría que el código fuese más claro, eficiente y fácil de entender.

Ilustración extraída de https://refactoring.guru/

Dispensables - Comments

- **Síntomas**: Un método contiene excesivos comentarios.
- **Razones del problema**:
  - La intención de los comentarios es buena. Sin embargo más que un comentario explicando la estructura, la verdadera solución es que la estructura sea buena y entendible por sí misma.
- **Soluciones**:
  - El mejor comentario es un buen nombre de clase o método.
  - Re-estructurar el código para que sea entendible por si mismo.

Dispensables - Código duplicado

- **Síntomas**: Fragmentos identicos de código.
- **Razones del problema**:
  - Puede ser explicita: mismo código repetido en distintas partes.
  - O implicita: dos funciones distintas que cumplen el mismo propósito.
- **Soluciones**:
  - Si es código repetido, usar un módulo común para importar esa clase, método, etc.-
  - Si son dos funciones distintas, elegir solo una forma y unificar ese método.
  - En general hay varias formas de re-estructurar el código duplicado.

Dispensables - Lazy Class

- **Síntomas**: Una clase no cumple un rol que realmente justifique su existencia.
- **Razones del problema**:
  - Cambios en la estructura del código han generado que la clase deje de ser importante.
  - Mala planificación: posibles funcionalidades que nunca se implementaron.
  - Uso de generadores.
- **Soluciones**:
  - Agrupar el código de acuerdo a las necesidades del problema.
  - Si no se programará en ese momento, mejor no considerarlo.

Dispensables - Data Class

- **Síntomas**: Una clase contiene solamente atributos, sin aplicar comportamiento.
- **Razones del problema**:
  - Cambios en la estructura del código que han implicado quitarle métodos.
  - Mala planificación.
- **Soluciones**:
  - Evaluar si efectivamente no tiene comportamiento: ¿Todos los atributos son públicos o privados? ¿Deberían haber getters y setters?.

Dispensables - Dead Code

- **Síntomas**: Una variable, parámetro, campo, método o clase ya no se utiliza.
- **Razones del problema**:
  - Los requerimientos cambian, y no se limpió el código.
  - Lógica de condicionales inaccesible.
- **Soluciones**:
  - Limpiar el código borrando el código que no se necesite.

Tip: los IDE generalmente tienen plugins para detectar código que no se usa.

Dispensables - Speculative Generality

- **Síntomas**: Una variable, parámetro, campo, método o clase que no se utiliza, *pero se planea usar*.
- **Razones del problema**:
  - Mala planificación por posibles funcionalidades.
  - Generalizar lógica de código que no es necesaria.
- **Soluciones**:
  - Evitar usar clases abstractas si es que no se tiene claro que más de alguna clase la usara.

Couplers

Exceso de acoplamiento en el código.

Ilustración extraída de https://refactoring.guru/

Couplers - Feature Envy

- **Síntomas**: Un método utiliza más información de un objeto que recibe, que del mismo objeto en el que está definido.
- **Razones del problema**:
  - Mala modelación.
  - Se crean clases para almacenar datos, pero los métodos que los modifican se crean en otras partes del código.
- **Soluciones**:
  - Mover los métodos a las clases que modifican.

Couplers - Inappropriate Intimacy

- **Síntomas**: Una clase utiliza métodos y atributos internos de otra clase.
- **Razones del problema**:
  - Clases muy ligadas entre sí.
- **Soluciones**:
  - Mover el código donde se usa realmente.
  - Hacer oficial la relación entre las clases (ya sea con composición o herencia).
  - Hacer que la asociación sea solo en una dirección y no cíclica.

Couplers - Message Chains

- **Síntomas**: Excesivas llamadas a métodos en cadena: a->b()->c()->d().
- **Razones del problema**:
  - Dependencia de quien hace la primera llamada sobre el flujo o la estructura.
- **Soluciones**:
  - Mover los métodos y re-definir las responsabilidades.
  - Mover el código importante al inicio de la cadena.

Couplers - Middle Man

- **Síntomas**: La única funcionalidad de la clase es delegar trabajo sin agregar funcionalidad.
- **Razones del problema**:
  - Al reorganizar código puede que haya quedado una clase sin responsabilidades.
- **Soluciones**:
  - Quitar el Middle Man.
    - No quitar si se añadió el Middle Man para evitar dependencias entre clases o si es parte de algún patrón de diseño.

Couplers - Incomplete Library Class

- **Síntomas**: Una librería presenta problemas y no responde a las necesidades del problema.
- **Razones del problema**:
  - Una librería ya no provee las funcionalidades necesarias o tiene un problema de compatibilidad, y no se actualiza en mucho tiempo.
- **Soluciones**:
  - Utilizar soluciones intermedias para hacer compatible la librería.
  - Extraer las funcionalidades importantes de la librería.

05. Refactoring

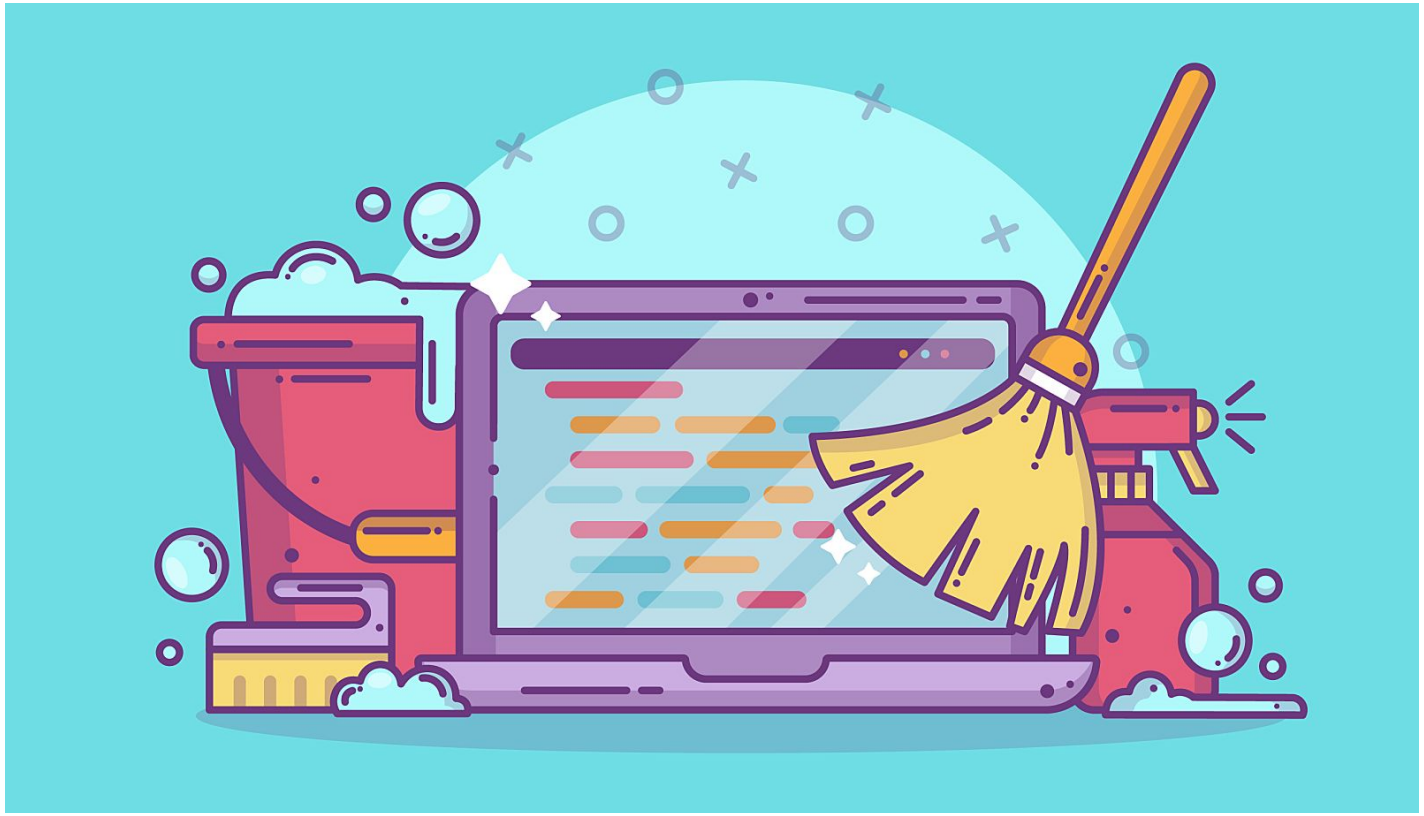
Refactoring

¿Qué es refactoring?

Es el proceso de cambiar un sistema de software de tal forma que no se altera el comportamiento externo, pero que **mejora la estructura interna del código**.

Testing

La forma más eficiente de hacer refactoring es teniendo buenos tests acompañando al código (en las próximas slides).



1: Ilustración de Charlize Cronje.

"Software is not created in one inspired moment. The usual focus on the artifacts of the development process obscures the fact that software development is in fact a process".

- Chad Fowler Co-Director, Ruby Central, Inc. CTO, InfoEther, Inc.

Refactoring - ¿Por qué hacer refactor?

- Mejora el diseño del software.
- Hace el código más fácil de entender.
- Ayuda a encontrar bugs.
- Ayuda a ganar experiencia y al equipo a programar más rápido.

Refactoring - ¿Cuándo hacer refactor?

- **Rule of three:** *Three strikes and you refactor*
  - La primera vez que haces algo, solo terminalo.
  - La segunda vez que haces algo similar, te preocupas por la duplicación, pero puedes dejarlo pasar.
  - La tercera vez, haz el refactor.
- Cuando añadas algo nuevo → considerando el ROI.
- Cuando corriges *bugs*.
- Cuando tu código vaya a pasar por *code review*.

Refactoring - metáfora de los “Dos Sombreros”

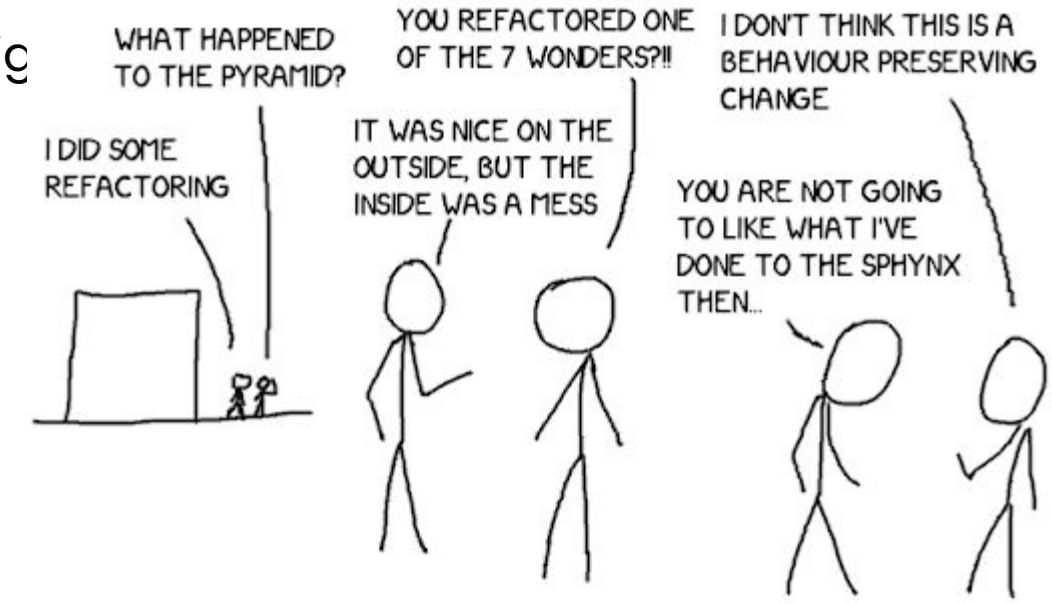
La metáfora de los Dos Sombreros (Two Hats) fue definida por Kent Beck, la cual dice que un desarrollador debería dividir su tiempo en 2 actividades: añadir funcionalidades y hacer refactoring:

- Dentro de las funcionalidades, solo se añaden nuevas funciones y tests para medir el avance.
- Cuando se hace refactor, no se añaden más tests, sólo se re-estructura el código.

Durante el desarrollo, el desarrollador estará cambiando de sombreros de forma constante. Sin embargo es importante para el mismo percatarse cual sombrero tiene puesto.

Refactoring - Problemas

- Managers poco técnicos no le ven el valor.
- No hacer bien el refactoring puede causar cambios en el comportamiento (importancia de los **tests**).
- Hacer refactoring sobre códig
  más problemas que beneficios.
- Hacer refactoring encima
  puede traer complicaciones.



06. Testing

¿Por qué falla el software?

Existen muchas respuestas a esta pregunta. Pero podemos verlo desde dos aristas:

En los **requisitos**:

- Faltan requisitos
- Requisitos mal definidos
- Requisitos no realizables
- Diseño de software defectuoso

En la **implementación**:

- Algoritmos incorrectos
- Implementación defectuosa

Testing

¿Qué es testing?

Son pruebas que permiten **validar** y **verificar** el funcionamiento del software.

Validación: ¿Estamos construyendo el producto correcto?

Verificación: ¿Estamos construyendo el producto correctamente?

Artefacto	**Tipo de test**
Código	Unit testing
Diseño de interfaz	Integration testing
Requerimientos	Validations testing
Sistema	System testing

En la tabla se muestra el orden en que debería hacerse el testing.

Testing - Unit Testing

Se centran en verificar las unidades más pequeñas del software (componentes y módulos).

```ruby
class Dog
    attr_reader :name

    def initialize(name)
        @name = name
    end
end
```

```ruby
RSpec.describe Dog do
    let(:dog_name) { 'Firulais' }

    it 'has a name' do
        dog = Dog.new(dog_name)
        expect(dog.name).to eq(dog_name)
    end
end
```

05. Próxima Clase

Próxima clase

- Repaso práctico
- Patrones de diseño

Clase 3 - Buenas prácticas

Bibliografía

- M. Fowler, UML Distilled.
- S. Ambler, The Elements of UML(TM) 2.0 Style.
- Kiruthika, R.S., Shobana, C. 4+1 View Model.
- P.B. Kruchten. The 4 + 1 view model of architecture.

IIC2113 - Diseño Detallado de Software

Fernanda Sepúlveda - mfsepulveda@uc.cl

Pontificia Universidad Católica de Chile
2020-2