Ingeniería de Software

14 - Diseño + Acoplamiento + Cohesión

IIC2143-3

Josefa España

jpespana@uc.cl

Diseño

Clean Code

"Me gusta que mi código sea elegante y eficiente. La lógica debe ser directa para hacer difícil que los errores se escondan, las dependencias mínimas para facilitar el mantenimiento, el manejo de errores completo de acuerdo a una estrategia articulada, y el rendimiento cercano a lo óptimo para no tentar a las personas a hacer el código desordenado con optimizaciones poco principistas. El código limpio hace una cosa bien."

- Bjarne Stroustrup, inventor de C+

Bjarne Stroustrup, inventor of C++ and author of *The C++ Programming Language*

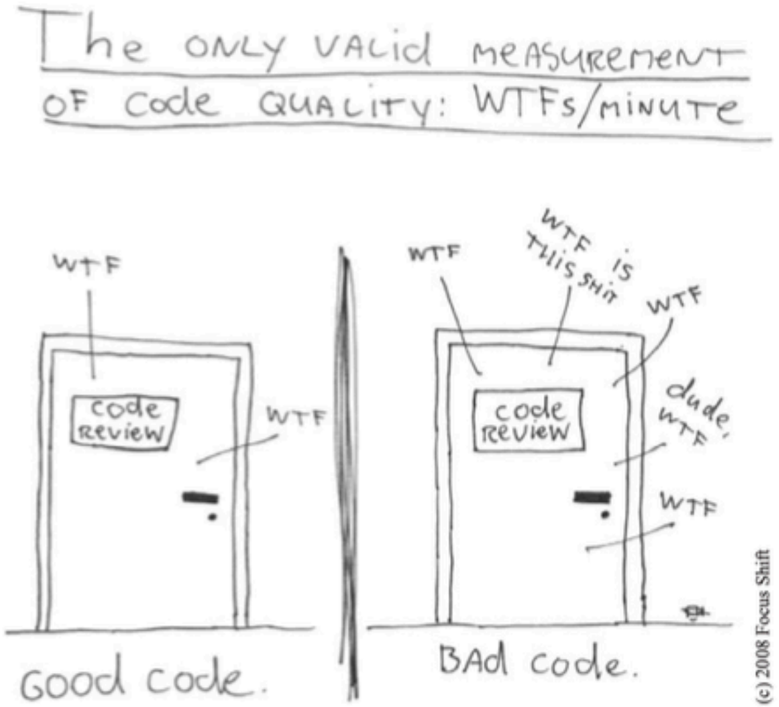
Clean Code

- "... elegante y eficiente ..."
- “... la lógica debería ser directa para hacer difícil que los errores se escondan ..."
- "... las dependencias mínimas para facilitar el mantenimiento ..."
- "... manejo de errores completo de acuerdo a una estrategia articulada ..."
- "... rendimiento cercano a lo óptimo ..."

Bjarne Stroustrup, inventor of C++ and author of *The C++ Programming Language*

Atributos de un buen diseño

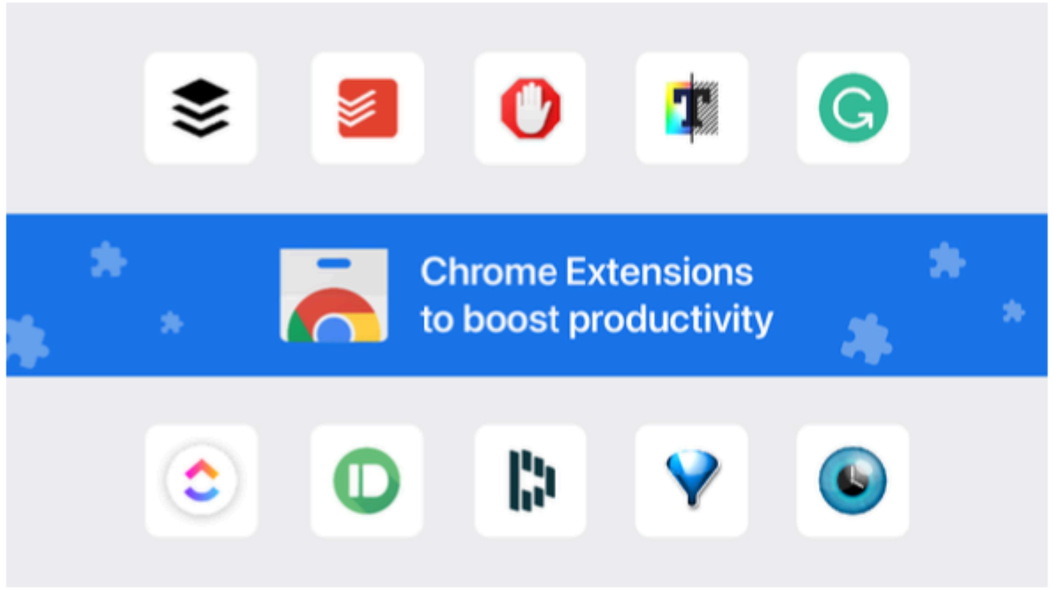
- fácil de entender
- fácil de modificar y extender
- fácil de reutilizar en otro problema
- fácil de testear la implementación
- fácil de integrar las distintas unidades
- facil de implementar (programar)
- ...



¿Qué hace un código extensible y flexible?

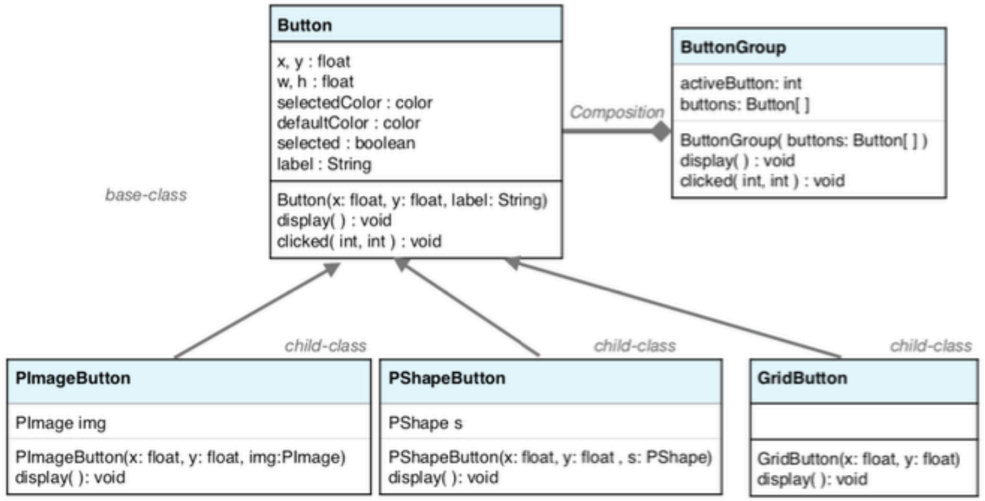
Cuando instalamos un nuevo add-on:

- ¿Hay que recompilar chrome?
- ¿Se modifica el software para que el nuevo add-on funcione?
- ¿Hay que cerrar y abrir chrome?



¿Qué hace un código extensible y flexible?

- Un programa es extensible si es posible extender (agregar) nuevas funcionalidades sin tener que modificar el código actual.
- Principio “abierto para extensiones y cerrado para modificaciones”.
- En programación orientada a objetos el mecanismo que favorece la extensión es la herencia.

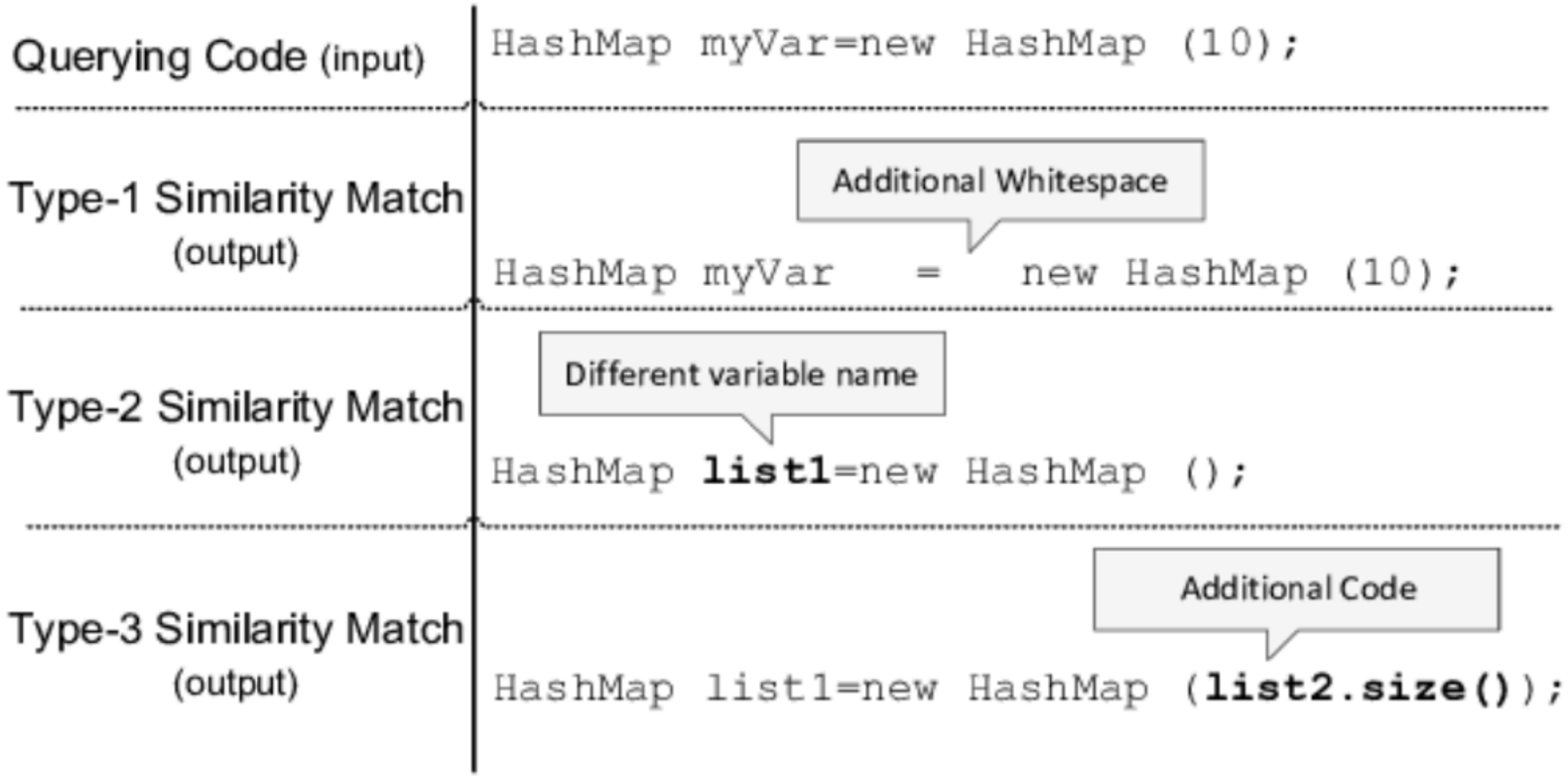


Heurísticas que usaremos

- Evitar código duplicado (code clones)
- Minimizar las dependencias entre clases (acoplamiento)
- Principio de Simple Responsabilidad (cohesion)
- Facilitar el mantenimiento y la extensibilidad.
  - Organizando el código para el cambio.
  - Aislando para el cambio.

Código duplicado

Código duplicado



Código duplicado

- Hipótesis: un programa es más difícil de mantener si existen varios clones.
  - Si se debe modificar un clon, hay que hacer la misma modificación para los demás.
  - A veces se modifica un clon, pero no los demás porque no sabes que existen, generando bugs.

Acoplamiento y Cohesión

Acoplamiento

El acoplamiento se refiere a la interacción entre clases, qué tanto están conectadas.

- Para un buen diseño buscamos reducir el acoplamiento del sistema.
- Ideal tener clases lo más independientes y que se comuniquen con otras clases a través de pocos métodos bien definidos.

Cohesión

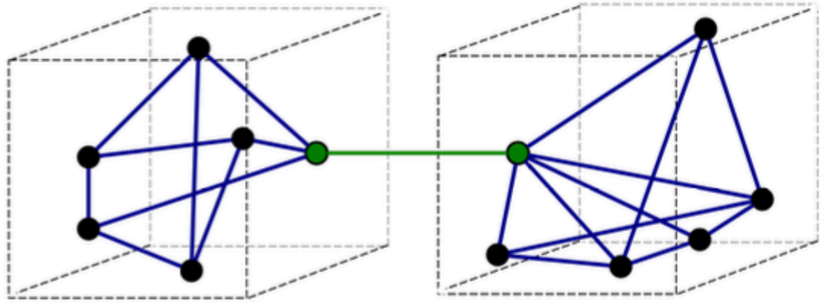
La cohesión describe qué tan bien un código se mapea a una entidad o tarea.

- En un sistema con alta cohesión, cada código (clase, método, módulo) es responsable de una tarea bien definida.
- Buenas clases tienen alto nivel de cohesión.

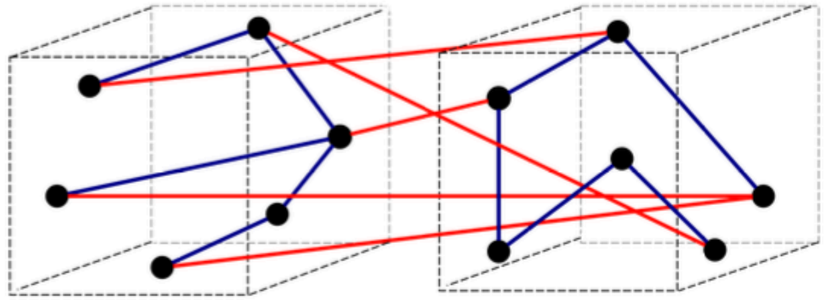
Cohesión

En el ejemplo los cubos representan clases, y cada punto son elementos de la clase, como atributos y métodos.
Si existe una l{inea entre dos elementos, es una dependencia, y entre menos, mejor.

Una dependencia se provoca por ejemplo cuando un método llama a otro, o cuando un atributo es accedido dentro de un método.



a) Good (loose coupling, high cohesion)



b) Bad (high coupling, low cohesion)

Resumen

Propiedades que buscamos que nuestro código tenga:

- Bajo acoplamiento
- Alta cohesión
- Sin código duplicado
- Favorecer el cambio y la extensibilidad
- Haciendo uso de conceptos básicos de POO:
  - Encapsulamiento.
  - Delegación
  - Herencia y Polimorfismo.

Ejemplos

Ejemplos: Shopping Car

ShoppingCar.rb

```ruby
require_relative 'item'
class ShoppingCar
  def initialize
    @items = []
  end

  def add(item)
    @items.push item
  end

  def print
    @items.each do |item|
      puts "name: #{item.name}"
      puts "quantity: #{item.quantity}"
      puts "cost: #{item.totalCost}"
    end
  end
end
```

Item.rb

```ruby
class Item
  def initialize(item_name,item_quantity,item_cost)
    @name = item_name
    @quantity = item_quantity
    @cost = item_cost
  end
  def name
    @name
  end
  def quantity
    @quantity
  end
  def totalCost
    return @cost * @quantity
  end
end

```

main.rb

```ruby
require_relative "shopping_car"
require_relative "item"

car = ShoppingCar.new
car.add(Item.new("Pancito",5,200))
car.print
```

Problemas shopping car

En el código anterior:

- La clase ShoppingCar depende de dos atributos y un método de la clase Item.
  - Incrementa el acoplamiento entre ambas clases de forma innecesaria.
  - Rompe con el encapsulamiento, es decir, que los datos de los objetos deben estar lo más ocultos posibles para evitar dependencias innecesarias.

Ejemplos: Shopping car

ShoppingCar.rb

```ruby
require_relative 'item'
class ShoppingCar
  def initialize
    @items = []
  end

  def add(item)
    @items.push item
  end

  def print
    @items.each do |item|
      item.print
    end
  end
end

```

Item.rb

```ruby
class Item
  def initialize(item_name,item_quantity,item_cost)
    @name = item_name
    @quantity = item_quantity
    @cost = item_cost
  end
  def print
    puts "name: #{@name}"
    puts "quantity: #{@quantity}"
    puts "cost: #{self.totalCost}"
  end
  private:
    def totalCost
      return @cost * @quantity
    end
end

```

main.rb

```ruby
require_relative "shopping_car"
require_relative "item"

car = ShoppingCar.new
car.add(Item.new("Pancito",5,200))
car.print
```

Ventajas nueva versión

En el código mejorado:

- La clase ShoppingCar solo depende del método print, disminuyendo el número de dependencias (acoplamiento).
- La clase Item tienen privados todos sus datos, e incluso el método totalCost ahora es privado. Esto mejora la cohesión y encapsulamiento.

Ejemplos: Book Search

BookStore.rb

```ruby
require_relative 'book'
class BookStore
  def initialize
    @books = []
  end
  def add(book)
    @books.push(book)
  end
  def filterByAuthor(name)
    @books.each do |book|
      if book.author == name
        book.print
      end
    end
  end
  def filterByTitle(title)
    @books.each do |book|
      if book.title == title
        book.print
      end
    end
  end
end
```

Book.rb

```ruby
class Book
  attr_reader :title
  attr_reader :author
  attr_reader :year
  def initialize(title, author, year)
    @title = title
    @author = author
    @year = year
  end
  def print
    puts "#@title by #@author version #@year"
  end
end
```

main.rb

```ruby
require_relative 'book.rb'
require_relative 'book_store.rb'

store = BookStore.new
store.add(Book.new("Testing","juampi",2022))
store.add(Book.new("Ing. Software","jaime",2022))
store.add(Book.new("Testing","rodrigo",2021))

puts 'searching for jaime'
store.filterByAuthor("jaime")

puts 'searching for testing'
store.filterByTitle("Testing")
```

Problemas Book Search

En el código anterior:

- El método filterByTitle y filterByAuthor tiene código duplicado.
  - Si uno quiere agregar nuevos tipos de filtros, se tendría que agregar más métodos a la clase Book, agregando por ende más código duplicado.

Ejemplos: Book Search

```ruby
require_relative 'book'
class BookStore
  def initialize
    @books = []
  end
  def add(book)
    @books.push(book)
  end
  def filter(strategy)
    @books.each do |book|
      if strategy.check(book)
        book.print
      end
    end
  end
end
```

```ruby
class FilterStrategy
  def check(book)
    raise NotImplementedError
  end
end
```

```ruby
require_relative 'filter_strategy'

class ByTitle < FilterStrategy
  def initialize(title)
    @title = title
  end

  def check(book)
    book.title == @title
  end
end
```

```ruby
require_relative 'filter_strategy'

class ByAuthor < FilterStrategy
  def initialize(author)
    @author = author
  end

  def check(book)
    book.author == @author
  end
end
```

Ventajas nueva solución

Para agregar un nuevo tipo de filtro implica:

- Ninguna modificación en la clase Book, ni BookStore.
- Solo se debería crear un nuevo archivo, con una nueva clase que herede de filter strategy.
- Por ejemplo, si queremos agregar un FilterByYear.

Por lo anterior se puede decir que este código es flexible facilita la agregación de nuevos tipos de filtros de forma sencilla (extender), sin tocar el código existente (sin modificar nada).

Lo anterior se conoce con la frase: “abierto para extensiones y cerrado para modificaciones”.

Ejemplos: Beverage

```ruby
class Tea
  def prepareRecipe
    boilWater
    steepTeaBag
    addLemon
    pourInCup
  end
  def boilWater
    puts 'boiling water'
  end
  def steepTeaBag
    puts 'steeping tea'
  end
  def addLemon
    puts 'adding lemon'
  end
  def pourInCup
    puts 'Pouring in cup'
  end
end
```

```ruby
class Coffee
  def prepareRecipe
    boilWater
    brewCoffeeGrinds
    pourInCup
    addSugarAndMilk
  end
  def boilWater
    puts 'boiling water'
  end
  def brewCoffeeGrinds
    puts 'dripping coffee through filter'
  end
  def pourInCup
    puts 'Pouring in cup'
  end
  def addSugarAndMilk
    puts 'adding sugar and milk'
  end
end
```

```ruby
require_relative 'coffee'
require_relative 'tea'

puts '---------'
Coffee.new.prepareRecipe
puts '---------'
Tea.new.prepareRecipe
```

Problemas Beverage

El código anterior tiene código duplicado entre las dos clases.

- Si agregamos un nuevo tipo de bebida agregaríamos más código duplicado.

Ejemplos: Beverage

```ruby
class CaffeineBeberage
  def prepareRecipe
    boilWater
    brew
    pourInCup
    addCondiments
  end
  def boilWater
    puts 'boiling water'
  end
  def pourInCup
    puts 'Pouring in cup'
  end
  def brew
    raise NotImplementedError
  end
  def addCondiments
    raise NotImplementedError
  end
end
```

```ruby
require_relative 'caffeine_beberage'
class Coffee < CaffeineBeberage
  def brew
    puts 'dripping coffee through filter'
  end
  def addCondiments
    puts 'adding sugar and milk'
  end
end
```

```ruby
require_relative 'caffeine_beberage'
class Tea < CaffeineBeberage
  def brew
    puts 'steeping tea'
  end
  def addCondiments
    puts 'adding lemon'
  end
end
```

```ruby
require_relative 'coffee'
require_relative 'tea'

puts '---------'
Coffee.new.prepareRecipe
puts '---------'
Tea.new.prepareRecipe
```

Ventajas nueva versión

Se elimina el código duplicado y facilita la agregación de nuevos tipos de bebida.

- Las nuevas clases que hereden de CaffeineBeberage podrán reutilizar sus método evitando así el código duplicado en futuras modificaciones.

¿Consultas?

Ingeniería de Software

14 - Diseño + Acoplamiento + Cohesión

IIC2143-3

Josefa España

jpespana@uc.cl