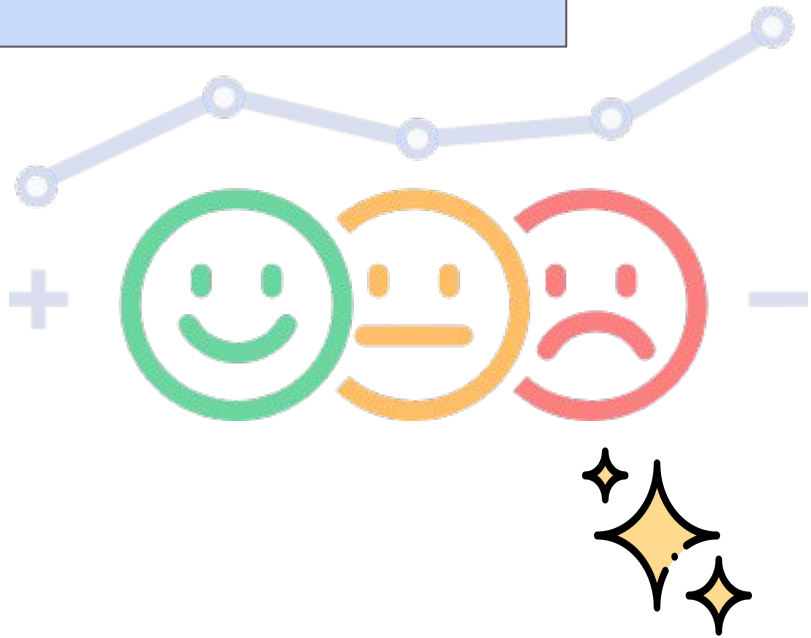
IIC 2143 – Ingeniería de Software

Diseño, acoplamiento y cohesión

M. Trinidad Vargas
mtvargas1@uc.cl

Evaluación Temprana

Si logramos el 61% todo el curso obtiene una décima extra



Canvas > Encuesta de Docencia

Reforzar lo que funciona bien

Cambiar lo que se puede mejorar

Clean Code

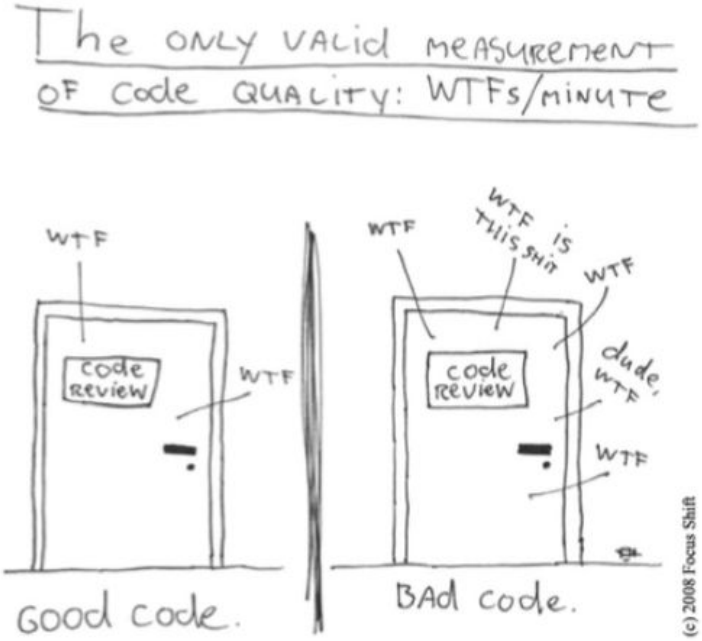
"Me gusta que mi código sea elegante y eficiente.
La lógica debe ser directa para hacer difícil que los errores se escondan,
las dependencias mínimas para facilitar el mantenimiento,
el manejo de errores completo de acuerdo a una estrategia articulada,
y el rendimiento cercano a lo óptimo para no tentar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

Bjarne Stroustrup, inventor de C++
y autor de The C++ Programming Language

Diseño de Software

Un **buen diseño** es

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



Buenas prácticas

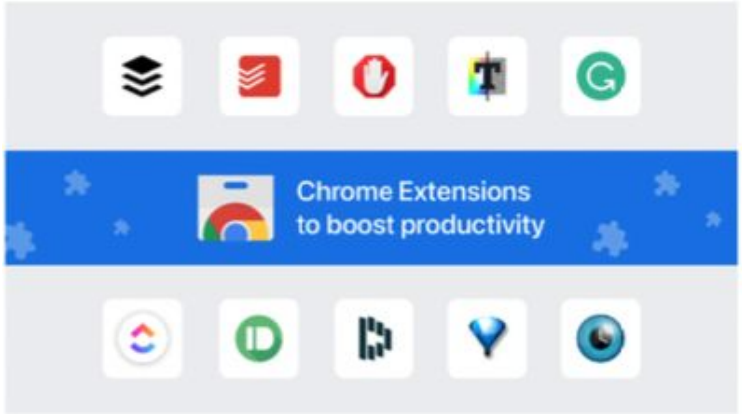
Buenas prácticas que usaremos en el curso:

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

Extensibilidad

Al instalar un add-on (Extensión) en Chrome

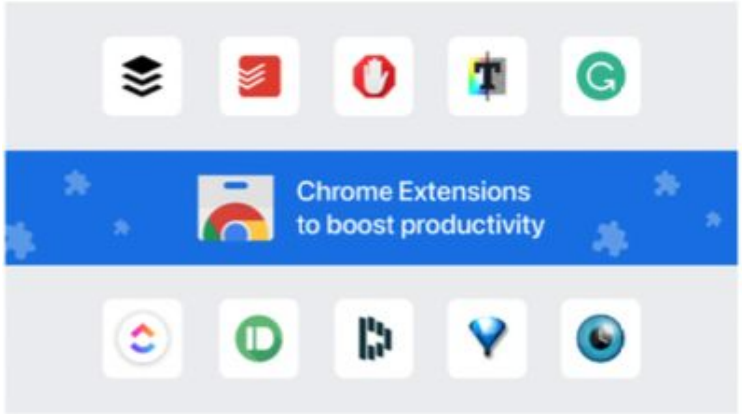
- ¿Hay que recompilar chrome?
- ¿Se modifica el software para que el nuevo add-on funcione?
- ¿Hay que cerrar y abrir chrome?



Extensibilidad

Al instalar un add-on (Extensión) en Chrome

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



No, Chrome permite agregar extensiones sin la necesidad de modificar su código fuente.

Extensibilidad

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.

Código duplicado

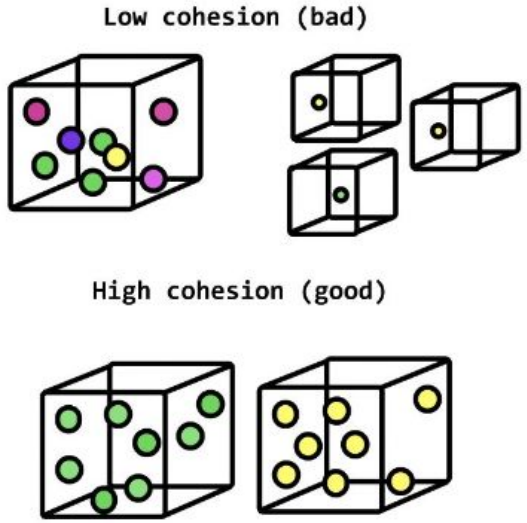
Sucede cuando un pedazo de código (clon) es escrito más de una vez dentro de un programa o una entidad. Usualmente es indeseable tener código duplicado, ya que:

- Si es necesario modificar un clon, entonces es necesario realizar la misma modificación en los demás clones.
- Es posible que alguien modifique un clon sin saber que ese código tiene varios clones, causando varios bugs.
- Al simplificar y acortar la estructura del código es más fácil de mantener

Cohesión

Es el nivel de dependencia de los elementos de un módulo o una clase (atributos y métodos).

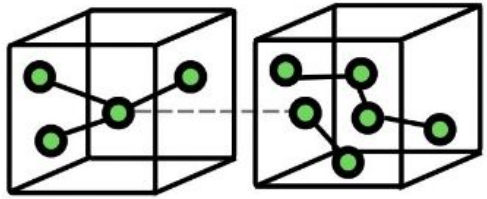
- Si un método hace muchas cosas distintas o tiene partes que no están bien conectadas, tiene baja cohesión.
- Si hay algún método que no utiliza algún atributo de la clase, es posible que ese método pertenezca a otra clase o simplemente no sea un método sino una función.



Acoplamiento

Es el nivel de dependencia entre módulos o clases

- 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 (públicos) bien definidos.

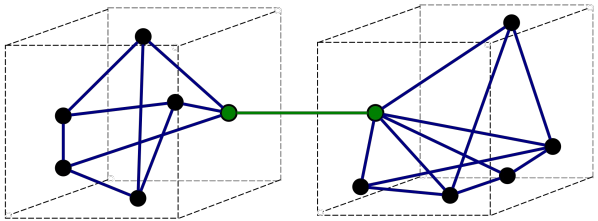


Low coupling

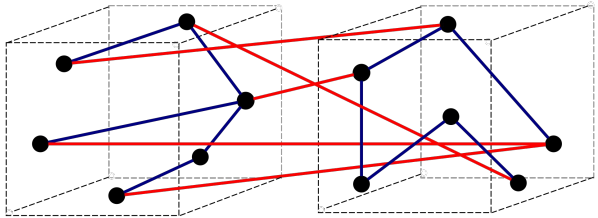
High cohesion

Bajo acoplamiento, alta cohesión

- Los elementos dentro de cada módulo deben estar altamente relacionados (alta cohesión).
- Los módulos deben estar lo menos relacionados posibles (bajo acoplamiento).



a) Good (loose coupling, high cohesion)



b) Bad (high coupling, low cohesion)

En el ejemplo los cubos representan clases, y cada punto son elementos de la clase, como atributos y métodos.

Principio SOLID

Son 5 principios fundamentales del diseño de software orientado a objetos

- **S:** Single Responsibility Principle
- **O:** Open/Closed Principle
- **L:** Liskov Substitution Principle
- **I:** Interface Segregation Principle
- **D:** Dependency Inversion Principle

Principio SOLID

- **S:** Single Responsibility Principle

  Una clase debe tener una sola razón para cambiar.

- **O:** Open/Closed Principle

  El software debe estar abierto a extensión pero cerrado a modificación.

- **L:** Liskov Substitution Principle

  Las subclases deben ser sustituibles por sus clases padre sin alterar el comportamiento del programa.

Principio SOLID

- **I:** Interface Segregation Principle

  Es mejor tener muchas interfaces pequeñas y específicas, que una grande y general.

- **D:** Dependency Inversion Principle

  Las clases deben depender de abstracciones, no de implementaciones concretas.

Code Smells

Es una señal en el código que indica un posible problema de diseño, estructura o mantenimiento, aunque el programa todavía funcione correctamente.

Código duplicado es uno de ellos y veremos otros las próximas clases pero hay muchos más

https://refactoring.guru/refactoring/smells

En resumen

Un código limpio tiene las siguientes propiedades

- 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

En resumen

- Cuando el código es limpio, modular y bien diseñado, el desarrollo se acelera porque cada cambio es más predecible, menos riesgoso y más fácil de implementar.

- Calidad de código no es lujo, es velocidad sostenible.

Ejemplos

Identificar las posibles mejoras argumentando el principio que no se cumple

Ejemplo Shopping Cart

```
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
```

```
require_relative 'item'
class ShoppingCart
 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
```

```
require_relative 'shoppingCart'
require_relative 'item'
car = ShoppingCart.new
car.add(Item.new("Pan", 5, 200))
car.print
```

Ejemplo Shopping Cart

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.

Ejemplo Shopping Cart

```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
```

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

 def add(item)
   @items.push item
 end

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

```ruby
require_relative 'shoppingCart'
require_relative 'item'
car = ShoppingCart.new
car.add(Item.new("Pan", 5, 200))
car.print
```

Ejemplo Shopping Cart

En el código mejorado:

- La clase ShoppingCart 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.

Ejemplo Book Search

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

```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
```

```ruby
require_relative 'book.rb'
require_relative 'book_store.rb'
store = BookStore.new
store.add(Book.new("EDD", "Yadran",
2025))
store.add(Book.new("Ing. Software",
"Josefa", 2022))
store.add(Book.new("Testing", "Juan
Pablo", 2022))
puts 'searching for Yadran'
store.filterByAuthor("Yadran")
puts 'searching for testing'
store.filterByTitle("Testing")
```

Ejemplo 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.

Ejemplo Book Search

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

require_relative 'filterStrategy'
class ByAuthor < FilterStrategy
 def initialize(author)
  @author = author
 end
 def check(book)
  book.author == @author
 end
end

require_relative 'filterStrategy'
class ByTitle < FilterStrategy
 def initialize(title)
   @title = title
 end
 def check(book)
  book.title == @title
 end
end
```

```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
require_relative 'book.rb'
require_relative 'book_store.rb'
store = BookStore.new
store.add(Book.new("EDD", "Yadran",
2025))
store.add(Book.new("Ing. Software",
"Josefa", 2022))
store.add(Book.new("Testing", "Juan
Pablo", 2022))
puts 'searching for Yadran'
store.filterByAuthor("Yadran")
puts 'searching for testing'
store.filterByTitle("Testing")
```

Ejemplo Book Search

Ventajas del código mejorado

Para agregar un nuevo tipo de filtro implica:

- Ninguna modificación en la clase Book, ni BookStore.
- Solo se debe 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”.

Ejemplo Beverage

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

```ruby
require_relative 'coffee'
require_relative 'tea'
puts '---------'
Coffee.new.prepareRecipe
puts '---------'
Tea.new.prepareRecipe
```

Ejemplo Beverage

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

```ruby
require_relative 'coffee'
require_relative 'tea'
puts '---------'
Coffee.new.prepareRecipe
puts '---------'
Tea.new.prepareRecipe
```

Ejemplo Beverage

El código anterior tiene código duplicado entre las dos clases.

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

Ejemplo Beverage

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

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

require_relative 'caffeineBeberage'
class Tea < CaffeinaBeberage
 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
```

Ejemplo Beverage

En el código mejorado 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.