Diseño Orientado al Objeto

Juan Pablo Sandoval

Clean Code

“I like my code to be elegant and efficient. The logic should be straightforward to make it hard for bugs to hide, the dependencies minimal to ease maintenance, error handling complete according to an articulated strategy, and performance close to optimal so as not to tempt people to make the code messy with unprincipled optimizations. Clean code does one thing well”

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

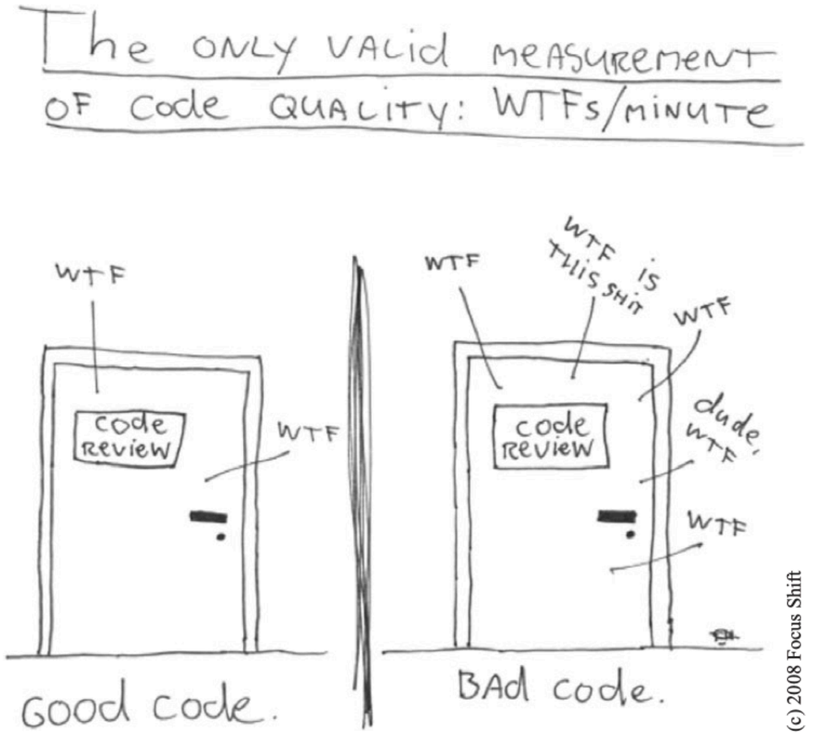
Clean Code

- *"... elegant and efficient …".*
- *" … logic should be straightforward to make it hard for bugs to hide …"*
- *" .. dependencies minimal to ease maintenance …"*
- *" … error handling complete according to an articulated strategy …"*
- *" … performance close to optimal …".*

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

Atributos que caracterizan un buen diseño

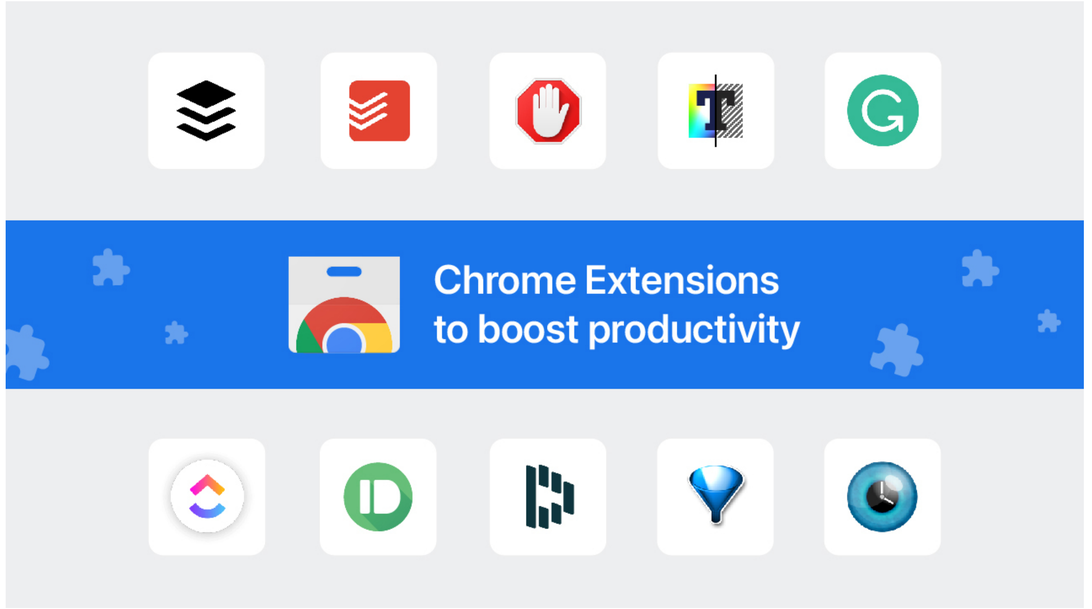
- *fácil de entender*
- ***fácil de modificar y extender***
- *fácil de reutilizar en otro problema*
- *fácil de testear la implementación*
- *fácil integrar las distintas unidades*
- *fácil de implementar (programar)*
- *…*



¿Qué hace un código extensible y flexible?

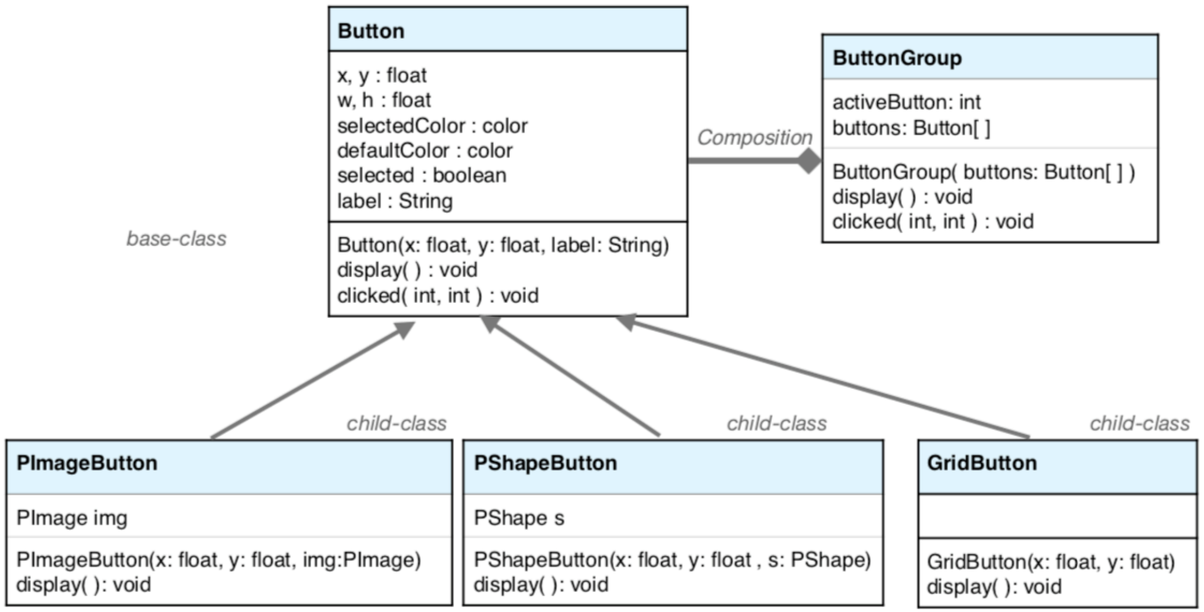
Cuando instalas un nuevo add-on:

- *¿Tienes que re-compilar chrome?*
- *¿Tienes que modificar el software para que el nuevo add-on funcione?*
- *¿Tienes que cerrar chrome y volver a abrir?*



¿Qué hace un código extensible y flexible?

- *En palabras sencillas un programa es extensible si es posible extender (agregar) nuevas funcionalidades sin tener que modificar el código actual.*
- *Esto sigue el principio de **“abierto para extensiones y cerrado para modificaciones”**.*
- *En programación orientada a objeto el mecanismo que favorece la extensión es la **herencia**.*

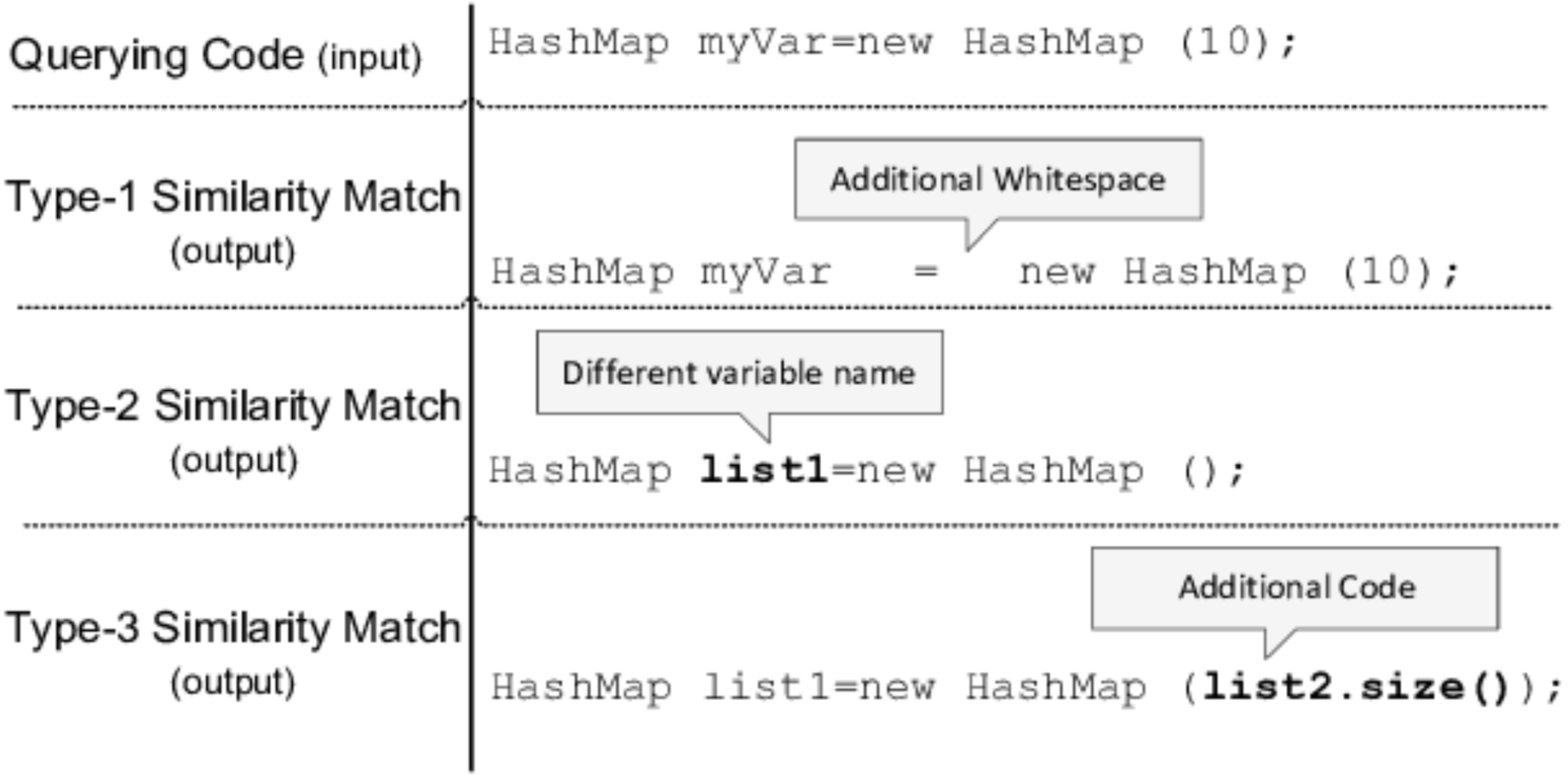


Heurísticas que utilizaremos en el curso

- *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

Code Clones


Querying Code (input)
HashMap myVar=new HashMap (10);
Type-1 Similarity Match
(output)
Additional Whitespace
HashMap myVar = new HashMap (10);
Type-2 Similarity Match
(output)
Different variable name
HashMap list1=new HashMap ();
Type-3 Similarity Match
(output)
Additional Code
HashMap list1=new HashMap (list2.size());

Code Clones

- *La hipótesis es que un programa es mas difícil de mantener si existen varios clones.*
- *Por que? Porque si se necesita modificar un clon, entonces es necesario realizar la misma modificación a los demás clones.*
- *Se encontraron muchos casos donde las personas que modificaron un clon, no estaban al tanto que ese código tiene varios clones. Causando bugs.*

Acoplamiento y Cohesion

Acoplamiento

- *Con el termino acoplamiento nos referimos a la inter conectividad entre clases.*
- *Para tener un buen diseño debemos apuntar a reducir el acoplamiento del sistema.*
- *Lo ideal es tener clases que sean lo mas independientes posibles y que se comuniquen con otras clases a través de pocos métodos bien definidos.*

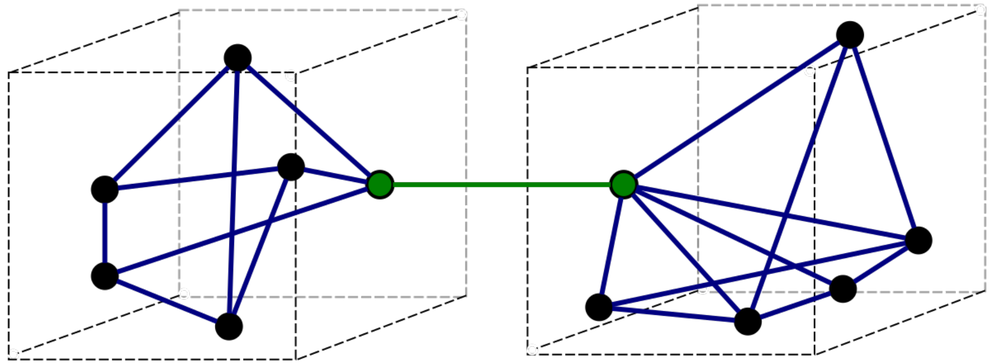
Cohesion

- *Con el termino cohesion describe cuan bien una unidad de código se mapea a una entidad o tarea.*
- *En un sistema con alta cohesion cada unidad de código (clase, método, o modulo) es responsable de una tarea bien definida.*
- *Buenas clases exhiben un alto nivel de cohesion.*

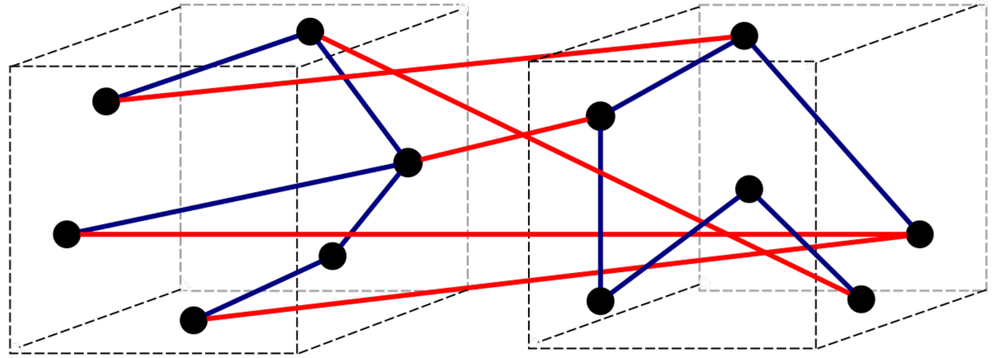
Cohesion

En la visualización de la derecha, las clases son cubos, los puntos dentro de cada cubo son elementos de la clase (atributos y métodos). Existen una linea entre dos elementos si es que una dependencia.

Por ejemplo, existen dependencia 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

Que propiedades queremos que nuestro código tenga:

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

Ejemplo 1: 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
```

Problema

En el código anterior:

- *La clase ShoppingCar depende de dos atributos y un método de la clase Item.*
  - *Lo anterior incrementa el acoplamiento entre ambas clases de forma innecesaria.*
  - *Ademas también rompe el encapsulamiento.*
    - *Encapsulamiento, se refiere a que los datos de los objetos deben estar lo mas ocultos posibles para evitar dependencias innecesarias como la anterior.*

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

```

Ventajas de esta nueva version. *En el código mejorado, la clase ShopppingCar solo depende del método print. Disminuyendo el numero de dependencias (acoplamiento).*

La clase Item, tiene privados todos sus datos, e incluso pusimos el método totalCost en privado. Mejorando la cohesion y encapsulamiento.

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
```

Ejemplo 2: 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")
```

Problema

En el código anterior:

- *El método filterByTitle y filterByAuthor tiene código duplicado.*
  - *El problema esta en que si uno quiere agregar nuevos tipos de filtros, se tendría que agregar mas métodos a la clase Book, agregando mas código duplicado.*

```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 del código anterior

- *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”*

Ejemplo 3: 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
```

***Problema**. El código anterior tiene código duplicado entre las dos clases.*
Si agregamos una nuevo tipo de bebida agregaríamos mas código duplicado.

```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 de esta 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 métodos evitando así el código duplicado en futuras modificaciones.*

Ejercicio en clase

- *Considere el código “students-bad-design” en canvas. El mismo permite subir las notas de los estudiantes de dos formas:*
  - *Forma 1: se sube las notas de todos los estudiantes de forma tal que el que tiene la mejor nota del curso tenga 7.*
  - *Forma 2: se sube las notas de todos los estudiantes de forma tal que el que tiene la peor nota del curso tenga 4.*

- ***Ejercicio**. El código en “students-bad-design” tiene varios problemas de diseño visto en clases. Modifique el código dentro de students-v1para mejorar su calidad.*
  - *El nuevo diseño debe eliminar el código duplicado.*
  - *Ademas debe permitir nuevas formas de subir las notas a los estudiantes.*
    - *Por ejemplo, sumar a todos los estudiantes 1 décima :)*