# Patrones de Diseño: Comportamiento

Juan Pablo Sandoval

# *Qué es un patrón de Diseño?*

- *Los patrones de diseño son soluciones habituales a problemas que ocurren con frecuencia en el diseño de software.*
- *Son como planos pre-fabricados que se pueden personalizar para resolver un problema de diseño recurrente.*
- *El patrón no es una función de código sino un concepto general para resolver un problema en particular.*

# *En que consiste un patron?*

- ***Propósito**: El patrón explica brevemente el problema y la solución.*
- ***Motivación**: La motivación explica en más detalles problema y la solución que brinda el patrón.*
- ***Estructura**: La estructura de las clases muestra cada una de las partes del patron y el modo en que se relacionan.*

# *Patrones de Comportamiento que veremos*

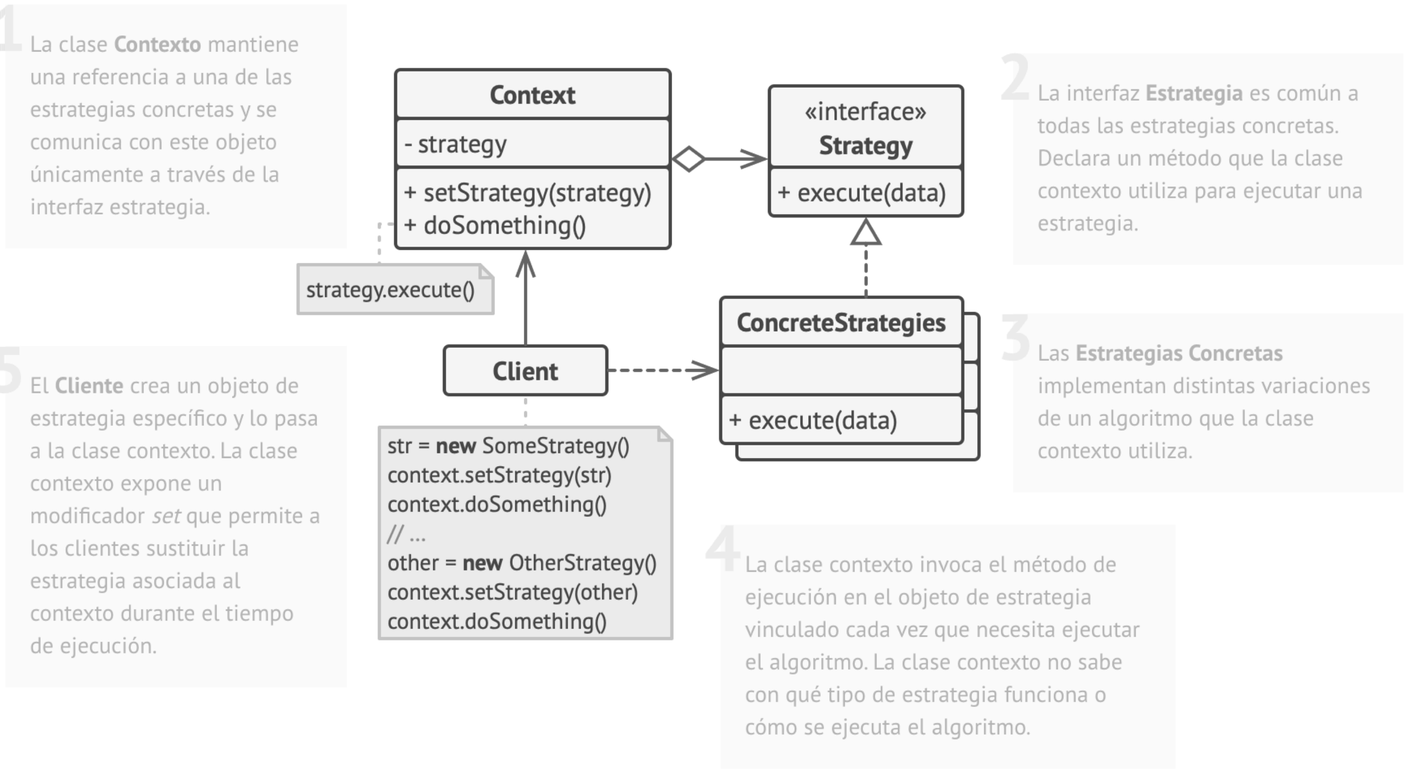
- *Strategy*
- *Template Method*
- *Observer*

# *Strategy*

*Strategy es un patrón de diseño de comportamiento que permite definir una familia de algoritmos, colocar cada uno de ellos en una clase separada y hacer sus objetos intercambiables.*

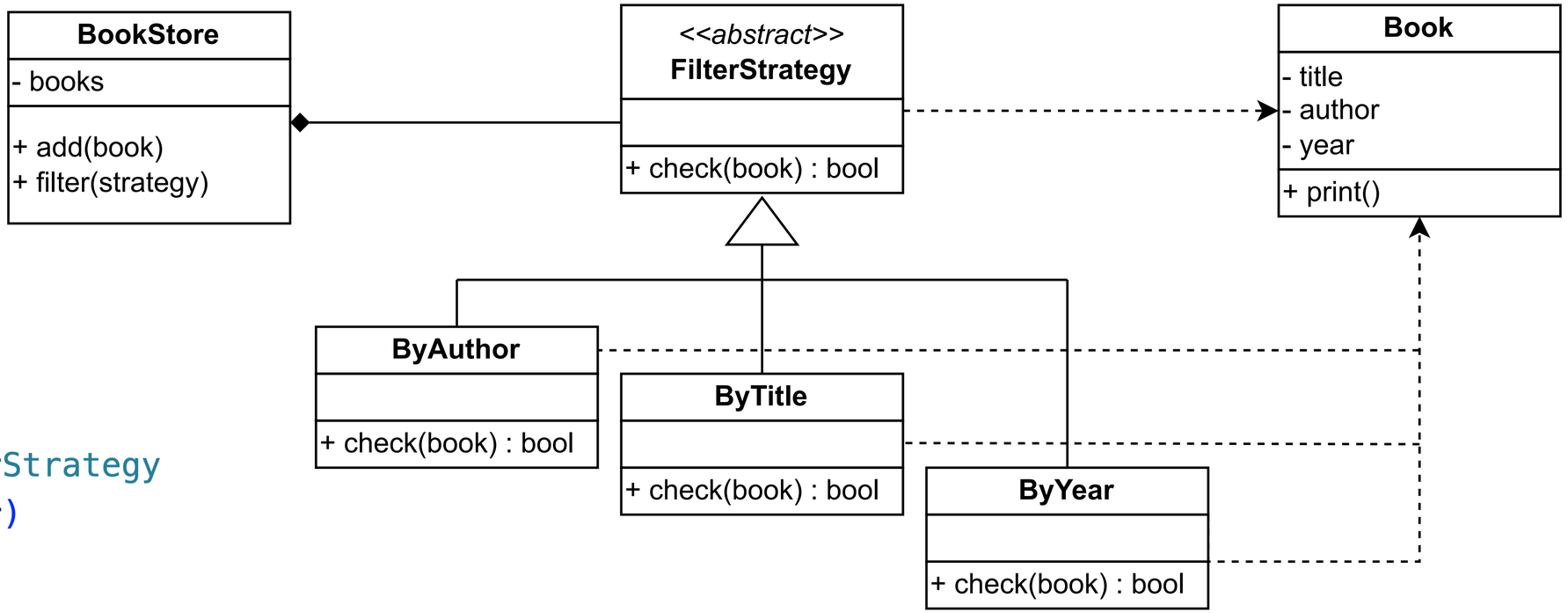
*La idea es que cada algoritmo este en una clase separada pero tengan uno o mas métodos públicos con la misma firma (nombre y número de argumentos). De esta forma la clase que usa las estrategias interactuara con estos objetos utilizando estos métodos sin saber qué estrategia (o algoritmo) en concreto esta utilizando.*

*Los métodos públicos en común deben estar definidos en la clase padre como “abstractos” (en ruby) para obligar a que las clases hijas los sobre-escriban.*




https://refactoring.guru/es/design-patterns/strategy

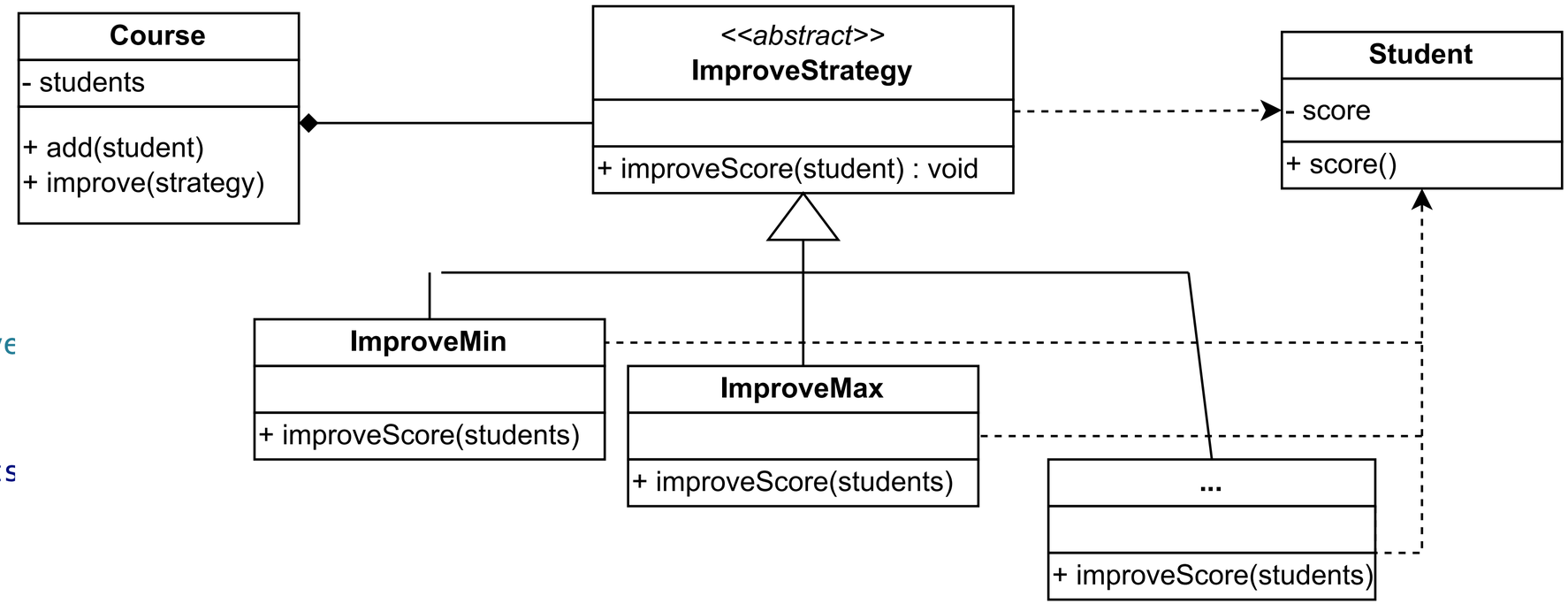
# *Ejemplo - Visto la anterior clase*




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

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

# *Ejercicio - Visto la anterior clase*




```ruby
class ImproveMinStrategy < Improve
    def improveScore(students)
        min = minScore(students)
        incrementBy(4-min,students
    end
    def minScore(students)
        minScore = 7
        @students.each do |student|
            if  minScore > student.score
                minScore = student.score
            end
        end
        minScore
    end
end
```

# *Template Method*

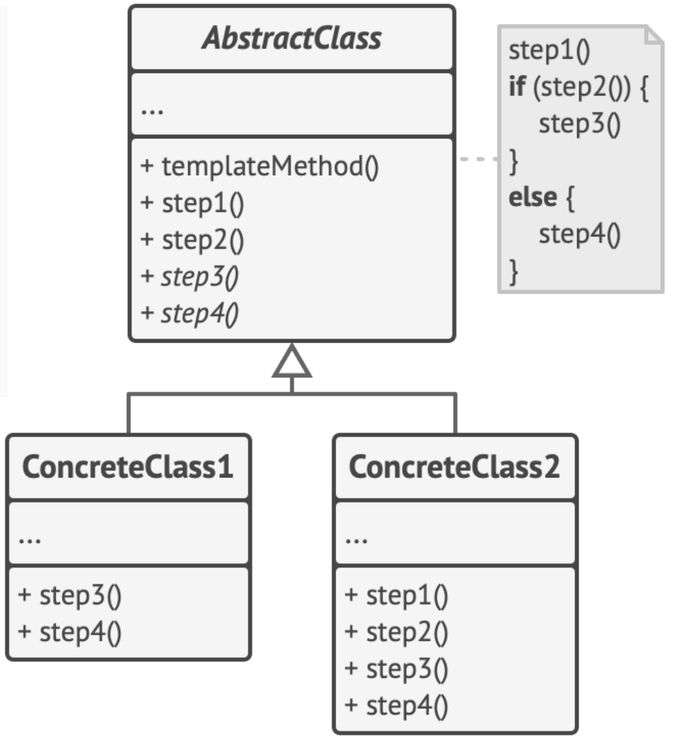
***Template Method*** *es un patrón de diseño de comportamiento que define el esqueleto de un algoritmo en la super clase pero permite que las subclases sobrescriban pasos del algoritmo sin cambiar su estructura.*

*La idea es que hay un método en la clase padre al cual se denomina plantilla. La misma llama a varios métodos (por ejemplo, step1, step2, step3) y amenos uno de estos métodos es abstracto.*

*Al tener un método abstracto la clase es abstracta. Las clases hijas tiene que definir los pasos faltantes, y si es necesario redefinir algunos pasos.*

1 La **Clase Abstracta** declara métodos que actúan como pasos de un algoritmo, así como el propio método plantilla que invoca estos métodos en un orden específico. Los pasos pueden declararse `abstractos` o contar con una implementación por defecto.

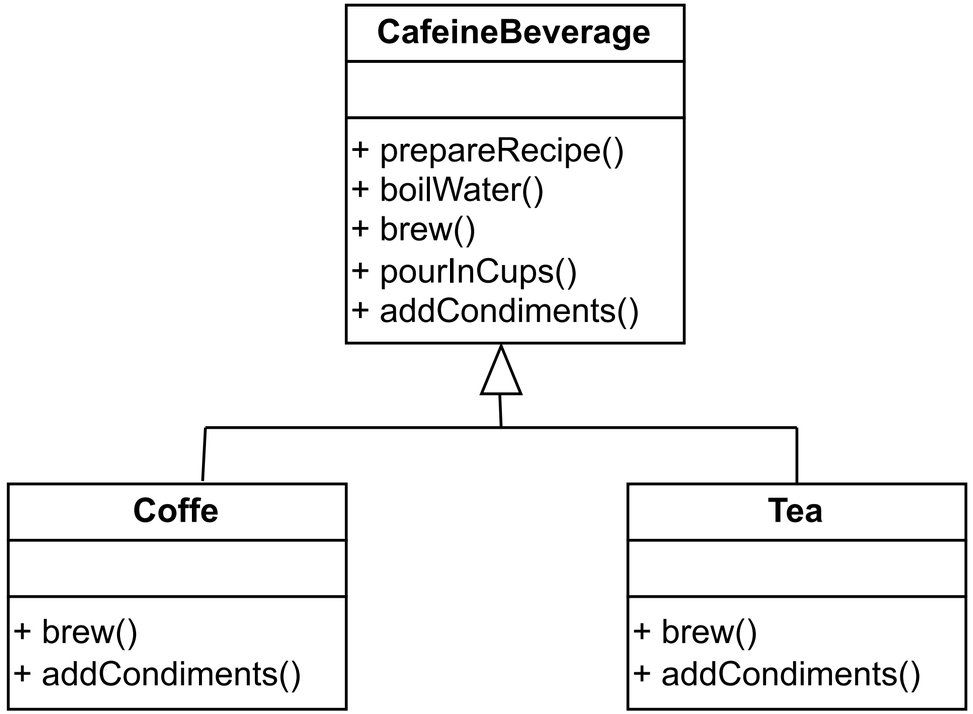
2 Las **Clases Concretas** pueden sobrescribir todos los pasos, pero no el propio método plantilla.




https://refactoring.guru/es/design-patterns/template-method

# *Ejemplo - Visto la anterior clase*

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




La clase Tea solo sobre escribe los pasos necesarios

# Ejercicio - Cola Virtual

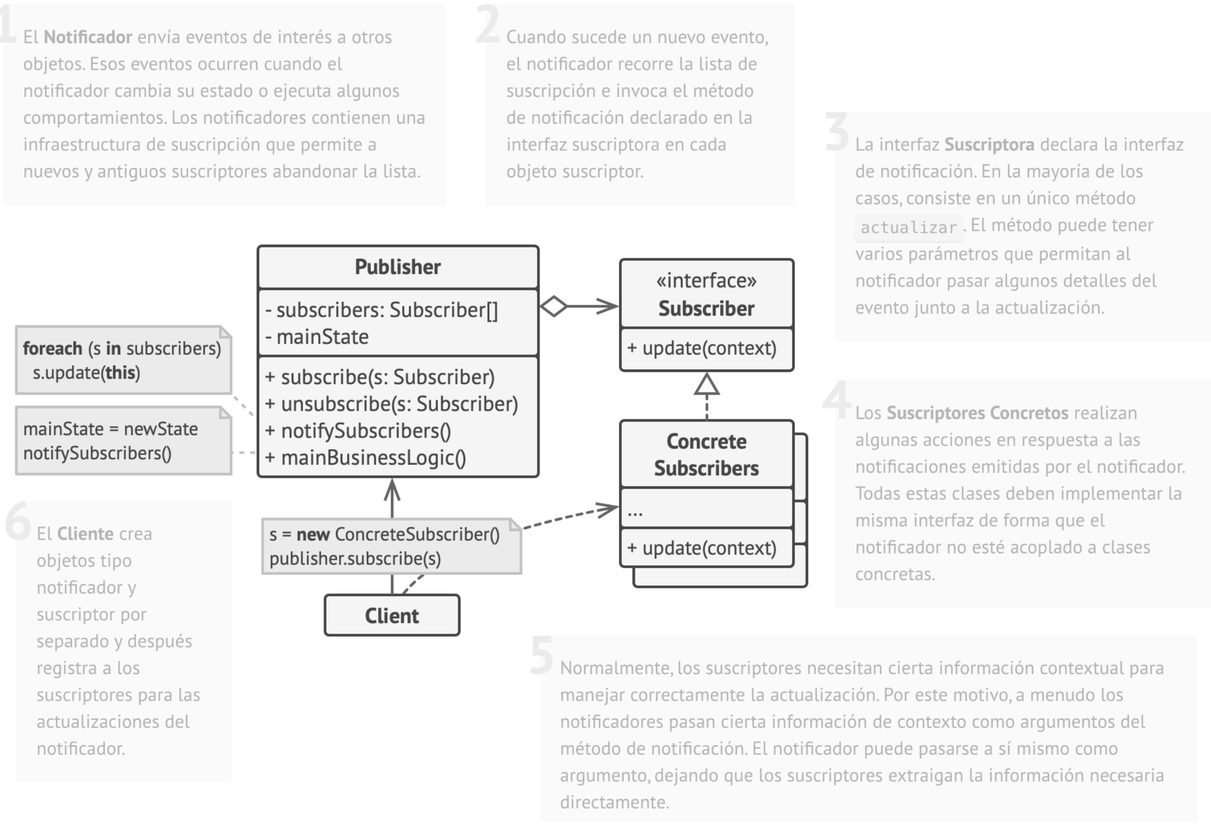
```ruby
def release()
        if @currentNumber < @nextClientNumber
            @currentNumber = @currentNumber + 1
            @onlyOneSubscriber.each do |s|
                if s.number >= @currentNumber
                    if s.number == @currentNumber
                        puts "Send, it is your turn, to: #{s.phoneNumber}"
                    end
                end
            end
            @lastTenSubscriber.each do |s|
                if s.number >= @currentNumber
                    if  (s.number - @currentNumber) < 10
                        puts "Send, there are #{s.number - @currentNumber} before you, to:#{s.phoneNumber
                    end
                end
            end
            @alwaysSubscriber.each do  |s|
                if s.number >= @currentNumber
                    puts "Send, the number #{@currentNumber} is being attended, to:#{s.phoneNumber}"
                end
            end
        end
    end
```

# *Observer*

*Observer es un patrón de diseño de comportamiento que te permite definir un mecanismo de suscripción para notificar a varios objetos sobre cualquier evento que le suceda al objeto que están observando.*

*La idea es que hay un objeto que tiene un estado que queremos observar, por ejemplo, el valor de unas de sus variables. Cada vez que cambie esta variable notificaremos a todos los objetos que lo están observando sin importar su tipo. Mandando toda la información que queramos que todos los observadores sepan.*

*Cada observador realizara las acciones que necesite con esa información. En este sentido, el objeto observado no sabe que acciones harán los observadores volviéndolo mas independiente. Reduciendo el acoplamiento.*




https://refactoring.guru/es/design-patterns/observer

# *Ejemplo - Version mejorada del ejemplo anterior*