PATRONES DE DISEÑO

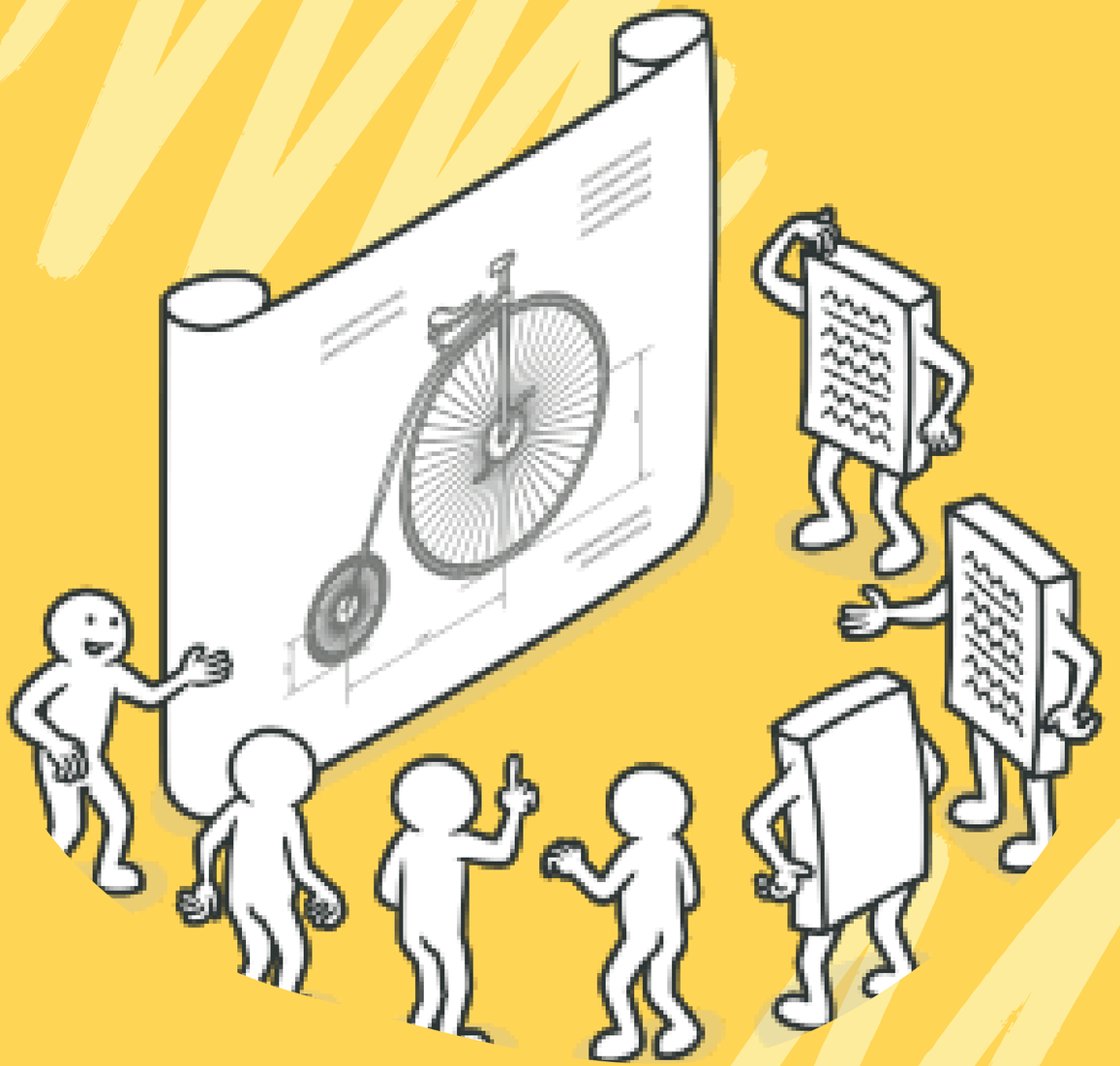
Soluciones existentes a problemas comunes

IIC2143 2023-2

Ayudantes: Trinidad G. Benjamín G. Jean F.

PATRONES DE DISEÑO

Un patrón de diseño es una solución general previamente pensada para resolver problemas que ocurren con alta frecuencia en el diseño de software.



TIPOS DE PATRONES

CREACIONALES

Proporcionan mecanismos de creación de objetos que hacen el codigo más flexible y permite reutilizar codigo existente.

ESTRUCTURALES

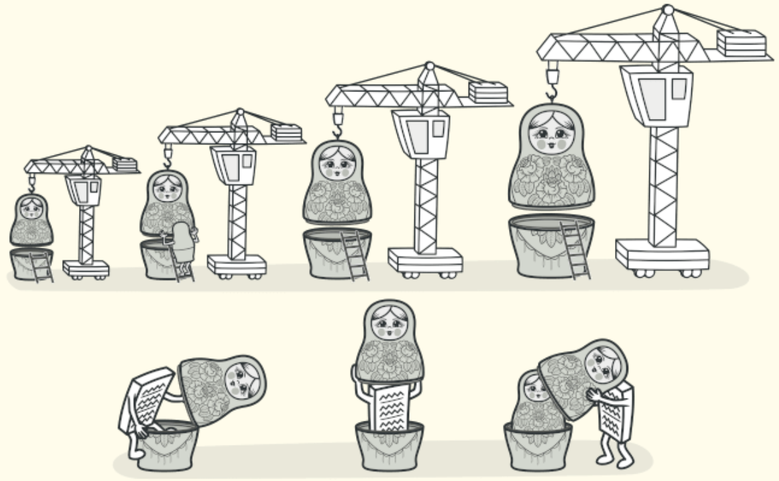
Explican como ensamblar objetos y clases en estructuras más grandes manteniendo flexibilidad y eficiencia en su estructura.

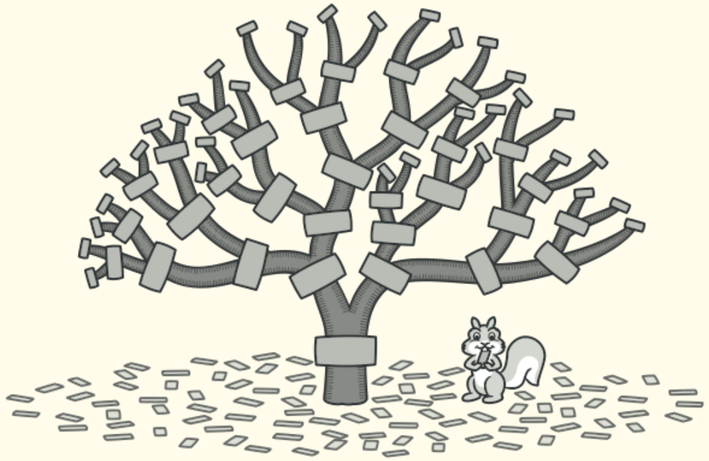
COMPORTAMIENTO

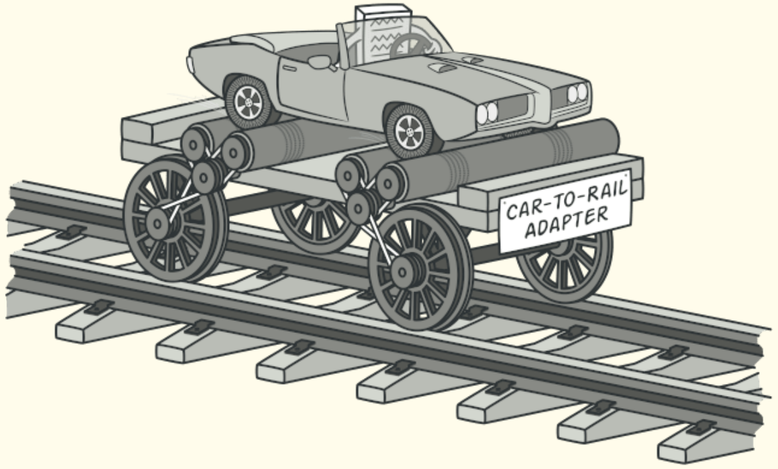
Permite manejar de manera efectiva la asignación de responsabilidades y la comunicación entre objetos.

PATRONES

ESTRUCTURALES

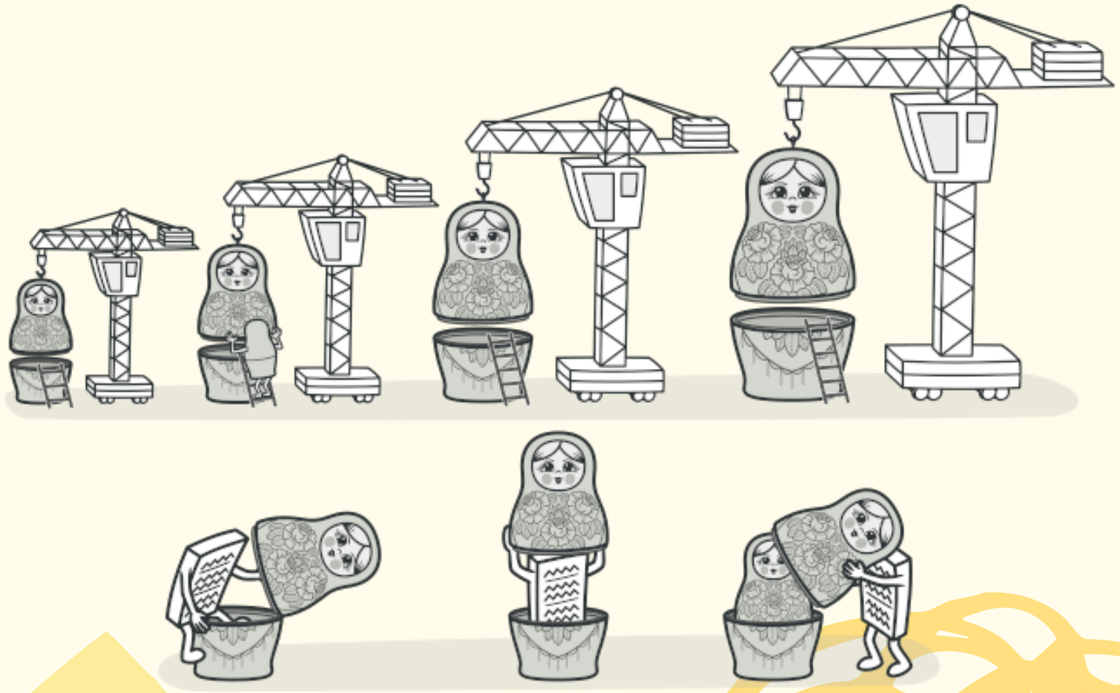




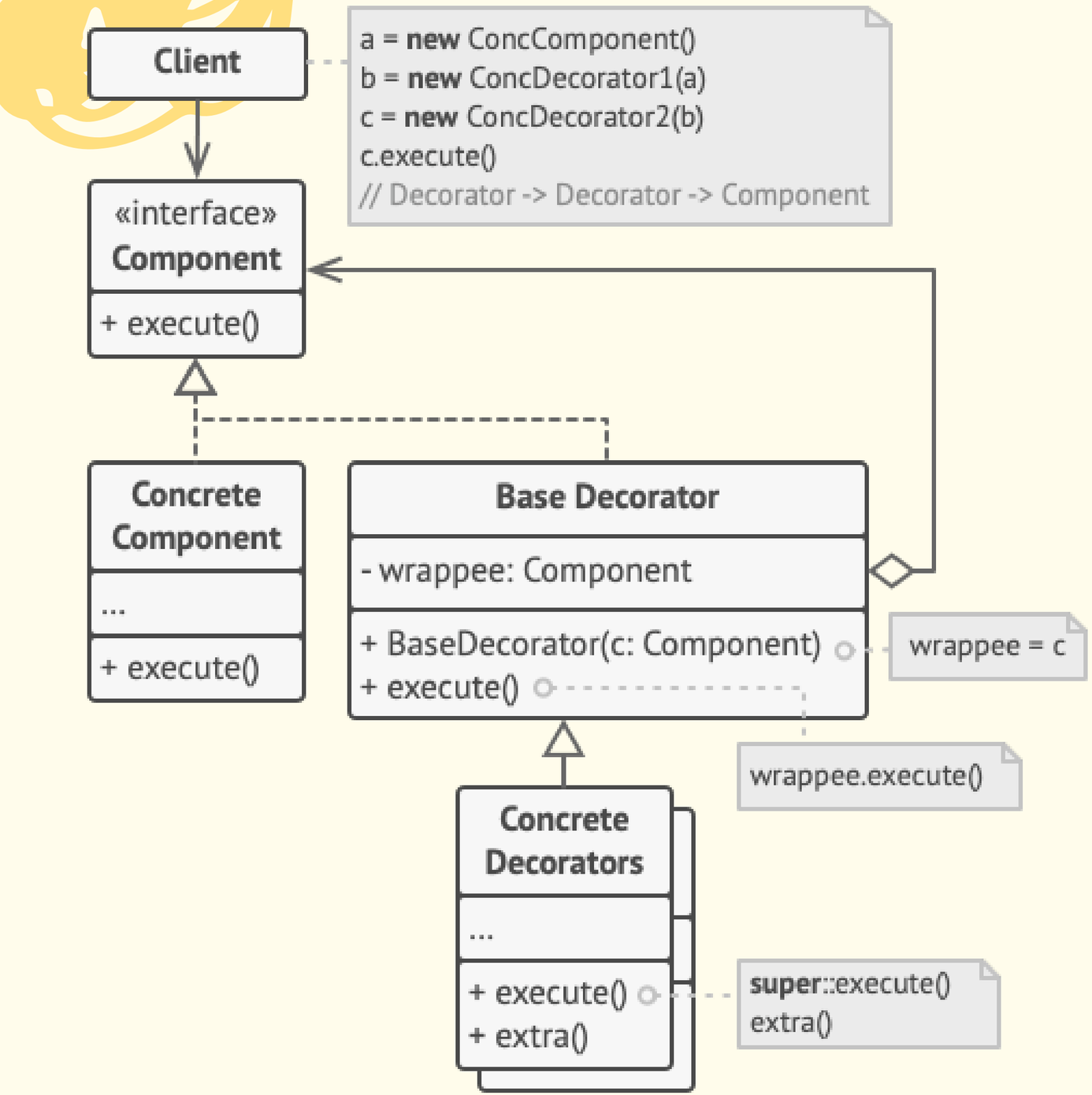


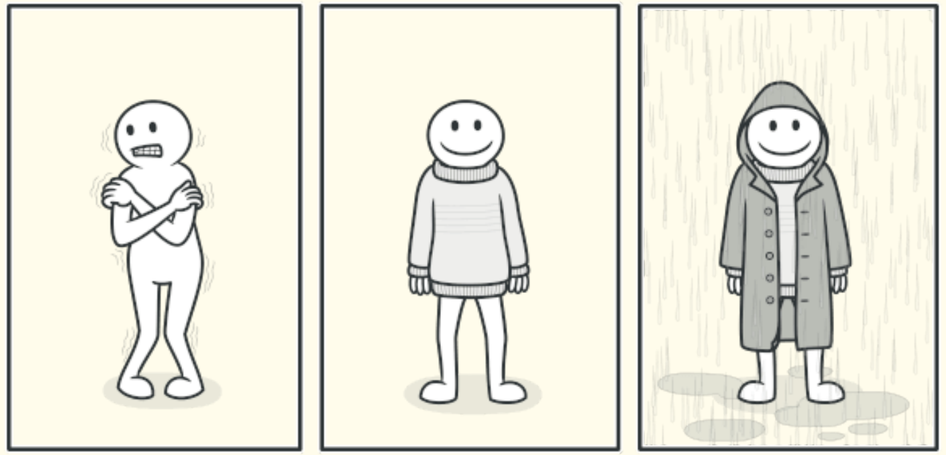
DECORATOR

- Permite agregar responsabilidades en forma dinámica a un objeto.
- Se agrega el comportamiento poniendo a estos en un objeto wrapper en runtime.
- Wrapper: Objeto que puede relacionarse con otro y modificar su comportamiento.



DECORATOR


Client
a = new ConcComponent()
b = new ConcDecorator1(a)
c = new ConcDecorator2(b)
c.execute()
// Decorator -> Decorator -> Component
«interface»
Component
+ execute()
Concrete
Component
...
+ execute()
Base Decorator
- wrappee: Component
+ BaseDecorator(c: Component)
+ execute()
wrappee = c
wrappee.execute()
Concrete
Decorators
...
+ execute()
+ extra()
super::execute()
extra()



P8 REC-12-2023-1

Laura es nueva en el equipo de desarrollo y observa un programa para pedidos de cafe. Por el momento, Laura nota que este programa está en su primera versión y es posible vender expressos con algunos agregados (p.ej., leche batida, mocha). Teniendo en cuenta el código de la primera versión, Laura debe modificar el programa para : (i) agregar más variedad de bebidas y agregados y (i) calcular los precios de la manera más simple y más manejable si se realizan cambios. Ayuda a Laura a modificar el código utilizando el patron Decorator para facilitar la extensibilidad.

```
class Beverage
      def description
            puts "Sample beverage"
      end
      def cost
            raise NotImplementedError
      end
end
```

```
class Expresso < Beverage
      def description
            puts "Expresso"
      end
      def cost
            return 5.5
      end
end
```

```
class ExpressoMocha < Beverage
      def description
            puts "Expresso"
            puts "Mocha"
      end
      def cost
            return 5.5 + 1.5
      end
end
```

```
class ExpressoMochaWhip < Beverage
      def description
            puts "Expresso"
            puts "Mocha"
            puts "Whip"
      end
      def cost
            return 5.5 + 1.5 + 0.5
      end
end
```

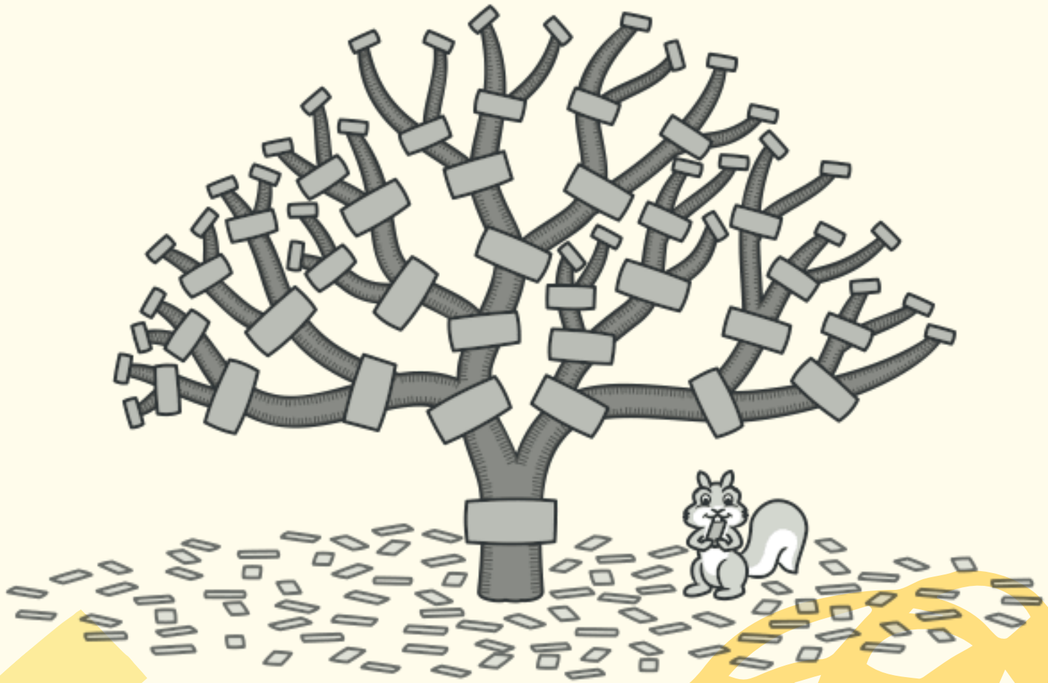
P8 12-2023-1

```
class Beverage
      def description
            puts "Sample beverage"
      end
      def cost
            raise NotImplementedError
      end
end

class Expresso < Beverage
      def description
            puts "Expresso"
      end
      def cost
            return 5.5
      end
end

class DecoratedExpresso < Beverage
    def initialize(decoratedExpresso)
        @decoratedExpresso = decoratedExpresso
    end
```

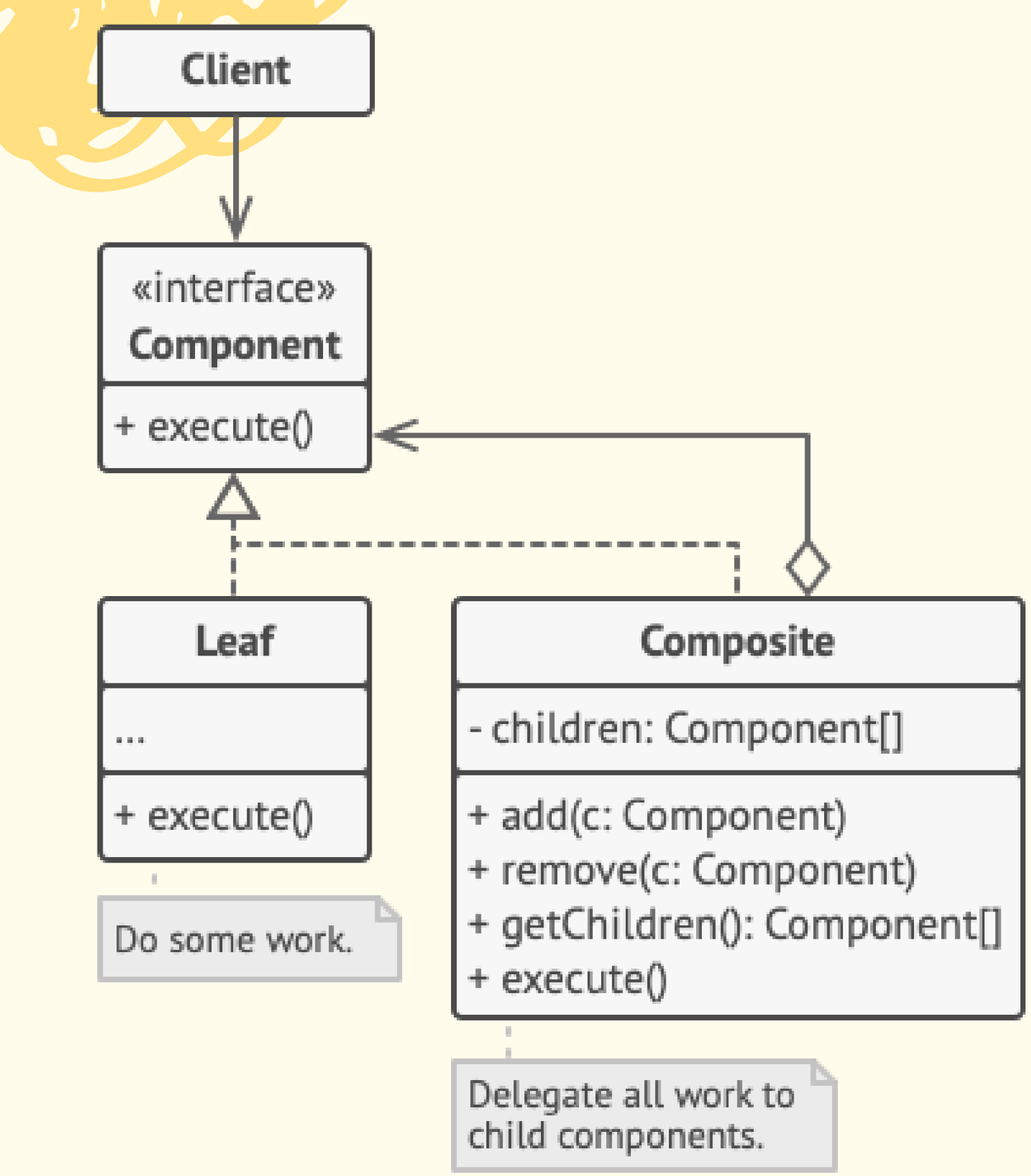
```
class ExpressoMocha < DecoratedExpresso
      def description
            @decoratedExpreso.description()
            puts "Mocha"
      end
      def cost
            return @decoratedExpreso.cost + 1.5
      end
end

class ExpressoMochaWhip < DecoratedExpresso
      def description
            @decoratedExpreso.description()
            puts "Whip"
      end
      def cost
            return @decoratedExpreso.cost + 0.5
      end
end
```

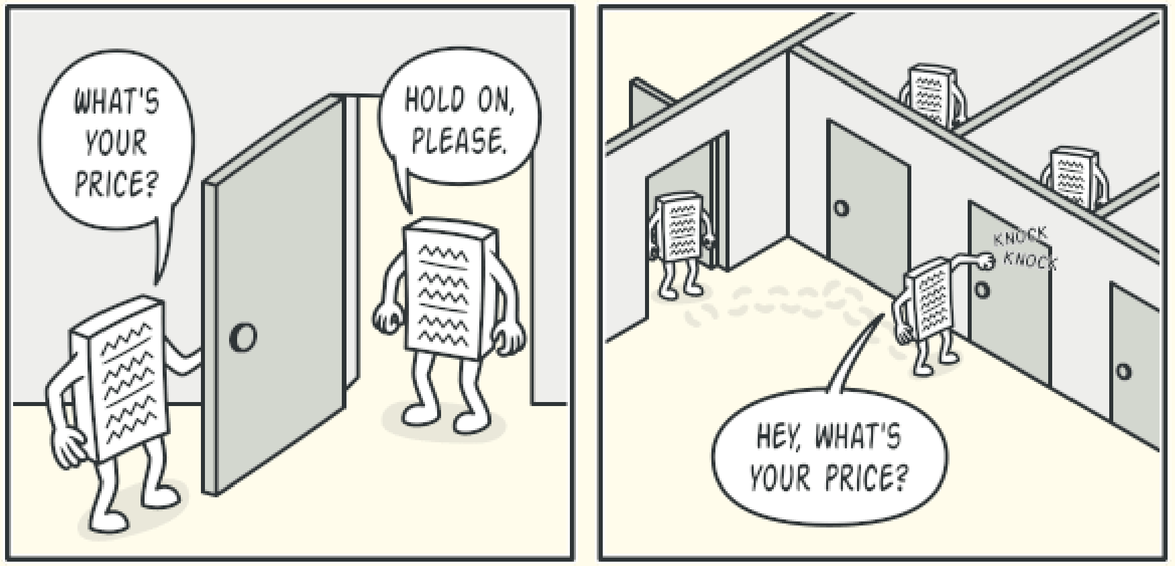
COMPOSITE

- Permite componer objetos en forma de estructuras de arbol y trabajar con estas estructuras de arbol como si fueran objetos individuales.
- Manejo de objetos que tiene estructuras jerárquicas de forma que subestructuras se manejan igual (como las carpetas).



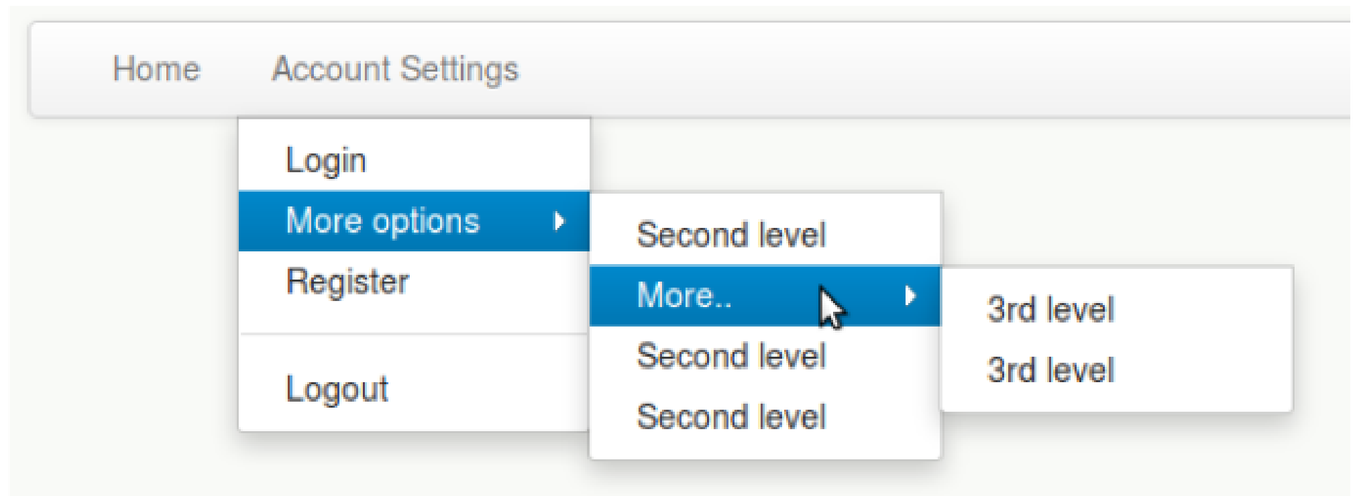
COMPOSITE


Client
«interface»
Component
+ execute()
Leaf
...
+ execute()
Do some work.
Composite
- children: Component[]
+ add(c: Component)
+ remove(c: Component)
+ getChildren(): Component[]
+ execute()
Delegate all work to child components.


WHAT'S YOUR PRICE?
HOLD ON, PLEASE.
KNOCK KNOCK
HEY, WHAT'S YOUR PRICE?

P12 12-2023-1

... se le asignó la tarea de crear un modelo que permita agregar menús y submenús a la barra de opciones. Existen dos tipos de componentes: menu-compuesto y menu-simple. Los menús compuestos pueden contener sub-menus que a su vez pueden ser compuestos o simples. Usted debe dibujar un diagrama de clases para la ilustrar la solución propuesta utilizando el patrón composite.

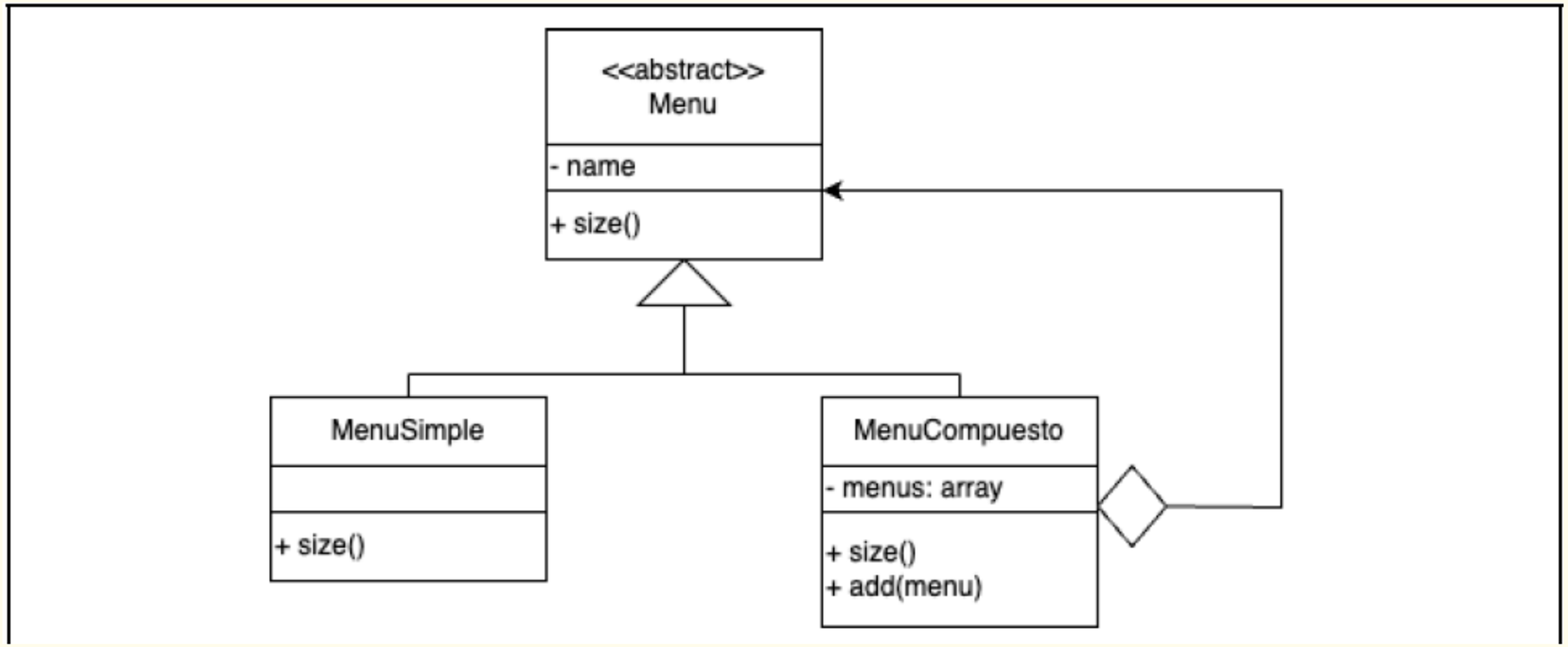


P12 12-2023-1

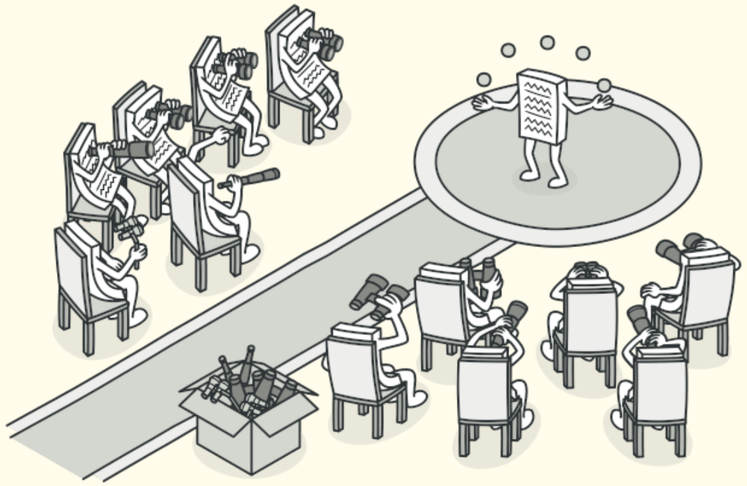
Ademas, se le entrega un caso de prueba para ayudarlo en su diagrama

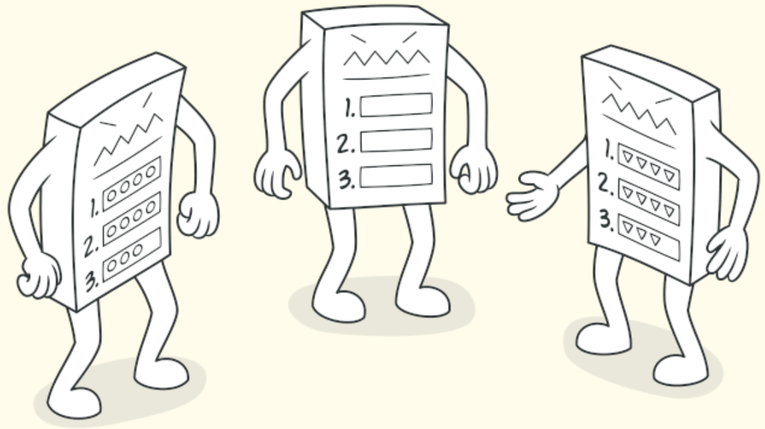
```
menu = Menu.new("file")
sub_menu1 = Menu.new("new...")
sub_menu11 = Menu.new("ruby ...")
sub_menu11.add(MenuItem.new("rb file ..."))
sub_menu11.add(MenuItem.new("rake file ..."))
sub_menu1.add(sub_menu11)
menu.add(sub_menu1)
sub_menu2 = Menu.new("open recent ...")
sub_menu2.add(MenuItem.new("interrogacion1 file"))
sub_menu2.add(MenuItem.new("interrogacion2 file"))
sub_menu2.add(MenuItem.new("interrogacion2 recuperativa"))
menu.add(sub_menu2)

puts "#{menu.size}"
puts "#{menu1.size}"
puts "#{menu2.size}"
```

P12 12-2023-1


<<abstract>>
Menu
- name
+ size()
MenuSimple
+ size()
MenuCompuesto
- menus: array
+ size()
+ add(menu)

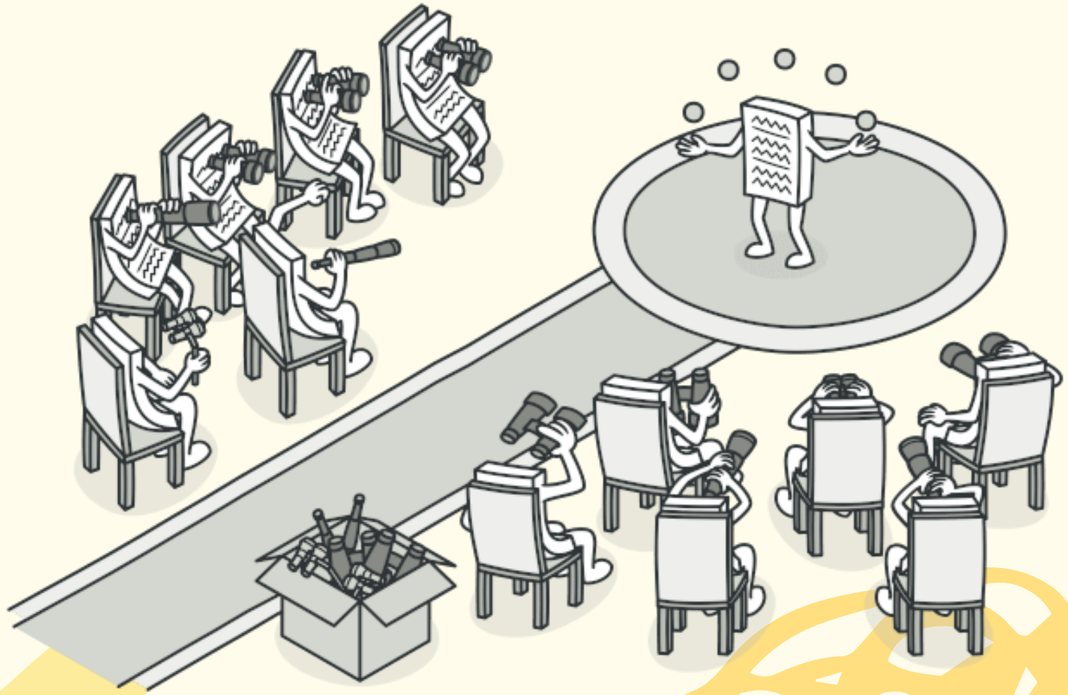
PATRONES DE COMPORTAMIENTO



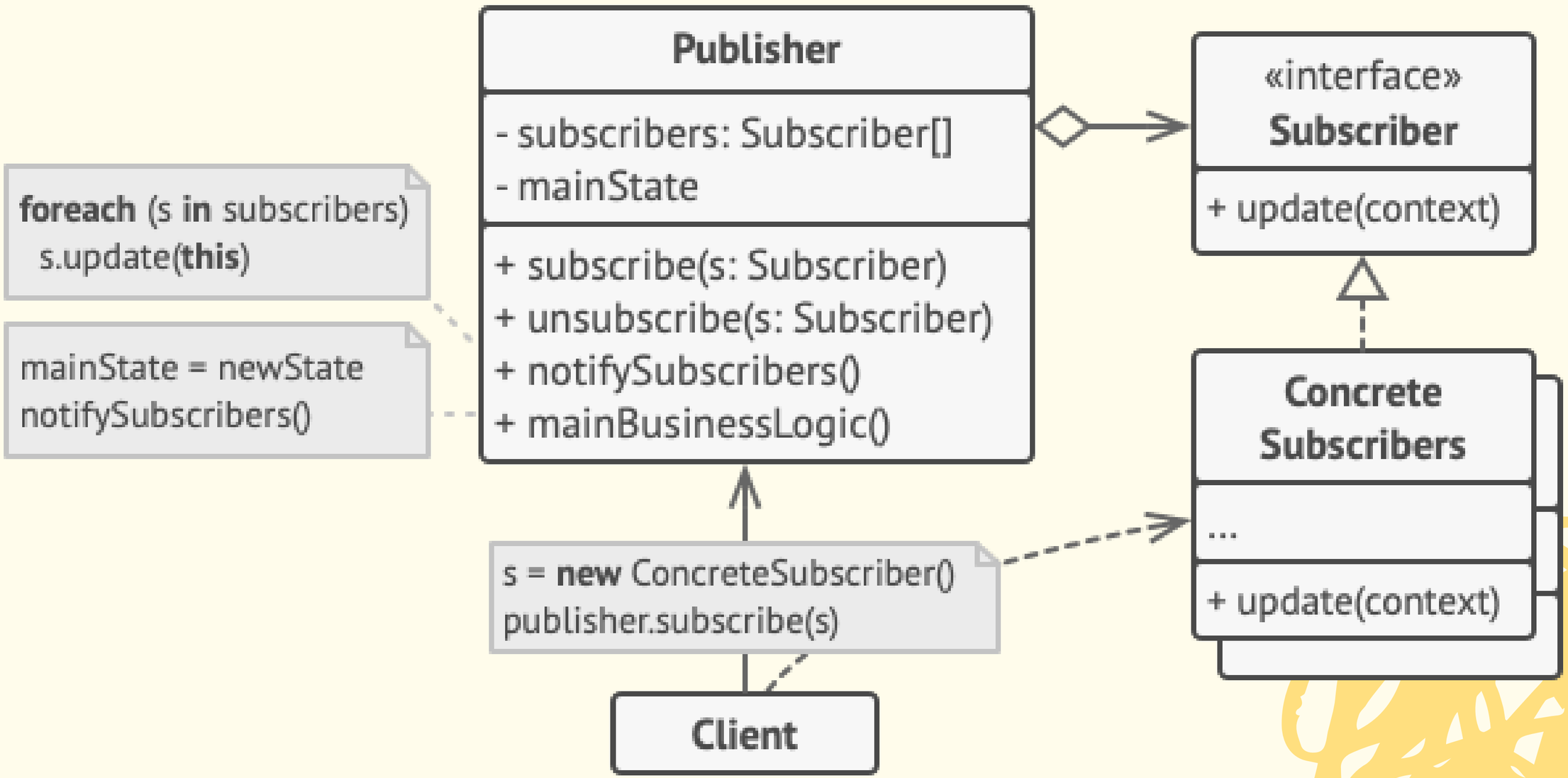


OBSERVER

- Patrón que permite definir un método de suscripción para notificar a otros multiples objetos cuando ocurre un evento en el objeto que está siendo observado.
- Es una relación 1 : N, múltiples objetos pueden observar al mismo.
- Ejemplo: Las notificaciones de youtube al subir un nuevo video.



OBSERVER


Publisher
- subscribers: Subscriber[]
- mainState
+ subscribe(s: Subscriber)
+ unsubscribe(s: Subscriber)
+ notifySubscribers()
+ mainBusinessLogic()
«interface»
Subscriber
+ update(context)
Concrete
Subscribers
...
+ update(context)
Client
foreach (s in subscribers)
s.update(this)
mainState = newState
notifySubscribers()
s = new ConcreteSubscriber()
publisher.subscribe(s)

P6 REC-12-2023-1

Ivan analiza una parte de un sistema de banco encargada de notificar al dueño de la cuenta si existe algun cambio en su balance. Nota que esa parte del sistema está ilustrada en el código siguiente, y que por el momento solo envia correos al dueño de la cuenta cuando corresponde utilizando la siguiente instruccion. Sin embargo, Ivan te comenta que muchos clientes del banco desean también recibir notificaciones a sus celulares mediante SMS. Para realizar este cambio, es necesario modificar el código actual (agregar una linea en 2 metodos de la clase) e Ivan no quiere hacer esto. Ayuda a Ivan a refactorizar está parte del programa aplicando el patron observer, tal que pueda agregar las notificaciones según sea necesario:

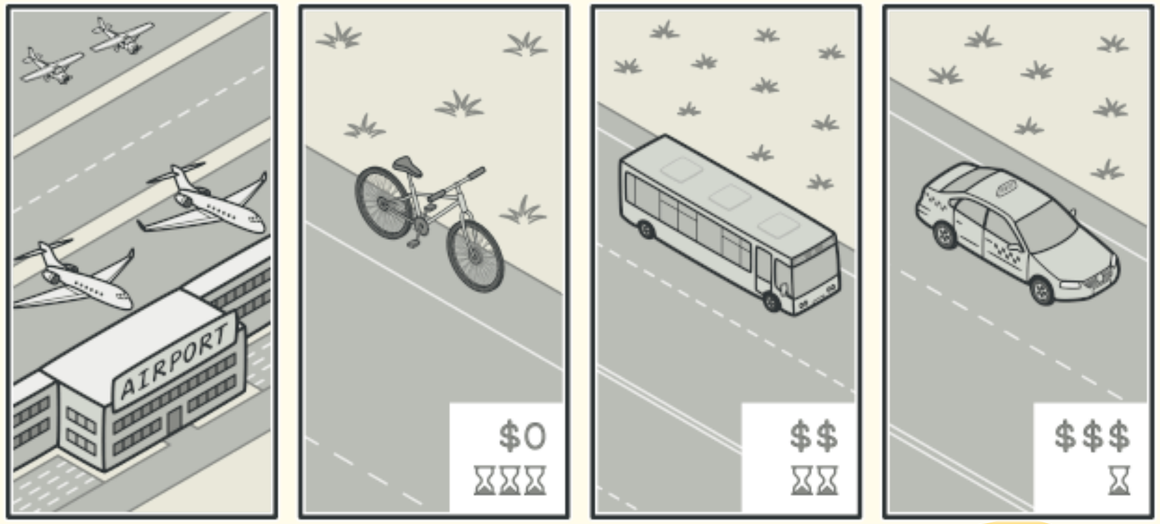
- Notificaciones por email: Cada que el balance cambia, se debe imprimir en consola “Send email to owner: the actual balance is ….”.
- Notificaciones por SMS: Cada que el balance cambia, se debe imprimir en consola “Send SMS to owner: the actual balance is ….”.

```
class Account
      def initialize()
            @balance = 0
      end
      def deposit(amount)
            @balance += amount
            puts "Send email to owner: the actual balance is #{@balance}"
      end
      def withdraw(amount)
            if @balance - amount >= 0
                  @balance -= amount
                  puts "Send email to owner: the actual balance is #{@balance}"
            end
      end
end
```

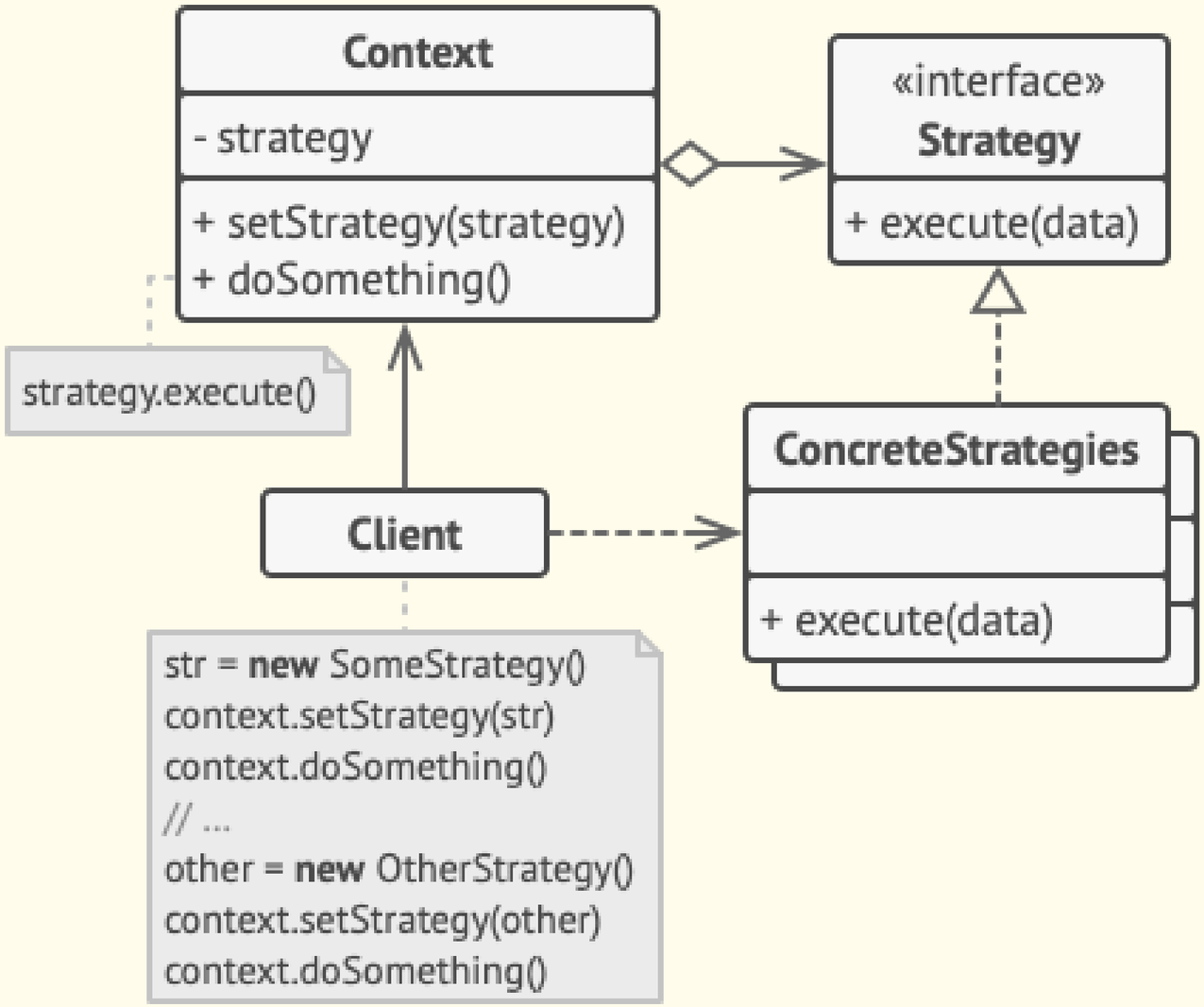
```
class Notification
  def update(balance)
    raise NotImplementedError
  end
end

class SMSNotification
  def update(balance)
    puts "Send SMS to owner: the actual balance is #{balance}"
  end
end

class EmailNotification
  def update(balance)
    puts "Send email to owner: the actual balance is #{balance}"
  end
end
class Account
        def initialize()
                @balance = 0
                @observers = []
        end
        def addObserver(obs)
          @observers.push(obs)
        end
        def notifyAll()
          for obs in @observers
            obs.update(@balance)
          end
        end
        def deposit(amount)
                @balance += amount
                notifyAll()
        end
        def withdraw(amount)
                if @balance - amount >= 0
                        @balance -= amount
                        notifyAll()
                end
        end
end
```

STRATEGY

- Permite definir una familia de algoritmos, ponerlos a cada uno en una clase separada y hacerlos intercambiables, es decir, que cualquiera pueda ser usado por otro objeto.
- Se puede pensar como la estrategia para ejecutar una acción. Viajar caminando, en auto, en bus, en vicicleta o en avión.
- Permite agregar facilmente nuevos algoritmos sin modificar la clase que lo usa.



STRATEGY


Context
- strategy
+ setStrategy(strategy)
+ doSomething()
«interface»
Strategy
+ execute(data)
strategy.execute()
Client
ConcreteStrategies
+ execute(data)
str = new SomeStrategy()
context.setStrategy(str)
context.doSomething()
// ...
other = new OtherStrategy()
context.setStrategy(other)
context.doSomething()

P5 REC-12-2023-1

Modifique este código de un reproductor de música aplicando el patron strategy

```
class Track
    attr_reader :title
    attr_reader :duration
    def initialize(title,duration)
        @title = title
        @duration = duration
        @playing = false
    end
    def playing?
        return @playing
    end
    def play()
        @playing = true
    end
    def stop()
        @playing = false
    end
end
```

```
class MusicPlayer
   def initialize
       @tracks =[]
       @current_index = 0
       @strategy = "sequence"
   end
   def add(track)
       @tracks.push(track)
   end
   def strategy=(name)
       @strategy = name
   end
   def playFirst()
       if @tracks.length > 0
           @tracks[0].play()
       end
   end
   def playNext()
       if @tracks.length > 0
           @tracks[@current_index].stop()
           if @strategy == "sequence"
               @current_index = (@current_index + 1) % @tracks.length
           elsif @strategy == "random"
               @current_index = rand(@tracks.length)
           end
           @tracks[@current_index].play()
       end
   end
   def print()
       @tracks.each do |track|
           if track.playing?
               puts (track.title + ":" + track.duration.to_s).green
           else
               puts (track.title + ":" + track.duration.to_s).blue
           end
       end
   end
end
```

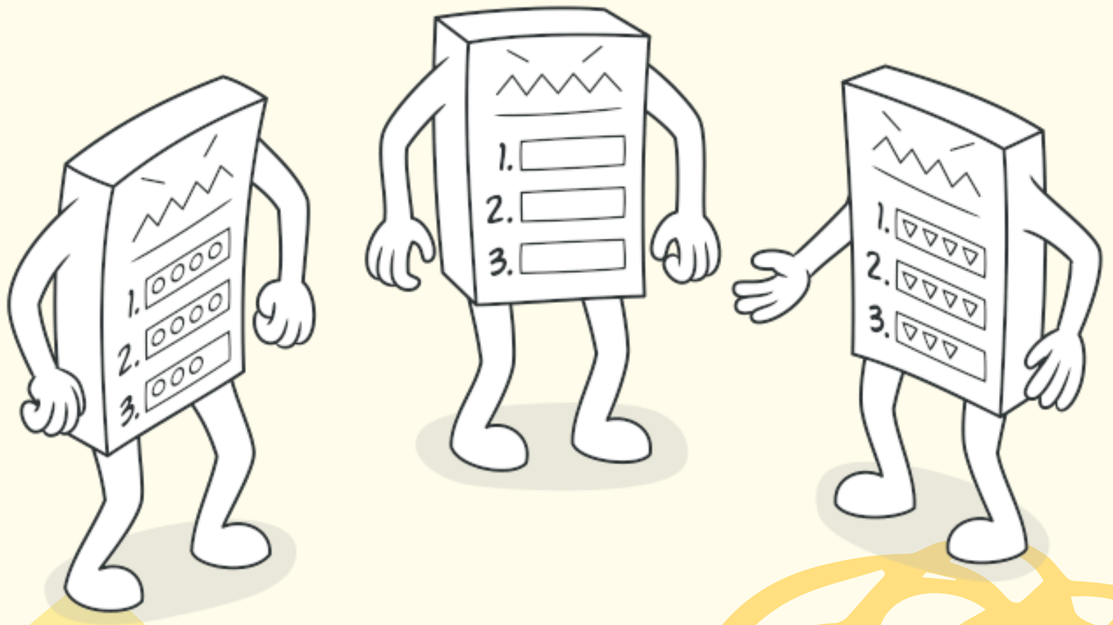
P5 REC-12-2023-1

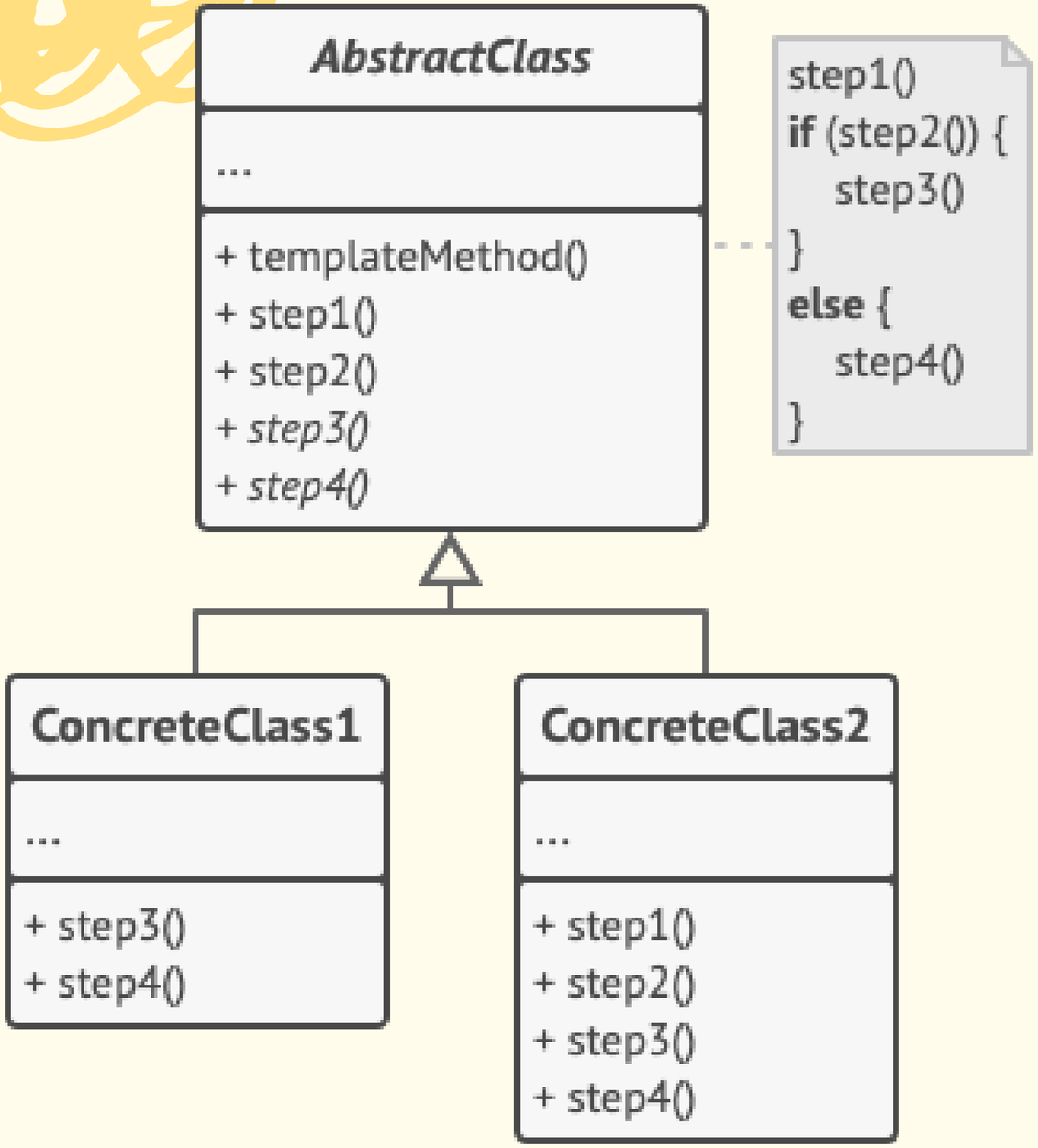
```
class PlayStrategy

    def nextIndex(current_index,len)
        raise NotImplementedError
    end
end
class RandomStrategy < PlayStrategy
    def nextIndex(current_index,len)
        return rand(len)
    end
end
class SequenceStrategy < PlayStrategy
    def nextIndex(current_index,len)
        return (current_index + 1) % len
    end
end
class RepeatStrategy < PlayStrategy
    def nextIndex(current_index,len)
        return current_index
    end
end
```

```
class MusicPlayer
  ...
  def playNext()
    if @tracks.length > 0
      @tracks[@current_index].stop()
      @current_index = @strategy.nextIndex(@current_index,@tracks.length)
      @tracks[@current_index].play()
    end
  end
end
```

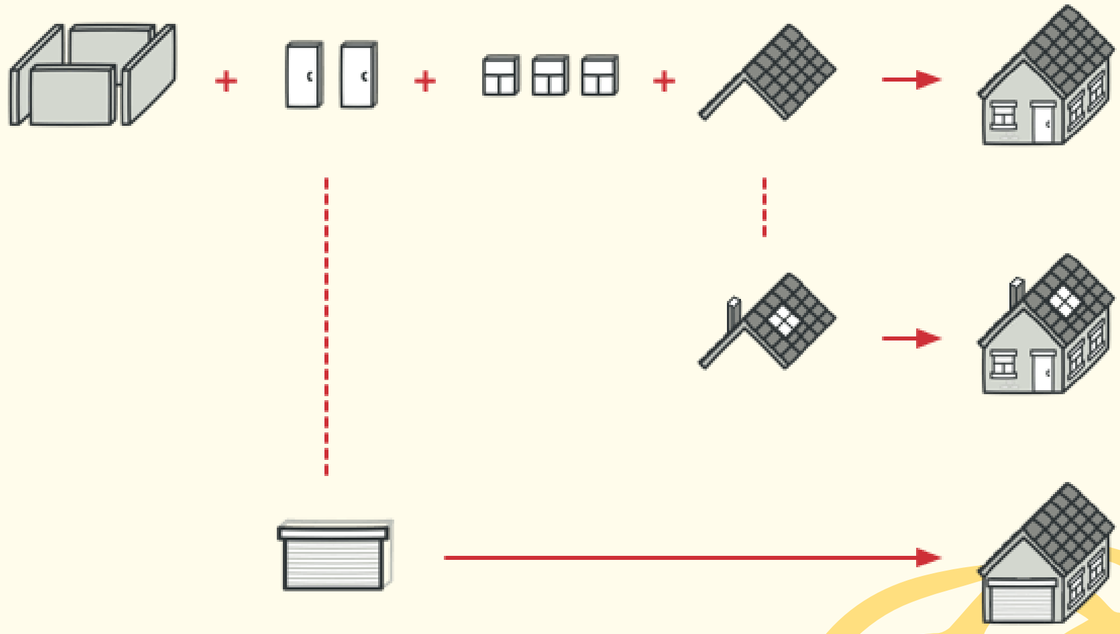
TEMPLATE METHOD

- Permite definir un esqueleto de un algoritmo en una superclase, pero permite sobreescribir pasos especificos sin cambiar la estructura en las clases hijas.
- Se puede pensar tal como funcionan las clases abstractas.
- Permite eliminar codigo duplicado y crear objetos separados que lo reutilicen, pero tengan ciertas diferencias entre si.





TEMPLATE METHOD



P10 12-2023-1

El siguiente código busca simular dos tipos de cobro: cobro en efectivo y cobro con tarjeta. Su tarea es mejorar el código utilizando el patrón template method, de forma tal que no exista código duplicado

```ruby
class CobroEfectivo
    def cobrar(monto)
        puts "Su cuenta es de #{monto.to_s}"
        puts "Mas 10 Bs de comision"
        puts "En total debe #{(monto + 10).to_s}"
        puts "Ingrese el monto a depositar:"
        pagoCliente = gets.chomp.to_i
        if pagoCliente < monto
            puts "Monto insuficiente"
        else
            puts "Su cambio es: #{(pagoCliente - (monto + 10)).to_s}"
            puts "Transacción exitosa"
        end
        puts "Gracias"
    end
end
```

```ruby
class CobroTarjeta
    def cobrar(monto)
        puts "Su cuenta es de #{monto.to_s}"
        puts "Mas 1 Bs de comision"
        puts "En total debe #{(monto + 1).to_s}"
        puts "Ingrese el número de tarjeta:"
        numeroTarjeta = gets.chomp.to_i
        if numeroTarjeta > 1000
            puts "Se ha descontado un total de #{(monto + 1).to_s}"
            puts "De su cuenta ***#{(numeroTarjeta % 1000).to_s}"
            puts "Transacción exitosa"
        else
            puts "Tarjeta no válida, consulte a su banco"
        end
        puts "Gracias"
    end
end
```

P10 12-2023-1

```ruby
class Cobro
  def initialize(comision)
    @comision = comision
  end

  def cobrar(monto)
    puts "Su cuenta es de #{monto}"
    puts "Mas #{@comision} Bs de comision"
    puts "En total debe #{monto + @comision}"
    realizarTransaccion(monto)
    gracias
  end

  def realizarTransaccion(monto)
    raise NotImplementedError
  end

  def gracias
    puts 'Gracias'
  end

  def trasaccionExitosa
    puts 'Transacción exitosa'
  end
end
```

```ruby
class CobroEfectivo < Cobro
  def initialize
    super(10)
  end
  def realizarTransaccion(monto)
    puts 'Ingrese el monto a depositar:'
    pagoCliente = gets.chomp.to_i
    if pagoCliente < monto
      puts 'Monto insuficiente'
    else
      puts "Su cambio es: #{pagoCliente - (monto + 10)}"
      trasaccionExitosa
    end
  end
end
class CobroTarjeta < Cobro
  def initialize
    super(1)
  end
  def realizarTransaccion(monto)
    puts 'Ingrese el número de tarjeta:'
    numeroTarjeta = gets.chomp.to_i
    if numeroTarjeta > 1000
      puts "Se ha descontado un total de #{monto + 1}"
      puts "De su cuenta ***#{numeroTarjeta % 1000}"
      trasaccionExitosa
    else
```

EJERCICIO RESUMEN

Completa la tabla con el nombre del patrón

Patrón	Descripción
	Permite agregar una funcionalidad a un objeto dinámicamente. Permite extender las capacidades de un objeto sin modificar su estructura básica.
	Permite una familia de algortimos, encapsulándolos y haciéndolos intercambiables.
	Permite tratar un grupo de objetos de manera similar a un objeto individual.
	Define el esqueleto de un algoritmo en una clase base, delegando la implementación de ciertos pasos a las clases hijas.
	Permite definir un mecanismo de suscripción para notificar varios objetos sobre un cambio de estado del objeto observado.

Completa la tabla con el nombre del patrón

Patrón	Descripción
Decorator	Permite agregar una funcionalidad a un objeto dinámicamente. Permite extender las capacidades de un objeto sin modificar su estructura básica.
Strategy	Permite una familia de algortimos, encapsulándolos y haciéndolos intercambiables.
Composite	Permite tratar un grupo de objetos de manera similar a un objeto individual.
Template	Define el esqueleto de un algoritmo en una clase base, delegando la implementación de ciertos pasos a las clases hijas.
Observer	Permite definir un mecanismo de suscripción para notificar varios objetos sobre un cambio de estado del objeto observado.