IIC 2143 - Ingeniería de Software

# Patrones de diseño

M. Trinidad Vargas
mtvargas1@uc.cl

# ¿Qué propiedades queremos que tenga nuestro código?

- Bajo acoplamiento
- Alta cohesión
- Sin código duplicado
- Favorezca el cambio y la extensibilidad

# Clase “auspiciada” por Refactoring Guru

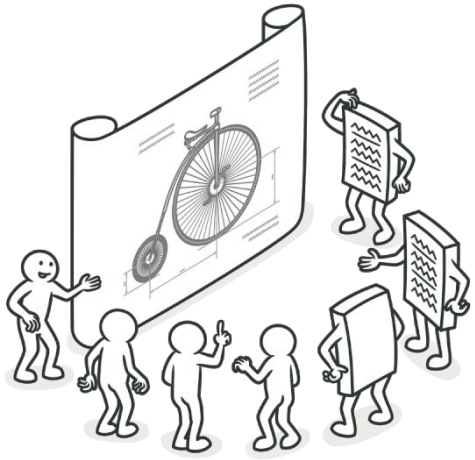
https://refactoring.guru/es/design-patterns/

# Patrones de diseño

Los patrones de diseño son **soluciones habituales a problemas** que ocurren con frecuencia en el diseño de software.

Son como planos prefabricados que se pueden personalizar para resolver un problema de diseño recurrente en tu código.

No es una función de código, si no un concepto general para resolver un problema particular.



# Patrones de diseño

## Ventajas

- Ayudan a estructurar el código siguiendo principios como separación de responsabilidades, bajo acoplamiento y alta cohesión.
- Permiten definir un lenguaje común que puedes utilizar con tu equipo para comunicarse de forma más eficiente.

## Críticas

- Aplicar patrones cuando no se necesitan puede hacer el código más complejo de lo necesario. (Especialmente cuando los acabas de aprender) “Si tienes un martillo, todo parecerá un clavo”

# Clasificación de patrones

**Según su propósito**

- **Comportamiento:** brindan flexibilidad para agregar comportamiento.
- **Creacionales:** incrementan la flexibilidad y reusabilidad al momento de crear objetos complejos..
- **Estructurales:** mantiene la flexibilidad y reusabilidad al momento de componer estructuras de objetos.

# Patrones de diseño

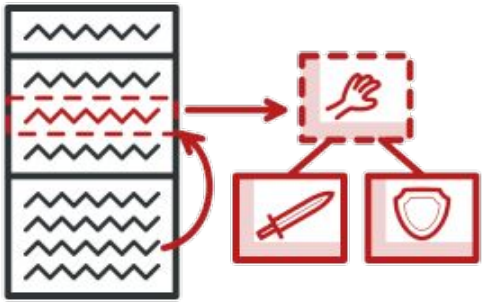
A continuación...

- Propósito: Explicar un problema y luego la solución
- Motivación: Más detalles del problema y la solución del patrón
- Estructura: Diagrama de clase muestra cada parte del patrón y cómo se relacionan

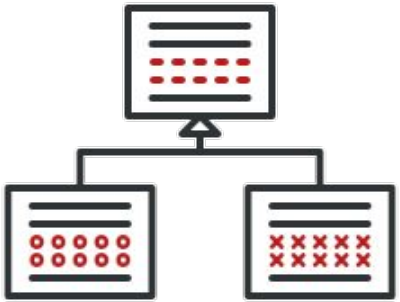
# Patrones de diseño

**Comportamiento**

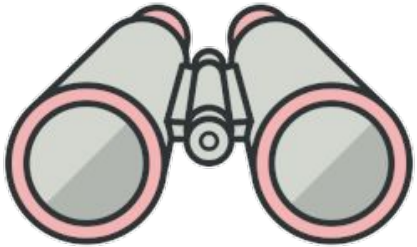
Strategy



Template Method

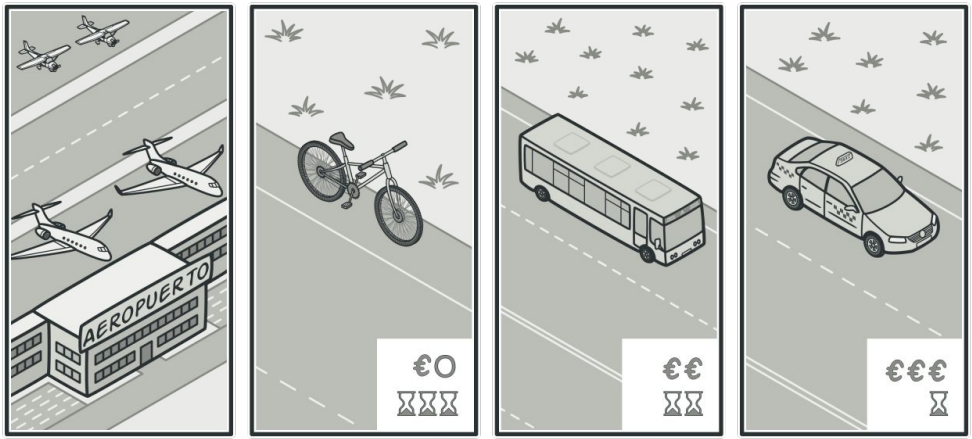


Observer



# Strategy

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



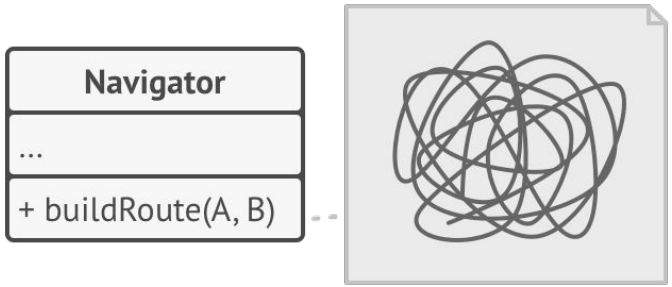

https://refactoring.guru/es/design-patterns/strategy

# Strategy

Caso: Considere una aplicación para viajeros con un mapa que permite orientar a las personas

- V1: El mapa indica rutas sobre carreteras
- V2: Se agrega rutas a pie
- V3: Se permite usar el transporte público
- V4: ...

Problema: La clase Navigator tiene mucha lógica diferente y hacer cambios se ha vuelto insostenible.



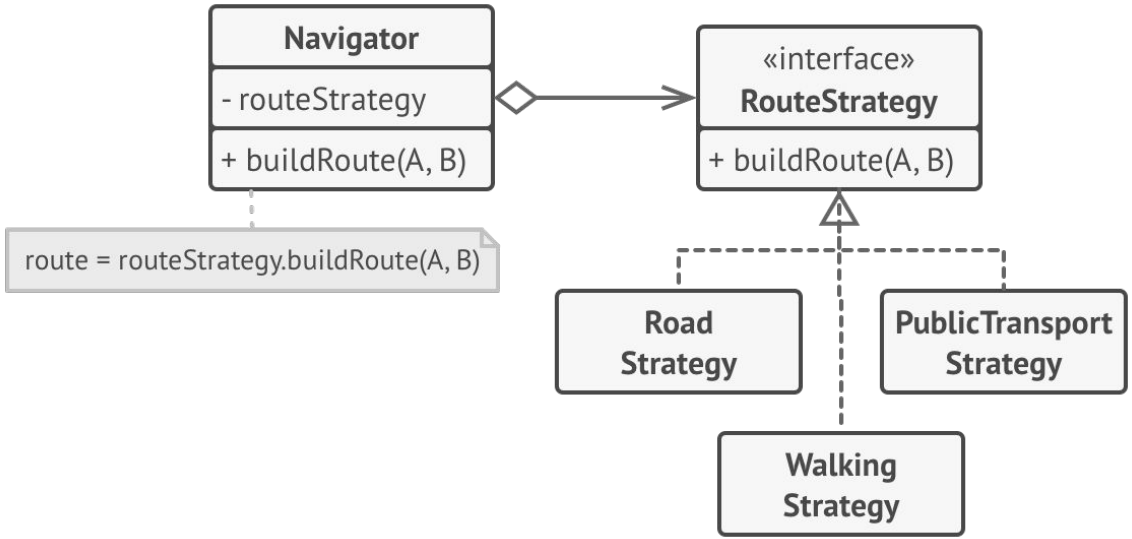
# Strategy

Solución

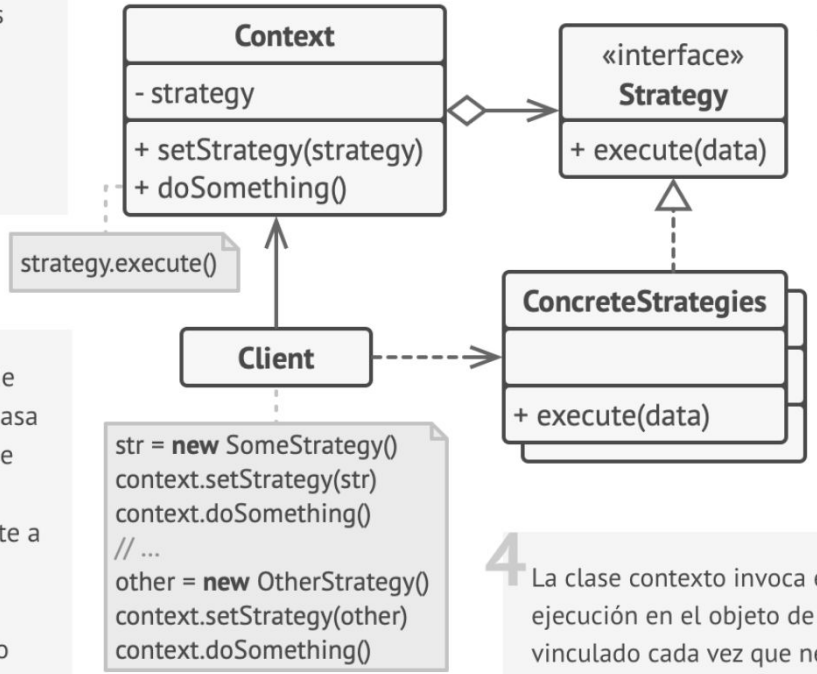
- Tomar la clase que hace algo específico de muchas formas diferentes y extraer todos los algoritmos para colocarlos en clases separadas, llamadas estrategias.
- Clase original llamada contexto tendrá un atributo para almacenar una referencia a la estrategia que se está usando.
- El cliente la pasa la estrategia deseada al contexto, y el contexto siempre llama a un método general que todas las estrategias tienen para ejecutar la acción.

# Strategy

Solución



1 La clase **Contexto** mantiene una referencia a una de las estrategias concretas y se comunica con este objeto únicamente a través de la interfaz estrategia.




2 La interfaz **Estrategia** es común a todas las estrategias concretas. Declara un método que la clase contexto utiliza para ejecutar una estrategia.

3 Las **Estrategias Concretas** implementan distintas variaciones de un algoritmo que la clase contexto utiliza.

4 La clase contexto invoca el método de ejecución en el objeto de estrategia vinculado cada vez que necesita ejecutar el algoritmo. La clase contexto no sabe con qué tipo de estrategia funciona o cómo se ejecuta el algoritmo.

5 El **Cliente** crea un objeto de estrategia específico y lo pasa a la clase contexto. La clase contexto expone un modificador *set* que permite a los clientes sustituir la estrategia asociada al contexto durante el tiempo de ejecución.

# Ejemplo clase anterior

```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")
```

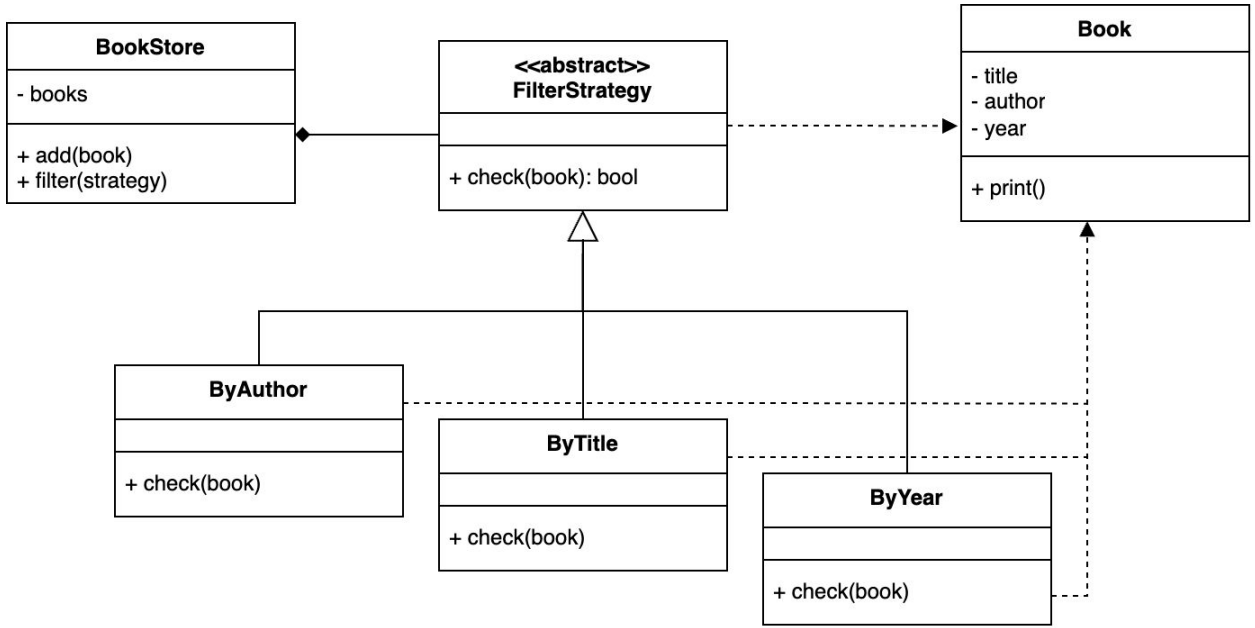
# Ejemplo clase anterior

```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 clase anterior



# Strategy

## ¿Cuándo aplicar strategy?

- Cuando tengas muchas clases similares que sólo se diferencien en la forma en que ejecutan cierto comportamiento.
- Cuando tu clase tenga un enorme operador condicional que cambia entre distintas variantes del mismo algoritmo.
- Para aislar la lógica de negocio de una clase, de los detalles de implementación de algoritmos que pueden no ser tan importantes en el contexto de esa lógica.

# Strategy

## Pros

Puedes intercambiar algoritmos usados dentro de un objeto durante el tiempo de ejecución.

Puedes aislar los detalles de implementación de un algoritmo del código que lo utiliza.

Puedes sustituir la herencia por composición.

Principio de abierto/cerrado. Puedes introducir nuevas estrategias sin tener que cambiar el contexto.

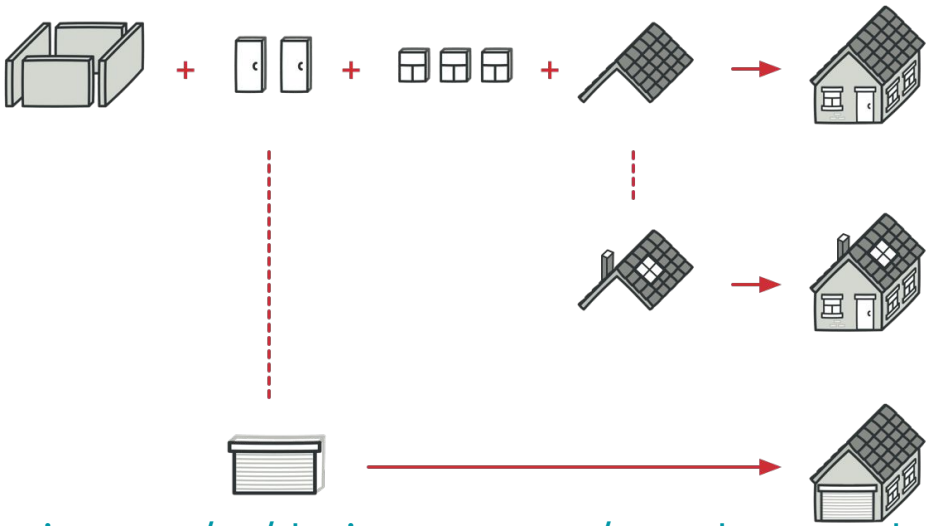
## Contras

Si sólo tienes un par de algoritmos que raramente cambian, no hay una razón real para complicar el programa en exceso con nuevas clases e interfaces que vengan con el patrón.

Los clientes deben conocer las diferencias entre estrategias para poder seleccionar la adecuada.

# Template Method

Es un patrón que define el esqueleto de un algoritmo en la superclase pero permite que las subclases sobre-escriban pasos del algoritmo sin cambiar su estructura.



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

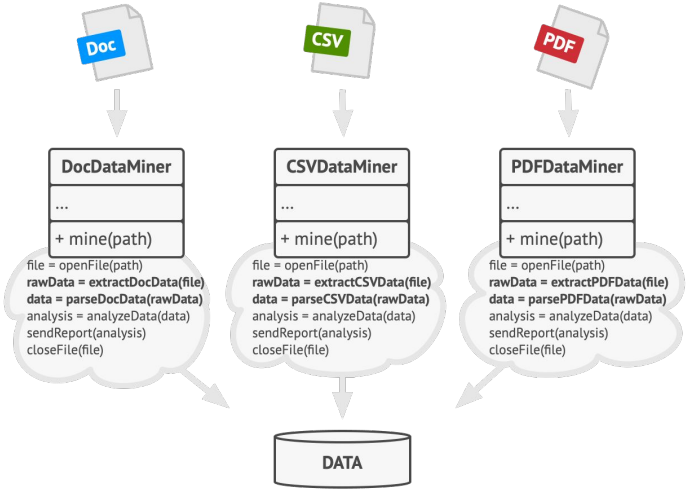
# Template Method

Caso: Considere una aplicación de minería de datos para analizar documentos

Los usuarios suben documentos en varios formatos (PDF, Doc, CSV) y la app intenta extraer información relevante en un formato uniforme.

- V1: Solo archivos Doc.
- V2: Archivos CSV
- V3: Archivos PDF
- V4: ...

Problema: Tenemos mucho código duplicado entre clases



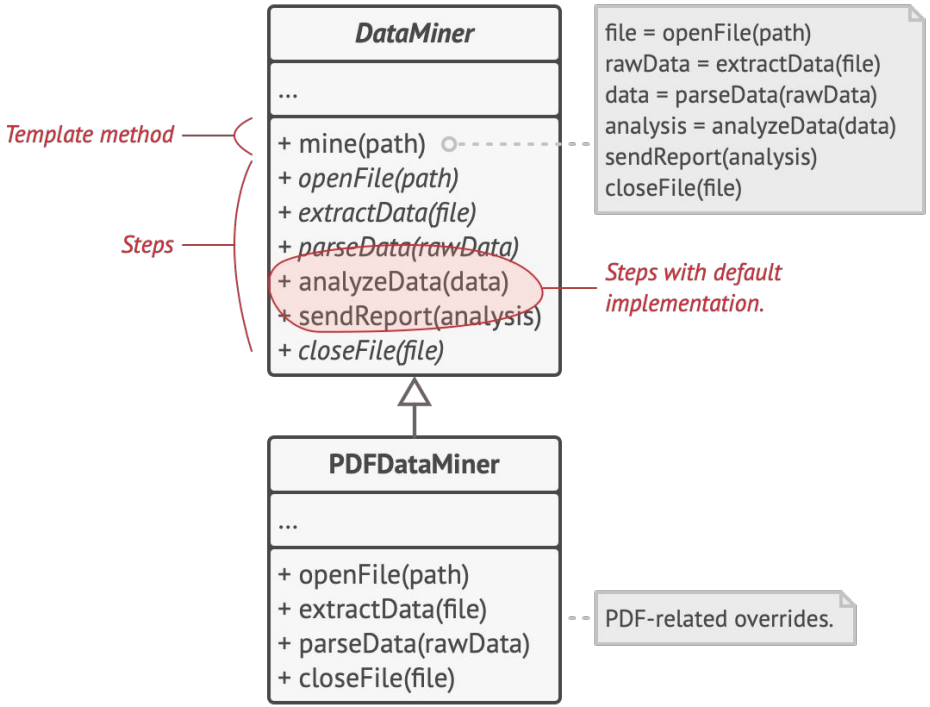
# Template Method

Solución

- Dividir un algoritmo en una serie de pasos, y que cada paso sea un método. Luego, la serie de llamadas a los métodos va en un método plantilla.
- Los pasos pueden ser abstractos o tener una implementación por defecto.
- El cliente debe crear su propia subclase e implementar los pasos abstractos y sobreescribir los opcionales si es necesario. (NUNCA sobreescribir el método plantilla).

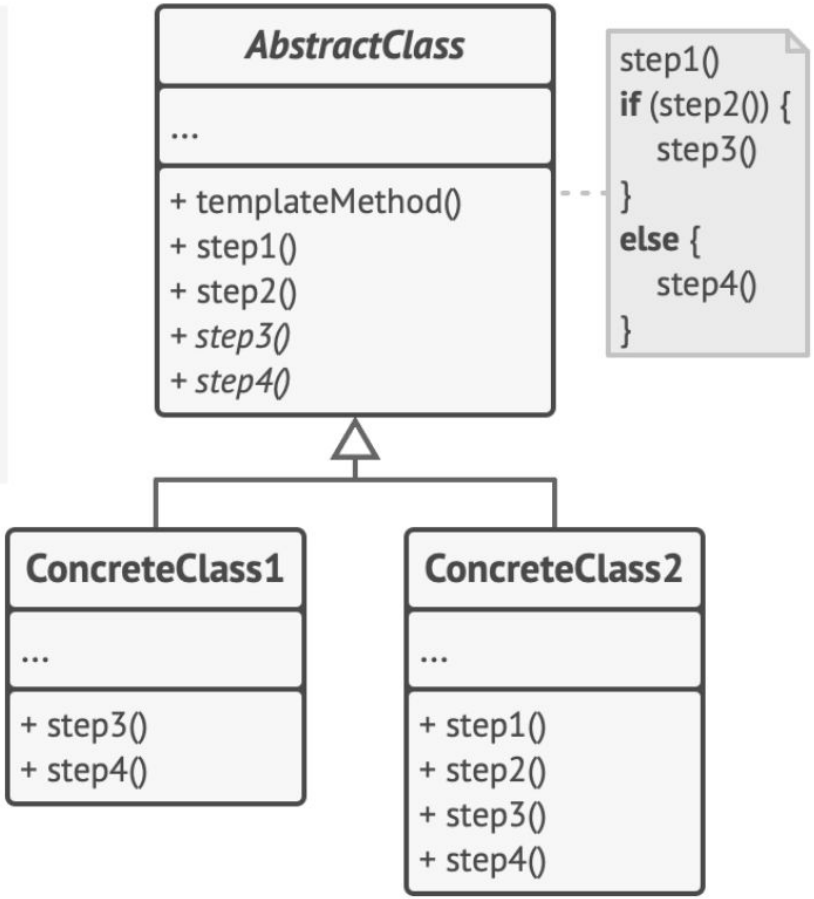
# Template Method

Solución



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.



# Ejemplo clase anterior

```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
```

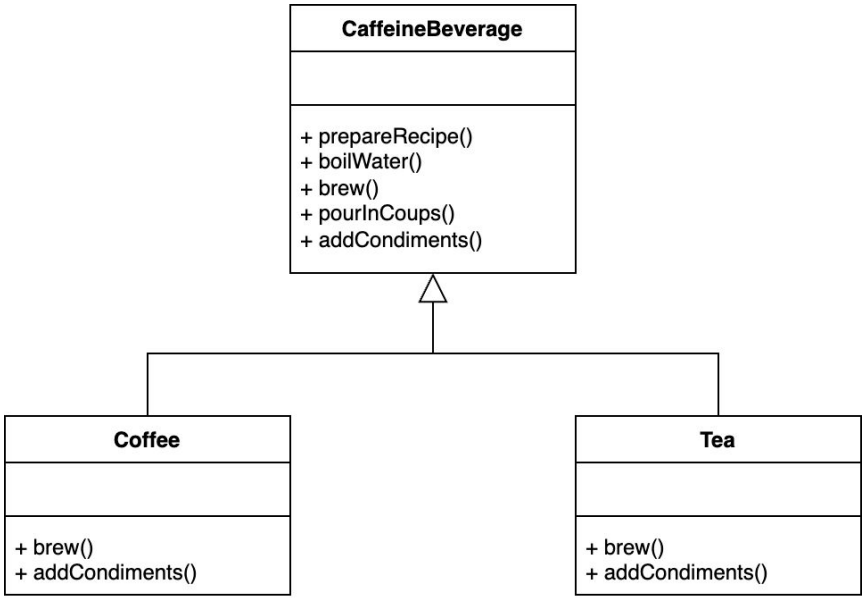
# Ejemplo Beverage

```ruby
class 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 clase anterior



# Template Method

## ¿Cuándo aplicar template method?

- Cuando quieras permitir a tus clientes que extiendan únicamente pasos particulares de un algoritmo, pero no todo el algoritmo o su estructura.
- Cuando tengas muchas clases que contengan algoritmos casi idénticos, pero con algunas diferencias mínimas. Como resultado, puede que tengas que modificar todas las clases cuando el algoritmo cambie.

# Template Method

## Pros

Puedes permitir a los clientes que sobrescriban tan solo ciertas partes de un algoritmo grande, para que les afecten menos los cambios que tienen lugar en otras partes del algoritmo.

Puedes colocar el código duplicado dentro de una superclase.

## Contras

Algunos clientes pueden verse limitados por el esqueleto proporcionado de un algoritmo.

Los métodos plantilla tienden a ser más difíciles de mantener cuantos más pasos tengan.

# 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.


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

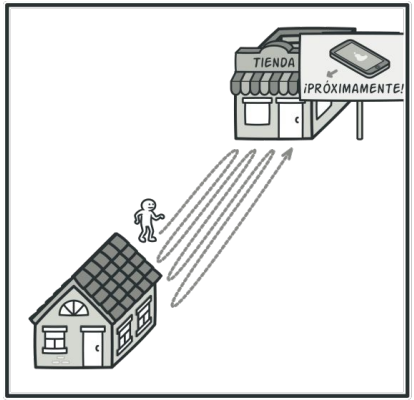
# Observer

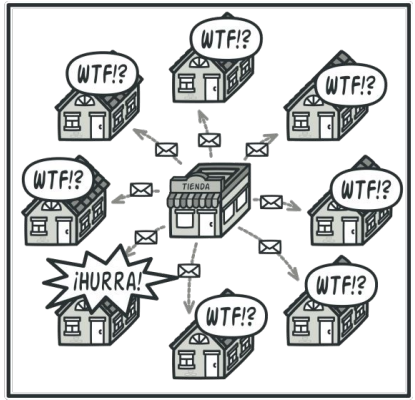
Caso: Tenemos una Tienda con clientes. El cliente está muy interesado en una marca particular de producto, y llegará pronto a la tienda.

- El cliente puede visitar la tienda cada día para comprobar la disponibilidad.
- La tienda podría enviar cientos de correos a todos los clientes cada vez que haya un producto disponible.

Problema:

- El cliente pierde el tiempo visitando la tienda cada día.
- La tienda desperdicia recursos notificando a los clientes equivocados (spam).



# Observer

Solución

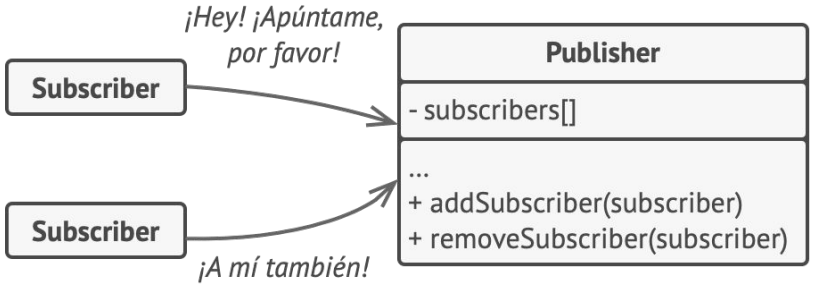
El objeto que tiene un estado interesante para otros objetos se llama sujeto, notificador o publicador.

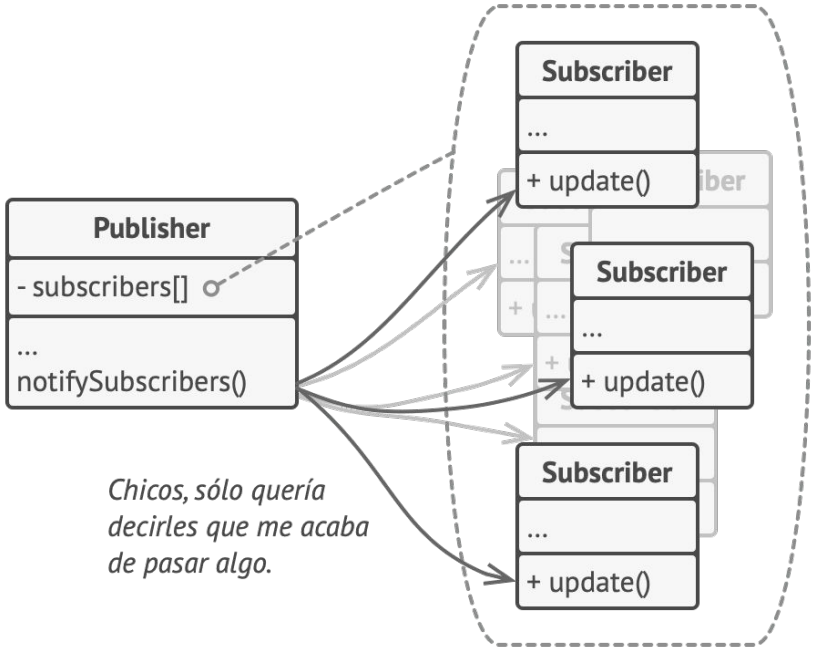
Los objetos que quieren conocer los cambios en el estado del notificador se llaman suscriptores.

El patrón Observer sugiere añadir un mecanismo de suscripción a la clase notificadora para que los objetos individuales puedan suscribirse o cancelar sus suscripción a un flujo de eventos de la clase notificadora.

# Observer

## Solución



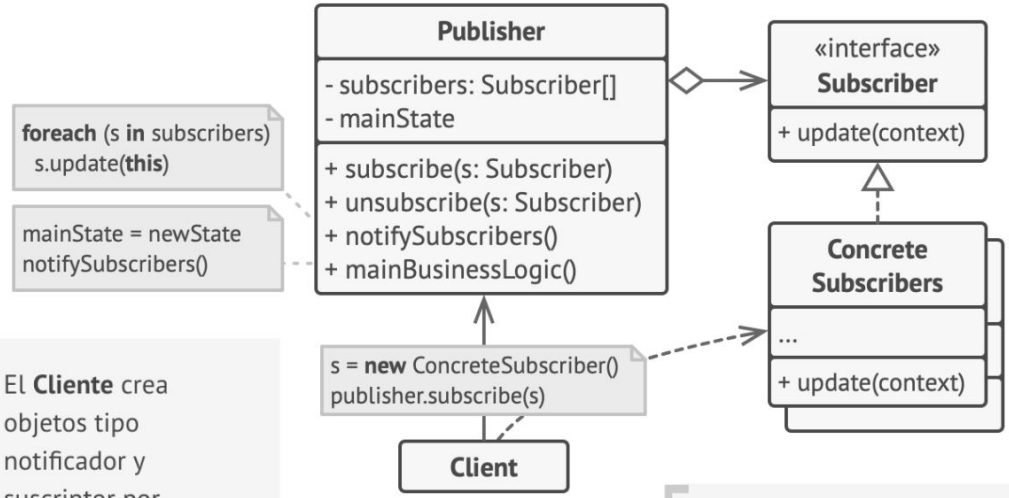




*El notificador notifica a los suscriptores invocando el método de notificación específico de sus objetos.*

1 El **Notificador** envía eventos de interés a otros objetos. Esos eventos ocurren cuando el notificador cambia su estado o ejecuta algunos comportamientos. Los notificadores contienen una infraestructura de suscripción que permite a nuevos y antiguos suscriptores abandonar la lista.

2 Cuando sucede un nuevo evento, el notificador recorre la lista de suscripción e invoca el método de notificación declarado en la interfaz suscriptora en cada objeto suscriptor.

3 La interfaz **Suscriptora** declara la interfaz de notificación. En la mayoría de los casos, consiste en un único método `actualizar`. El método puede tener varios parámetros que permitan al notificador pasar algunos detalles del evento junto a la actualización.




4 Los **Suscriptores Concretos** realizan algunas acciones en respuesta a las notificaciones emitidas por el notificador. Todas estas clases deben implementar la misma interfaz de forma que el notificador no esté acoplado a clases concretas.

5 Normalmente, los suscriptores necesitan cierta información contextual para manejar correctamente la actualización. Por este motivo, a menudo los notificadores pasan cierta información de contexto como argumentos del método de notificación. El notificador puede pasarse a sí mismo como argumento, dejando que los suscriptores extraigan la información necesaria directamente.

6 El **Cliente** crea objetos tipo notificador y suscriptor por separado y después registra a los suscriptores para las actualizaciones del notificador.

# Observer

## ¿Cuándo aplicar observer?

- Cuando los cambios en el estado de un objeto puedan necesitar cambiar otros objetos y el grupo de objetos sea desconocido de antemano o cambie dinámicamente.
- Cuando algunos objetos de tu aplicación deban observar a otros, pero sólo durante un tiempo limitado o en casos específicos.

# Observer

**Pros**

Principio de abierto/cerrado. Puedes introducir nuevas clases suscriptoras sin tener que cambiar el código de la notificadora (y viceversa si hay una interfaz notificadora).

Puedes establecer relaciones entre objetos durante el tiempo de ejecución.

**Contras**

Los suscriptores son notificados en un orden aleatorio.

# Ejercicios adicionales

1. En tu proyecto ¿qué parte del código cambia frecuentemente y se beneficiaría de ser intercambiable mediante estrategias?
2. En el patrón Template-Method ¿por qué se usa herencia y no composición?
3. Escribe en Ruby una implementación del patrón Observer
4. ¿Qué aplicaciones de tu celular usan el patrón Observer? ¿Por qué?
5. ¿En qué contexto conviene notificar a todos los miembros en vez de tener una lista de suscriptores?

# Material Adicional

Existen muchos patrones de comportamiento

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