

# Ingeniería de Software

## 15 - Patrones de Diseño
## Comportamiento

IIC2143-3

**Josefa España**

jpespana@uc.cl

# Clase “patrocinada” por Refactoring Guru



# Patrones de Diseño

## Definición

- Soluciones habituales a problemas que ocurren con frecuencia al diseñar software.
- Planos pre-fabricados que se pueden personalizar para resolver un problema de diseño recurrente.
- No es una función de código, si no un concepto general para resolver un problema particular.

# Patrones de Diseño

## Estructura

- **Propósito**: El patrón explica brevemente el problema y luego la solución.
- **Motivación**: Explica más detalles del problema y la solución que brinda el patrón.
- **Estructura**: La estructura de clases muestra cada parte del patrón y cómo se relacionan.

# Patrones de Diseño

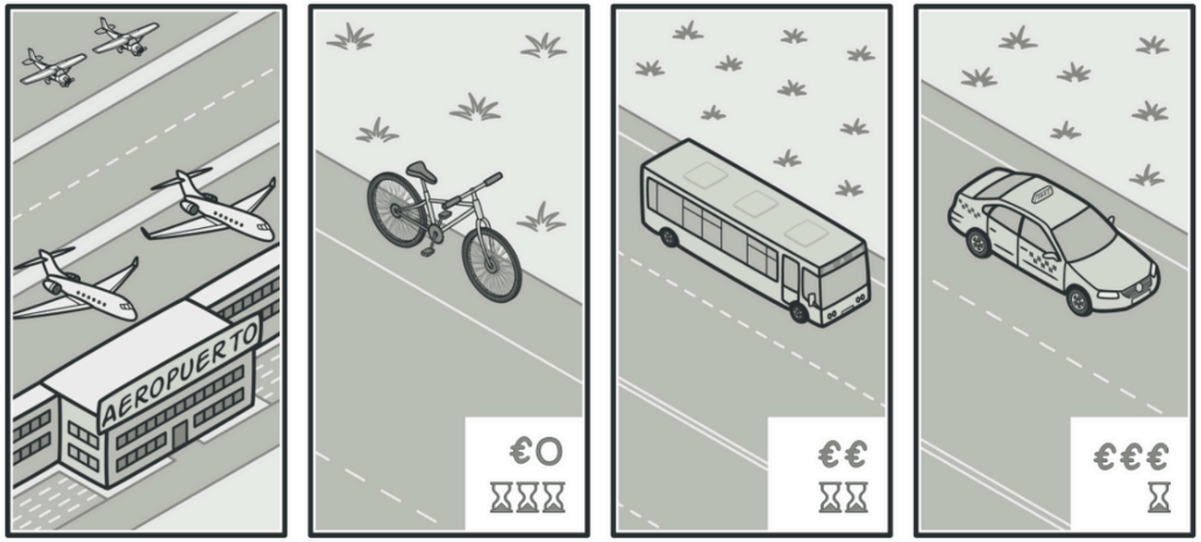
## Patrones de Comportamiento

- Strategy
- Template Method
- Observer

# Strategy

## Propósito

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.



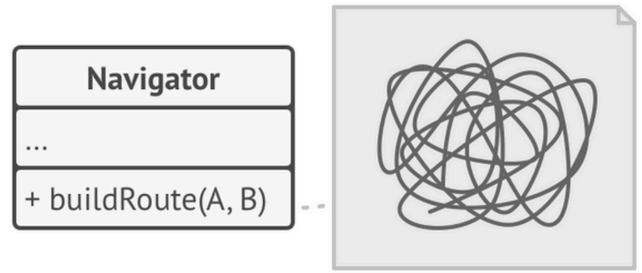

*Varias estrategias para llegar al aeropuerto.*

# Strategy

## Motivación - Problema

- Aplicación de Navegación para viajeros ocasionales, con un mapa que permite orientar a las personas.
- Versión 1: En el mapa se pueden automatizar las rutas sobre carretera.
- Versión 2: Rutas a pie.
- Versión 3: Rutas en transporte público.
- Versión 4: .....
- ....

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




*El código del navegador se saturó.*

# Strategy

## Motivación - 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

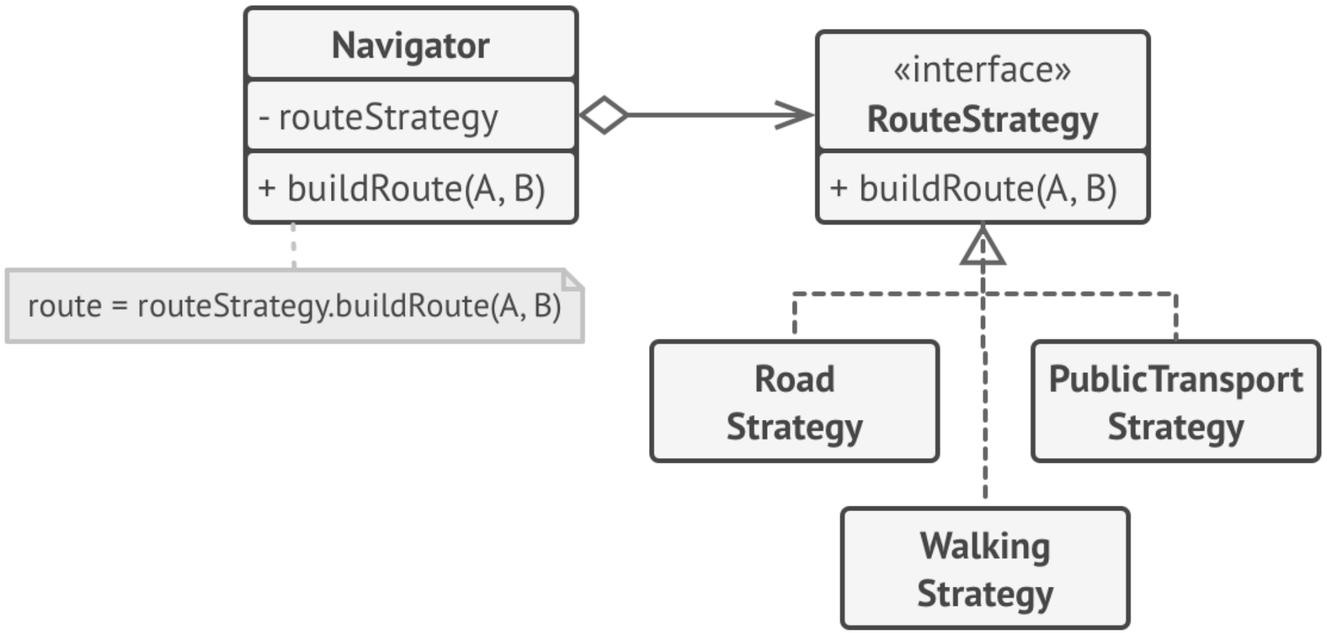
## ⚖ Pros y contras

- ✓ 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.

- ✗ 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.
- ✗ Muchos lenguajes de programación modernos tienen un soporte de tipo funcional que te permite implementar distintas versiones de un algoritmo dentro de un grupo de funciones anónimas. Entonces puedes utilizar estas funciones exactamente como habrías utilizado los objetos de estrategia, pero sin saturar tu código con clases e interfaces adicionales.

# Strategy

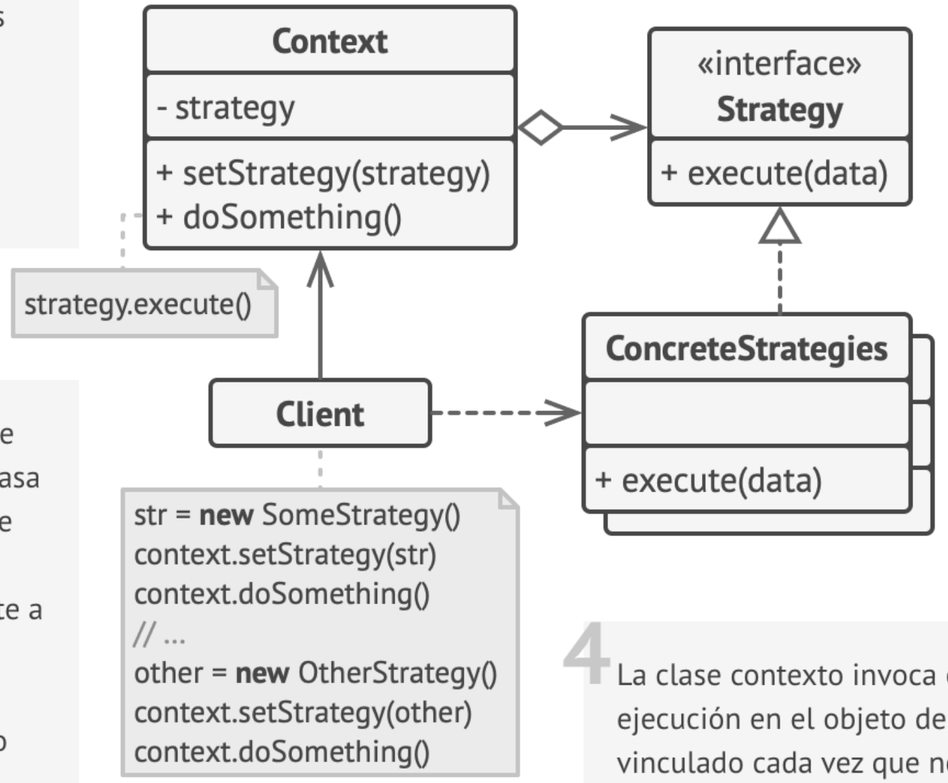
## Motivación - Solución




*Estrategias de planificación de rutas.*

# Strategy

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.



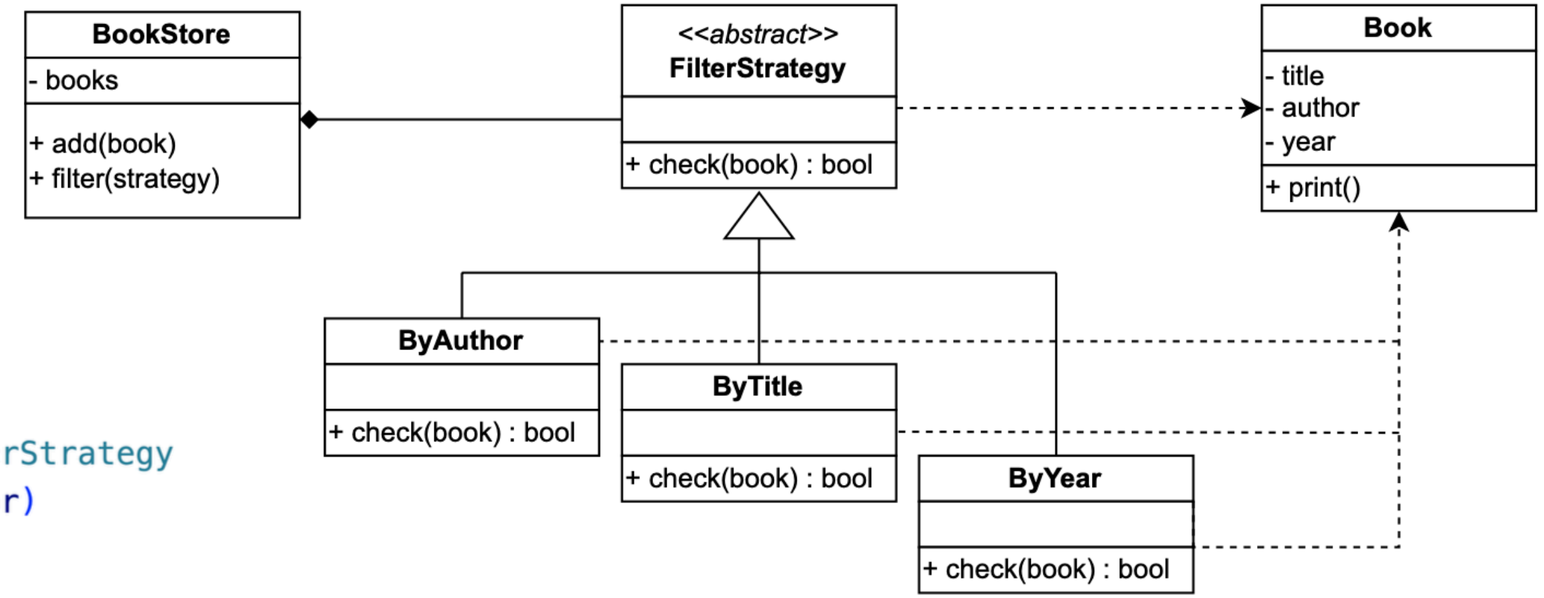

2 La interfaz **Estrategia** es común a todas las estrategias concretas. Declara un método que la clase contexto utiliza para ejecutar una estrategia.

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



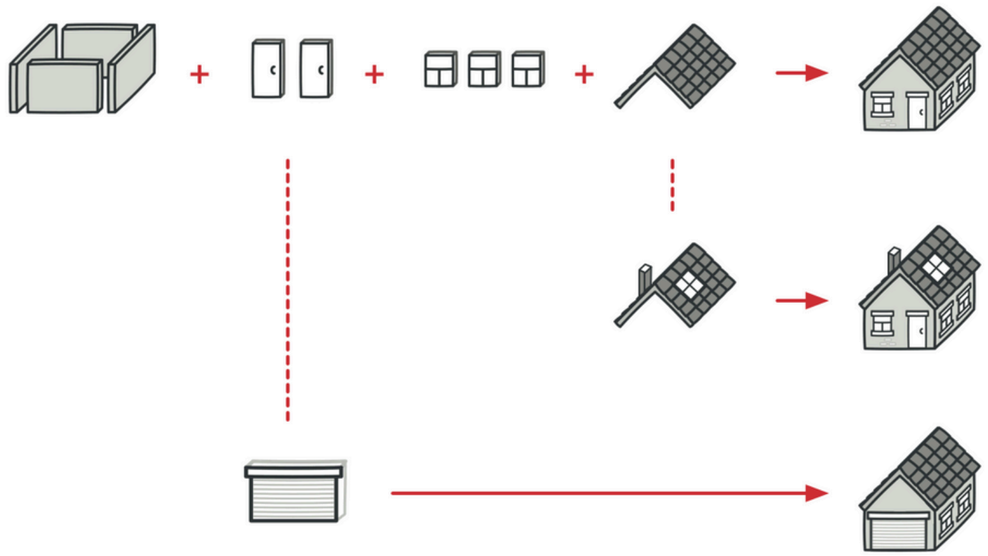

```ruby
class ByAuthor < FilterStrategy
  def initialize(author)
    @author = author
  end

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

# Template Method

## Propósito

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



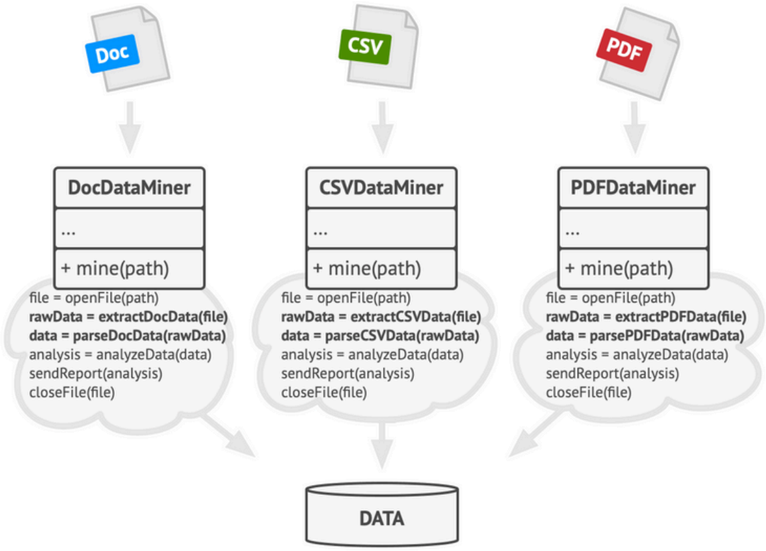
*Un plan arquitectónico típico puede alterarse ligeramente para que encaje mejor con las necesidades del cliente.*

# Template Method

## Motivación - Problema

- Aplicación de minería de datos para analizar documentos corporativos.
- Los usuarios suben documentos en varios formatos (PDF, Doc, CSV) y la app intenta extraer información relevante en un formato uniforme.
- Versión 1: Solo archivos DOC.
- Versión 2: Archivos CSV.
- Versión 3: Archivos PDF.
- Versión 4: ....
- ....

**Problema**
Tenemos mucho código duplicado entre las clases.




*Las clases de minería de datos contenían mucho código duplicado.*

# Template Method

## Motivación - 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).

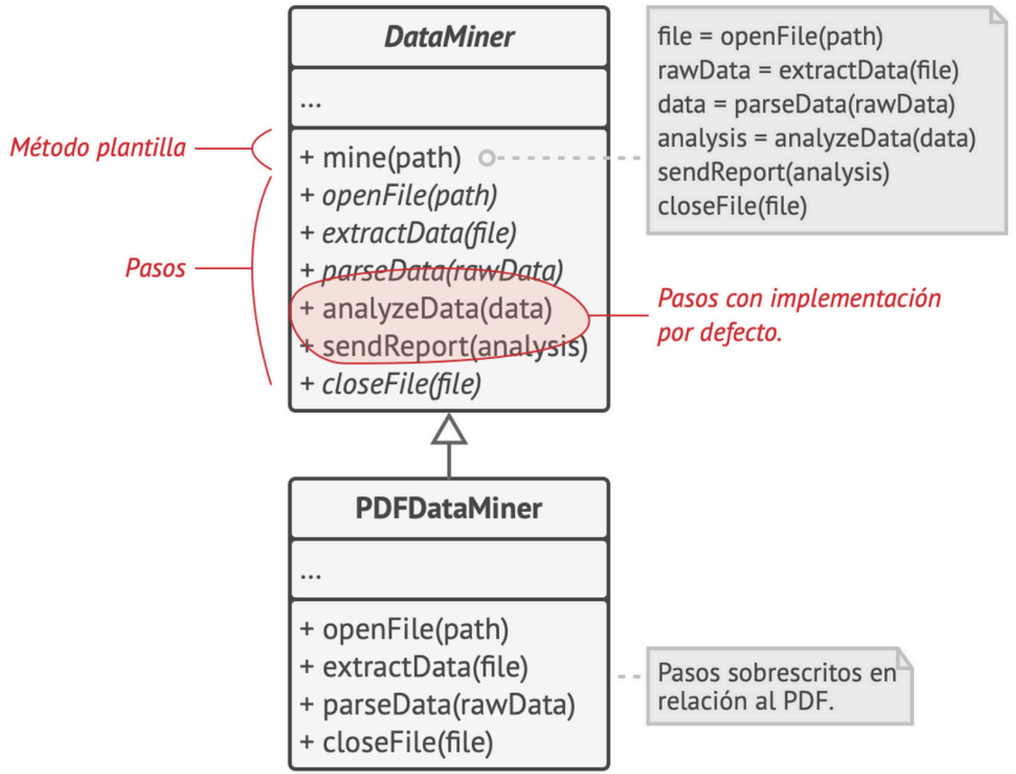
# Template Method

## ⚖ Pros y contras

- ✓ 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.
- ✗ Algunos clientes pueden verse limitados por el esqueleto proporcionado de un algoritmo.
- ✗ Puede que violes el *principio de sustitución de Liskov* suprimiendo una implementación por defecto de un paso a través de una subclase.
- ✗ Los métodos plantilla tienden a ser más difíciles de mantener cuantos más pasos tengan.

# Template Method

## Motivación - Solución

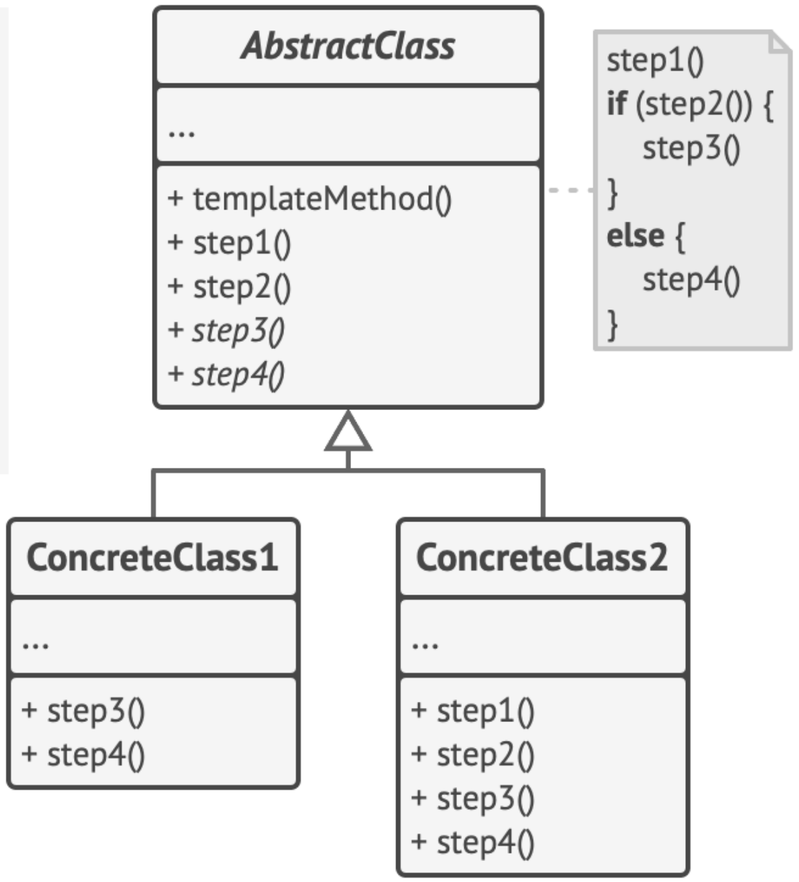



*El método plantilla divide el algoritmo en pasos, permitiendo a las subclases sobrescribir estos pasos pero no el método en sí.*

# Template Method

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.

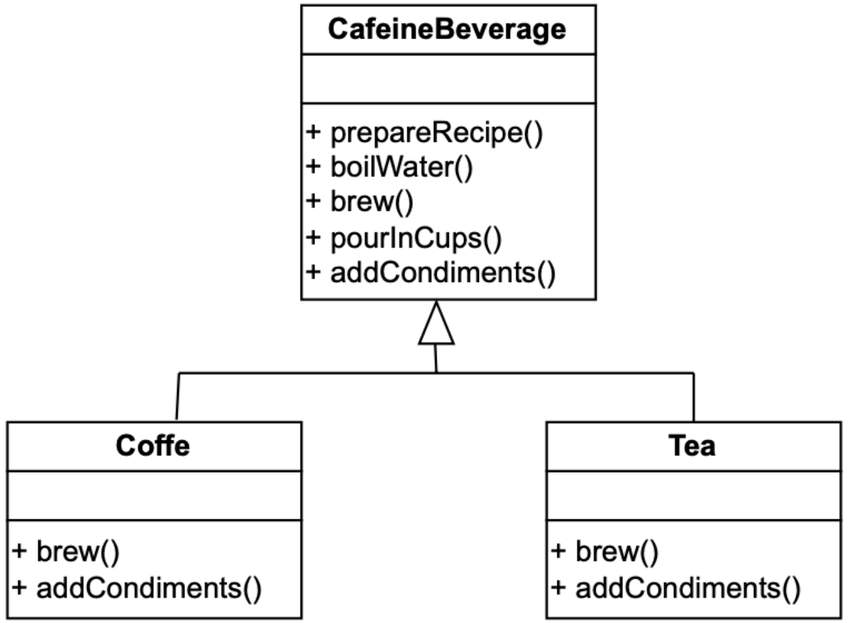
2 Las **Clases Concretas** pueden sobrescribir todos los pasos, pero no el propio método plantilla.



# Ejemplo clase anterior

```ruby
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



# Observer

## Propósito

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.

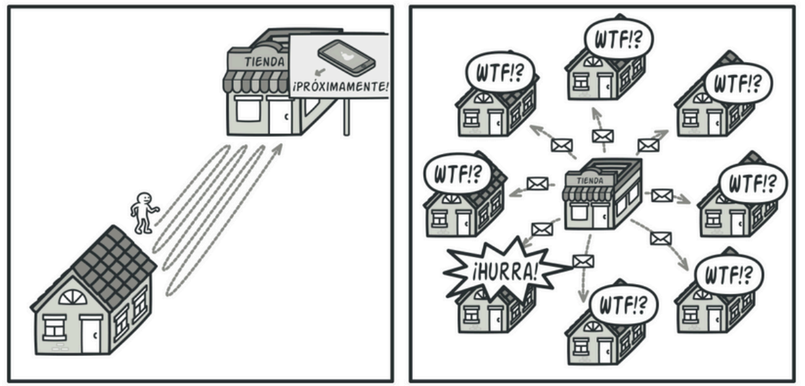



*Suscripciones a revistas y periódicos.*

# Observer

## Motivación - Problema

- 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, pero mientras el producto está en camino, la mayoría de los viajes será en vano.
  - La tienda podría enviar cientos de correos a todos los clientes cada vez que haya un producto disponible. Algunos serían felices, otros tendrían mucho spam.




*Visita a la tienda vs. envío de spam*

# Observer

## Motivación - 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.

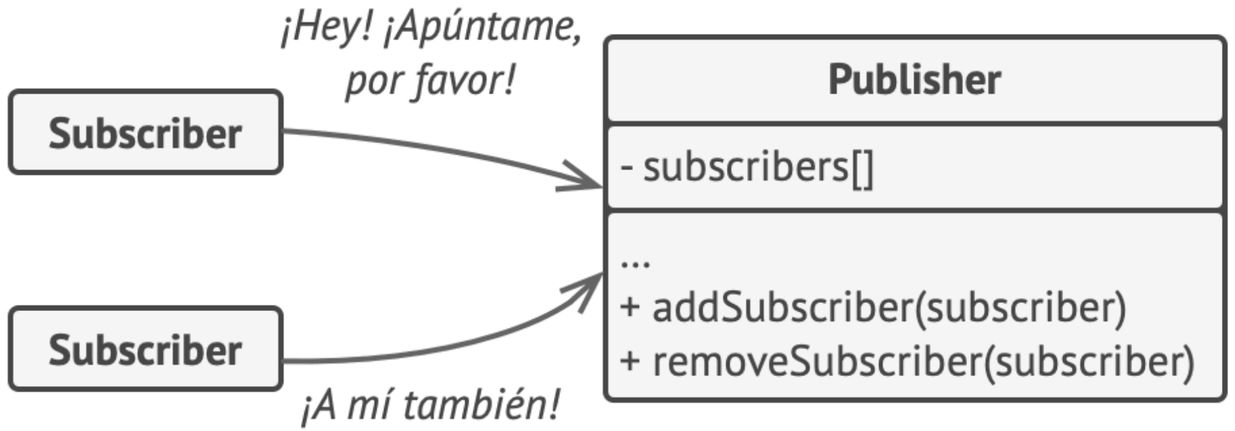
# Observer

## ⚖ Pros y contras

- ✓ *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.
- ✗ Los suscriptores son notificados en un orden aleatorio.

# Observer

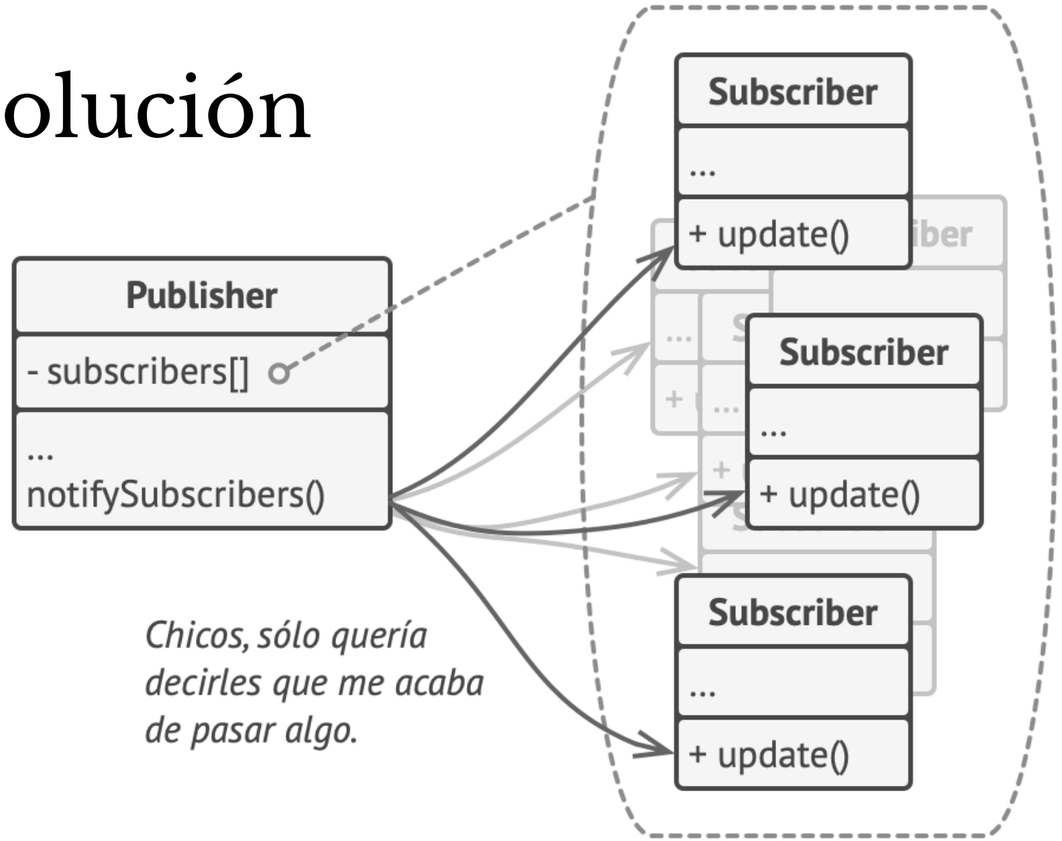
## Motivación - Solución




*Un mecanismo de suscripción permite a los objetos individuales suscribirse a notificaciones de eventos.*

# Observer

## Motivación - Solución

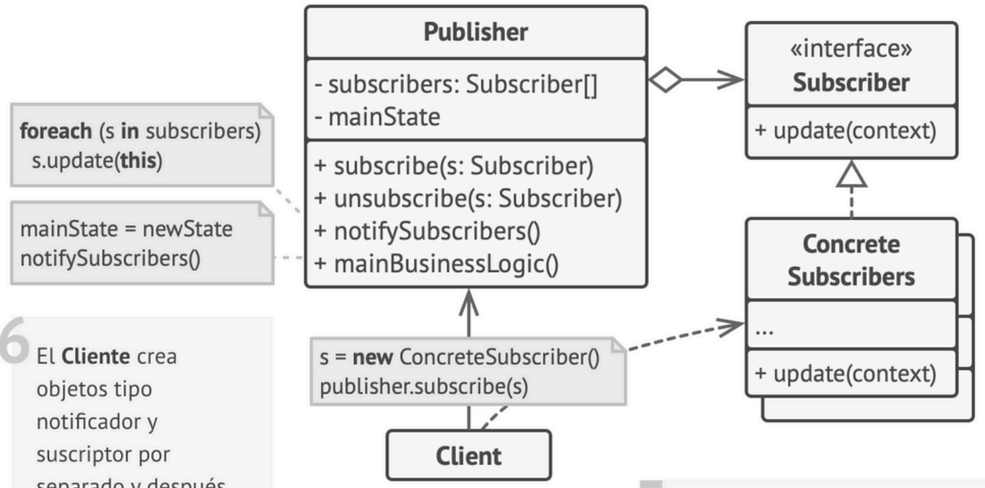



*El notificador notifica a los suscriptores invocando el método de notificación específico de sus objetos.*

# Observer

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.

# Bibliografía

## Enlaces útiles

- Introducción Patrones: https://refactoring.guru/es/design-patterns/what-is-pattern
- Patrón strategy: https://refactoring.guru/es/design-patterns/strategy
- Template Method: https://refactoring.guru/es/design-patterns/template-method
- Observer: https://refactoring.guru/es/design-patterns/observer

# Ejercicio con décimas

A partir del código mostrado en la siguiente diapositiva:

1. Indique qué patrón podría ser utilizado para mejorar el diseño.
2. Proponer una estructura de clases para el código (como los vistos en motivación - solución)
3. Indicar las ventajas de este nuevo diseño.
4. **Bonus +1 décima**: Hacer una implementación en Ruby del diseño (ver ejemplos aquí)

**Entrega Individual:**

- **Lunes 13 de Mayo a las 22:00 hrs.**
- Entrega vía Canvas en formato PDF.
- 1 décima: Indicar los 3 puntos en el documento.
- 2 décimas: Indicar los 4 puntos en el documento.

```ruby
def virtual_queue(priority_queue, normal_queue, current_number)
  if current_number < current_number + 1
    current_number = current_number + 1
    priority_queue.each do |person|
      if person[:number] == current_number
        puts "Enviando 'es tu turno' a: #{person[:phone_number]}"
      end
    end
    normal_queue.each do |person|
      if person[:number] ⩾ current_number
        puts "Enviando 'el número #{current_number} está siendo atendido' a: #{person[:phone_number]}"
      end
    end
  end
end

priority_queue = [
  { number: 1, phone_number: '123456789' },
  { number: 2, phone_number: '987654321' },
  { number: 3, phone_number: '123456789' },
  { number: 4, phone_number: '987654321' },
]

normal_queue = [
  { number: 5, phone_number: '123456789' },
  { number: 6, phone_number: '987654321' },
  { number: 7, phone_number: '123456789' },
]

current_number = 3

virtual_queue(priority_queue, normal_queue, current_number)
```

# ¿Consultas?



# Ingeniería de Software

## 15 - Patrones de Diseño
## Comportamiento

IIC2143-3

**Josefa España**

jpespana@uc.cl