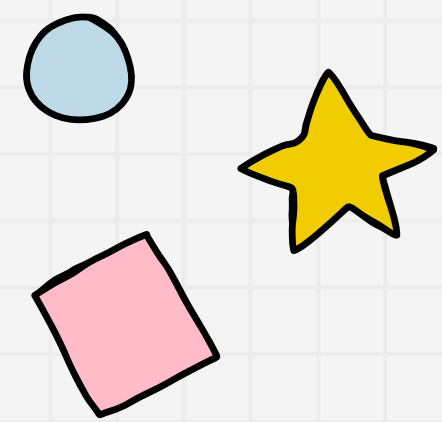
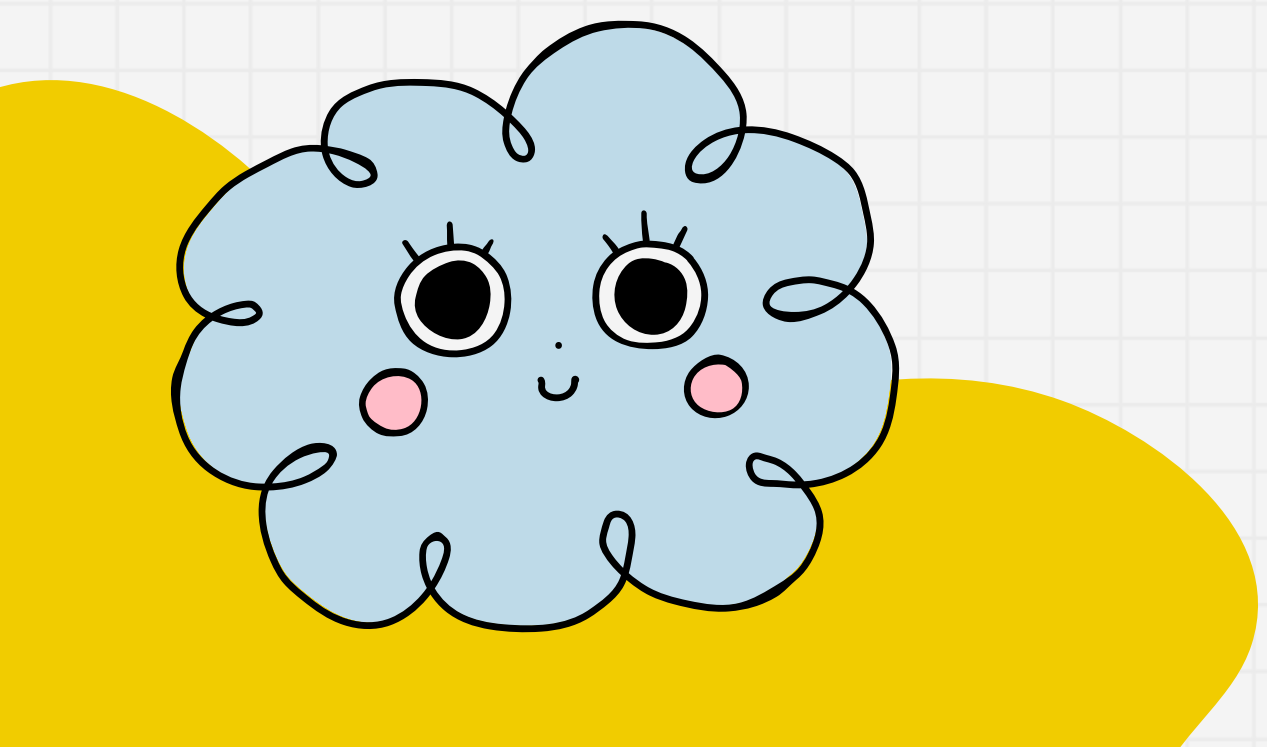
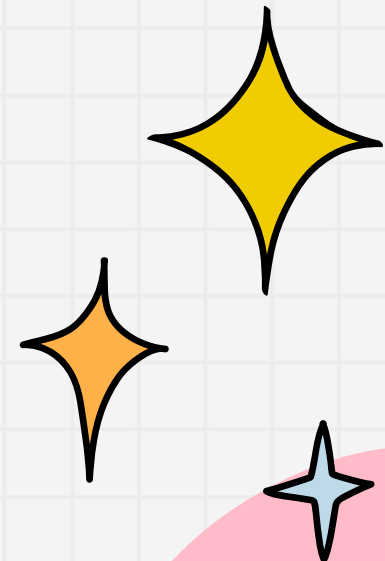


DISEÑO DE SOFTWARE

TEMPLATE METHOD



Jimena Rivera Álvarez
C36561



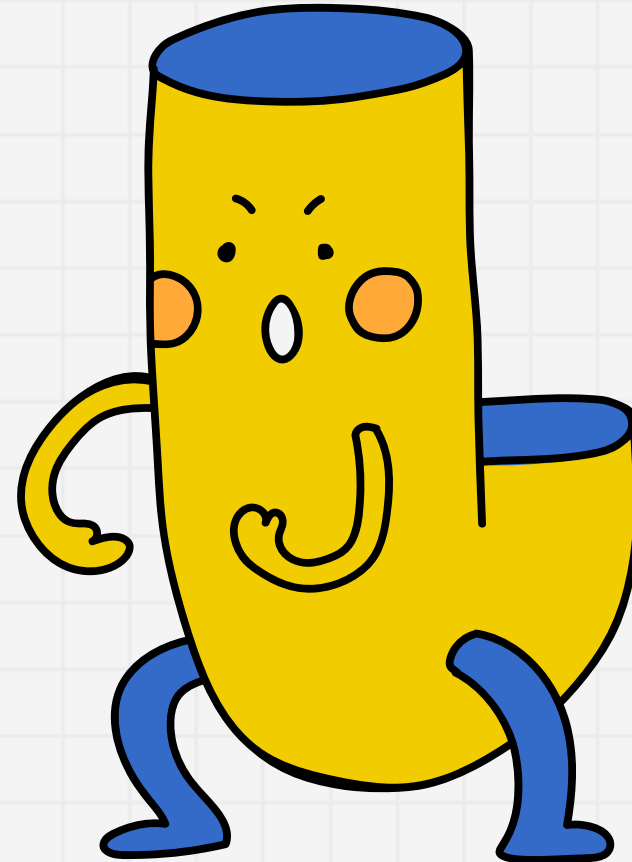
CONTENIDOS

Definición

Problema y solución

Estructura general

Ejemplo del mundo
real

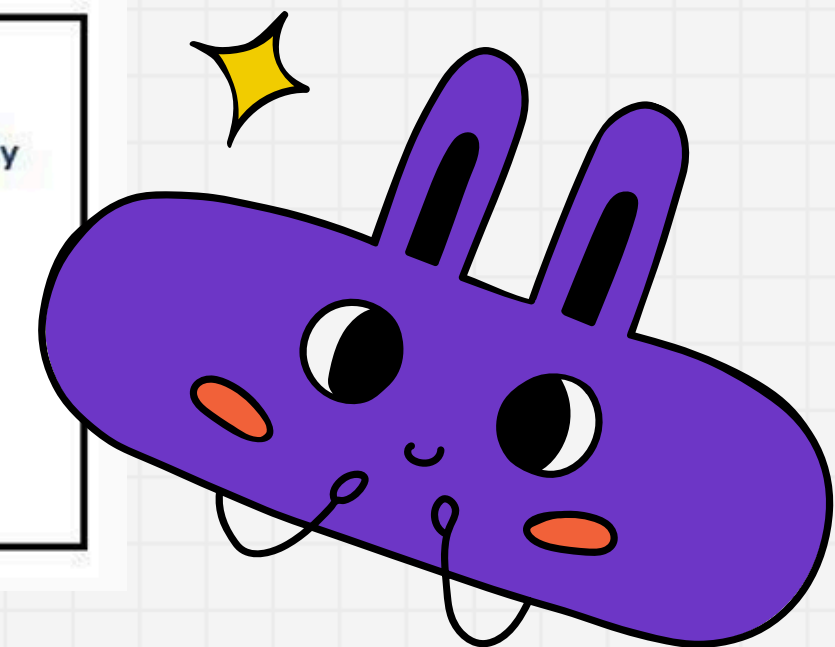
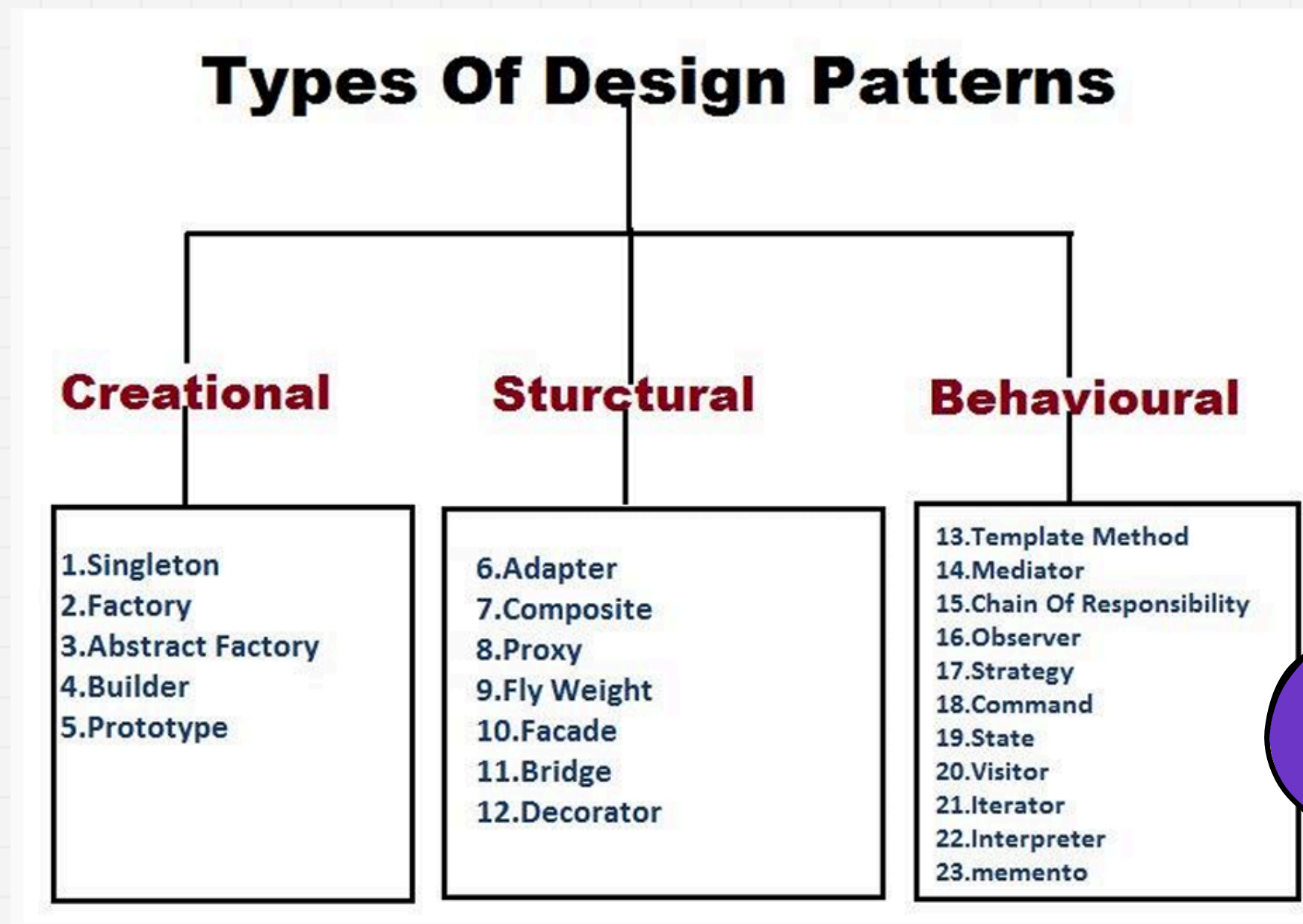
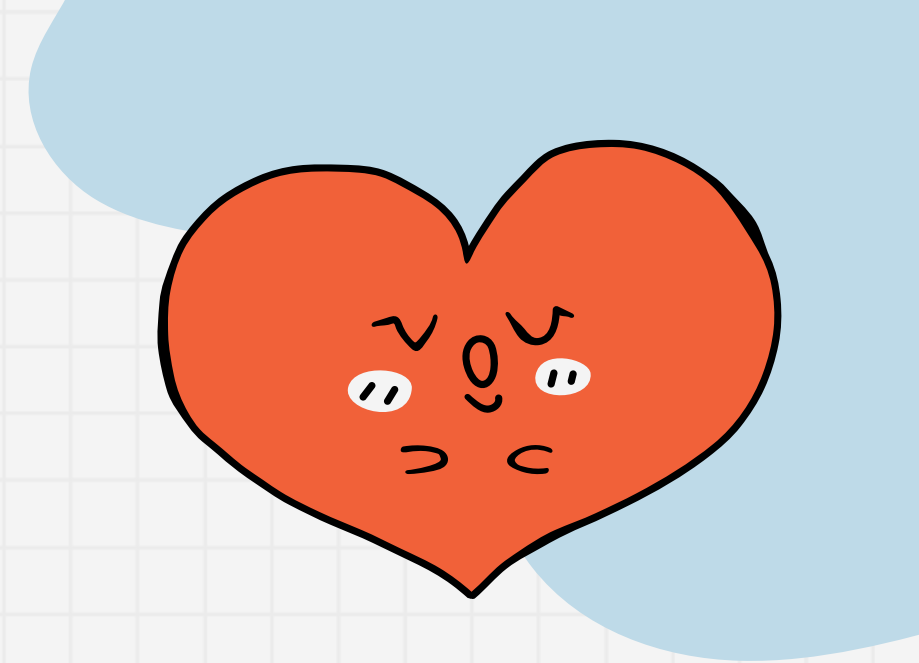
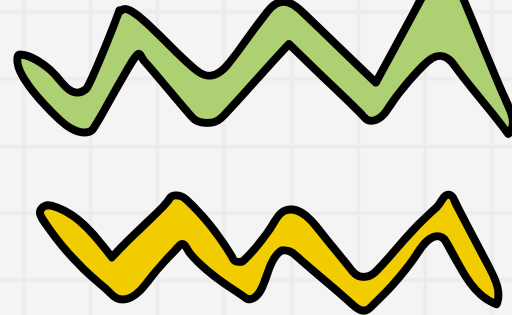


Ventajas y desventajas

Cuando usarlo y
cuando no

Relaciones con
otros patrones

Demo

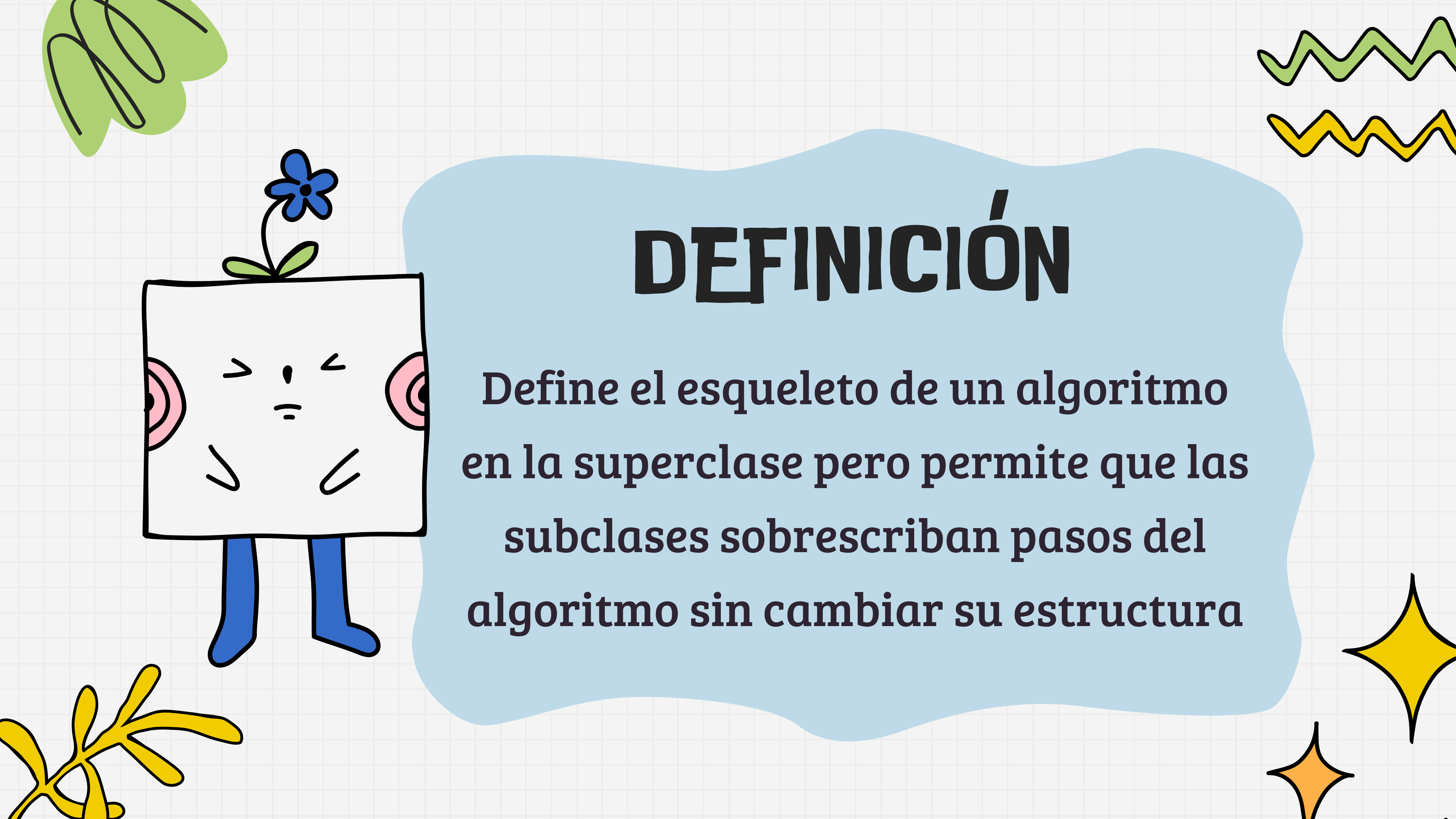


Template Method es un patrón de diseño de
comportamiento

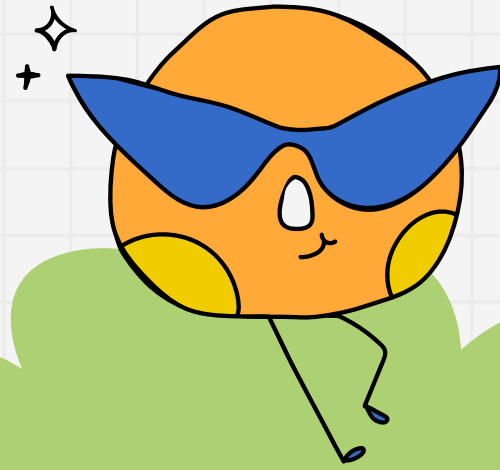


DEFINICIÓN

Define el esqueleto de un algoritmo en la superclase pero permite que las subclases sobrescriban pasos del algoritmo sin cambiar su estructura

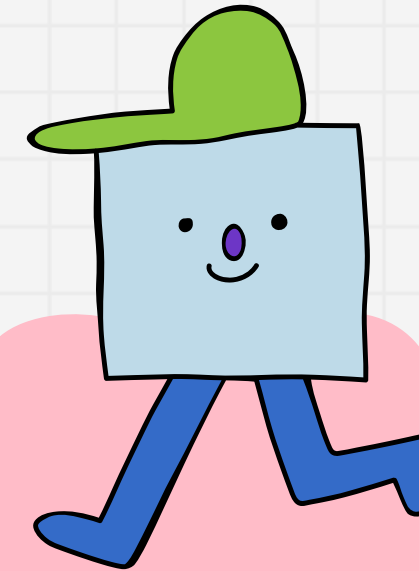


PROBLEMA Y SOLUCIÓN



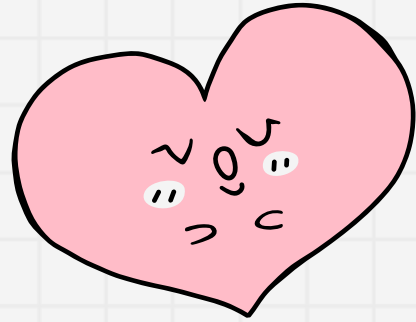
Dos componentes distintos comparten gran parte de su lógica, pero no reutilizan interfaz ni implementación común.

- **Duplicación de código**
- **Doble modificación de componentes**



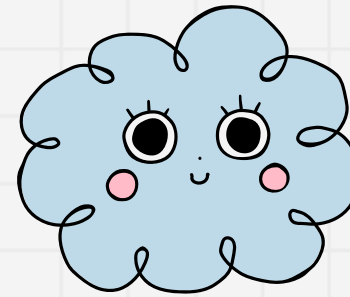
→ Template Method divide el algoritmo en una serie de pasos, los convierte en métodos y y agrupa las llamadas en un único método plantilla que los invoca en un orden específico

ESTRUCTURA GENERAL



Clase Base Abstracta

Define el método plantilla que contiene la secuencia fija del algoritmo



Subclases Concretas

Implementan los pasos variantes y, si lo necesitan, los hooks



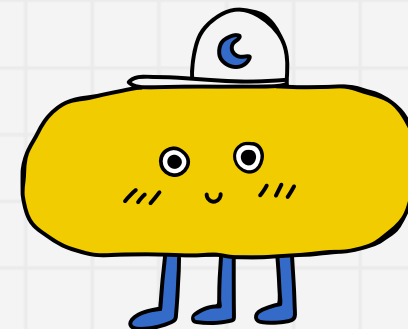
Pasos Fijos

Lógica común que no cambia entre subclases



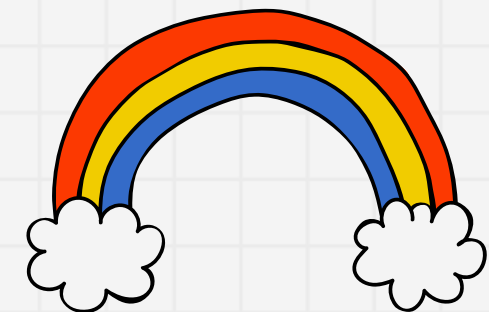
Pasos Variables

Métodos que las subclases deben sobrescribir para personalizar partes concretas del algoritmo



Hooks (opcionales)

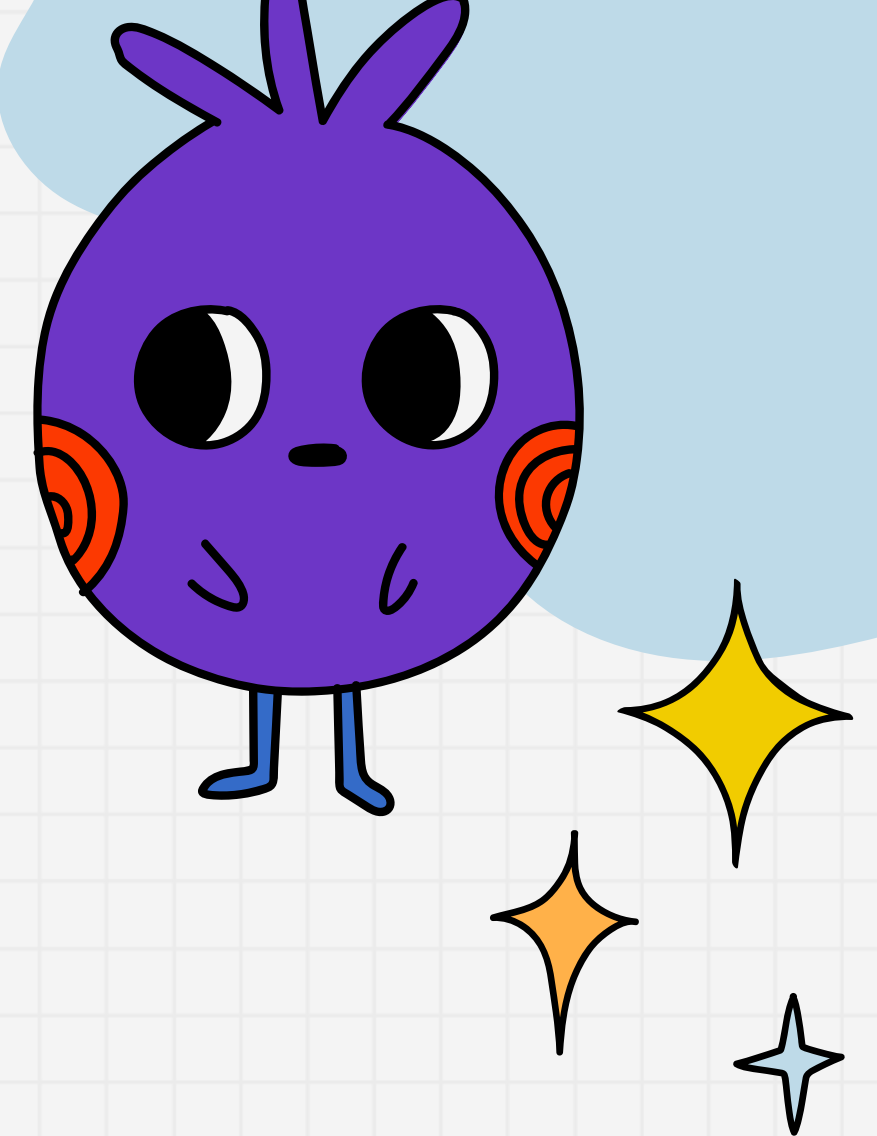
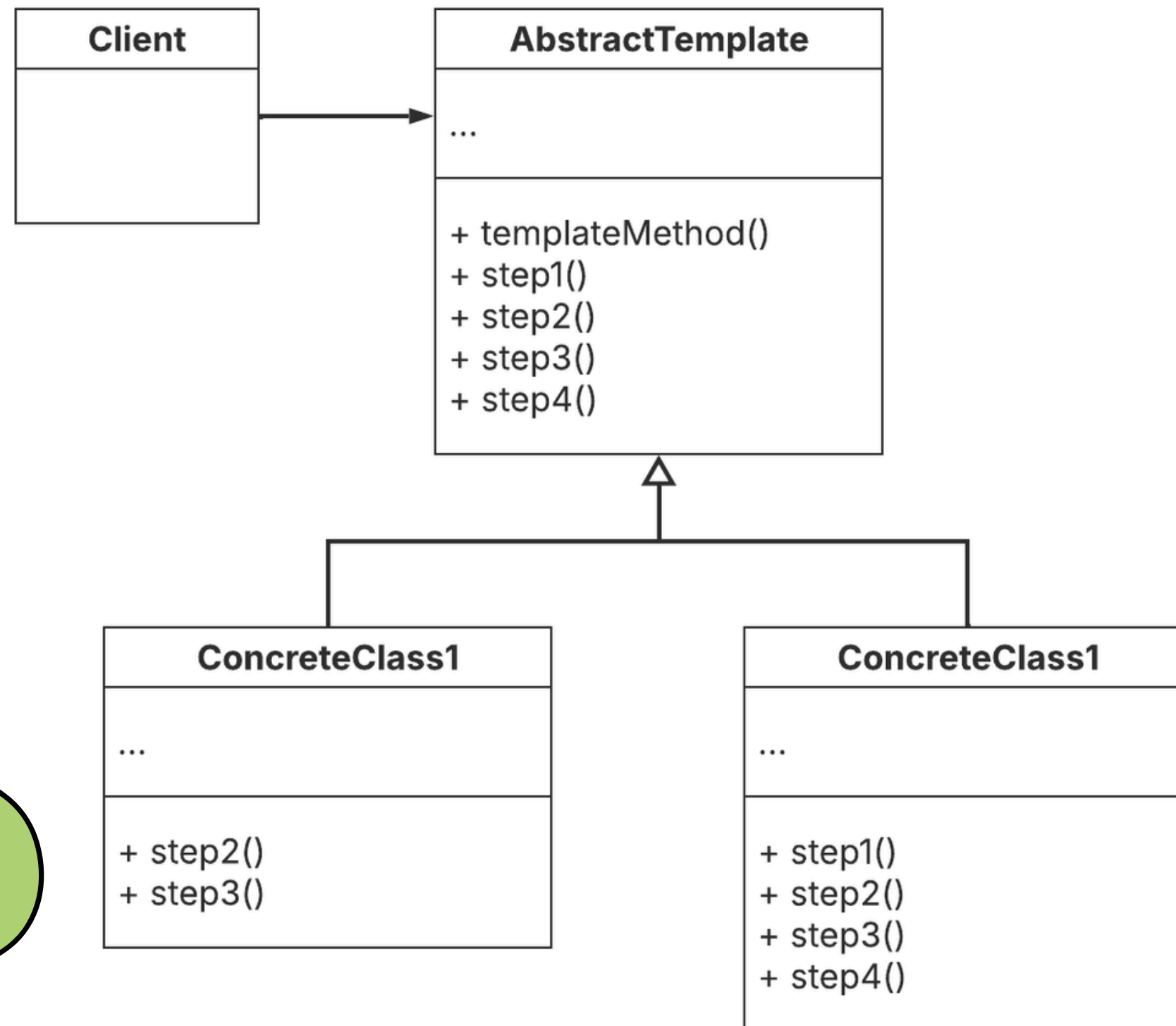
Métodos que las subclases pueden sobrescribir para agregar comportamiento



Cliente

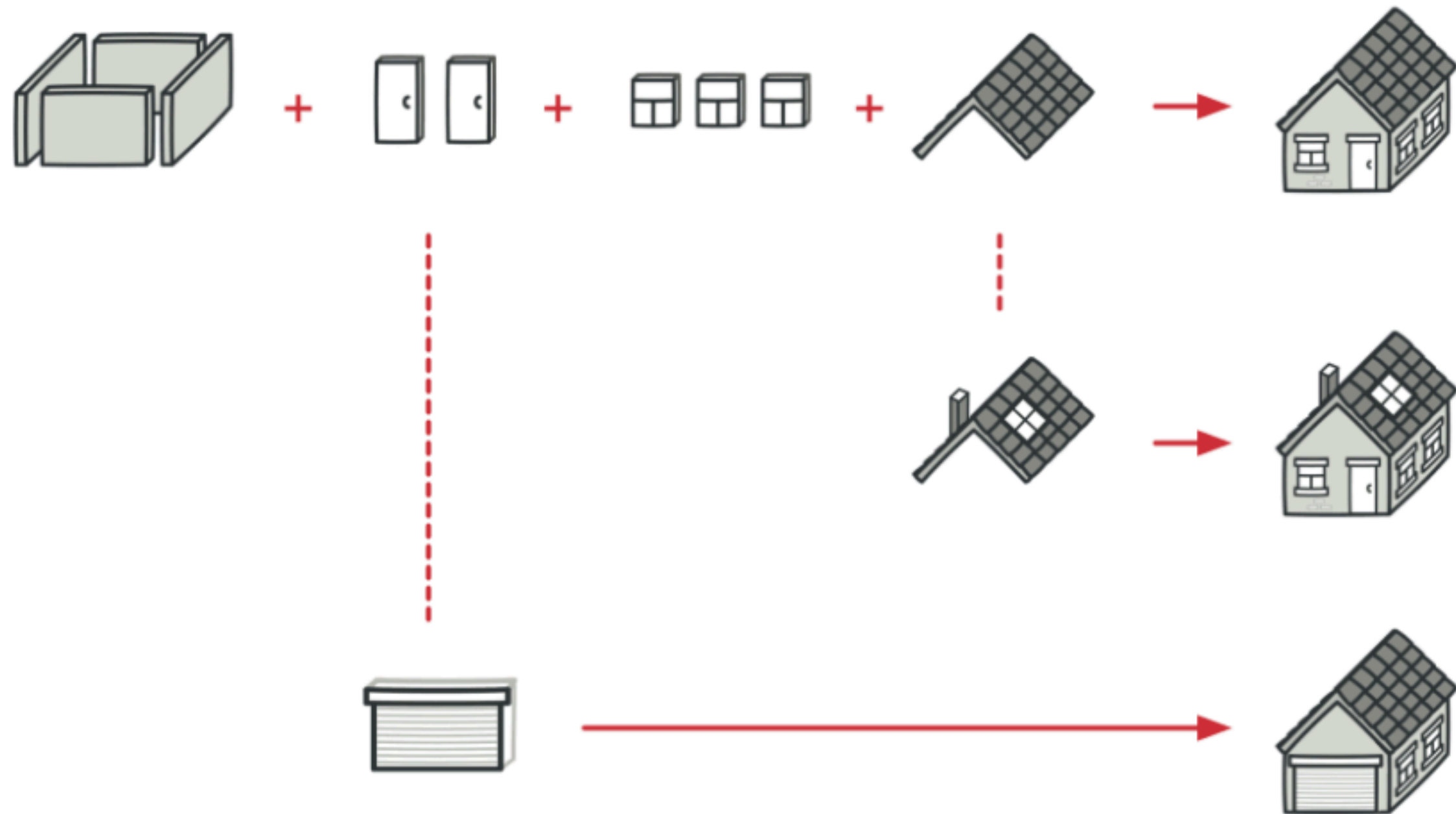
Invoca el método plantilla sobre la clase concreta elegida.
No controla la secuencia

DIAGRAMA DE LA ESTRUCTURA

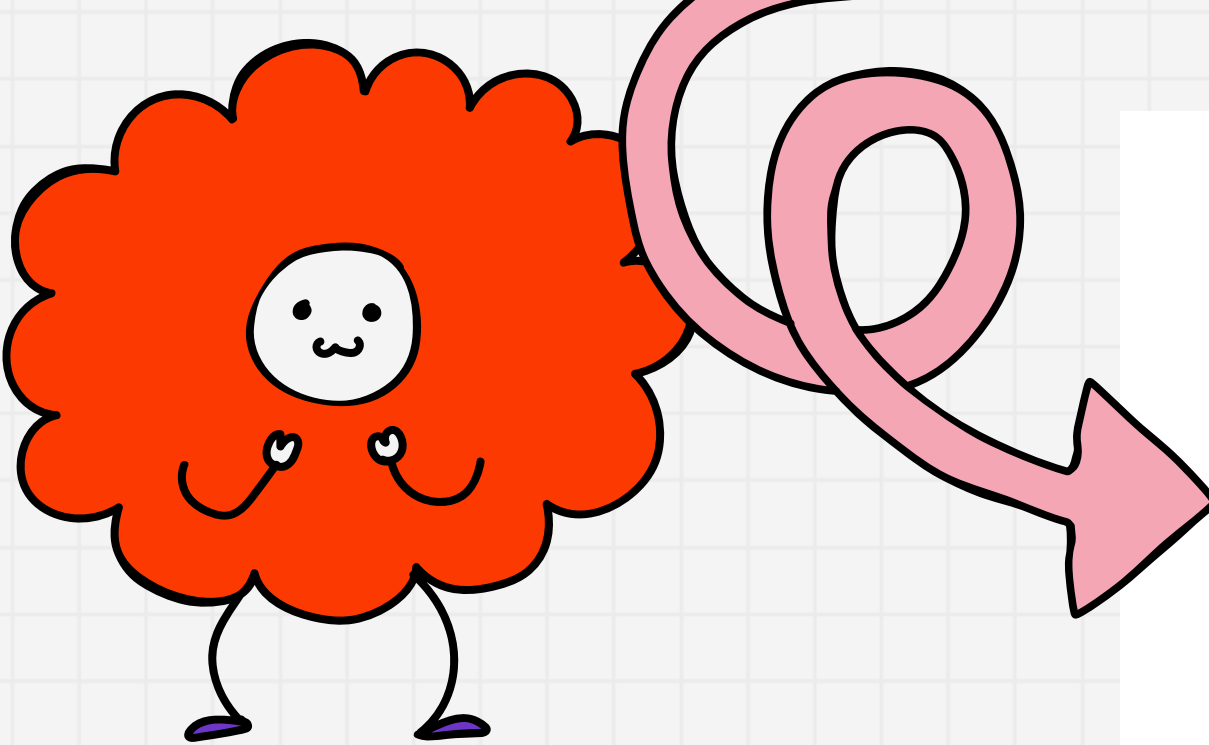


EJEMPLO DEL MUNDO REAL

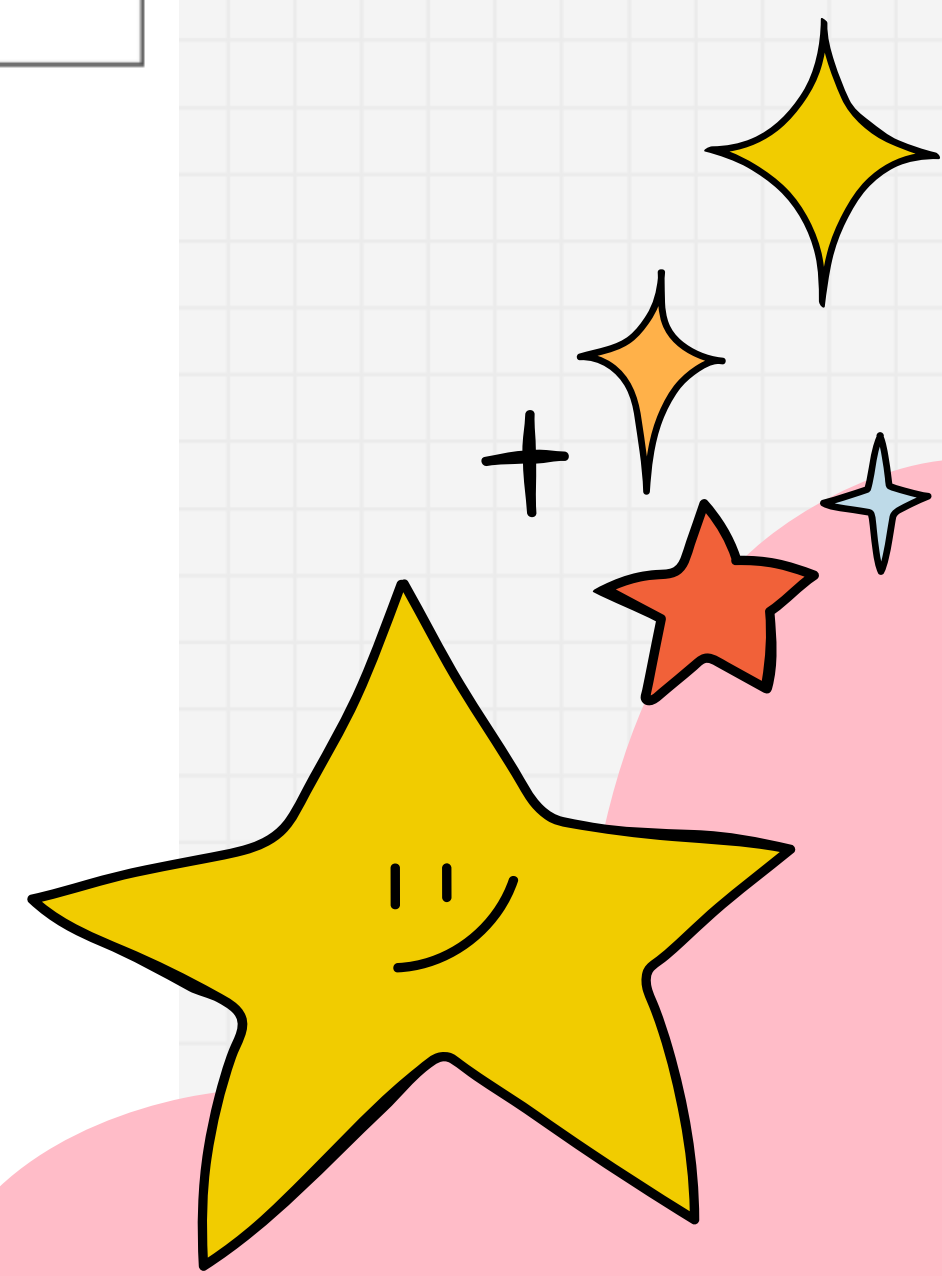
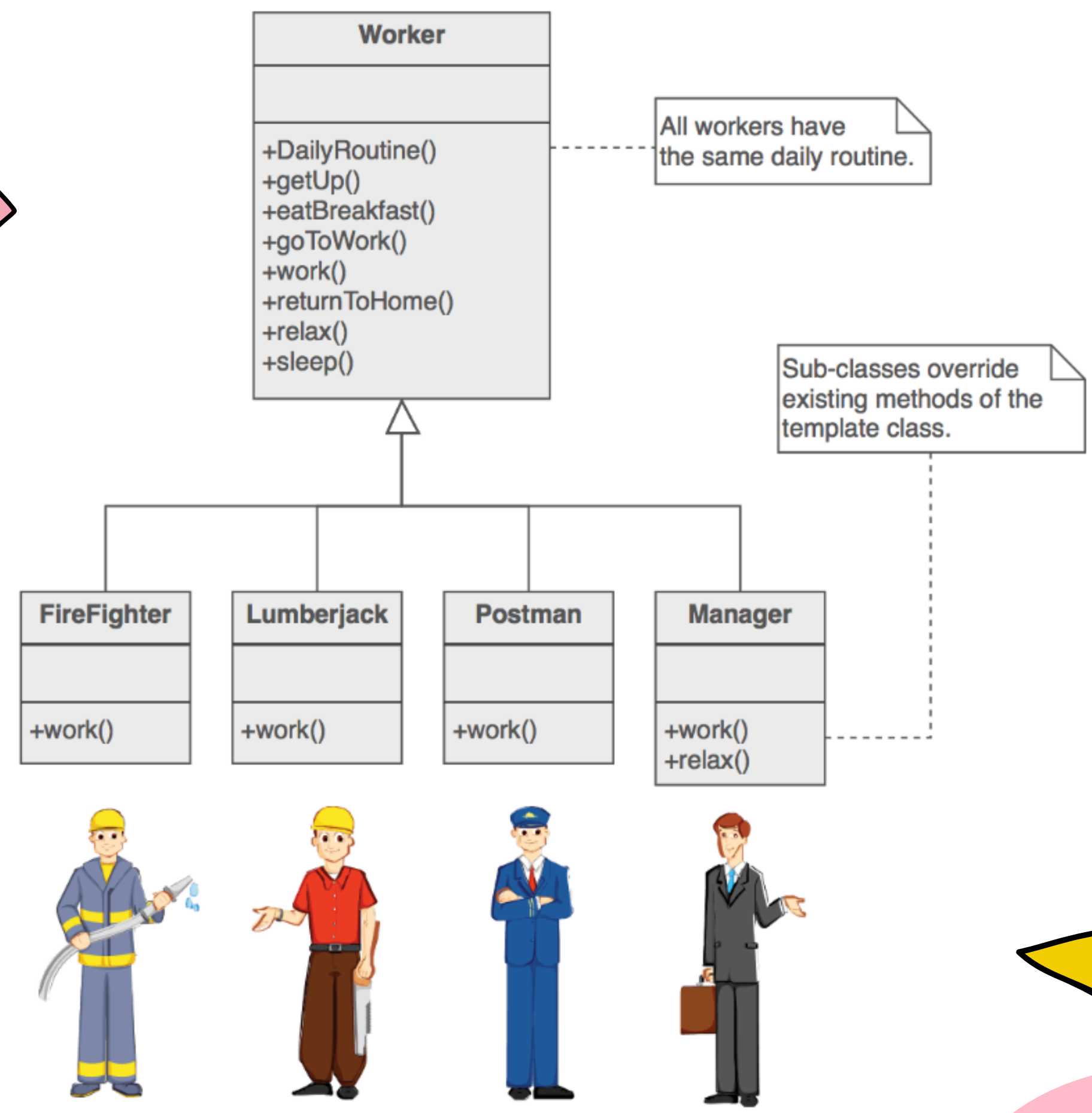
PLAN BASE : CIMIENTOS → ESTRUCTURA → PAREDES → TUBERÍAS → CABLEADO → ACABADOS



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



EJEMPLO DEL MUNDO REAL





**PLANTILLAS LARGAS
(MUCHOS PASOS)
SON DIFÍCILES DE
MANTENER**

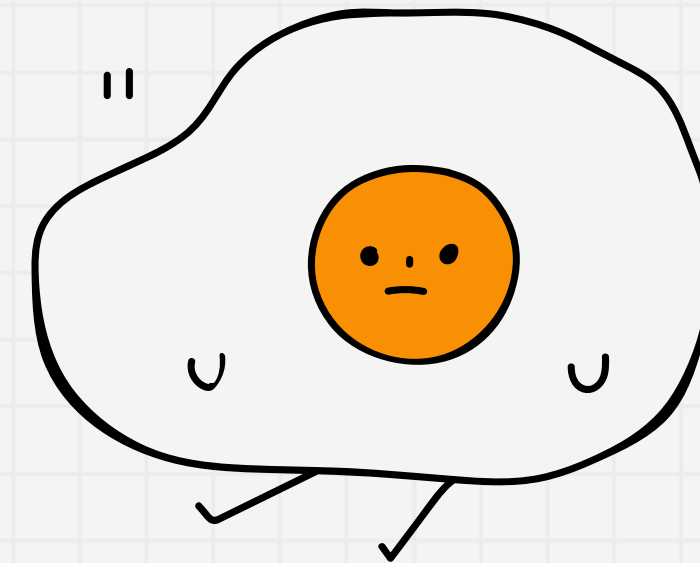
**AUMENTO DE
JERARQUÍA DE
CLASES**

DESVENTAJAS



**ESQUELETO
RESTRICTIVO**



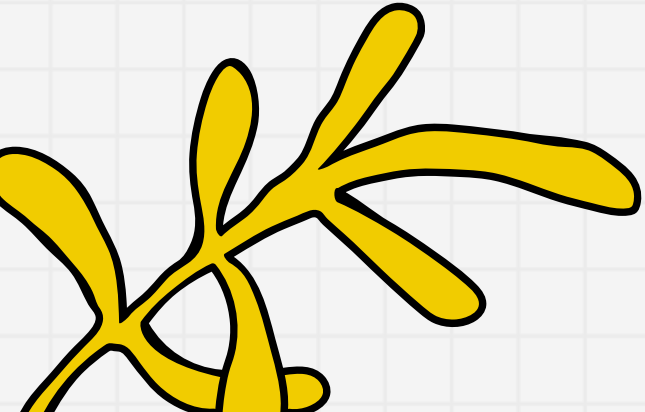
**SUBCLASES MAL
DISEÑADAS PUEDEN
VIOLAR LAS
EXPECTATIVAS DEL
FLUJO**



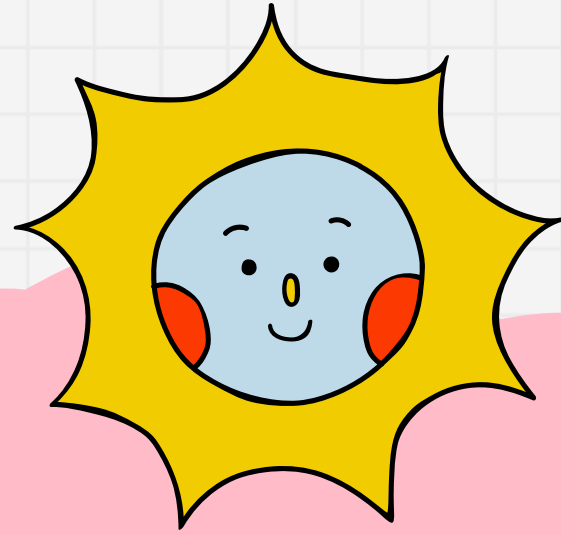


¿CÓMO MEJORA EL MANTENIMIENTO O ESCALABILIDAD DEL SISTEMA?



1. Convierte un algoritmo monolítico en pasos individuales fácilmente extensibles
 2. El mantenimiento puede ser más sencillo ya que un cambio de orden en la secuencia se define una sola vez
 3. Al permitir trabajo en paralelo, el sistema puede crecer de forma rápida, ordenada y sin conflictos con el esqueleto común
- 
- 
- 

CUANDO USARLO Y CUANDO NO USARLO

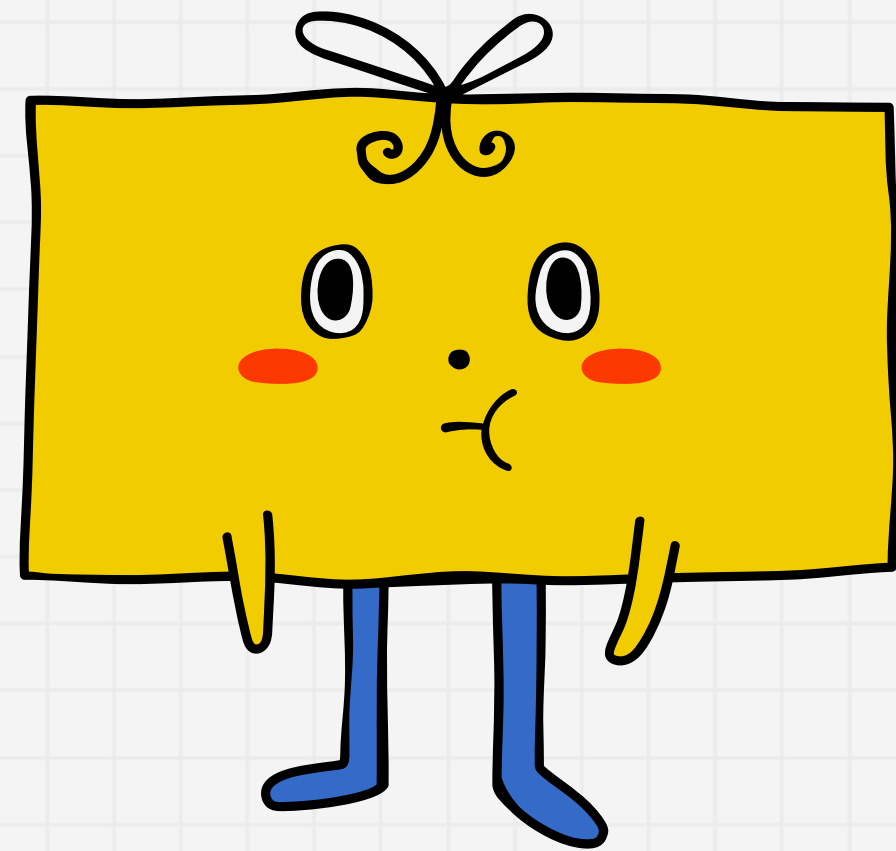


- Tienes muchas clases con algoritmos casi idénticos y pequeñas diferencias.
- Quieres permitir extensión solo en algunos pasos, manteniendo intacta la estructura.
- Quieres estandarizar pasos y reducir duplicación.
- Buscas mantenimiento más fácil en el que un cambio en el orden afecte todas las variantes.



- Si cada variante cambia todo el algoritmo, no solo pasos puntuales mejor usar Strategy o Policy.
- Si la jerarquía crece sin control (demasiadas subclases por pequeñas diferencias).

RELACIONES CON OTROS PATRONES



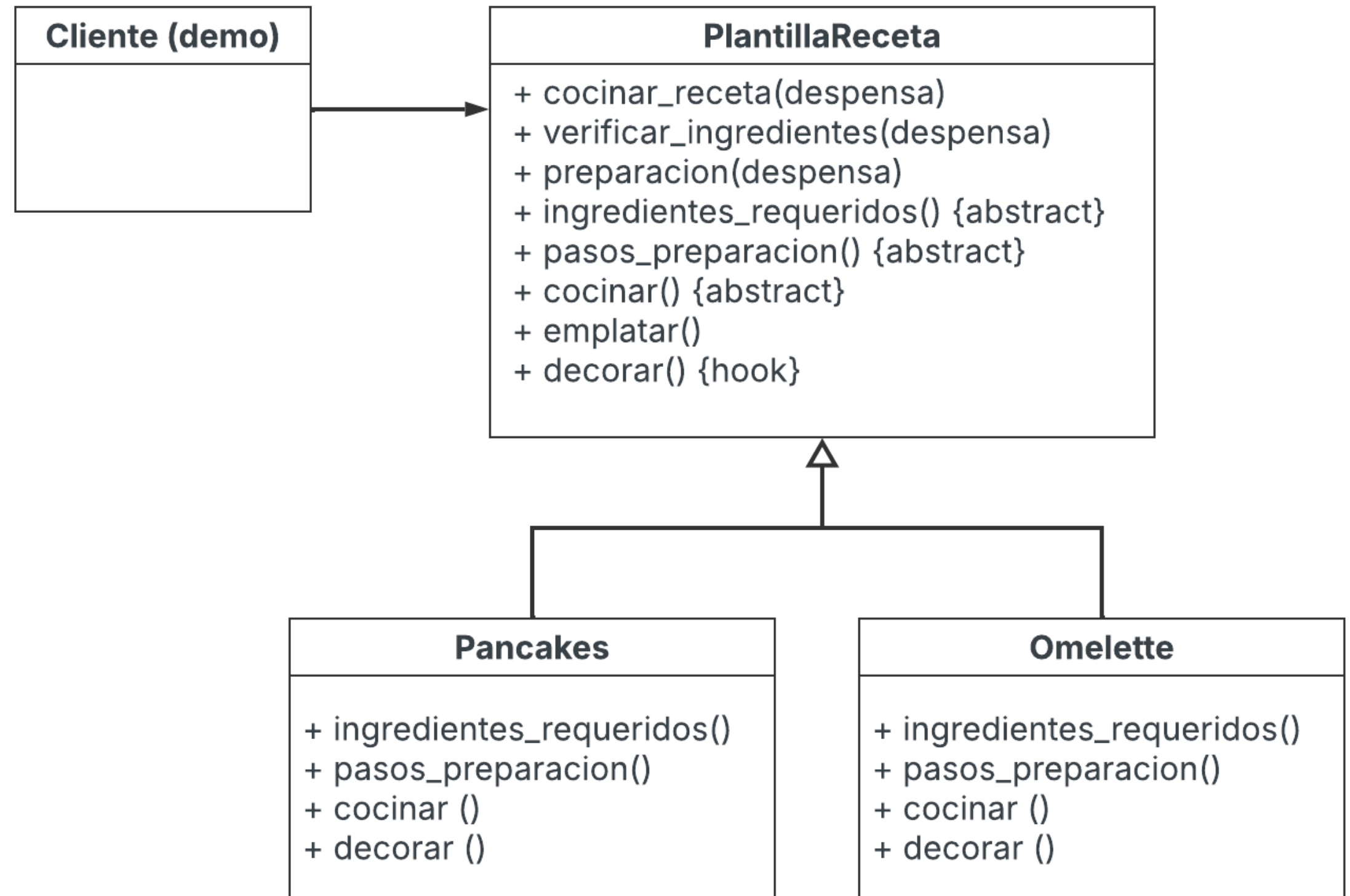
1

Factory Method es una especialización d.el Template Method; este también puede ser un paso dentro de un Template Method mayor.

2

Template Method trabaja al nivel de la clase, por lo que es estático. Mientras que Strategy trabaja al nivel del objeto, permitiéndote cambiar los comportamientos durante el tiempo de ejecución.

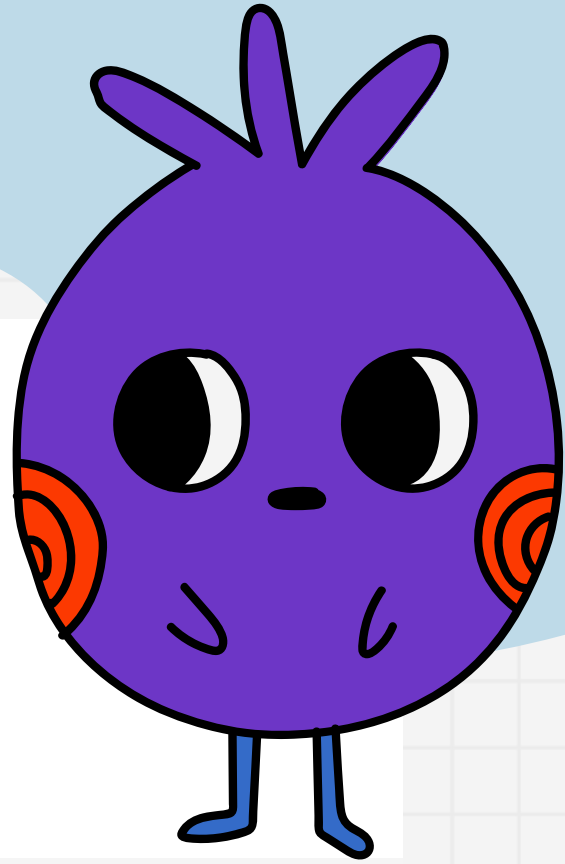
DEMO: TEMPLATE METHOD APLICADO A RECETAS DE COCINA



Clase Base

`PlantillaReceta` es la **clase Base** que implementa el **método plantilla**:

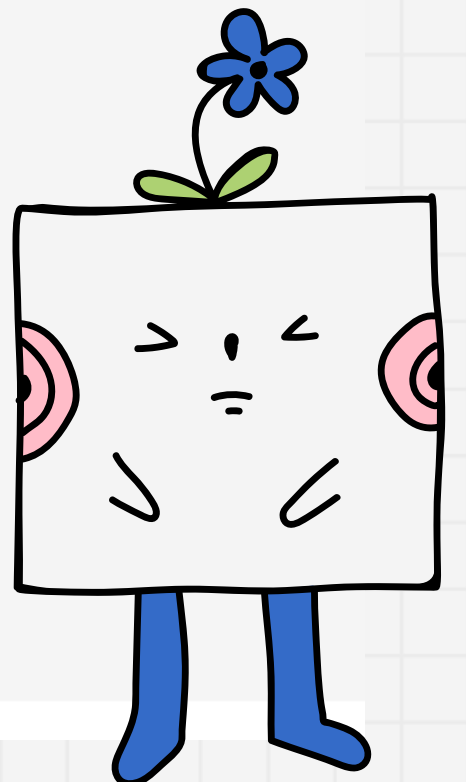
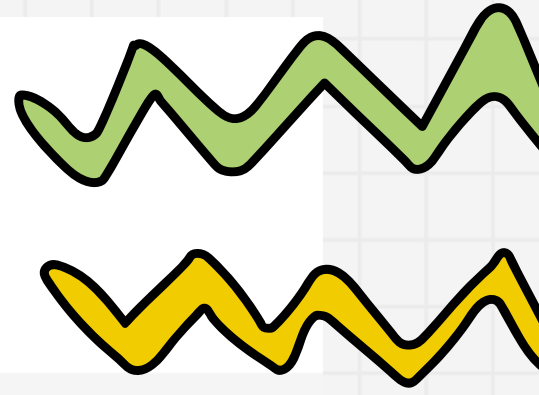
```
def cocinar_receta(self, despensa):  
    self._verificar_ingredientes(despensa) # Paso fijo  
    self._preparacion(despensa) # Paso fijo que llama a pasos_preparacion()  
    self._cocinar() # Paso variable  
    self._emplatar() # Paso fijo  
    self._decorar() # Hook opcional
```



Subclases Concretas

Pancakes es una **Subclase concreta** que implementa los pasos variables:

```
class Pancakes(PlantillaReceta):  
    def ingredientes_requeridos(self):  
        return ["harina", "azucar", "huevos", "leche", "polvo_de_hornear", "vainilla"]  
  
    def pasos_preparacion(self):  
        print(  
            " - Mezclar harina, azúcar, sal y polvo de hornear\n"  
            " - Aparte, batir huevos, leche y vainilla; integrar con la harina y mezclar hasta que quede sin grumos\n"  
            " - Engrasar la sartén con un poco de aceite"  
        )  
  
    def cocinar(self):  
        print(  
            "Verter pequeñas porciones en el sartén a fuego medio; cuando salgan burbujas, voltear.\n"  
            "Cocinar 1-2 min más hasta dorar."  
        )  
  
    def decorar(self):  
        print("Servir con miel, fruta, crema batida o helado. ")
```



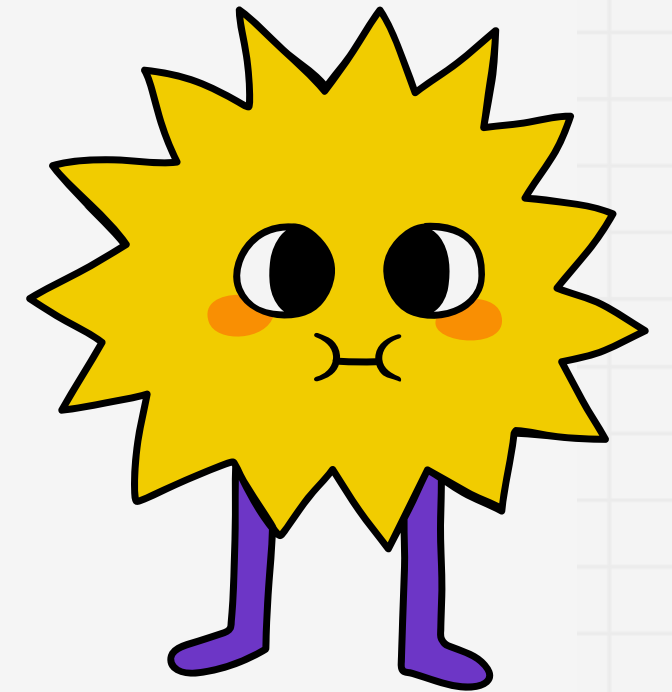
Omelette es otra **Subclase concreta**. Igual que los pancakes, implementa:

```
class Omelette(PlantillaReceta):
    def ingredientes_requeridos(self):
        return ["huevos", "queso", "jamon"]

    def pasos_preparacion(self):
        print(
            " - Batir los huevos con una pizca de sal\n"
            " - Rallar/trozear el queso y cortar el jamón en cuadritos o tiritas\n"
            " - Precalentar sartén y engrasar con un poco de aceite"
        )

    def cocinar(self):
        print(
            "Verter los huevos batidos; cuando cuaje por bordes y el centro siga jugoso,\n"
            "repartir jamón y queso; doblar en media luna y cocinar 1-2 min más según el punto deseado."
        )

    def decorar(self):
        print("Echarle un poco de pimienta y un toque de perejil picado.")
```



Función Auxiliar

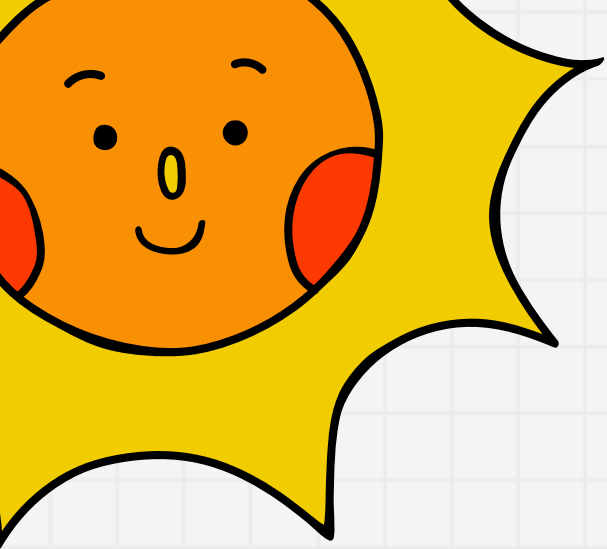
Despensa proporciona funciones para comprobar ingredientes.

Es llamado por `_verificar_ingredientes(...)` para asegurar que la receta se pueda empezar.

No conoce de recetas ni del orden: **no forma parte del patrón, es infraestructura.**

```
class Despensa:
    @staticmethod
    def ingredientes_faltantes(requeridos, disponibles):
        """Devuelve lista de ingredientes faltantes"""
        return [i for i in requeridos if i not in disponibles]
```





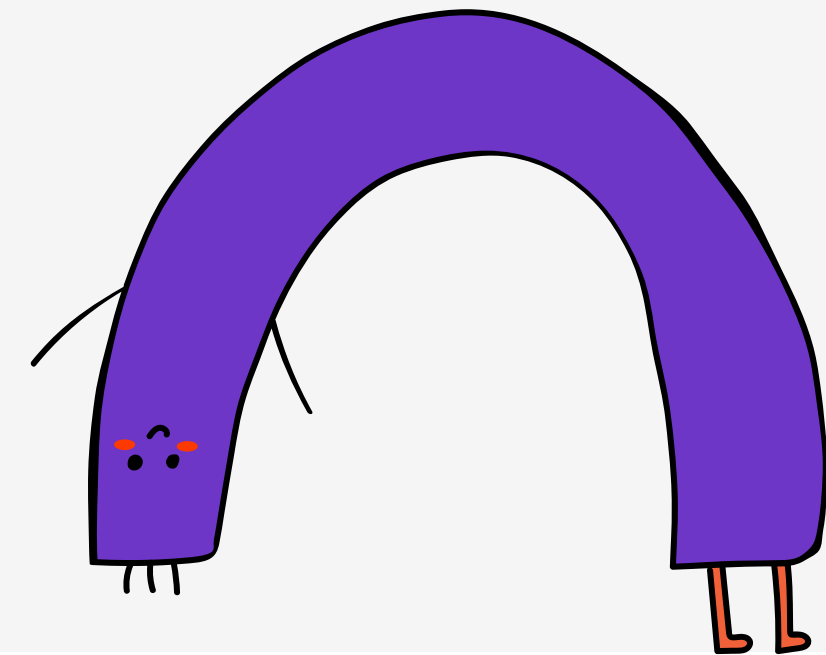
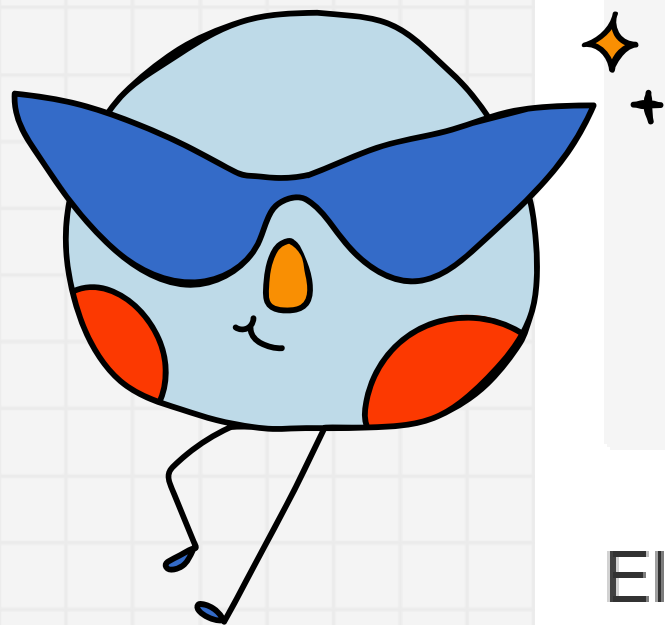
Cliente

demo actua como el **cliente que ejecuta el método plantilla** en las recetas concretas:

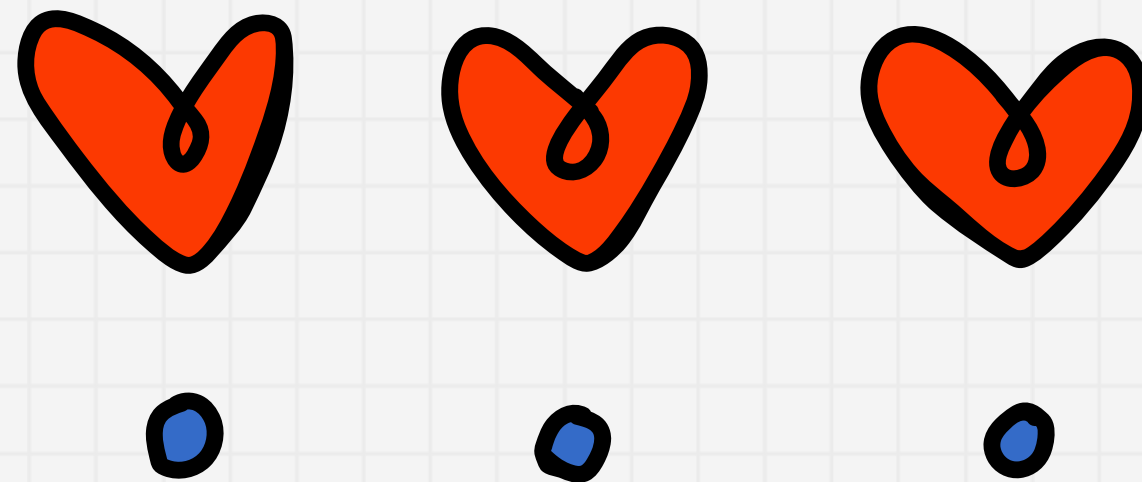
```
if __name__ == "__main__":
    despensa = {
        "sal", "pimienta", "huevos", "queso", "tocino", "harina", "jamon",
        "azucar", "mantequilla", "aceite", "vainilla", "polvo_de_hornear", "leche"
    }

    print("\n==>", Pancakes.__name__)
    Pancakes().cocinar_receta(despensa)

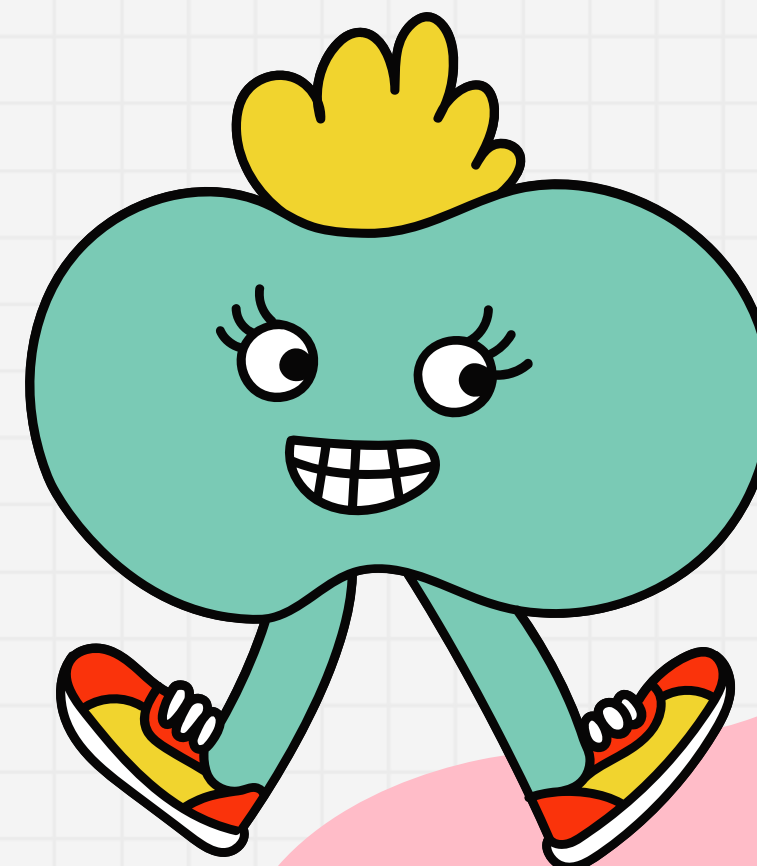
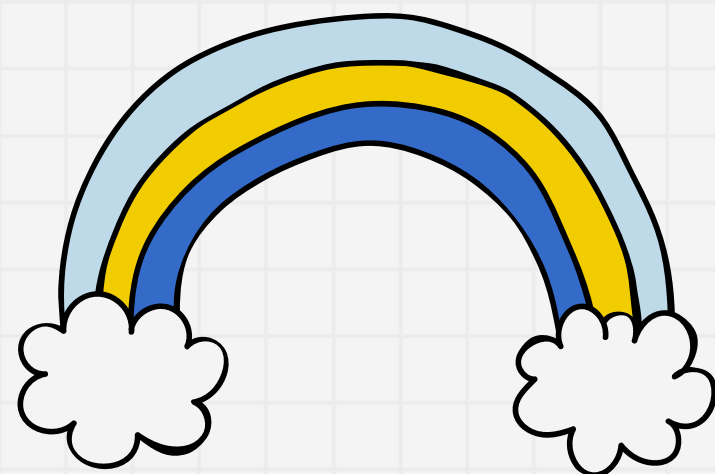
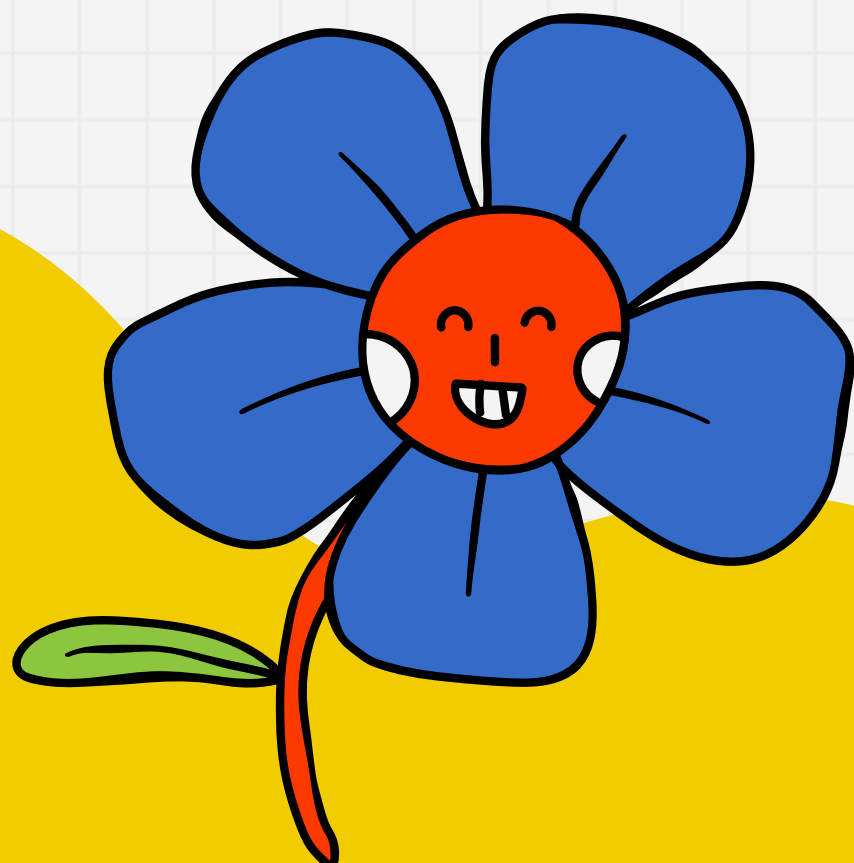
    print("\n==>", Omelette.__name__)
    Omelette().cocinar_receta(despensa)
```



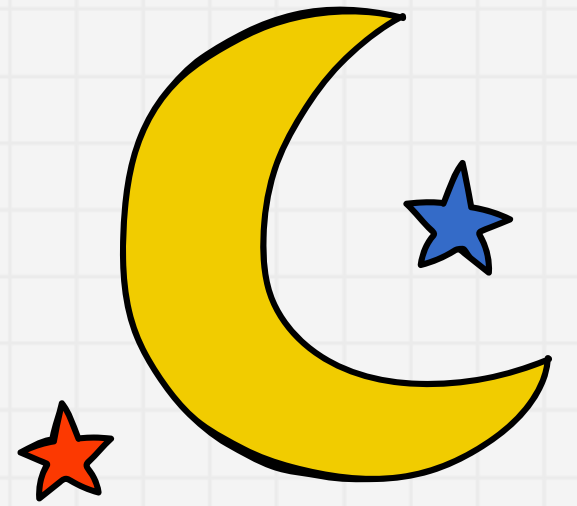
El cliente **no controla la secuencia interna**; solo elige qué receta correr y con qué ingredientes en la despensa.



MUCHÍSIMAS
Gracias



REFERENCIAS



[1] Refactoring.Guru — *Template Method*.

**[<https://refactoring.guru/es/design-patterns/template-method>]
(<https://refactoring.guru/es/design-patterns/template-method>)**

[2] SourceMaking — *Template Method Design Pattern*.

**[https://sourcemaking.com/design_patterns/template_method]
(https://sourcemaking.com/design_patterns/template_method)**

[3] Reactive Programming — *Patrones de Diseño: Template Method*.

<https://reactiveprogramming.io/blog/es/patrones-de-diseno/template-method>

