Template Method

Patrón de comportamiento (POO)

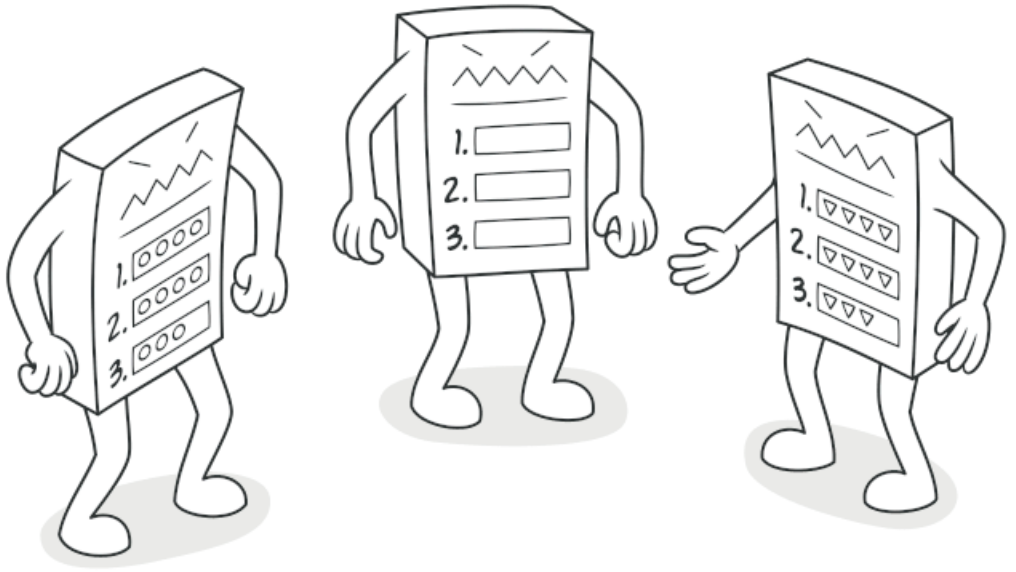
Fuente principal:
https://refactoring.guru/es/design-patterns/template-method

Propósito

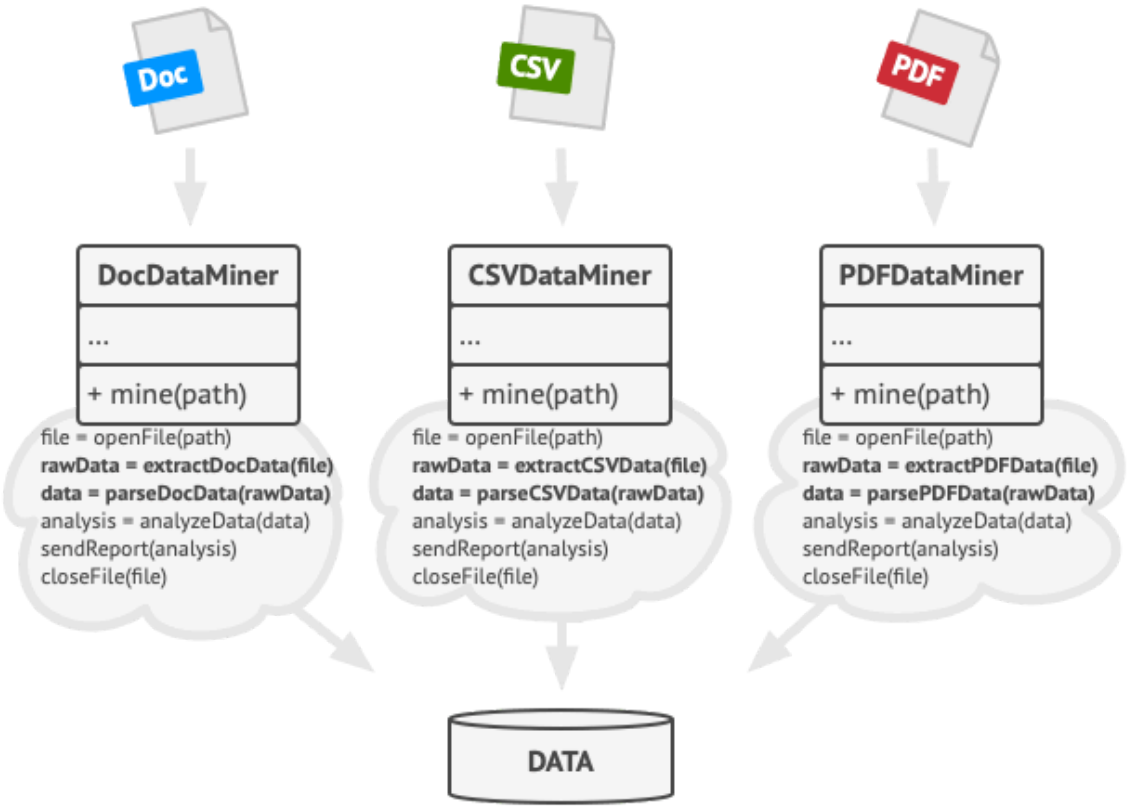
Definir el **esqueleto** de un algoritmo en una clase base y permitir que las subclases **redefinan pasos concretos** sin cambiar la **estructura global**.

Claves:

- Flujo general fijo en el **método plantilla**.
- Variaciones encapsuladas en **operaciones primitivas** y/o **ganchos (hooks)**.



Imágenes extraídas de Refactoring.Guru



Problema

Cuando tienes varias clases que hacen *casi* lo mismo:

- Comparten gran parte del algoritmo.
- Cambian algunos pasos según el caso (p.ej., formato CSV/DOC/PDF).
- Terminas con **duplicación** y cambios costosos (si el flujo cambia, hay que tocar muchas clases).

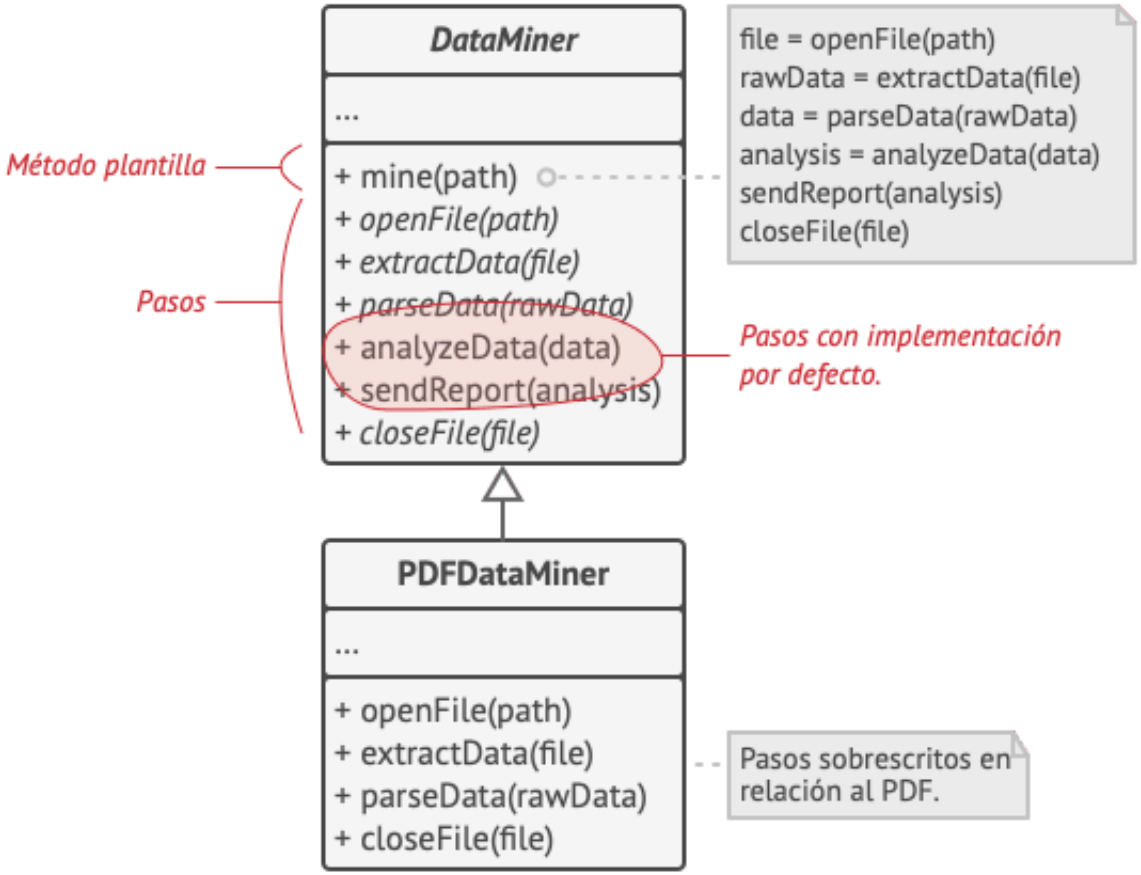
Además, el cliente suele acabar con condicionales para elegir la variante (y pierdes polimorfismo).

Imágenes extraídas de Refactoring.Guru

Solución

1. Divide el algoritmo en **pasos**.
2. Crea un **método plantilla** que llame a esos pasos en un orden fijo.
3. Haz que los pasos variables sean:
   - `abstract` (obligatorios)
   - o con implementación por defecto (opcionales)
   - y añade **hooks** (métodos vacíos o booleanos) para puntos de extensión.

Resultado: reuso del flujo y variación controlada.

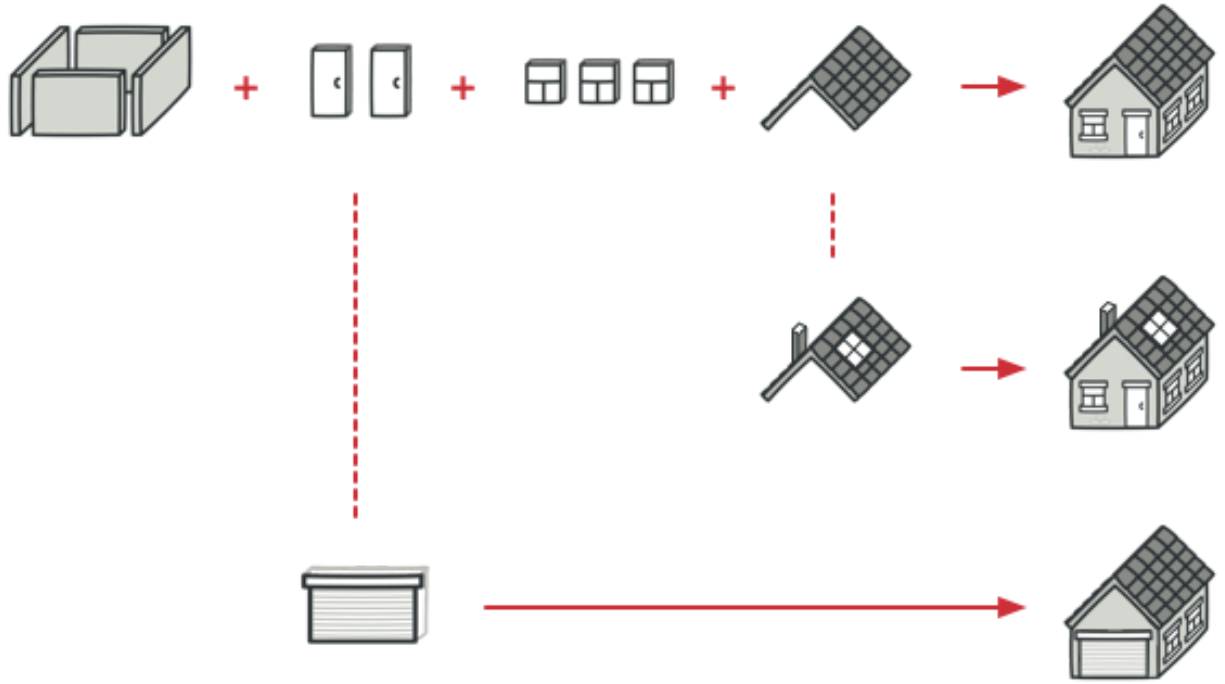


Imágenes extraídas de Refactoring.Guru

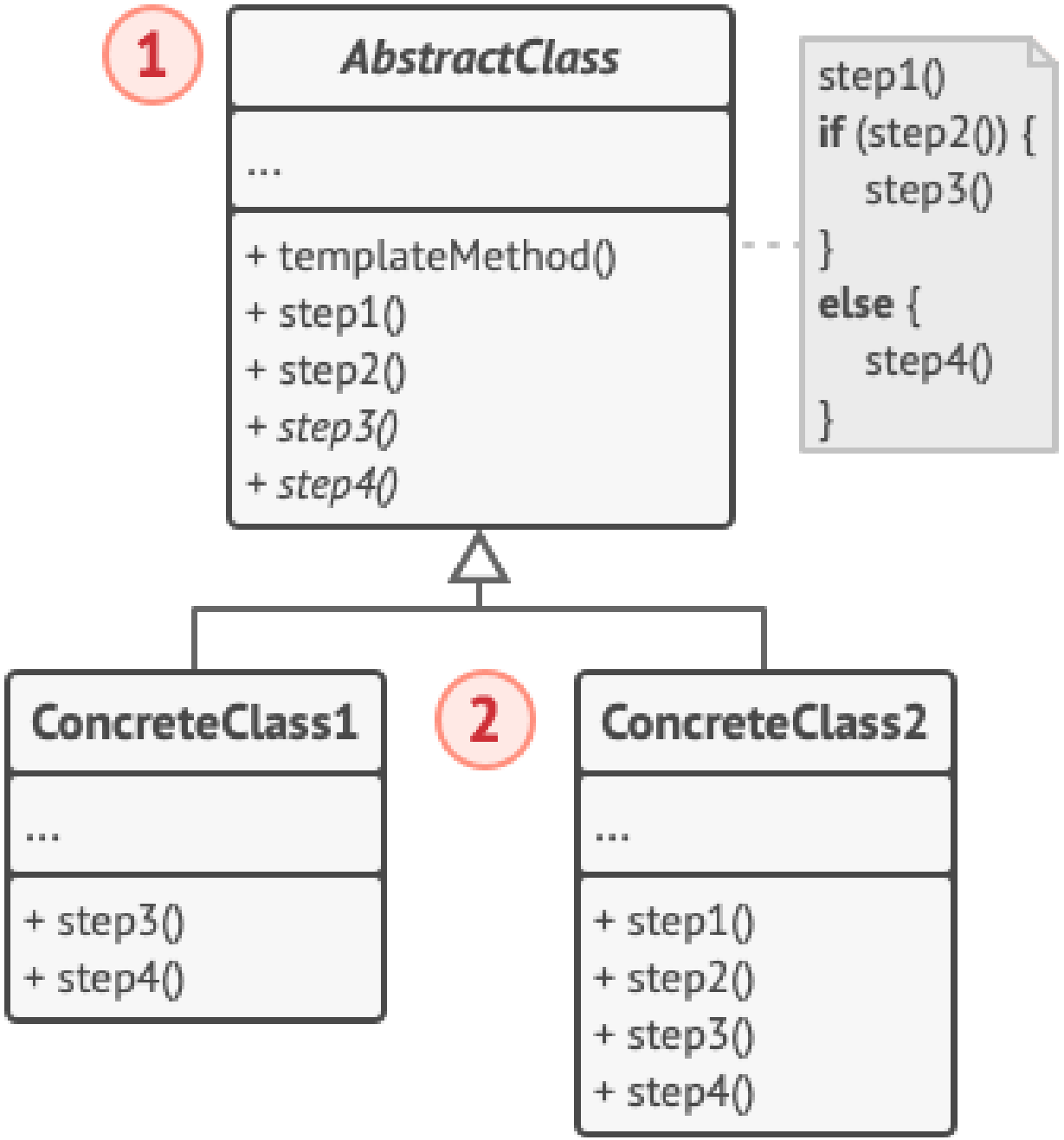
Analogía en el mundo real

Un **plano** de vivienda define pasos (cimientos → estructura → instalaciones → acabados), pero permite personalizar detalles:

- misma “plantilla” de construcción
- variaciones en ciertos puntos (materiales, distribución, extras)



Imágenes extraídas de Refactoring.Guru



Estructura

- **Clase abstracta**: declara los pasos del algoritmo y define el **método plantilla**.
- **Clases concretas**: implementan/sobrescriben pasos, pero **no** deberían alterar el orden global.

Imágenes extraídas de Refactoring.Guru

Aplicabilidad

Úsalo cuando:

- Quieras permitir extender **partes** del algoritmo, pero no el algoritmo completo.
- Tengas muchas clases con algoritmos **casi idénticos** (pequeñas diferencias) y quieras eliminar duplicación.
- Te interese mover comportamiento común a una superclase y dejar en subclases solo lo variable.

Pros y contras

Pros

- Reduce **duplicación** al elevar código común.
- Cambios en el flujo general se hacen en un sitio (la clase base).
- Facilita polimorfismo: el cliente usa la **abstracción**.

Contras

- Algunas subclases pueden sentirse “encorsetadas” por la plantilla.
- Riesgo de romper Liskov si una subclase “anula” un paso por defecto de forma inesperada.
- Puede ser difícil de mantener si la plantilla tiene demasiados pasos.

Código Java (puntos clave)

Basado en `code/es/uva/poo/templatemethod`.

- `MineriaDatos#minar()` fija el flujo y se marca `final`.
- Dos pasos *variables* se declaran `abstract`.
- Se añaden **hooks** para personalización/salto de fases.

```java
public abstract class MineriaDatos {
  public final void minar() {
    abrirArchivo();

    String brutos = extraerDatosBrutos();
    List<String> datos = parsearDatos(brutos);

    if (debeAnalizar()) {         // hook booleano
      antesDeAnalizar(datos);     // hook vacío
      analizarDatos(datos);       // paso común
      despuesDeAnalizar(datos);   // hook vacío
    }

    enviarInforme(datos);
    cerrarArchivo();
  }

  protected abstract String extraerDatosBrutos();
  protected abstract List<String> parsearDatos(String datosBrutos);

  protected boolean debeAnalizar() { return true; }
  protected void antesDeAnalizar(List<String> datos) {}
}
```

Código Java (variación con hooks)

Una subclase puede **cambiar pasos concretos** sin tocar el flujo:

```java
public class MineriaDatosPDF extends MineriaDatos {
  @Override
  protected boolean debeAnalizar() {
    return false; // se salta el análisis
  }

  @Override
  protected void enviarInforme(List<String> datos) {
    System.out.println("Informe PDF (resumen), secciones: " + datos.size());
  }
}
```