TEMPLATE METHOD

{

<Por="Carlota Ruiz Campesino"/>

}

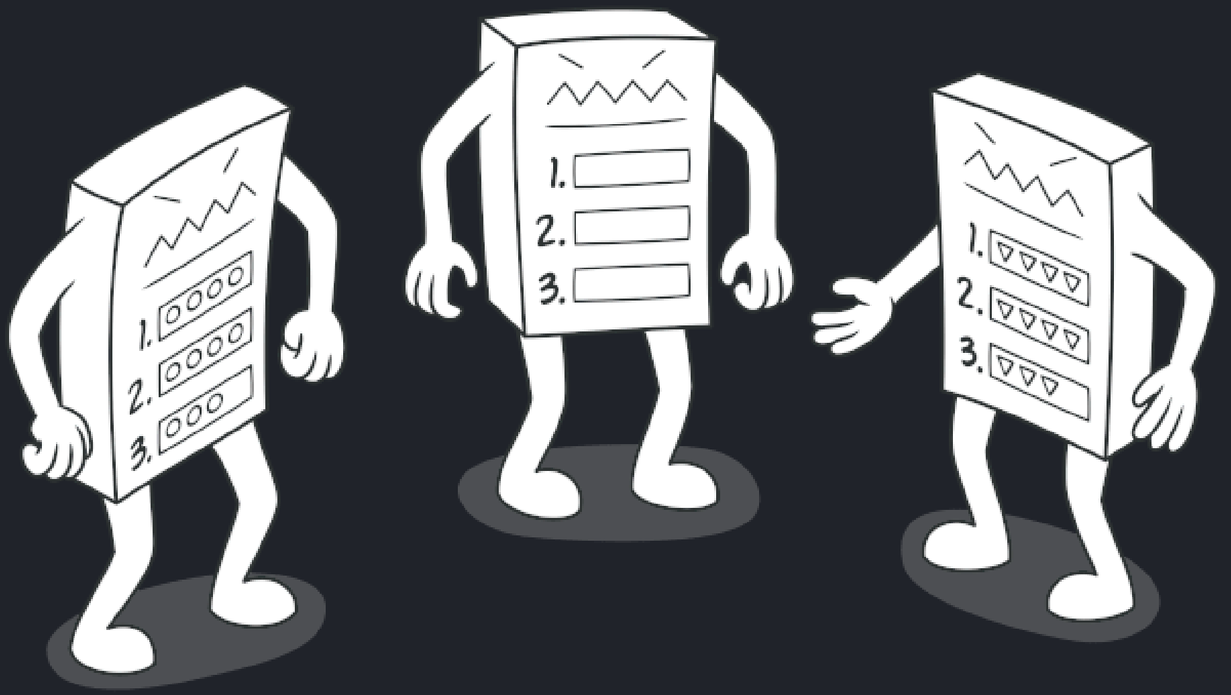
Contenidos

Próposito{

Template Method define el esqueleto de un algoritmo en una operación y delega algunos pasos en subclases, permitiéndoles redefinir ciertos pasos sin alterar la estructura del algoritmo.

Ejemplos:

- Frameworks de Interfaces Gráficas (GUI)
- Sistemas de Autenticación
- Análisis de Datos
- Videojuegos
- Sistemas de Procesamiento de Documentos



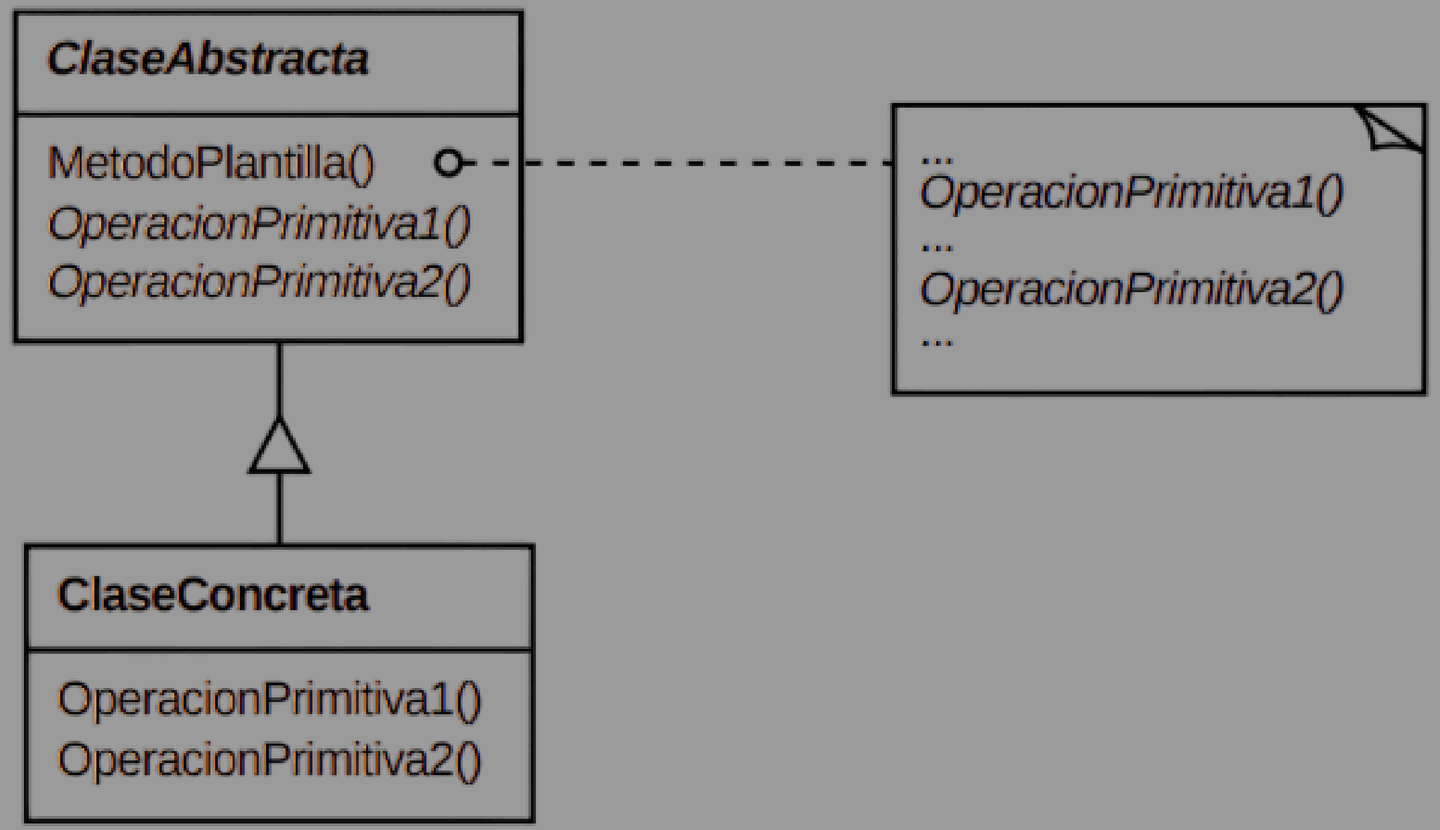
}

Aplicabilidad {

- Se usa para implementar las partes invariantes de un algoritmo y delegar el comportamiento variable a las subclases.
- Sirve para factorizar comportamiento repetido de subclases en una clase común, evitando código duplicado y facilitando la refactorización.
- Permite controlar las extensiones en subclases definiendo un método plantilla que permita solo ciertas extensiones en puntos específicos.

}

Estructura {


ClaseAbstracta
MetodoPlantilla()
OperacionPrimitiva1()
OperacionPrimitiva2()
...
OperacionPrimitiva1()
...
OperacionPrimitiva2()
...
ClaseConcreta
OperacionPrimitiva1()
OperacionPrimitiva2()

}

Participantes {

- ClaseAbstracta (Aplicación): Define operaciones primitivas abstractas que las subclases implementan y contiene el método plantilla que estructura el algoritmo.
- ClaseConcreta (MiAplicacion): Implementa las operaciones primitivas específicas del algoritmo.

Colaboraciones:

- ClaseConcreta depende de ClaseAbstracta para ejecutar los pasos fijos del algoritmo.

}

Variaciones del Patrón

{

1. Template Hook
2. Template Callback
3. Template Strategy
4. Template Factory Method

Patrones relacionados:

- Strategy

}

Ventajas y desventajas

{

Ventajas:

1. Reutilización de Código: Permite definir una estructura común para un algoritmo en la clase base y reutilizar el código en subclases, lo que reduce la duplicación y facilita el mantenimiento del software.

2. Claridad y Mantenimiento: Al separar las partes del algoritmo en métodos específicos, se mejora la claridad del código. Esto hace que sea más fácil de entender y mantener, lo que es beneficioso para el trabajo en equipo y el desarrollo a largo plazo.

Desventajas:

1. Dependencia de la Clase Base: Las subclases dependen de la implementación de la clase base, lo que puede limitar la libertad de diseño en la personalización de comportamientos y dificultar la realización de cambios en el algoritmo sin afectar a todas las subclases.

}

Ejemplo videojuego de coches {

```
abstract class CarreraCoche {

    // Método plantilla que define el flujo general de la carrera
    public final void realizarCarrera() {
        iniciarMotor();     // Iniciar el motor del coche
        for (int i = 0; i < 3; i++) {  // Simula 3 vueltas en la pista
            System.out.println("\n--- Vuelta " + (i + 1) + " ---");
            acelerar();     // Acelerar el coche
            tomarCurvas();  // Tomar las curvas
            frenar();       // Frenar el coche
        }
        cruzarMeta();      // Cruza la meta al finalizar la carrera
    }

    // Métodos abstractos que las subclases deben implementar
    protected abstract void acelerar();     // Método para acelerar el coche
    protected abstract void tomarCurvas();  // Método para tomar curvas
    protected abstract void frenar();       // Método para frenar el coche

    // Método concreto, común para todos los tipos de coches
    private void iniciarMotor() {
        System.out.println("El motor del coche ha sido iniciado.");
    }

    // Método concreto, común para todos los tipos de coches
    private void cruzarMeta() {
        System.out.println("¡El coche ha cruzado la meta!");
    }
}
```

}

Ejemplo videojuego de coches {

```
class CocheDeCalle extends CarreraCoche {

    @Override
    protected void acelerar() {
        System.out.println("El coche de calle acelera lentamente...");
        for (int i = 0; i < 4; i++) {
            System.out.println("Velocidad actual: " + (i * 10) + " km/h");
        }
        System.out.println("El coche de calle ha alcanzado su velocidad máxima.");
    }

    @Override
    protected void tomarCurvas() {
        System.out.println("El coche de calle toma las curvas con precaución.");
        for (int i = 0; i < 2; i++) {
            System.out.println("Curva " + (i + 1) + " tomada a baja velocidad.");
        }
    }

    @Override
    protected void frenar() {
        System.out.println("El coche de calle frena suavemente...");
        for (int i = 3; i > 0; i--) {
            System.out.println("Velocidad actual: " + (i * 10) + " km/h");
        }
        System.out.println("El coche de calle ha frenado completamente.");
    }
}
```

}

Ejemplo videojuego de coches {

```
class CocheDeCarreras extends CarreraCoche {

  @Override
  protected void acelerar() {
    System.out.println("El coche de carreras está acelerando...");
    int velocidad = 0;
    while (velocidad < 200) {
      velocidad += 50;
      System.out.println("Velocidad actual: " + velocidad + " km/h");
    }
    System.out.println("El coche de carreras ha alcanzado su velocidad máxima.");
  }

  @Override
  protected void tomarCurvas() {
    System.out.println("El coche de carreras toma la curva a gran velocidad.");
    for (int i = 0; i < 3; i++) {
      System.out.println("Curva " + (i + 1) + " tomada a " + (150 - i * 20) + " km/h.");
    }
  }

  @Override
  protected void frenar() {
    System.out.println("El coche de carreras está frenando...");
    int velocidad = 200;
    while (velocidad > 0) {
      velocidad -= 50;
      System.out.println("Velocidad actual: " + velocidad + " km/h");
    }
    System.out.println("El coche de carreras ha frenado completamente.");
  }
}
```

}

Ejemplo videojuego de coches {

```
class CocheDeportivo extends CarreraCoche {

  @Override
  protected void acelerar() {
    System.out.println("El coche deportivo está acelerando...");
    for (int i = 0; i < 5; i++) {
      System.out.println("Velocidad actual: " + (i * 20) + " km/h");
    }
    System.out.println("El coche deportivo ha alcanzado su velocidad máxima.");
  }

  @Override
  protected void tomarCurvas() {
    System.out.println("El coche deportivo toma la curva con precisión.");
    for (int i = 0; i < 3; i++) {
      System.out.println("Curva " + (i + 1) + " completada.");
    }
  }

  @Override
  protected void frenar() {
    System.out.println("El coche deportivo está frenando...");
    for (int i = 5; i > 0; i--) {
      System.out.println("Velocidad actual: " + (i * 20) + " km/h");
    }
    System.out.println("El coche deportivo ha frenado completamente.");
  }
}
```

}

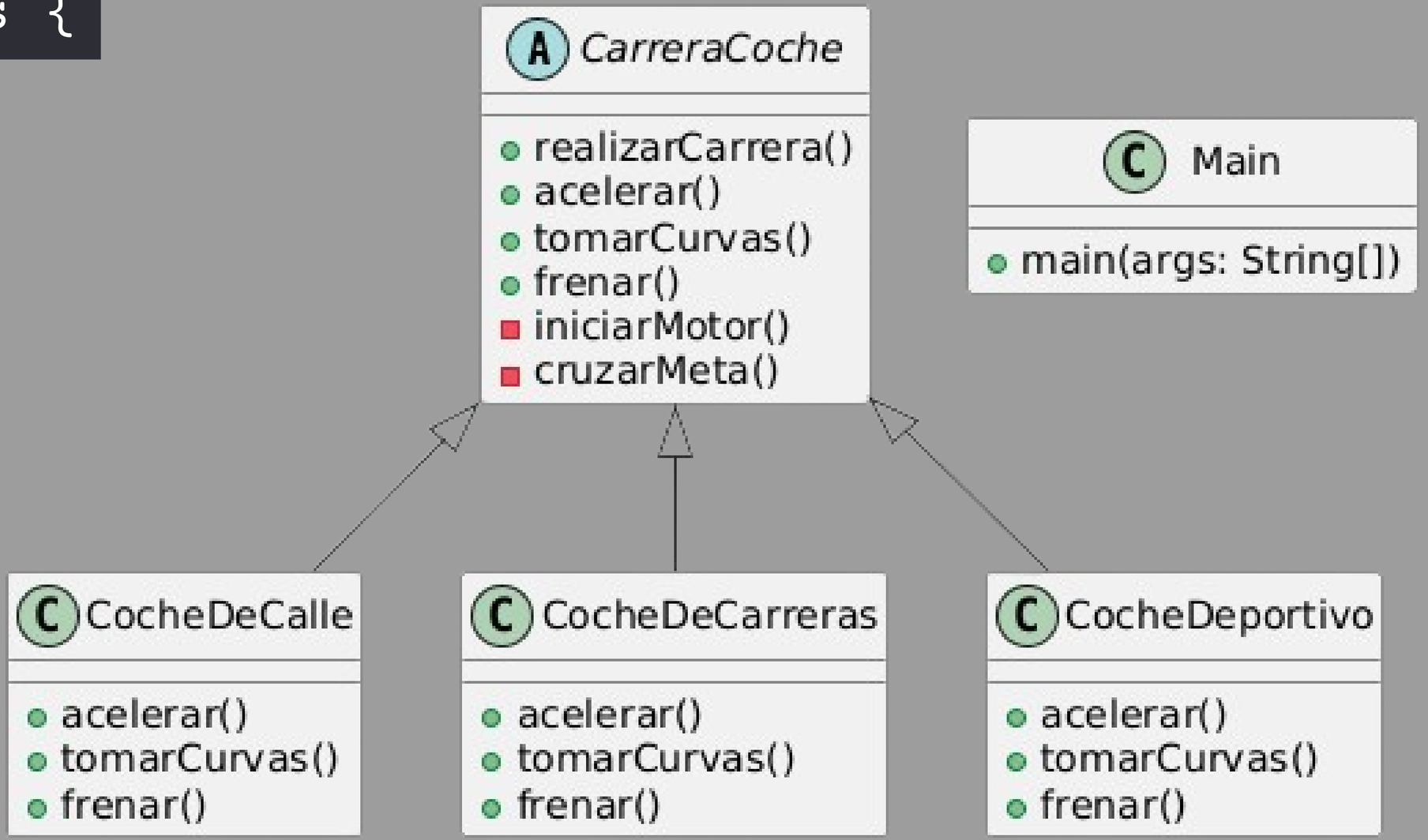
Ejemplo videojuego de coches {

```
public class Main {
  public static void main(String[] args) {
    // Crear instancias de los diferentes tipos de coches
    CarreraCoche cocheDeportivo = new CocheDeportivo();
    CarreraCoche cocheDeCarreras = new CocheDeCarreras();
    CarreraCoche cocheDeCalle = new CocheDeCalle();

    // Realizar carreras con diferentes coches
    System.out.println("Carrera con coche deportivo:");
    cocheDeportivo.realizarCarrera();

    System.out.println("\nCarrera con coche de carreras:");
    cocheDeCarreras.realizarCarrera();

    System.out.println("\nCarrera con coche de calle:");
    cocheDeCalle.realizarCarrera();
  }
}
```

Conclusión:
Este código ilustra cómo el patrón Template Method permite la definición de un algoritmo en una clase base, mientras que las subclases pueden personalizar partes específicas del algoritmo.

}

Ejemplo videojuego coches {



}

```
Gracias!!! {

}
```