Template Method Pattern

Introducción y nombre

Template Method. De Comportamiento. Define una estructura algorítmica.

Intención

Escribe una clase abstracta que contiene parte de la lógica necesaria para realizar su finalidad. Organiza la clase de tal forma que sus métodos concretos llaman a un método abstracto donde la lógica buscada tendría que aparecer. Facilita la lógica buscada en métodos de subclases que sobrescriben a los métodos abstractos. Define un esqueleto de un algoritmo, delegando algunos pasos a las subclases. Permite redefinir parte del algoritmo sin cambiar su estructura.

También conocido como

Método plantilla.

Motivación

Usando el Template Method, se define una estructura de herencia en la cual la superclase sirve de plantilla ("Template" significa plantilla) de los métodos en las subclases. Una de las ventajas de este método es que evita la repetición de código, por tanto la aparición de errores.

Solución

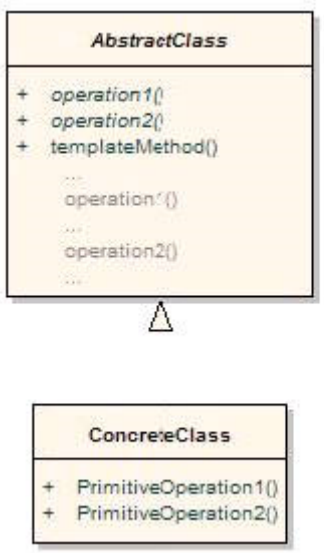
Se debe utilizar este patrón cuando:

Se quiera factorizar el comportamiento común de varias subclases.

Se necesite implementar las partes fijas de un algoritmo una sola vez y dejar que las subclases implementen las partes variables.

Se busque controlar las ampliaciones de las subclases, convirtiendo en métodos plantillas aquéllos métodos que pueden ser redefinidos.

Diagrama UML



Participantes

AbstractTemplate o AbstractClass: implementa un método plantilla que define el esqueleto de un algoritmo y define métodos abstractos que deben implementar las subclases concretas

TemplateConcreto o ConcreteClass: implementa los métodos abstractos para realizar los pasos del algoritmo que son específicos de la subclase.

Colaboraciones

Las clases concretas confían en que la clase abstracta implemente la parte fija del algoritmo.

Consecuencias

Favorece la reutilización del código.

Lleva a una estructura de control invertido: la superclase base invoca los métodos de las subclases.

Implementación

Una clase ClaseAbstracta proporciona la guía para forzar a los programadores a sobrescribir los métodos abstractos con la intención de proporcionar la lógica que rellena los huecos de la lógica de su método plantilla.

Los métodos plantilla no deben redefinirse. Los métodos abstractos deben ser protegidos (accesible a las subclases pero no a los clientes) y abstractos. Se debe intentar minimizar el número de métodos abstractos a fin de facilitar la implementación de las subclases.

Código de muestra

Como ejemplo, imaginemos una empresa que posee socios, clientes, empleados, etc. Cuando se les solicite que se identifiquen, cada uno lo realizará de distinta manera: quizás un empleado tiene un legajo, pero un cliente tiene un numero de cliente, etc.

[code]
public abstract class Persona {

    private String nombre;
    private String dni;

```
    public Persona() {
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public String getDni() {
        return dni;
    }

    public void setDni(String dni) {
        this.dni = dni;
    }

    /*
     * Define el esquelo del algoritmo y luego las subclases deben
     * implementar los métodos: getIdentificacion() y getTipoId()
     */
    public String identificate() {
        String frase = "Me identifico con: ";
        frase = frase + getTipoId();
        frase = frase + ". El numero es: ";
        frase = frase + getIdentificacion();
        return frase;
    }

    protected abstract String getIdentificacion();

    protected abstract String getTipoId();
}


public class Socio extends Persona {

    private int numeroDeSocio;

    public Socio(int numeroDeSocio) {
        this.numeroDeSocio = numeroDeSocio;
    }

    @Override
    protected String getIdentificacion() {
        return String.valueOf(numeroDeSocio);
    }

    @Override
    protected String getTipoId() {
        return "numero de socio";
    }

    public int getNumeroDeSocio() {
        return numeroDeSocio;
    }

    public void setNumeroDeSocio(int numeroDeSocio) {
        this.numeroDeSocio = numeroDeSocio;
    }
}


public class Cliente extends Persona {
```

```
    private int numeroDeCliente;

    public Cliente(int numeroDeCliente) {
        this.numeroDeCliente = numeroDeCliente;
    }

    @Override
    protected String getIdentificacion() {
        return String.valueOf(numeroDeCliente);
    }

    @Override
    protected String getTipoId() {
        return "numero de cliente";
    }

    public int getNumeroDeCliente() {
        return numeroDeCliente;
    }

    public void setNumeroDeCliente(int numeroDeCliente) {
        this.numeroDeCliente = numeroDeCliente;
    }
}


public class Empleado extends Persona {

    private String legajo;

    public Empleado(String legajo) {
        this.legajo = legajo;
    }

    @Override
    protected String getIdentificacion() {
        return legajo;
    }

    @Override
    protected String getTipoId() {
        return "numero de legajo";
    }

    public String getLegajo() {
        return legajo;
    }

    public void setLegajo(String legajo) {
        this.legajo = legajo;
    }
}
```

Veamos el ejemplo en práctica:

```
                public static void main(String[] args) {
                Persona p = new Cliente(12121);
                System.out.println("El cliente dice: ");
                System.out.println(p.identificate());

                System.out.println("El empleado dice: ");
                p = new Empleado("AD 41252");
                System.out.println(p.identificate());

                System.out.println("El socio dice: ");
```

```
                p= new Socio(46232);
                System.out.println(p.identificate());
        }
```

El resultado por consola es:
El cliente dice:
Me identifico con: numero de cliente. El numero es: 12121

El emplado dice:
Me identifico con: numero de legajo. El numero es: AD 41252

El socio dice:
Me identifico con: numero de socio. El numero es: 46232

[/code]

Cuándo utilizarlo

Este patrón se vuelve de especial utilidad cuando es necesario realizar un algoritmo que sea común para muchas clases, pero con pequeñas variaciones entre una y otras. En este caso, se deja en las subclases cambiar una parte del algoritmo.