Patrones de diseño II

UNJu - Programación Orientada a Objetos

Ing. José Zapana

Clasificación

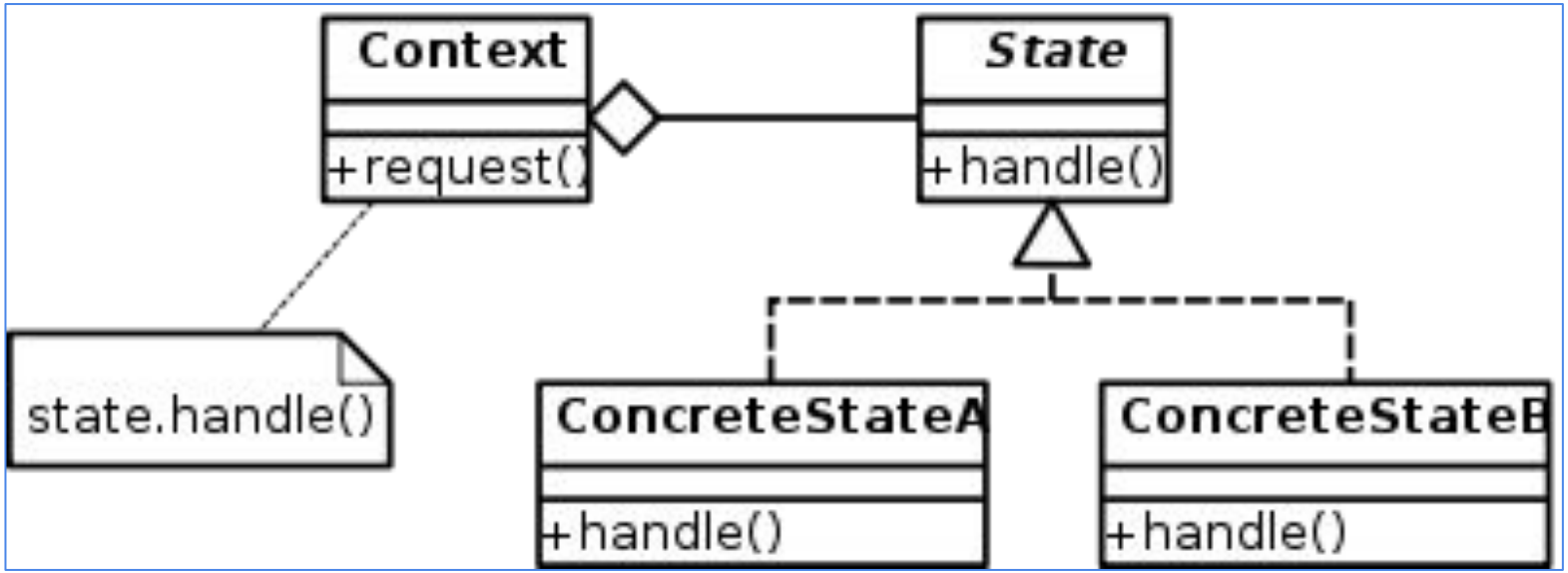
- Creacionales
  - Builder
  - Singleton
  - Abstract Factory, otros
- Estructurales
  - Decorator
  - DAO
  - Service Layer
  - otros
- Comportamiento
  - State
  - Strategy
  - Template Method
  - otros

Patrón State

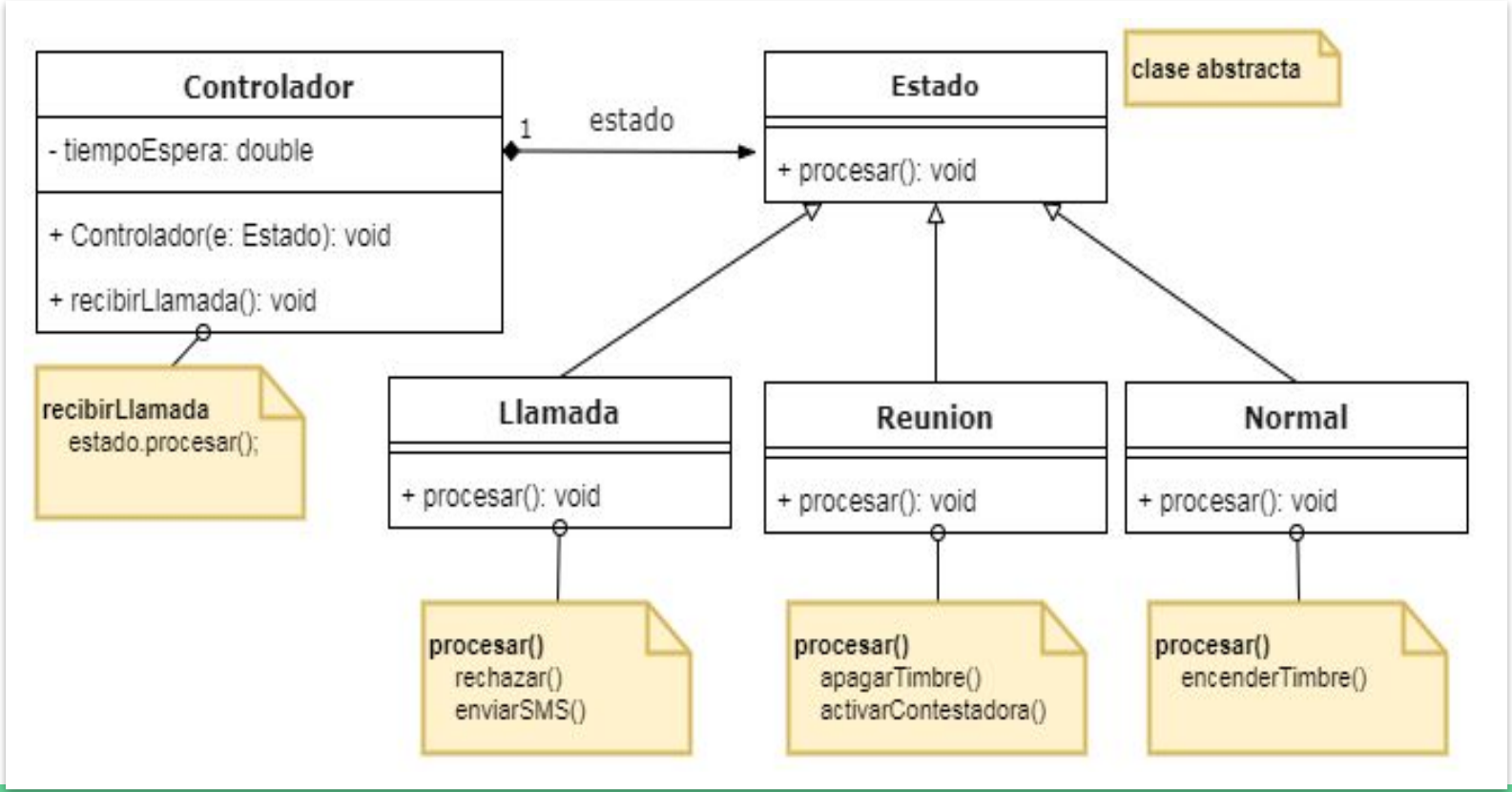
- Objetivo:
  - Modificar el comportamiento de un objeto cuando su estado interno se modifica.
  - Externamente parecería que la clase del objeto ha cambiado.
- Problema:
  - Muchas veces el comportamiento de un objeto depende del estado en el que se encuentre.
  - En la programación procedural se suelen utilizar enumerativos y sentencias condicionales.
  - El código no escala.
  - En objetos tenemos delegación y polimorfismo.
- Solución:
  - Desacoplar el estado interno del objeto en una jerarquía de clases.
  - Cada clase de la jerarquía representa un estado en el que puede estar el objeto.
  - Todos los mensajes del objeto que dependan de su estado interno son delegados a las clases concretas de la jerarquía (polimorfismo).
- Consecuencias:
  - Localiza el comportamiento relacionado con cada estado.
  - Las transiciones entre estados son explícitas.
  - En el caso que los estados no tengan variables de instancia pueden ser compartidos. Se pueden implementar como singletons, donde múltiples objetos en el mismo sistema pueden compartir el estado ya que serían cajas de comportamiento puro.

Patrón State - Estructura

- Estructura General



Patrón State - Ejemplo



Implementación del ejemplo

```java
public abstract class Estado {

    // Este es el "punto de entrada" común que usa el Controlador
    public abstract void procesar(Controlador controlador);

}
```

```java
public class Llamada extends Estado {

    @Override
    public void procesar(Controlador controlador) {
        // Lógica cuando ya estoy en otra llamada:
        rechazar();
        enviarSMS();
        System.out.println("Estado: EN LLAMADA. Se rechazó la nueva llamada.");
    }

    private void rechazar() {
        System.out.println("Llamada entrante rechazada automáticamente.");
    }

    private void enviarSMS() {
        System.out.println("Enviando SMS:'Estoy llamada, te devuelvo en unos
minutos'.");
    }
}
```

```java
public class Normal extends Estado {

    @Override
    public void procesar(Controlador controlador) {
        // Comportamiento esperado cuando el teléfono está en estado normal:
        encenderTimbre();
        System.out.println("Estado: NORMAL. Recibiendo llamada normalmente.");
    }

    private void encenderTimbre() {
        System.out.println("Timbre encendido.");
    }
}
```

```java
public class Reunion extends Estado {

    @Override
    public void procesar(Controlador controlador) {
        // Lógica cuando esta en reunión:
        apagarTimbre();
        activarContestadora();
        System.out.println("Estado: REUNIÓN. No interrumpir.");
    }

    private void apagarTimbre() {
        System.out.println("Timbre apagado (silencio).");
    }

    private void activarContestadora() {
        System.out.println("Contestadora activada.");
    }
}
```

Implementación del ejemplo II

```java
public class Controlador {
    private double tiempoEspera;
    private Estado estado;

    public Controlador(Estado estadoInicial) {
        this.estado = estadoInicial;
        this.tiempoEspera = 0.0;
    }

    // Procesa una llamada
    public void recibirLlamada() {
        estado.procesar(this);
    }
}
```

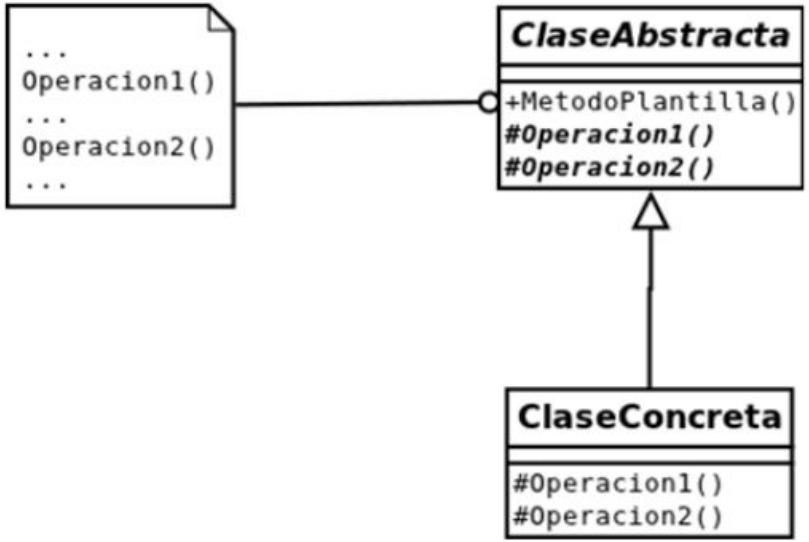
```java
public class DemoMain {

    public static void main(String[] args) {
        // Caso 1: Estoy libre (estado NORMAL)
        Controlador telefono = new Controlador(new Normal());
        telefono.recibirLlamada();
        System.out.println();

        // Caso 2: Estoy en reunión
        telefono.setEstado(new Reunion());
        telefono.recibirLlamada();
        System.out.println();

        // Caso 3: Estoy hablando por teléfono con otra persona
        telefono.setEstado(new Llamada());
        telefono.recibirLlamada();
        System.out.println();
    }
}
```

Template Method

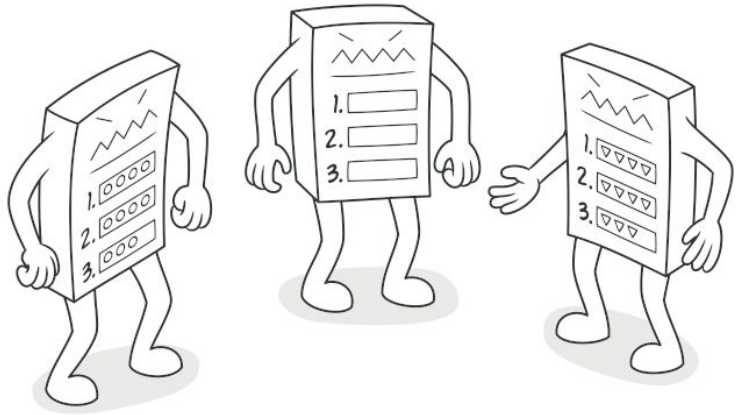
- Propósito
  - Establecer el esqueleto de un algoritmo en una clase base, dejando que las subclases implementen los pasos específicos. Así se garantiza que el flujo general del proceso se mantenga constante, mientras se permite flexibilidad en los detalles.
- Motivación
  - Reutilización de código común en la clase base.
  - Controlar el flujo
  - Permitir la personalización de algunos pasos del proceso en subclases.
  - Mantener la consistencia en la estructura del algoritmo.
  - Aplicar el principio Hollywood: “Don’t call us, we’ll call you”. La clase base llama a los métodos definidos por las subclases, no al revés.
  - Aplica el principio Open/Close

Estructura del patrón Template Method

- **Diagrama UML**



- **Representación**



Aplicabilidad

- Utiliza el patrón Template Method cuando quiera permitir a sus clientes que extiendan únicamente pasos particulares de un algoritmo, pero no todo el algoritmo o su estructura.
- Permite convertir un algoritmo monolítico en una serie de pasos individuales que se pueden extender fácilmente con subclases, manteniendo intacta la estructura definida en una superclase.

Pasos para implementarlo

1. Analizar el algoritmo objetivo para ver si puedes dividirlo en pasos. Considera qué pasos son comunes a todas las subclases y cuáles siempre serán únicos.
2. Crea la clase base abstracta y declara el método plantilla y un grupo de métodos abstractos que representen los pasos del algoritmo.
3. Los pasos pueden ser abstractos, sin embargo algunos pueden estar implementados.
4. Para cada variación del algoritmo, crea una nueva subclase concreta. Ésta debe implementar todos los pasos abstractos, pero también puede sobrescribir algunos de los opcionales.

Ejemplo Template Method

```java
public abstract class DataExporter {
    // Método plantilla: define el flujo general
    public final void export() {
        connect();
        fetchData();
        transformData();
        saveData();
        disconnect();
    }

    protected abstract void connect();
    protected abstract void fetchData();
    protected abstract void transformData();
    protected abstract void saveData();

    // Paso común
    protected void disconnect() {
        System.out.println("Desconectando...");
    }
}
```

```java
public class CsvDataExporter extends DataExporter {
    @Override
    protected void connect() {
        System.out.println("Conectando a fuente de datos CSV...");
    }

    @Override
    protected void fetchData() {
        System.out.println("Leyendo datos desde archivo CSV...");
    }

    @Override
    protected void transformData() {
        System.out.println("Transformando datos para formato CSV...");
    }

    @Override
    protected void saveData() {
        System.out.println("Guardando datos en archivo CSV...");
    }
}
```

Ejemplo Template Method II

```java
public class DatabaseDataExporter extends DataExporter {
    @Override
    protected void connect() {
        System.out.println("Conectando a base de datos...");
    }

    @Override
    protected void fetchData() {
        System.out.println("Ejecutando consulta SQL...");
    }

    @Override
    protected void transformData() {
        System.out.println("Normalizando datos para inserción...");
    }

    @Override
    protected void saveData() {
        System.out.println("Insertando datos en tabla...");
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        DataExporter exporter1 = new CsvDataExporter();
        exporter1.export();

        DataExporter exporter2 = new DatabaseDataExporter();
        exporter2.export();
    }
}
```

Referencias

- Why to use Service Layer in Spring MVC,
- Patterns of Enterprise Application Architecture
- Inyección de dependencias
- Implementando Contextos Java e Inyección de Dependencia (CDI)
- Data Transfer Object (DTO)
- https://refactoring.guru/es/design-patterns/template-method