Clasificación de los Patrones de Diseño

Programación Orientada a Objetos

San Salvador de Jujuy

UNJu – Facultad de Ingeniería
Ing. José Zapana

Clasificación

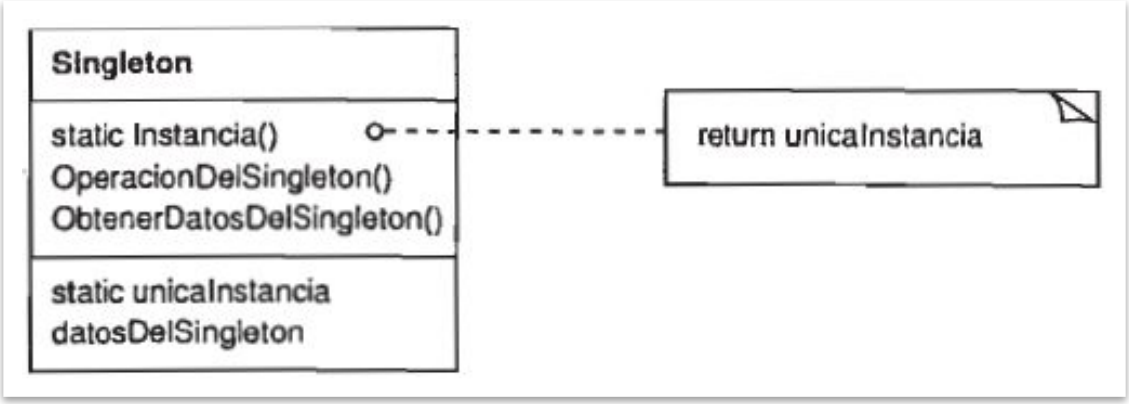
- **Creacionales**
  - Builder
  - Singleton
  - Abstract Factory, otros
- **Estructurales**
  - Decorator
  - DAO
  - Service Layer, otros
- **Comportamiento**
  - State
  - Strategy
  - otros

Patrón Singleton (único)

- **Propósito**
  - Garantiza que una clase tenga solo una instancia, y proporciona un punto de acceso global a ella.
- **Motivación**
  - Cola de impresión, sistema de ficheros, gestor de ventanas
  - Una solución es hacer que sea la propia clase la responsable de su única instancia, y puede proporcionar un modo de acceder a la instancia.
- **Aplicabilidad**
  - Cuando deba haber exactamente una instancia de una clase, y ésta deba ser accesible a los clientes desde un punto de acceso conocido.
  - La única instancia debería ser extensible mediante herencia, y los clientes deberían ser capaces de usar una instancia extendida sin modificar su código.

Singleton - Estructura

- Estructura y Ejemplo



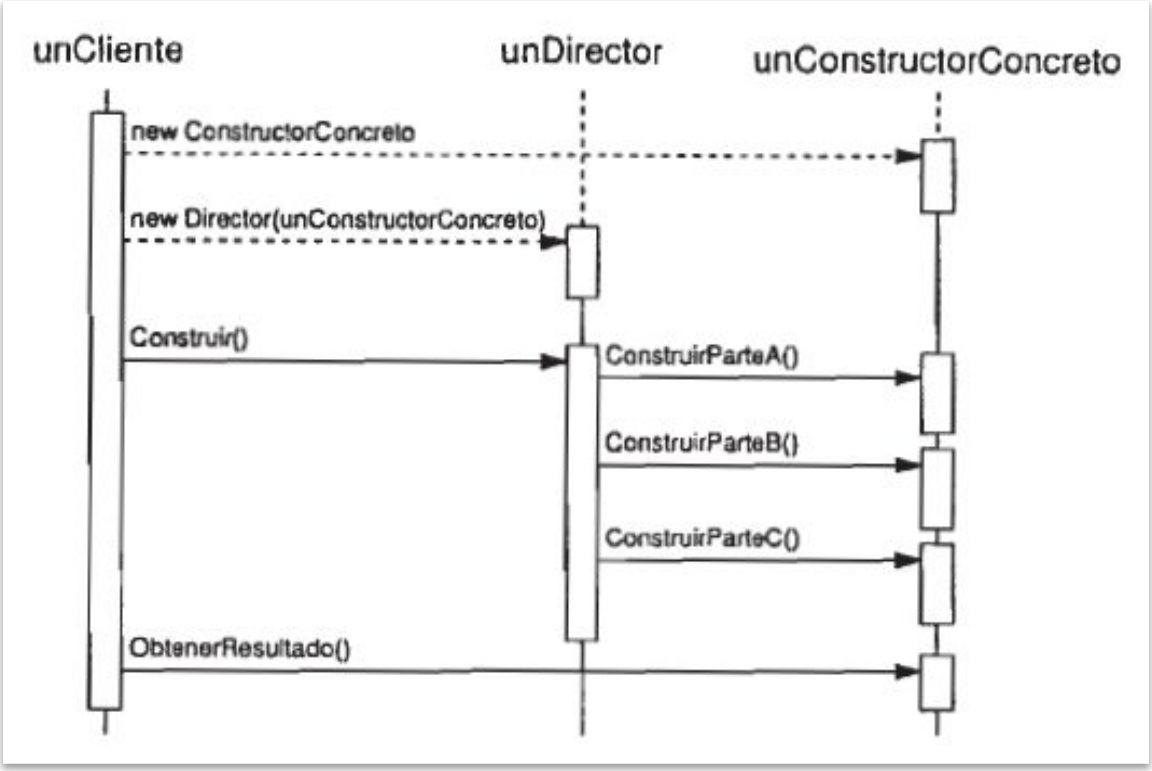

```
public class ApplicationProperties {
    private static ApplicationProperties instance;

    private ApplicationProperties(){

    }

    public static ApplicationProperties getInstance() {
        if (instance == null)
            instance = new ApplicationProperties();
        return instance;
    }
```

Patrón Builder

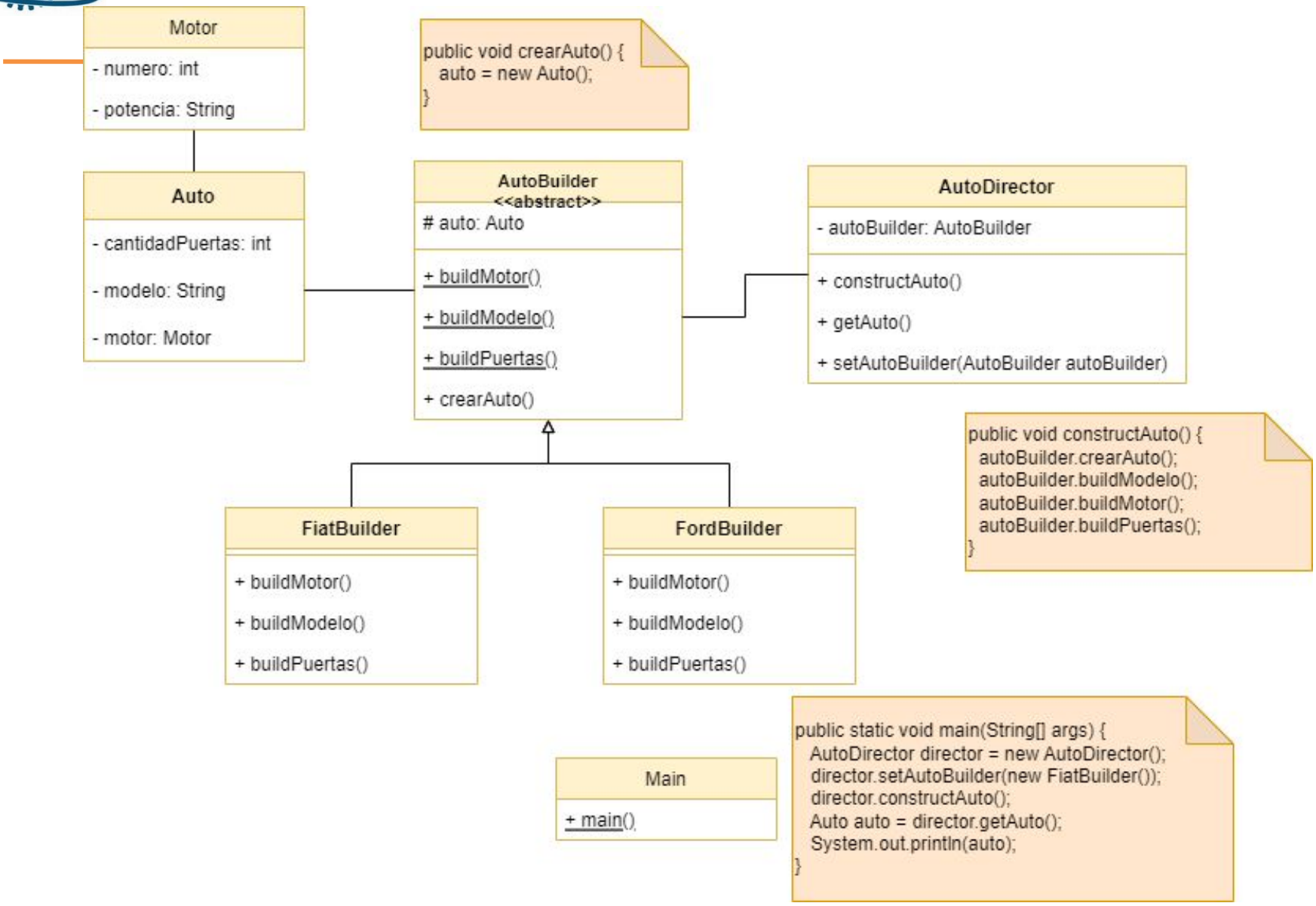
- **Propósito**
  - Separa la construcción de un objeto complejo de su representación, de forma que el mismo proceso de construcción pueda crear diferentes representaciones.
- **Descripción**
  - Permite la creación de un objeto complejo, a partir de una variedad de partes que contribuyen individualmente a la creación y ensamblación del objeto mencionado. Por otro lado, centraliza el proceso de creación en un único punto, de tal forma que el mismo proceso de construcción pueda crear representaciones diferentes.
  - Por ejemplo: la construcción de un objeto Computadora, se compondrá de otros muchos objetos, como puede ser un objeto PlacaDeSonido, Procesador, PlacaDeVideo, Gabinete, Monitor, etc.
- **Consecuencias**
  - Permite la variación interna de un producto:
  - Aísla el código de la construcción y representación: los clientes no necesitan saber nada de las clases que definen la estructura interna del producto.Dichas clases no aparecen en la interfaz del constructor.
  - Proporciona un control más fino sobre el proceso de construcción: Construye al objeto paso a paso con control del director.

Patrón Builder - Estructura

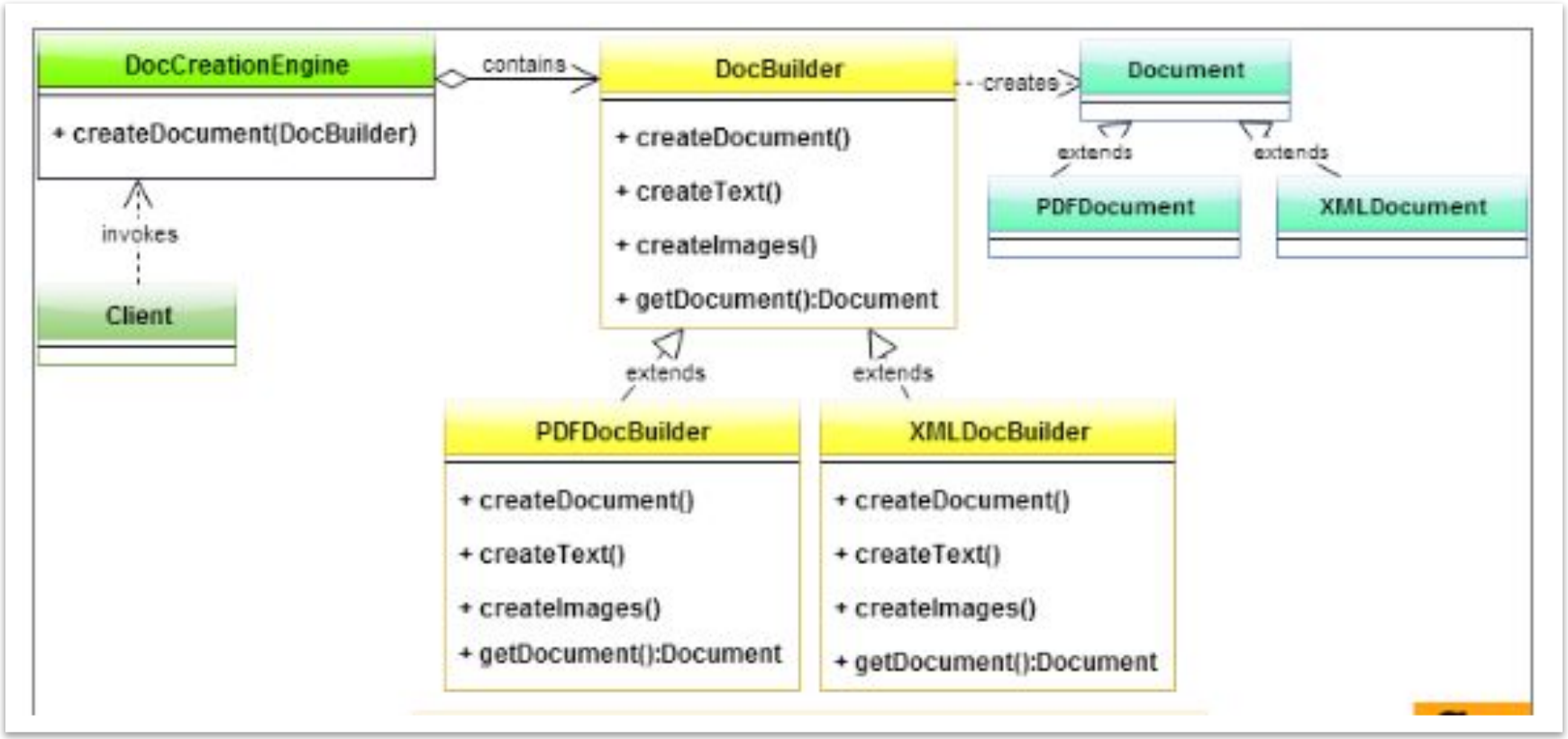
- Diagrama de secuencia



Ejemplo Patrón Builder



Patrón Builder - Ejemplo

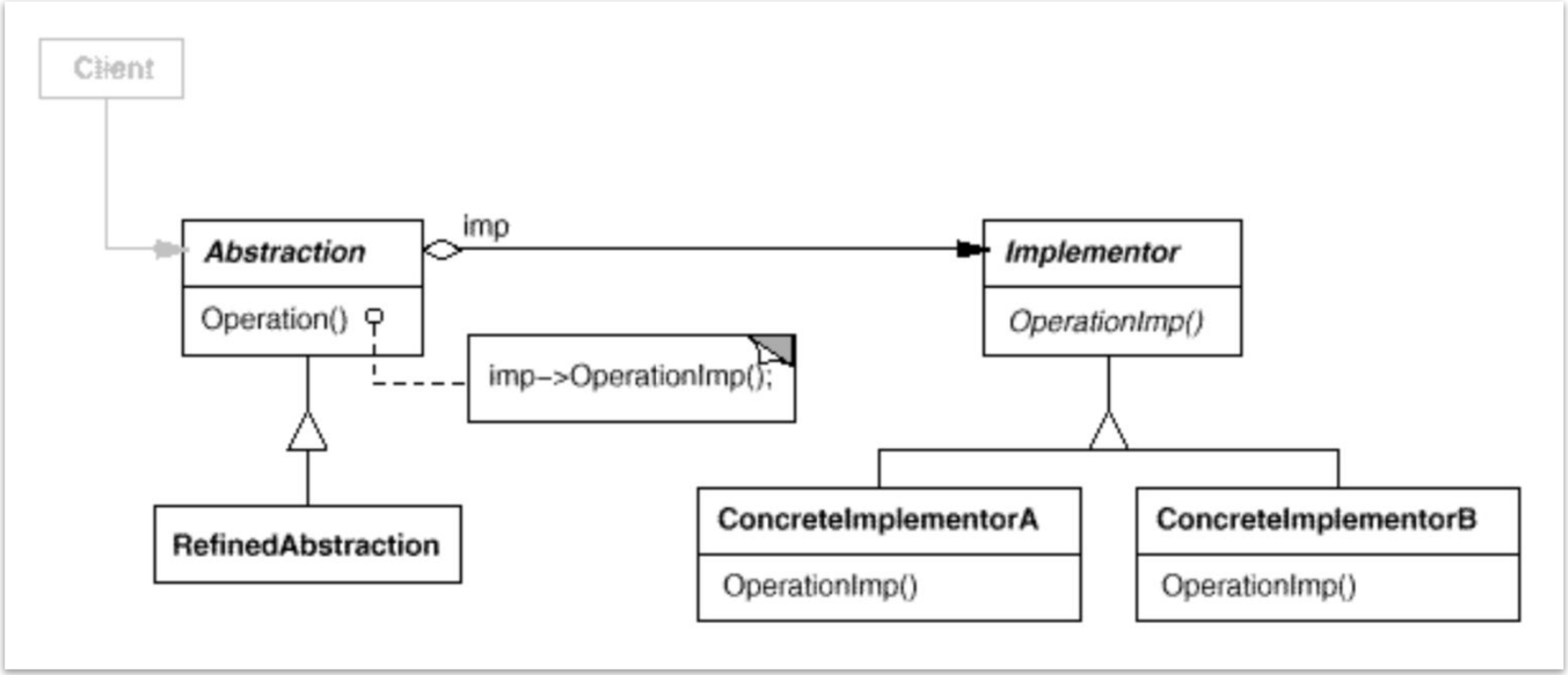


Patrón Abstract Factory

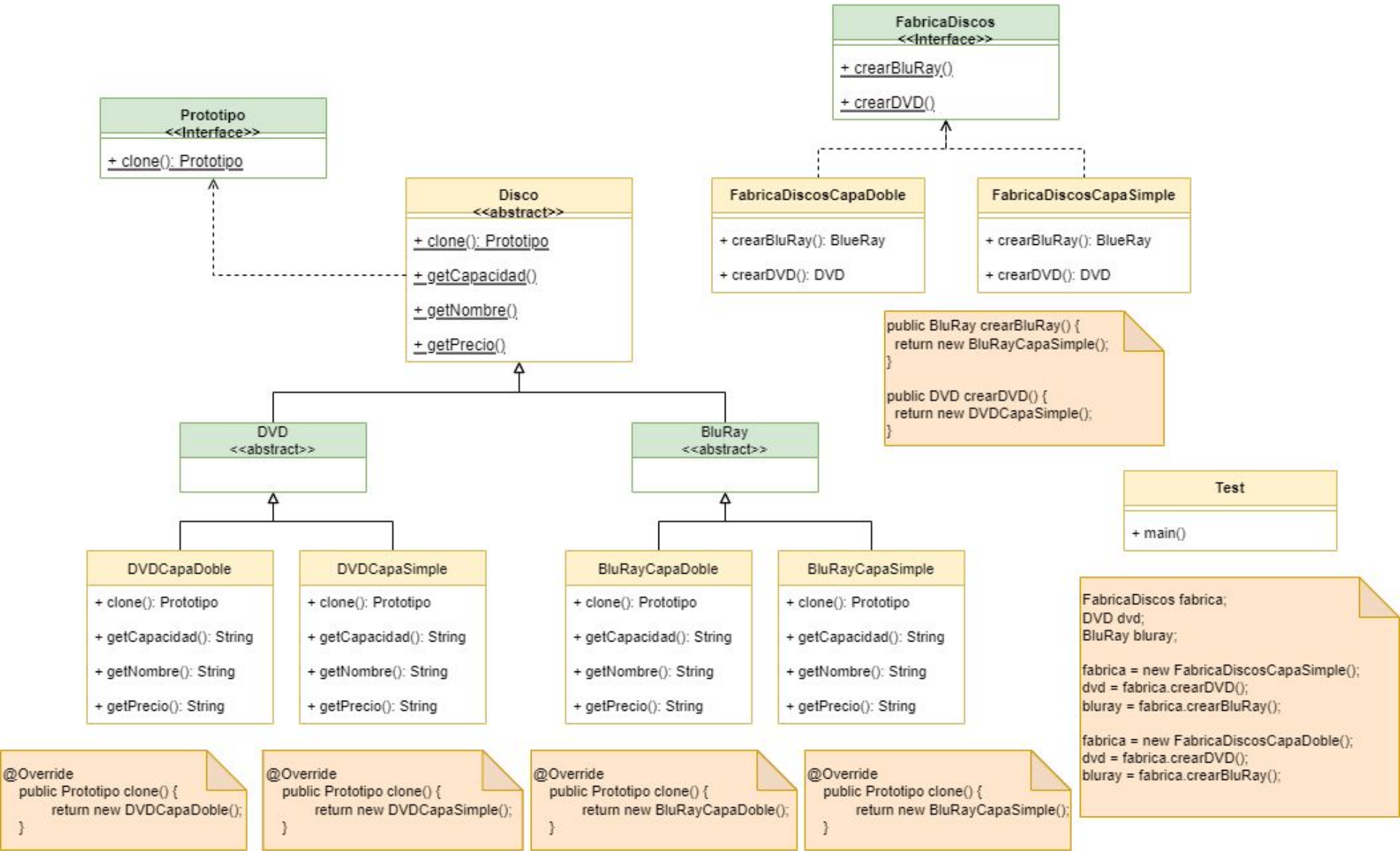
- **Propósito**
  - Proporciona una interfaz para crear familias de objetos relacionados o que dependen entre sí, sin especificar sus clases concretas.
- **Motivación**
  - Ejemplo crear familias de discos
- **Aplicabilidad**
  - Use este patrón cuando
    - Un sistema debe ser independiente de cómo se crean, componen y representan sus productos
    - Un sistema debe ser configurado con una familia de productos entre varias
    - Una familia de objetos producto relacionados está diseñada para ser usada conjuntamente, y es necesario hacer cumplir esa restricción
    - Quiere proporcionar una biblioteca de clases de productos, y sólo quiere revelar sus interfaces, no sus implementaciones

Patrón Abstract Factory - Estructura

- Estructura general

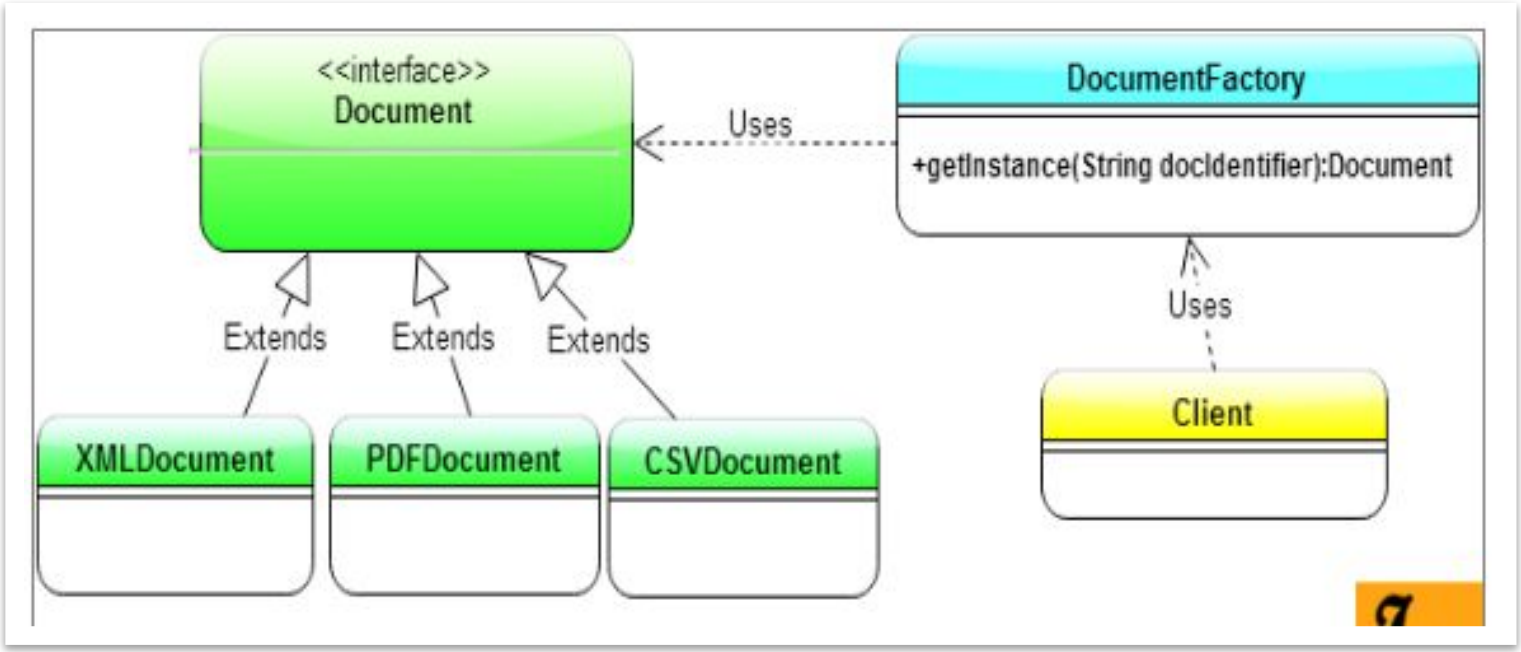


Ejemplo Patrón Abstract Factory



Patrón Abstract Factory - Ejemplo

- Ejemplo

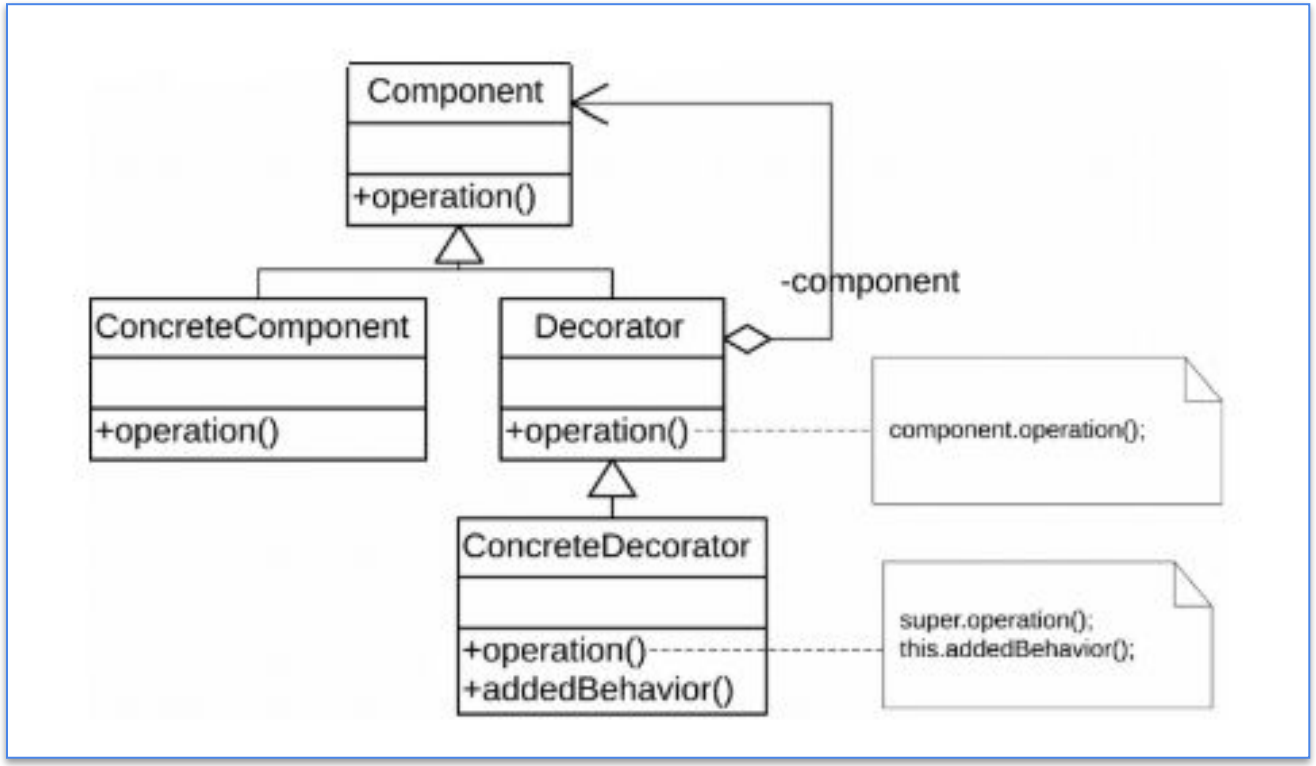


Patrón Decorator

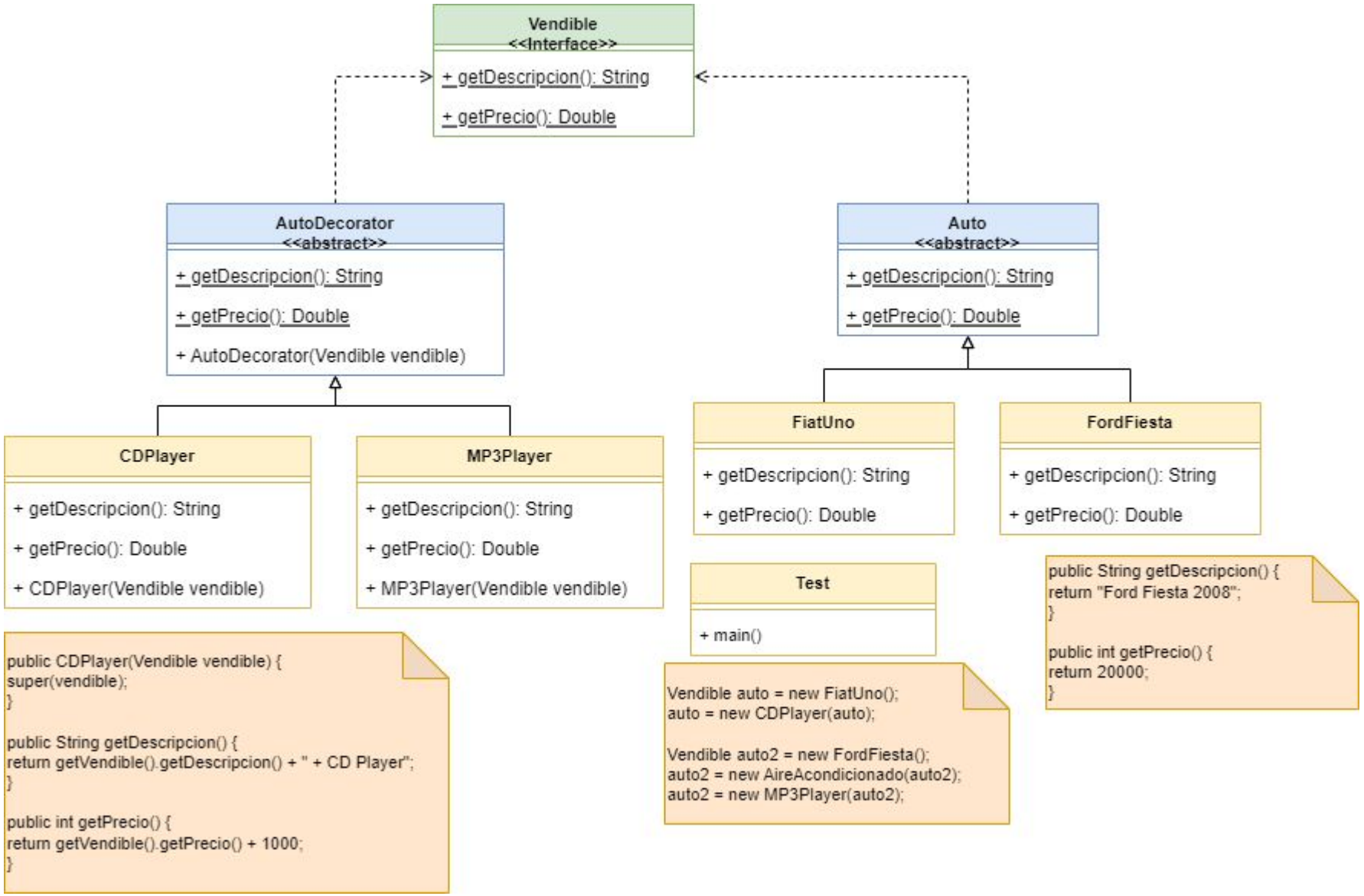
- **Objetivo**
  - Agregar comportamiento a un objeto dinámicamente y en forma transparente.
- **Problema**
  - Cuando queremos agregar comportamiento extra a algunos objetos de una clase puede usarse herencia. El problema es cuando necesitamos que el comportamiento se agregue o quite dinámicamente, porque en ese caso los objetos deberían “mutar de clase”. El problema que tiene la herencia es que se decide estáticamente.
- **Solución**
  - Definir un decorador (o “wrapper”) que agregue el comportamiento cuando sea necesario.
  - Prestar atención al polimorfismo-

Patrón Decorator - Estructura

- Estructura general



Ejemplo Decorator

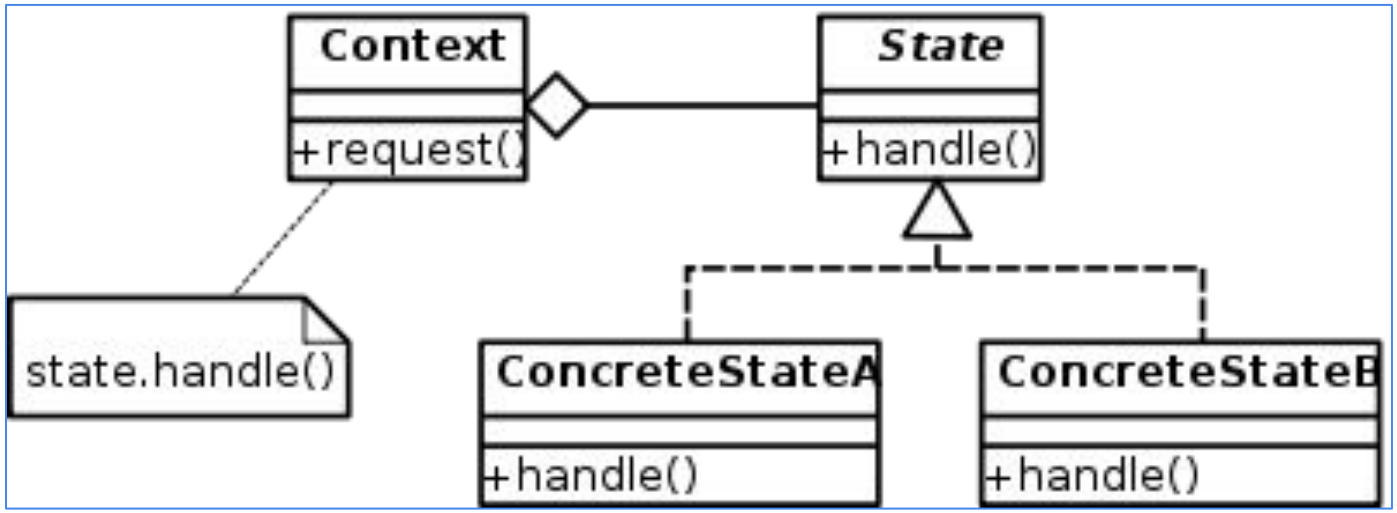


Patrón State

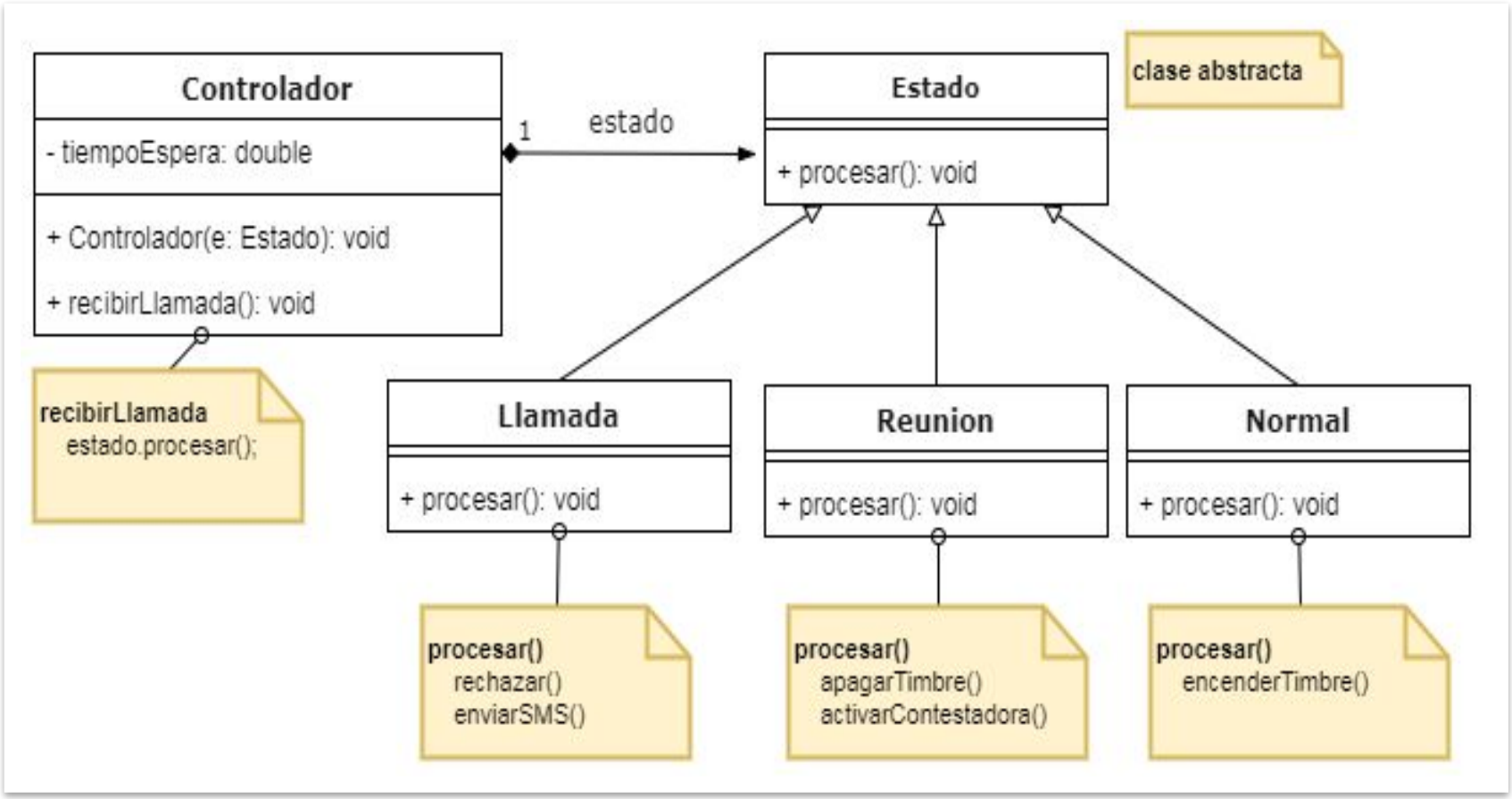
- **Objetivo:**
  - Modificar el comportamiento de un objeto cuando su estado interno se modifica.
  - Externamente parecería que la clase del objeto ha cambiado.
- **Problema:**
  - Muchas veces el comportamiento de un objeto depende del estado en el que se encuentre.
  - En la programación procedural 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

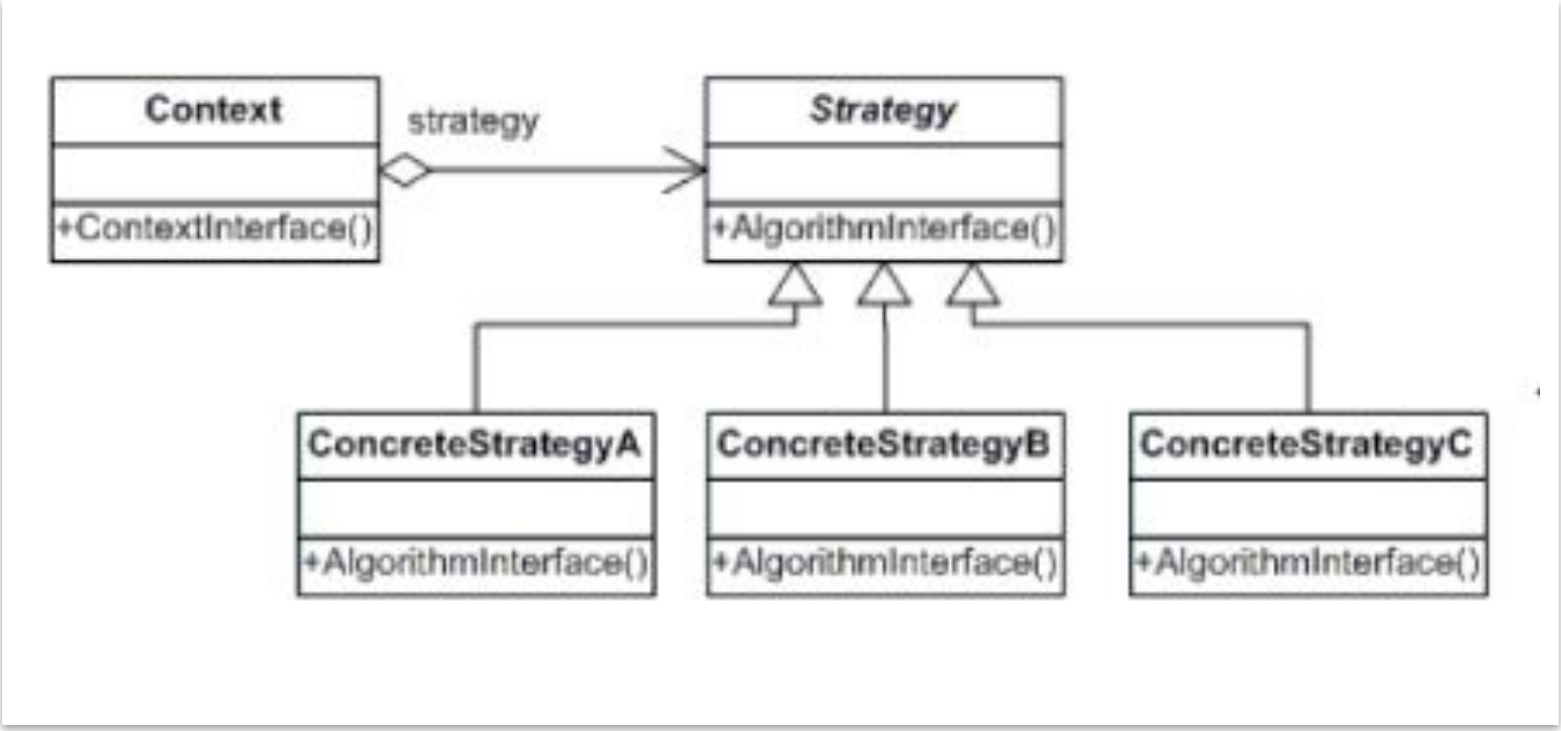


Patrón Strategy

- **Objetivo:**
  - Desacoplar un algoritmo del objeto que lo utiliza.
  - Permitir cambiar el algoritmo que un objeto utiliza en forma dinámica.
  - Brindar flexibilidad para agregar nuevos algoritmos que lleven a cabo una función determinada.
- **Problema:**
  - Existen muchos algoritmos para llevar a cabo una tarea.
  - No es deseable codificarlos todos en una clase y seleccionar cuál utilizar por medio de sentencias condicionales.
  - Cada algoritmo utiliza información propia. Colocar esto en los clientes lleva a tener clases complejas y difíciles de mantener.
  - Es necesario cambiar el algoritmo en forma dinámica.
- **Aplicabilidad:**
  - Muchas clases relacionadas difieren solo en su comportamiento. Las estrategias permiten configurar una clase con un determinado comportamiento de entre muchos posibles.
  - Se necesitan distintas variantes de un algoritmo
  - Un algoritmo usa datos que los clientes no deberían conocer. Evita exponer estructuras de datos complejas y dependientes del algoritmo.
  - Una clase define muchos comportamientos, y estos se representan como múltiples sentencias condicionales en sus operaciones.
- **Solución:**
  - Definir una familia de algoritmos, encapsulando a cada uno en un objeto.

Patrón Strategy - Estructura

- Estructura General



Strategy vs State

- Mismo diagrama de clases.
- Misma idea de delegación.
- Pero
  - El estado es privado del objeto, ningún otro objeto sabe de él.
  - Un State define una máquina de estados con sus transiciones.
  - Un strategy suele tener un único mensaje público

Referencias

- Builder Design Pattern in Java
- Factory Method Design Pattern in Java
- Catálogo de ejemplos en Java