Catálogo de Patrones de Diseño

75.07 | Algoritmos y Programación III

Pablo Rodríguez Massuh

Temario

- Elementos que componen un patrón.
- Clasificación de los patrones de diseño.
- Patrones Creacionales
  - Singleton
  - Multiton
  - Factory Method
  - Abstract Factory
- Patrones de Organización del trabajo
  - Command
- Patrones de control de Acceso
  - Proxy
  - Facade
- Patrones variación de servicios
  - Strategy
  - Template
  - State
- Patrones de extensión de servicios
  - Decorator
- Patrones de descomposición estructural
  - Composite

Elementos de un Patrón

- Nombre
- Problema
- Solución
- Consecuencias

Clasificaciones

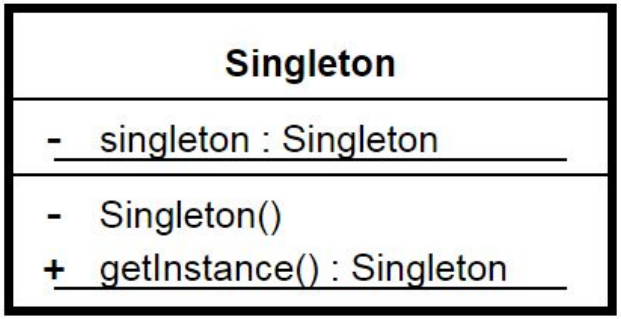
Los patrones se pueden clasificar según su intención

Creacionales	Organización del Trabajo	Control de Acceso
Variación de Servicios	Extensión de Servicios	Descomposición Estructural

Singleton

- Propósito:
  - Garantiza que una clase solo tenga una instancia y proporciona un punto de acceso global a ella
- Solución:
  - La propia clase es responsable de crear la única instancia.
  - Permite el acceso global a dicha instancia mediante un método de clase.
  - Declara el constructor de clase como privado para que no sea instanciable directamente.
- Consecuencias:
  - Acceso controlado a la instancia
  - Se logra el objetivo pero a cambio de ensuciar la clase
  - Se lo considera un patrón fácil pero intrusivo

Singleton - Diagrama




```
public class Singleton {
    private static Singleton INSTANCE = new Singleton();

    // El constructor privado no permite que se genere un constructor por defecto.
    // (con mismo modificador de acceso que la definición de la clase)
    private Singleton() {}

    public static Singleton getInstance() {
        return INSTANCE;
    }
}
```

Multiton (2..n)

- Propósito:
  - Garantiza que una clase solo tenga varias instancias conocidas, y proporciona un punto de acceso global a ellas
- Solución:
  - Se implementa igual al singleton pero con un mapa (identificador,instancia) en vez de con un atributo. El método getInsntance recibe el nombre de la instancia.
- Aplicabilidad:
  - Application loggers
    - Log de producción
    - Log de Debug

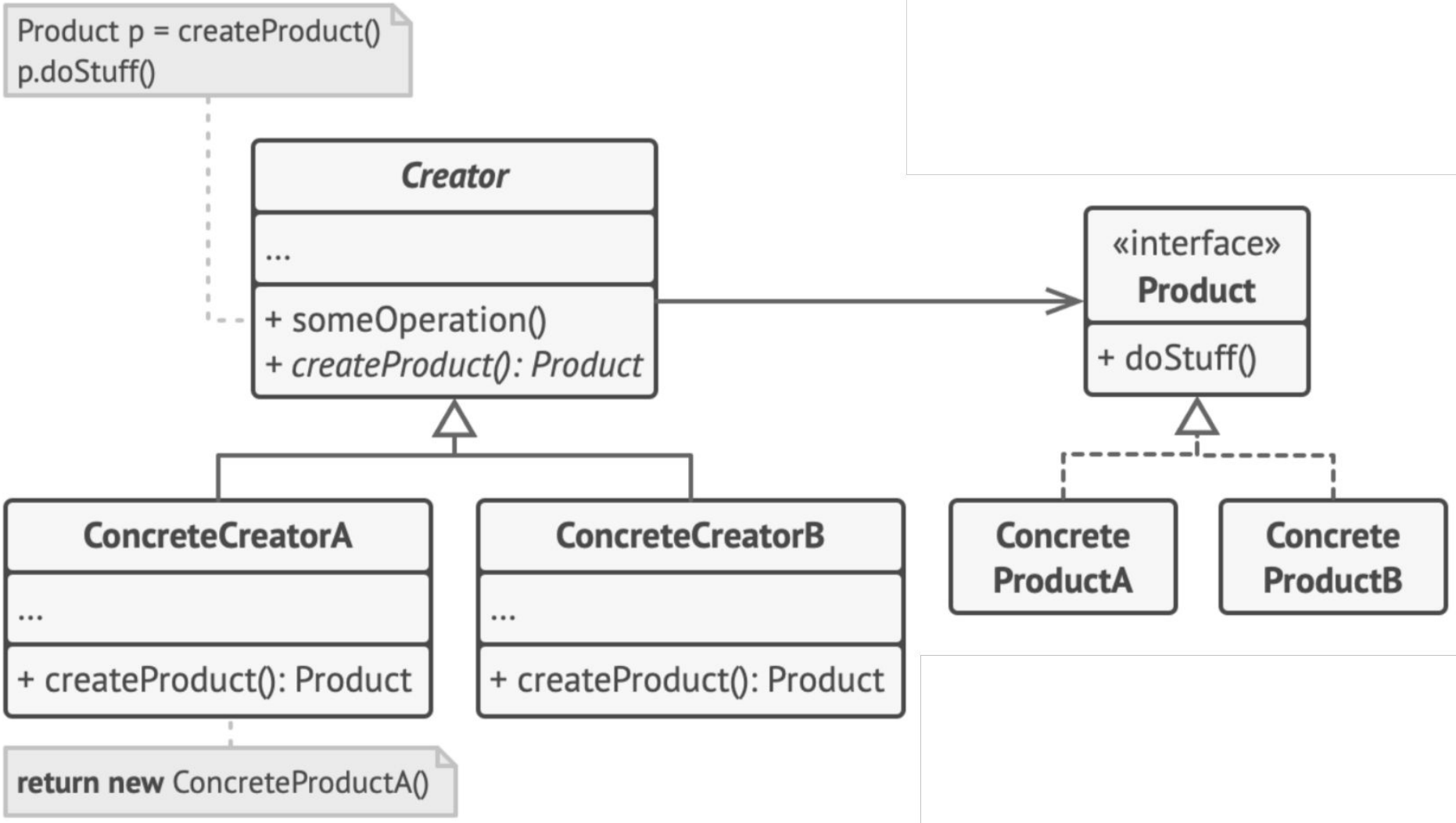
Factory Method

- Propósito:
  - Proporciona una interfaz para crear un objeto, pero delega en sus hijas la decisión de qué objeto instanciar.

- Solución:
  - Crear una clase Factory / Creator que sea abstracta que provea una interfaz común para la creación del objeto en cuestión.
  - Crear clases Factories que hereden de la clase Factory abstracta y que implementen EL método definido creando la instancia concreta
  - En la aplicación utilizar solo la clase FactoryAbstracta / Creator una vez determinada el objeto a crear

- Consecuencias:
  - Independencia de las clases concretas
  - Permite intercambiar el objeto creado de manera rápida y transparente

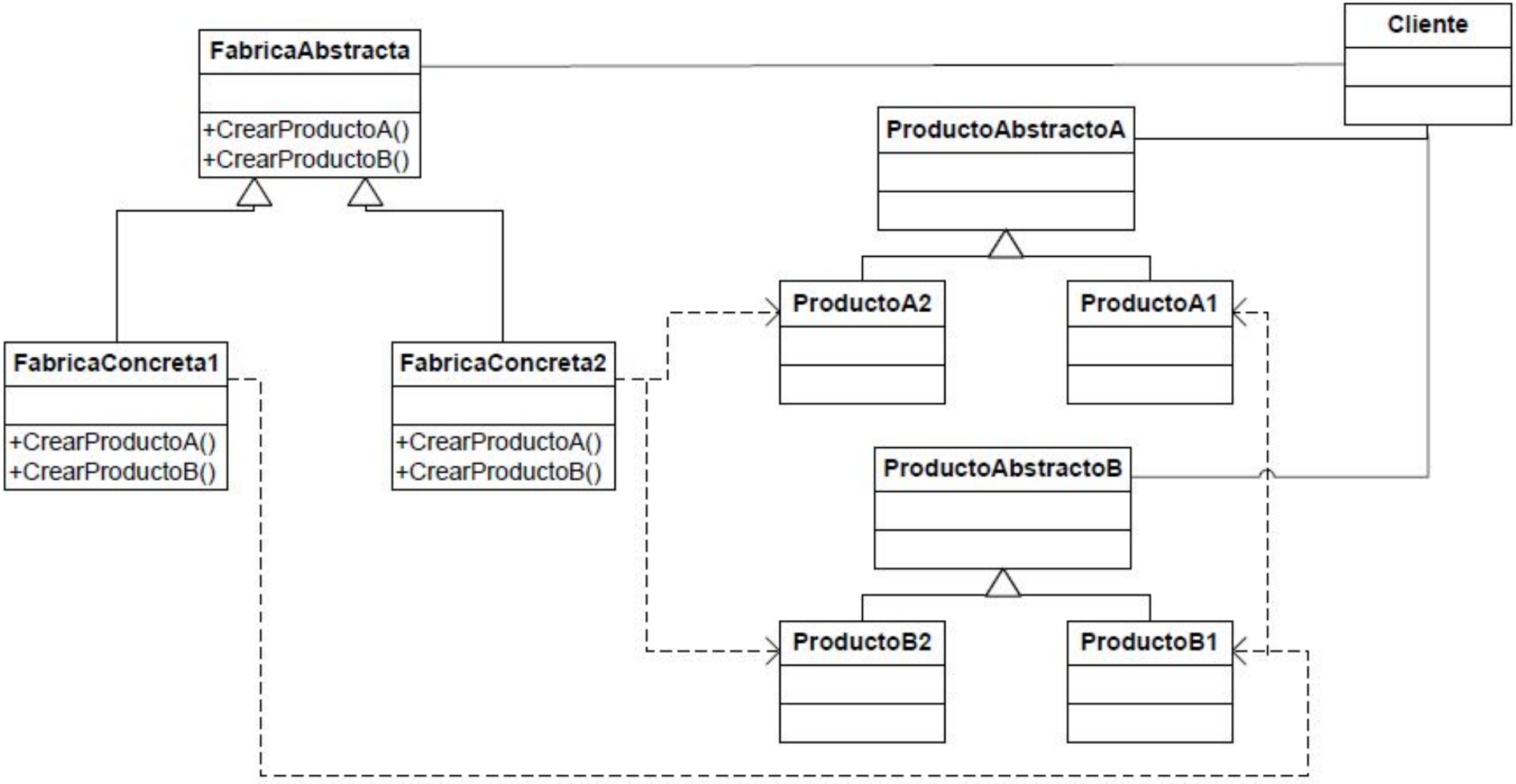
Factory Method- Diagrama



Abstract Factory

- Propósito:
  - Proporciona una interfaz para crear familias de objetos relacionados o que dependen entre si, sin especificar sus clases concretas
- Solución:
  - Crear una clase Factory que sea abstracta que provea una interfaz común de creación de las familias de objetos
  - Crear clases Factories que hereden de la clase Factory abstracta y que implementen los métodos definidos creando las instancias concretas
  - En la aplicación utilizar solo la clase FactoryAbstracta una vez determinada la familia de objetos a crear
- Consecuencias:
  - Independencia de las clases concretas
  - Permite intercambio de familias de objetos de manera rápida y transparente

Abstract Factory - Diagrama



Command

Propósito:

- Encapsula una petición en un objeto, permitiendo así parametrizar a los clientes con diferentes peticiones, hacer cola o llevar un registro de peticiones, y poder deshacer las operaciones. Desacopla el código que solicita un servicio del que lo presta.

Solución:

- Crear una clase abstracta o una interfaz con un solo método excecute()
- Cada clase descendiente implementará el método.
- Para invocar el método se instanciará una de las clases y se invocará al método excecute()

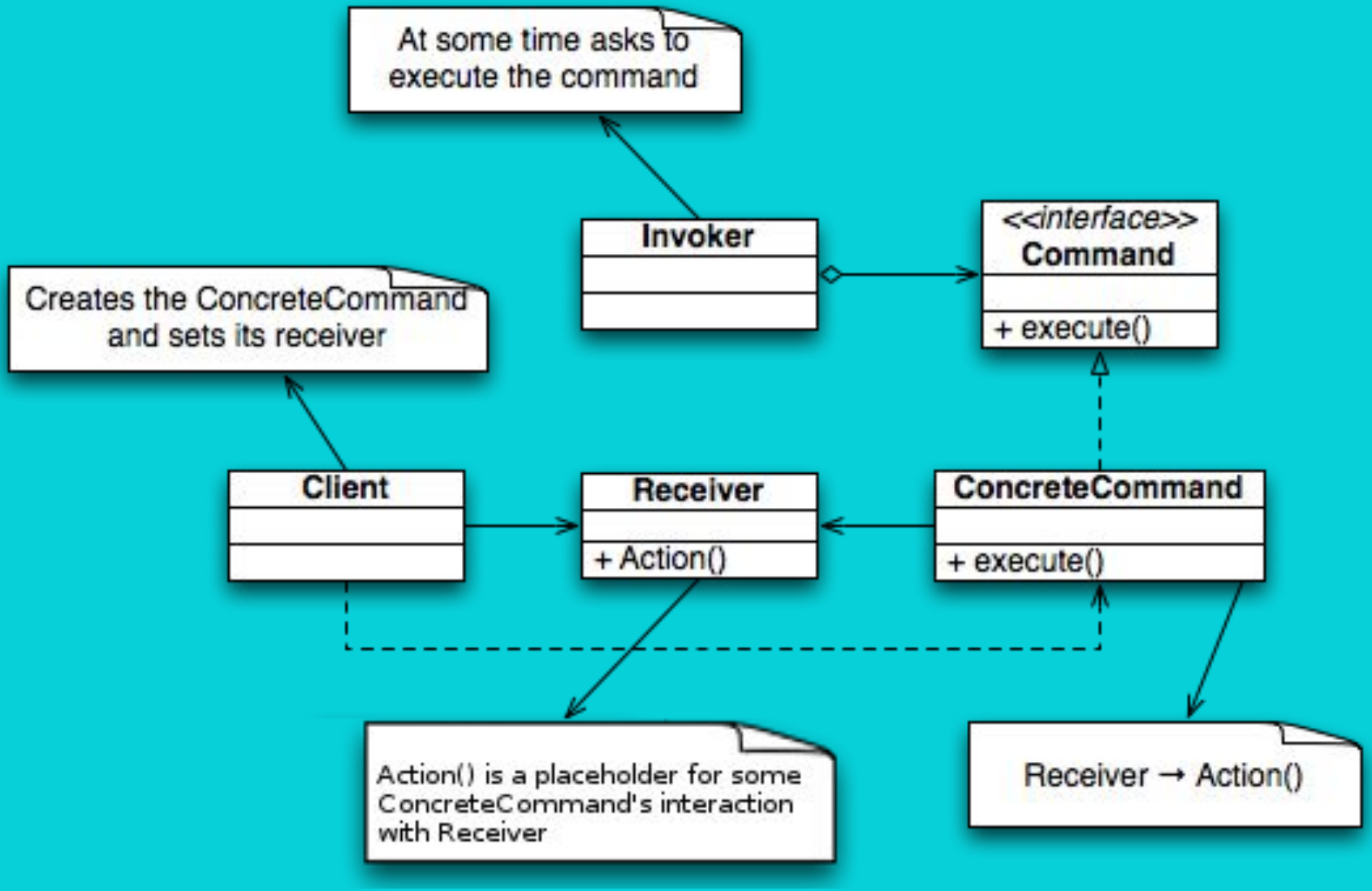
Motivación:

- Objetos como botones y menús que realizan una petición en respuesta a una entrada de usuario
- Transacciones

Consecuencias:

- Permite mantener referencias a métodos
- Desacopla el objeto que invoca la operación de aquel que sabe como realizarla
- Las órdenes son objetos de primera clase. Pueden ser manipulados y extendidos
- como cualquier otro objeto
- Se pueden ensamblar ordenes en una orden compuesta
- Es fácil añadir nuevas ordenes, ya que no hay que cambiar las clases existentes

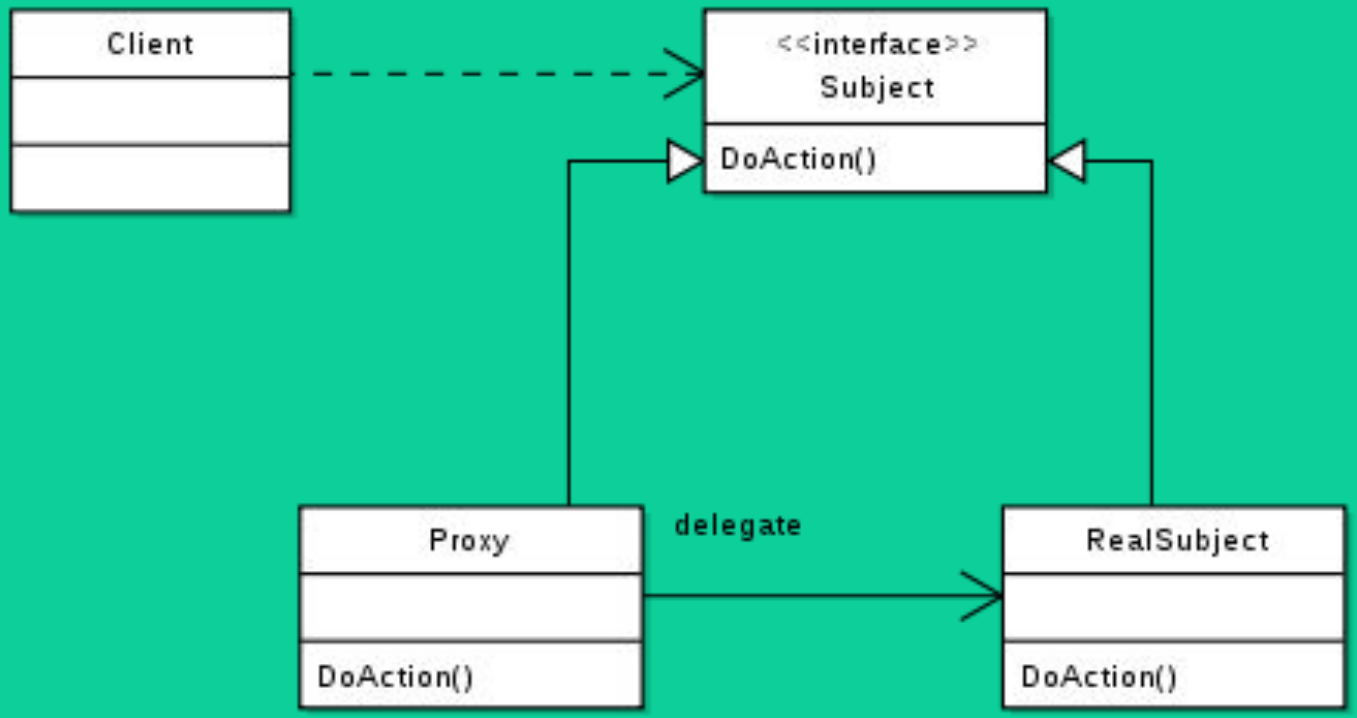
Command - Diagrama



Proxy

- Propósito:
  - Proporciona un representante o sustituto de otro objeto para controlar el acceso a este.
- Solución:
  - Crear una jerarquía en la que intervengan el objeto original y el objeto proxy
  - En el objeto proxy habrá una referencia al objeto original
  - Se redefinen todas las llamadas en el proxy incorporando código antes de derivarlas al objeto original
- Motivación:
  - Retrasar el coste de creación e inicialización hasta que realmente sea necesario (Ej., de archivos con imágenes)
- Consecuencias:
  - Posibilidad de agregar funcionalidad de manera transparente a la aplicación
  - Permite realizar optimizaciones, ocultar complejidad, establecer mecanismos de seguridad en los accesos

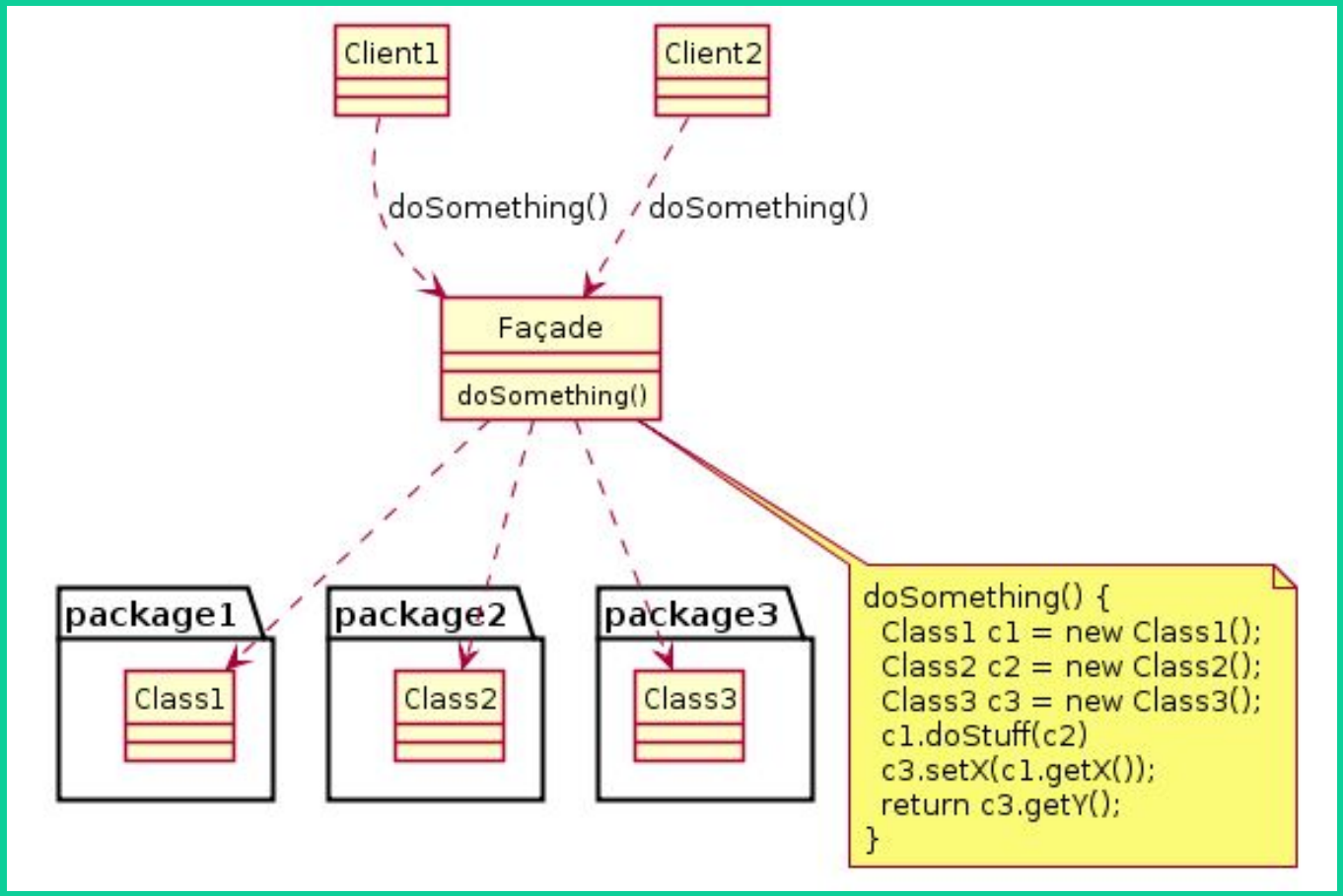
Proxy - Diagrama



Facade

- Propósito:
  - proporciona una interfaz unificada para un conjunto de interfaces de un subsistema.
  - Define una interfaz de alto nivel que hace que el subsistema sea más fácil de usar
- Motivación:
  - Minimizar la comunicación y dependencias entre subsistemas
- Aplicabilidad:
  - Queramos proporcionar una interfaz simple para un subsistema complejo
  - Cuando haya muchas dependencias entre los clientes y las clases que implementan una abstracción
  - Queremos dividir en capas nuestros subsistemas
- Beneficios:
  - Oculta a los clientes los componentes del subsistema, reduciendo así el numero de objetos con los que tratan los clientes y haciendo que el subsistema sea más fácil de usar
  - Promueve un débil acoplamiento entre el subsistema y sus clientes
  - No impide que las aplicaciones usen las clases del subsistema en caso de que sea necesario. De este modo se puede elegir entre facilidad de uso y generalidad

Facade - Diagrama



Strategy

Necesidad

- Un mismo objeto debe poder tener un comportamiento que debe ser determinado en tiempo de ejecución.

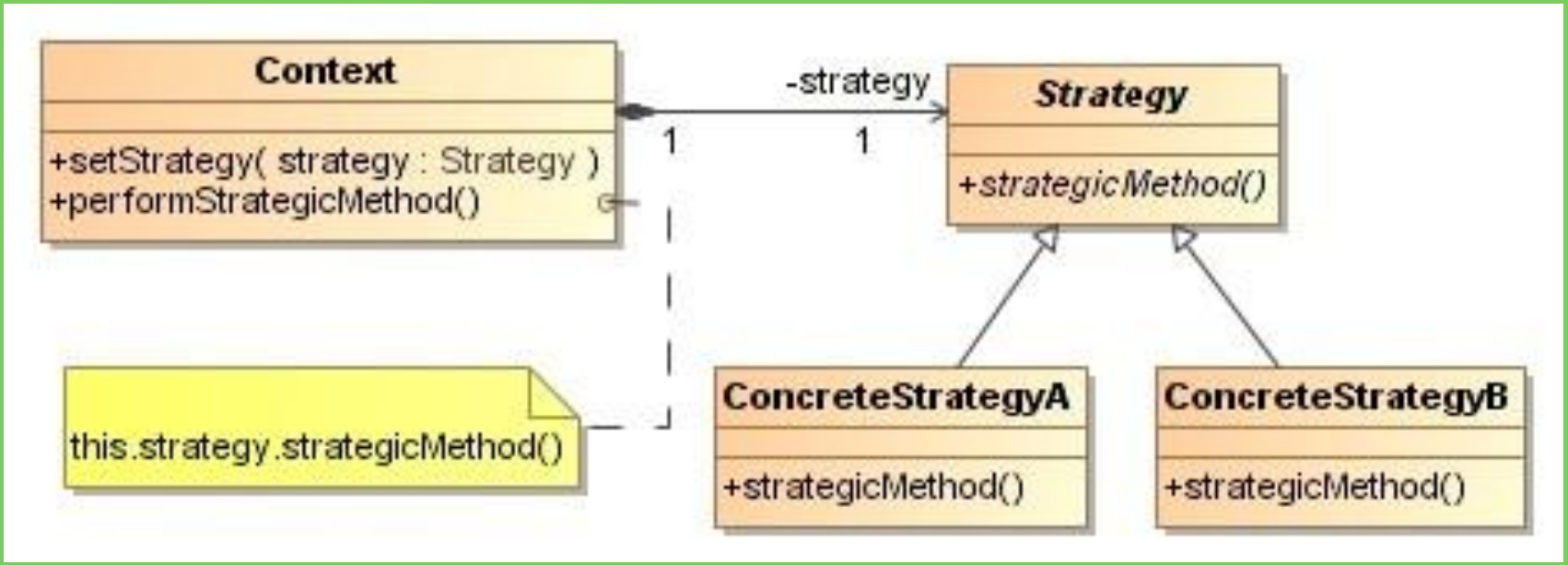
Solución

- Delegar el comportamiento en otro objeto
- Armar una jerarquía con los diferentes comportamientos
- ***Inyectar*** el comportamiento al objeto a través de un método o de su constructor

Consecuencias

- Se eliminan los condicionales
- Se crea una jeraquía paralela a la jerarquía base
- Los comportamientos quedan agrupados por familias
- A veces no es tan fácil aislar el comportamiento
- A veces no alcanza

Strategy - Diagrama



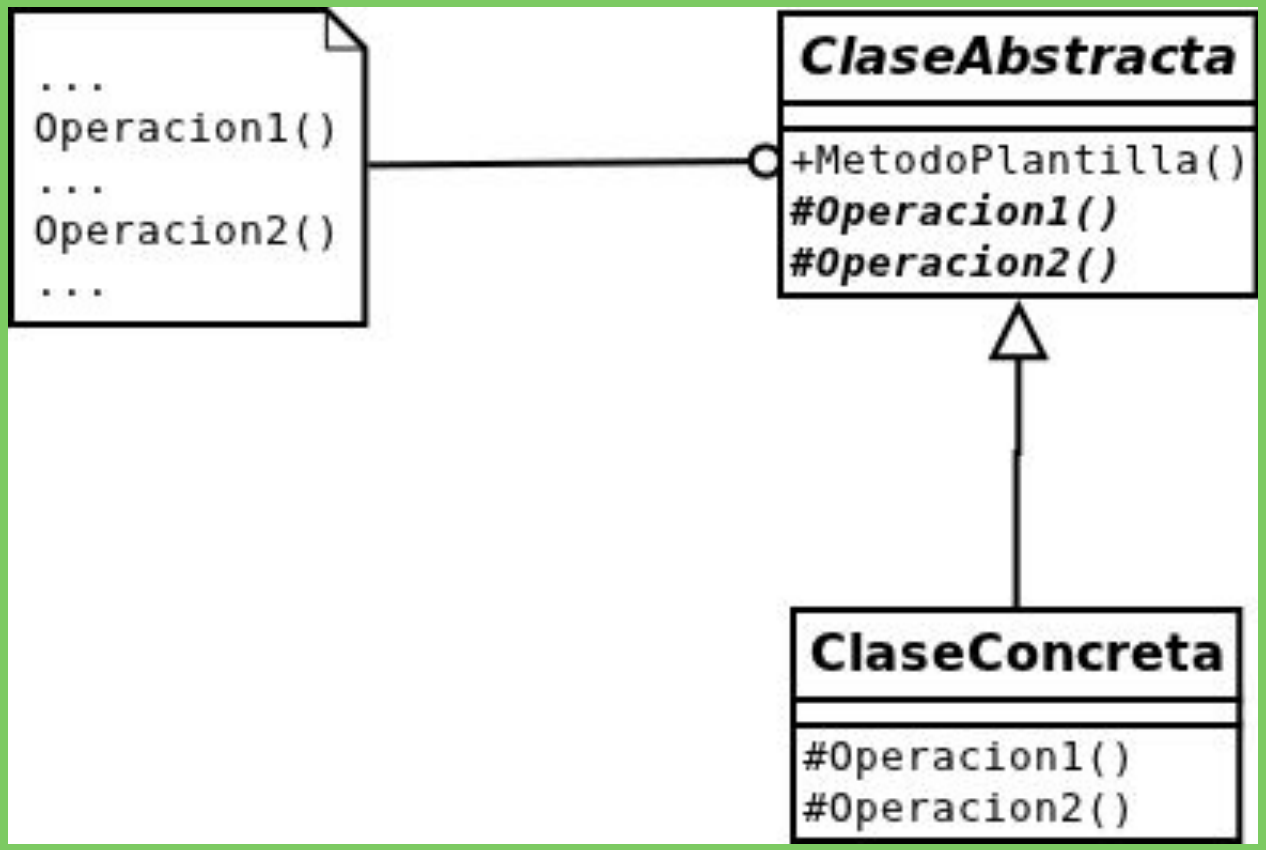
Template

- **Propósito**:
  - Define en una operación el esqueleto de un algoritmo, delegando en las subclases algunos de sus pasos. Permite que las subclases redefinan ciertos pasos de un algoritmo sin cambiar su estructura.

- **Aplicabilidad**:
  - Para implementar las partes de un algoritmo que no cambian y dejar que sean las subclases quienes implementen el comportamiento puede variar
  - Cuando el comportamiento repetido de varias subclases debería factorizarse y ser localizado en una clase común para evitar código duplicado
  - Para controlar las extensiones de las subclases

- **Beneficios**:
  - Son una técnica fundamental de reutilización de código
  - Extraen el comportamiento común de las clases de la biblioteca
  - “Principio de Hollywood”, una clase padre llama a las operaciones de unasubclase y no al revés
  - Operaciones de enganche, proporcionan el comportamiento predeterminado que pude ser modificado por las subclases

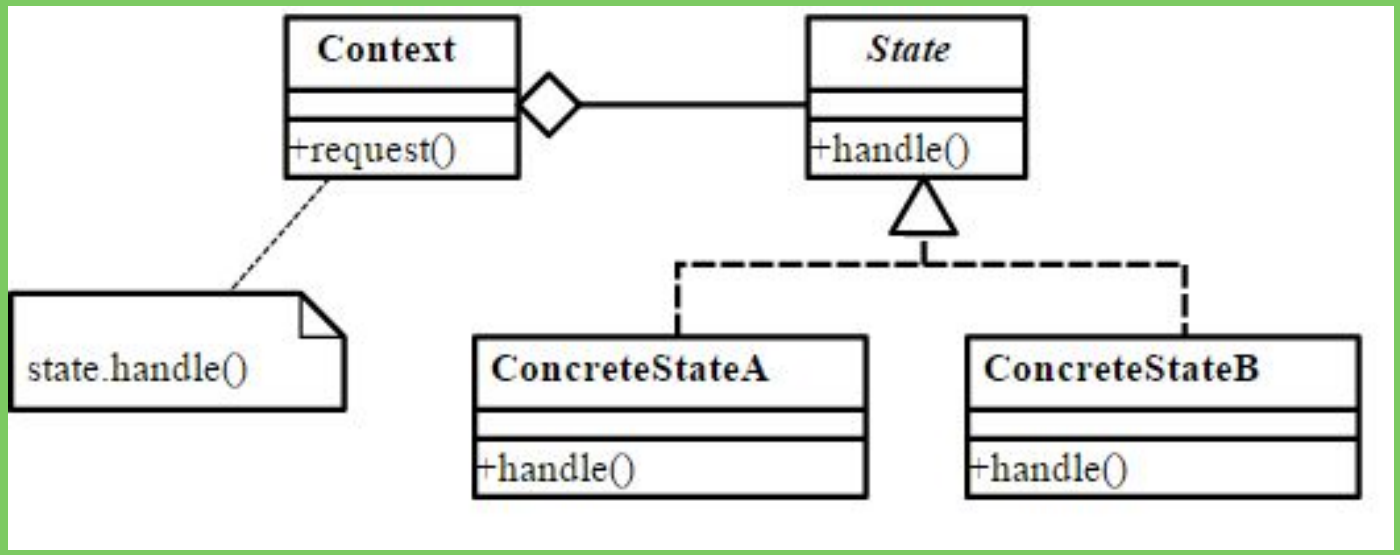
Template - Diagrama



State

- Propósito:
  - permite que un objeto modifique su comportamiento cada vez que cambie su estado interno. Parecerá que cambia la clase del objeto
- Motivación:
  - Conexión TCP, que presenta 3 estados, establecida, escuchando y cerrada
- Aplicabilidad:
  - El comportamiento de un objeto depende de su estado, y debe cambiar en tiempo de ejecución dependiendo de ese estado
  - Las operaciones tienen largas sentencias condicionales con múltiples ramas que dependen del estado del objeto. Este estado se suele representar por una o más constantes enumeradas.
- Beneficios:
  - Localiza el comportamiento dependiendo del estado y divide dicho comportamiento en diferentes estados.
  - Hace explicitas las transiciones entre estados
  - Los objetos estado pueden compartirse

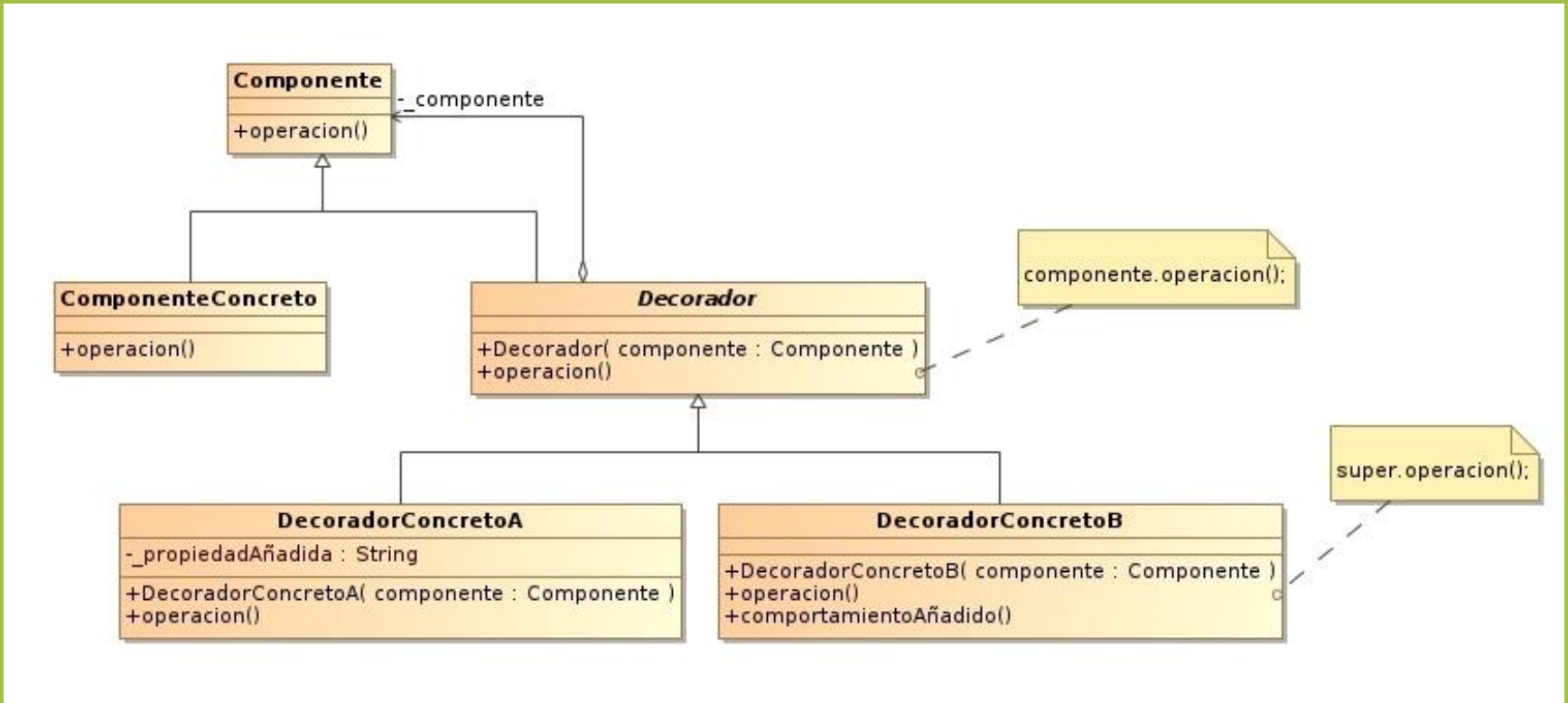
State - Diagrama



Decorator

- Propósito:
  - Asigna responsabilidades adicionales a un objeto dinámicamente, proporcionando una alternativa flexible a la herencia para extender la funcionalidad
- Motivación:
  - Interfaces de usuarios a las que se agregan propiedades o comportamientos
  - Cambio de piel
- Aplicabilidad:
  - Para añadir objetos individuales de forma dinámica y transparente, es decir, sin afectar a otros objetos
  - Para responsabilidades que pueden ser retiradas
  - Cuando la extensión mediante herencia no es viable.
- Beneficios:
  - Mas flexibilidad que la herencia estática
  - Evitar clases cargadas de funciones en la parte de arriba de la jerarquía
- Desventajas:
  - Un decorador y su componente no son idénticos
  - Muchos objetos pequeños

Decorator - Diagrama



Composite

- Propósito:
  - compone objetos en estructuras de árbol para representar jerarquías de parte-todo. Permite que los clientes traten de manera uniforme a los objetos individuales y a los compuestos
- Motivación:
  - Las aplicaciones graficas como los editores de dibujo y los sistemas de diseño, permiten agrupar componentes simples en mas grandes
  - Manejos de excepciones
- Aplicabilidad:
  - Quiera representar jerarquías de objetos parte-todo
  - Quiera que los clientes sean capaces de obviar las diferencias entre composiciones de objetos y los objetos individuales. Los clientes trataran a todos los objetos de la estructura compuesta de manera uniforme
- Beneficios:
  - Defines jerarquías de clases formadas por objetos primitivos y compuestos
  - Simplifica el cliente
  - Facilita añadir nuevos tipos de componentes
- Desventajas:
  - Puede hacer que un diseño sea demasiado general

Composite - Diagrama