

Patrones de diseño

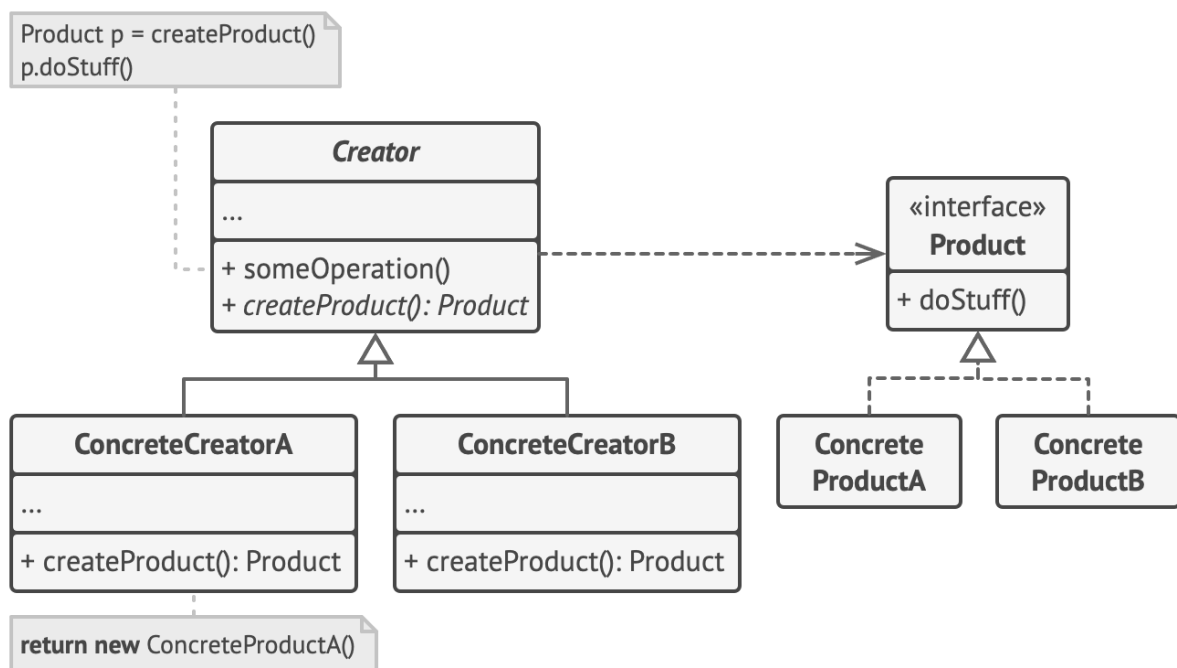
Patrones Creacionales

Proporcionan mecanismos para crear objetos de forma flexible y para reutilizar el código existente.

Factory Method

Proporciona una interfaz para crear objetos en una superclase, permitiendo que las subclases decidan qué tipo de objeto instanciar.

En vez de llamar al *new* se invoca un método fábrica (que en realidad sí usa el *new* internamente). Los productos son los objetos devueltos por la fábrica.



- La interfaz Product es común a todos los objetos que puede producir la clase creadora y sus subclases.
- La clase Creator declara el método fábrica.
- Los Concrete Creators sobrescriben el método de fábrica base para que devuelva diferentes tipos de productos.

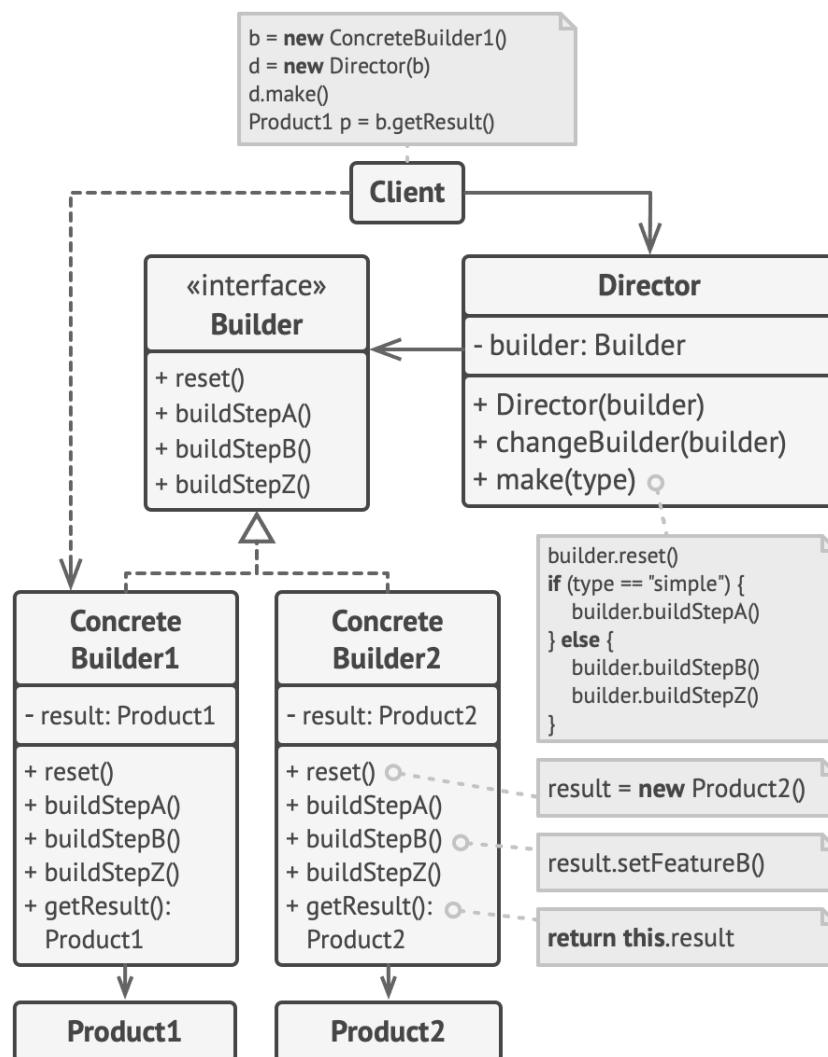
Ejemplo: crear elementos de interfaz de usuario (UI) multiplataforma sin acoplar el código cliente a clases UI concretas.

Builder

Nos permite construir objetos complejos paso a paso, de esta forma, podemos producir distintos tipos y representaciones de un objeto empleando el mismo código de construcción.

En el caso de extender una clase base y terminar con un gran grupo de subclasses que cubren todas las combinaciones posibles de los parámetros, si quisiéramos añadir un parámetro nuevo, tendríamos que incrementar esta jerarquía de clase aún más. También si tenemos un constructor infinito dentro de la clase base.

Para solucionar estos problemas, se sugiere que saquemos el código de construcción del objeto de su propia clase y lo coloquemos dentro de objetos independientes llamados *constructores*. Para construir un objeto lo haremos en una serie de pasos, invocando solo los necesarios.



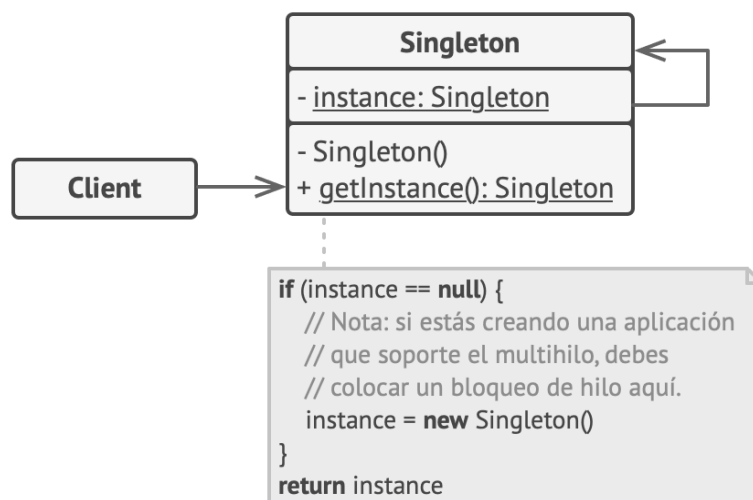
Ejemplo: crear una hamburguesa con solo los elementos deseados.

Singleton

Con este patrón nos aseguramos de que una clase tenga una única instancia, y a la vez, obtenemos un punto de acceso global a la misma.

¿Para qué querríamos lograr esto?

- Para controlar el acceso a un recurso compartido.
- Podemos acceder a un objeto desde cualquier parte del programa, sin necesidad de utilizar variables globales, ya que no son seguras.



Patrones Estructurales

Explican cómo ensamblar objetos y clases en estructuras más grandes, manteniendo la flexibilidad y eficiencia de las mismas.

Decorator

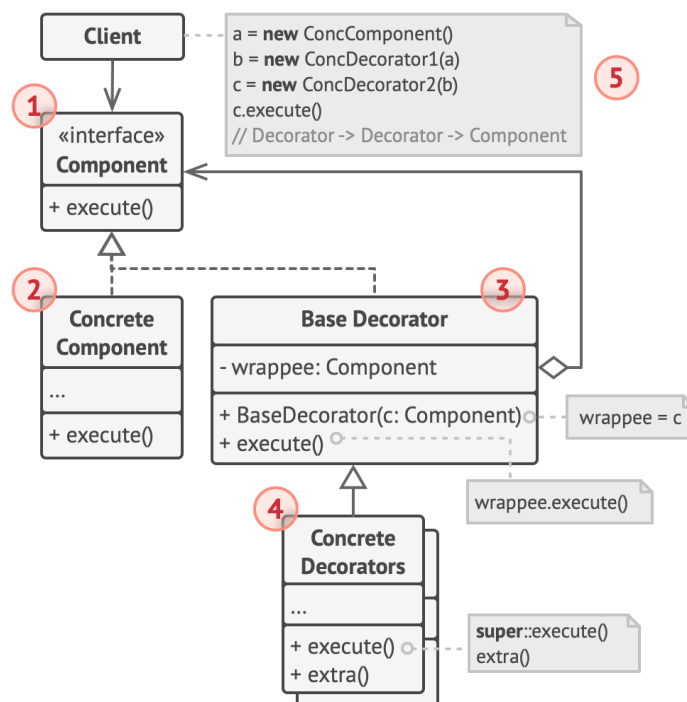
Permite añadir funcionalidades a objetos colocándolos dentro de otros objetos encapsuladores especiales que contienen estas funcionalidades.

En lugar de la herencia para extender una clase, este patrón emplea la Agregación o Composición. Donde un objeto tiene una referencia a otro y le delega parte del trabajo.



Este patrón utiliza wrappers, que son objetos que pueden vincularse con un objeto objetivo. El wrapper con tiene el mismo grupo de métodos que el objetivo y le delega todas las solicitudes que recibe, y puede alterar el resultado haciendo (“decorando”) algo antes o después de pasar la solicitud al objetivo.

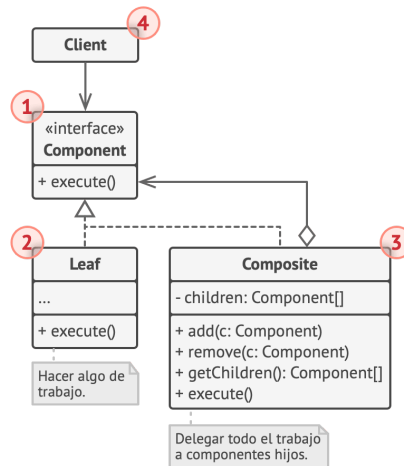
El wrapper implementa la misma interfaz que el objeto envuelto, así podremos envolver un objeto en varios wrappers.



Ejemplo: decorar pizza con distintos ingredientes.

Composite

Permite componer objetos en estructuras de árbol y trabajar con esas estructuras como si fueran objetos individuales. Facilita operaciones recursivas sobre estructuras de datos anidadas.



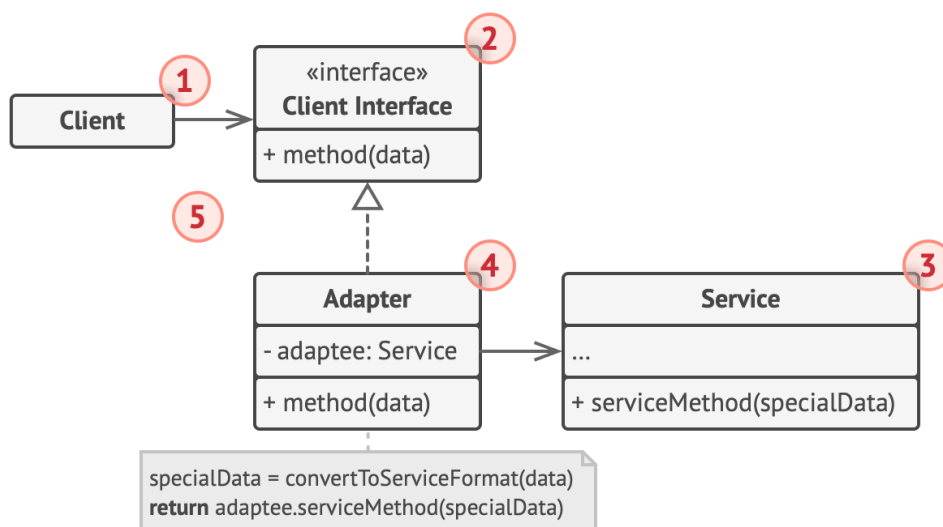
- La interfaz Component describe operaciones que son comunes a los elementos simples y complejos del árbol.
- Leaf es un elemento hoja del árbol (no tiene hijos-subelementos).
- El Composite es un elemento que tiene subelementos: hojas u otros contenedores. Al recibir una solicitud delega el trabajo a sus subelementos, procesa los resultados intermedios y devuelve el resultado final al Client.

Adapter

Permite la colaboración entre objetos con interfaces incompatibles.

Tendremos un objeto especial Adaptador que convierte la interfaz de un objeto, de forma que otro objeto pueda comprenderla.

El adaptador obtiene una interfaz compatible con uno de los objetos existentes. De esta forma, el objeto existente puede invocar con seguridad los métodos del adaptador. Al recibir una llamada, el adaptador envía la solicitud al segundo objeto, pero en un formato y orden que ese segundo objeto espera.



La clase Adaptadora es capaz de trabajar tanto con la clase cliente como con la clase de servicio: implementa la interfaz con el cliente, mientras envuelve (contiene una instancia de esa clase y la usa internamente) el objeto de la clase de servicio. La clase adaptadora recibe llamadas del cliente a través de la interfaz de cliente y las traduce en llamadas al objeto envuelto de la clase de servicio, pero en un formato que pueda comprender.

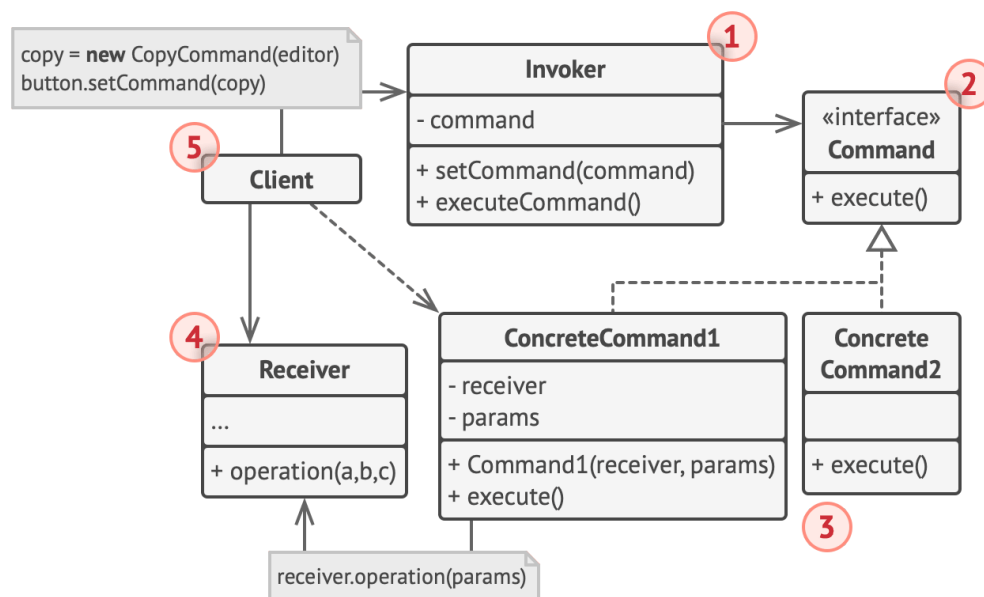
Ejemplo: permitir a una clase AudioPlayer reproducir archivos de audio de tipo MP3, WAV y OGG.

Patrones de Comportamiento

Tratan con algoritmos y la asignación de responsabilidades entre objetos.

Command

Encapsula una solicitud como un objeto, permitiendo desacoplar el emisor y receptor de una acción (GUI y lógica de negocio), ejecución diferida o en cola de comandos e implementar operaciones reversibles (copiar/pegar, undo/redo).



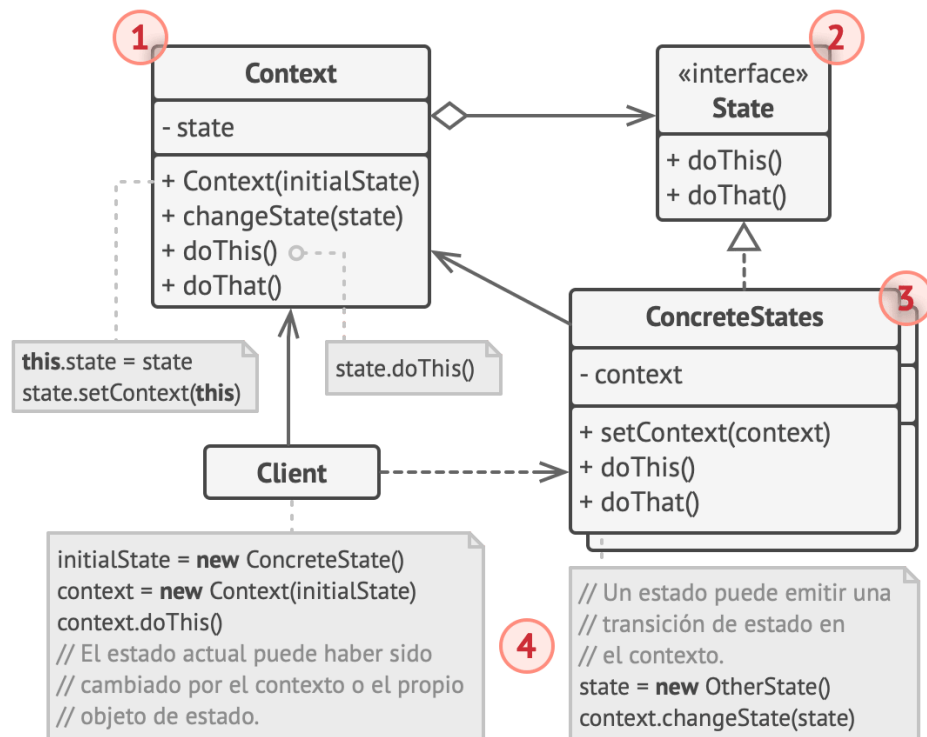
- El Client crea los comandos y se asignan al Invoker.
- El Invoker solo ejecuta los comandos.
- Cada ConcreteCommand encapsula una acción del Receiver.

State

Permite a un objeto alterar su comportamiento cuando su estado interno cambia. Parece como si el objeto cambiara de clase.

Este patrón sugiere crear nuevas clases para todos los estados posibles de un objeto y extraer todos los comportamientos específicos del estado para colocarlos dentro de esas clases.

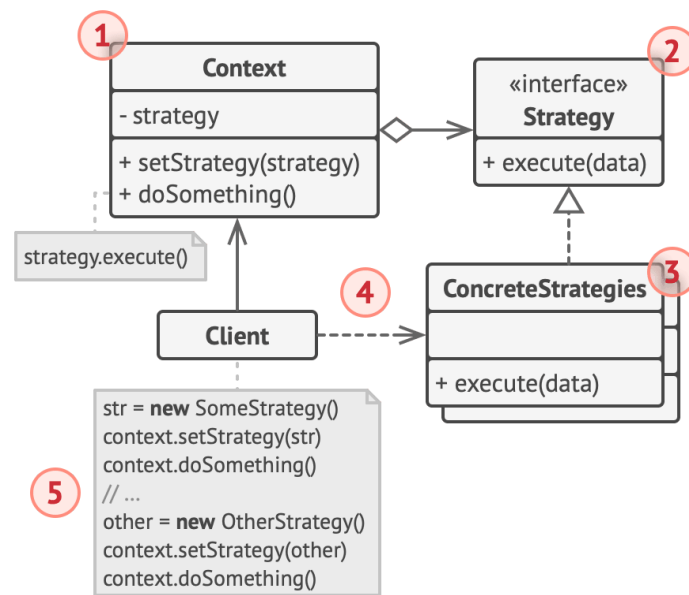
El objeto original llamado Context, almacena una referencia a uno de los objetos de estado representando su estado actual y delega todo el trabajo relacionado con el estado a ese objeto. Para poder realizar el cambio de estado en el Context, todas las clases de estado deben seguir la misma interfaz.





Ejemplo: diferentes estados de un cajero automático.

Strategy

Permite definir una familia de algoritmos, colocar cada uno de ellos en una clase separada y hacer sus objetos intercambiables.



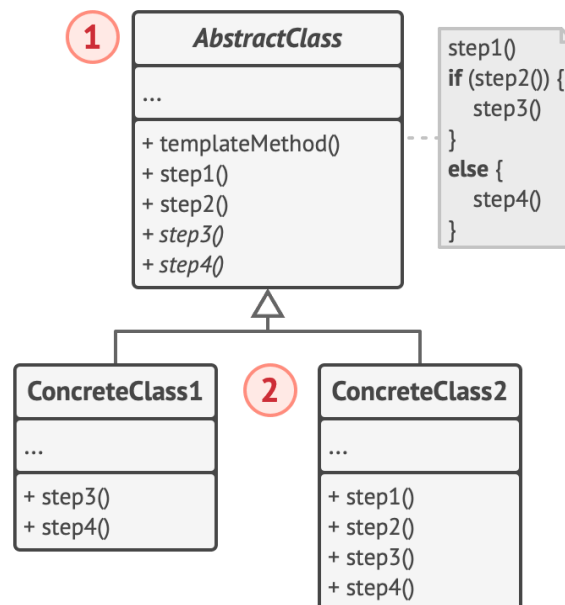
Característica	State 	Strategy 
Propósito	Permitir que un objeto cambie su comportamiento dependiendo de su estado interno.	Permitir que un objeto elija un algoritmo entre múltiples opciones.
Cuándo usarlo	Cuando el objeto tiene estados bien definidos y el comportamiento cambia en función del estado actual.	Cuando hay múltiples algoritmos intercambiables y queremos cambiar el comportamiento en tiempo de ejecución.
Ejemplo típico	Máquina expendedora, semáforo, ciclo de vida de una conexión.	Métodos de pago, estrategias de ordenamiento, formas de compresión de archivos.
Cambio de comportamiento	El objeto cambia automáticamente de estado sin intervención del cliente.	El cliente elige explícitamente la estrategia a usar.

Template Method

Define el esqueleto de un algoritmo en la superclase pero permite a las subclases sobrescribir pasos del algoritmos sin cambiar su estructura.

Se divide al algoritmo en una serie de pasos, se convierte estos pasos en métodos y se colocan una serie de llamadas a esos métodos dentro de un único método plantilla. Para

utilizar el algoritmo, el cliente debe aportar su propia subclase, implementar todos los pasos abstractos y sobrescribir algunos de los opcionales si es necesario.



Ejemplo: los pasos de preparar una comida siempre los mismos, el template sería la receta con los pasos `añadirIngrediente()`, `añadirCondimento()` que pueden variar y ser pisados por las subclases.

Visitor

Permite separar algoritmos de los objetos sobre los que operan.

Se coloca el nuevo comportamiento en una clase separada visitante, en lugar de integrarlo dentro de las clases existentes. El objeto que originalmente tenía que realizar el comportamiento se pasa ahora a uno de los métodos del visitante como argumento, para que el método acceda a toda la información necesaria.

```
class ExportVisitor implements Visitor is
    method doForCity(City c) { ... }
    method doForIndustry(Industry f) { ... }
    method doForSightSeeing(SightSeeing ss) { ... }
    // ...
```

```
foreach (Node node in graph)
    if (node instanceof City)
        exportVisitor.doForCity((City) node)
    if (node instanceof Industry)
        exportVisitor.doForIndustry((Industry) node)
    // ...
}
```

```
// Código cliente
foreach (Node node in graph)
    node.accept(exportVisitor)

// Ciudad
class City is
    method accept(Visitor v) is
        v.doForCity(this)
    // ...

// Industria
class Industry is
    method accept(Visitor v) is
        v.doForIndustry(this)
    // ...
```

