Metodología de Sistemas II

Clase 6

Patrones de Diseño: Comportamiento

Jeremías Fassi

2025

Patrones de Diseño: Comportamiento

Introducción a los Patrones de Comportamiento

Jeremías Fassi

2025

Introducción a los patrones de comportamiento

- ¿Qué son los patrones de comportamiento?

- Los patrones de comportamiento se enfocan en cómo los objetos se comunican y reparten responsabilidades, encapsulando algoritmos, flujos y reacciones ante eventos para reducir acoplamiento y aumentar flexibilidad.

- En otras palabras, ayudan a organizar la **interacción** entre objetos para que el sistema sea extensible sin romper lo existente.

Introducción a los patrones de comportamiento

- Características principales

    - Se enfocan en la **interacción entre objetos**: definen cómo se comunican y colaboran sin estar fuertemente acoplados.
    - **Distribuyen responsabilidades**: buscan que cada clase tenga un rol claro en la dinámica de trabajo.
    - **Favorecen la flexibilidad**: permiten cambiar algoritmos, secuencias o respuestas a eventos sin modificar el resto del sistema.

Introducción a los patrones de comportamiento

- Características principales
    - **Reducen dependencias rígidas**: el emisor no necesita conocer los detalles de los receptores (ej. *Observer*), el cliente no sabe cómo se ejecuta el algoritmo (*Strategy*), o el invocador no sabe cómo se ejecuta la acción (*Command*).
    - Permiten **extender comportamientos**: nuevos algoritmos, comandos u observadores pueden añadirse sin modificar el código existente.
    - Se relacionan mucho con el “flujo” de ejecución: qué pasa primero, qué sigue después, y quién decide el camino.

Introducción a los patrones de comportamiento

→ Analogía, imaginen una orquesta:

- El Director (Contexto/Invoker/Subject): no toca instrumentos, pero indica cuándo y cómo deben actuar los músicos.
- Los Músicos (Observers/Strategies/Commands): cada uno sabe qué hacer cuando recibe la señal. Algunos pueden tocar una melodía distinta (estrategias), otros se suman o bajan del escenario (observadores), y otros ejecutan una acción específica como un solo (comandos).
- El Público (sistema que observa el resultado): percibe la música coordinada gracias a la interacción organizada, aunque cada músico toque de forma independiente.

Patrón Observer

Jeremías Fassi

2025

Introducción

El patrón Observer define una relación **uno a muchos** entre objetos: cuando un objeto (llamado *Subject*) cambia de estado, todos los objetos dependientes (llamados *Observers*) son notificados de forma automática.

- Aclaración: como el objeto *Subject* también notificará a otros objetos sobre los cambios en su estado, lo llamaremos ***Publisher***. Y todos los demás objetos que desean saber sobre los cambios en el estado del *Publisher* se llaman ***Subscribers***.

- Este patrón se usa cuando necesitamos que varias partes de un sistema reaccionen en simultáneo a un evento o cambio de estado, pero sin generar dependencias rígidas entre ellas.

Motivación

Imaginemos el siguiente escenario:

- Hay una tienda (*Store*) que va a recibir un nuevo producto.
- Varias personas (*Customer*) están interesadas en saber cuándo ese producto estará disponible.
- Las personas podrían chequear todos los días la tienda, pero claramente eso es ineficiente.
- Alternativamente, la tienda podría enviar mensajes (emails) cada vez que recibe algo nuevo... pero eso generaría muchísimo spam si se hace indiscriminadamente.

Motivación

Se necesita un mecanismo que permita que cada cliente interesado reciba la notificación, pero sin que la tienda deba enviar información a todos, o depender de que los interesados consulten constantemente.

- La solución: la tienda actúa como *publisher/subject* y los clientes como *subscribers/observers*, y cada cliente decide si se “suscribe” o no.

Solución

→ Mecanismo de suscripción

- El *publisher* (o *subject*) incluye:
  - Una lista de suscriptores (observadores).
  - Métodos públicos para suscribir (subscribe / attach) y desuscribir (unsubscribe / detach) observadores.
  - Un método interno para notificar (*notify*) a todos los observadores cuando ocurre un cambio relevante.

Cuando ocurre un evento importante o un cambio de estado, el sujeto recorre su lista de suscriptores y llama un método (por ejemplo `update`) con información contextual.

Solución

→ Separación de responsabilidades

- El *publisher* no conoce las clases concretas de los observadores, solo la interfaz que deben cumplir (por ejemplo: `update(data)`).
- Los observadores reaccionan de manera independiente a esa notificación, pudiendo usar los datos que les pasan o incluso consultar el publisher para más información.
- La suscripción y notificación es dinámica: nuevos observadores pueden entrar o salir en tiempo de ejecución.

Estructura

- Publisher

```
class Publisher {
    listeners: List<Subscriber>
    method subscribe(listener)
    method unsubscribe(listener)
    method notify(eventData) {
        for each listener in listeners:
            listener.update(eventData)
    }
}

class ConcretePublisher extends Publisher {
    // lógica de negocio, cuando ocurre algo relevante
    // llama: notify(eventData)
}
```

Estructura

- Subscriber

```
interface Subscriber {
    method update(eventData)
}

class ConcreteSubscriber implements Subscriber
{
    method update(eventData) {
        // reaccionar, usando eventData
    }
}
```

Cuándo usar

Usar **Observer** cuando:

- Cambios en un objeto pueden requerir que otros objetos cambien también, y no es posible saber cuántos ni cuáles serán esos otros objetos de antemano.
- Quieran que los objetos “escuchen” a otros durante un tiempo limitado o bajo ciertas condiciones dinámicas.
- En GUI / frameworks de eventos, donde hay botones, controles o elementos del UI notifican a *handlers* de eventos.

La lista de observadores es dinámica, y la comunicación se hace a través de la interfaz común, lo que reduce el acoplamiento.

Observer

- Ventajas
  - Permite que los objetos permanezcan desacoplados entre sí: el *subject* no necesita conocer los detalles de los observadores, sólo su interfaz.
  - Facilita la extensión: nuevos observadores pueden agregarse sin modificar el sujeto.
  - Permite relaciones en tiempo de ejecución: suscripciones y cancelaciones dinámicas.

Observer

- Desventajas
  - El orden de notificación puede no estar definido (aleatorio).
  - Si hay muchos observadores o notificaciones frecuentes, puede generarse sobrecarga o notificaciones en cascada.
  - Riesgo de *overflow* si los observadores no se desuscriben correctamente cuando ya no se necesitan.

Observer - Ejemplos

- Python
  - observer.py
- TS
  - observer.ts

Consideraciones

- La interfaz del observador (`update(...)`) puede recibir parámetros: los datos del evento o incluso la referencia al publisher, para que el observador obtenga más información.
- El mecanismo de suscripción puede estar en una clase base o delegarse a un *EventManager* (composición) si no es posible modificar la jerarquía existente.
- Importancia de darse de baja (*unsubscribe*) para evitar que objetos “muertos” sigan recibiendo notificaciones.

Patrón Strategy

Jeremías Fassi

2025

Intención

El patrón **Strategy** propone separar un conjunto de algoritmos relacionados, en clases independientes y proveer una interfaz común que los unifique. El objeto que usa esos algoritmos (llamado Contexto) no implementa directamente la lógica, sino que delegará el trabajo a la estrategia seleccionada.

Notar que de esta manera, el algoritmo se vuelve **intercambiable en tiempo de ejecución**, dándole al sistema flexibilidad sin necesidad de escribir condicionales gigantes ni modificar el código del contexto.

Motivación

Supongamos que tenemos una aplicación de navegación:

Al principio solo generaba rutas para autos. Más tarde se pidió que también sugiriera rutas a pie, en transporte público y en bicicleta. Y sin darnos cuenta, el código del planificador de rutas terminó lleno de *ifs* o *switch* anidados:

```
if transporte == AUTO: calcularRutaEnAuto()
elif transporte == PIE: calcularRutaAPie()
elif transporte == BUS: calcularRutaEnColectivo()
elif transporte == BICI: calcularRutaEnBicicleta()
...
```

Motivación

Cada vez que aparece un nuevo medio, hay que modificar esta clase central, aumentando el riesgo de errores y dificultando su mantenimiento.

Solución

Se extraen los algoritmos en clases separadas que implementan una interfaz común (ej. `IRouteStrategy`).

- El Contexto (ej. `Navigator`) mantiene una referencia a la estrategia actual.
- El Cliente (ej. la UI) elige y asigna la estrategia deseada en runtime.
- Cuando se llama al método `buildRoute()`, el contexto delegará en la estrategia actual.

Esto logra que el contexto sea independiente de los detalles concretos de cada algoritmo.

Analogía

Imaginen ir al aeropuerto:

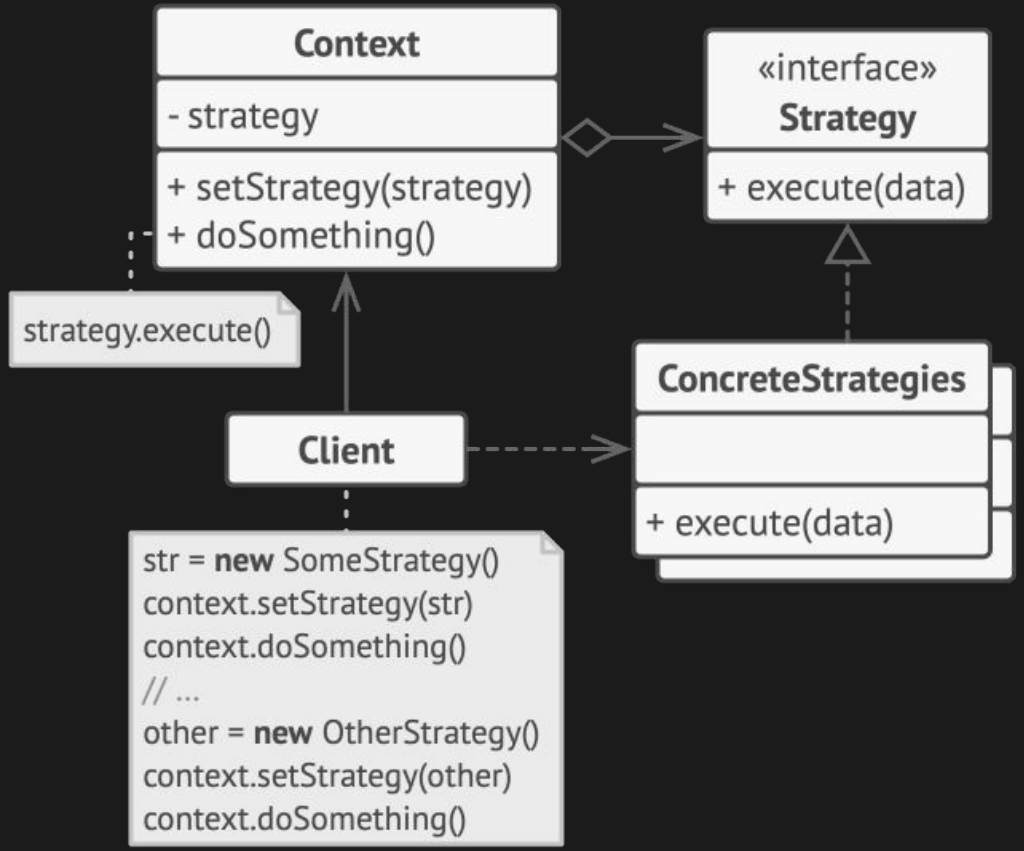
- Se puede ir en colectivo, taxi, auto propio o bicicleta.
- Todas las opciones cumplen el mismo objetivo: llegar al aeropuerto.
- Cada opción es una estrategia distinta con sus ventajas y desventajas (precio, tiempo, comodidad).
- El cliente elige la estrategia más conveniente y puede cambiarla según la situación.

El aeropuerto sigue siendo el mismo (el contexto), lo que cambia es el algoritmo/estrategia para llegar.

Estructura

- **Strategy** (interfaz): define el método común (`buildRoute`, `sort`, `compress`, etc.).
- **ConcreteStrategies**: implementan el método o función de acuerdo a cada variante.
- **Context**: mantiene una referencia a **Strategy** y delega la ejecución.
- **Client**: crea y asigna la estrategia al contexto.

Estructura


Context
- strategy
+ setStrategy(strategy)
+ doSomething()
«interface»
Strategy
+ execute(data)
strategy.execute()
Client
ConcreteStrategies
+ execute(data)
str = new SomeStrategy()
context.setStrategy(str)
context.doSomething()
// ...
other = new OtherStrategy()
context.setStrategy(other)
context.doSomething()

Aplicabilidad (cuándo usarlo)

Cuándo usar Strategy:

- Cuando existen muchas variantes de un algoritmo y no queremos mantener un código lleno de condicionales.
- Cuando una clase necesita comportamientos intercambiables (ej: diferentes formas de ordenar, comprimir, pagar, autenticar).
- Cuando se quiere agregar nuevas variantes sin tocar el código existente (principio de abierto/cerrado).
- Cuando es preferible composición sobre herencia, es decir, no extender una clase con subclases infinitas, sino inyectar estrategias.

Strategy

- Ventajas
  - Intercambio dinámico de algoritmos en tiempo de ejecución.
  - Favorece pruebas unitarias (cada estrategia se testea de manera independiente).
  - Contexto desacoplado de las implementaciones concretas.
  - El sistema cumple con el principio abierto/cerrado: agregar nuevas estrategias sin romper lo existente.

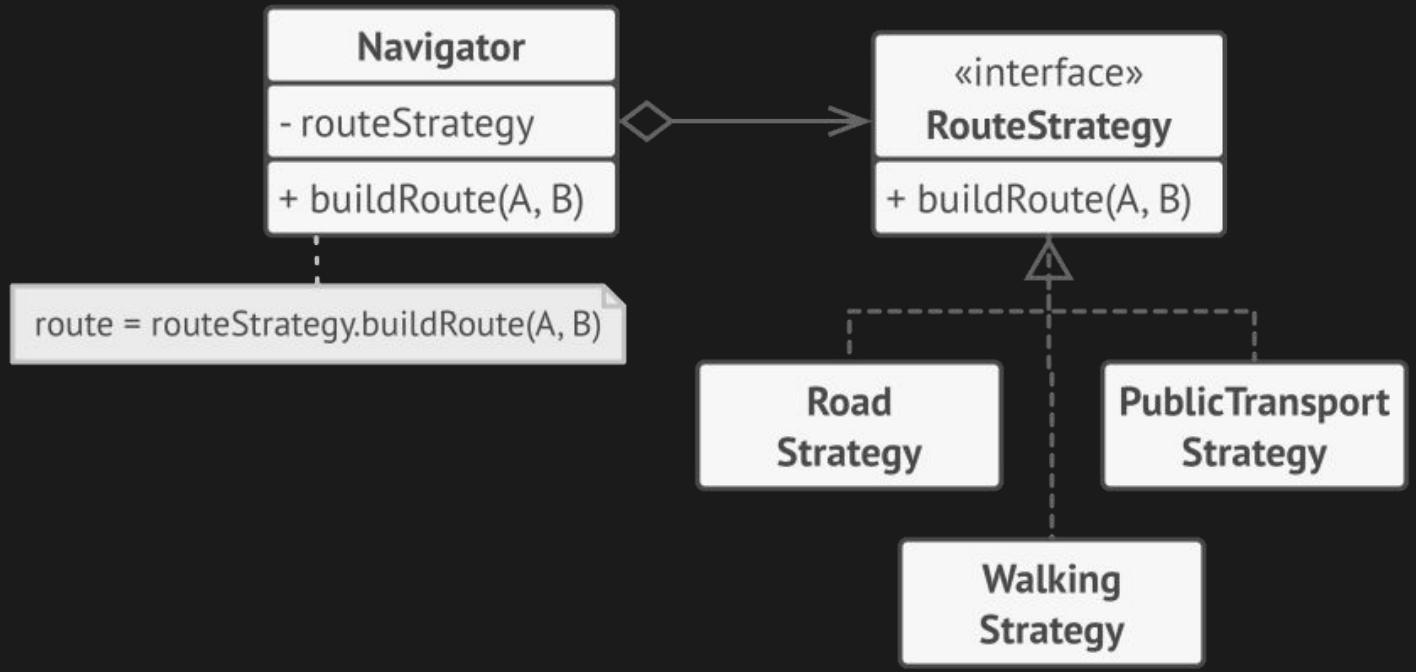
Strategy

- Desventajas
  - Aumenta la cantidad de clases en el proyecto.
  - El cliente debe conocer las diferencias entre estrategias para elegir la correcta.
  - Si hay pocas variantes, puede resultar sobre-ingeniería.

Strategy - Ejemplos

- Python
  - strategy.py
- TS
  - strategy.ts

Strategy


Navigator
- routeStrategy
+ buildRoute(A, B)
route = routeStrategy.buildRoute(A, B)
«interface»
RouteStrategy
+ buildRoute(A, B)
Road Strategy
PublicTransport Strategy
Walking Strategy

Patrón Command

Jeremías Fassi

2025

Intención

El patrón **Command** encapsula una solicitud o acción en un objeto independiente, permitiendo parametrizar clientes con distintas peticiones, encolar operaciones o deshacerlas, y mantener un historial de acciones ejecutadas.

En otras palabras: convierte una petición (ej. “encender la luz”) en un objeto que se puede guardar, pasar como argumento, ejecutar en otro momento, o incluso deshacer.

Motivación

Imaginen una aplicación con una barra de herramientas o un control remoto:

- Cada botón ejecuta una acción distinta (copiar, pegar, deshacer; encender luz, apagar luz, etc.).
- Un código sin un patrón de diseño haría que cada botón estuviera acoplado directamente a una función o clase concreta.
- Entonces, si mañana hay que agregar historial (para implementar *Undo*) o programar tareas (ejecutar más tarde), las llamadas directas no se pueden manipular tan fácilmente.
- **Command** resuelve este problema encapsulando cada acción en un objeto que sigue la misma interfaz (`execute()`), desacoplando quién invoca la acción de quién la ejecuta.

Solución

El patrón **Command** propone:

- Definir una interfaz **Command** con un método común (execute).
- Implementar ConcreteCommand para cada acción específica. Cada comando guarda una referencia al *Receiver* (el objeto que realmente hace el trabajo).
- El *Invoker* (ej. un botón o menú) recibe un objeto *Command* y lo ejecuta cuando el usuario lo dispara.
- El *Client* configura todo, creando los comandos y asignándolos al invocador.

Esto permite que el *Invoker* no sepa nada de la lógica de negocio; solo llama a `execute()`.

Analogía

Imaginen un pedido en un restaurante:

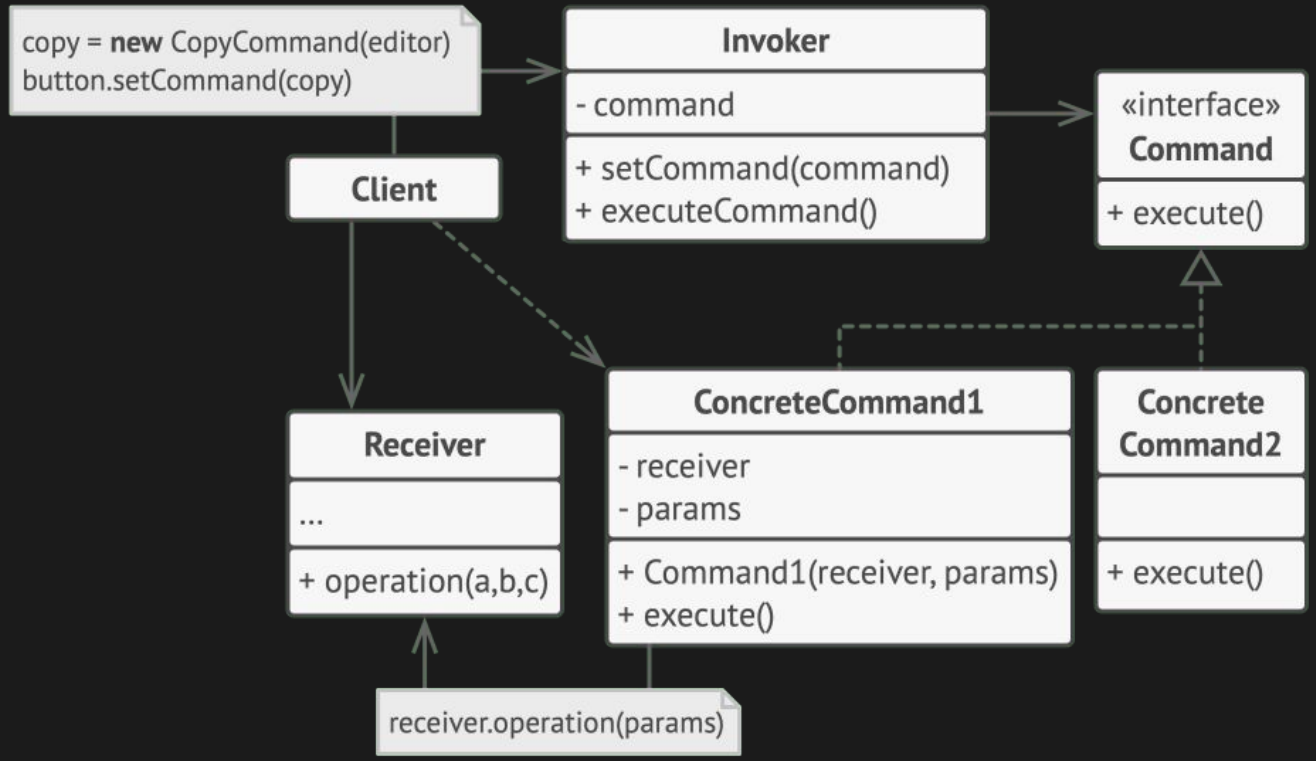
- El cliente hace un pedido y lo entrega al mesero.
- El pedido escrito es el *Command*.
- El mesero (*Invoker*) no cocina, solo lleva el pedido.
- El chef (*Receiver*) recibe el pedido y lo ejecuta preparando la comida.

La gracia de esta separación, es que el cliente no interactúa directamente con el chef.

Estructura

- **Command** (interfaz): declara `execute()`.
- **ConcreteCommand**: implementa `execute()`, delegando la acción al **Receiver**.
- **Receiver**: objeto que contiene la lógica de negocio real.
- **Invoker**: sabe cómo ejecutar un comando, pero no cómo se implementa.
- **Client**: instancia comandos, setea el *Receiver* y asigna el *Command* al *Invoker*.

Estructura


copy = new CopyCommand(editor)
button.setCommand(copy)
Client
Invoker
- command
+ setCommand(command)
+ executeCommand()
«interface»
Command
+ execute()
Receiver
...
+ operation(a,b,c)
ConcreteCommand1
- receiver
- params
+ Command1(receiver, params)
+ execute()
Concrete
Command2
+ execute()
receiver.operation(params)

Aplicabilidad (cuándo usarlo)

- Cuando es necesario parametrizar objetos con operaciones (ej. pasar acciones como parámetros).
- Cuando es necesario deshacer/rehacer operaciones.
- Cuando se desea mantener un historial de acciones.
- Cuando se necesite ejecutar acciones en diferido o en una cola.

→ Ejemplos típicos: editores de texto con Undo/Redo, controles remotos, sistemas de colas de trabajo.

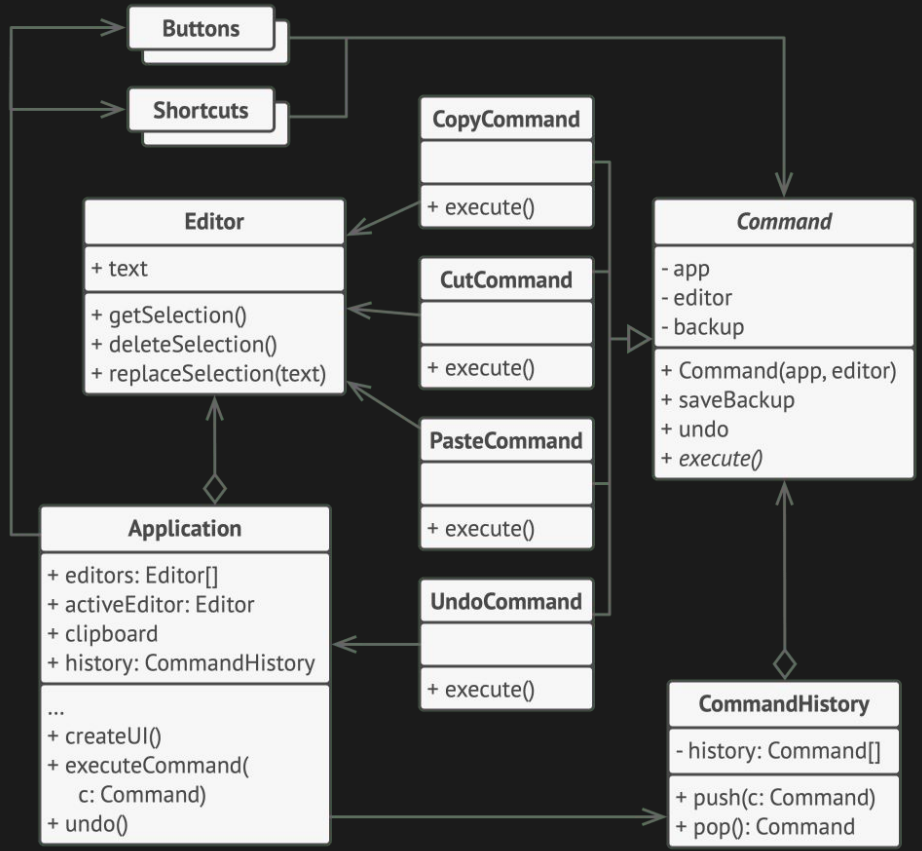
Command

- Ventajas
  - Desacopla al invocador de la acción concreta.
  - Permite undo/redo fácilmente guardando historial de comandos.
  - Posibilita ejecutar acciones encoladas o diferidas.
  - Facilita combinar varias operaciones en una macro.
- Desventajas
  - Aumenta la cantidad de clases (un comando por acción).
  - Puede agregar complejidad innecesaria si no se requiere undo/cola/historial.

Command - Ejemplos

- Python
  - command.py
- TS
  - command.ts

Command


Buttons
Shortcuts
CopyCommand
+ execute()
Editor
+ text
+ getSelection()
+ deleteSelection()
+ replaceSelection(text)
CutCommand
+ execute()
Command
- app
- editor
- backup
+ Command(app, editor)
+ saveBackup
+ undo
+ execute()
PasteCommand
+ execute()
Application
+ editors: Editor[]
+ activeEditor: Editor
+ clipboard
+ history: CommandHistory
...
+ createUI()
+ executeCommand(
c: Command)
+ undo()
UndoCommand
+ execute()
CommandHistory
- history: Command[]
+ push(c: Command)
+ pop(): Command

Comparación de Patrones Estructurales

Patrón	Concepto	Motivación	Cuándo	Caso de Uso
Observer	Suscripciones a eventos: un sujeto notifica a varios observadores.	Necesidad de que varios objetos reaccionen automáticamente cuando otro cambia de estado, sin acoplarlos directamente.	Cuando múltiples módulos deben enterarse de un cambio en tiempo real (suscripción/desuscripción dinámica).	Una `WeatherStation` notifica a `MobileApp` y `Billboard` cada vez que cambia la temperatura.
Strategy	Familia de algoritmos intercambiables en tiempo de ejecución	Evitar condicionales enormes para seleccionar un algoritmo; separar la lógica de uso de la implementación del algoritmo.	Cuando una clase necesita comportamientos intercambiables (pago, rutas, ordenamiento, compresión).	`Navigator` selecciona `CarRoute`, `WalkRoute` o `PublicTransportRoute` según la preferencia del usuario.
Command	Encapsula una petición en un objeto, permitiendo ejecutar, deshacer o encolar.	Desacoplar quién invoca de quién ejecuta; soportar undo/redo, macros, historial o ejecución diferida.	Cuando se necesita historial de operaciones, undo/redo, o ejecutar acciones encoladas.	Un `RemoteControl` ejecuta comandos `LightOnCommand` o `LightOffCommand` sobre un objeto `Light`.

Comparación de Patrones Estructurales

Resumen intuitivo

- Observer → “Cuando uno cambia, todos se enteran.”
- Strategy → “Distintas formas de hacer lo mismo, elegís la que quieras.”
- Command → “Las acciones se guardan como objetos que podés ejecutar o deshacer.”

Recursos:

- Patrones de Comportamiento

- DigitalOcean Community – GoF Patterns Explained

Muchas Gracias

Jeremías Fassi

Javier Kinter