

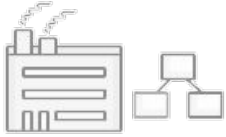


Patrones de Diseño

Capítulo III:
Command

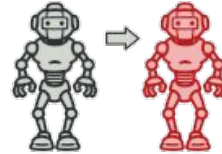


Patrones creacionales



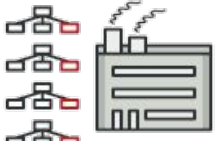
Factory Method

Popularidad: ★★
Complejidad: 🧑



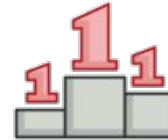
Prototype

Popularidad: ★★
Complejidad: 🧑



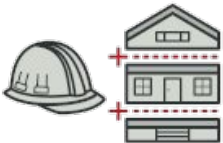
Abstract Factory

Popularidad: ★★★
Complejidad: 🧑🧑



Singleton

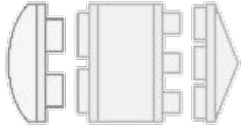
Popularidad: ★★
Complejidad: 🧑



Builder

Popularidad: ★★★
Complejidad: 🧑🧑

Patrones estructurales



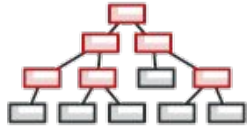
Adapter

Popularidad: ★★ ★
Complejidad: 🧑



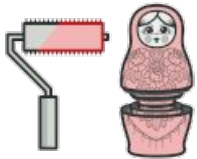
Bridge

Popularidad: ★
Complejidad: 🧑 🧑 🧑



Composite

Popularidad: ★★ ★
Complejidad: 🧑 🧑



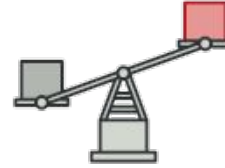
Decorator

Popularidad: ★★ ★
Complejidad: 🧑 🧑



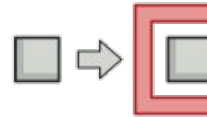
Facade

Popularidad: ★★ ★
Complejidad: 🧑



Flyweight

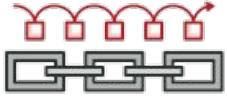
Popularidad:
Complejidad: 🧑 🧑 🧑



Proxy

Popularidad: ★
Complejidad: 🧑 🧑

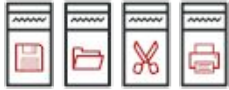
Patrones comportamiento



Chain of Responsibility

Popularidad: ★

Complejidad: 🧑🧑



Command

Popularidad: ★★★

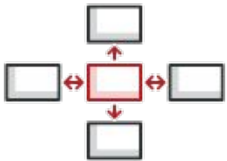
Complejidad: 🧑



Iterator

Popularidad: ★★★

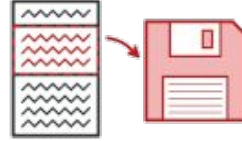
Complejidad: 🧑🧑



Mediator

Popularidad:

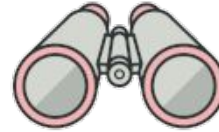
Complejidad: 🧑🧑



Memento

Popularidad: ★

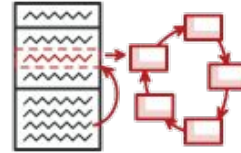
Complejidad: 🧑🧑🧑



Observer

Popularidad: ★★★

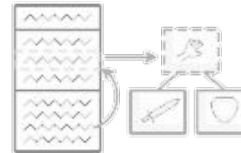
Complejidad: 🧑🧑



State

Popularidad: ★★

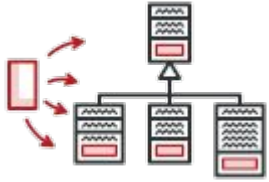
Complejidad: 🧑



Strategy

Popularidad: ★★ ★

Complejidad: 🧑

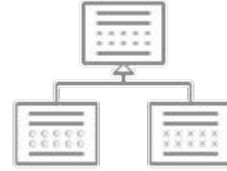


Visitor

Popularidad:



Complejidad:



Template Method

Popularidad:

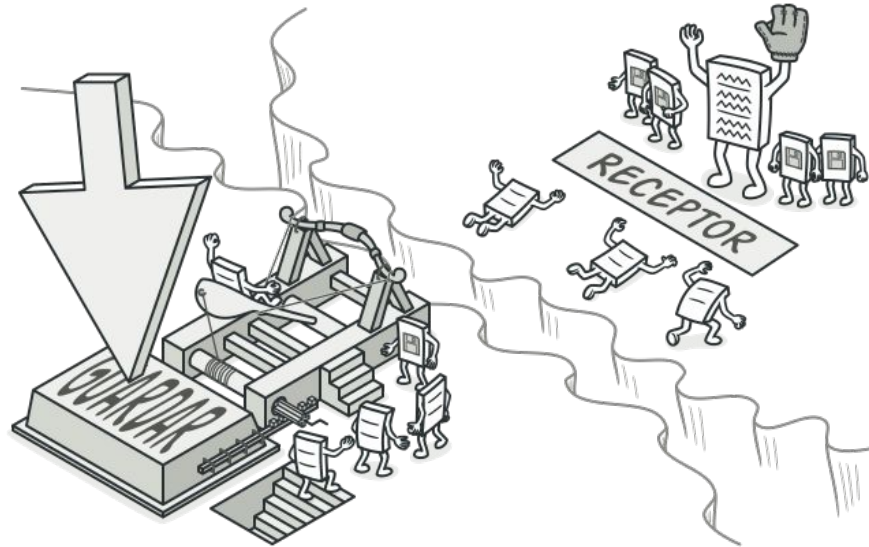


Complejidad:

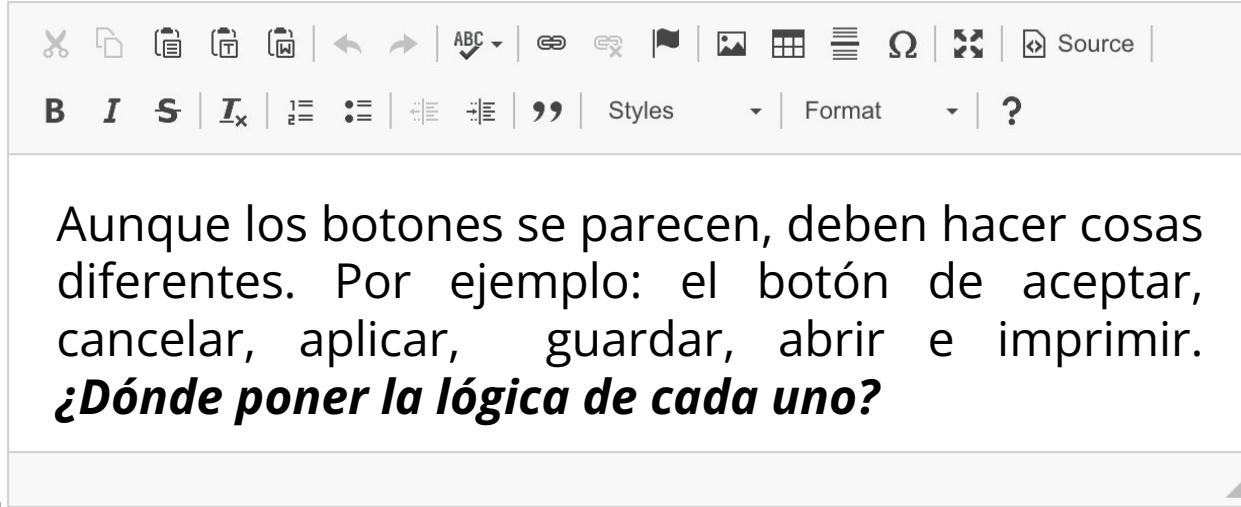


Command

“Comando”, “Orden”, “Action”, “Transaction”

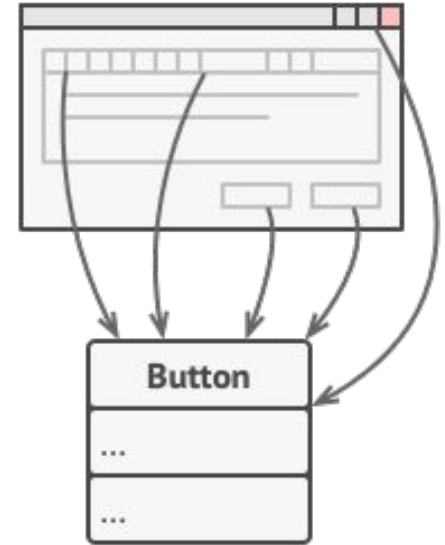


Problema



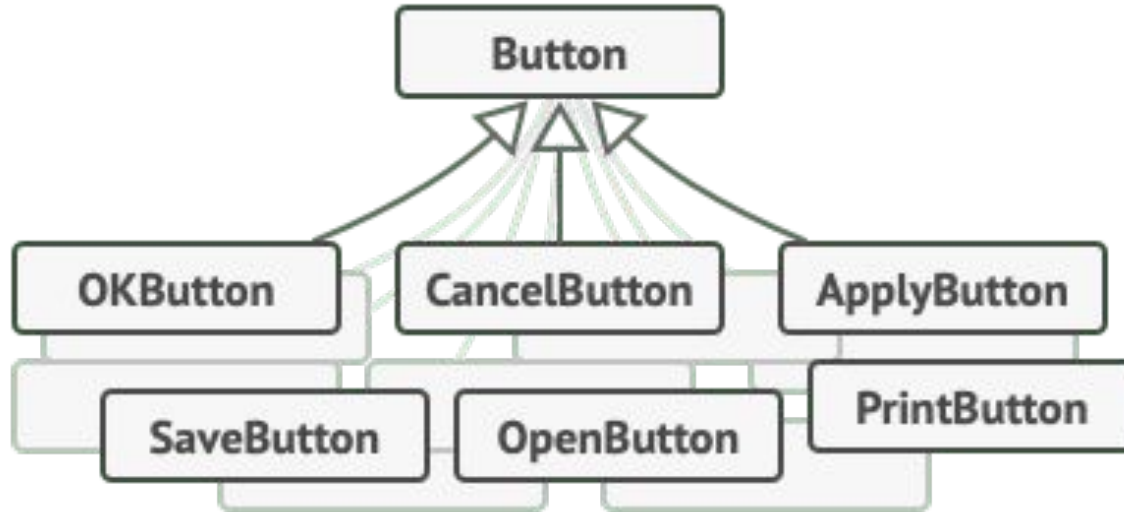
A screenshot of a rich text editor interface. The top part shows a toolbar with various icons for editing (cut, copy, paste, undo, redo, bold, italic, strikethrough, bulleted list, numbered list, link, unlink, image, table, horizontal line, link, unlink, source). Below the toolbar is a text area containing the following text:

Aunque los botones se parecen, deben hacer cosas diferentes. Por ejemplo: el botón de aceptar, cancelar, aplicar, guardar, abrir e imprimir.
¿Dónde poner la lógica de cada uno?



Solución

Crear varias subclases de *Button*, y cada una de ellas tendrá el código que deberá ejecutarse al hacer clic.





Riesgo a descomponer código

Existen muchas subclases que corren el riesgo de descomponerse en cada modificación de la super clase *Button*.

Es decir, el código GUI se encuentra mezclado con el volátil código de la lógica de negocios.



Duplicidad

Algunas operaciones, como copiar/pegar texto, deben ser invocadas desde varios lugares como menús contextuales, atajos y otros elementos.

Esto provoca que se deba duplicar el código de la operación en muchas clases.



Una mejor solución

Principio de separación de responsabilidades



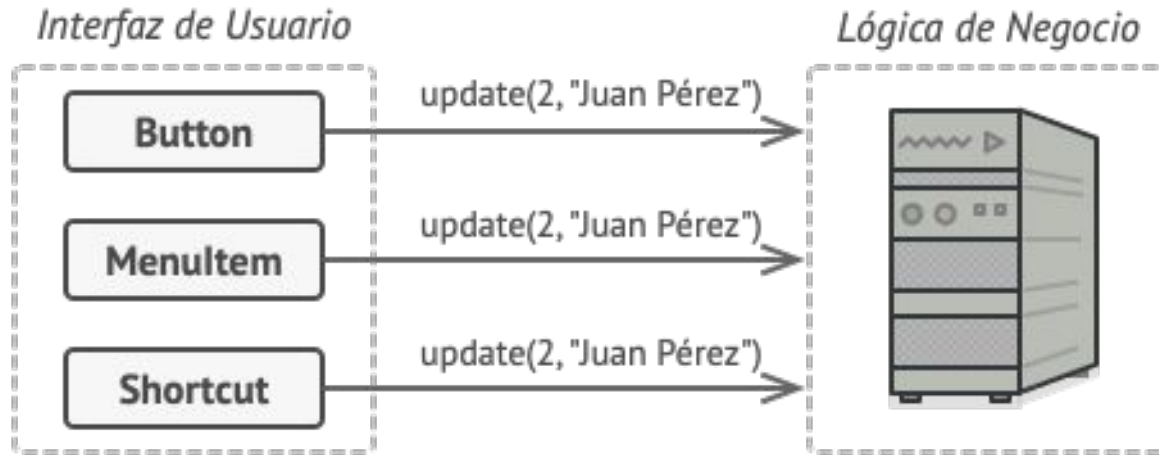
Esto tiene como resultado la división de la aplicación en capas:

Una capa para la interfaz gráfica de usuario (GUI)

- Responsable de representar una bonita imagen en pantalla, capturar entradas y mostrar resultados de lo que el usuario y la aplicación están haciendo.

Otra capa para la lógica de negocio.

- Operaciones más importantes y propias del negocio.



Un objeto GUI invoca a un método de un objeto de la lógica de negocio, pasándole algunos argumentos.


“Un objeto que envía a otro una solicitud”

Patrón Command



Convierte una solicitud en un objeto independiente y contiene toda la información sobre la solicitud.


Esto permite parametrizar los métodos con diferentes solicitudes, retrasar o poner en cola la ejecución de una solicitud y soportar operaciones que no se pueden realizar.



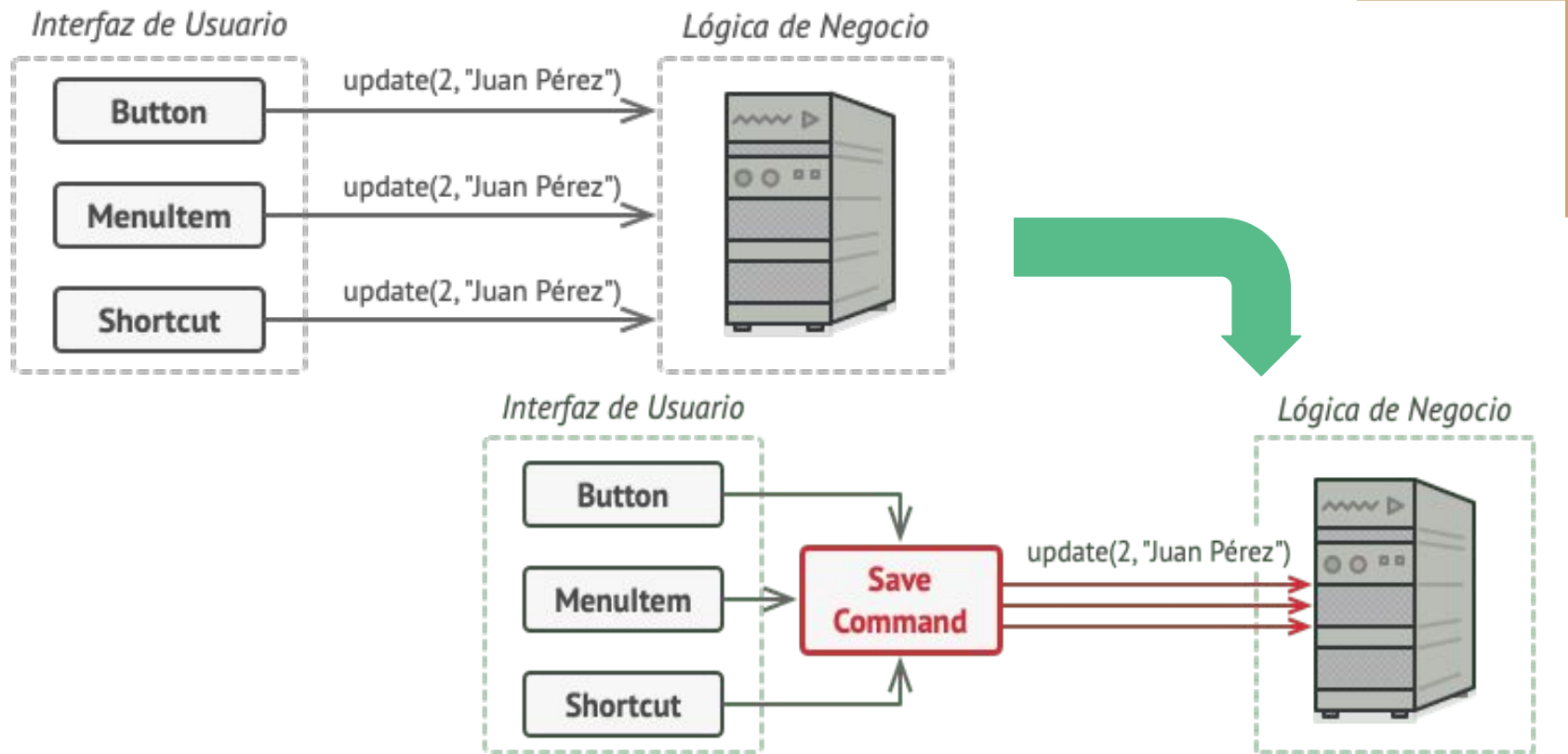


El patrón sugiere que los objetos GUI no envíen estas solicitudes directamente.

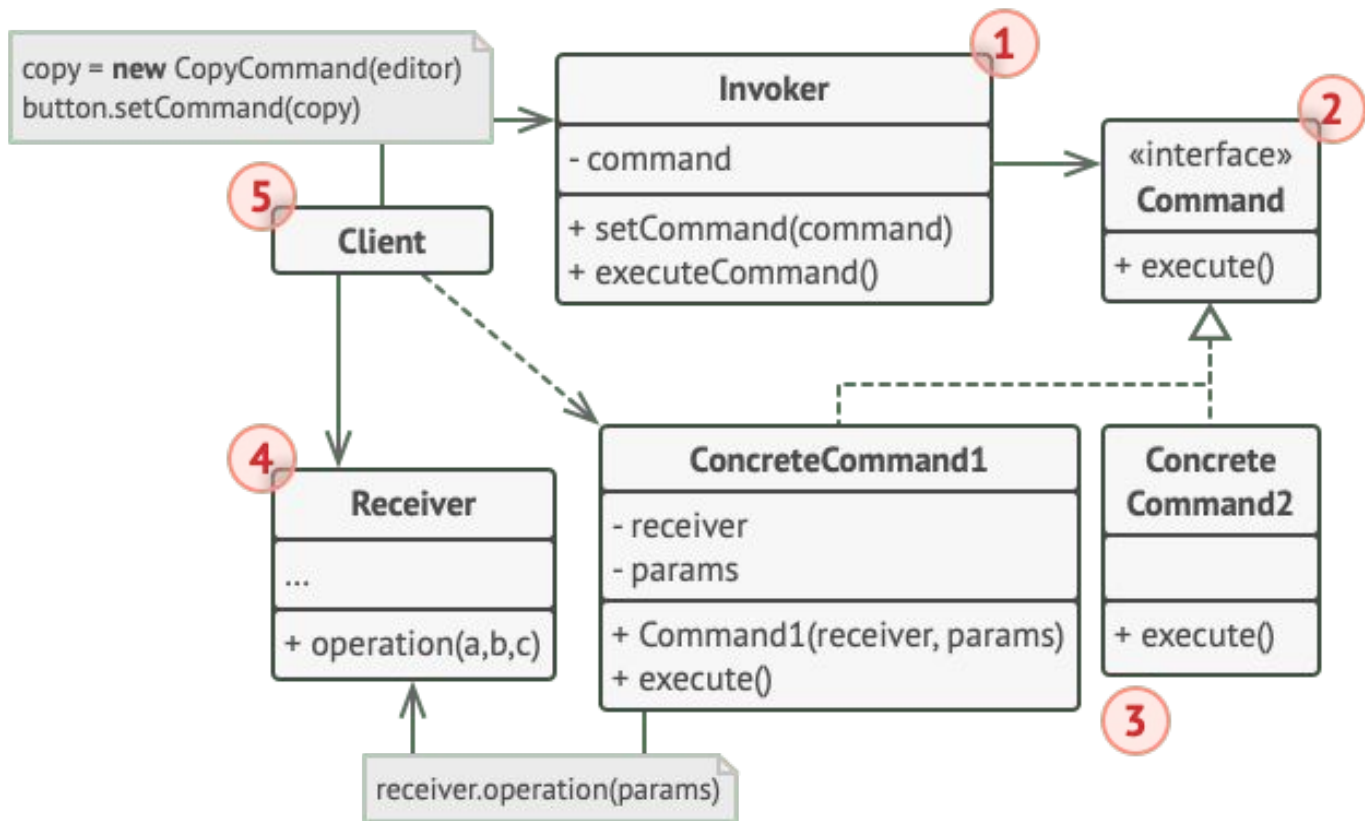
En su lugar, debes **extraer todos los detalles de la solicitud y ponerlos dentro de una clase comando** separada con un **único método que activa esta solicitud**.

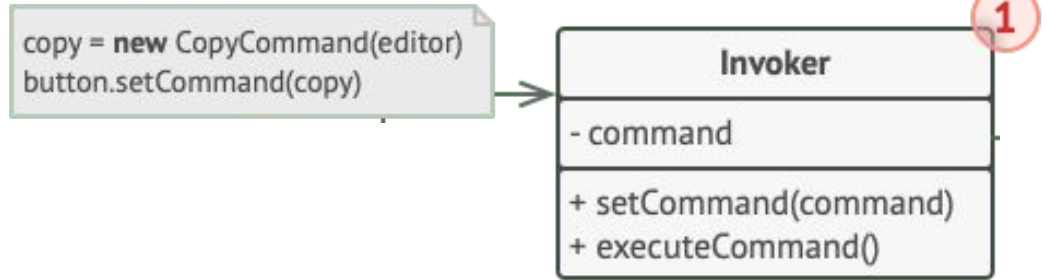


Los objetos de comando sirven como vínculo entre varios objetos GUI y de lógica de negocio. El objeto GUI activa el comando, que gestiona todos los detalles.

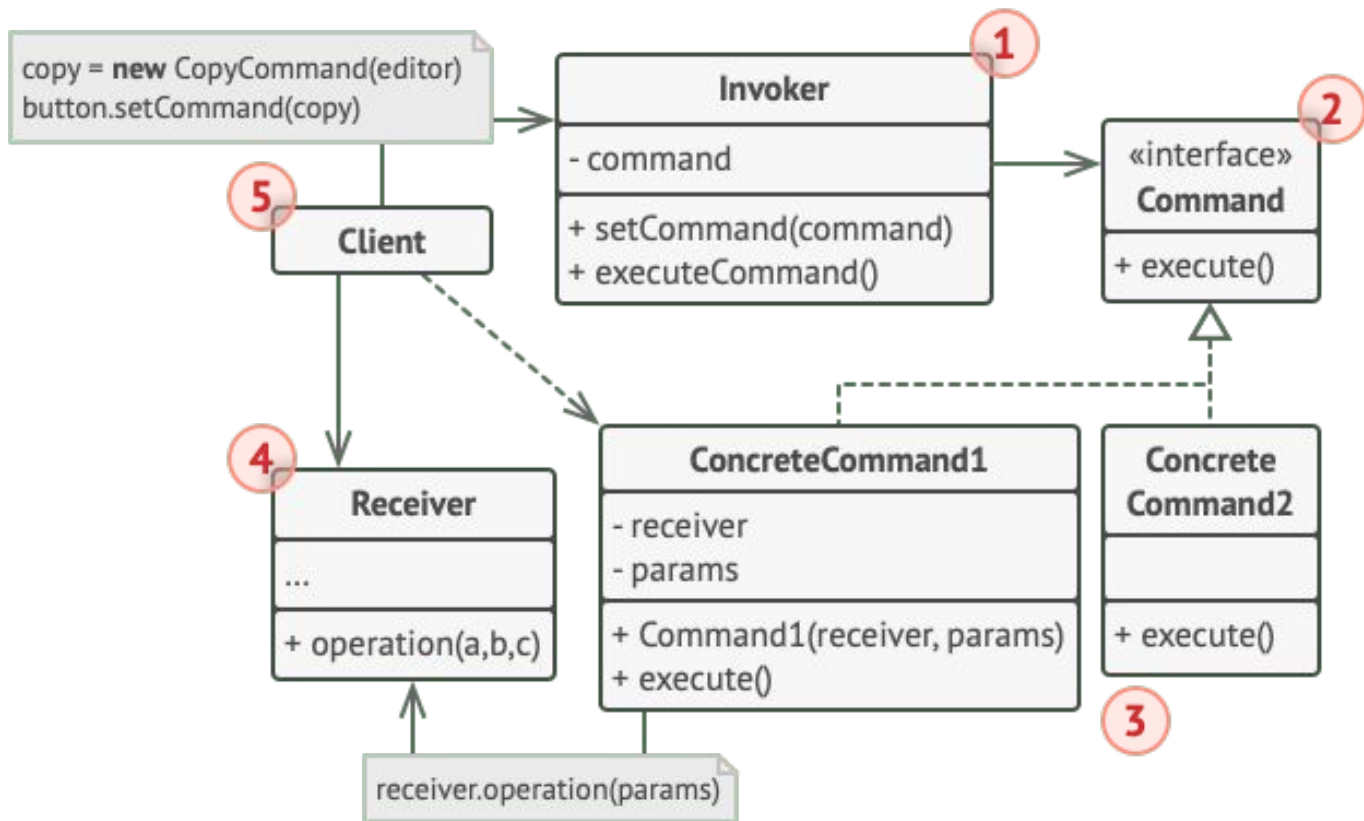


Implementando el patrón Command



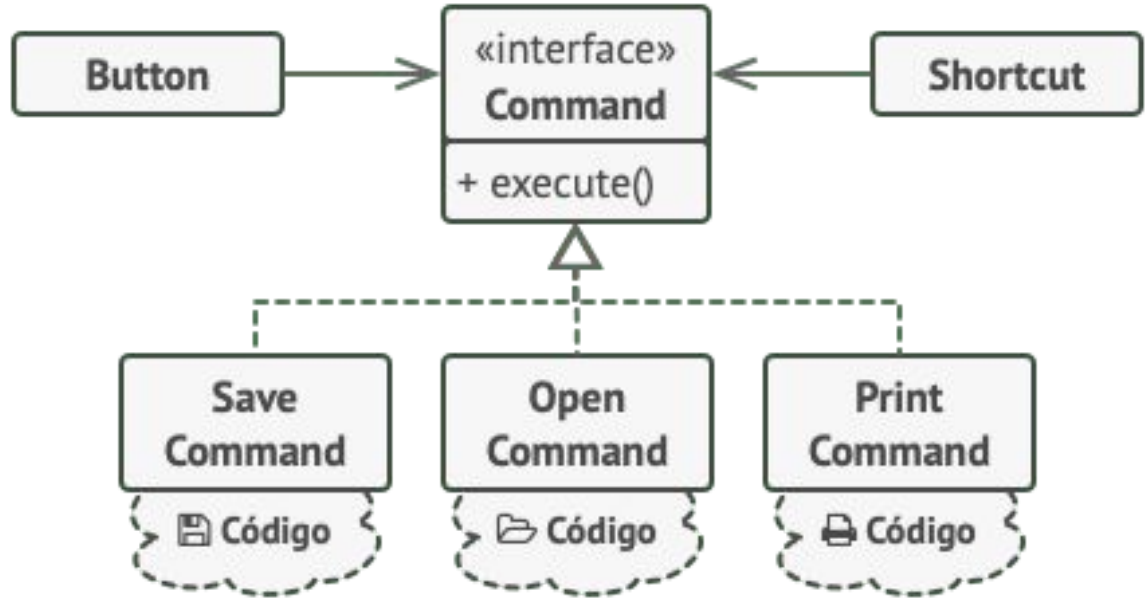


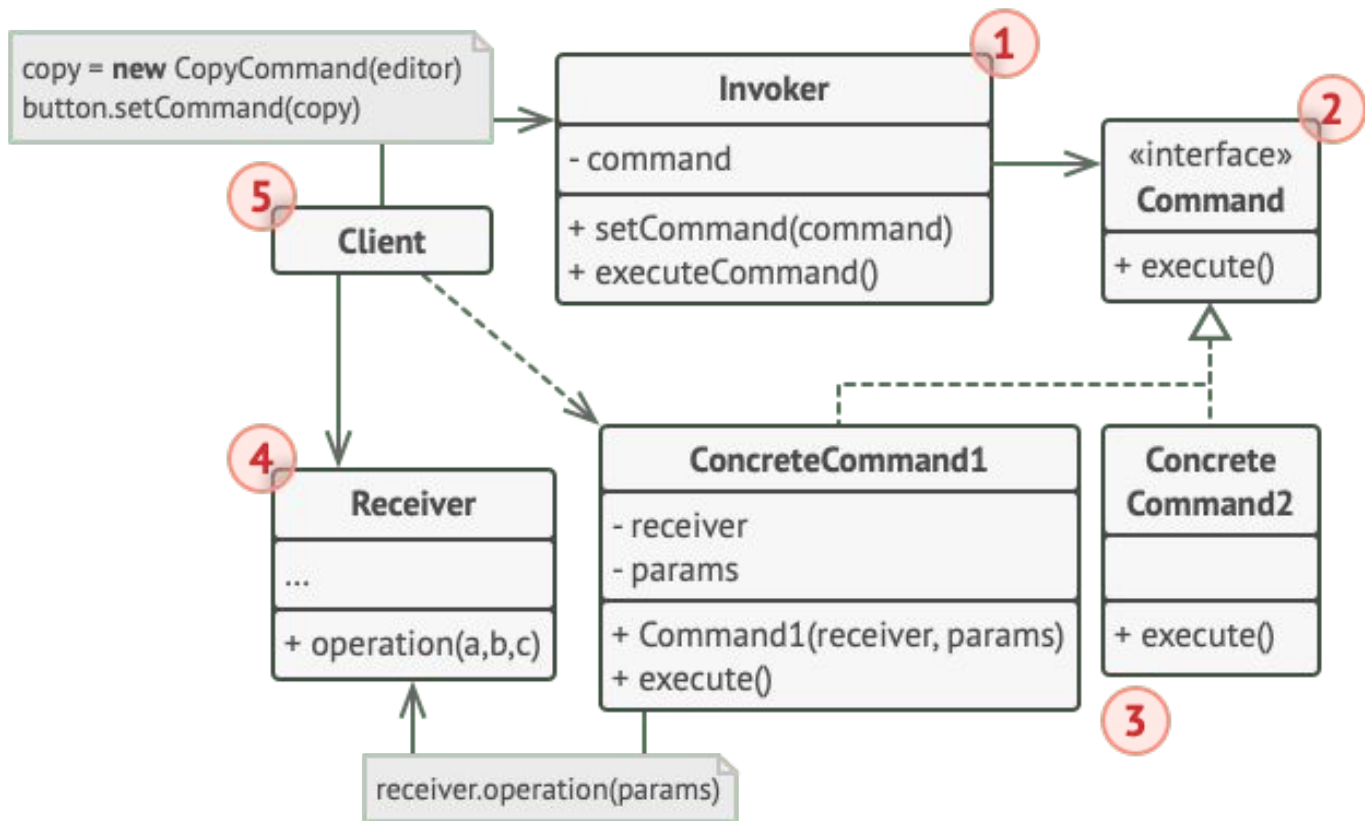
- Debe tener un **campo para almacenar una referencia a un objeto de comando**.
 - **NO es responsable de crear el objeto de comando.**
Normalmente, obtiene un comando precreado de parte del cliente a través del constructor.
- **Responsable de activar este comando** en lugar de enviar la solicitud directamente al receptor.



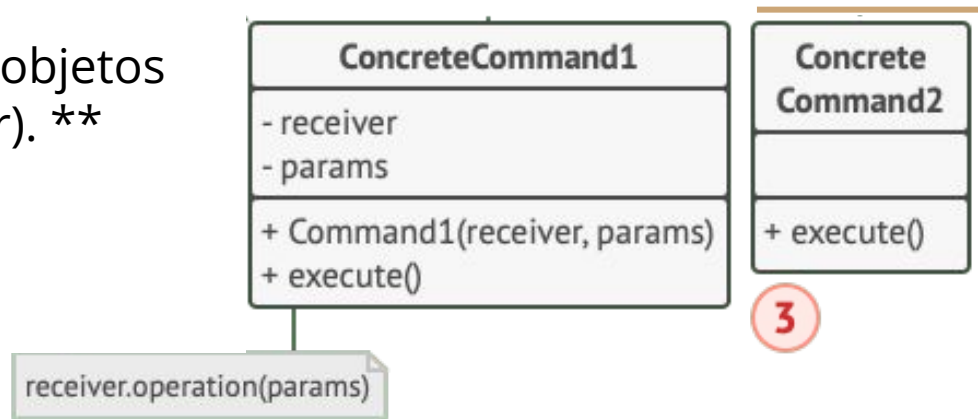
Sugerentemente los comandos deben implementar la misma interfaz.

Esto permite utilizar varios comandos con el mismo emisor de la solicitud, sin acoplarse a clases concretas de comandos.





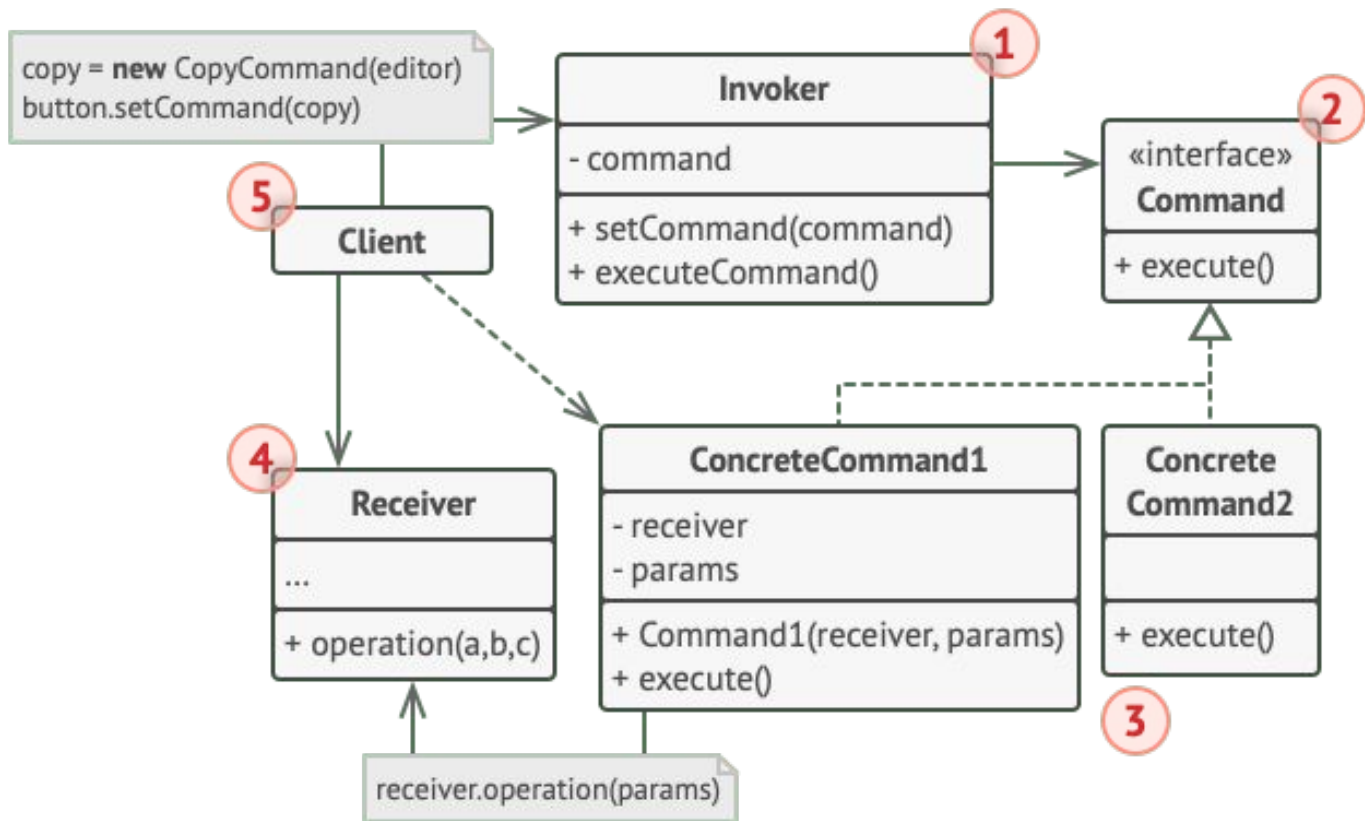
Pasa la llamada a uno de los objetos de la lógica de negocio (receiver). **



Los parámetros necesarios para ejecutar un método en un objeto receptor, **pueden declararse como campos en el comando concreto.**

Puedes hacer inmutables los objetos de comando permitiendo la inicialización de estos campos únicamente a través del constructor.

Un comando puede contener y activar otros comandos.



5

Client

Crea y configura los objetos de comando concretos.

Debe pasar todos los parámetros de la solicitud, incluyendo una instancia del receptor, dentro del constructor del comando.

Después de eso, el comando resultante puede asociarse con uno o varios emisores.

Aplicabilidad



Utiliza el patrón para poner operaciones en cola, programar su ejecución, o ejecutarlas de forma remota.

Utiliza el patrón para implementar operaciones reversibles.



Gracias

