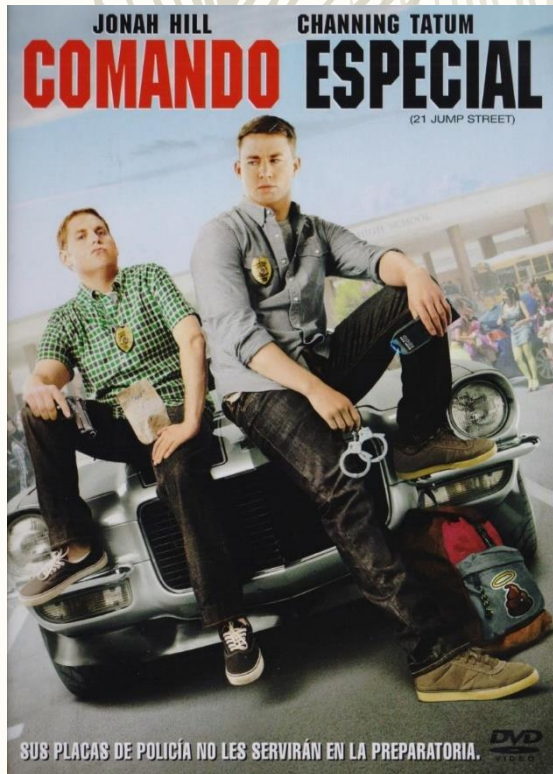


Design

Pattern

Nos Dice...

- Que es un comando.



También se le conoce como...

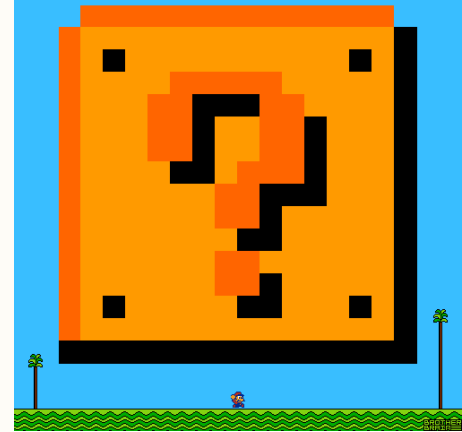
Comando



Y

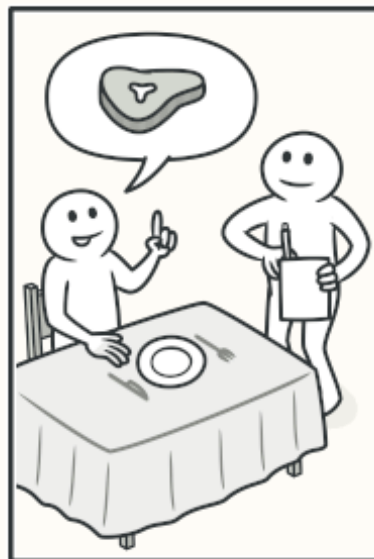
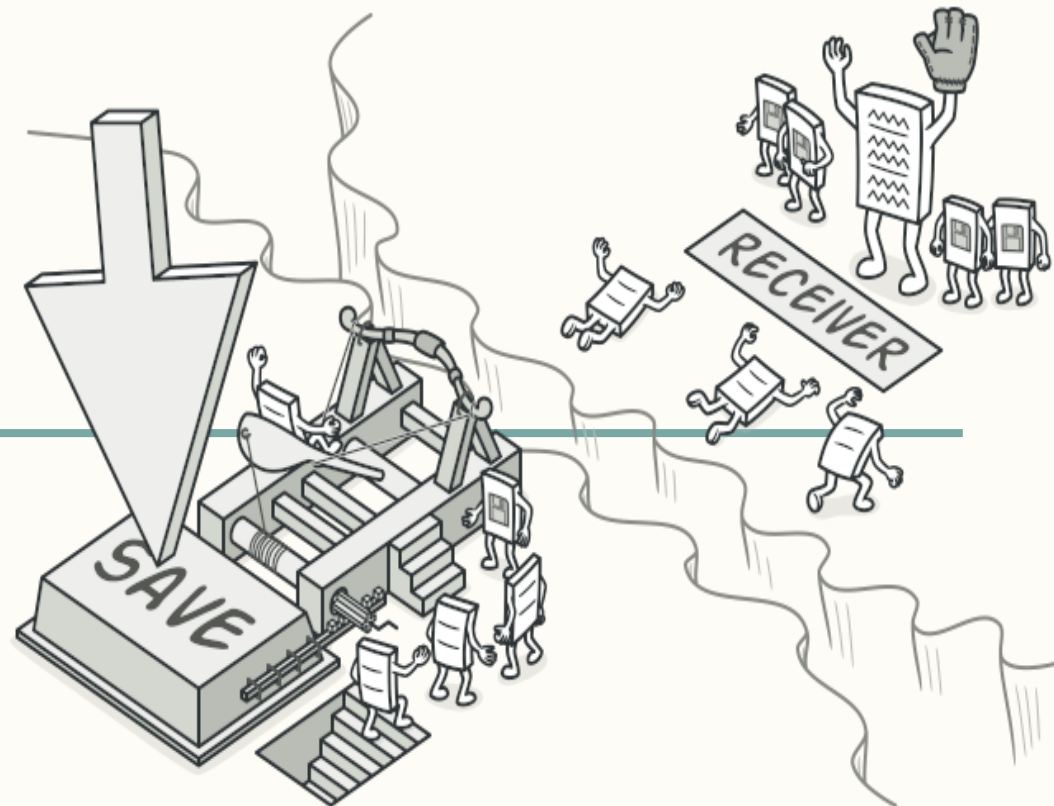
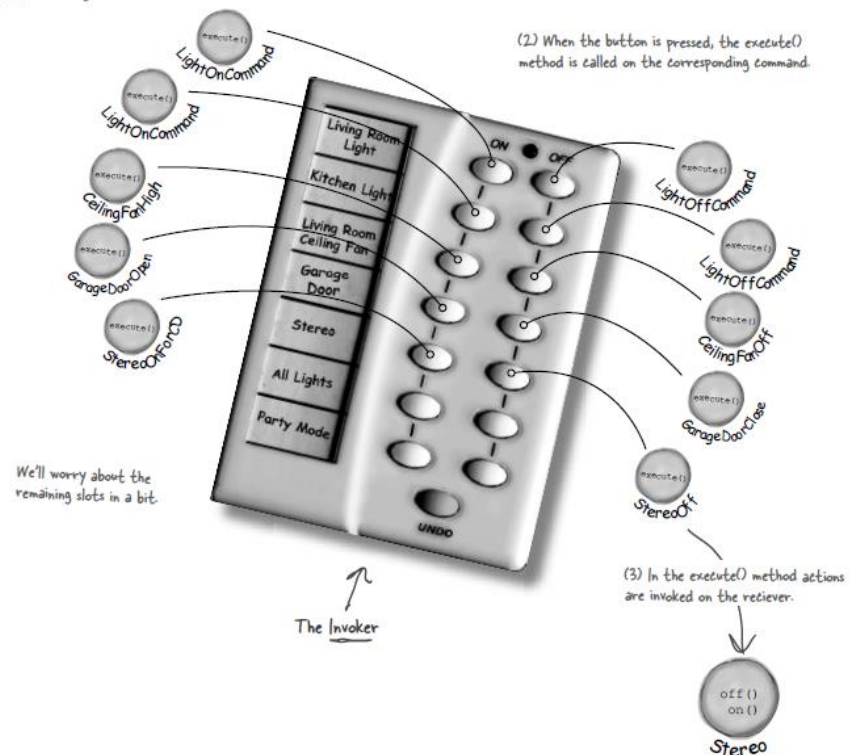
Command

Cuando usar???



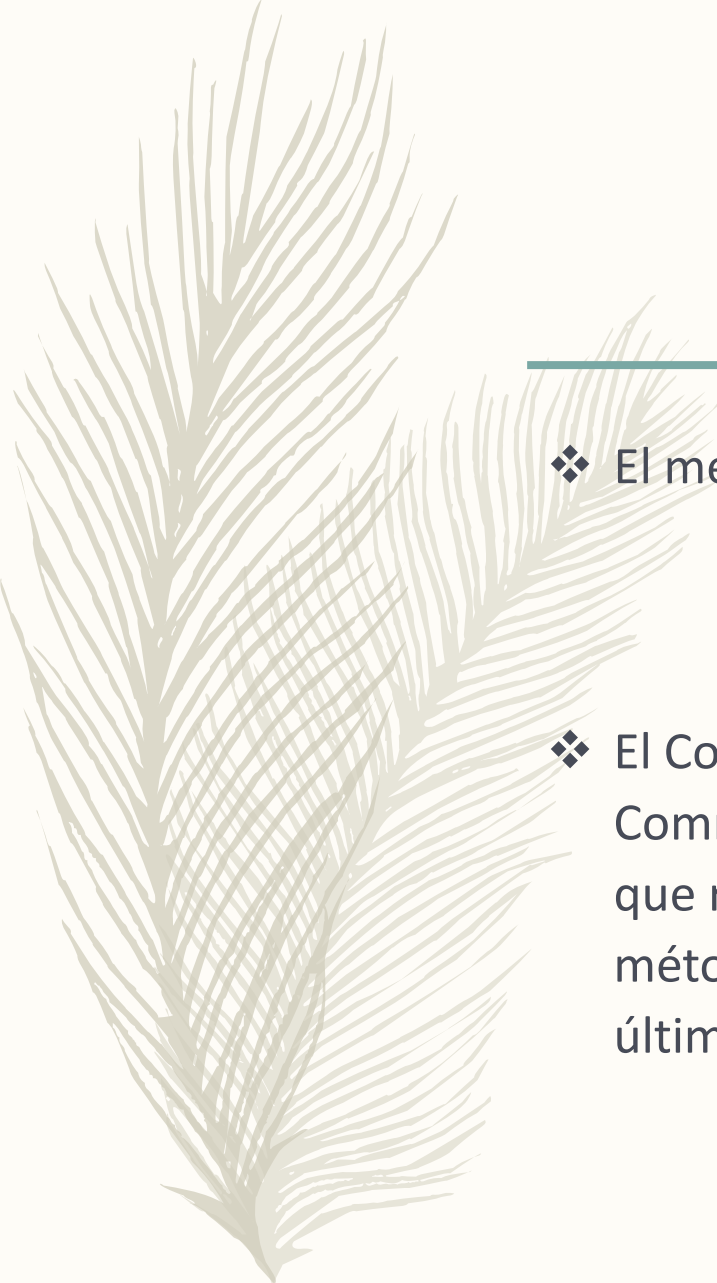
- Resulta útil en escenarios en los que se necesitan enviar peticiones a otros objetos sin saber qué operaciones se han de realizar, y ni tan siquiera quién es el receptor de dicha petición.
- Dicho de otro modo: tenemos varios objetos que realizan acciones similares de forma diferente, y queremos que se procese la adecuada dependiendo del objeto solicitado.
- Permite parametrizar métodos con diferentes solicitudes, retrasar o poner en cola la ejecución de una solicitud y admitir operaciones que no se pueden deshacer.

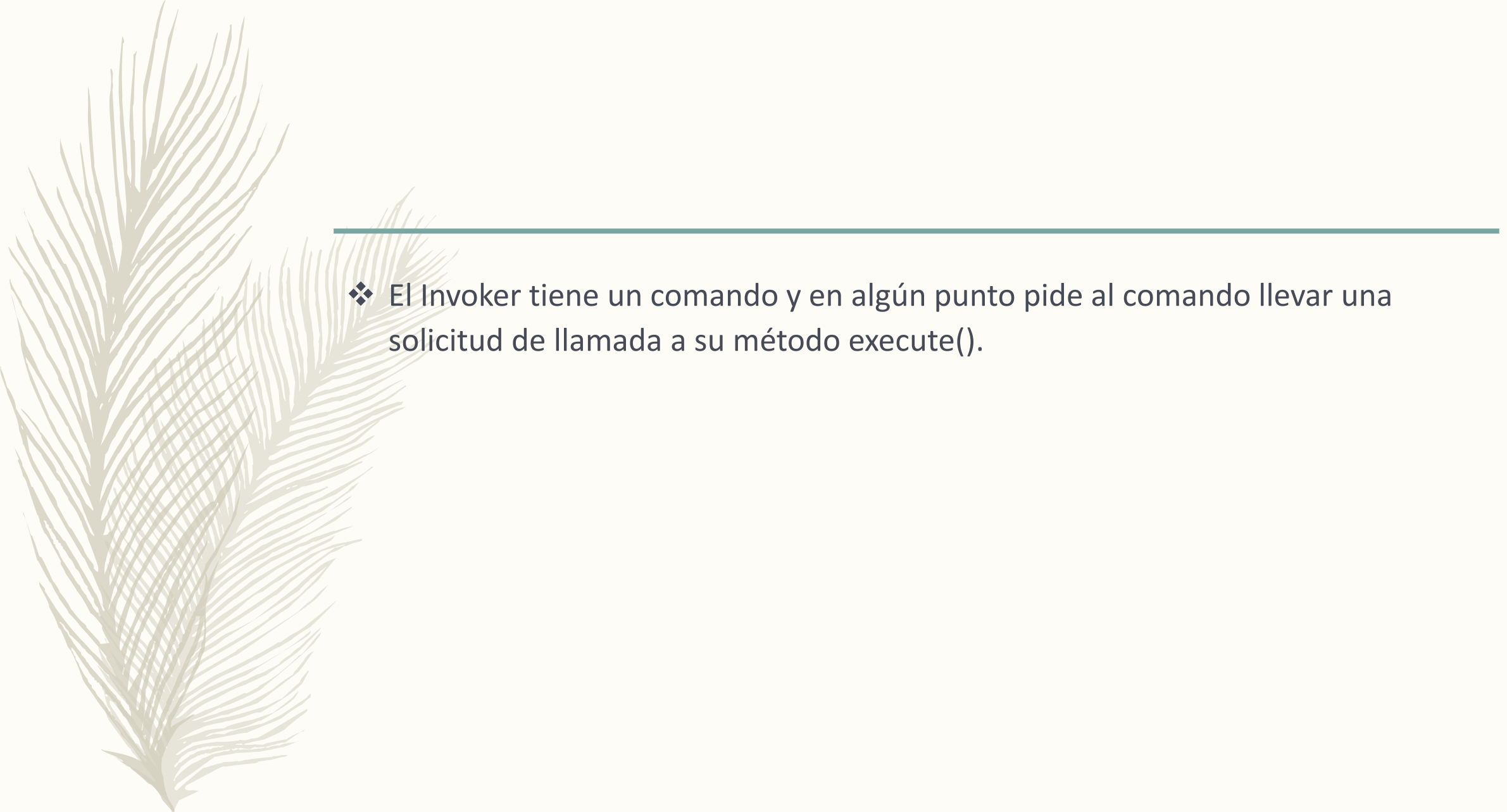
(1) Each slot gets a command.





- ❖ El cliente es responsable de crear un ConcreteCommand y configurar su Receiver.
- ❖ El Receiver sabe cómo realizar el trabajo necesario para llevar a cabo la solicitud, cualquier clase puede actuar como un Receiver.
- ❖ El ConcreteCommand define un enlace entre una action() y un Receiver. El Invoker hace una solicitud llamando execute () y el ConcreteCommand lo lleva a cabo por llamada a una o más acciones en el Receiver.

- 
-
- ❖ El método `execute()` invoca las `action()` en el Receiver para cumplir la solicitud.
 - ❖ El Command declara una interfaz para todos los comandos. Como ya sabes, un Command se invoca a través de su método `execute()`, que le pide a un Receiver que realice un `action()`. Cuando los comandos admiten deshacer, tienen un método `undo()` que refleja el método `execute()`. Lo que sea que ejecute por última vez, `undo()` lo deshace.

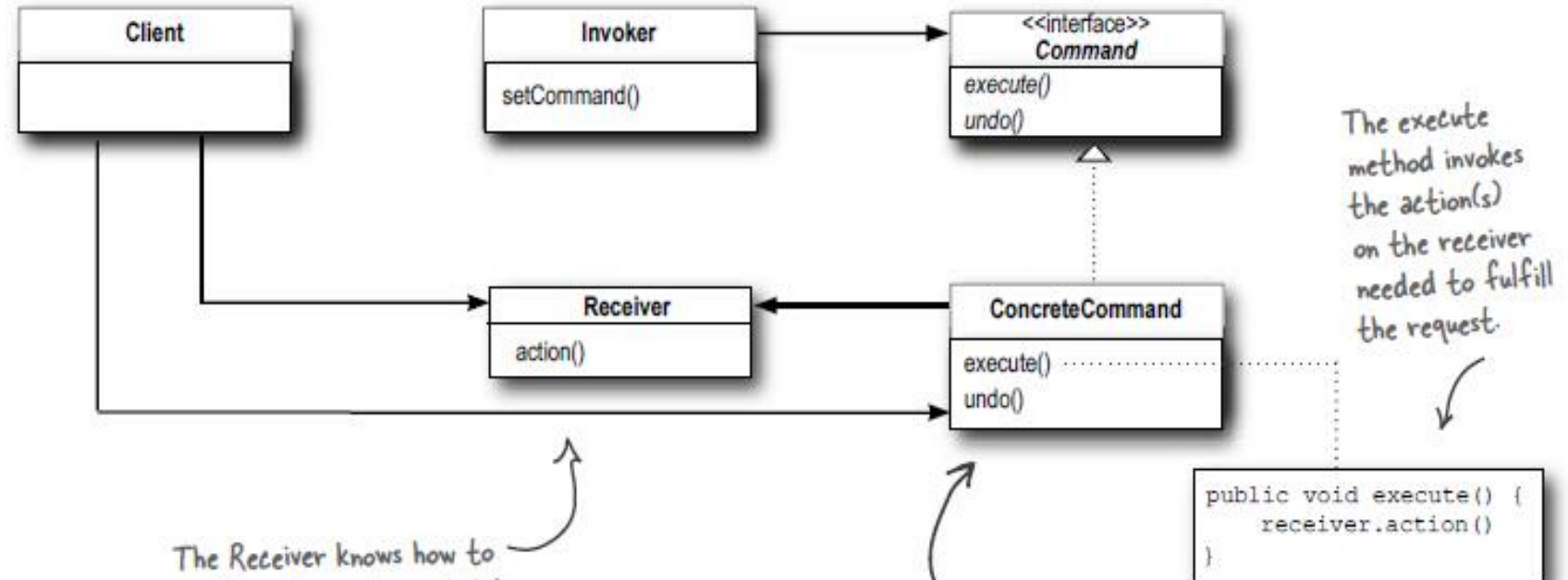
- 
- ❖ El Invoker tiene un comando y en algún punto pide al comando llevar una solicitud de llamada a su método `execute()`.

UML

The Client is responsible for creating a ConcreteCommand and setting its Receiver.

The Invoker holds a command and at some point asks the command to carry out a request by calling its execute() method.

Command declares an interface for all commands. As you already know, a command is invoked through its execute() method, which asks a receiver to perform an action. You'll also notice this interface has an undo() method, which we'll cover a bit later in the chapter.



The Receiver knows how to perform the work needed to carry out the request. Any class can act as a Receiver.

The **ConcreteCommand** defines a binding between an action and a **Receiver**. The **Invoker** makes a request by calling **execute()** and the **ConcreteCommand** carries it out by calling one or more actions on the **Receiver**.

The **execute** method invokes the action(s) on the receiver needed to fulfill the request.

PROS

- ❖ *Principio de responsabilidad única* . Puede desacoplar clases que invocan operaciones de clases que realizan estas operaciones.
- ❖ *Principio abierto / cerrado* . Puede introducir nuevos comandos en la aplicación sin romper el código del cliente existente.
- ❖ Puede implementar deshacer / rehacer.
- ❖ Puede implementar la ejecución diferida de operaciones.
- ❖ Puede ensamblar un conjunto de comandos simples en uno complejo.


CONS

- ❖ El código puede volverse más complicado ya que está introduciendo una capa completamente nueva entre remitentes y receptores.
- ❖ Puede aumentar el volumen de nuestro código.



Relaciones

- ❖ Mediator: elimina las conexiones directas entre remitentes y receptores, obligándolos a comunicarse indirectamente a través de un objeto mediator.
- ❖ Observer: permite a los receptores suscribirse dinámicamente y des-Subscribirse de recibir solicitudes.
- ❖ Command y Memento juntos al implementar "deshacer undo()". En este caso, los commands son responsables de realizar varias operaciones sobre un objeto de destino, mientras que los mementos guardan el estado de ese objeto justo antes de que se ejecute un command.

- 
-
- ❖ El Command y el Strategy pueden ser similares porque puede usar ambos para parametrizar un objeto con alguna acción.
 - ❖ Prototype puede ayudar cuando necesita guardar copias de Command en el historial.
 - ❖ Puede tratar a Visitor como una versión poderosa del patrón de Command. Sus objetos pueden ejecutar operaciones sobre varios objetos de diferentes clases.



Usos Cotidianos...

- ❖ La arquitectura [CQRS](#), que aboga por la separación de los contextos de lectura y escritura en nuestra aplicación, usualmente hace uso de buses de comandos, de forma que estos puedan ejecutarse síncrona o asíncronamente según requiera el caso de uso.
- ❖ Las librerías de CommandBus, como Tactician o Broadway hacen uso del patrón.
- ❖ Las aplicaciones de escritorio basadas en ventanas hacen uso de una variante del patrón command, que permite además que las operaciones puedan deshacerse (el clásico Ctrl+Z).