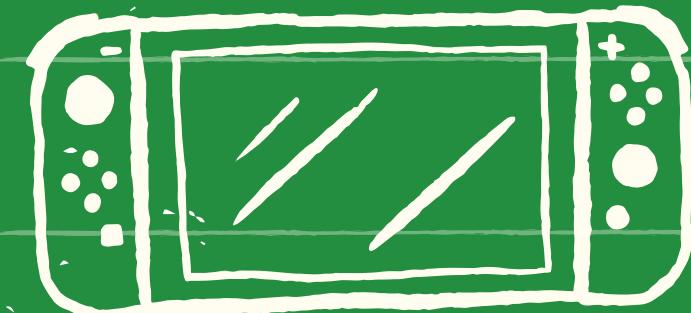
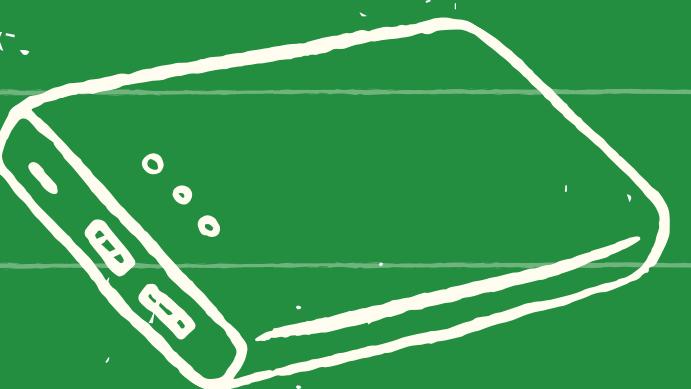


Empowerment Technology

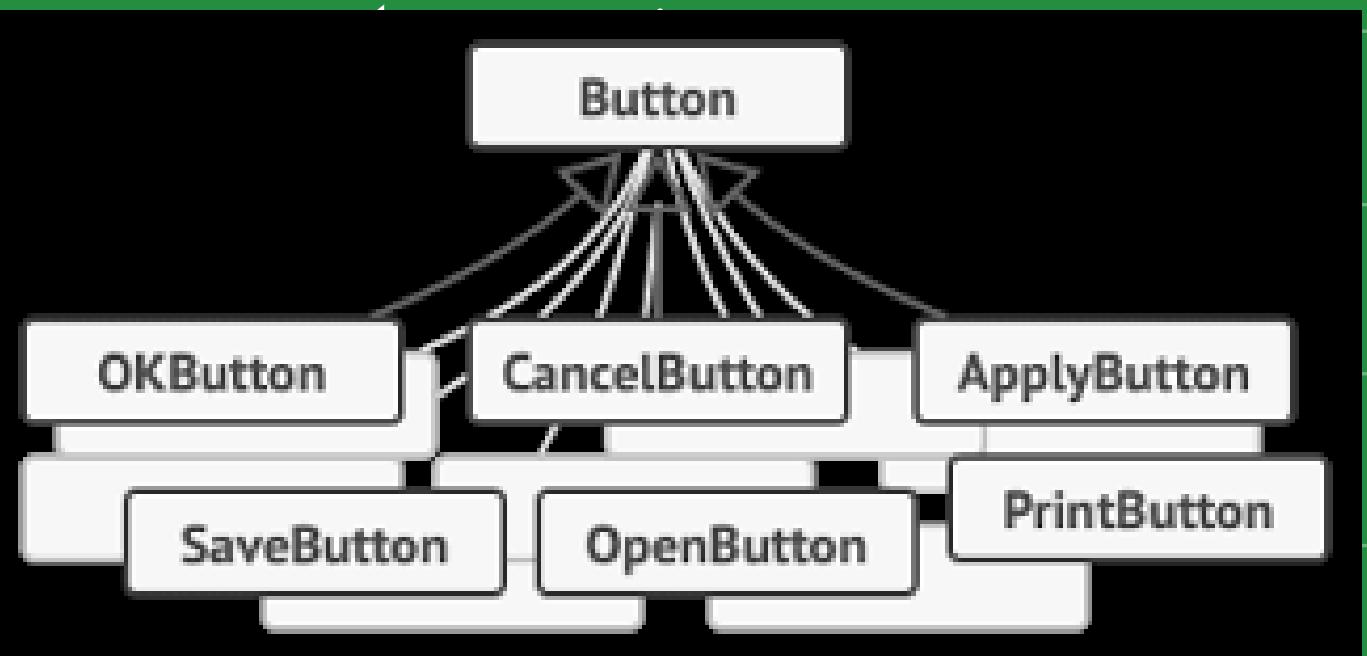
Information and Communication Technology

Geiner Montoya

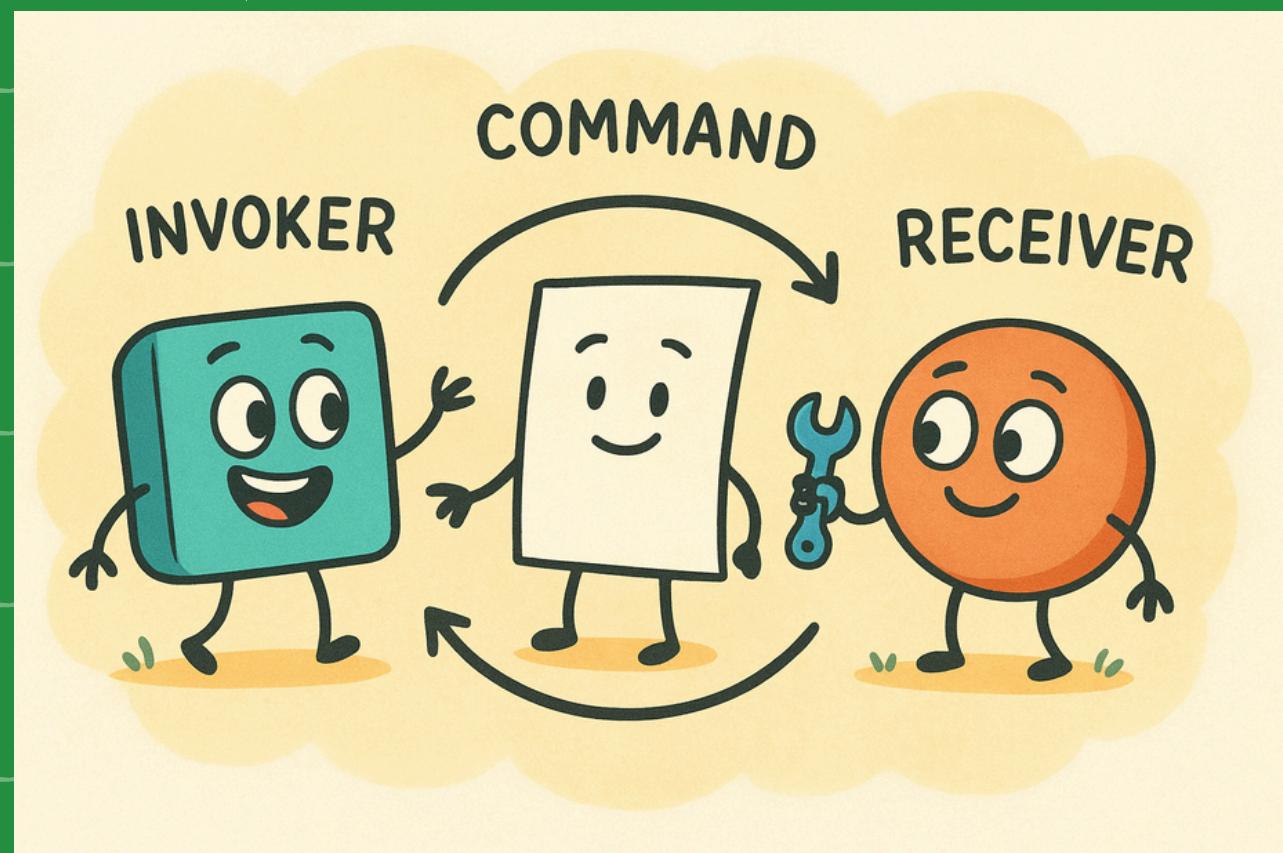


Que es Command?

El Patrón Command es un patrón de diseño de comportamiento que tiene como objetivo encapsular una solicitud (una acción a realizar, junto con todos sus parámetros) como un objeto.



Problema que resuelve



El problema principal que resuelve el Patrón Command es el desacoplamiento entre el objeto que emite la solicitud (Invoker) y el objeto que sabe cómo realizarla (Receiver).

Escalabilidad

- Desacoplamiento: Al separar el invocador del receptor, se consigue un acoplamiento débil.

Puedes cambiar o extender el receptor o el comando concreto sin modificar la clase del invocador, lo cual facilita la extensibilidad y el mantenimiento.

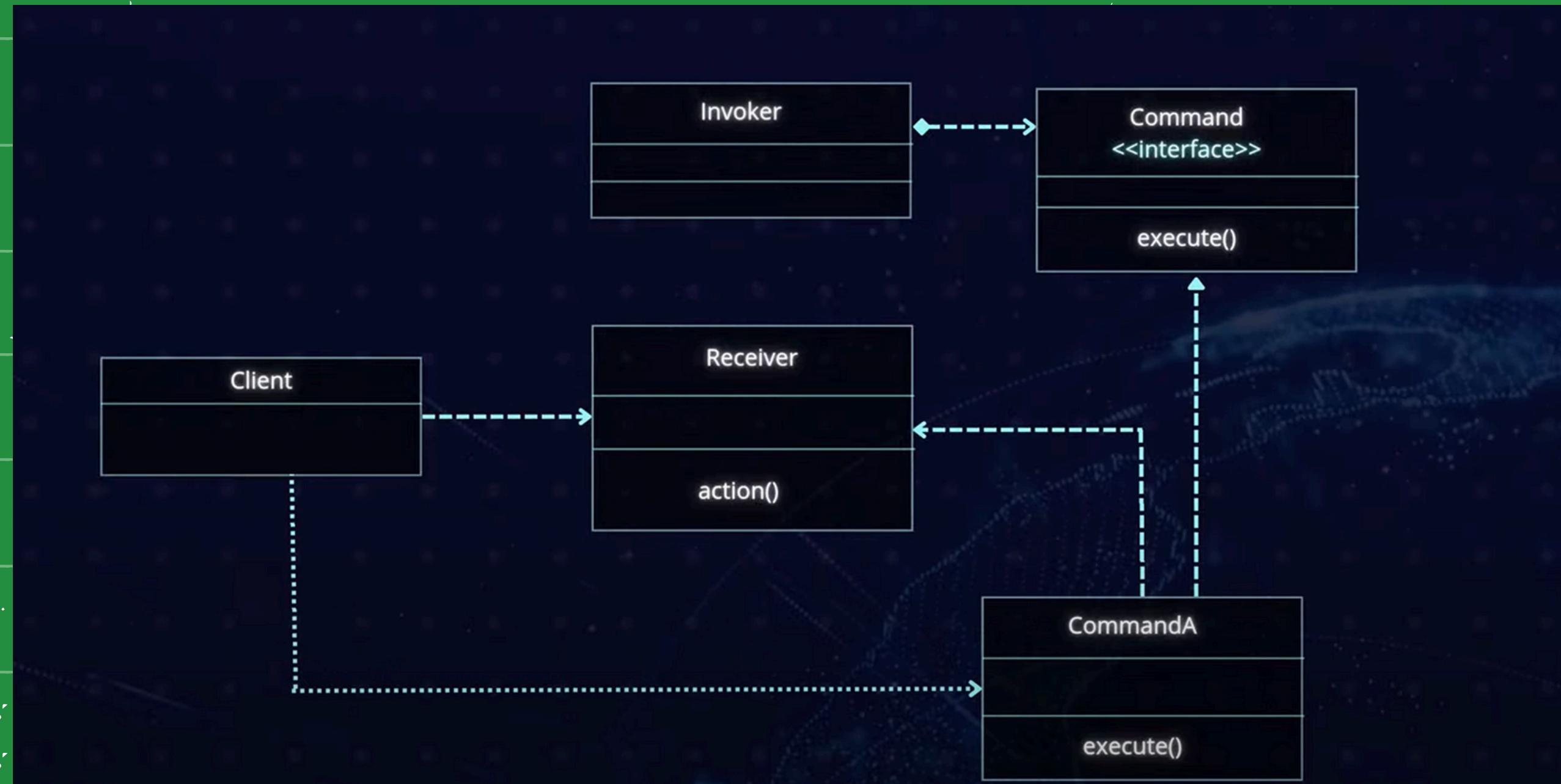
Escalabilidad

- Extensión de Comandos: Agregar una nueva acción o comando es tan sencillo como crear una nueva clase `ConcreteCommand` sin necesidad de modificar el código existente del invocador o de otros comandos (cumpliendo el Principio Abierto/Cerrado).
- Flexibilidad: Permite implementar fácilmente funcionalidades avanzadas como colas de comandos.

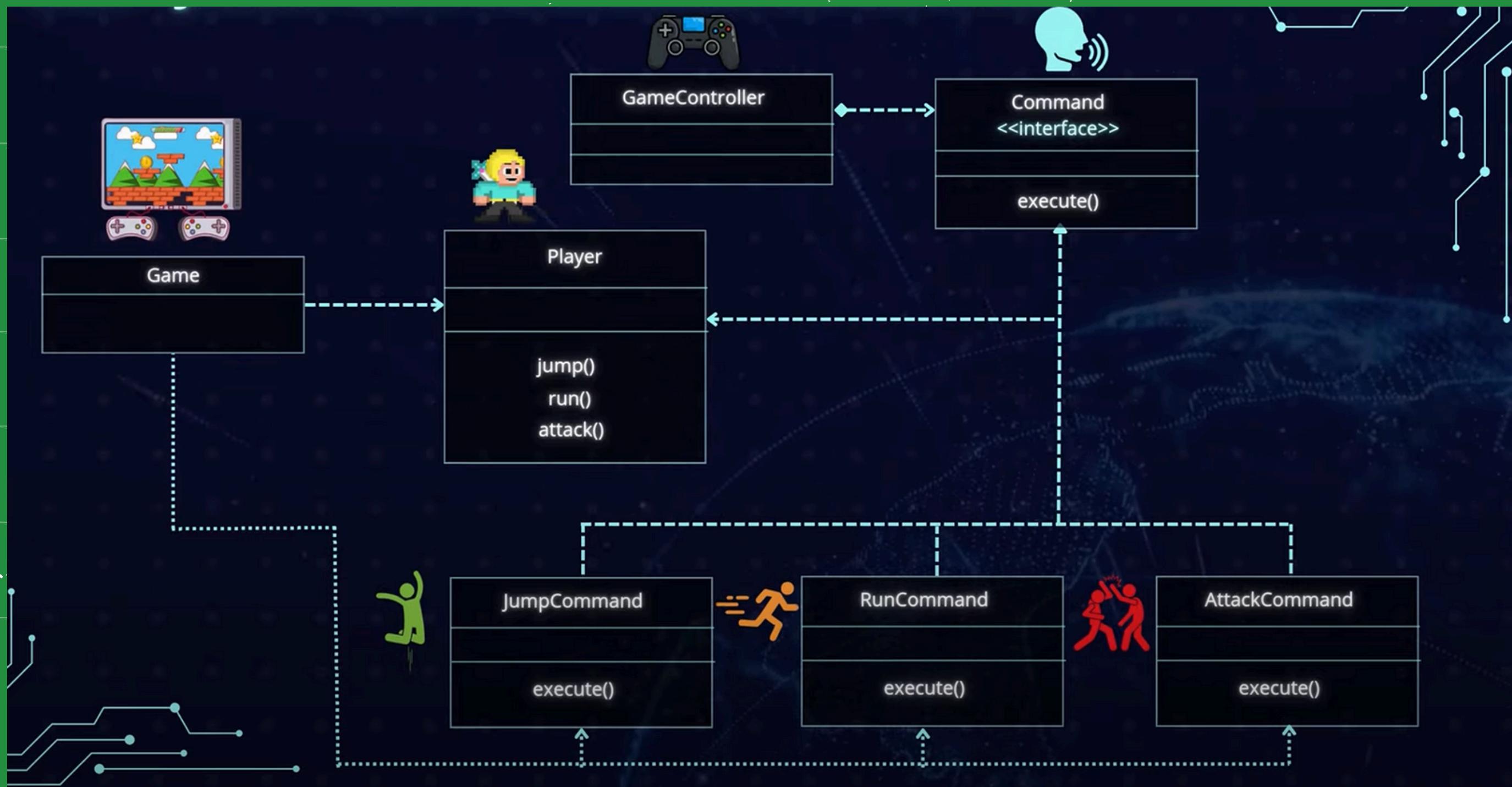
Ventajas y desventajas

- Desacoplamiento del Invocador y Receptor.
- Soporte para operaciones Deshacer/Rehacer (Undo/Redo).
- Facilita el uso de Colas de Comandos y procesamiento asíncrono.
- Aumento de la complejidad y de la cantidad de clases.
- Mayor sobrecarga para tareas muy simples.
- Posible aumento del uso de memoria (al almacenar el historial de comandos).

Ejemplo de programa



Ejemplo de programa



Ejemplo de programa

```
# --- 1. El Receptor (Receiver) ---
```

```
class Player:
```

```
    """
```

```
    El objeto que realiza las acciones reales (la lógica del juego).
```

```
    Es el 'Receptor' del comando.
```

```
    """
```

```
    def jump(self):
```

```
        print("Player: Saltando...")
```

```
    def run(self):
```

```
        print("Player: Corriendo a toda velocidad!")
```

```
    def attack(self):
```

```
        print("Player: Atacando al enemigo!")
```

```
# --- 2. La Interfaz del Comando (Command) ---
```

```
class Command:
```

```
    """Define la interfaz para todos los comandos del juego."""
```

```
    def execute(self):
```

```
        raise NotImplementedError
```

```
# --- 3. Comandos Concretos (ConcreteCommands) ---
```

```
class JumpCommand(Command):
```

```
    """Encapsula la solicitud de Saltar."""
```

```
    def __init__(self, player: Player):
```

```
        self._player = player
```

```
    def execute(self):
```

```
        """Llama al método 'jump' del Receptor."""
```

```
        self._player.jump()
```

```
class RunCommand(Command):
```

```
    """Encapsula la solicitud de Correr."""
```

```
    def __init__(self, player: Player):
```

```
        self._player = player
```

```
    def execute(self):
```

```
        """Llama al método 'run' del Receptor."""
```

```
        self._player.run()
```

```
class AttackCommand(Command):
```

```
    """Encapsula la solicitud de Atacar."""
```

```
    def __init__(self, player: Player):
```

```
        self._player = player
```

```
    def execute(self):
```

```
        """Llama al método 'attack' del Receptor."""
```

```
        self._player.attack()
```

Ejemplo de programa

```
# --- 4. El Invocador (Invoker) ---
class GameController:
    """
    La interfaz de control (ej. un gamepad o teclado).
    Mantiene una referencia a un Command y NO conoce la clase Player.
    """
    def __init__(self):
        self._button_A_command = None # Comando asignado al botón A

    def set_command(self, command: Command):
        """Configura el comando para el botón de acción."""
        self._button_A_command = command

    def press_button_A(self):
        """
        Dispara la acción. Solo llama a execute() del Command.
        Está desacoplado de la lógica del juego.
        """
        if self._button_A_command:
            print("\nControlador: Botón A presionado.")
            self._button_A_command.execute()
        else:
            print("\nControlador: Botón A no tiene comando asignado.")

    def press_button_B(self):
        """
        Dispara la acción. Solo llama a execute() del Command.
        Está desacoplado de la lógica del juego.
        """
        if self._button_B_command:
            print("\nControlador: Botón B presionado.")
            self._button_B_command.execute()
        else:
            print("\nControlador: Botón B no tiene comando asignado.")
```

```
# --- Cliente: Configuración y Uso ---

# 1. Crear el Receptor (el objeto que realizará las acciones)
mario = Player()

# 2. Crear el Invocador (el control del juego)
controller = GameController()

# --- Configuración 1: El Cliente asigna el Salto al botón A ---
jump_cmd = JumpCommand(mario)
controller.set_command(jump_cmd)

print("--- Configuración: Botón A = SALTAR ---")
controller.press_button_A() # Mario Salta

# --- Configuración 2: El Cliente reasigna Correr al botón A ---
run_cmd = RunCommand(mario)
controller.set_command(run_cmd)

print("\n--- Reconfiguración: Botón A = CORRER ---")
controller.press_button_A() # Mario Corre

# --- Configuración 3: El Cliente asigna Atacar al botón A ---
attack_cmd = AttackCommand(mario)
controller.set_command(attack_cmd)

print("\n--- Reconfiguración: Botón A = ATACAR ---")
controller.press_button_A() # Mario Ataca
```