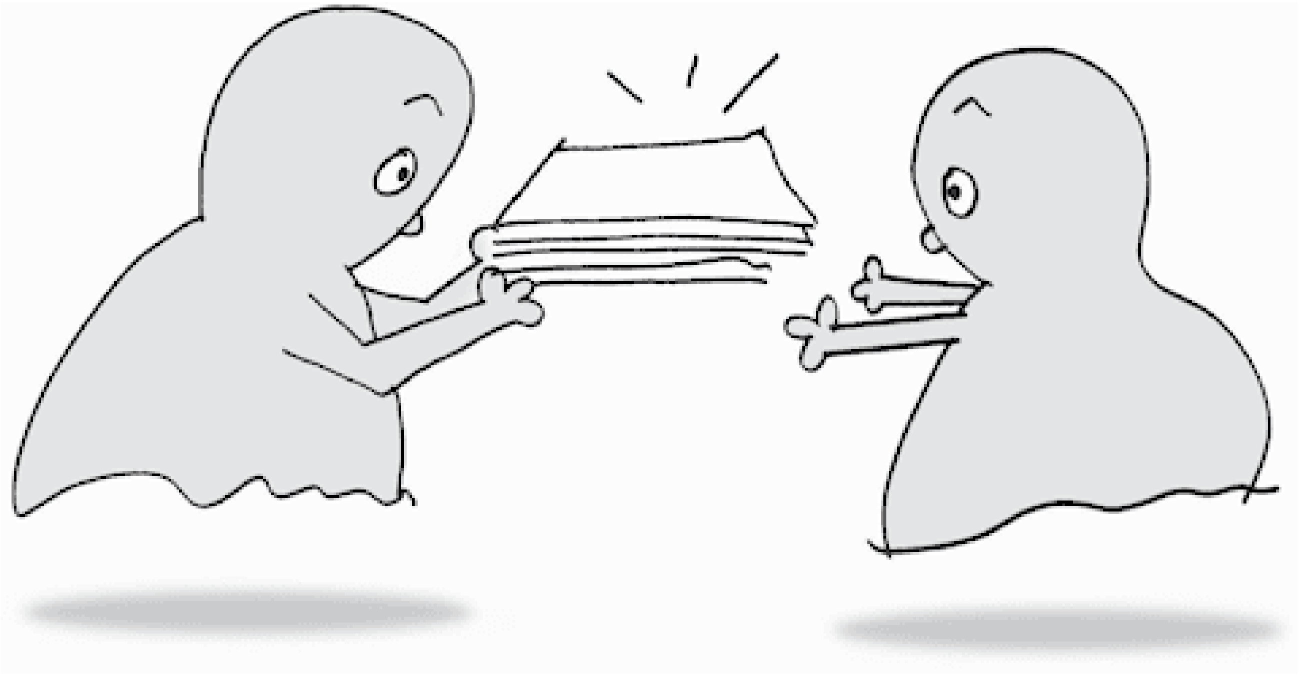
Command Pattern

Encapsulate a Request as an Object to Parameterize Clients and Support Queuing, Logging, and Undo



Command Pattern

Think of **requests (Commands)** that need to be handled in flexible ways:

- **Restaurant order**: Waiter takes order (command), passes to kitchen (receiver), enables cancellation/modification
- **Remote control**: Each button creates a specific command for the TV/stereo (receivers)

The Problem

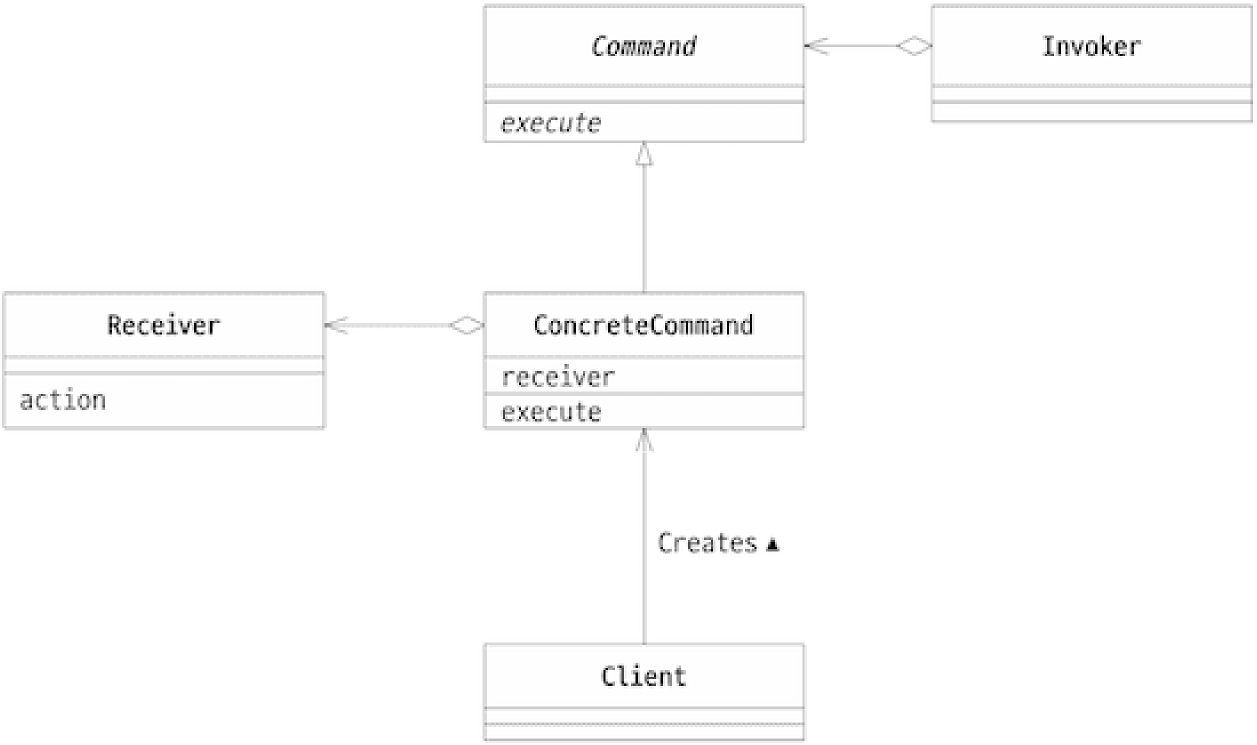
- We have a **drawing application** where users create drawings with a mouse/keyboard.
- We need **undo functionality** to reverse user actions.
- We want to **record and replay** drawing sequences (macro commands).
- Traditional approach: UI components directly calling drawing methods → no undo, hard to extend.

The challenge: how to parameterize UI elements with operations and support flexible request handling?

The *Command* as the Solution

- **Encapsulate** each request as a command object with **execute()** method.
- **Command objects** store the receiver and all parameters needed.
- **Invokers** (UI) work with command interface, **receivers** perform actual work.

The Solution (Design)



Step 1: Understand the Players

In this design, we have four key components:

1. Command defines an interface for executing operations
2. The receiver receives the command, and it knows how to perform the actual operations
3. Invoker handles the command to carry out the request (Command Executor)
4. Client creates Command

In a restaurant

🧑‍💼 Client (Customer)

- Creates orders
- Knows what to order
- Sets up the command-receiver relationship

📋 Command (Order)

- Encapsulates the request
- Binds receiver with action
- Contains all info needed

🧑‍🍳 Receiver (Chef)

- Does the actual work
- Knows how to cook
- Performs the operation

🤵 Invoker (Waiter)

- Manages commands
- Executes when ready
- Doesn't know cooking details

```
👦 Customer → 📋 Order → 🤵 Waiter → 👨 Chef
   (Client)    (Command)  (Invoker)   (Receiver)
```

Flow:

1. **Customer** creates **Order** with specific **Chef** and dish
2. **Customer** gives **Order** to **Waiter**
3. **Waiter** stores **Order** (maybe multiple orders)
4. **Waiter** tells **Order** to execute when ready
5. **Order** tells **Chef** to cook the specific dish

Step 2: Decoupling through abstraction

- **Invokers** depend only on the Command interface.
- **ConcreteCommands** encapsulate receiver details.

Step 3: Understand the Command Interface

- *Command* defines **execute()** method for performing operations.
- **Common interface** allows treating different commands uniformly.
- **May include** undo(), isUndoable() for advanced functionality.

Step 4: Understand ConcreteCommand

- **ConcreteCommand** binds **receiver** with specific **action**.
- **Stores parameters** needed for the operation.
- **execute()** method calls the appropriate receiver method with stored parameters.
- **May store state** for undo operations.

Step 5: Understand Receiver and Invoker

- **Receiver**: Object that performs the actual work (DrawCanvas, Light, etc.)
- **Invoker**: Object that holds commands and calls execute() (MacroCommand, RemoteControl)

Separation: The invoker doesn't know which receiver or operation will be performed.

Code

- Main Method
- Command Interface
- ConcreteCommand Implementation
- MacroCommand (Invoker)

Main Method

```python
from draw_canvas import DrawCanvas
from draw_command import DrawCommand
from macro_command import MacroCommand

def main():
    # Receiver
    living_room = Light("Living Room")
    kitchen = Light("Kitchen")

    # The main (client) creates the Command
    # Commands
    living_on = LightOnCommand(living_room)
    living_off = LightOffCommand(living_room)
    kitchen_on = LightOnCommand(kitchen)
    kitchen_off = LightOffCommand(kitchen)

    # The main (invoker) executes the command
    living_on.execute()
    kitchen_on.execute()
```

Step 1: Create receiver

For simplicity, we use a CUI application for the explanation, so in this example, the Client is the main method.

```
# Receiver
living_room = Light("Living Room")
kitchen = Light("Kitchen")
```

We have two Light receivers: one is living_room and the other is kitchen.

Step 2: Create command objects & Invoker

```
# Commands
living_on = LightOnCommand(living_room)
living_off = LightOffCommand(living_room)
kitchen_on = LightOnCommand(kitchen)
kitchen_off = LightOffCommand(kitchen)

# Invoker
history = MacroCommand()
```

- **LightOnCommand/LightOffCommand** encapsulates on/off request
- **Each command** knows its receiver (living_room/kitchen) and possibly operation parameters.

Step 3: Queue and execute commands

```
history.add(living_off)
history.add(kitchen_off)
history.execute()    # Execute all commands in sequence
```

- **Commands queued** for execution without immediate execution.
- **Batch execution** allows for macro operations and replay functionality.

Command Interface

```python
from abc import ABC, abstractmethod

class Command(ABC):
    @abstractmethod
    def execute(self):
        """Execute the command."""
        pass
```

ConcreteCommand Implementation

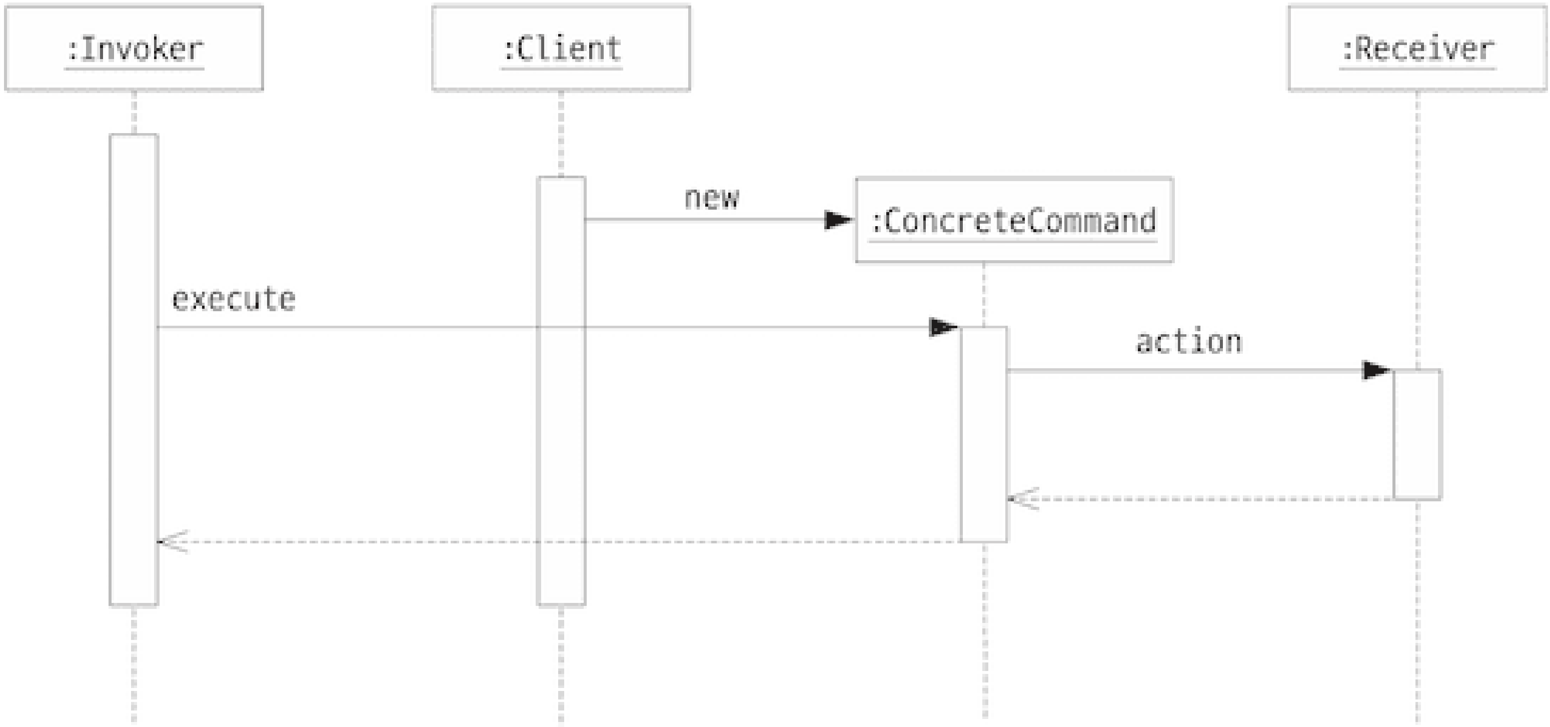
```python
from command import Command

class LightOnCommand(Command):
    def __init__(self, light):
        self.light = light

    def execute(self):
        self.light.turn_on()

class LightOffCommand(Command):
    def __init__(self, light):
        self.light = light

    def execute(self):
        self.light.turn_off()
```

MacroCommand (Invoker)

```python
"""Macro command - composite pattern"""
from command import Command

class MacroCommand(Command):
    def __init__(self):
        self.commands = []

    def add(self, command):
        self.commands.append(command)

    def execute(self):
        for cmd in self.commands:
            cmd.execute()

    def undo_last(self):
        if self.commands:
            return self.commands.pop()
```

Discussion

Sequence of Operations



1. **Client** creates **ConcreteCommand** with **Receiver**
2. **Client** gives command to **Invoker**
3. **Invoker** stores command for later execution
4. **Invoker** calls **execute()** on command
5. **ConcreteCommand** calls method on **Receiver**

Key Benefits

1. **Decoupling**: Invoker and receiver are completely decoupled
2. **Flexibility**: Commands can be parameterized, queued, and logged
3. **Undo/Redo**: Easy to implement with command objects
4. **Macro operations**: Combine simple commands into complex operations
5. **Extensibility**: New commands added without changing existing code

Command Pattern Applications

GUI Applications: Menu items, toolbar buttons, keyboard shortcuts

Database Systems: Each SQL statement is a command with rollback

Network Protocols: Requests as command objects for reliable transmission

Game Development: Player actions as commands for replay/undo

Task Scheduling: Queue commands for background execution

When to Use Command

- **Parameterize objects** by action to perform (different buttons, same interface)
- **Queue, schedule, and execute** requests at different times
- **Support undo operations** (store command state)
- **Support logging** changes for crash recovery or auditing
- **Structure system** around high-level operations built on primitive operations

Implementation Variations

1. **Undoable commands**: Store state for reversal
2. **Smart commands**: Commands that know when they can be optimized
3. **Queued commands**: Commands with priorities, delays, and retry logic
4. **Persistent commands**: Commands that survive application restart

Related Patterns

- **Composite**: MacroCommand uses Composite to build complex commands
- **Memento**: Store command state for undo operations
- **Prototype**: Commands can be copied using the Prototype pattern
- **Strategy**: Command encapsulates algorithms, Strategy encapsulates families of algorithms

Command vs Strategy

Command Pattern:

- **Focus**: Encapsulating **requests** as objects
- **Intent**: Parameterize clients, support queuing/undo
- **Structure**: Command → Receiver relationship

Strategy Pattern:

- **Focus**: Encapsulating **algorithms** as objects
- **Intent**: Make algorithms interchangeable
- **Structure**: Context → Strategy relationship

Advantages and Disadvantages

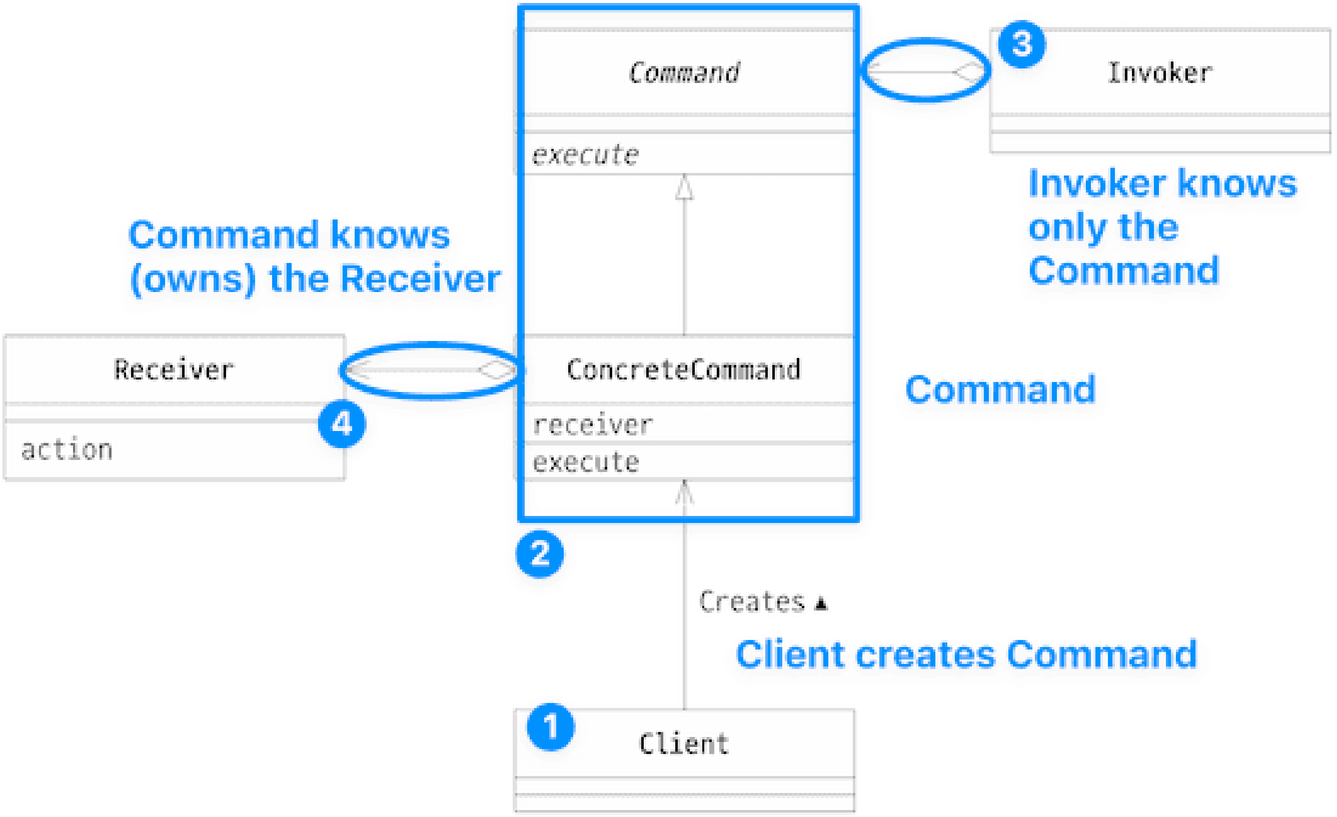
Advantages:

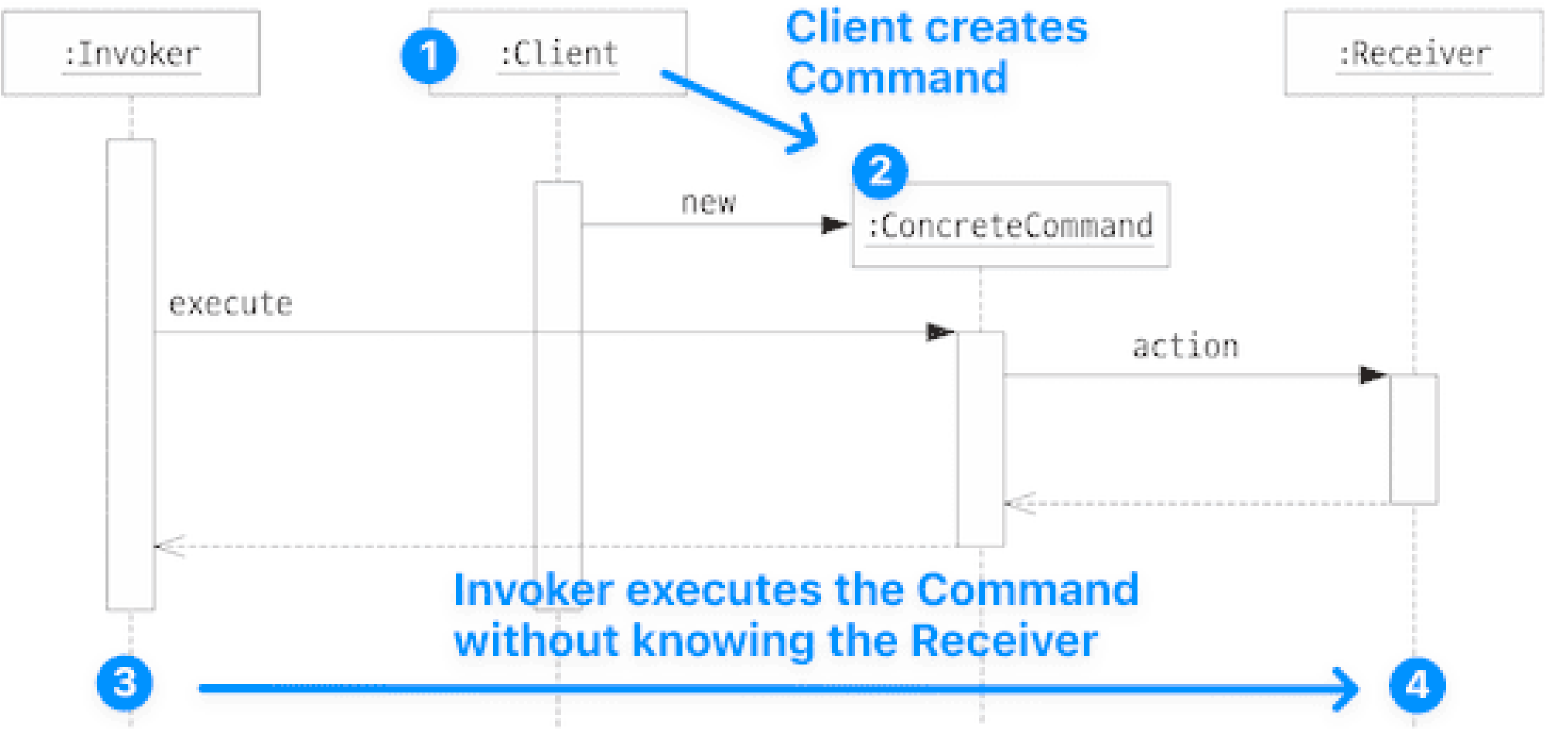
- **Decoupling**: Invoker independent of receiver
- **Flexibility**: Commands can be combined, scheduled, and logged
- **Extensibility**: Easy to add new commands

Disadvantages:

- **Complexity**: May introduce many small command classes
- **Memory overhead**: Storing commands uses memory
- **Performance**: Extra indirection through command objects

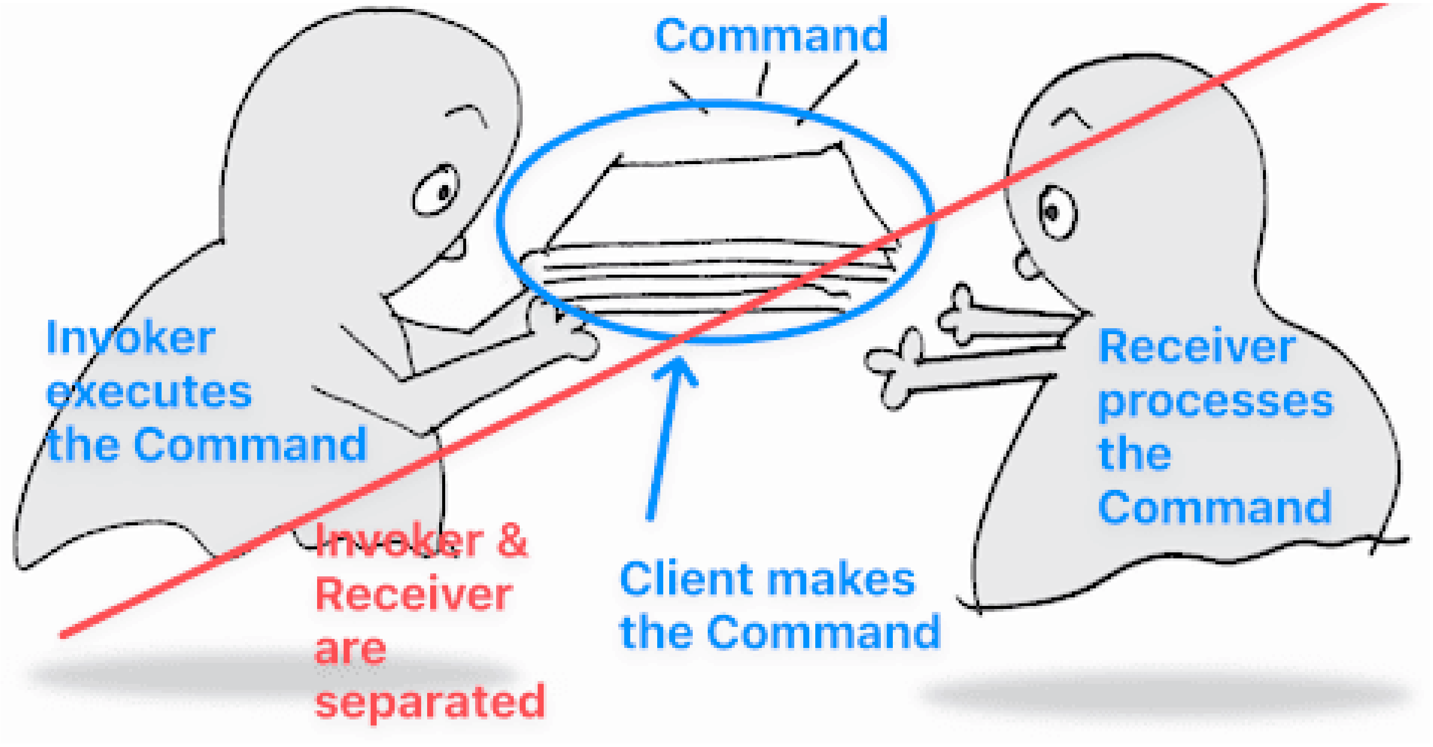
UML


Command
execute
Invoker
3
Invoker knows only the Command
Command knows (owns) the Receiver
Receiver
action
4
ConcreteCommand
receiver
execute
Command
2
Creates ▲
Client creates Command
1
Client



Client creates Command

Invoker executes the Command without knowing the Receiver


Command
Invoker executes the Command
Receiver processes the Command
Invoker & Receiver are separated
Client makes the Command