Command Pattern

CS356 Object-Oriented Design and Programming
http://cs356.yusun.io
November 13, 2014

Yu Sun, Ph.D.
http://yusun.io
yusun@csupomona.edu

CAL POLY POMONA

Command

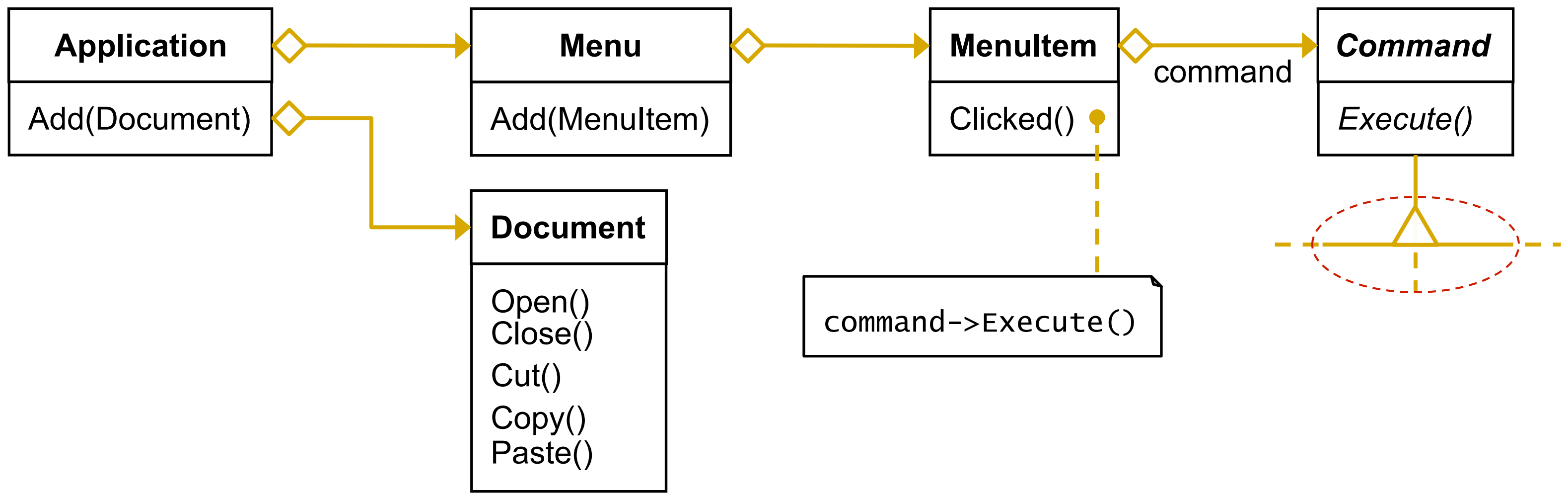
- Encapsulate requests for service from an object inside other objects
  - You can then manipulate the requests in various ways

Request
On Command
Off Command

Motivation

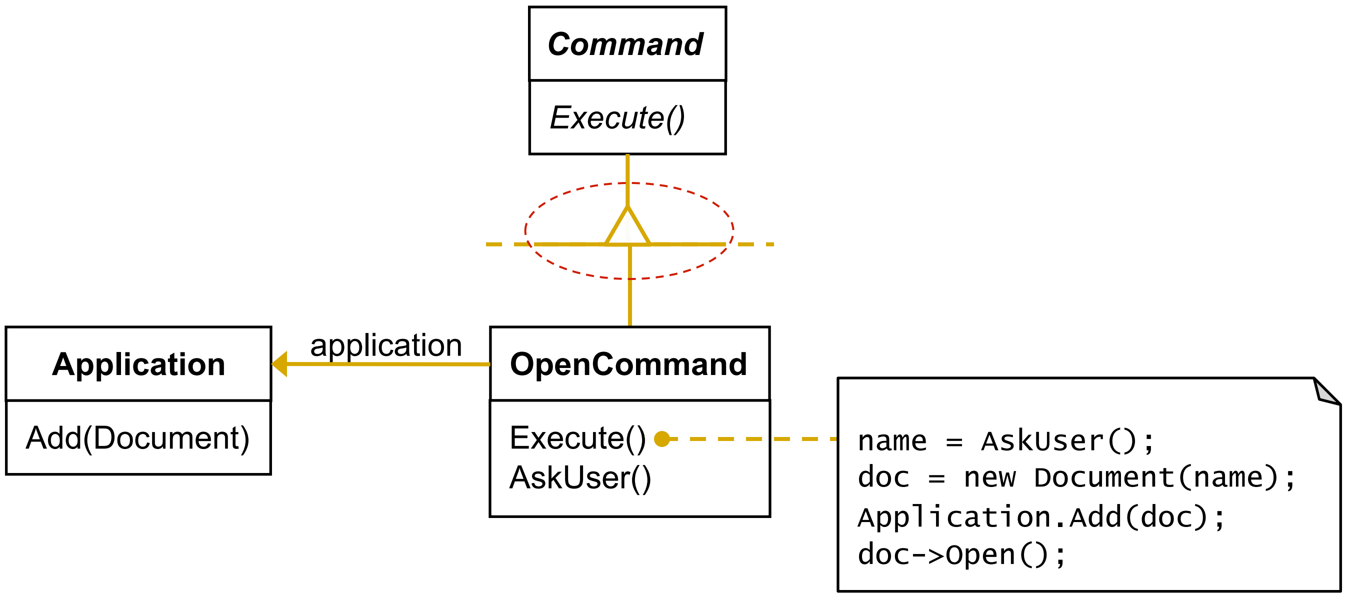
- In a user interface toolkit one can specify buttons and menu’s that carry out actions in response to user input
  - However, the toolkit is independent of the implementation
- The Command pattern lets toolkit objects make requests of unspecified application objects by turning the request into an object
  - Command objects can be stored and passed around like other objects
  - The simplest form of Command objects can execute one method: “Execute”

Motivation



- Every MenuItem contains a Command object
- When the MenuItem is clicked, the Command is executed (MenuItem requires no knowledge of action)
- The Command stores the receiver of the request and executes one or more operations on the receiver

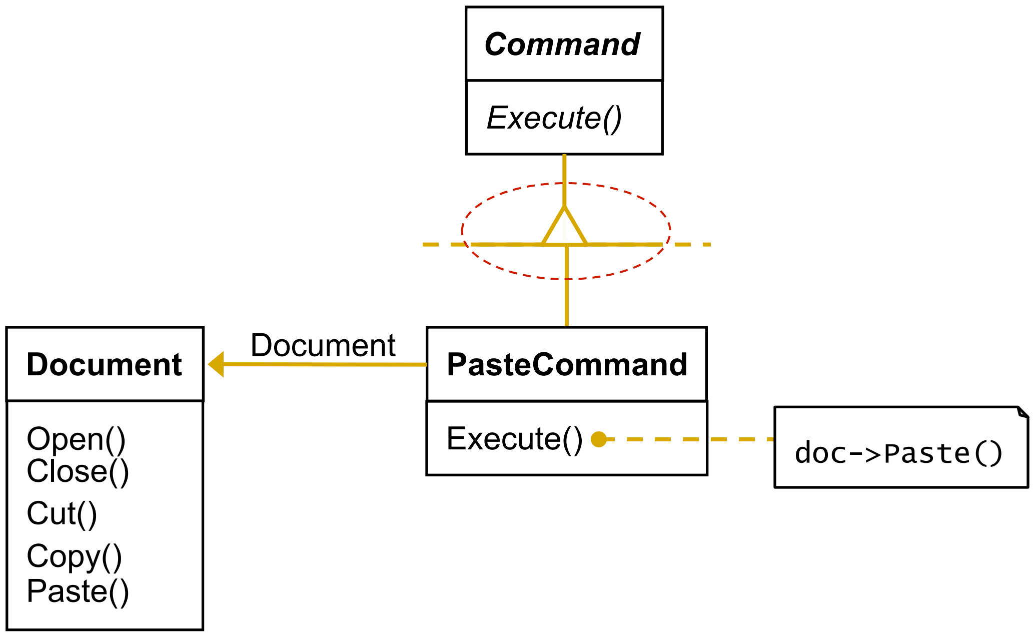
OpenCommand



```
name = AskUser();
doc = new Document(name);
Application.Add(doc);
doc->Open();
```

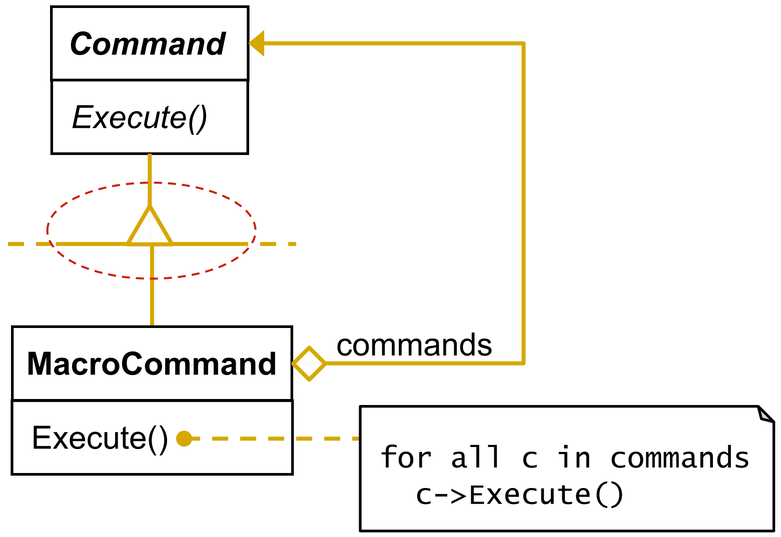
- OpenCommand asks the user for a document name, creates the corresponding Document object, adds the document to the receiving application, and opens the document

PasteCommand



- PasteCommand copies text from a clipboard into an open document
- Execute() invokes paste() on the receiving document, supplied when PasteCommand was instantiated

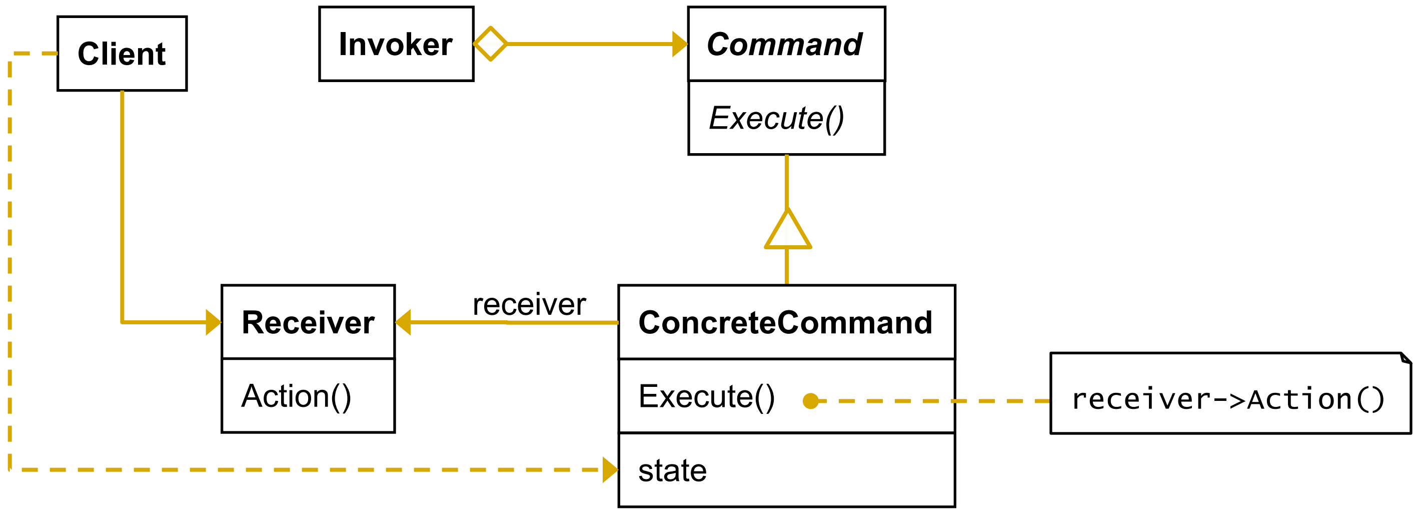
Composite Commands



```
for all c in commands
  c->Execute()
```

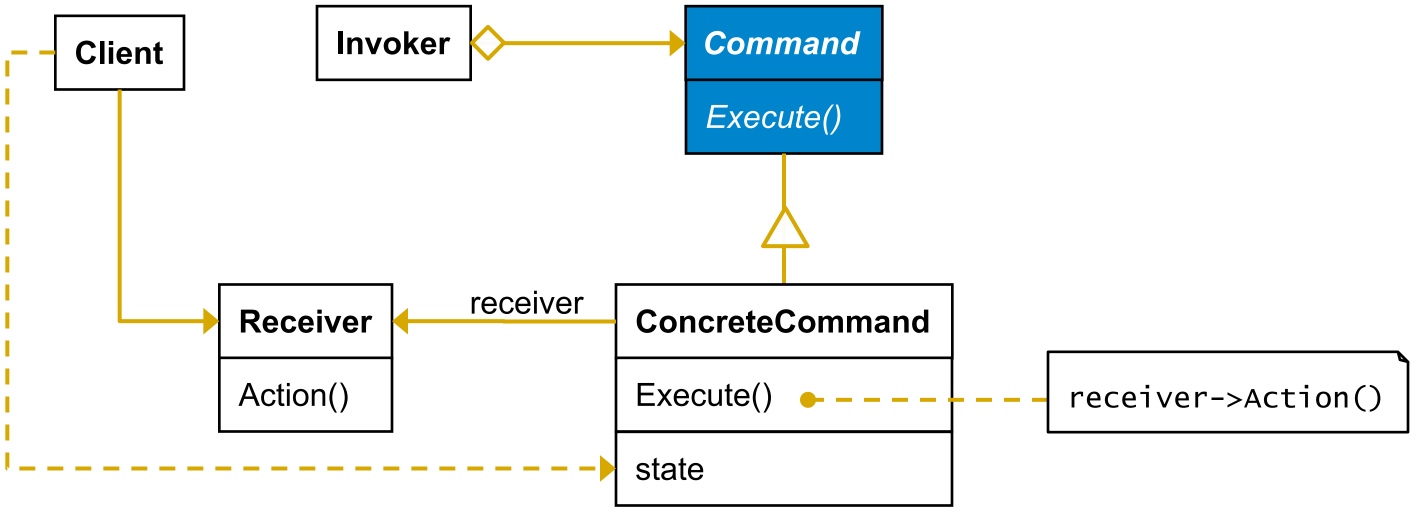
- MacroCommand executes a sequence of commands

Structure


Client
Invoker
Command
Execute()
Receiver
Action()
receiver
ConcreteCommand
Execute()
state
receiver->Action()

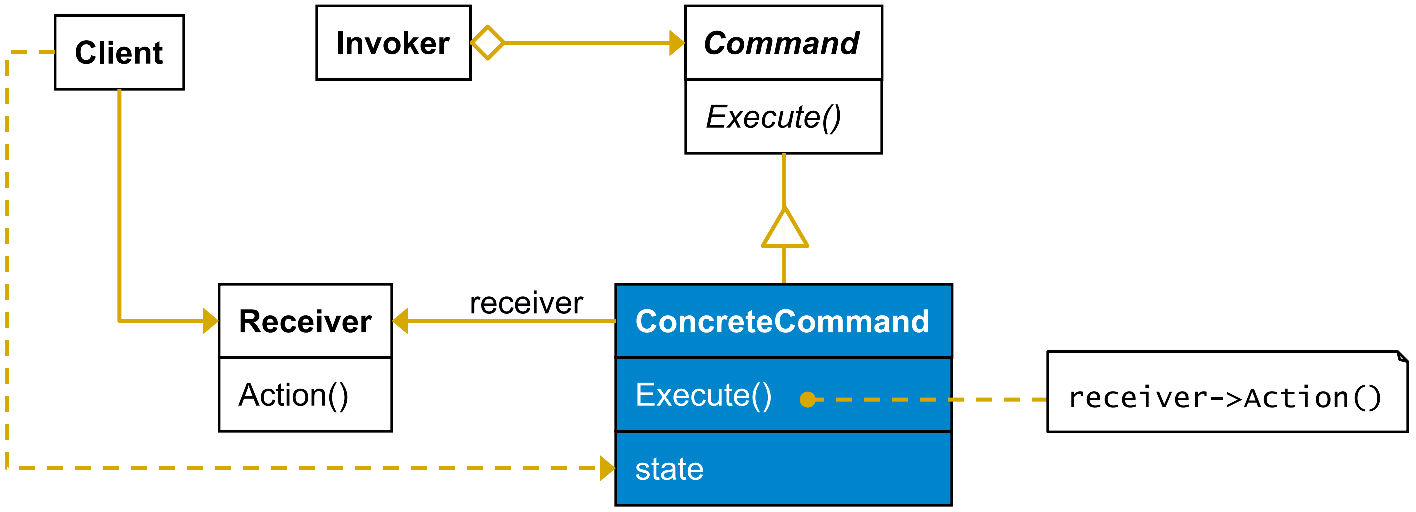
Command

- Declares the interface for executing the operation



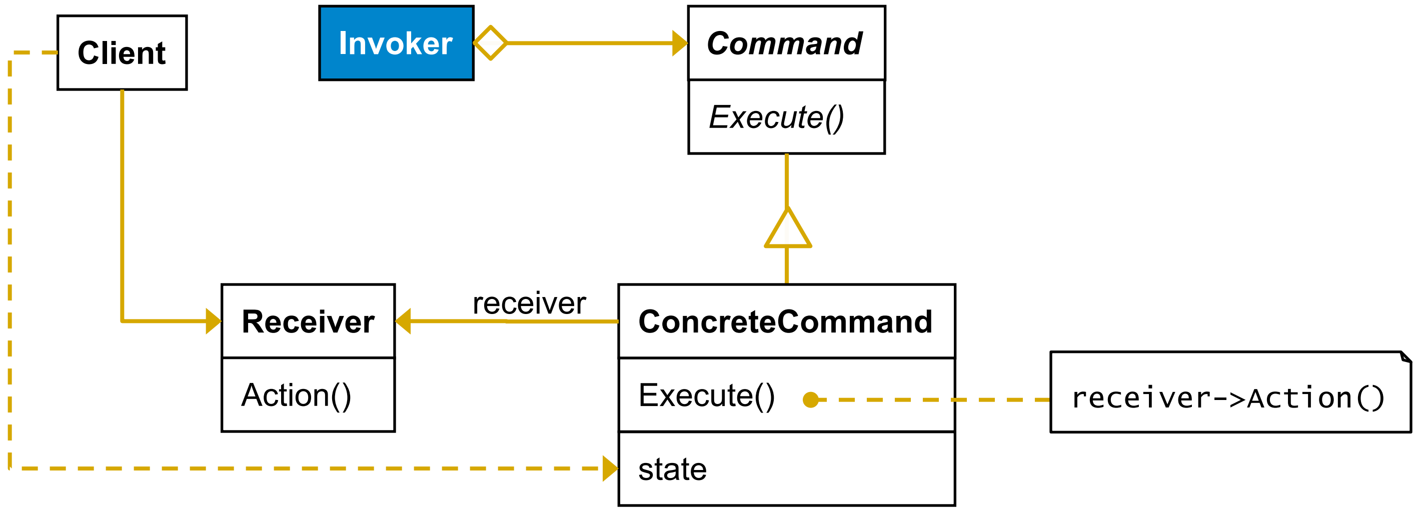
ConcreteCommand

- Binds a request with a concrete action



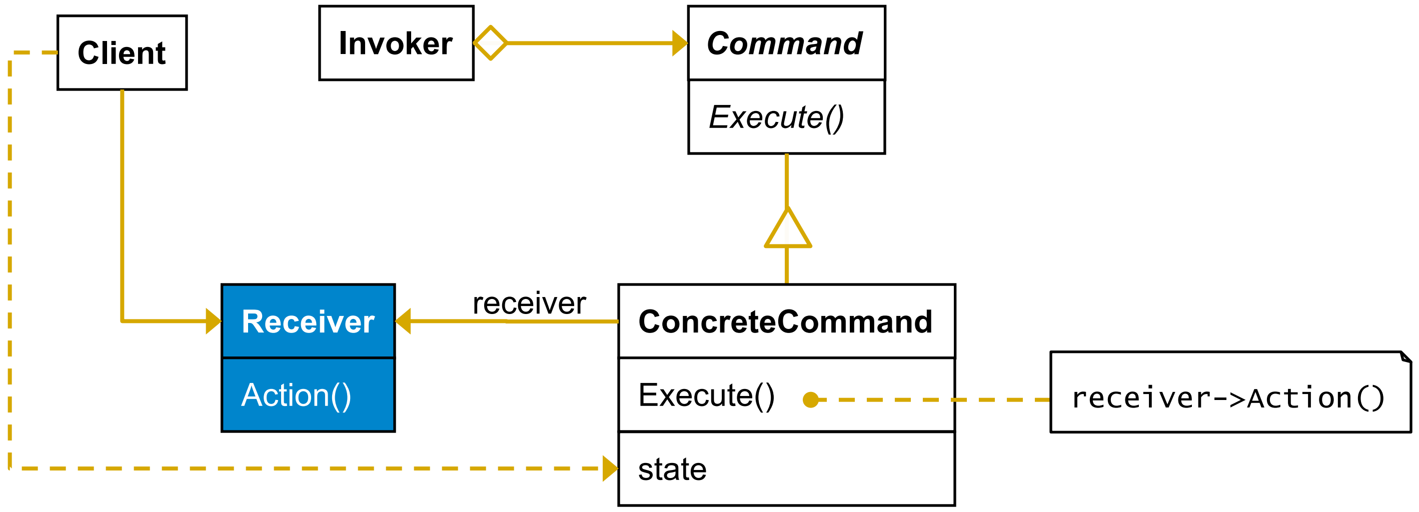
Invoker

- Asks the command to carry out the request



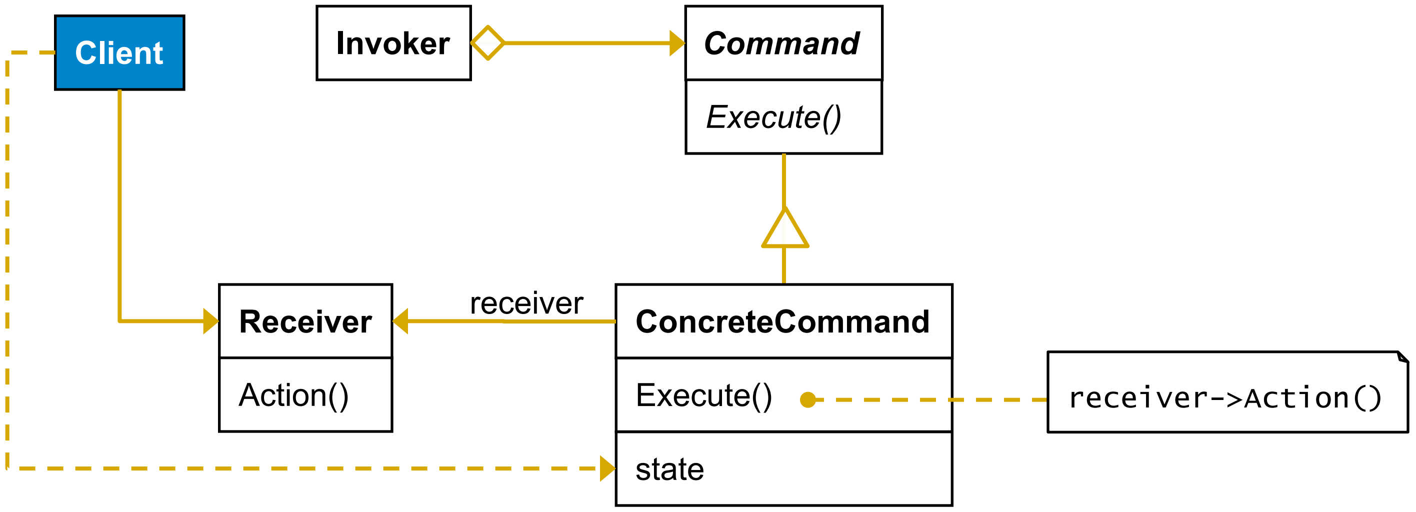
Receiver

- Knows how to perform the operations associated with carrying out a request

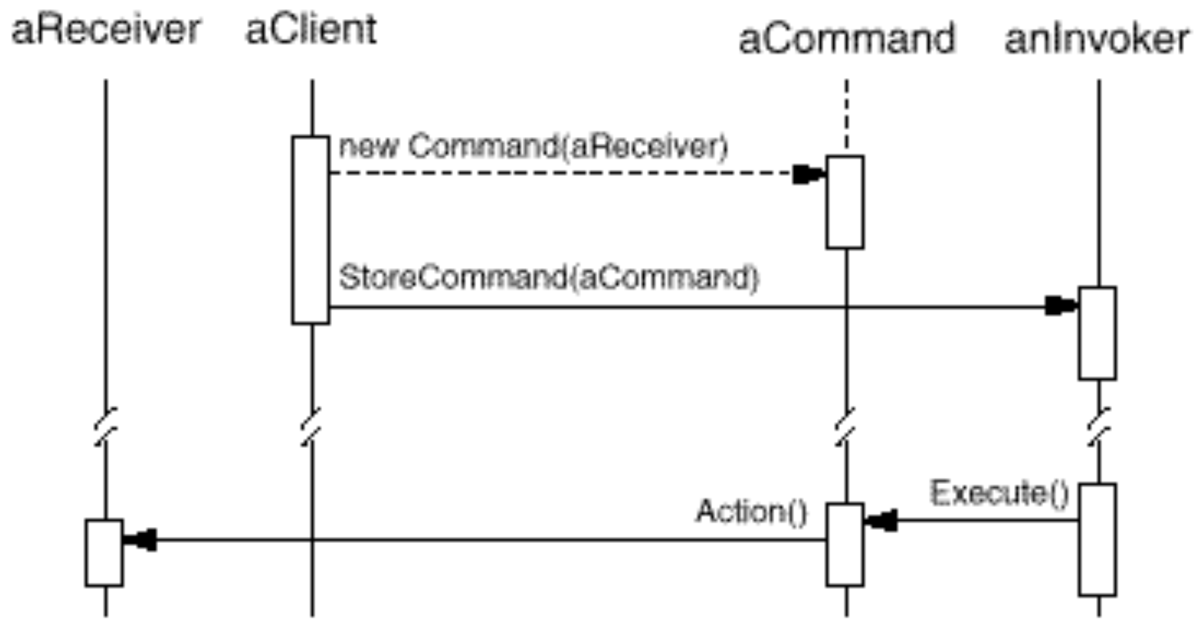


Client

- Creates a ConcreteCommand and sets its receiver



Collaborations



- Client → ConcreteCommand
  - Creates and specifies receiver
- Invoker stores the ConcreteCommand
- ConcreteCommand invokes Receiver

Intelligence of Command Objects

- "Dumb"
  - Delegate everything to Receiver
  - Used just to decouple Sender from Receiver
- "Smart"
  - Find receiver dynamically
- "Genius"
  - Does everything itself without delegating at all
  - Useful if no receiver exists
  - Let ConcreteCommand be independent of further classes

Applicability

- Parameterize objects
  - Replacement for callbacks and function pointers
- Specify, queue, and execute requests at different times
- Support undo, redo
- Support for logging changes
- Separate the user interface from the actions it performs
  - Allowing GUI and program execution to vary independently common

Example – Inserting into a TextArea

- Traditional Usage

```
TextArea textArea = new TextArea("Hello”);
textArea.insert("World", 6);
System.out.println(textArea.getText());
```

As a Command Pattern

```
public class InsertText {

  private TextArea textArea;
  private String text;
  private int offset;

  public InsertText(TextArea target, String str, int pos) {
    textArea = target;
    text = str;
    offset = pos;
  }

  public void execute() {
    textArea.insert(text, offset);
  }
}
```

```
TextArea textArea = new TextArea("Hello”);
InsertText insertCommand = new InsertText(textArea, "World", 6);
insertCommand.execute();
System.out.println(textArea.getText());
```

New Invocation process

- Adding an Undo is easy
  - Implement `unexecute()` method

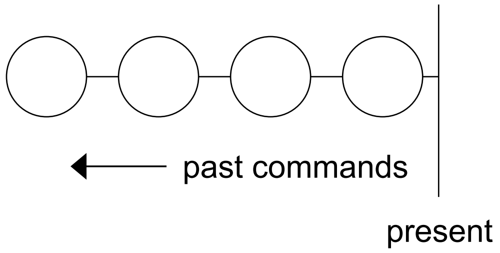
```
public void unexecute() {
  textArea.replaceRange("", offset, offset + text.length());
}
```

Undoable Commands

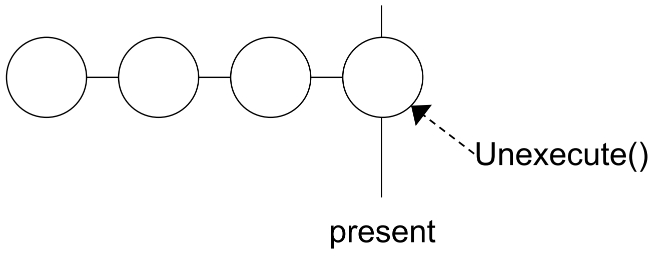
- Need to store additional state to reverse execution
  - Receiver object
  - Parameters of the operation performed on receiver
  - Original values in receiver that may change due to request
    - Receiver must provide operations that makes it possible for command object to return it to its prior state
      - E.g., delete file operation must know name of file to undelete
- History list
  - Sequence of commands that have been executed
    - Used as LIFO with reverse-execution → undo
    - Used as LIFO with execution → redo

History List

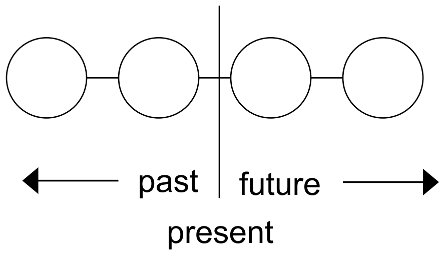
- Each circle represents a command object



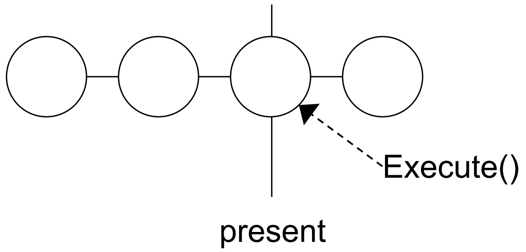
- To *undo*, simply call `Unexecute()` on the most recent command



- After one more *undo*



- To *redo*, execute the command to the right of the present context



Enable Undo

- Introduce a stack (`executedCommands`)

```
public void execute(Command command) {
  command.execute();
  executedCommands.push(command);
}

public void unexecute() {
  Command command = (Command)executedCommands.pop();
  command.unexecute();
}
```

Enable Redoing an Undo

- Separate stacks needed (`executedCommands` and `unexecutedCommands`)

```
public void unexecute() {
   Command command = (Command)executedCommands.pop();
   command.unexecute();
   unexecutedCommands.push(command);
}

public void reexecute() {
  Command command = (Command)unexecutedCommands.pop();
  execute(command);
}
```

Consequences

- Decouples Invoker from Receiver
- Commands are first-class objects
  - Can be manipulated and extended
- Assemble commands into a composite command
- Easy to add new commands
  - Invoker does not change
  - It is Open-Closed