Command

A behavioral pattern

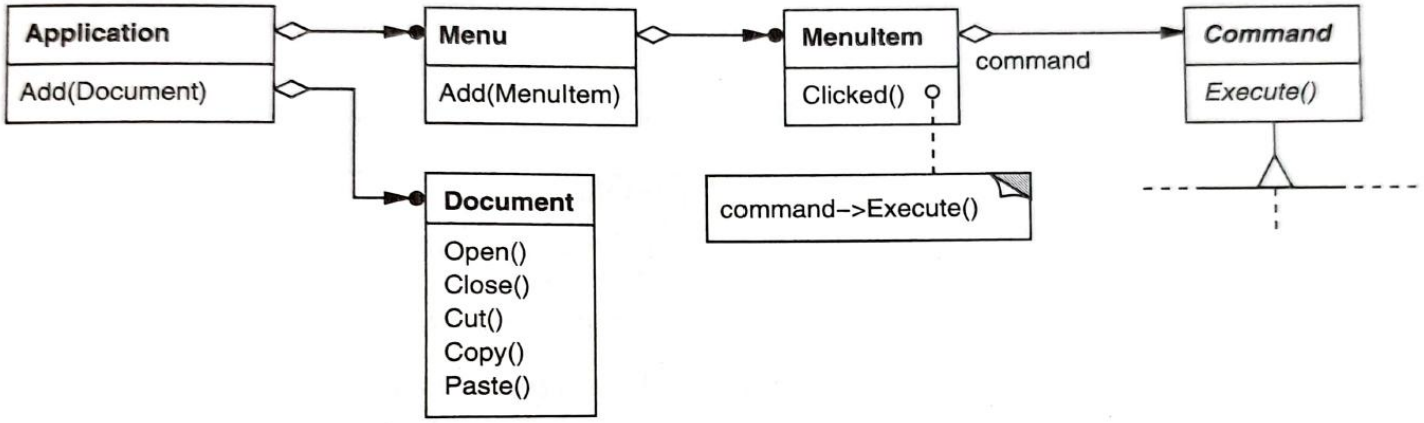
Learning goals

1. Learn the idea, structure, and Java implementation of the Command design pattern.
2. Learn to apply the Command DP in your own programming.

Idea of Command

- The Command Design Pattern **encapsulates a request as an object**.
  - This allows for the parameterization of clients with different requests, queuing of requests, and execution at varied times.
- A common **execute()** method acts as a simple front door to any concrete command implementation.
- The **receiver** of the command performs the actual action, while the **invoker** triggers the command to execute.
- Commands can be modified and extended without altering the core application structure.
  - This adheres to the open/closed principle.

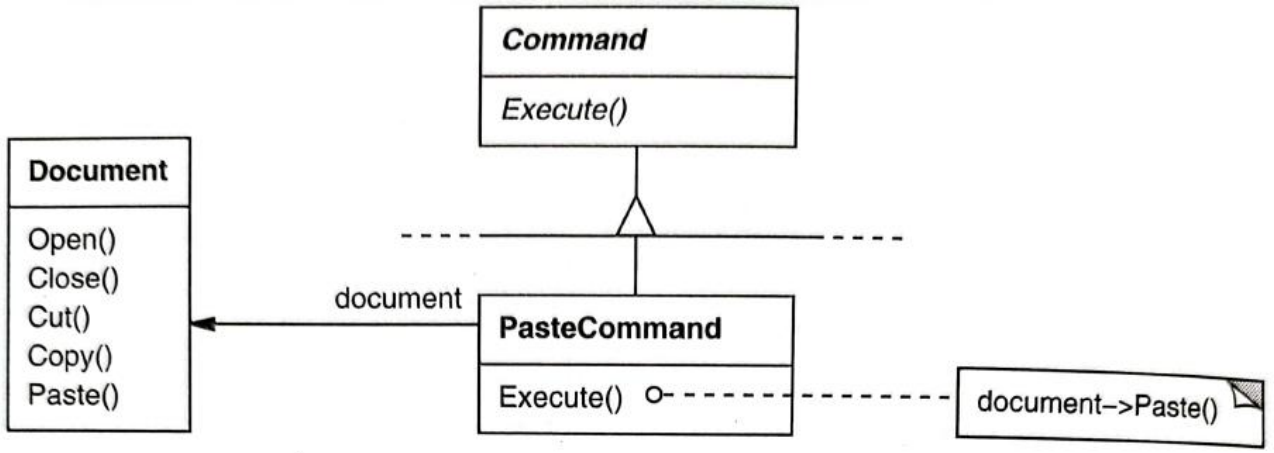
Example: UI toolkit



- The UI toolkit cannot anticipate the possible commands that the UI components may need to trigger.
- Solution: Create a uniform Command interface that is common to all commands.
- Each component (e.g. MenuItem) is provided with the information of which command it shall trigger.

Image: Gamma et al., Design Patterns. Elements of Reusable Object-Oriented Software. Addison Wesley Longman (1995), p. 233

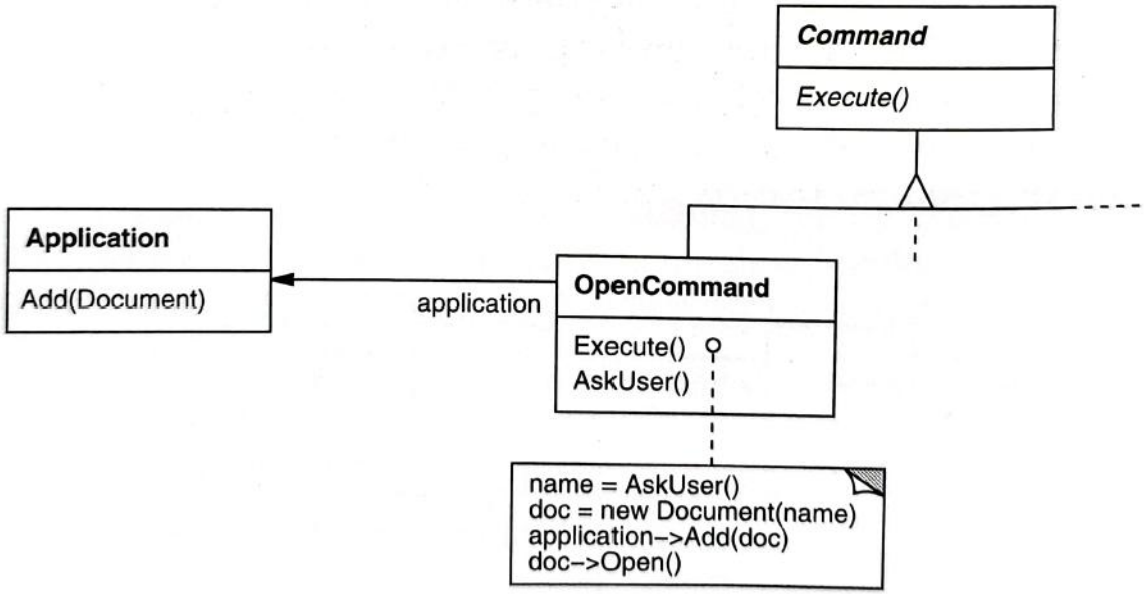
Example: UI toolkit



- Each concrete command knows the receiver of the request as well as its operation that it will invoke.
  - In the example, the PasteCommand knows that it must call the Paste() method of a Document object.

Image: Gamma et al., Design Patterns. Elements of Reusable Object-Oriented Software. Addison Wesley Longman (1995), p. 234

Example: UI toolkit



- On the other hand, the execute() method of OpenCommand is coded so that it will:
  1. Ask for the document's name.
  2. Invoke the Application's add() method, providing the newly created Document object.

Image: Gamma et al., Design Patterns. Elements of Reusable Object-Oriented Software. Addison Wesley Longman (1995), p. 234

General structure

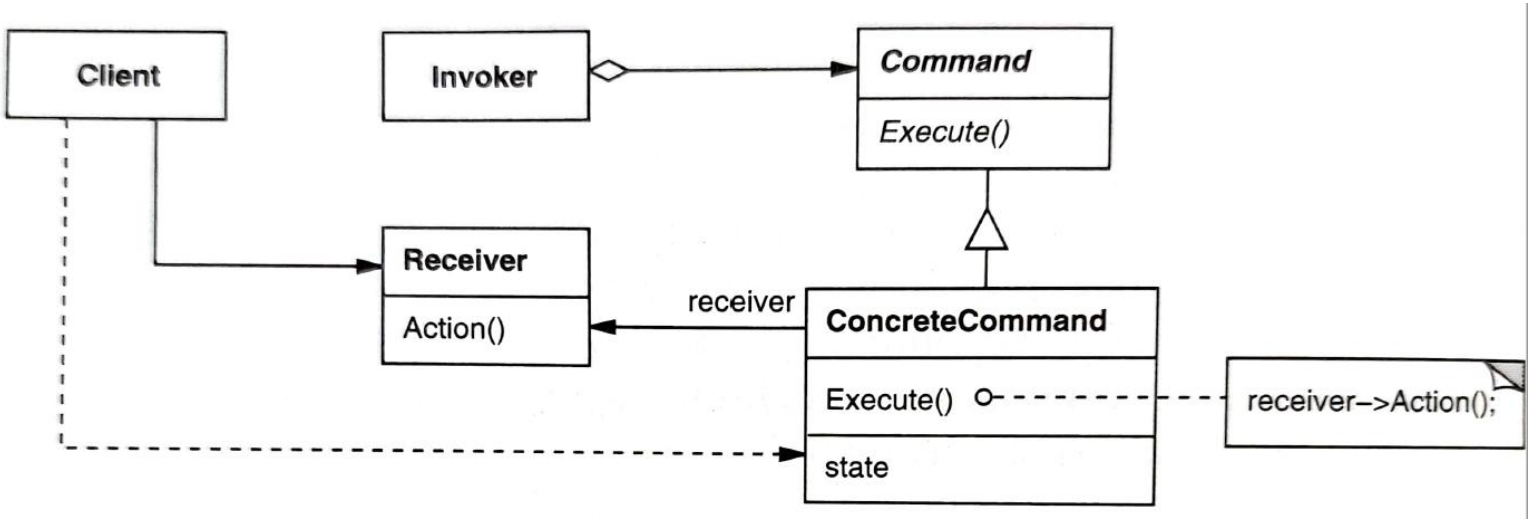


Image: Gamma et al., Design Patterns. Elements of Reusable Object-Oriented Software. Addison Wesley Longman (1995), p. 236

Roles

- **Command**: Declares an interface for executing an operation.
  - The command interface typically includes a single execution method.
- **Concrete Command**: Implements the Command interface, defining the binding between a Receiver object and an action.
  - Contains the execution logic for the command.
  - Each Concrete Command has a reference to a receiver.
- **Receiver**: The object that performs the actual work when the command's execute() method is called.
- **Invoker**: Holds a command and at some point asks the command to carry out a request by calling its execute() method.
  - An Invoker might queue commands, schedule their execution, or execute them immediately, depending on the implementation.
- **Client**: Creates a Concrete Command object and sets its receiver.
  - The client might also specify the parameters of the command and associate the command with the invoker.

Command pattern in JavaFX GUI updates

```
Platform.runLater(() -> gui.updateButtonText());
```

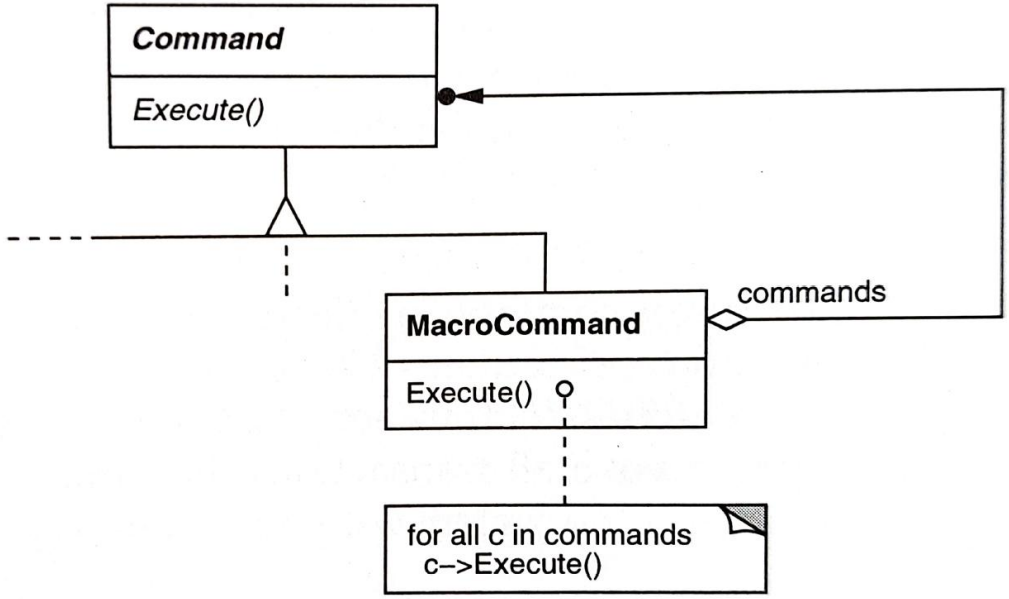
- In JavaFX, GUI update requests from different threads are sent with the **Platform.runLater()** method call.
  - This ensures that an update request from a different thread will not interfere with the rendering in the GUI thread (aka. JavaFX Application thread).
- Its implementation follows the idea of the Command DP:
  - **Command**: a **Runnable** object provided above as a lambda expression.
  - **Receiver**: the GUI thread, which takes responsibility of updating the user interface.
  - **Invoker**: The **Platform** class which manages the queue for requested updates.
  - **Client**: the code that calls **Platform.runLater(),** and runs in a non-GUI thread.

Command pattern in JavaFX GUI updates

```
Platform.runLater(() -> gui.updateButtonText());
```

- **Platform.runLater()** accepts an implementation of the Runnable interface as a parameter. This is usually provided as a lambda expression or an anonymous inner class in Java.
- The lambda expression **() -> gui.updateButtonText()** serves as the implementation of the **run()** method defined in the **Runnable** interface.
- When **Platform.runLater(Runnable)** is called, the **Runnable** object is placed into a queue for execution.
- The GUI thread, at an appropriate time, dequeues and executes the **run()** method of the **Runnable** object.
  - This execution occurs on the GUI Thread, allowing for safe updates to the GUI components.
  - The method **run()** is invoked directly by the GUI thread, distinguishing it from the **start()** method which is used with **Thread** objects to initiate a new thread.

Macro commands



- The DP allows for easy creation of **macro commands**, i.e. Command objects that contain a list of Commands.

Image: Gamma et al., Design Patterns. Elements of Reusable Object-Oriented Software. Addison Wesley Longman (1995), p. 235

Practical issues

- Commands can be queued for later execution, as is done in the **Platform.runLater()** example.
  - Java provides **java.util.Queue** that is well-suited for these purposes.
- Commands can be dynamically created at runtime, which can lead to an increase in the complexity of the client code.
  - Using a factory or builder pattern can help manage this complexity.
- Hard-coding the **Receiver** inside the **Command** can simplify the design for static contexts but reduces flexibility.