Software Design Patterns

Lecture 9

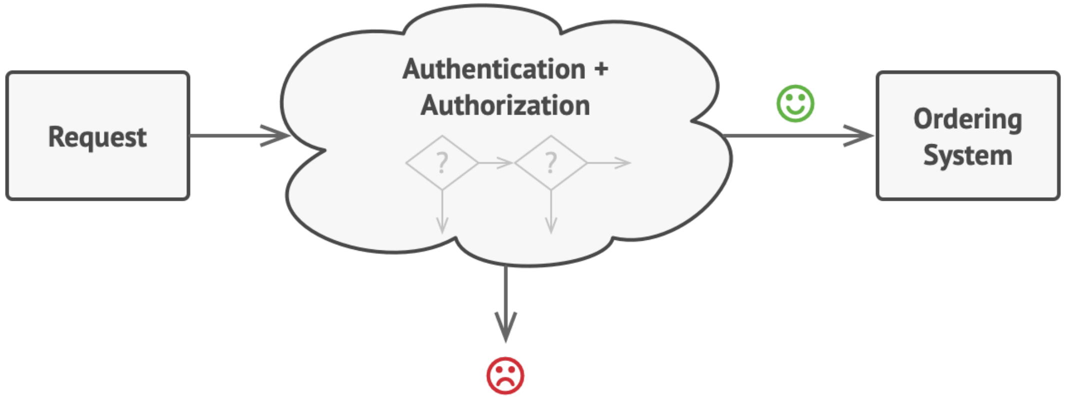
Chain of Responsibility
Command

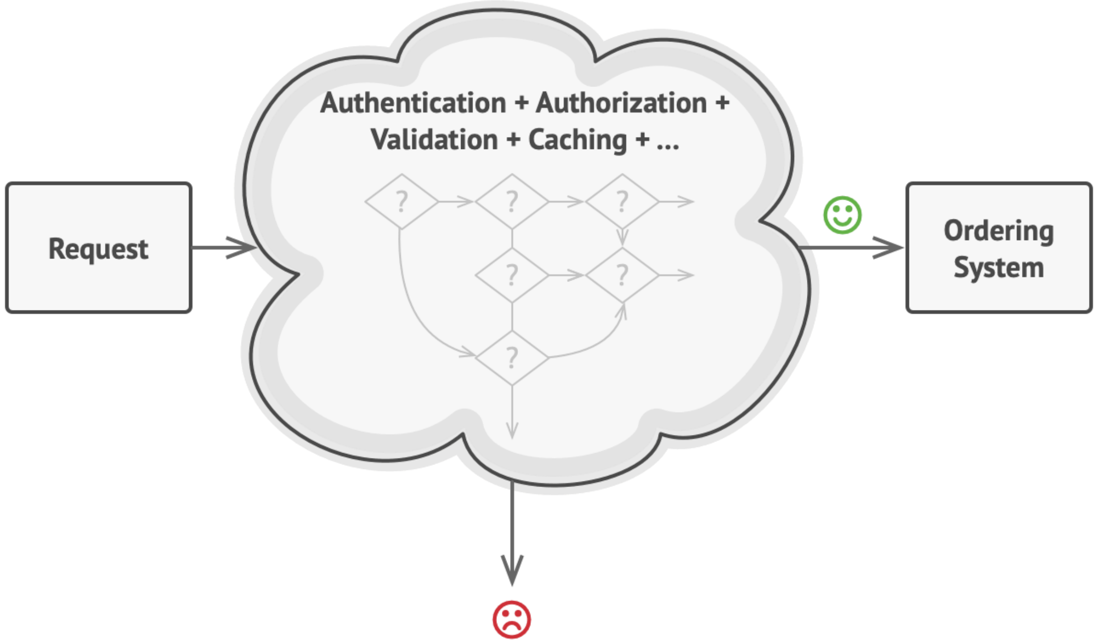
Dr. Fan Hongfei

31 October 2024

Chain of Responsibility: Problem

- **Example: an online ordering system**
  - The request must pass **a series of checks**
  - **New requirements**: validation, filtering repeated failed requests, speeding up by returning cached results, and more

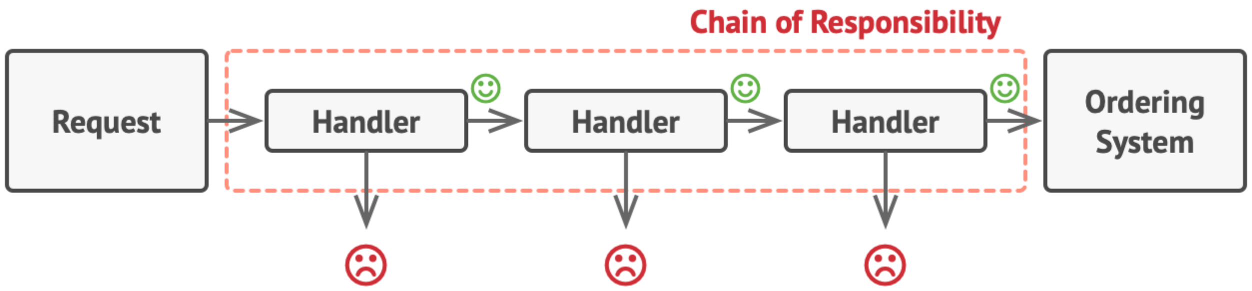




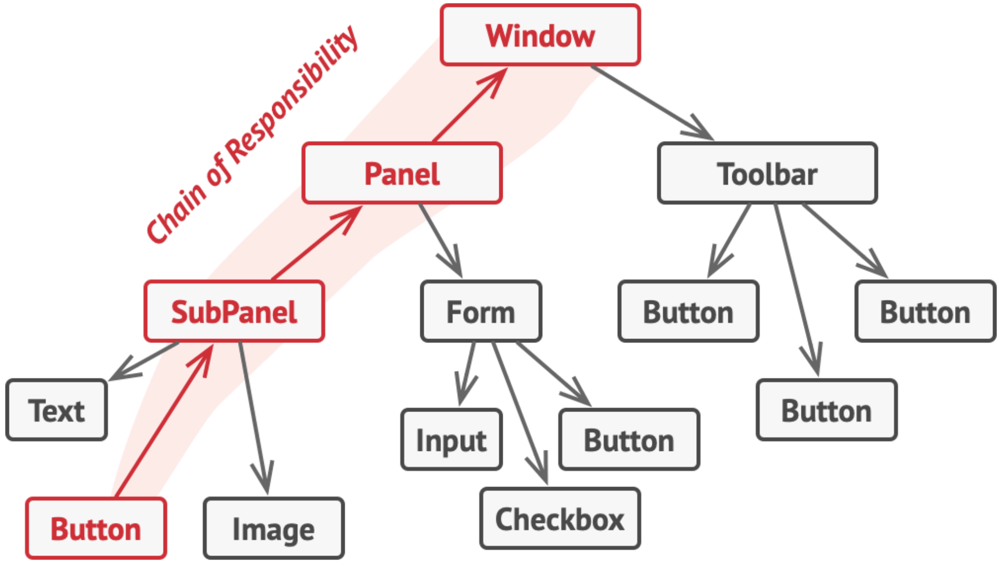
[Figures in this slide are extracted from https://refactoring.guru/design-patterns/chain-of-responsibility]

Chain of Responsibility: Solution

- **Chain of Responsibility (CoR):** Transforming behaviors into standalone objects called **handlers**
  - Handlers linked into a chain
  - Each has a field with a reference to the next
  - Handlers process the request, and pass the request along the chain
  - A handler can decide not to pass it further



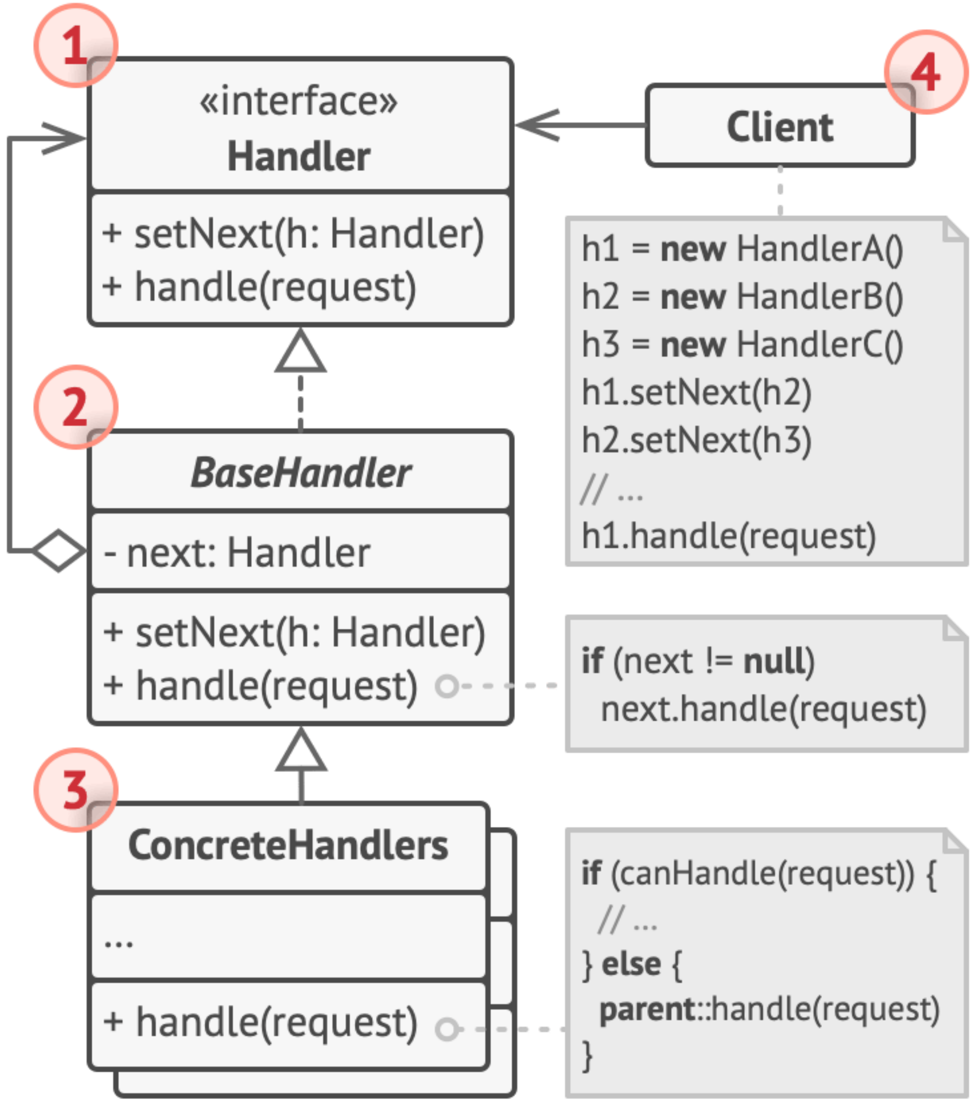
- **A slightly different approach**
  - Upon receiving a request, if a handler can process it, it does not pass it further



- All handler classes implement the same interface
- Each only cares about the following one
- Chains can be composed at runtime

[Figures in this slide are extracted from https://refactoring.guru/design-patterns/chain-of-responsibility]

Chain of Responsibility: Structure



```
h1 = new HandlerA()
h2 = new HandlerB()
h3 = new HandlerC()
h1.setNext(h2)
h2.setNext(h3)
// ...
h1.handle(request)
```

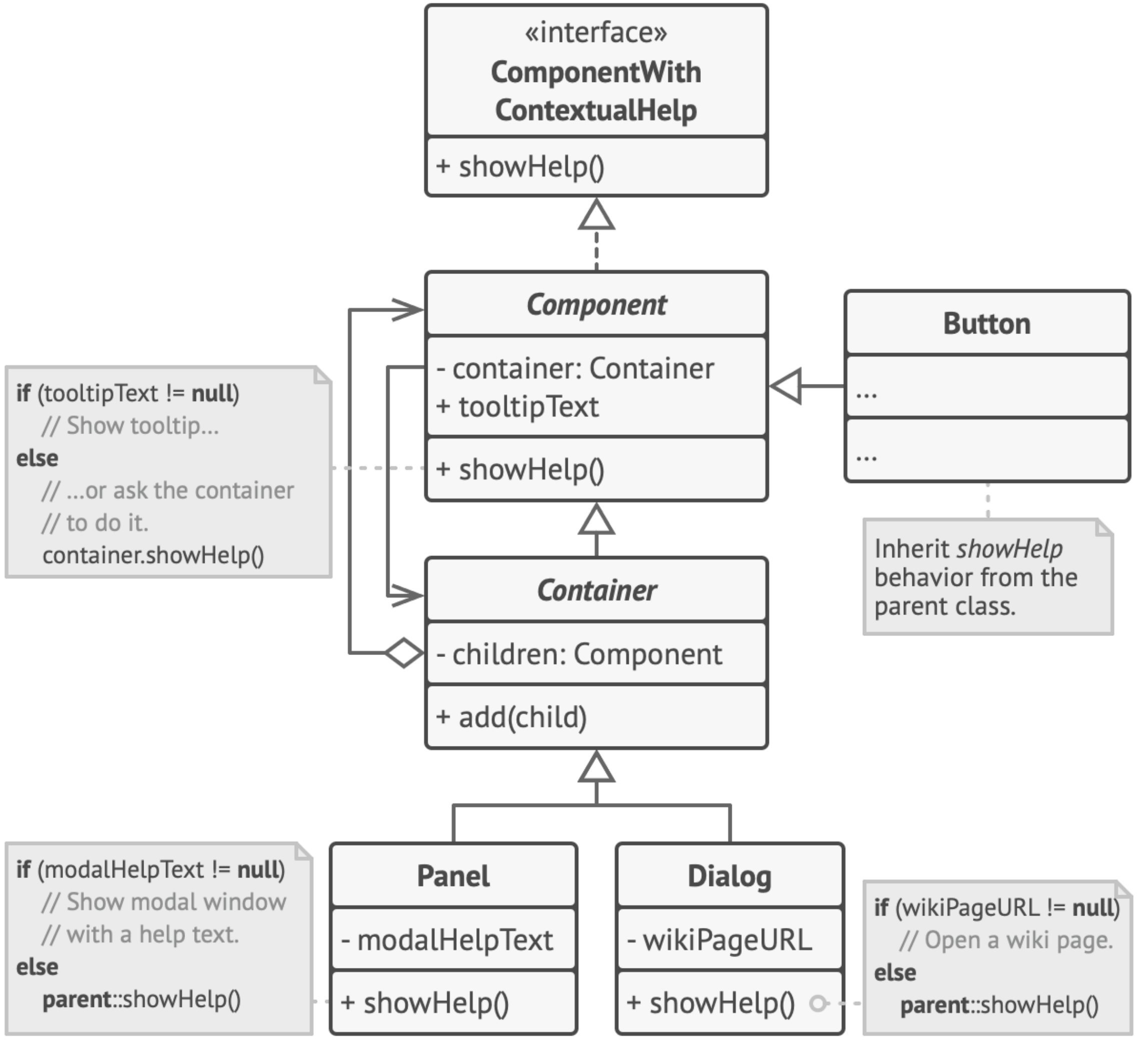
```
if (next != null)
  next.handle(request)
```

```
if (canHandle(request)) {
  // ...
} else {
  parent::handle(request)
}
```

1. **Handler:** the interface common for all concrete handlers
2. **Base Handler:** optional, for boilerplate code common to all handlers
3. **Concrete Handlers:** actual code for processing requests
   - Each handler decides whether to process it, and whether to pass it along the chain
   - Self-contained and immutable
4. **Client:** composing chains once or dynamically
   - A request can be sent to any handler (not necessarily the first)

[Figures in this slide are extracted from https://refactoring.guru/design-patterns/chain-of-responsibility]

Chain of Responsibility: Example

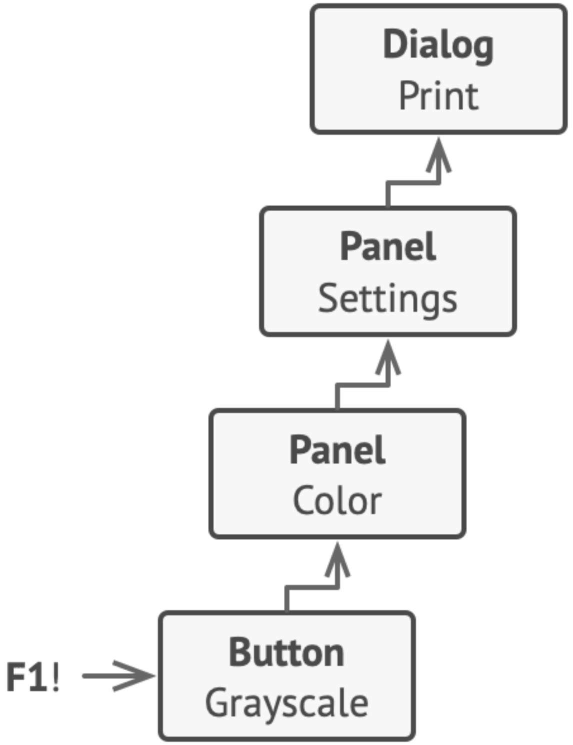


```
if (tooltipText != null)
    // Show tooltip...
else
    // ...or ask the container
    // to do it.
    container.showHelp()
```

Inherit *showHelp* behavior from the parent class.

```
if (modalHelpText != null)
    // Show modal window
    // with a help text.
else
    parent::showHelp()
```

```
if (wikiPageURL != null)
    // Open a wiki page.
else
    parent::showHelp()
```



[Figures in this slide are extracted from https://refactoring.guru/design-patterns/chain-of-responsibility]

Chain of Responsibility: Applicability

- Process different kinds of requests in various ways, but the exact types of requests and their sequences are unknown beforehand
  - CoR lets you link handlers into one chain and, upon receiving a request, “ask” each handler whether it can process it
- Execute several handlers in a particular order
  - All requests get through the chain exactly as planned
- When the set of handlers and their order are supposed to change at runtime
  - With the help of the setter for a reference field

Chain of Responsibility: Implementation

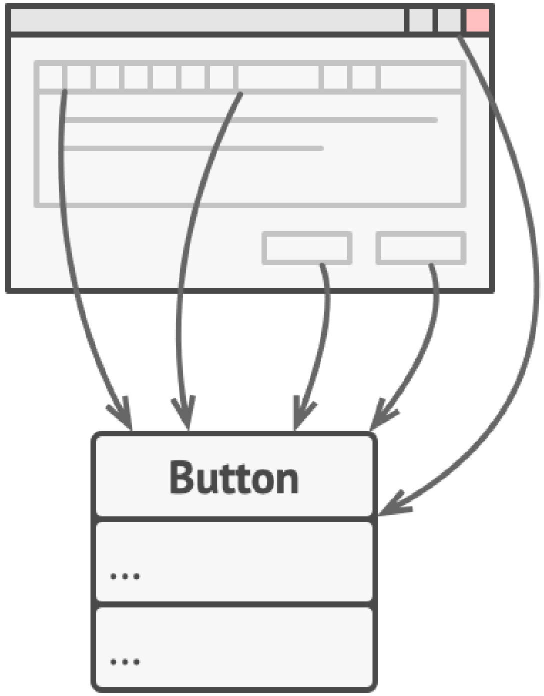
1. Declare the **handler interface**, and describe the signature of a method for handling requests
2. Optional: create an **abstract base handler class**, for boilerplate code
   - **A field for storing a reference** to the next handler, and consider making the class immutable
   - To modify chains at runtime, a **setter** is needed
   - Consider the convenient **default behavior**: forward the request to the next object unless there is none left
3. Create **concrete handler subclasses** and implement handling methods
4. The client may either assemble chains on its own, or receive pre-built chains from other objects
5. The client may trigger any handler in the chain

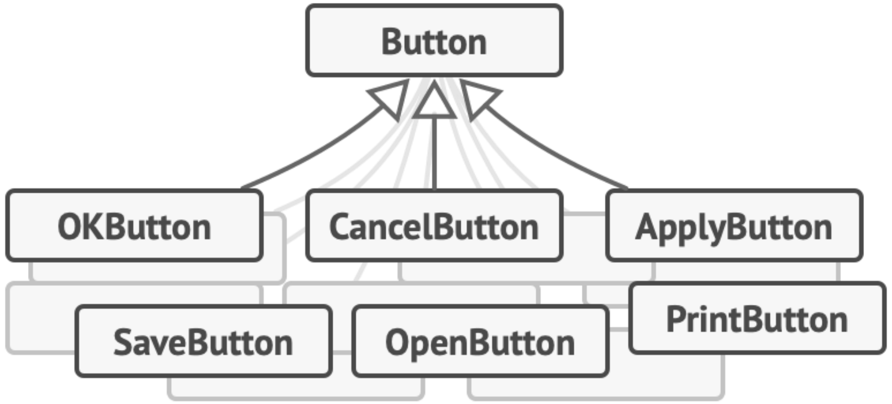
Chain of Responsibility: Pros and Cons

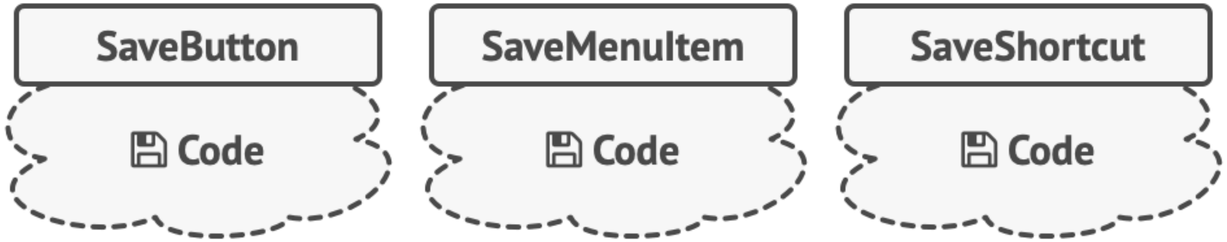
- **Pros**
  - Control the order of request handling
  - Single Responsibility Principle: decoupling classes that invoke operations from classes that perform operations
  - Open/Closed Principle: introducing new handlers into the app without breaking the existing client code
- **Cons**
  - Some requests may end up unhandled

Command: Problem

- **Example: a text editor app**
  - Task: create a toolbar with buttons for various operations





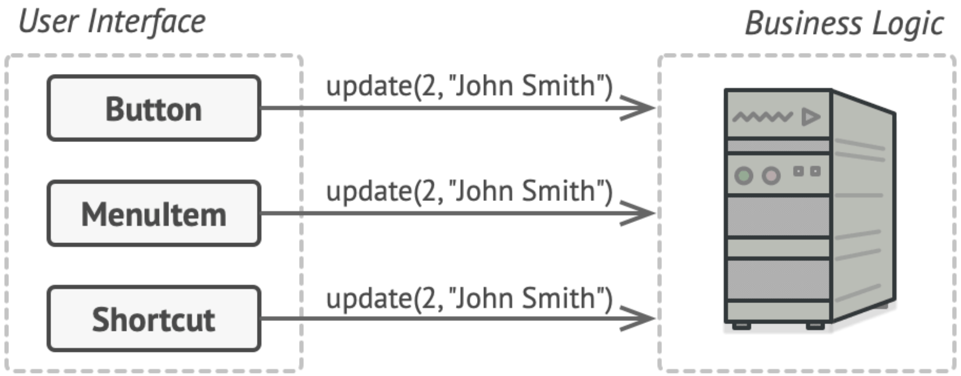


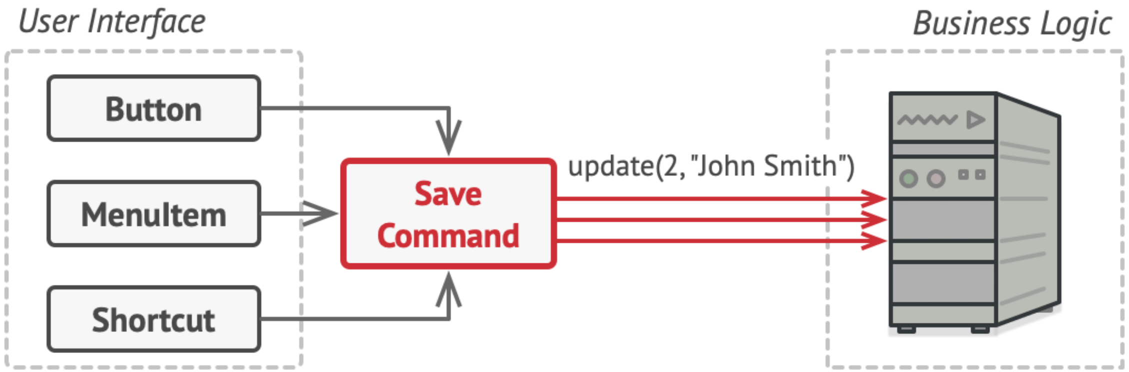
[Figures in this slide are extracted from https://refactoring.guru/design-patterns/command]

Command: Solution

- **Principle of separation of concerns**
  - Breaking an app into layers
  - Common example: a layer for GUI, and another for business logic

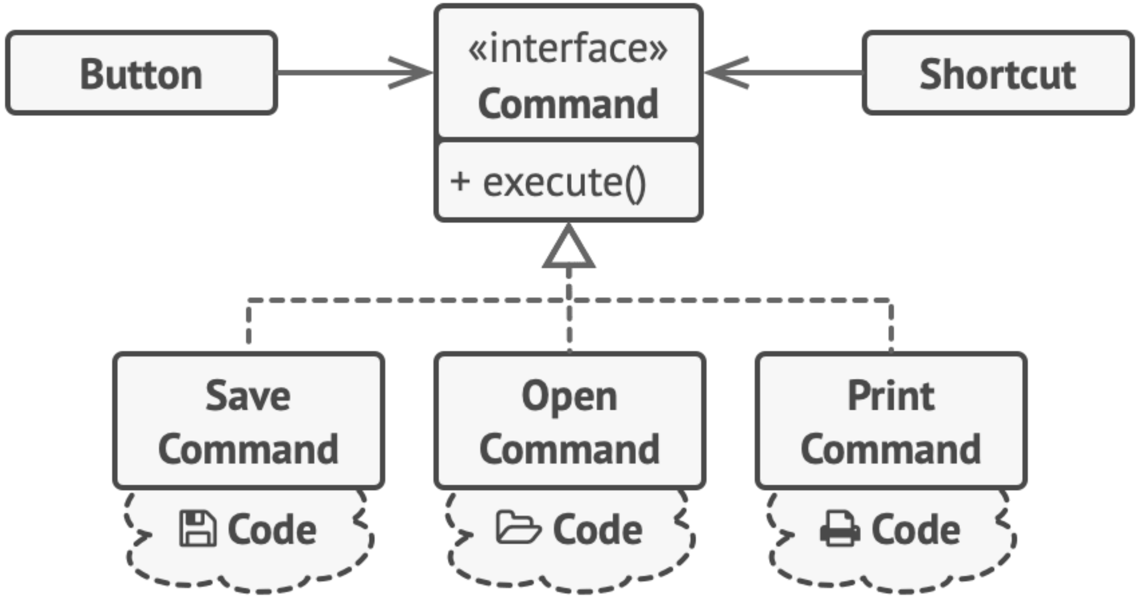
- **Command** (aka Action or Transaction)
  - GUI objects should not send request directly
  - Extract request details into a separate **Command** class with a single method that triggers this request





[Figures in this slide are extracted from https://refactoring.guru/design-patterns/command]

Command: Solution (cont.)

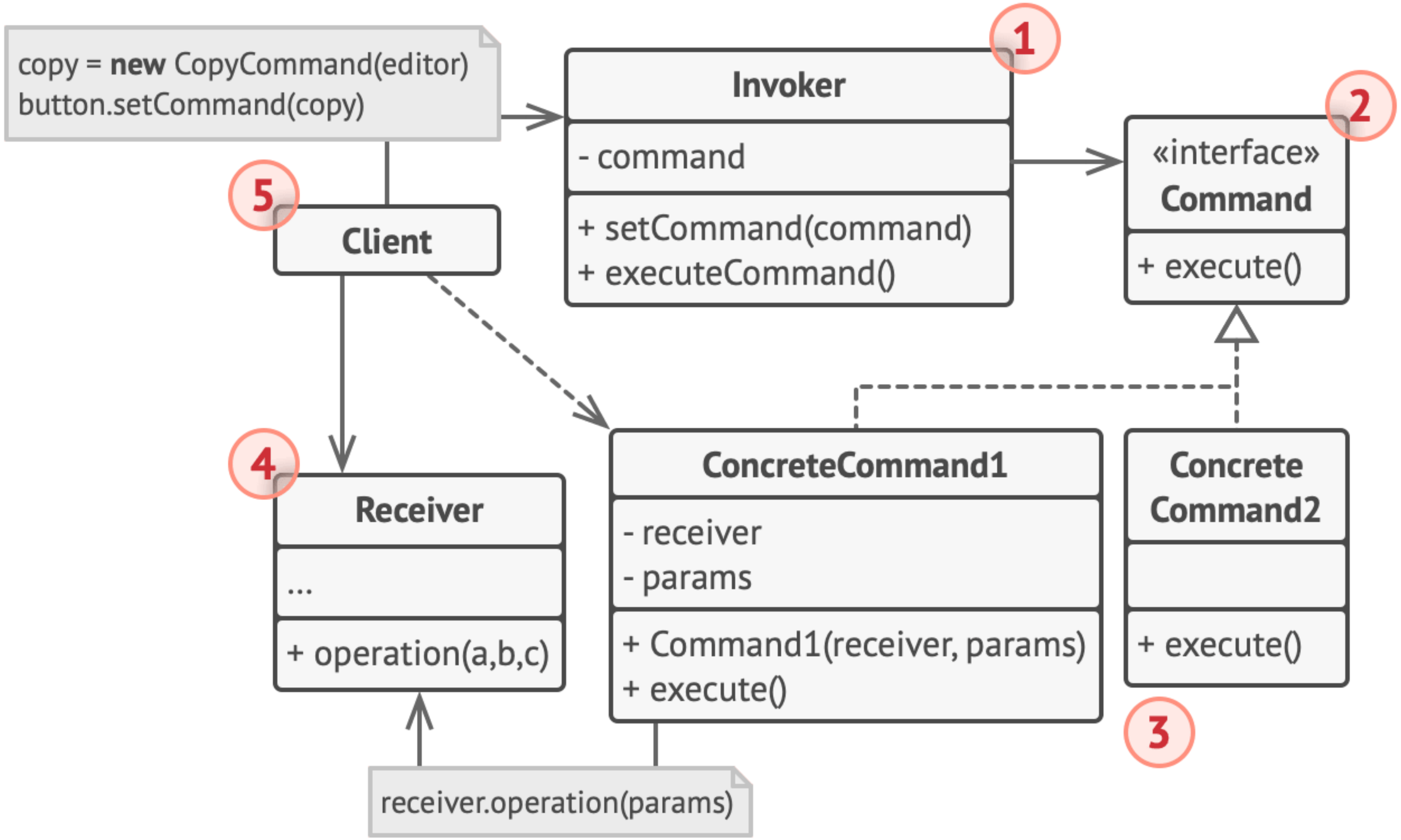


- Commands become a **convenient middle layer**, reducing coupling between the GUI and business logic

- Making commands implement the same interface
  - Use commands with the same request sender
  - Switch command objects linked to the sender
  - Problem: request parameters
    - Pre-configured with the data, or capable of getting it on its own

[Figures in this slide are extracted from https://refactoring.guru/design-patterns/command]

Command: Structure

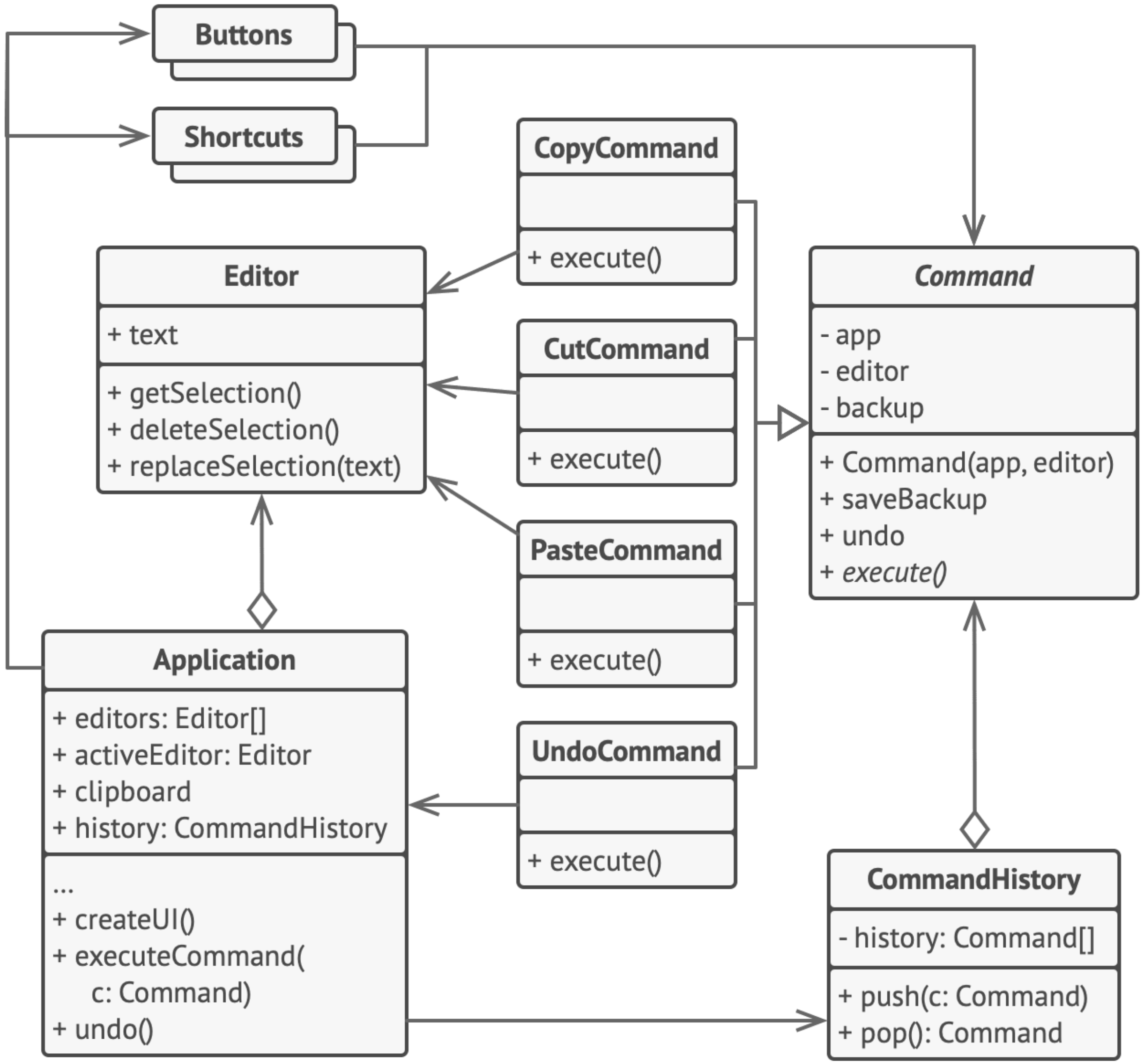


1. **Invoker (sender):** initiating requests
2. **Command:** usually, just a single method
3. **Concrete Commands:** passing calls to business logic objects
   - Parameters declared as fields
4. **Receiver:** business logic
5. **Client:** creating and configuring concrete command objects

[Figures in this slide are extracted from https://refactoring.guru/design-patterns/command]

Command: Example

- Tracking the history of executed operations and making it possible to revert an operation if needed
- The client code is not coupled to concrete command classes



[Figures in this slide are extracted from https://refactoring.guru/design-patterns/command]

Command: Applicability

- Parametrize objects with operations
  - Command turns a method call into a standalone object
    - Pass commands as method arguments, store them inside other objects, switch linked commands at runtime, etc.
- Queue operations, schedule their execution, or execute them remotely
  - Can be serialized (thus can be saved and restored)
  - Can be queued, logged and sent over the network
- Implement reversible operations
  - Most popular for implementing undo/redo
  - Implement the history of performed operations, with related backups of the app’s state

Command: Implementation

1. Declare the **command interface** with a single **execution** method
2. Extracting requests into **concrete command classes** that implement the command interface
   - Each class has a set of **fields** for storing the request arguments, and a **reference** to the actual receiver object
3. Identify classes that will act as **senders**
   - Add the fields for storing commands into these classes
   - Senders should communicate with commands only via the interface
4. Change the senders so they execute the command instead of sending a request to the receiver directly
5. The client should initialize objects in the following order
   1. Create receivers
   2. Create commands, and associate them with receivers if needed
   3. Create senders, and associate them with specific commands

Command: Pros and Cons

- **Pros**
  - Single Responsibility Principle: decoupling classes that invoke operations from classes that perform these operations
  - Open/Closed Principle: introducing new commands into the app without breaking existing client code
  - Implement undo/redo
  - Implement deferred execution of operations
  - Assemble a set of simple commands into a complex one
- **Cons**
  - The code may become more complicated, since a whole new layer between senders and receivers is introduced