Lecture 11

Design Patterns (part 3)

Topics covered

- Behavioral design patterns
  - Chain of Responsibility
  - Command
  - Interpreter
  - Mediator
  - Strategy
  - Template

Behavioral design pattern

- ✧ Concerned with communication & better interaction between objects
- ✧ Provides loose coupling and flexibility to extend easily
- ✧ Purpose: To manage communication (algorithms, relationships, interactions, and responsibilities) between objects
  - The interaction between the objects should be in such a way that they can easily talk to each other and still should be loosely coupled
  - The implementation and the client should be loosely coupled in order to avoid hard coding and dependencies

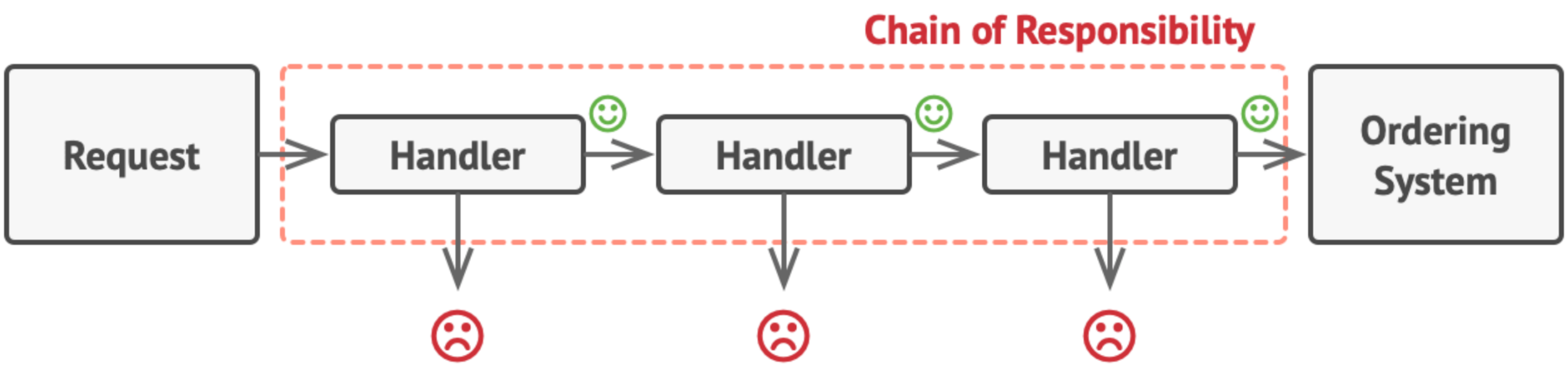
Chain of Responsibility pattern

- ✧ Avoids coupling the sender of a request to its receiver by giving multiple objects a chance to handle the request
- ✧ Normally each receiver contains reference of another receiver. If one object cannot handle the request then it passes the same to the next receiver, and so on
- ✧ Usage:
  - When more than one object can handle a request and the handler is unknown
  - When the group of objects that can handle the request must be specified in dynamic way

Chain of Responsibility pattern

✧ Each handler receives the request and may:

- Does nothing and pass the request to the next handler
- Modifies the request, then pass it to the next handler
- Throws an error so that the processing chain will be stopped
- Finish the processing chain early (without passing the request any further)



Chain of Responsibility in Spring framework

- Spring framework utilizes the Chain of Responsibility design pattern in several areas
- **Spring Security:** chaining security filters together
  - Each filter gets a chance to process the request and potentially decide whether to allow it or not.
  - If a filter doesn't handle the request, it gets passed on to the next filter in the chain
- **Spring Aspect Oriented Programming (AOP):** allows you to intercept method calls and manipulate them before or after execution using interceptors
  - These interceptors can be chained together, forming a sequence where each interceptor can decide how to proceed with the method call

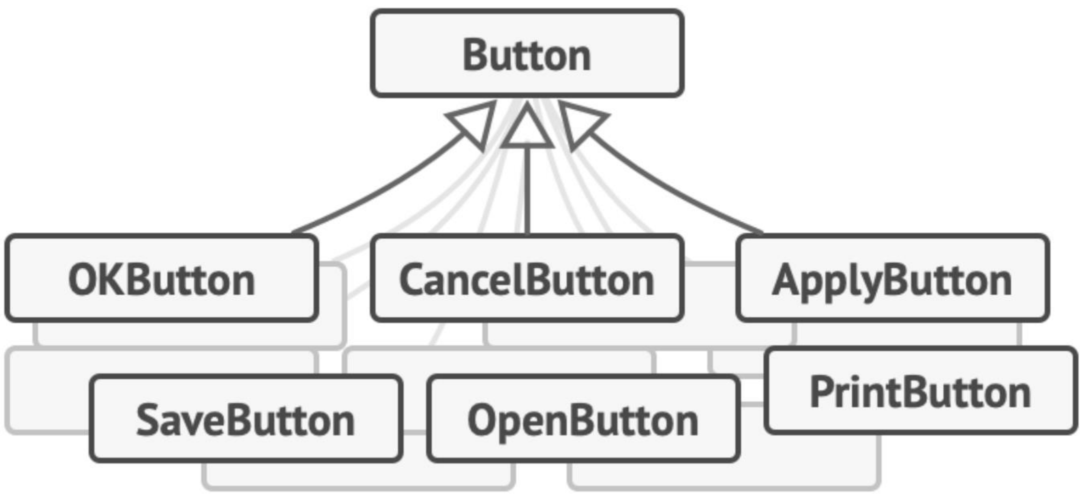
Chain of Responsibility in Node.js

- Express.js leverages the Chain of Responsibility design pattern to handle middleware in a flexible way.
- You define middleware functions and register them with the Express app using `app.use()` function call
- When a request arrives, it's passed to the first middleware in the chain.
- Each middleware can either:
  - Handle the request and end the processing chain using the `res.end()` or `res.send()` function
  - Pass the request to the next middleware function using the `next()` function call

Command pattern

✧ **Motivation Example:** your app needs to display many buttons, each button has a different functionality

- You designed a common `Button` class for all buttons in your application
- For each specific button, you create a subclass of Button
- However, **there are too many sub-classes!**



Command pattern

- ✧ Encapsulates a request (task) under a Command object and pass it to Invoker object
- ✧ Invoker object looks for the appropriate object which can handle this command and pass the command to the corresponding object and that object executes the Command
- ✧ Usage:
  - When you need parameterize objects according to action perform.
  - When you need to create, execute requests at different times.
  - When you need to support rollback, logging or transaction functionality.

Command pattern

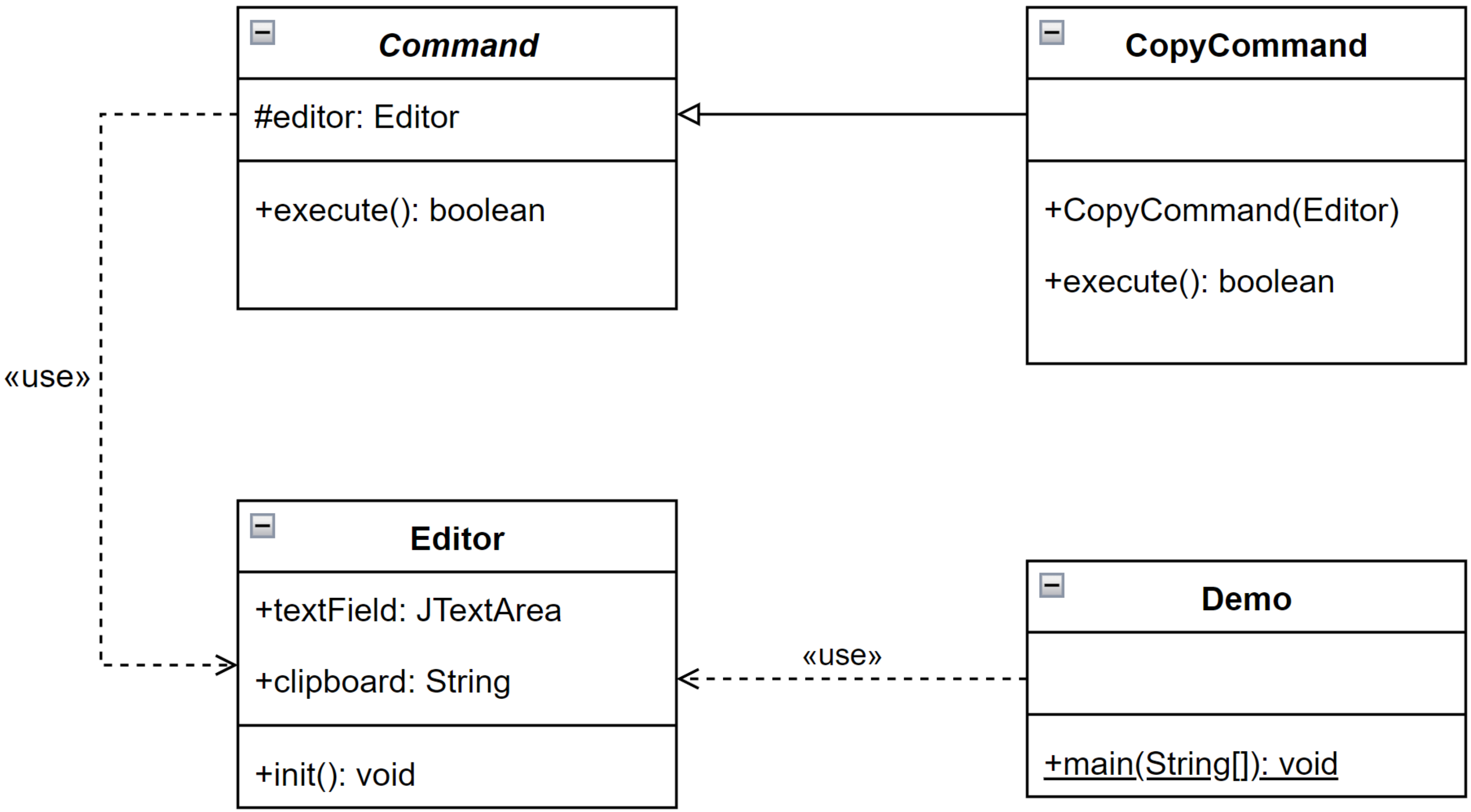
✧ Roles involved:

- **Command:** the object that encapsulates the request and knows how to execute it
- **Receiver:** the object that actually performs the action triggered by the command
- **Invoker:** the object that initiates the command execution (e.g., a button click)
- **Client** (optional): the object that creates and configures the command object

✧ Benefits:

- **Flexibility:** Commands can be parameterized, queued, or even undone/redone
- **Decoupling:** Separates the sender of the request from the receiver (object that performs the action)

Command pattern example: Text Editor



Command pattern example: Text Editor

```java
// Command
public class CopyCommand extends Command {
    public CopyCommand(Editor editor) {
        super(editor);
    }

    @Override
    public boolean execute() {
        editor.clipboard = editor.textField.getSelectedText();
        return false;
    }

}
```

Command pattern example: Text Editor

```java
public class Editor { // Invoker & Receiver
    public JTextArea textField;
    public String clipboard;

    public void init() {
        // initialize attributes
        // create GUI (JFrame, Button, JTextArea...)
        JButton ctrlC = new JButton("Ctrl+C");
        Editor editor = this;
        ctrlC.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                new CopyCommand(editor).execute();
            }
        });
        // display GUI
    }

}
```

Command pattern example: Text Editor

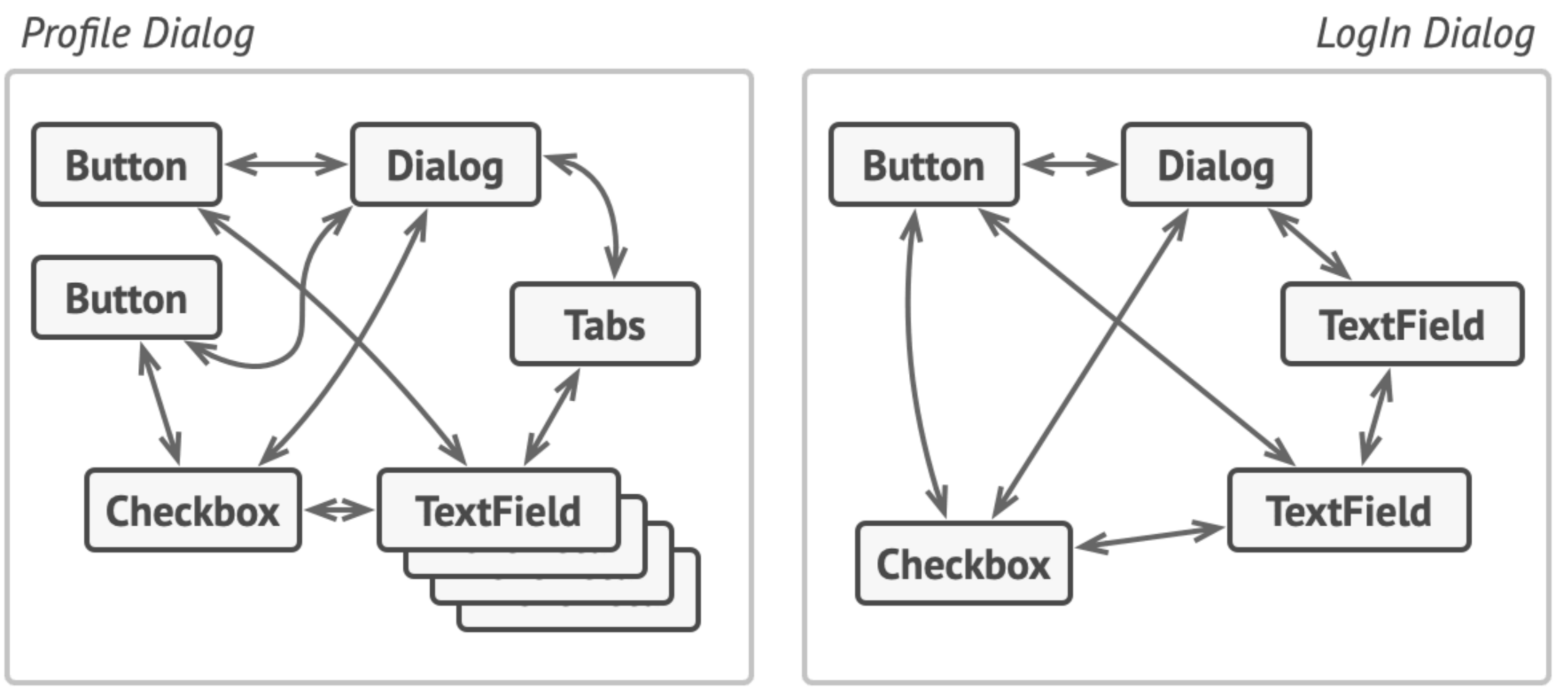
```java
// Client code
public class Demo {
    public static void main(String[] args) {
        Editor editor = new Editor();
        editor.init();
    }
}
```

Mediator pattern

- Defines an object that encapsulates how a set of objects interact
- Advantages:
  - Decouples the number of classes
  - Simplifies object protocols
  - Centralizes the control
- Usage:
  - It is commonly used in message-based systems likewise chat applications
  - When the set of objects communicate in complex but in well-defined ways

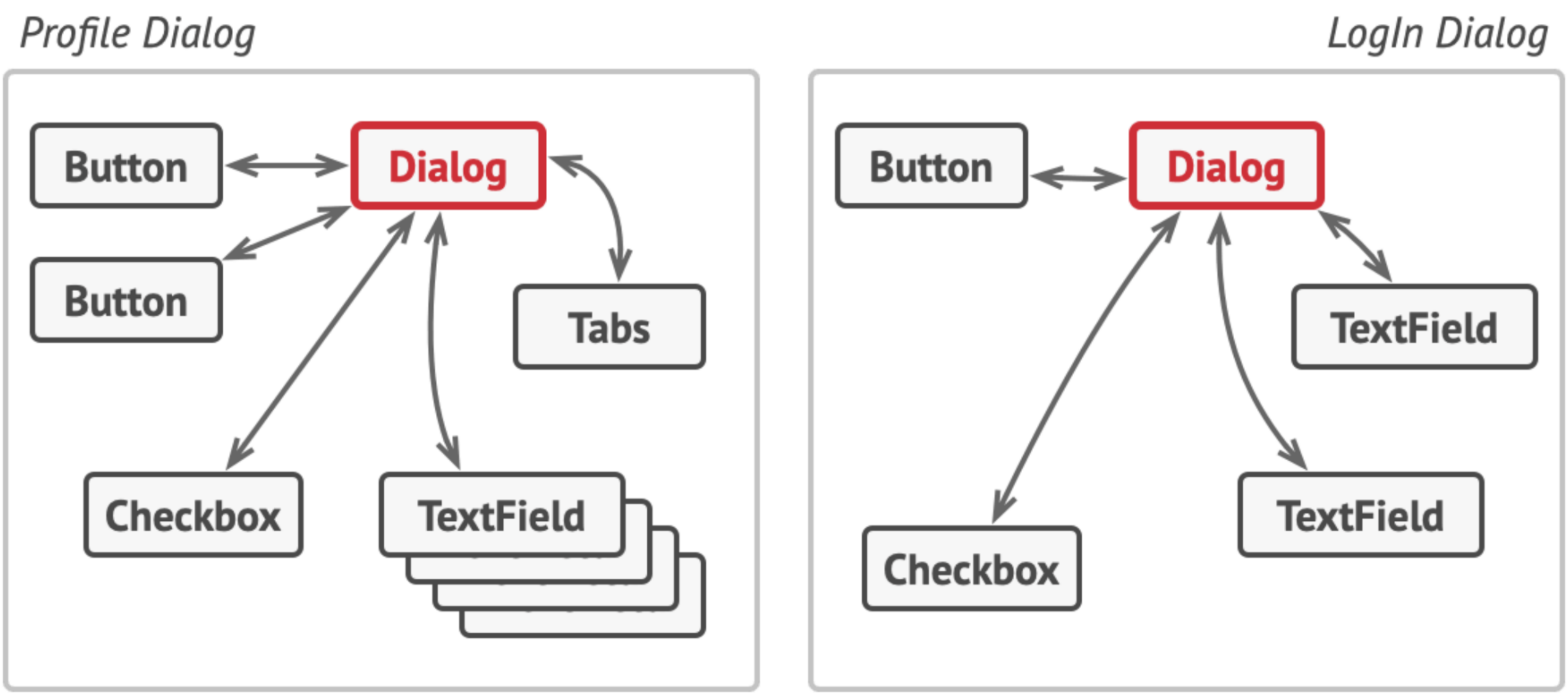
Mediator pattern's problem

✧ Component classes have complex dependencies between them

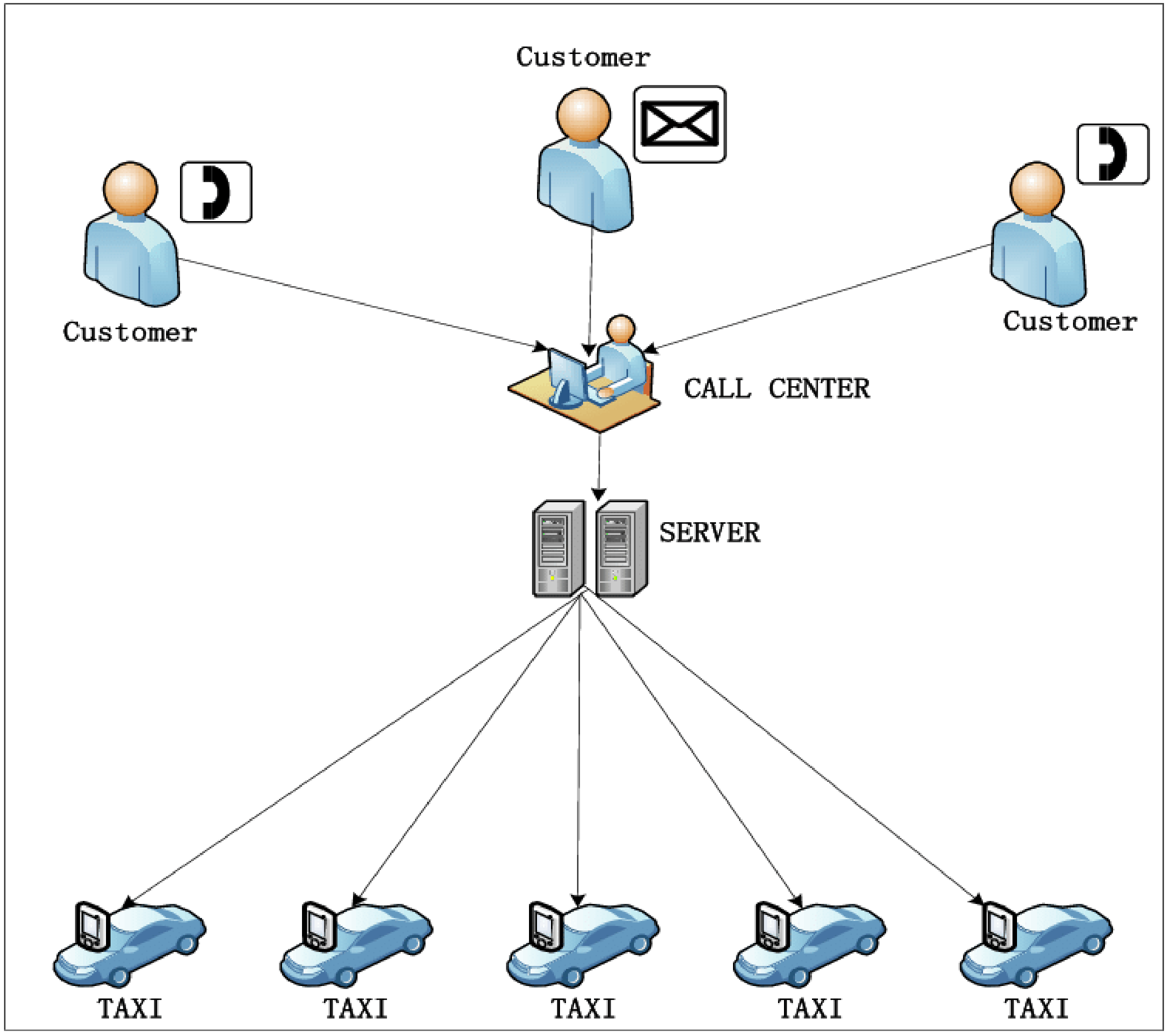


Mediator pattern: The solution

✧ Different components must collaborate indirectly, by calling a special mediator object that redirects the calls to appropriate components



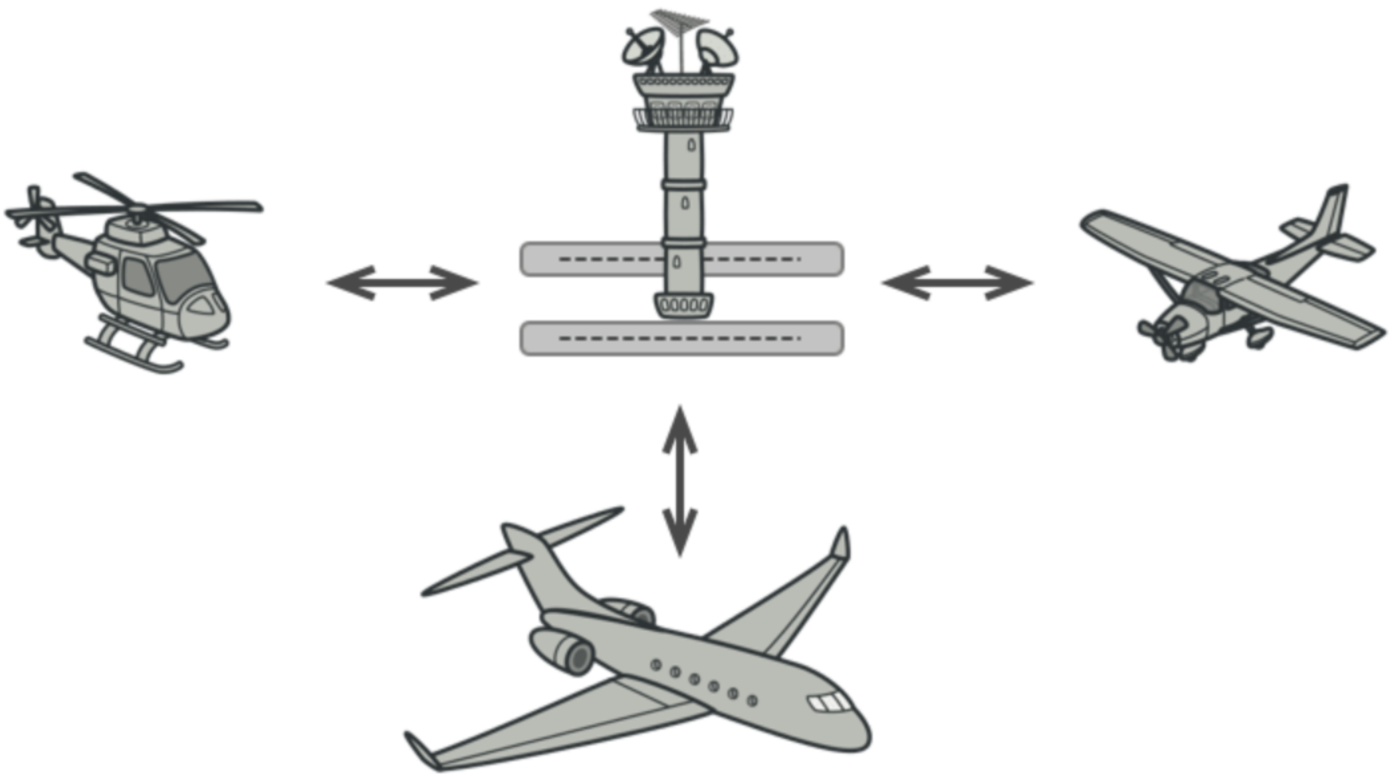
Mediator pattern: Real world examples



Taxi drivers don't communicate with each other, they all talk to a common call center.

Mediator pattern: Real world examples

✧ Air planes communicate through a communication center

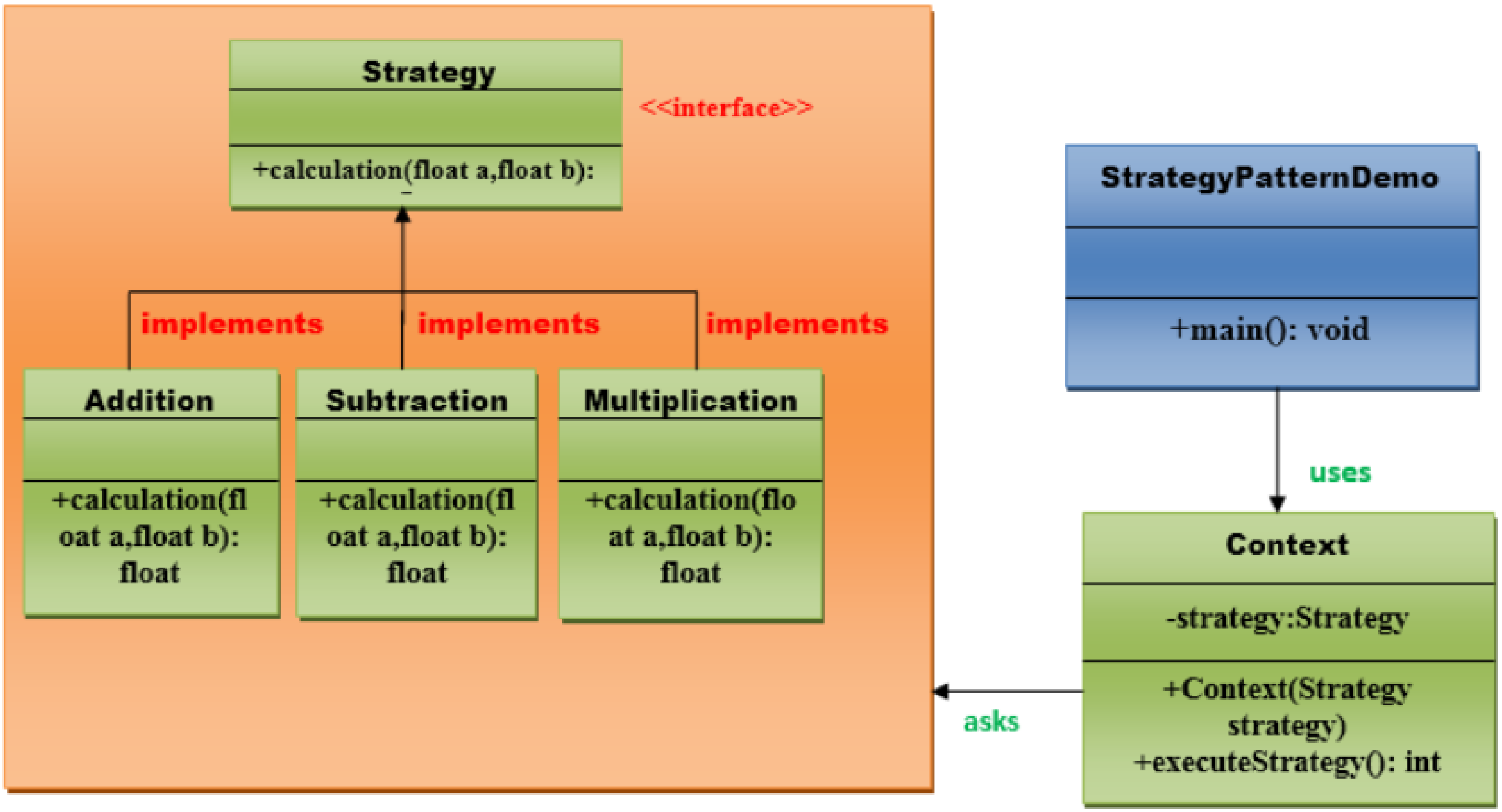


Strategy pattern

- ✧ Define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.
- ✧ Usage:
  - When the multiple classes differ only in their behaviors
  - It is used when you need different variations of an algorithm

Strategy pattern example

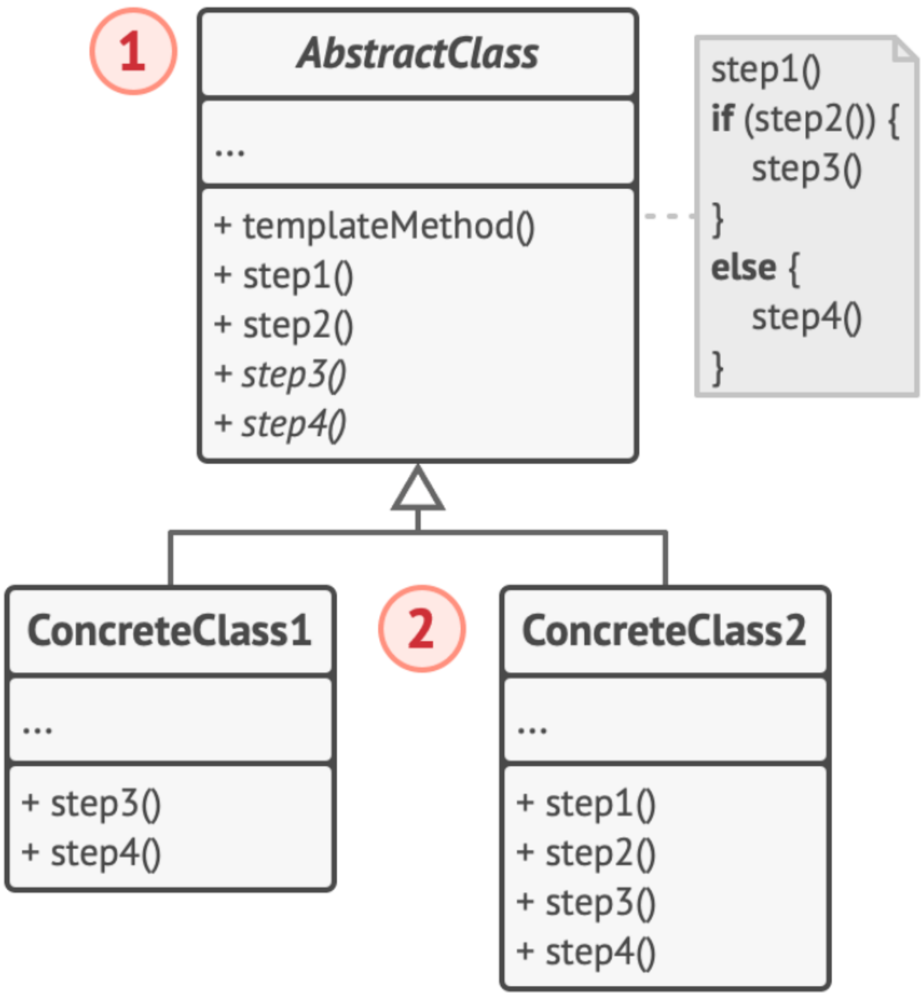
✧ Treat each operation (addition, subtraction…) as a strategy.



Template pattern

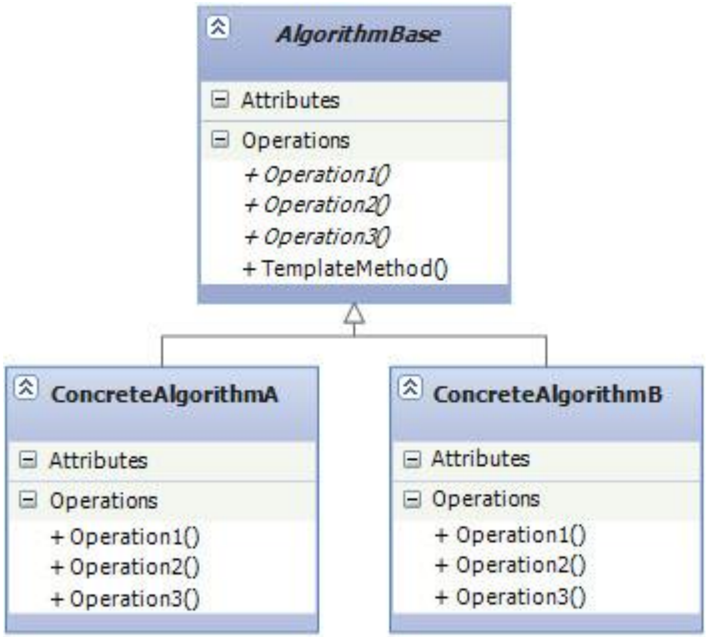
- ✧ Also known as *Template Method* design pattern
- ✧ Define the skeleton of an algorithm in a superclass but lets subclasses override specific steps of the algorithm without changing its structure
  - This pattern is particularly useful when you want to break down an algorithm into a series of steps
- ✧ Benefit
  - Reuse common code while allowing flexibility for variations in specific steps within subclasses
- ✧ Usage:
  - When the common behavior among sub-classes should be moved to a single common class to avoid duplication

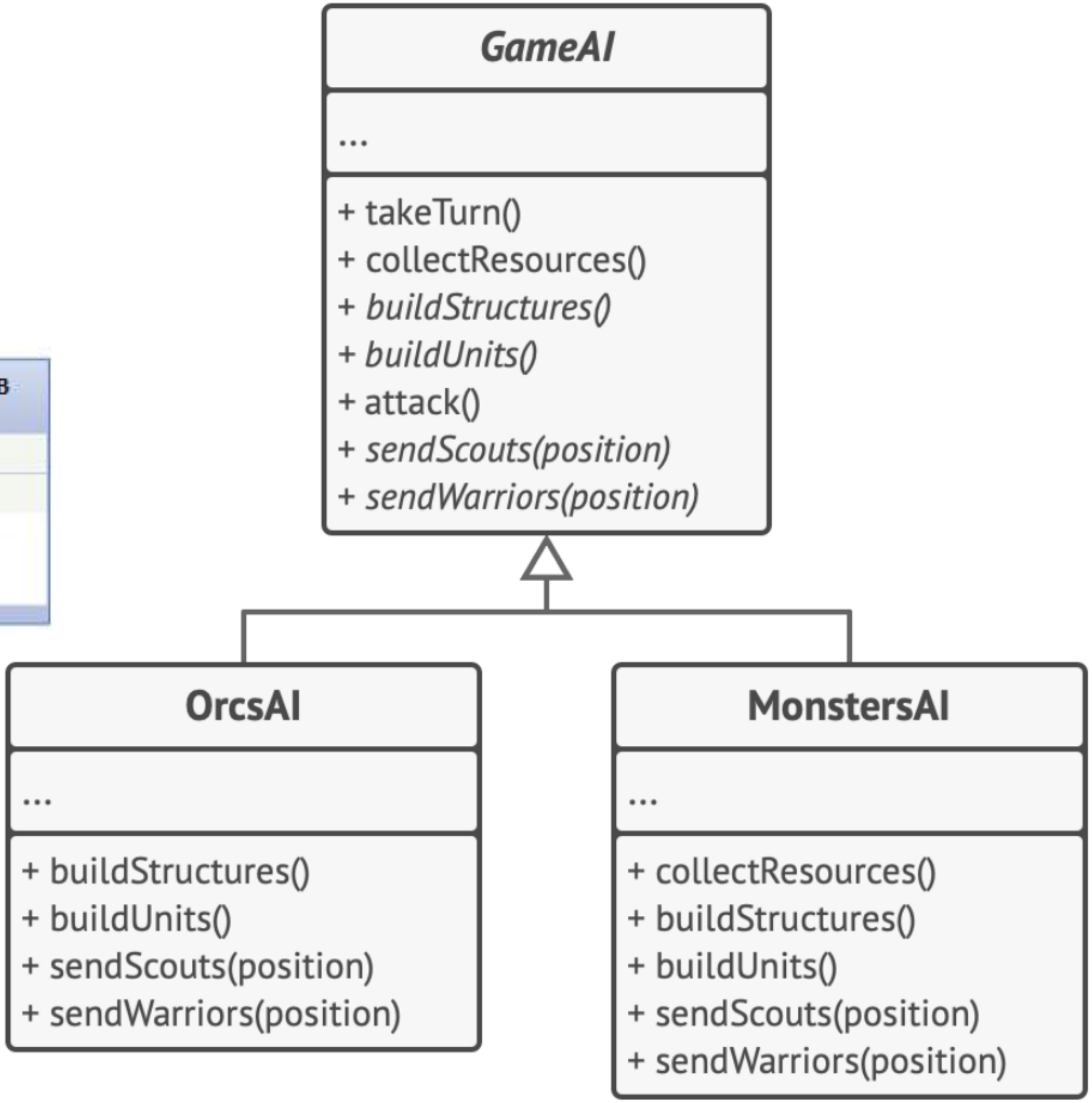
Template pattern diagram



- **Abstract class** declares methods that act as steps of an algorithm and the actual *template method* which calls these methods in a specific order.
- **Concrete classes** can override some or all of the steps, but not the template method itself.

Template pattern examples





Summary

- ✧ **Chain of Responsibility**: Passes a request along a chain of handlers where each handler can choose to process the request or pass it on.
  - Usage: support ticket systems, web server middleware…
- ✧ **Command**: Encapsulates a request as an object, allowing you to parameterize clients with queues, log requests, and support undo.
  - Usage: GUI buttons…
- ✧ **Interpreter**: Defines a grammar and an interpreter to process language elements defined in that grammar.
  - Usage: expression evaluators, rule engines…

Summary

- ✧ **Mediator**: Centralizes complex communications and control logic between related objects to reduce direct dependencies.
  - Usage: GUI form fields coordination, chat rooms, Redux…
- ✧ **Strategy**: Enables selecting an algorithm's behavior at runtime by encapsulating each algorithm in a separate class.
  - Usage: payment method switching, sorting strategies….
- ✧ **Template Method**: Defines the skeleton of an algorithm in a superclass but lets subclasses override specific steps.
  - Usage: framework lifecycle hooks, file readers…