SE 350 - Object-Oriented Software Development

Software Design Patterns: Command

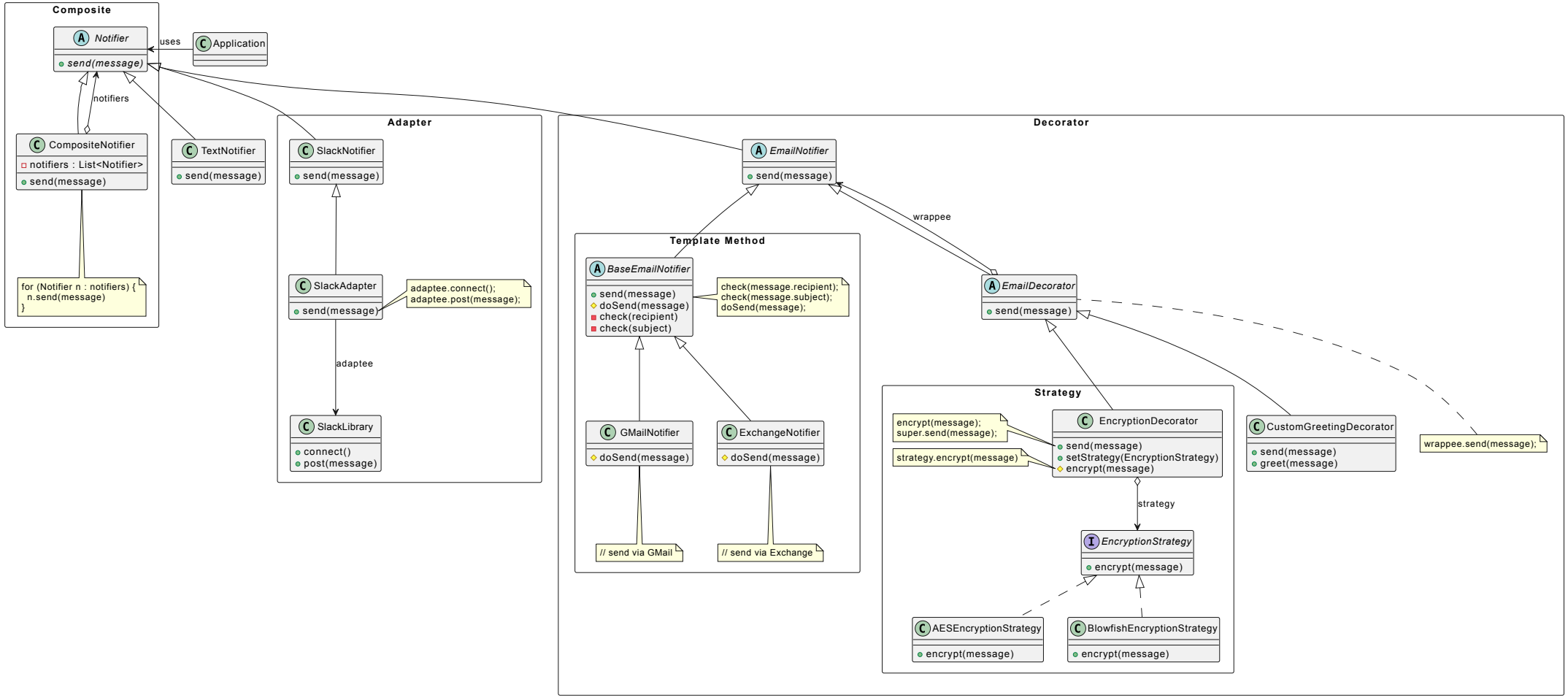
Instructor: Stefan Mitsch

Learning Objectives

- Understand the problem solved by Command
- Understand the consequences of using the pattern

Design Challenge

✓ Application that can send notifications by email, text message, Slack



? Implement a command-line interface to send messages

Design Challenge

```
public class Application {
  public static void main(String[] args) {
    Map<String, String> options = getOptions(args, Map.of());
    String message = options.get("message");
    String method = getNotification(options.get("recipient"));
    switch (method) {
      case "[email]" -> new EmailNotifier().send(message);
      case "[text]" -> // ...
      case "[email,slack]" -> // ...
      // ...
    }
  }

  private Map<String, String> getOptions(String[] args, Map<String, String> options) {
    if (args == null || args.length == 0) return options;
    switch (args[0]) {
      case "-message" -> {
        options.put("message", args[1]);
        return getOptions(Arrays.copyOfRange(args, 2, args.length), options);
      }
      case "-schedule" -> // ...
      case "-recipent" -> // ...
    }
  }
}
```

⚠ Direct coupling between command line interpretation and business logic

Design Challenge

❓ Schedule sending for later

```
Map<String, String> options = getOptions(args, Map.of());
String message = options.get("message");
String method = getNotification(options.get("recipient"));
String schedule = options.get("schedule");
switch (method) {
  case "[email]" -> new EmailNotifier().send(message);
  case "[text]" -> // ...
  case "[email,slack]" -> // ...
  // ...
}
```

⚠️ add an argument to `send`?

⚠️ add a decorator that sleeps until the scheduled time and then calls `send`?

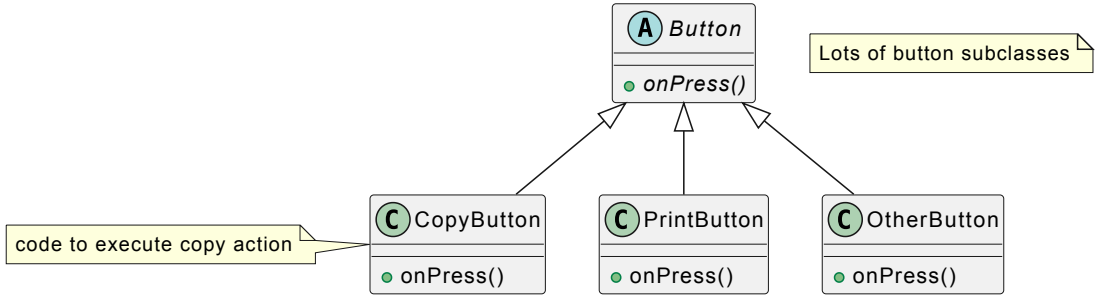
Command Introduction

Command turns a request into a stand-alone object

- Behavioral design pattern
- Lets clients pass commands
- Lets clients delay or queue command execution
- Supports undoable operations

Problem illustration: Separate user interface from business logic

❓ Where to put the code for executing a request, how to invoke the same command from multiple places?



Command Overview

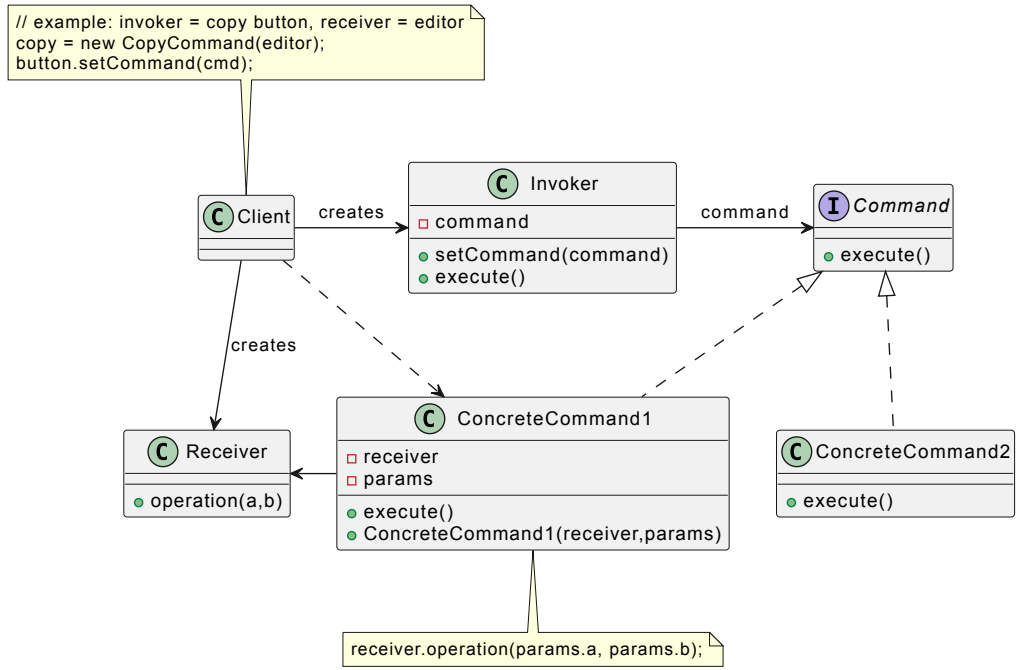
⚠ Problem

Want to avoid tight coupling between user interface and business logic; want to encapsulate requests

Intent

- Parameterize objects with operations
- Pass commands and delay execution
- Support undo/redo/command logs

Structure

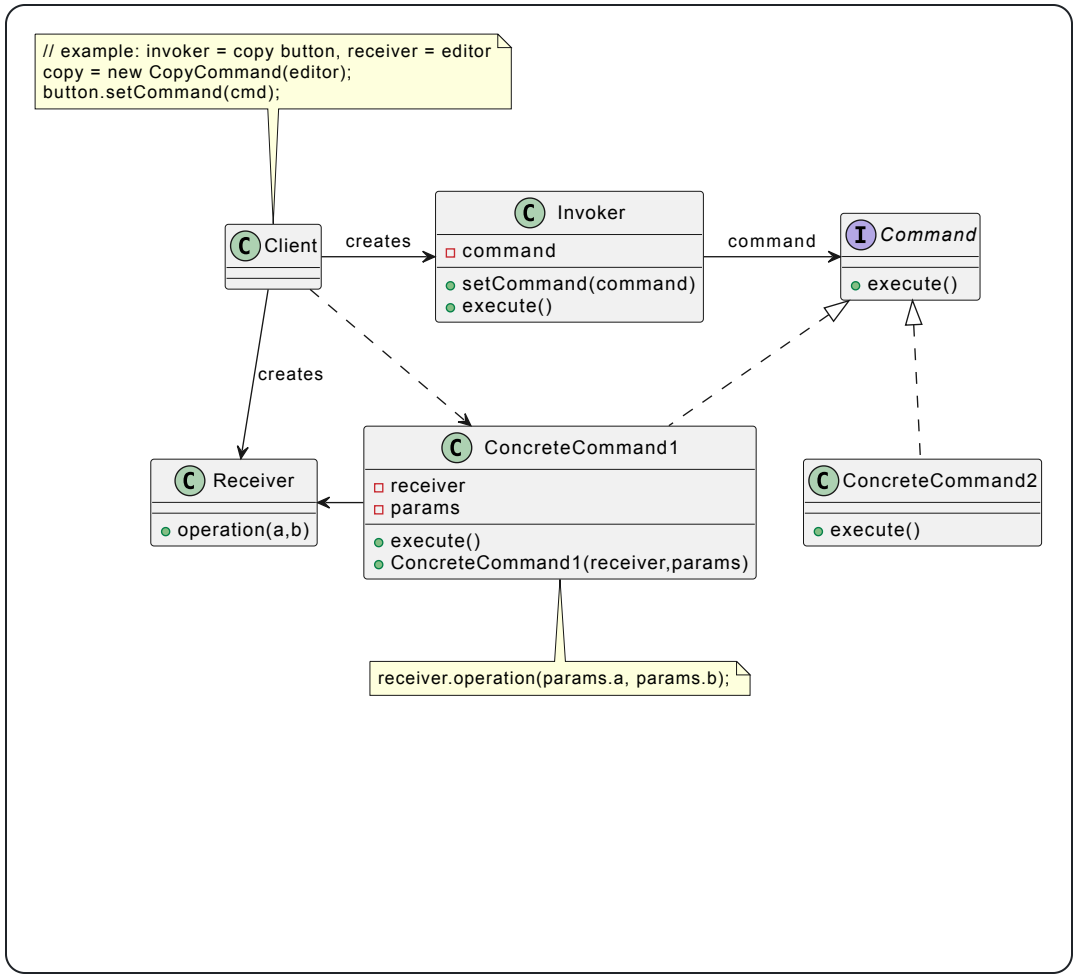


Command Participants

1. The `Invoker` (aka `Sender`) initiates a command; it has a reference for storing a command object and triggers the command instead of directly addressing the command receiver

2. The `Command` interface declares a method for executing the command

3. `ConcreteCommand` classes implement different commands; often, a concrete command does not perform work itself, it passes the call to a `Receiver`. Parameters required to execute the command or pass to the `Receiver` can be declared as fields of the `ConcreteCommand` class.



4. The `Receiver` knows how to perform work. Any object can be the receiver of a command.

5. The `Client` creates and configures concrete command objects. It must pass all required parameters, including the `Receiver` instance, to the command. The `Client` also associates the command with one or more `Invoker` instances.

Design Challenge Revisited

❓ Use the Command pattern to send messages

```
Map<String, String> options = getOptions(args, Map.of());
String message = options.get("message");
String method = getNotification(options.get("recipient"));
String schedule = options.get("schedule");
switch (method) {
  case "[email]" -> new EmailNotifier().send(message);
  case "[text]" -> // ...
  case "[email,slack]" -> // ...
  // ...
}
```

Design Challenge Improved

```
// command interface
public interface NotificationCommand {
  void execute();
  Date getSchedule();
}

// concrete commands
public class SendCommand {
  private Notifier n;
  private String message;
  private Instant schedule;
  public SendCommand(Notifier n, String message, Instant schedule) { /* ... */ }
  public void execute() { n.send(message); }
  public Instant getSchedule() { /* ... */ }
}

// command queue and scheduler
public class NotificationManager implements Runnable {
  private List<NotificationCommand> queue = List.of();
  private List<NotificationCommand> executed = List.of();
  public void addCommand(NotificationCommand cmd) {
    // determine where to add the command in the send queue
    int idx = // ...
    queue.add(idx, cmd);
  }

  public void run() {
    while (true) {
      while (queue.isEmpty()
          || queue.get(0).getSchedule().isAfter(Instant.now()))
        // wait on empty queue, then fetch first element
      NotificationCommand next = queue.remove(0);
      next.execute();
      executed.add(next);
    }
  }
}
```

```
public interface CommandFactory {
  NotificationCommand create(String message, Instant schedule);
}

public class Application {
  // populate command factories
  private Map<String,CommandFactory> commands = Map.of(
    "[email]", (message, schedule) -> new SendCommand(new EmailNotifier(), message, schedule),
    "[text]", // ...
    // ...
  );
  // command scheduler
  private NotificationManager mgr;

  public Application(NotificationManager mgr) {
    this.mgr = mgr;
  }

  public void process(String[] args) {
    // read options
    Map<String, String> options = getOptions(args, Map.of());
    String message = options.get("message");
    String method = getNotification(options.get("recipient"));
    Instant schedule = getSchedule(options.get("schedule"));

    // queue command
    mgr.add(commands.get(method).create(message, schedule));
  }
}
```

Command Applicability

- **Want to parametrize objects with operations**:
  Command pattern turns a method into a stand-alone object; can pass commands as arguments (higher-order functions), store commands, switch commands
- **Want to queue or schedule execution**:
  Commands can be serialized, stored for later execution, or sent for remote execution.
- **Want to implement reversible operations**:
  Command pattern can implement command and its undo command in a single class; command history keeps track of sequence of executed commands; combine with the Memento pattern to obtain state backups.

Command Implementation

- Declare the `Command` interface with an `execute` method (optional `undo`)
- Identify classes that act as senders, add fields for storing commands
- Change the sender to execute the command instead of calling the receiver
- Client sets up the pattern as follows:
  - Create `Receiver` objects
  - Create `Command` objects, associate them with receivers
  - Create `Sender` objects, associate them with commands

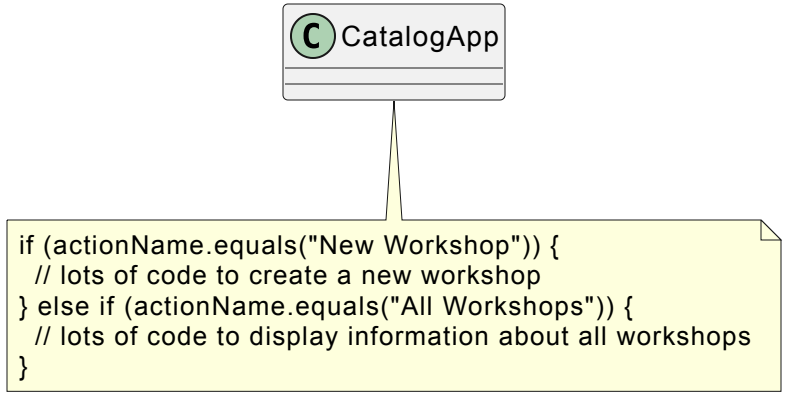
```
// command interface
public interface PrintCommand {
  void execute(String text);
}

// concrete commands
public class PrintConsoleCommand implements PrintCommand {
  // implicit receiver: System.out
  public void execute(String text) { System.out.println(text); }
}
public class PrintFileCommand implements PrintCommand {
  private String fileName;
  public PrintFileCommand(String fileName) {
    this.fileName = fileName;
  }
  public void execute(String text) {
    // exception handling omitted
    BufferedWriter w = new BufferedWriter(new FileWriter(fileName));
    w.write(text);
    w.close
  }
}

// sender
public class Fibonacci {
  private PrintCommand cmd;
  public Fibonacci(PrintCommand cmd) { this.cmd = cmd; }
  public long getNthFibonacci(int n) {
    long result = 0;
    if (n <= 1) result = n;
    else result = getNthFibonacci(n-2) + getNthFibonacci(n-1);
    cmd.execute(n + "-th fibonacci number is " + result);
    return result;
  }
}

// client
public class Demo {
  public static void main(String[] args) {
    // create command + receiver
    PrintCommand c = new PrintFileCommand("fib.txt");
    // create sender and associate with command
    Fibonacci f = new Fibonacci(c);
    // call sender; sender uses command to print to file
    f.getNthFibonacci(5);
  }
}
```

Replace Conditional Dispatcher with Command

Code Smell: Conditional logic is used to dispatch requests and execute actions

Before: Extensive conditional logic with action details

▶ After Refactoring



Command Discussion

Benefits

- Single Responsibility Principle: decouples sender from receiver
- Open/Closed Principle: implement `Command` interface for new commands
- Implement undo/redo, delayed commands
- Assemble simple commands into Composite commands

Drawbacks

- Can introduce unnecessary complexity
- Prefer functional language features to implement higher-order functions

Relations with Other Patterns

- Combine with `Memento` to implement state backups for undo
- `Command` and `Strategy` look similar but have different intent: `Command` converts an operation into an object so it can be passed around and stored; `Strategy` swaps algorithms