# Week 4. Behavioral Design Patterns

1. **Template Design Pattern Experiment**

**Objective:** Implement a simple framework for data processing tasks using the Template Method pattern.

* 1. Create an abstract class DataProcessor with a template method process() that defines the steps for processing data (e.g., loadData, processData, saveData).
  2. Implement at least two concrete subclasses (CSVDataProcessor and JSONDataProcessor) that override the necessary steps, providing specific implementations for handling CSV and JSON data formats.
  3. Analyze the issue in this scenario that the Template pattern is meant to address.

**Code : Class TemplateMethodPattern**

public class TemplateMethodPattern {

abstract static class DataProcessor {

public final void process() {

loadData();

processData();

saveData();

}

protected abstract void loadData();

protected abstract void processData();

protected abstract void saveData();

}

static class CSVDataProcessor extends DataProcessor {

@Override

protected void loadData() {

System.out.println("Loading data from a CSV file...");

}

@Override

protected void processData() {

System.out.println("Processing data from the CSV file...");

}

@Override

protected void saveData() {

System.out.println("Saving processed data to a CSV file...");

}

}

static class JSONDataProcessor extends DataProcessor {

@Override

protected void loadData() {

System.out.println("Loading data from a JSON file...");

}

@Override

protected void processData() {

System.out.println("Processing data from the JSON file...");

}

@Override

protected void saveData() {

System.out.println("Saving processed data to a JSON file...");

}

}

public static void main(String[] args) {

System.out.println("CSV Data Processing:");

DataProcessor csvProcessor = new CSVDataProcessor();

csvProcessor.process();

System.out.println();

System.out.println("JSON Data Processing:");

DataProcessor jsonProcessor = new JSONDataProcessor();

jsonProcessor.process();

}

}

**Output:**

A screen shot of a computer

Description automatically generated

# Request Processing Pipeline

**Objective:** Implement a request processing pipeline where each handler processes a request or passes it to the next handler.

* 1. Create Handler Interface: Define a Handler interface with a method handleRequest(request: Request).
  2. Implement Concrete Handlers: Implement handlers like AuthenticationHandler, ValidationHandler, BusinessLogicHandler that process specific aspects of the request.
  3. Create Chain: Link handlers together to form a chain where each handler can pass the request to the next handler if it cannot handle it.
  4. Test Chain: Create a client class that sends requests through the chain and observes how they are processed by different handlers.
  5. Analyze the issue in this scenario that the Chain of Responsibility pattern is meant to address.

**Code : Class ChainOfResponsibilityDemo**

public class ChainOfResponsibilityDemo {

interface Handler {

void setNextHandler(Handler next);

void handleRequest(Request request);

}

static class Request {

private String message;

private boolean authenticated;

private boolean valid;

public Request(String message) {

this.message = message;

this.authenticated = false;

this.valid = false;

}

public String getMessage() {

return message;

}

public boolean isAuthenticated() {

return authenticated;

}

public void setAuthenticated(boolean authenticated) {

this.authenticated = authenticated;

}

public boolean isValid() {

return valid;

}

public void setValid(boolean valid) {

this.valid = valid;

}

}

static class AuthenticationHandler implements Handler {

private Handler next;

@Override

public void setNextHandler(Handler next) {

this.next = next;

}

@Override

public void handleRequest(Request request) {

if (authenticate(request)) {

System.out.println("Authentication successful.");

request.setAuthenticated(true);

if (next != null) {

next.handleRequest(request);

}

} else {

System.out.println("Authentication failed. Request denied.");

}

}

private boolean authenticate(Request request) {

return request.getMessage().startsWith("AUTH");

}

}

static class ValidationHandler implements Handler {

private Handler next;

@Override

public void setNextHandler(Handler next) {

this.next = next;

}

@Override

public void handleRequest(Request request) {

if (request.isAuthenticated() && validate(request)) {

System.out.println("Validation successful.");

request.setValid(true);

if (next != null) {

next.handleRequest(request);

}

} else {

System.out.println("Validation failed. Request denied.");

}

}

private boolean validate(Request request) {

return request.getMessage().contains("VALID");

}

}

static class BusinessLogicHandler implements Handler {

private Handler next;

@Override

public void setNextHandler(Handler next) {

this.next = next;

}

@Override

public void handleRequest(Request request) {

if (request.isAuthenticated() && request.isValid()) {

System.out.println("Business logic processed: " + request.getMessage());

} else {

System.out.println("Request did not pass previous steps. Business logic not applied.");

}

}

}

public static void main(String[] args) {

Request request1 = new Request("AUTH\_VALID\_REQUEST");

Request request2 = new Request("AUTH\_INVALID\_REQUEST");

Request request3 = new Request("INVALID\_REQUEST");

Handler authHandler = new AuthenticationHandler();

Handler validationHandler = new ValidationHandler();

Handler businessHandler = new BusinessLogicHandler();

authHandler.setNextHandler(validationHandler);

validationHandler.setNextHandler(businessHandler);

System.out.println("Processing request 1:");

authHandler.handleRequest(request1);

System.out.println("\nProcessing request 2:");

authHandler.handleRequest(request2);

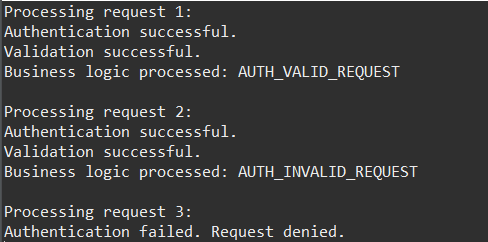
System.out.println("\nProcessing request 3:");

authHandler.handleRequest(request3);

}

}

**Output:**



# Command Design Pattern

**Objective:** Develop a command-based task execution system for a text editor.

* 1. Create a command interface Command with an execute() method.
  2. Implement concrete command classes (CopyCommand, PasteCommand, UndoCommand) that perform specific actions in a text editor.
  3. Design a TextEditor class that can execute commands and maintain a history for undo functionality.
  4. Analyze the issue in this scenario that the Command pattern is meant to address.

**Code : Class CommandPatternExample**

import java.util.Stack;

public class CommandPatternExample {

interface Command {

void execute();

}

static class TextEditor {

private String text;

private Stack<String> history;

public TextEditor() {

text = "";

history = new Stack<>();

}

public void copy() {

history.push(text);

System.***out***.println("Copying: " + text);

}

public void paste() {

history.push(text);

text = "Pasted Content";

System.***out***.println("Pasting: " + text);

}

public void undo() {

if (!history.isEmpty()) {

text = history.pop();

System.***out***.println("Undo: " + text);

} else {

System.***out***.println("No action to undo.");

}

}

public void setText(String newText) {

text = newText;

}

public String getText() {

return text;

}

}

static class CopyCommand implements Command {

private TextEditor editor;

public CopyCommand(TextEditor editor) {

this.editor = editor;

}

*@Override*

public void execute() {

editor.copy();

}

}

static class PasteCommand implements Command {

private TextEditor editor;

public PasteCommand(TextEditor editor) {

this.editor = editor;

}

*@Override*

public void execute() {

editor.paste();

}

}

static class UndoCommand implements Command {

private TextEditor editor;

public UndoCommand(TextEditor editor) {

this.editor = editor;

}

*@Override*

public void execute() {

editor.undo();

}

}

public static void main(String[] args) {

TextEditor editor = new TextEditor();

editor.setText("Hello, world!");

Command copy = new CopyCommand(editor);

Command paste = new PasteCommand(editor);

Command undo = new UndoCommand(editor);

copy.execute();

paste.execute();

undo.execute();

editor.setText("New text.");

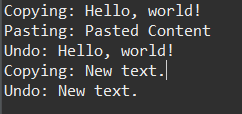
copy.execute();

undo.execute();

}

}

**Output:**



# Customer support ticketing system

The system must handle customer issues based on their complexity and escalate them if necessary. The support hierarchy consists of three levels:

* 1. **Level 1 Support**: Handles basic issues (e.g., password reset).
  2. **Level 2 Support**: Handles intermediate issues (e.g., software installation).
  3. **Level 3 Support**: Handles advanced issues (e.g., debugging critical software errors).

Each level can either resolve the issue or escalate it to the next level if it is beyond their scope.

# Requirements:

1. Implement the **Chain of Responsibility pattern** to model this support system.
2. Each support level should log a message when it handles or escalates a ticket.
3. If no level can handle the ticket, log a message saying the issue cannot be resolved.

**Code : Class CustomerSupportSystem**

public class CustomerSupportSystem {

abstract static class SupportLevel {

protected SupportLevel nextLevel;

public void setNextLevel(SupportLevel nextLevel) {

this.nextLevel = nextLevel;

}

public abstract void handleTicket(Issue issue);

}

static class Level1Support extends SupportLevel {

@Override

public void handleTicket(Issue issue) {

if ("low".equals(issue.getComplexity())) {

System.out.println("Level 1 Support: Handling basic issue - " + issue.getDescription());

} else if (nextLevel != null) {

System.out.println("Level 1 Support: Escalating issue - " + issue.getDescription());

nextLevel.handleTicket(issue);

} else {

System.out.println("Level 1 Support: Cannot resolve issue - " + issue.getDescription());

}

}

}

static class Level2Support extends SupportLevel {

@Override

public void handleTicket(Issue issue) {

if ("medium".equals(issue.getComplexity())) {

System.out.println("Level 2 Support: Handling intermediate issue - " + issue.getDescription());

} else if (nextLevel != null) {

System.out.println("Level 2 Support: Escalating issue - " + issue.getDescription());

nextLevel.handleTicket(issue);

} else {

System.out.println("Level 2 Support: Cannot resolve issue - " + issue.getDescription());

}

}

}

static class Level3Support extends SupportLevel {

@Override

public void handleTicket(Issue issue) {

if ("high".equals(issue.getComplexity())) {

System.out.println("Level 3 Support: Handling advanced issue - " + issue.getDescription());

} else {

System.out.println("Level 3 Support: Cannot resolve issue - " + issue.getDescription());

}

}

}

static class Issue {

private String description;

private String complexity;

public Issue(String description, String complexity) {

this.description = description;

this.complexity = complexity;

}

public String getDescription() {

return description;

}

public String getComplexity() {

return complexity;

}

}

public static void main(String[] args) {

Level1Support level1 = new Level1Support();

Level2Support level2 = new Level2Support();

Level3Support level3 = new Level3Support();

level1.setNextLevel(level2);

level2.setNextLevel(level3);

Issue issue1 = new Issue("Password reset", "low");

Issue issue2 = new Issue("Software installation", "medium");

Issue issue3 = new Issue("Critical software error", "high");

Issue issue4 = new Issue("Unresolvable issue", "unknown");

level1.handleTicket(issue1);

level1.handleTicket(issue2);

level1.handleTicket(issue3);

level1.handleTicket(issue4);

}

}

**Output:**

