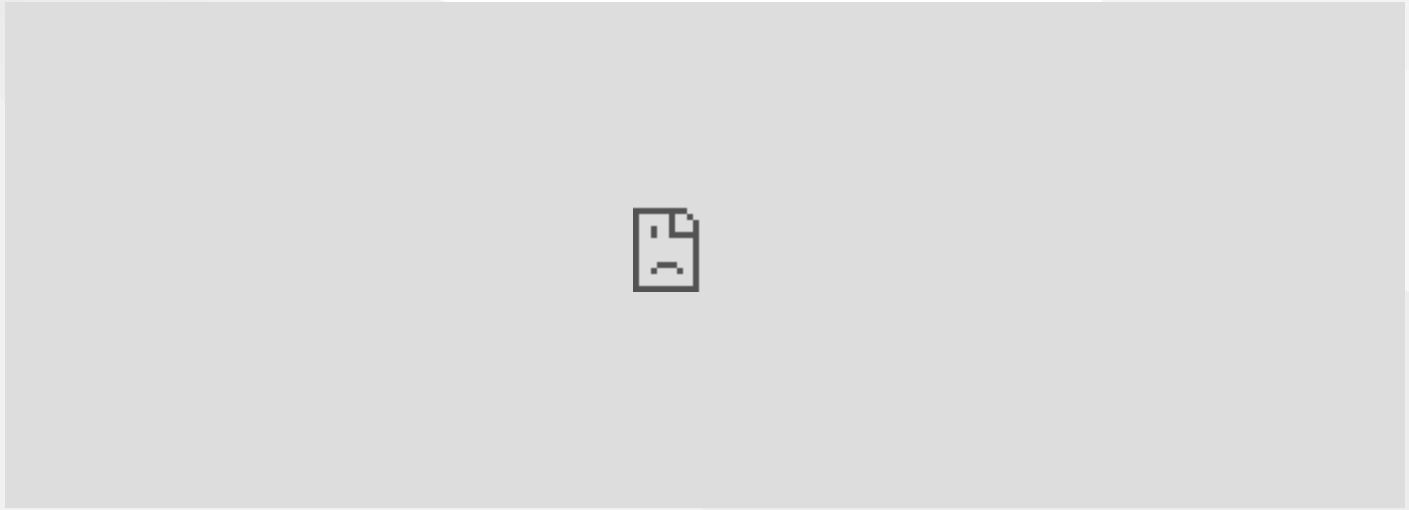
CEN206 Object-Oriented Programming

Week-11 (Advanced Design Patterns & Practical Applications)

Spring Semester, 2024-2025

Download DOC-PDF, DOC-DOCX, SLIDE, PPTX



Advanced Design Patterns & Practical Applications

Outline

- Additional Design Patterns
  - Composite Pattern
  - Façade Pattern
  - Proxy Pattern
  - Command Pattern
  - Template Method Pattern
- Real-world Applications of Design Patterns
- Combining Multiple Patterns
- Java Framework Case Studies

Composite Pattern

Allows you to compose objects into tree structures to represent part-whole hierarchies.

Composite lets clients treat individual objects and compositions of objects uniformly.

```
// Component
interface Component {
    void operation();
}

// Leaf
class Leaf implements Component {
    private String name;

    public Leaf(String name) {
        this.name = name;
    }

    @Override
    public void operation() {
        System.out.println("Leaf " + name + " operation");
    }
}

// Composite
class Composite implements Component {
    private List<Component> children = new ArrayList<>();
    private String name;

    public Composite(String name) {
        this.name = name;
    }

    public void add(Component component) {
        children.add(component);
    }

    public void remove(Component component) {
        children.remove(component);
    }

    @Override
    public void operation() {
        System.out.println("Composite " + name + " operation");
        for (Component component : children) {
            component.operation();
        }
    }
}
```

When to Use Composite Pattern

Application Scenarios

- File and folder structures
- GUI hierarchies and containers
- Organization hierarchies
- Menu systems with submenus
- Any tree-like structure where individual elements and groups need the same interface

Implementation Considerations

- Define a common interface for both leaf and composite elements
- Consider whether to include parent references
- Choose between transparency and safety approaches
- Determine how to handle child management operations

Façade Pattern

Provides a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.

```
// Complex subsystem classes
class CPU {
    public void freeze() { System.out.println("CPU: Freezing"); }
    public void jump(long position) { System.out.println("CPU: Jumping to " + position); }
    public void execute() { System.out.println("CPU: Executing"); }
}

class Memory {
    public void load(long position, byte[] data) {
        System.out.println("Memory: Loading from " + position);
    }
}

class HardDrive {
    public byte[] read(long lba, int size) {
        System.out.println("HardDrive: Reading from " + lba + " with size " + size);
        return new byte[size];
    }
}

// Façade
class ComputerFacade {
    private CPU cpu;
    private Memory memory;
    private HardDrive hardDrive;

    public ComputerFacade() {
        this.cpu = new CPU();
        this.memory = new Memory();
        this.hardDrive = new HardDrive();
    }

    public void start() {
        cpu.freeze();
        memory.load(0, hardDrive.read(0, 1024));
        cpu.jump(0);
        cpu.execute();
    }
}
```

When to Use Façade Pattern

Application Scenarios

- Simplifying complex subsystems
- Creating a cleaner API for client code
- Decoupling client code from subsystem implementation details
- When a system is being refactored and APIs need to evolve
- Creating entry points to different layers of your application

Implementation Considerations

- The façade should be a lightweight wrapper
- Multiple façades can be created for the same subsystem
- Consider making subsystem classes package-private when possible
- Don't overload the façade with too many responsibilities

Proxy Pattern

Provides a surrogate or placeholder for another object to control access to it.

```
// Subject interface
interface Image {
    void display();
}

// Real Subject
class RealImage implements Image {
    private String filename;

    public RealImage(String filename) {
        this.filename = filename;
        loadFromDisk();
    }

    private void loadFromDisk() {
        System.out.println("Loading " + filename);
    }

    @Override
    public void display() {
        System.out.println("Displaying " + filename);
    }
}

// Proxy
class ProxyImage implements Image {
    private String filename;
    private RealImage realImage;

    public ProxyImage(String filename) {
        this.filename = filename;
    }

    @Override
    public void display() {
        if (realImage == null) {
            realImage = new RealImage(filename);
        }
        realImage.display();
    }
}
```

Virtual Proxy

- Creates expensive objects on demand (lazy loading)
- Example: Image loading in a document

Protection Proxy

- Controls access to the original object
- Example: Access control systems

Remote Proxy

- Represents an object located in a different address space
- Example: RMI (Remote Method Invocation) in Java

Smart Proxy

- Performs additional actions when the object is accessed

Command Pattern

Encapsulates a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

```
// Command interface
interface Command {
    void execute();
}

// Receiver
class Light {
    public void turnOn() {
        System.out.println("Light is on");
    }

    public void turnOff() {
        System.out.println("Light is off");
    }
}

// Concrete Commands
class LightOnCommand implements Command {
    private Light light;

    public LightOnCommand(Light light) {
        this.light = light;
    }

    @Override
    public void execute() {
        light.turnOn();
    }
}

class LightOffCommand implements Command {
    private Light light;

    public LightOffCommand(Light light) {
        this.light = light;
    }

    @Override
    public void execute() {
        light.turnOff();
    }
}
```

```
// Invoker
class RemoteControl {
    private Command command;

    public void setCommand(Command command) {
        this.command = command;
    }

    public void pressButton() {
        command.execute();
    }
}

// Client code
// Light light = new Light();
// Command lightOn = new LightOnCommand(light);
// Command lightOff = new LightOffCommand(light);
// RemoteControl remote = new RemoteControl();
// remote.setCommand(lightOn);
// remote.pressButton();
// remote.setCommand(lightOff);
// remote.pressButton();
```

Command Pattern with Undo Functionality

```
interface Command {
    void execute();
    void undo();
}

class LightOnCommand implements Command {
    private Light light;

    public LightOnCommand(Light light) {
        this.light = light;
    }

    @Override
    public void execute() {
        light.turnOn();
    }

    @Override
    public void undo() {
        light.turnOff();
    }
}

class RemoteControlWithUndo {
    private Command command;
    private Stack<Command> history = new Stack<>();

    public void setCommand(Command command) {
        this.command = command;
    }

    public void pressButton() {
        command.execute();
        history.push(command);
    }

    public void pressUndo() {
        if (!history.isEmpty()) {
            Command lastCommand = history.pop();
            lastCommand.undo();
        }
    }
}
```

Template Method Pattern

Defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

```
// Abstract class with template method
abstract class DocumentProcessor {
    // Template method
    public final void processDocument() {
        openDocument();
        content = readContent();
        processContent(content);
        saveDocument();
        closeDocument();
    }

    // Methods that will be implemented by subclasses
    protected abstract String readContent();
    protected abstract void processContent(String content);

    // Common operations with default implementations
    protected void openDocument() {
        System.out.println("Opening document");
    }

    protected void saveDocument() {
        System.out.println("Saving document");
    }

    protected void closeDocument() {
        System.out.println("Closing document");
    }
```

```
// Concrete implementation
class PDFProcessor extends DocumentProcessor {
    @Override
    protected String readContent() {
        return "PDF content";
    }

    @Override
    protected void processContent(String content) {
        System.out.println("Processing PDF content: " + content);
    }
}

class WordProcessor extends DocumentProcessor {
    @Override
    protected String readContent() {
        return "Word document content";
    }

    @Override
    protected void processContent(String content) {
        System.out.println("Processing Word content: " + content);
    }

    @Override
    protected void saveDocument() {
        System.out.println("Saving Word document with special format");
    }
}
```

Java Collections Framework

- **Iterator Pattern**: `java.util.Iterator`
- **Composite Pattern**: Component hierarchies in Swing
- **Strategy Pattern**: `java.util.Comparator`
- **Adapter Pattern**: `java.io.InputStreamReader`, `OutputStreamWriter`

Spring Framework

- **Factory Pattern**: `BeanFactory`
- **Singleton Pattern**: Default bean scope
- **Proxy Pattern**: AOP implementation
- **Template Method Pattern**: `JdbcTemplate`, `HibernateTemplate`

Android Development

- **Builder Pattern**: `AlertDialog.Builder`
- **Observer Pattern**: Event listeners

Combining Multiple Patterns

Model-View-Controller (MVC)

- **Observer Pattern**: Views observe the Model
- **Composite Pattern**: Views can contain sub-views
- **Strategy Pattern**: Different controllers implement different strategies
- **Factory Pattern**: Often used to create views or controllers

Example: E-commerce System

- **Factory/Builder**: Create product objects
- **Decorator**: Add features to products (warranty, gift wrapping)
- **Observer**: Notify users about price changes
- **Strategy**: Different payment methods
- **Command**: Order processing pipeline
- **Proxy**: Lazy loading of product images

Pattern Usage in Swing

- **Composite**: `JComponent` hierarchy
- **Decorator**: Visual decorations like borders
- **Observer**: Event listeners
- **Strategy**: Layout managers (BorderLayout, FlowLayout, etc.)
- **Factory Method**: UI element creation

Code Example: Observer Pattern in Swing

```
// Using the Observer pattern via ActionListener
JButton button = new JButton("Click Me");
button.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("Button clicked!");
    }
});

// Using the Composite pattern
JPanel panel = new JPanel();
```

Case Study: Java Stream API

Pattern Usage in Streams

- **Builder Pattern**: Stream creation and configuration
- **Iterator Pattern**: Underlying traversal mechanism
- **Strategy Pattern**: Different operations (map, filter, etc.)
- **Decorator Pattern**: Chaining operations

Code Example

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David");

// Using multiple patterns together
names.stream()
     .filter(name -> name.length() > 4)  // Strategy pattern
     .map(String::toUpperCase)           // Another strategy
     .sorted()                           // Yet another strategy
     .forEach(System.out::println);      // Terminal operation
```

Exercise: Implementing a Mini Document Management System

Requirements

1. Create a system that can handle different document types (PDF, Word, Text)
2. Support operations like opening, reading, editing, and saving documents
3. Implement access control for different user types
4. Enable document composition (documents can contain other documents)
5. Support undo/redo functionality for edits

Patterns to Apply

- **Factory**: For document creation
- **Strategy**: For different document processors
- **Composite**: For document composition
- **Command**: For operations with undo/redo
- **Proxy**: For access control

Exercise Solution Outline

```
// Factory pattern for document creation
class DocumentFactory {
    public Document createDocument(String type, String name) {
        switch (type.toLowerCase()) {
            case "pdf": return new PDFDocument(name);
            case "word": return new WordDocument(name);
            case "text": return new TextDocument(name);
            default: throw new IllegalArgumentException("Unknown document type");
        }
    }
}

// Composite pattern for document structure
interface Document {
    void open();
    void save();
    String getName();
}

class CompositeDocument implements Document {
    private String name;
    private List<Document> children = new ArrayList<>();

    // Implementation details...
}
```

References

- Freeman, E., Robson, E., Bates, B., Sierra, K. (2004). Head First Design Patterns. O'Reilly Media.
- Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.
- Bloch, J. (2018). Effective Java (3rd Edition). Addison-Wesley.
- Martin, R. C. (2002). Agile Software Development, Principles, Patterns, and Practices. Pearson.
- Refactoring Guru: https://refactoring.guru/design-patterns
- Java Design Patterns: https://java-design-patterns.com/
- Spring Framework Reference: https://docs.spring.io/spring-framework/reference/

Next Week

We'll explore UML (Unified Modeling Language) and UMPLE, focusing on modeling object-oriented systems and automatically generating code from these models.