Design Patterns

blibli backend training

November 25, 2025

What are Design Patterns?

Definition: Design patterns are reusable solutions to commonly occurring problems in software design.

Key Characteristics:

- **Not code** - they are templates or blueprints for solving problems
- **Language-independent** - applicable across different programming languages
- **Proven solutions** - refined through years of practice and experience
- **Named vocabulary** - provide a common language for developers

Origin: Popularized by the "Gang of Four" (GoF) book in 1994

- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
- Documented 23 classic design patterns

Why are Design Patterns Required?

1. **Code Reusability**
   - Avoid reinventing the wheel for common problems
   - Leverage proven solutions that have stood the test of time
2. **Maintainability & Scalability**
   - Well-structured code is easier to understand and modify
   - Patterns promote loose coupling and high cohesion
3. **Communication & Collaboration**
   - Shared vocabulary accelerates team discussions
   - "Use the Singleton pattern" vs. lengthy explanations
4. **Best Practices & Quality**
   - Encapsulate industry best practices
   - Reduce bugs through proven architectural approaches
5. **Flexibility & Extensibility**
   - Make systems more adaptable to change
   - Support new requirements without major refactoring

Agenda

Morning (9:30 AM - 12:30 PM)

- Singleton
- Proxy
- Factory Method
- **Assignment 1**

Afternoon 1 (1:30 PM - 3:30 PM)

- Command
- Observer
- Builder
- **Assignment 2**

Afternoon 2 (4:00 PM - 6:00 PM)

- Prototype
- Decorator
- Facade
- WebMVC
- **Assignment 3**

Table of Contents

What are Creational Patterns?

Definition: Creational patterns deal with object creation mechanisms, trying to create objects in a manner suitable to the situation.

Purpose:

- **Abstract the instantiation process** - hide how objects are created
- **Decouple client code** from concrete class implementations
- **Provide flexibility** in what gets created, who creates it, how it's created, and when
- **Control object creation** - ensure constraints and optimization

Key Benefits:

- Increase flexibility and reusability of code
- Eliminate the need to hard-code classes into the application
- Make the system independent of how objects are composed and represented

Patterns Covered Today: Singleton, Factory Method, Builder, Prototype

Singleton Pattern - Definition

Definition: Ensures a class has only one instance and provides a global point of access to it.

Real-World Use Case: Application Configuration Manager

A centralized configuration manager that loads settings from files/environment once and provides global access throughout the application. Creating multiple instances would waste memory and cause inconsistent configuration.

Key Characteristics:

- Single instance across application
- Global access point
- Lazy or eager initialization
- Thread-safe implementation

Singleton - Implementation (Part 1)

```java
package com.gdn.patterns.creational.singleton;

import java.util.HashMap;
import java.util.Map;

public class ConfigurationManager {
    // volatile ensures atomic visibility across threads
    private static volatile ConfigurationManager instance;
    private final Map<String, String> configurations;
    private final long loadedTimestamp;

    private ConfigurationManager() {
        // Simulate expensive initialization
        System.out.println("[EXPENSIVE] Loading configuration from files...");
        configurations = new HashMap<>();

        // Simulate loading config
        configurations.put("app.name", "Enterprise App");
        configurations.put("db.host", "localhost");
        configurations.put("db.port", "5432");
        configurations.put("cache.ttl", "3600");
        configurations.put("api.timeout", "30000");

        loadedTimestamp = System.currentTimeMillis();
```

Singleton - Implementation (Part 2)

```java
        try {
            Thread.sleep(500); // Simulate expensive I/O
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }

        System.out.println("$\checkmark$ Configuration loaded with " +
                         configurations.size() + " properties");
    }

    // Double-Checked Locking for thread-safe lazy initialization
    public static ConfigurationManager getInstance() {
        if (instance == null) { // First check (no locking)
            synchronized (ConfigurationManager.class) {
                if (instance == null) { // Second check (with locking)
                    instance = new ConfigurationManager();
                }
            }
        }
        return instance;
    }
```

Singleton - Implementation (Part 3)

```java
    public String getConfig(String key) {
        return configurations.getOrDefault(key, null);
    }

    public String getConfig(String key, String defaultValue) {
        return configurations.getOrDefault(key, defaultValue);
    }

    public void setConfig(String key, String value) {
        configurations.put(key, value);
        System.out.println("Config updated: " + key + " = " + value);
    }

    public int getConfigAsInt(String key, int defaultValue) {
        String value = configurations.get(key);
        try {
            return value != null ? Integer.parseInt(value) : defaultValue;
        } catch (NumberFormatException e) {
            return defaultValue;
        }
    }
```

Singleton - Implementation (Part 4)

```
    public void displayAllConfigs() {
        System.out.println("\n=== Application Configuration ===");
        configurations.forEach((key, value) ->
            System.out.println(key + " = " + value));
        System.out.println("Loaded at: " + new java.util.Date(loadedTimestamp));
    }
}
```

Singleton - Usage Example

```java
// Usage from different parts of application
class Application {
    public static void main(String[] args) {
        System.out.println("=== Application Starting ===\n");

        // Multiple components accessing the SAME configuration instance

        // Component 1: Web Server
        ConfigurationManager config1 = ConfigurationManager.getInstance();
        String appName = config1.getConfig("app.name");
        System.out.println("[WebServer] Starting " + appName);

        // Component 2: Database Service
        ConfigurationManager config2 = ConfigurationManager.getInstance();
        String dbHost = config2.getConfig("db.host");
        int dbPort = config2.getConfigAsInt("db.port", 5432);
        System.out.println("[Database] Connecting to " + dbHost + ":" + dbPort);
```

Singleton - Usage Example (cont.)

```
        // Component 3: Cache Service
        ConfigurationManager config3 = ConfigurationManager.getInstance();
        int cacheTTL = config3.getConfigAsInt("cache.ttl", 3600);
        System.out.println("[Cache] TTL set to " + cacheTTL + " seconds");

        // Verify all references point to the SAME instance
        System.out.println("\n=== Singleton Verification ===");
        System.out.println("config1 == config2: " + (config1 == config2));
        System.out.println("config2 == config3: " + (config2 == config3));
        System.out.println("All components share the SAME configuration instance!"
            );

        // Display all configurations
        config1.displayAllConfigs();
    }
}
```

Singleton - Why Use It?

Benefits:

- **Single Source of Truth**: All components access the same configuration
- **Resource Efficiency**: Expensive loading happens only once
- **Consistency**: No risk of conflicting configurations
- **Global Access**: Available anywhere in the application
- **Thread Safety**: Double-checked locking ensures safe concurrent access

Common Use Cases:

- Configuration managers
- Logger instances
- Database connection pools
- Cache managers
- Thread pools

What are Structural Patterns?

Definition: Structural patterns deal with object composition, creating relationships between objects to form larger, more complex structures.

Purpose:

- **Simplify structure** - organize relationships between entities
- **Compose objects and classes** to form larger structures
- **Add new functionality** without modifying existing code
- **Facilitate communication** between incompatible interfaces

Key Benefits:

- Identify simple ways to realize relationships between entities
- Increase flexibility in how objects can be composed
- Promote code reuse through inheritance and composition
- Simplify complex systems by hiding complexity

Patterns Covered Today: Proxy, Decorator, Facade

Proxy Pattern - Definition

Definition: Controls access to the original object, allowing for lazy loading or access control.

Real-World Use Case: Hibernate/JPA Lazy Loading

Loading heavy database entities (like Images) only when display() is actually called.

Types of Proxies:

- **Virtual Proxy**: Lazy initialization of expensive objects
- **Protection Proxy**: Access control
- **Remote Proxy**: Represents object in different address space
- **Smart Proxy**: Additional functionality (reference counting, caching)

Proxy - Implementation

```java
package com.gdn.patterns.structural.proxy;

interface Image {
    void display();
}

class RealImage implements Image {
    private String filename;

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

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

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

Proxy - Implementation (cont.)

```java
class ProxyImage implements Image {
    private RealImage realImage;
    private String filename;

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

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

Proxy - Usage Example

```java
// Client usage
public class ProxyDemo {
    public static void main(String[] args) {
        System.out.println("=== Proxy Pattern Demo ===\n");

        ProxyImage image = new ProxyImage("large_photo.jpg");
        System.out.println("Image proxy created (not loaded yet!)");

        // NOW it loads and displays
        System.out.println("\nFirst call to display():");
        image.display();

        // Already loaded, just displays
        System.out.println("\nSecond call to display():");
        image.display();

        System.out.println("\n$\checkmark$ Proxy pattern provides lazy loading!");
    }
}
```

Proxy - Why Use It?

Benefits:

- **Performance**: Delay expensive operations until needed
- **Security**: Control access to sensitive objects
- **Remote Access**: Hide network complexity
- **Smart Reference**: Add behavior without modifying original

Common Use Cases:

- ORM lazy loading (Hibernate, JPA)
- Image loading in GUI applications
- Network proxies for remote services
- Access control and security
- Caching proxies

Factory Method - Definition

Definition: Defines an interface for creating an object but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Real-World Use Case: Document Export System

An application needs to export documents to different formats (PDF, Word, Excel). The specific exporter is determined by user selection, and each exporter has its own complex initialization logic.

Key Characteristics:

- Delegates object creation to subclasses
- Provides abstraction over instantiation
- Follows Open/Closed Principle

Factory Method - Implementation (Part 1)

```java
package com.gdn.patterns.creational.factory;

import java.util.List;

// Product Interface
interface DocumentExporter {
    void export(String content, String filename);
    String getFormat();
}

// Concrete Products
class PDFExporter implements DocumentExporter {
    private boolean compressionEnabled;

    public PDFExporter() {
        this.compressionEnabled = true;
        System.out.println("[PDF] Initializing PDF libraries...");
    }

    @Override
    public void export(String content, String filename) {
        System.out.println("[PDF] Exporting to: " + filename + ".pdf");
        System.out.println("[PDF] Content length: " + content.length() + " chars");
        System.out.println("[PDF] Compression: " + (compressionEnabled ? "ON" : "OFF")
            );
        System.out.println("[PDF] $\checkmark$ Export complete\n");
    }

    @Override
    public String getFormat() { return "PDF"; }
}
```

Factory Method - Implementation (Part 2)

```java
class WordExporter implements DocumentExporter {
    private String version;

    public WordExporter() {
        this.version = "DOCX";
        System.out.println("[WORD] Initializing Word XML processor...");
    }

    @Override
    public void export(String content, String filename) {
        System.out.println("[WORD] Exporting to: " + filename + ".docx");
        System.out.println("[WORD] Format: " + version);
        System.out.println("[WORD] Applying styles and formatting...");
        System.out.println("[WORD] $\checkmark$ Export complete\n");
    }

    @Override
    public String getFormat() { return "WORD"; }
}
```

Factory Method - Implementation (Part 3)

```java
class ExcelExporter implements DocumentExporter {
    private int maxColumns;

    public ExcelExporter() {
        this.maxColumns = 256;
        System.out.println("[EXCEL] Initializing spreadsheet engine...");
    }

    @Override
    public void export(String content, String filename) {
        System.out.println("[EXCEL] Exporting to: " + filename + ".xlsx");
        System.out.println("[EXCEL] Max columns: " + maxColumns);
        System.out.println("[EXCEL] Converting to table format...");
        System.out.println("[EXCEL] $\checkmark$ Export complete\n");
    }

    @Override
    public String getFormat() { return "EXCEL"; }
}
```

Factory Method - Factory Implementation

```java
// Creator / Factory
class ExporterFactory {
    public DocumentExporter createExporter(String format) {
        return switch (format.toUpperCase()) {
            case "PDF" -> new PDFExporter();
            case "WORD", "DOCX" -> new WordExporter();
            case "EXCEL", "XLSX" -> new ExcelExporter();
            default -> throw new IllegalArgumentException(
                "Unsupported format: " + format +
                ". Supported: PDF, WORD, EXCEL");
        };
    }

    public List<String> getSupportedFormats() {
        return List.of("PDF", "WORD", "EXCEL");
    }
}
```

Factory Method - Usage Example

```java
// Client Code
class DocumentExportService {
    private ExporterFactory factory = new ExporterFactory();

    public void exportDocument(String content, String filename, String format) {
        try {
            System.out.println("=== Starting Export Process ===");
            System.out.println("Format requested: " + format + "\n");

            // Factory creates the appropriate exporter
            DocumentExporter exporter = factory.createExporter(format);

            // Use the exporter (client doesn't know concrete type)
            exporter.export(content, filename);

        } catch (IllegalArgumentException e) {
            System.err.println("ERROR: " + e.getMessage());
            System.out.println("Available formats: " + factory.getSupportedFormats
                 ());
        }
    }
```

Factory Method - Usage Example (cont.)

```java
    public static void main(String[] args) {
        DocumentExportService service = new DocumentExportService();
        String content = "This is a sample document with important content.";

        // Export to different formats
        service.exportDocument(content, "report_2024", "PDF");
        service.exportDocument(content, "report_2024", "WORD");
        service.exportDocument(content, "report_2024", "EXCEL");

        // Try unsupported format
        service.exportDocument(content, "report_2024", "HTML");
    }
}
```

Factory Method - Why Use It?

Benefits:

- **Extensibility**: Easy to add new export formats without modifying existing code
- **Encapsulation**: Complex initialization logic hidden in concrete classes
- **Loose Coupling**: Client code doesn't depend on concrete exporter classes
- **Single Responsibility**: Each exporter handles its own format
- **Open/Closed Principle**: Open for extension, closed for modification

Common Use Cases:

- Framework creation (e.g., UI components)
- Plugin systems
- Document/Report generators
- Database connection factories

Assignment 1: Configuration Management System

Scenario: You are building a configuration management system for a microservices application.

Requirements:

1. **Singleton Pattern**: Implement a `ConfigurationManager` class that:
    - Uses thread-safe Singleton pattern (Double-Checked Locking)
    - Stores configuration as key-value pairs in a HashMap
    - Provides `getConfig(String key)` and `setConfig(String key, String value)` methods
    - Has a method `loadFromSource(ConfigSource source)` to load configurations

Assignment 1: Configuration Management System (cont.)

2. **Factory Pattern**: Implement a `ConfigSourceFactory` that:
   - Creates different types of configuration sources: `JsonConfigSource`, `YamlConfigSource`, `PropertiesConfigSource`
   - Each source should implement a `ConfigSource` interface with a `readConfig()` method
   - Factory should have a `createSource(String type, String filePath)` method

Deliverables:

- ConfigurationManager class (Singleton)
- ConfigSource interface
- Three concrete implementations
- ConfigSourceFactory class
- Demo showing thread safety and loading from different sources

Assignment 1: Bonus Challenges

Bonus Challenges:

- Add a `refreshConfig()` method that reloads configuration from the source
- Implement a caching mechanism in the factory to reuse ConfigSource instances

Time: 30 minutes

Tip: Start with the interfaces, then implement concrete classes, and finally test with a main class.

What are Behavioral Patterns?

Definition: Behavioral patterns focus on communication between objects, defining how objects interact and distribute responsibilities.

Purpose:

- **Define communication patterns** between objects
- **Assign responsibilities** - who does what and when
- **Describe algorithms and control flow** in object-oriented systems
- **Encapsulate behavior** - separate what varies from what stays the same

Key Benefits:

- Increase flexibility in carrying out communication
- Reduce coupling between objects
- Make algorithms and responsibilities more dynamic
- Enable objects to cooperate without being tightly bound

Patterns Covered Today: Command, Observer

Command Pattern - Definition

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

Real-World Use Case: Text Editor with Undo/Redo

A text editor needs to support operations like typing, deleting, formatting - each operation must be undoable. Commands encapsulate these operations and maintain state needed for undo.

Key Components:

- Command interface
- Concrete commands
- Receiver (performs actual work)
- Invoker (executes commands)

Command - Implementation (Part 1)

```java
package com.gdn.patterns.behavioral.command;

import java.util.Stack;

// Command Interface with undo support
interface Command {
    void execute();
    void undo();
    String getDescription();
}

// Receiver - Text Document
class TextDocument {
    private StringBuilder content = new StringBuilder();

    public void insertText(int position, String text) {
        content.insert(position, text);
        System.out.println("  Inserted: '" + text + "' at position " + position);
    }

    public void deleteText(int position, int length) {
        String deleted = content.substring(position, position + length);
        content.delete(position, position + length);
        System.out.println("  Deleted: '" + deleted + "' from position " +
             position);
    }
```

Command - Implementation (Part 2)

```java
    public String getContent() {
        return content.toString();
    }

    public void display() {
        System.out.println("Document: [" + content.toString() + "]");
    }
}

// Concrete Commands
class InsertTextCommand implements Command {
    private TextDocument document;
    private String text;
    private int position;

    public InsertTextCommand(TextDocument doc, int position, String text) {
        this.document = doc;
        this.position = position;
        this.text = text;
    }

    @Override
    public void execute() {
        document.insertText(position, text);
    }
```

Command - Implementation (Part 3)

```java
    @Override
    public void undo() {
        // Undo insert by deleting what was inserted
        document.deleteText(position, text.length());
    }

    @Override
    public String getDescription() {
        return "Insert '" + text + "'";
    }
}

class DeleteTextCommand implements Command {
    private TextDocument document;
    private int position;
    private int length;
    private String deletedText; // Store for undo

    public DeleteTextCommand(TextDocument doc, int position, int length) {
        this.document = doc;
        this.position = position;
        this.length = length;
    }
```

Command - Implementation (Part 4)

```java
    @Override
    public void execute() {
        // Save text before deleting (for undo)
        deletedText = document.getContent().substring(position, position + length)
            ;
        document.deleteText(position, length);
    }

    @Override
    public void undo() {
        // Restore deleted text
        document.insertText(position, deletedText);
    }

    @Override
    public String getDescription() {
        return "Delete " + length + " chars";
    }
}
```

Command - Invoker Implementation (Part 1)

```java
// Invoker - Command Manager with Undo/Redo
class TextEditor {
    private Stack<Command> undoStack = new Stack<>();
    private Stack<Command> redoStack = new Stack<>();
    private TextDocument document;

    public TextEditor() {
        this.document = new TextDocument();
    }

    public void executeCommand(Command command) {
        System.out.println("\nExecuting: " + command.getDescription());
        command.execute();
        undoStack.push(command);
        redoStack.clear(); // Clear redo stack on new command
        document.display();
    }
```

Command - Invoker Implementation (Part 2)

```java
    public void undo() {
        if (undoStack.isEmpty()) {
            System.out.println("\nNothing to undo!");
            return;
        }

        Command command = undoStack.pop();
        System.out.println("\nUndo: " + command.getDescription());
        command.undo();
        redoStack.push(command);
        document.display();
    }

    public void redo() {
        if (redoStack.isEmpty()) {
            System.out.println("\nNothing to redo!");
            return;
        }

        Command command = redoStack.pop();
        System.out.println("\nRedo: " + command.getDescription());
        command.execute();
        undoStack.push(command);
        document.display();
    }
```

Command - Usage Example

```java
    public static void main(String[] args) {
        TextEditor editor = new TextEditor();

        System.out.println("========== Text Editor Demo ==========");

        // Type some text
        editor.executeCommand(new InsertTextCommand(editor.document, 0, "Hello"));
        editor.executeCommand(new InsertTextCommand(editor.document, 5, " World"))
            ;
        editor.executeCommand(new InsertTextCommand(editor.document, 11, "!"));

        // Undo last operation
        editor.undo(); // Remove "!"

        // Redo
        editor.redo(); // Add "!" back

        // Delete some text
        editor.executeCommand(new DeleteTextCommand(editor.document, 5, 6));

        System.out.println("\n$\checkmark$ All operations are encapsulated as
            commands!");
        System.out.println("$\checkmark$ Full undo/redo support!");
    }
}
```

Command - Why Use It?

Benefits:

- **Undo/Redo**: Each command stores state needed to reverse operation
- **Parameterization**: Commands are objects that can be stored, queued, logged
- **Macro Commands**: Can create composite commands (multiple ops as one)
- **Loose Coupling**: Invoker doesn't know what operation does
- **Audit Trail**: Command history provides full operation log

Common Use Cases:

- Text editors (undo/redo)
- Transaction systems
- Task scheduling and queuing
- GUI button actions
- Macro recording

Observer Pattern - Definition

Definition: Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Real-World Use Case: Weather Monitoring System

A weather station (Subject) measures temperature, humidity, and pressure. Multiple displays (Observers) - CurrentConditions, Statistics, Forecast - need to update when measurements change. New displays can be added without modifying the weather station.

Key Components:

- Subject interface (publisher)
- Observer interface (subscriber)
- Concrete Subject
- Concrete Observers

Observer - Implementation (Part 1)

```java
package com.gdn.patterns.behavioral.observer;

import java.util.ArrayList;
import java.util.List;

// Subject Interface
interface Subject {
    void registerObserver(Observer o);
    void removeObserver(Observer o);
    void notifyObservers();
}

// Observer Interface
interface Observer {
    void update(float temperature, float humidity, float pressure);
}

// Display Interface (for observer implementations)
interface DisplayElement {
    void display();
}
```

Observer - Concrete Subject

```java
// Concrete Subject - Weather Station
class WeatherStation implements Subject {
    private List<Observer> observers;
    private float temperature;
    private float humidity;
    private float pressure;

    public WeatherStation() {
        observers = new ArrayList<>();
    }

    @Override
    public void registerObserver(Observer o) {
        observers.add(o);
        System.out.println("  [WeatherStation] Observer registered. Total: " +
                          observers.size());
    }

    @Override
    public void removeObserver(Observer o) {
        observers.remove(o);
        System.out.println("  [WeatherStation] Observer removed. Total: " +
                          observers.size());
    }
```

Observer - Concrete Subject (cont.)

```
    @Override
    public void notifyObservers() {
        System.out.println("\n[WeatherStation] Notifying " +
                           observers.size() + " observers...");
        for (Observer observer : observers) {
            observer.update(temperature, humidity, pressure);
        }
    }

    public void setMeasurements(float temperature, float humidity, float pressure)
        {
        System.out.println("\n========== New Measurements ==========");
        System.out.println("Temperature: " + temperature + "$^\circ$F");
        System.out.println("Humidity: " + humidity + "%");
        System.out.println("Pressure: " + pressure + " inHg");

        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;

        notifyObservers();
    }
}
```

Observer - Concrete Observers (Part 1)

```
// Concrete Observer 1 - Current Conditions Display
class CurrentConditionsDisplay implements Observer, DisplayElement {
    private float temperature;
    private float humidity;
    private Subject weatherStation;
    private String name;

    public CurrentConditionsDisplay(Subject weatherStation, String name) {
        this.weatherStation = weatherStation;
        this.name = name;
        weatherStation.registerObserver(this);
    }

    @Override
    public void update(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        display();
    }

    @Override
    public void display() {
        System.out.println("  [" + name + "] Current: " + temperature +
                          "$^\circ$F and " + humidity + "% humidity");
    }
}
```

Observer - Concrete Observers (Part 2)

```java
// Concrete Observer 2 - Statistics Display
class StatisticsDisplay implements Observer, DisplayElement {
    private float maxTemp = Float.MIN_VALUE;
    private float minTemp = Float.MAX_VALUE;
    private float tempSum = 0.0f;
    private int numReadings = 0;

    public StatisticsDisplay(Subject weatherStation) {
        weatherStation.registerObserver(this);
    }

    @Override
    public void update(float temperature, float humidity, float pressure) {
        tempSum += temperature;
        numReadings++;

        if (temperature > maxTemp) {
            maxTemp = temperature;
        }
        if (temperature < minTemp) {
            minTemp = temperature;
        }

        display();
    }
```

Observer - Concrete Observers (Part 3)

```
    @Override
    public void display() {
        System.out.println("  [Statistics] Avg/Max/Min: " +
                          (tempSum / numReadings) + "$^\circ$F / " +
                          maxTemp + "$^\circ$F / " + minTemp + "$^\circ$F");
    }
}
```

Observer - Concrete Observers (Part 4)

```java
// Concrete Observer 3 - Forecast Display
class ForecastDisplay implements Observer, DisplayElement {
    private float currentPressure = 29.92f;
    private float lastPressure;

    public ForecastDisplay(Subject weatherStation) {
        weatherStation.registerObserver(this);
    }

    @Override
    public void update(float temperature, float humidity, float pressure) {
        lastPressure = currentPressure;
        currentPressure = pressure;
        display();
    }

    @Override
    public void display() {
        System.out.print("  [Forecast] ");
        if (currentPressure > lastPressure) {
            System.out.println("Improving weather on the way!");
        } else if (currentPressure == lastPressure) {
            System.out.println("More of the same");
        } else {
            System.out.println("Watch out for cooler, rainy weather");
        }
    }
}
```

Observer - Usage Example

```java
// Demo
class WeatherMonitoringApp {
    public static void main(String[] args) {
        System.out.println("========== Weather Monitoring System ==========\n");

        // Create subject
        WeatherStation weatherStation = new WeatherStation();

        // Create and register observers
        System.out.println("=== Registering Observers ===");
        CurrentConditionsDisplay currentDisplay =
            new CurrentConditionsDisplay(weatherStation, "Living Room");
        StatisticsDisplay statsDisplay = new StatisticsDisplay(weatherStation);
        ForecastDisplay forecastDisplay = new ForecastDisplay(weatherStation);

        // Simulate weather measurements
        weatherStation.setMeasurements(80, 65, 30.4f);
        weatherStation.setMeasurements(82, 70, 29.2f);
        weatherStation.setMeasurements(78, 90, 29.2f);

        System.out.println("\n$\checkmark$ Observers automatically notified on
             state changes!");
    }
}
```

Observer - Why Use It?

Benefits:

- **Loose Coupling**: Subject knows nothing about concrete observers
- **Dynamic Relationships**: Add/remove observers at runtime
- **Broadcast Communication**: One change notifies all subscribers
- **Extensibility**: New observer types without changing subject
- **Event-Driven**: Foundation for reactive programming patterns

Common Use Cases:

- Event handling systems
- Model-View-Controller (MVC)
- Publish-subscribe messaging
- Data binding in UI frameworks
- Monitoring and alerting systems

Builder Pattern - Definition

Definition: Separates the construction of a complex object from its representation, allowing the same construction process to create different representations.

Real-World Use Case: SQL Query Builder

Building complex SQL queries with multiple optional clauses (WHERE, JOIN, GROUP BY, ORDER BY, LIMIT) without creating telescoping constructors or dealing with numerous null parameters.

Key Characteristics:

- Fluent interface (method chaining)
- Immutable result object
- Optional parameters
- Validation before construction

Builder - Implementation (Part 1)

```java
package com.gdn.patterns.creational.builder;

import java.util.ArrayList;
import java.util.List;

// Complex Product
public class SQLQuery {
    private final String table;
    private final List<String> columns;
    private final List<String> whereClauses;
    private final List<String> joins;
    private final String groupBy;
    private final String orderBy;
    private final Integer limit;
    private final boolean distinct;

    private SQLQuery(Builder builder) {
        this.table = builder.table;
        this.columns = builder.columns;
        this.whereClauses = builder.whereClauses;
        this.joins = builder.joins;
        this.groupBy = builder.groupBy;
        this.orderBy = builder.orderBy;
        this.limit = builder.limit;
        this.distinct = builder.distinct;
    }
```

Builder - Implementation (Part 2)

```java
    public String toSQL() {
        StringBuilder sql = new StringBuilder("SELECT ");

        if (distinct) sql.append("DISTINCT ");

        sql.append(columns.isEmpty() ? "*" : String.join(", ", columns));
        sql.append(" FROM ").append(table);

        for (String join : joins) {
            sql.append(" ").append(join);
        }

        if (!whereClauses.isEmpty()) {
            sql.append(" WHERE ").append(String.join(" AND ", whereClauses));
        }

        if (groupBy != null) {
            sql.append(" GROUP BY ").append(groupBy);
        }

        if (orderBy != null) {
            sql.append(" ORDER BY ").append(orderBy);
        }

        if (limit != null) {
            sql.append(" LIMIT ").append(limit);
        }

        return sql.toString();
    }
```

Builder - Builder Class (Part 1)

```
    // Builder Inner Class
    public static class Builder {
        // Required parameters
        private final String table;

        // Optional parameters - initialized to default values
        private List<String> columns = new ArrayList<>();
        private List<String> whereClauses = new ArrayList<>();
        private List<String> joins = new ArrayList<>();
        private String groupBy;
        private String orderBy;
        private Integer limit;
        private boolean distinct = false;

        // Constructor with required parameters
        public Builder(String table) {
            this.table = table;
        }

        public Builder select(String... columns) {
            for (String col : columns) {
                this.columns.add(col);
            }
            return this;
        }
```

Builder - Builder Class (Part 2)

```java
        public Builder where(String condition) {
            this.whereClauses.add(condition);
            return this;
        }

        public Builder join(String joinClause) {
            this.joins.add(joinClause);
            return this;
        }

        public Builder groupBy(String groupBy) {
            this.groupBy = groupBy;
            return this;
        }

        public Builder orderBy(String orderBy) {
            this.orderBy = orderBy;
            return this;
        }

        public Builder limit(int limit) {
            this.limit = limit;
            return this;
        }
```

Builder - Builder Class (Part 3)

```
        public Builder distinct() {
            this.distinct = true;
            return this;
        }

        public SQLQuery build() {
            // Validation can be added here
            if (table == null || table.isEmpty()) {
                throw new IllegalStateException("Table name is required");
            }
            return new SQLQuery(this);
        }
    }
}
```

Builder - Usage Example

```java
// Usage Examples
public class QueryBuilderDemo {
    public static void main(String[] args) {
        System.out.println("========== SQL Query Builder Demo ==========\n");

        // Simple query
        SQLQuery query1 = new SQLQuery.Builder("users")
                .select("id", "name", "email")
                .where("age > 18")
                .orderBy("name ASC")
                .limit(10)
                .build();

        System.out.println("Query 1:");
        System.out.println(query1.toSQL());
        // Output: SELECT id, name, email FROM users
        //         WHERE age > 18 ORDER BY name ASC LIMIT 10

        // Complex query with joins
        SQLQuery query2 = new SQLQuery.Builder("orders")
                .distinct()
                .select("orders.id", "users.name", "products.title")
                .join("INNER JOIN users ON orders.user_id = users.id")
                .join("INNER JOIN products ON orders.product_id = products.id")
                .where("orders.status = 'completed'")
                .where("orders.total > 100")
                .groupBy("users.id")
                .orderBy("orders.created_at DESC")
                .limit(50)
                .build();

        System.out.println("\nQuery 2:");
        System.out.println(query2.toSQL());

        // Simple query with all defaults
        SQLQuery query3 = new SQLQuery.Builder("products")
                .build();

        System.out.println("\nQuery 3 (defaults):");
        System.out.println(query3.toSQL());

        System.out.println("\n$\checkmark$ Builder pattern provides fluent, readable
            API!");
        System.out.println("$\checkmark$ Optional parameters handled elegantly!");
    }
}
```

Builder - Why Use It?

Benefits:

- **Readability**: Method chaining makes code self-documenting
- **Flexibility**: Add any combination of optional clauses
- **Immutability**: Final fields prevent modification after construction
- **Validation**: Can validate in build() method before object creation
- **No Telescoping Constructors**: Avoids constructors with many parameters

Common Use Cases:

- Complex object construction (HTTP requests, SQL queries)
- Configuration objects
- Test data builders
- Fluent APIs

Assignment 2: Smart Home Automation System

Scenario: You are building a smart home automation system that controls various devices and notifies users of events.

Part 1 - Command Pattern: Implement device controls

- Create a `SmartDevice` interface with methods like `turnOn()`, `turnOff()`, `getStatus()`
- Implement concrete devices: `SmartLight`, `SmartThermostat`, `SmartDoorLock`
- Create `Command` interface and concrete commands: `TurnOnCommand`, `TurnOffCommand`, `SetTemperatureCommand`
- Implement a `RemoteControl` class (Invoker) that executes commands
- **Bonus**: Add undo functionality to reverse the last command

Assignment 2: Smart Home Automation System (cont.)

Part 2 - Observer Pattern: Implement event notifications

- Create a `DeviceEventPublisher` (Subject) that tracks device state changes
- Implement observers: `MobileAppNotifier`, `EmailNotifier`, `LoggingService`
- When any device state changes, all registered observers should be notified
- Each observer should display the notification in its own way

Part 3 - Builder Pattern: Implement automation rule creation

- Create an `AutomationRule` class with: `ruleName`, `triggerDevice`, `triggerCondition`, `actionCommand`, `schedule`, `enabled`
- Implement a `AutomationRule.Builder` for constructing complex rules
- Support method chaining

Assignment 2: Deliverables

Deliverables:

- All device classes implementing SmartDevice interface
- Command pattern implementation with at least 3 commands
- Observer pattern with DeviceEventPublisher and 3 observers
- AutomationRule class with Builder
- A demo class showing:
  - Creating devices and executing commands
  - Registering observers and triggering notifications
  - Building complex automation rules using the Builder
  - Executing an automation rule that triggers commands and notifies observers

Time: 45 minutes

Prototype Pattern - Definition

Definition: Creates new objects by copying an existing object, known as a prototype.

Real-World Use Case: Game Character Creation

In a game, instead of manually configuring each enemy character from scratch (setting health, armor, weapons, skills), we clone a pre-configured prototype and customize specific attributes. This is especially useful when creating hundreds of similar objects with slight variations.

Key Characteristics:

- Cloning instead of constructing
- Prototype registry for managing templates
- Performance optimization for expensive objects
- Customization after cloning

Prototype - Implementation (Part 1)

```java
package com.gdn.patterns.creational.prototype;

import java.util.HashMap;
import java.util.Map;

// Prototype Interface
interface GameCharacter extends Cloneable {
    GameCharacter clone();
    void display();
}

// Concrete Prototype
class Enemy implements GameCharacter {
    private String name;
    private int health;
    private int attackPower;
    private String weaponType;
    private int level;

    public Enemy(String name, int health, int attackPower,
                 String weaponType, int level) {
        this.name = name;
        this.health = health;
        this.attackPower = attackPower;
        this.weaponType = weaponType;
        this.level = level;
        // Simulate expensive initialization
        System.out.println("[EXPENSIVE] Loading 3D model and AI for: " + name);
    }
```

Prototype - Implementation (Part 2)

```
    // Clone method - cheap operation, no expensive initialization
    @Override
    public GameCharacter clone() {
        try {
            System.out.println("[FAST] Cloning " + this.name);
            return (Enemy) super.clone();
        } catch (CloneNotSupportedException e) {
            return null;
        }
    }

    // Setters for customization after cloning
    public void setName(String name) { this.name = name; }
    public void setLevel(int level) {
        this.level = level;
        this.health += (level * 10); // Scale with level
        this.attackPower += (level * 2);
    }

    @Override
    public void display() {
        System.out.println("Enemy: " + name + " | HP: " + health +
                          " | Attack: " + attackPower +
                          " | Weapon: " + weaponType +
                          " | Level: " + level);
    }
}
```

Prototype - Registry Implementation

```java
// Prototype Registry - Manages pre-configured prototypes
class CharacterRegistry {
    private Map<String, GameCharacter> prototypes = new HashMap<>();

    public CharacterRegistry() {
        // Pre-load prototypes (expensive operation done once)
        prototypes.put("GOBLIN", new Enemy("Goblin Scout", 100, 15, "Dagger", 1));
        prototypes.put("ORC", new Enemy("Orc Warrior", 200, 30, "Axe", 1));
        prototypes.put("DRAGON", new Enemy("Dragon", 500, 100, "Fire Breath", 1));
    }

    public GameCharacter getPrototype(String type) {
        return prototypes.get(type).clone(); // Return a clone, not original
    }
}
```

Prototype - Usage Example

```java
// Client Code
public class GameWorld {
    public static void main(String[] args) {
        System.out.println("========== Prototype Pattern Demo ==========\n");
        System.out.println("=== Initializing Character Registry ===");
        CharacterRegistry registry = new CharacterRegistry();

        System.out.println("\n=== Creating Level 1 Enemies ===");
        GameCharacter goblin1 = registry.getPrototype("GOBLIN");
        ((Enemy) goblin1).setName("Goblin Scout #1");
        goblin1.display();

        GameCharacter goblin2 = registry.getPrototype("GOBLIN");
        ((Enemy) goblin2).setName("Goblin Scout #2");
        goblin2.display();
```

Prototype - Usage Example (cont.)

```
        System.out.println("\n=== Creating Boss Enemies (Higher Level) ===");
        GameCharacter orcBoss = registry.getPrototype("ORC");
        ((Enemy) orcBoss).setName("Orc Chieftain");
        ((Enemy) orcBoss).setLevel(5);
        orcBoss.display();

        GameCharacter dragon = registry.getPrototype("DRAGON");
        dragon.display();

        System.out.println("\n$\checkmark$ Created 4 enemies with only 3 expensive
              initializations!");
        System.out.println("$\checkmark$ Cloning is much faster than creating from
              scratch!");
    }
}
```

Prototype - Why Use It?

Benefits:

- **Performance**: Loading 3D models and AI is expensive - done once per type
- **Flexibility**: Easy to create variations of the same enemy type
- **Memory Efficient**: Share immutable data, clone only what's needed
- **Runtime Configuration**: Can add new prototypes dynamically

Common Use Cases:

- Game object creation (characters, items)
- Document templates
- Database record copying
- Complex object initialization

Decorator Pattern - Definition

Definition: Adds new functionality to an existing object dynamically without altering its structure. Decorators provide a flexible alternative to subclassing for extending functionality.

Real-World Use Case: Coffee Shop Ordering System

A base coffee can be decorated with various add-ons (milk, sugar, whipped cream, caramel) where each add-on modifies the cost and description. Decorators can be stacked in any combination without creating dozens of subclasses.

Key Characteristics:

- Wraps original object
- Adds behavior dynamically
- Transparent to client
- Stackable decorators

Decorator - Implementation (Part 1)

```java
package com.gdn.patterns.structural.decorator;

// Component Interface
interface Coffee {
    String getDescription();
    double getCost();
}

// Concrete Component - Base coffee
class SimpleCoffee implements Coffee {
    @Override
    public String getDescription() {
        return "Simple Coffee";
    }

    @Override
    public double getCost() {
        return 2.00;
    }
}

class Espresso implements Coffee {
    @Override
    public String getDescription() {
        return "Espresso";
    }

    @Override
    public double getCost() {
        return 2.50;
    }
}
```

Decorator - Abstract Decorator

```java
// Abstract Decorator
abstract class CoffeeDecorator implements Coffee {
    protected Coffee decoratedCoffee;

    public CoffeeDecorator(Coffee coffee) {
        this.decoratedCoffee = coffee;
    }

    @Override
    public String getDescription() {
        return decoratedCoffee.getDescription();
    }

    @Override
    public double getCost() {
        return decoratedCoffee.getCost();
    }
}
```

Decorator - Concrete Decorators (Part 1)

```java
// Concrete Decorators
class MilkDecorator extends CoffeeDecorator {
    public MilkDecorator(Coffee coffee) {
        super(coffee);
    }

    @Override
    public String getDescription() {
        return decoratedCoffee.getDescription() + ", Milk";
    }

    @Override
    public double getCost() {
        return decoratedCoffee.getCost() + 0.50;
    }
}

class SugarDecorator extends CoffeeDecorator {
    private int cubes;

    public SugarDecorator(Coffee coffee, int cubes) {
        super(coffee);
        this.cubes = cubes;
    }

    @Override
    public String getDescription() {
        return decoratedCoffee.getDescription() + ", Sugar(x" + cubes + ")";
    }

    @Override
    public double getCost() {
        return decoratedCoffee.getCost() + (0.20 * cubes);
    }
}
```

Decorator - Concrete Decorators (Part 2)

```java
class WhippedCreamDecorator extends CoffeeDecorator {
    public WhippedCreamDecorator(Coffee coffee) {
        super(coffee);
    }

    @Override
    public String getDescription() {
        return decoratedCoffee.getDescription() + ", Whipped Cream";
    }

    @Override
    public double getCost() {
        return decoratedCoffee.getCost() + 0.70;
    }
}

class CaramelDecorator extends CoffeeDecorator {
    public CaramelDecorator(Coffee coffee) {
        super(coffee);
    }

    @Override
    public String getDescription() {
        return decoratedCoffee.getDescription() + ", Caramel Syrup";
    }

    @Override
    public double getCost() {
        return decoratedCoffee.getCost() + 0.60;
    }
}
```

Decorator - Usage Example

```java
// Client Code
class CoffeeShop {
    public static void printOrder(Coffee coffee) {
        System.out.println("Order: " + coffee.getDescription());
        System.out.println("Cost: $" + String.format("%.2f", coffee.getCost()));
        System.out.println();
    }

    public static void main(String[] args) {
        System.out.println("========== Coffee Shop Orders ==========\n");

        // Order 1: Simple coffee
        Coffee order1 = new SimpleCoffee();
        printOrder(order1);

        // Order 2: Coffee with milk
        Coffee order2 = new SimpleCoffee();
        order2 = new MilkDecorator(order2);
        printOrder(order2);

        // Order 3: Coffee with milk and sugar (stacking decorators!)
        Coffee order3 = new SimpleCoffee();
        order3 = new MilkDecorator(order3);
        order3 = new SugarDecorator(order3, 2); // 2 cubes
        printOrder(order3);
```

Decorator - Usage Example (cont.)

```java
        // Order 4: Fancy espresso with everything!
        Coffee order4 = new Espresso();
        order4 = new MilkDecorator(order4);
        order4 = new WhippedCreamDecorator(order4);
        order4 = new CaramelDecorator(order4);
        printOrder(order4);

        // Order 5: Double milk latte
        Coffee order5 = new SimpleCoffee();
        order5 = new MilkDecorator(order5);
        order5 = new MilkDecorator(order5); // Extra milk!
        order5 = new SugarDecorator(order5, 1);
        printOrder(order5);

        System.out.println("$\checkmark$ Each order dynamically composed at
             runtime!");
        System.out.println("$\checkmark$ No need for dozens of subclasses!");
    }
}
```

Decorator - Why Use It?

Benefits:

- **Runtime Flexibility**: Add/remove features dynamically
- **Avoid Class Explosion**: Don't need MilkSugarCoffee, MilkCaramelCoffee, etc.
- **Single Responsibility**: Each decorator adds one specific feature
- **Open/Closed Principle**: Extend behavior without modifying existing code
- **Unlimited Combinations**: Stack decorators in any order and quantity

Common Use Cases:

- Java I/O streams (BufferedReader, InputStreamReader)
- GUI component enhancement
- Middleware in web frameworks
- Data processing pipelines

Facade Pattern - Definition

Definition: Provides a simplified interface to a complex subsystem, hiding the complexity and making it easier to use.

Real-World Use Case: E-Commerce Order Processing

When a customer places an order, the system needs to coordinate between inventory, payment, shipping, notification, and loyalty subsystems. A Facade provides a single `placeOrder()` method that orchestrates all these complex operations.

Key Characteristics:

- Unified interface to subsystems
- Simplifies client code
- Decouples client from subsystems
- Can provide additional logic (error handling, rollback)

Facade - Complex Subsystems (Part 1)

```java
package com.gdn.patterns.structural.facade;

// ========== Complex Subsystems ==========

class InventoryService {
    public boolean checkStock(String productId, int quantity) {
        System.out.println("  [Inventory] Checking stock for product: " +
             productId);
        System.out.println("  [Inventory] Quantity requested: " + quantity);
        boolean inStock = Math.random() > 0.1; // 90% success rate
        System.out.println("  [Inventory] " + (inStock ? "$\checkmark$ In stock" :
              "$\times$ Out of stock"));
        return inStock;
    }

    public void reserveStock(String productId, int quantity) {
        System.out.println("  [Inventory] Reserved " + quantity + " units of " +
             productId);
    }

    public void releaseStock(String productId, int quantity) {
        System.out.println("  [Inventory] Released " + quantity + " units of " +
             productId);
    }
}
```

Facade - Complex Subsystems (Part 2)

```java
class PaymentService {
    public boolean processPayment(String customerId, double amount) {
        System.out.println("  [Payment] Processing payment for customer: " +
            customerId);
        System.out.println("  [Payment] Amount: $" + amount);
        try {
            Thread.sleep(100); // Simulate API call
            boolean success = Math.random() > 0.05; // 95% success rate
            System.out.println("  [Payment] " + (success ? "$\checkmark$ Payment
                successful" : "$\times$ Payment failed"));
            return success;
        } catch (InterruptedException e) {
            return false;
        }
    }

    public void refund(String customerId, double amount) {
        System.out.println("  [Payment] Refunding $" + amount + " to " +
            customerId);
    }
}
```

Facade - Complex Subsystems (Part 3)

```java
class ShippingService {
    public String scheduleDelivery(String customerId, String address) {
        System.out.println("  [Shipping] Scheduling delivery to: " + address);
        String trackingNumber = "TRACK-" + System.currentTimeMillis();
        System.out.println("  [Shipping] $\checkmark$ Tracking number: " + trackingNumber);
        return trackingNumber;
    }

    public void cancelDelivery(String trackingNumber) {
        System.out.println("  [Shipping] Cancelled delivery: " + trackingNumber);
    }
}

class NotificationService {
    public void sendOrderConfirmation(String customerId, String orderId) {
        System.out.println("  [Notification] Sending order confirmation email");
        System.out.println("  [Notification] $\checkmark$ Email sent to customer: " + customerId);
    }

    public void sendFailureNotification(String customerId, String reason) {
        System.out.println("  [Notification] Sending failure notification: " + reason);
    }
}
```

Facade - Order Processing Facade (Part 1)

```java
// ========== FACADE ==========
class OrderProcessingFacade {
    // Complex subsystems
    private InventoryService inventory = new InventoryService();
    private PaymentService payment = new PaymentService();
    private ShippingService shipping = new ShippingService();
    private NotificationService notification = new NotificationService();

    // Simple unified interface
    public boolean placeOrder(Order order) {
        System.out.println("\n========== Processing Order: " + order.orderId + "
            ==========");

        String trackingNumber = null;

        try {
            // Step 1: Check and reserve inventory
            System.out.println("\nStep 1: Inventory Check");
            if (!inventory.checkStock(order.productId, order.quantity)) {
                notification.sendFailureNotification(order.customerId, "Product
                    out of stock");
                return false;
            }
            inventory.reserveStock(order.productId, order.quantity);
```

Facade - Order Processing Facade (Part 2)

```
            // Step 2: Process payment
            System.out.println("\nStep 2: Payment Processing");
            if (!payment.processPayment(order.customerId, order.amount)) {
                inventory.releaseStock(order.productId, order.quantity);
                notification.sendFailureNotification(order.customerId, "Payment
                     failed");
                return false;
            }

            // Step 3: Schedule shipping
            System.out.println("\nStep 3: Shipping Arrangement");
            trackingNumber = shipping.scheduleDelivery(order.customerId, order.
                 shippingAddress);

            // Step 4: Send notifications
            System.out.println("\nStep 4: Customer Notifications");
            notification.sendOrderConfirmation(order.customerId, order.orderId);

            System.out.println("\n========== $\checkmark$ Order " + order.orderId
                 + " Completed ==========");
            return true;
        } catch (Exception e) {
            // Rollback on any failure
            System.out.println("\n========== $\times$ Order Failed - Rolling Back
                 ==========");
            // ... rollback logic ...
            return false;
        }
    }
}
```

Facade - Usage Example

```java
// Client Code - Simple!
public class ECommerceApp {
    public static void main(String[] args) {
        System.out.println("========== E-Commerce Facade Pattern Demo ==========")
            ;

        OrderProcessingFacade orderFacade = new OrderProcessingFacade();

        // Client just calls ONE method - all complexity hidden!
        Order order1 = new Order("ORD-12345", "CUST-001", "PROD-789",
                                 2, 149.99, "123 Main St, City, State");

        orderFacade.placeOrder(order1);

        // Another order
        Order order2 = new Order("ORD-12346", "CUST-002", "PROD-456",
                                 1, 99.99, "456 Oak Ave, Town, State");

        orderFacade.placeOrder(order2);

        System.out.println("\n$\checkmark$ Client code is simple - Facade handles
            all complexity!");
        System.out.println("$\checkmark$ One method call orchestrates multiple
            subsystems!");
    }
}
```

Facade - Why Use It?

Benefits:

- **Simplicity**: One method call vs coordinating 5 subsystems
- **Reduced Coupling**: Client doesn't depend on all subsystems
- **Error Handling**: Facade handles rollback logic
- **Orchestration**: Manages complex workflow and dependencies
- **Maintainability**: Changes to subsystems don't affect client code

Common Use Cases:

- Complex library/framework initialization
- Multi-step business processes
- Legacy system integration
- Service orchestration in microservices

WebMVC Pattern - Definition

Definition: Separates application into three interconnected components - Model (data), View (UI), and Controller (business logic) to separate internal representations from how information is presented to the user.

Real-World Use Case: Spring WebMVC / Java Servlets

A web application where Controllers handle HTTP requests, Models represent business data, and Views render the response (JSP, Thymeleaf, etc.).

Key Components:

- **Model**: Data and business logic
- **View**: Presentation logic
- **Controller**: Request handling and coordination

WebMVC - Implementation (Part 1)

```java
package com.gdn.patterns.architectural.webmvc;

import java.util.HashMap;
import java.util.Map;

// Model - Represents the data and business logic
class User {
    private String username;
    private String email;

    public User(String username, String email) {
        this.username = username;
        this.email = email;
    }

    public String getUsername() { return username; }
    public String getEmail() { return email; }
    public void setEmail(String email) { this.email = email; }
}
```

WebMVC - Implementation (Part 2)

```java
// View - Handles presentation logic
class UserView {
    public void displayUserDetails(User user) {
        System.out.println("=== User Details ===");
        System.out.println("Username: " + user.getUsername());
        System.out.println("Email: " + user.getEmail());
    }

    public void displayError(String message) {
        System.out.println("ERROR: " + message);
    }
}
```

WebMVC - Implementation (Part 3)

```java
// Controller - Handles user input and updates Model/View
class UserController {
    private Map<String, User> userRepository = new HashMap<>();
    private UserView view;

    public UserController(UserView view) {
        this.view = view;
    }

    // Simulates handling a GET request
    public void getUser(String username) {
        User user = userRepository.get(username);
        if (user != null) {
            view.displayUserDetails(user);
        } else {
            view.displayError("User not found: " + username);
        }
    }

    // Simulates handling a POST request
    public void createUser(String username, String email) {
        if (userRepository.containsKey(username)) {
            view.displayError("User already exists: " + username);
            return;
        }
```

WebMVC - Implementation (Part 4)

```java
        User newUser = new User(username, email);
        userRepository.put(username, newUser);
        view.displayUserDetails(newUser);
    }

    // Simulates handling a PUT request
    public void updateUserEmail(String username, String newEmail) {
        User user = userRepository.get(username);
        if (user != null) {
            user.setEmail(newEmail);
            view.displayUserDetails(user);
        } else {
            view.displayError("User not found: " + username);
        }
    }
}
```

WebMVC - Usage Example

```java
// Example Usage (simulating web requests)
public class WebMVCDemo {
    public static void main(String[] args) {
        System.out.println("========== WebMVC Pattern Demo ==========\n");

        UserView view = new UserView();
        UserController controller = new UserController(view);

        // Simulate POST /users
        System.out.println(">>> POST /users (Create User)");
        controller.createUser("john_doe", "john@example.com");

        System.out.println("\n>>> POST /users (Create Another User)");
        controller.createUser("jane_smith", "jane@example.com");

        // Simulate GET /users/john_doe
        System.out.println("\n>>> GET /users/john_doe");
        controller.getUser("john_doe");
```

WebMVC - Usage Example (cont.)

```
        // Simulate PUT /users/john_doe
        System.out.println("\n>>> PUT /users/john_doe (Update Email)");
        controller.updateUserEmail("john_doe", "newemail@example.com");

        // Simulate GET /users/unknown
        System.out.println("\n>>> GET /users/unknown (User Not Found)");
        controller.getUser("unknown");

        // Try to create duplicate user
        System.out.println("\n>>> POST /users (Duplicate User)");
        controller.createUser("john_doe", "duplicate@example.com");

        System.out.println("\n$\checkmark$ Model-View-Controller separation
             achieved!");
        System.out.println("$\checkmark$ Each component has single responsibility!
             ");
    }
}
```

WebMVC - Why Use It?

Key Benefits:

- **Separation of Concerns**: Business logic, data, and presentation are separated
- **Parallel Development**: Teams can work on Model, View, and Controller independently
- **Testability**: Each component can be tested in isolation
- **Reusability**: Same Model can be used with different Views (Web, Mobile, API)

Common Use Cases:

- Web applications (Spring MVC, JSP, Servlets)
- RESTful APIs
- Desktop applications
- Mobile applications

WebMVC - Spring Framework Example

```java
// Spring WebMVC Example Structure:

@Controller
public class UserController {
    @Autowired
    private UserService userService; // Model layer

    @GetMapping("/users/{id}")
    public String getUser(@PathVariable Long id, Model model) {
        User user = userService.findById(id);
        model.addAttribute("user", user);
        return "userDetails"; // View name (JSP/Thymeleaf template)
    }

    @PostMapping("/users")
    public String createUser(@ModelAttribute User user) {
        userService.save(user);
        return "redirect:/users/" + user.getId();
    }
}
```

Assignment 3: Document Processing System

Scenario: You are building an enterprise document processing system that handles various document types and applies formatting.

Part 1 - Prototype Pattern: Implement document templates

- Create an abstract `Document` class implementing `Cloneable`
- Implement concrete types: `Invoice`, `Contract`, `Report`
- Each document should have: `title`, `content`, `metadata` (Map¡String, String¿)
- Implement `clone()` method for creating copies
- Create a `DocumentRegistry` that stores template documents and returns clones

Assignment 3: Document Processing System (cont.)

Part 2 - Decorator Pattern: Implement document processing pipeline

- Create a `DocumentProcessor` interface with `process(Document doc)` method
- Base implementation: `BasicDocumentProcessor`
- Decorators to add:
    - `WatermarkDecorator` - adds watermark to document
    - `EncryptionDecorator` - encrypts document content
    - `CompressionDecorator` - compresses document
    - `AuditLogDecorator` - logs processing activity
- Allow chaining multiple decorators

Part 3 - Facade Pattern: Simplify document operations

- Create a `DocumentServiceFacade` that hides complexity of: creating documents from templates, processing through multiple decorators, validating documents, saving to storage
- Provide simple methods like `createInvoice()`, `createSecureContract()`, `generateReport()`

Assignment 3: Deliverables

Deliverables:

- Document class hierarchy with Prototype pattern
- DocumentRegistry for managing templates
- DocumentProcessor interface and 4 decorators
- DocumentServiceFacade
- A comprehensive demo showing:
  - Creating documents from templates
  - Cloning and modifying documents
  - Processing documents through decorator chain
  - Using facade for simplified operations

Bonus Challenges:

- Add validation in the Facade before processing
- Implement a decorator that can be removed (like undo functionality)
- Add different types of documents to the registry

Time: 45 minutes

Summary - Patterns Covered Today

Creational Patterns

- **Singleton**: Single instance
- **Factory**: Delegate creation
- **Builder**: Complex construction
- **Prototype**: Clone objects

Structural Patterns

- **Proxy**: Lazy loading
- **Decorator**: Add behavior
- **Facade**: Simplify subsystems

Behavioral Patterns

- **Command**: Undo/redo
- **Observer**: Event notification

Architectural Patterns

- **WebMVC**: MVC pattern

Assignments

- Configuration Management
- Smart Home Automation
- Document Processing

Key Takeaways

1. **Design patterns solve recurring problems** in software design
2. **Patterns promote code reuse, maintainability, and extensibility**
3. **Choose the right pattern** based on the problem context
4. **Patterns work together** - combine them for complex solutions
5. **Don't over-engineer** - use patterns when they add value
6. **Practice makes perfect** - apply patterns in real projects

Thank You!

Questions?