# Composite Design Pattern - Complete Guide

## 🎯 Pattern Overview

The Composite Pattern is a structural design pattern that allows you to compose objects into tree structures to represent part-whole hierarchies. It lets clients treat individual objects and compositions of objects uniformly.

**In simple terms: Build tree structures where individual items and groups of items are treated the same way!**

## 🧩 Core Components

### 1. Component (Abstract)

- Declares interface for objects in the composition
- Defines common operations for both simple and complex objects
- May implement default behavior common to all classes
- Declares interface for managing child components

### 2. Leaf

- Represents leaf objects in the composition (no children)
- Implements Component interface
- Defines behavior for primitive objects
- Cannot have children

### 3. Composite

- Defines behavior for components having children
- Stores child components
- Implements child-related operations in Component interface
- Usually delegates work to child components and combines results

### 4. Client

- Manipulates objects through Component interface
- Treats individual objects and compositions uniformly
- Doesn't need to know if working with leaf or composite

## 🏛️ Real-Life Example: File System

```
File System (Root Directory)
├── Documents (Folder)
│   ├── resume.pdf (File)
│   ├── Projects (Folder)
│   │   ├── project1.txt (File)
│   │   └── project2.java (File)
│   └── Notes (Folder)
│       └── meeting.doc (File)
├── Pictures (Folder)
│   ├── photo1.jpg (File)
│   └── photo2.png (File)
└── readme.txt (File)
```

Both Files and Folders can be treated uniformly - you can get size, copy, move, or delete either one!

## 🔄 Working Flow

1. **Client Request**: Client calls operation on Component (could be Leaf or Composite)
2. **Leaf Processing**: If it's a Leaf, performs operation directly and returns result
3. **Composite Processing**: If it's a Composite, iterates through children
4. **Recursive Calls**: Each child processes the request (may be Leaf or Composite)
5. **Result Aggregation**: Composite combines results from all children
6. **Response**: Final result is returned to client

## ✅ Advantages

### 1. Uniform Treatment

- Clients can treat individual objects and compositions identically
- Simplifies client code - no need to differentiate between types
- Common interface for all objects in hierarchy

### 2. Hierarchical Structure

- Natural representation of tree-like structures
- Easy to add new kinds of components
- Flexible composition of objects

### 3. Open/Closed Principle

- Easy to add new component types without changing existing code
- Existing client code works with new components
- Follows Open/Closed Principle perfectly

### 4. Recursive Operations

- Operations can be applied to entire tree structures
- Automatic traversal through composition
- Clean and elegant recursive implementations

### 5. Dynamic Composition

- Can build complex structures at runtime
- Easy to add/remove components dynamically
- Flexible object composition

## ❌ Disadvantages

### 1. Overly General Design

- Component interface might become too general
- Hard to restrict components of composite
- May need runtime type checking

### 2. Type Safety Issues

- Difficult to enforce type constraints at compile time
- May need to rely on runtime checks
- Generic component interface loses some type safety

### 3. Complex Implementation

- Can be overkill for simple hierarchies
- Requires careful design of component interface
- May add unnecessary complexity

## 🛠️ Implementation Example

```java
```

```
import java.util.*;

// Component - Abstract base class
abstract class FileSystemComponent {
    protected String name;

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

    // Common operations
    public String getName() {
        return name;
    }

    // Operations that should be implemented by both Leaf and Composite
    public abstract long getSize();
    public abstract void display(int depth);
    public abstract FileSystemComponent copy();

    // Operations for Composite only (default implementations)
    public void add(FileSystemComponent component) {
        throw new UnsupportedOperationException("Cannot add to a leaf component");
    }

    public void remove(FileSystemComponent component) {
        throw new UnsupportedOperationException("Cannot remove from a leaf component");
    }

    public List<FileSystemComponent> getChildren() {
        throw new UnsupportedOperationException("Leaf components don't have children");
    }

    // Utility method for display indentation
    protected String getIndent(int depth) {
        return "  ".repeat(depth);
    }
}

// Leaf - File
class File extends FileSystemComponent {
    private long size;
    private String extension;

    public File(String name, long size) {
        super(name);
```

```
        this.size = size;
        this.extension = getFileExtension(name);
    }

    private String getFileExtension(String fileName) {
        int lastDot = fileName.lastIndexOf('.');
        return lastDot > 0 ? fileName.substring(lastDot + 1) : "";
    }

    @Override
    public long getSize() {
        return size;
    }

    @Override
    public void display(int depth) {
        System.out.println(getIndent(depth) + "📄 " + name + " (" + size + " bytes)");
    }

    @Override
    public FileSystemComponent copy() {
        return new File(name, size);
    }

    public String getExtension() {
        return extension;
    }

    // File-specific operations
    public void open() {
        System.out.println("Opening file: " + name);
    }

    public void edit(String content) {
        System.out.println("Editing file: " + name);
        // Simulate size change
        this.size = content.length();
    }
}

// Composite - Directory/Folder
class Directory extends FileSystemComponent {
    private List<FileSystemComponent> children;

    public Directory(String name) {
        super(name);
        this.children = new ArrayList<>();
```

```
}

@Override
public long getSize() {
    long totalSize = 0;
    for (FileSystemComponent child : children) {
        totalSize += child.getSize();
    }
    return totalSize;
}

@Override
public void display(int depth) {
    System.out.println(getIndent(depth) + "📁 " + name + "/ (" + getSize() + " bytes total)");
    for (FileSystemComponent child : children) {
        child.display(depth + 1);
    }
}

@Override
public FileSystemComponent copy() {
    Directory copyDir = new Directory(name);
    for (FileSystemComponent child : children) {
        copyDir.add(child.copy());
    }
    return copyDir;
}

@Override
public void add(FileSystemComponent component) {
    children.add(component);
    System.out.println("Added " + component.getName() + " to " + this.name);
}

@Override
public void remove(FileSystemComponent component) {
    children.remove(component);
    System.out.println("Removed " + component.getName() + " from " + this.name);
}

@Override
public List<FileSystemComponent> getChildren() {
    return new ArrayList<>(children);
}

// Directory-specific operations
public FileSystemComponent find(String fileName) {
```

```
        for (FileSystemComponent child : children) {
            if (child.getName().equals(fileName)) {
                return child;
            }
            if (child instanceof Directory) {
                FileSystemComponent found = ((Directory) child).find(fileName);
                if (found != null) {
                    return found;
                }
            }
        }
        return null;
    }

    public List<File> getAllFiles() {
        List<File> allFiles = new ArrayList<>();
        for (FileSystemComponent child : children) {
            if (child instanceof File) {
                allFiles.add((File) child);
            } else if (child instanceof Directory) {
                allFiles.addAll(((Directory) child).getAllFiles());
            }
        }
        return allFiles;
    }

    public int getFileCount() {
        int count = 0;
        for (FileSystemComponent child : children) {
            if (child instanceof File) {
                count++;
            } else if (child instanceof Directory) {
                count += ((Directory) child).getFileCount();
            }
        }
        return count;
    }
}

// Usage Example
public class CompositePatternDemo {
    public static void main(String[] args) {
        System.out.println("=== Composite Pattern Demo: File System ===\n");

        // Create files
        File resume = new File("resume.pdf", 1024);
        File project1 = new File("project1.txt", 2048);
```

```
File project2 = new File("project2.java", 4096);
File meetingNotes = new File("meeting.doc", 512);
File photo1 = new File("photo1.jpg", 8192);
File photo2 = new File("photo2.png", 6144);
File readme = new File("readme.txt", 256);

// Create directories
Directory root = new Directory("Root");
Directory documents = new Directory("Documents");
Directory projects = new Directory("Projects");
Directory notes = new Directory("Notes");
Directory pictures = new Directory("Pictures");

// Build the tree structure
root.add(documents);
root.add(pictures);
root.add(readme);

documents.add(resume);
documents.add(projects);
documents.add(notes);

projects.add(project1);
projects.add(project2);

notes.add(meetingNotes);

pictures.add(photo1);
pictures.add(photo2);

// Display the entire file system
System.out.println("File System Structure:");
root.display(0);

// Demonstrate uniform treatment
System.out.println("\n=== Uniform Operations ===");
demonstrateUniformOperations(root);
demonstrateUniformOperations(resume);
demonstrateUniformOperations(pictures);

// Directory-specific operations
System.out.println("\n=== Directory Operations ===");
System.out.println("Total files in root: " + root.getFileCount());
System.out.println("All files: ");
root.getAllFiles().forEach(file ->
    System.out.println("  - " + file.getName() + " (" + file.getSize() + " bytes)"));
```

```
        // Search functionality
        System.out.println("\n=== Search Operations ===");
        FileSystemComponent found = root.find("project1.txt");
        if (found != null) {
            System.out.println("Found: " + found.getName());
        }

        // Copy operation
        System.out.println("\n=== Copy Operations ===");
        FileSystemComponent rootCopy = root.copy();
        System.out.println("Copied file system:");
        rootCopy.display(0);

        // Dynamic modification
        System.out.println("\n=== Dynamic Modifications ===");
        File newFile = new File("temp.txt", 128);
        documents.add(newFile);
        System.out.println("After adding temp.txt:");
        root.display(0);

        documents.remove(newFile);
        System.out.println("After removing temp.txt:");
        root.display(0);
    }

    // This method treats all components uniformly
    private static void demonstrateUniformOperations(FileSystemComponent component) {
        System.out.println("Component: " + component.getName());
        System.out.println("Size: " + component.getSize() + " bytes");
        System.out.println("Type: " + (component instanceof File ? "File" : "Directory"));
        System.out.println("---");
    }
}
```

## 🔧 Advanced Features

### 1. Visitor Pattern Integration

java

```
// Visitor interface for different operations
interface FileSystemVisitor {
    void visitFile(File file);
    void visitDirectory(Directory directory);
}

// Add accept method to FileSystemComponent
abstract class FileSystemComponent {
    // ... existing code ...

    public abstract void accept(FileSystemVisitor visitor);
}

// Implementation in File
class File extends FileSystemComponent {
    // ... existing code ...

    @Override
    public void accept(FileSystemVisitor visitor) {
        visitor.visitFile(this);
    }
}

// Implementation in Directory
class Directory extends FileSystemComponent {
    // ... existing code ...

    @Override
    public void accept(FileSystemVisitor visitor) {
        visitor.visitDirectory(this);
        for (FileSystemComponent child : children) {
            child.accept(visitor);
        }
    }
}

// Concrete visitors
class SizeCalculatorVisitor implements FileSystemVisitor {
    private long totalSize = 0;

    @Override
    public void visitFile(File file) {
        totalSize += file.getSize();
    }

    @Override
```

```
    public void visitDirectory(Directory directory) {
        // Directory size is calculated by visiting children
    }

    public long getTotalSize() {
        return totalSize;
    }
}

class BackupVisitor implements FileSystemVisitor {
    private List<String> backupPaths = new ArrayList<>();

    @Override
    public void visitFile(File file) {
        backupPaths.add("Backing up file: " + file.getName());
    }

    @Override
    public void visitDirectory(Directory directory) {
        backupPaths.add("Creating backup directory: " + directory.getName());
    }

    public List<String> getBackupPaths() {
        return backupPaths;
    }
}
```

## 2. Iterator Pattern for Tree Traversal

java

```
class TreeIterator implements Iterator<FileSystemComponent> {
    private Stack<FileSystemComponent> stack;

    public TreeIterator(FileSystemComponent root) {
        stack = new Stack<>();
        stack.push(root);
    }

    @Override
    public boolean hasNext() {
        return !stack.isEmpty();
    }

    @Override
    public FileSystemComponent next() {
        if (!hasNext()) {
            throw new NoSuchElementException();
        }

        FileSystemComponent current = stack.pop();

        if (current instanceof Directory) {
            Directory dir = (Directory) current;
            // Add children in reverse order for correct traversal
            List<FileSystemComponent> children = dir.getChildren();
            for (int i = children.size() - 1; i >= 0; i--) {
                stack.push(children.get(i));
            }
        }

        return current;
    }
}

// Add to FileSystemComponent
public Iterator<FileSystemComponent> iterator() {
    return new TreeIterator(this);
}
```

## 3. Caching and Performance Optimization

java

```
class CachedDirectory extends Directory {
    private Long cachedSize;
    private long lastModified;
    private boolean isDirty;

    public CachedDirectory(String name) {
        super(name);
        this.isDirty = true;
    }

    @Override
    public long getSize() {
        if (isDirty || cachedSize == null) {
            cachedSize = super.getSize();
            lastModified = System.currentTimeMillis();
            isDirty = false;
        }
        return cachedSize;
    }

    @Override
    public void add(FileSystemComponent component) {
        super.add(component);
        invalidateCache();
    }

    @Override
    public void remove(FileSystemComponent component) {
        super.remove(component);
        invalidateCache();
    }

    private void invalidateCache() {
        this.isDirty = true;
        this.cachedSize = null;
    }
}
```

## 🎯 When to Use Composite Pattern

### ✅ Use When:

- You want to represent part-whole hierarchies of objects
- You want clients to treat individual objects and compositions uniformly
- You have tree-like structures (file systems, organizational charts, UI components)

- You need to perform operations on entire object hierarchies
- Structure can be represented as a tree with uniform interface
- You want to simplify client code by eliminating type distinctions

### ❌ Avoid When:

- Your hierarchy is very simple and doesn't need uniform treatment
- Different components have vastly different interfaces
- Performance is critical and you can't afford the overhead
- Type safety is more important than uniform treatment
- You don't need recursive operations on the structure

## 💥 Real-World Examples

### 1. UI Component Hierarchy

java

```
// UI Component system
abstract class UIComponent {
    protected String name;
    protected boolean visible = true;

    public abstract void render();
    public abstract void handleEvent(Event event);

    public void setVisible(boolean visible) {
        this.visible = visible;
    }

    // Composite operations
    public void add(UIComponent component) {
        throw new UnsupportedOperationException();
    }

    public void remove(UIComponent component) {
        throw new UnsupportedOperationException();
    }
}

class Button extends UIComponent {
    private String text;

    public Button(String name, String text) {
        this.name = name;
        this.text = text;
    }

    @Override
    public void render() {
        if (visible) {
            System.out.println("Rendering button: " + text);
        }
    }

    @Override
    public void handleEvent(Event event) {
        if (event.getType() == EventType.CLICK) {
            System.out.println("Button " + name + " clicked!");
        }
    }
}

class Panel extends UIComponent {
```

```
    private List<UIComponent> components = new ArrayList<>();

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

    @Override
    public void render() {
        if (visible) {
            System.out.println("Rendering panel: " + name);
            for (UIComponent component : components) {
                component.render();
            }
        }
    }

    @Override
    public void handleEvent(Event event) {
        for (UIComponent component : components) {
            component.handleEvent(event);
        }
    }

    @Override
    public void add(UIComponent component) {
        components.add(component);
    }

    @Override
    public void remove(UIComponent component) {
        components.remove(component);
    }
}
```

## 2. Organization Structure

java

```
abstract class Employee {
    protected String name;
    protected String position;
    protected double salary;

    public Employee(String name, String position, double salary) {
        this.name = name;
        this.position = position;
        this.salary = salary;
    }

    public abstract void showDetails();
    public abstract double getTotalSalary();

    // Composite operations
    public void addSubordinate(Employee employee) {
        throw new UnsupportedOperationException();
    }

    public void removeSubordinate(Employee employee) {
        throw new UnsupportedOperationException();
    }
}

class Developer extends Employee {
    private String programmingLanguage;

    public Developer(String name, double salary, String language) {
        super(name, "Developer", salary);
        this.programmingLanguage = language;
    }

    @Override
    public void showDetails() {
        System.out.println("Developer: " + name + ", Language: " + programmingLanguage +
                           ", Salary: $" + salary);
    }

    @Override
    public double getTotalSalary() {
        return salary;
    }
}

class Manager extends Employee {
    private List<Employee> subordinates = new ArrayList<>();
```

```
    public Manager(String name, double salary) {
        super(name, "Manager", salary);
    }

    @Override
    public void showDetails() {
        System.out.println("Manager: " + name + ", Salary: $" + salary);
        System.out.println("Subordinates:");
        for (Employee subordinate : subordinates) {
            subordinate.showDetails();
        }
    }

    @Override
    public double getTotalSalary() {
        double total = salary;
        for (Employee subordinate : subordinates) {
            total += subordinate.getTotalSalary();
        }
        return total;
    }

    @Override
    public void addSubordinate(Employee employee) {
        subordinates.add(employee);
    }

    @Override
    public void removeSubordinate(Employee employee) {
        subordinates.remove(employee);
    }
}
```

## 3. Mathematical Expressions

java

```
abstract class Expression {
    public abstract double evaluate();
    public abstract String toString();
}

class Number extends Expression {
    private double value;

    public Number(double value) {
        this.value = value;
    }

    @Override
    public double evaluate() {
        return value;
    }

    @Override
    public String toString() {
        return String.valueOf(value);
    }
}

class BinaryOperation extends Expression {
    private Expression left;
    private Expression right;
    private String operator;

    public BinaryOperation(Expression left, String operator, Expression right) {
        this.left = left;
        this.operator = operator;
        this.right = right;
    }

    @Override
    public double evaluate() {
        double leftValue = left.evaluate();
        double rightValue = right.evaluate();

        switch (operator) {
            case "+": return leftValue + rightValue;
            case "-": return leftValue - rightValue;
            case "*": return leftValue * rightValue;
            case "/": return leftValue / rightValue;
            default: throw new IllegalArgumentException("Unknown operator: " + operator);
        }
```

```
    }

    @Override
    public String toString() {
        return "(" + left.toString() + " " + operator + " " + right.toString() + ")";
    }
}

// Usage: (3 + 4) * (2 - 1)
Expression expr = new BinaryOperation(
    new BinaryOperation(new Number(3), "+", new Number(4)),
    "*",
    new BinaryOperation(new Number(2), "-", new Number(1))
);
```

## 4. Menu System

java

```
abstract class MenuComponent {
    protected String name;

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

    public abstract void display();
    public abstract void select();

    // Composite operations
    public void add(MenuComponent component) {
        throw new UnsupportedOperationException();
    }

    public void remove(MenuComponent component) {
        throw new UnsupportedOperationException();
    }
}

class MenuItem extends MenuComponent {
    private Runnable action;

    public MenuItem(String name, Runnable action) {
        super(name);
        this.action = action;
    }

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

    @Override
    public void select() {
        System.out.println("Executing: " + name);
        action.run();
    }
}

class Menu extends MenuComponent {
    private List<MenuComponent> menuComponents = new ArrayList<>();

    public Menu(String name) {
        super(name);
    }
```

```
    @Override
    public void display() {
        System.out.println(name + ":");
        for (MenuComponent component : menuComponents) {
            component.display();
        }
    }

    @Override
    public void select() {
        display();
    }

    @Override
    public void add(MenuComponent component) {
        menuComponents.add(component);
    }

    @Override
    public void remove(MenuComponent component) {
        menuComponents.remove(component);
    }

    public MenuComponent getChild(int index) {
        return menuComponents.get(index);
    }
}
```

## 📊 Performance Considerations

### 1. Lazy Loading

```
java
```

```
class LazyDirectory extends Directory {
    private boolean loaded = false;
    private String path;

    public LazyDirectory(String name, String path) {
        super(name);
        this.path = path;
    }

    private void loadIfNeeded() {
        if (!loaded) {
            loadChildrenFromFileSystem(path);
            loaded = true;
        }
    }

    @Override
    public long getSize() {
        loadIfNeeded();
        return super.getSize();
    }

    @Override
    public List<FileSystemComponent> getChildren() {
        loadIfNeeded();
        return super.getChildren();
    }

    private void loadChildrenFromFileSystem(String path) {
        // Load actual files and directories from filesystem
        // This is just a simulation
        System.out.println("Loading directory contents for: " + path);
    }
}
```

## 2. Memory Optimization with Flyweight

java

```
class FileType {
    private String extension;
    private String icon;
    private String description;

    public FileType(String extension, String icon, String description) {
        this.extension = extension;
        this.icon = icon;
        this.description = description;
    }

    // Getters...
}

class FileTypeFactory {
    private static Map<String, FileType> fileTypes = new HashMap<>();

    public static FileType getFileType(String extension) {
        return fileTypes.computeIfAbsent(extension, ext -> {
            // Create file type based on extension
            return createFileType(ext);
        });
    }

    private static FileType createFileType(String extension) {
        switch (extension.toLowerCase()) {
            case "pdf": return new FileType("pdf", "📄", "PDF Document");
            case "jpg": return new FileType("jpg", "🖼️", "JPEG Image");
            case "txt": return new FileType("txt", "📝", "Text File");
            default: return new FileType(extension, "📄", "Unknown File");
        }
    }
}
```

## 3. Concurrent Access Support

java

```
class ThreadSafeDirectory extends Directory {
    private final ReadWriteLock lock = new ReentrantReadWriteLock();

    public ThreadSafeDirectory(String name) {
        super(name);
    }

    @Override
    public long getSize() {
        lock.readLock().lock();
        try {
            return super.getSize();
        } finally {
            lock.readLock().unlock();
        }
    }

    @Override
    public void add(FileSystemComponent component) {
        lock.writeLock().lock();
        try {
            super.add(component);
        } finally {
            lock.writeLock().unlock();
        }
    }

    @Override
    public void remove(FileSystemComponent component) {
        lock.writeLock().lock();
        try {
            super.remove(component);
        } finally {
            lock.writeLock().unlock();
        }
    }
}
```

## 🎓 Best Practices

### 1. Design Component Interface Carefully

java

```java
// Good: Focused interface with clear responsibilities
interface DrawableComponent {
    void draw(Graphics g);
    Rectangle getBounds();
    boolean contains(Point point);

    // Optional operations with default implementations
    default void add(DrawableComponent component) {
        throw new UnsupportedOperationException("Not a composite");
    }

    default void remove(DrawableComponent component) {
        throw new UnsupportedOperationException("Not a composite");
    }

    default List<DrawableComponent> getChildren() {
        return Collections.emptyList();
    }
}
```

## 2. Handle Invalid Operations Gracefully

```java
abstract class Component {
    // Use enum for better type safety
    public enum ComponentType {
        LEAF, COMPOSITE
    }

    public abstract ComponentType getType();

    public void add(Component component) {
        if (getType() == ComponentType.LEAF) {
            throw new UnsupportedOperationException(
                "Cannot add children to leaf component: " + getClass().getSimpleName());
        }
        doAdd(component);
    }

    protected void doAdd(Component component) {
        // Override in composite classes
    }
}
```

## 3. Implement Proper Equality and Hashing

java

```java
class FileSystemComponent {
    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass()) return false;

        FileSystemComponent that = (FileSystemComponent) obj;
        return Objects.equals(name, that.name);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name);
    }
}
```

## 4. Use Builder Pattern for Complex Composites

java

```
class DirectoryBuilder {
    private Directory directory;

    public DirectoryBuilder(String name) {
        this.directory = new Directory(name);
    }

    public DirectoryBuilder addFile(String name, long size) {
        directory.add(new File(name, size));
        return this;
    }

    public DirectoryBuilder addDirectory(String name, Consumer<DirectoryBuilder> builderConsumer) {
        DirectoryBuilder subBuilder = new DirectoryBuilder(name);
        builderConsumer.accept(subBuilder);
        directory.add(subBuilder.build());
        return this;
    }

    public Directory build() {
        return directory;
    }
}

// Usage
Directory projectDir = new DirectoryBuilder("Project")
    .addFile("README.md", 1024)
    .addFile("pom.xml", 512)
    .addDirectory("src", src -> src
        .addDirectory("main", main -> main
            .addDirectory("java", java -> java
                .addFile("Main.java", 2048))))
    .build();
```

## 🔗 Related Patterns

### 1. Composite + Visitor

Already shown above - perfect for adding operations without modifying the composite structure.

### 2. Composite + Iterator

```
java
```

```java
// Different traversal strategies
enum TraversalType {
    DEPTH_FIRST, BREADTH_FIRST, LEAVES_ONLY
}

class CompositeIteratorFactory {
    public static Iterator<FileSystemComponent> createIterator(
            FileSystemComponent root, TraversalType type) {
        switch (type) {
            case DEPTH_FIRST: return new DepthFirstIterator(root);
            case BREADTH_FIRST: return new BreadthFirstIterator(root);
            case LEAVES_ONLY: return new LeavesOnlyIterator(root);
            default: throw new IllegalArgumentException("Unknown traversal type");
        }
    }
}
```

## 3. Composite + Command

java

```java
class CompositeCommand implements Command {
    private List<Command> commands = new ArrayList<>();

    public void add(Command command) {
        commands.add(command);
    }

    @Override
    public void execute() {
        for (Command command : commands) {
            command.execute();
        }
    }

    @Override
    public void undo() {
        // Undo in reverse order
        for (int i = commands.size() - 1; i >= 0; i--) {
            commands.get(i).undo();
        }
    }
}
```