SWE 4501: Design Pattern

Afrin Jahan Era
ID: 220042132
Department: CSE
Programme: SWE

DP_Assignment_2

The *Age of Villagers (AoV)* game allows users to create villages composed of different objects such as houses, trees, and water sources. Each object is built using simple shapes, and users can choose between different combinations of components. Additionally, the system must support multiple operations like cost calculation, resource estimation, and report generation on the same village structure.

To meet these requirements in a clean, extensible, and maintainable way, the system is designed using the Composite, Factory, Builder, and Visitor design patterns.

System Requirements Addressed

The implementation fulfills the following requirements:

1. A village can contain houses, trees, and water sources
2. Each object is created using simple shapes
3. Users can choose between different combinations (e.g., Brick house + Mango tree + Pool)
4. Multiple operations can be performed on the same structure
5. The design should use appropriate object-oriented design patterns

All requirements are fully satisfied by the final implementation.

Design Pattern Justification

Composite Pattern

Purpose:
The Composite pattern is used to represent the hierarchical structure of the village.

Justification:

- A village consists of multiple objects (houses, trees, water sources)
- Each object itself consists of multiple shapes
- Both individual shapes and groups of objects must be treated uniformly

Implementation:

- VillageComponent acts as the common interface
- Shape represents the leaf nodes (simple shapes)
- CompositeObject represents composite nodes (House, Tree, Water Source, Village)

Benefit:
The Composite pattern allows recursive traversal of the village structure and enables operations to be applied uniformly to both simple and complex elements.

Component Interface

```
interface VillageComponent {
    void accept(Visitor visitor);
}
```

Leaf Class

```
class Shape implements VillageComponent {
    private String name;
    private int cost;

    public Shape(String name, int cost) {
        this.name = name;
        this.cost = cost;
    }

    public String getName() { return name; }
    public int getCost() { return cost; }

    public void accept(Visitor visitor) {
        visitor.visitShape(this);
    }
}
```

Composite Class

```java
class CompositeObject implements VillageComponent {
    private String name;
    private List<VillageComponent> children = new ArrayList<>();

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

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

    public List<VillageComponent> getChildren() {
        return children;
    }

    public String getName() {
        return name;
    }

    public void accept(Visitor visitor) {
        visitor.visitComposite(this);
        for (VillageComponent c : children) {
            c.accept(visitor);
        }
    }
}
```

Factory Pattern

Purpose:
The Factory pattern encapsulates the creation of simple shape objects.

Justification:

- Shape creation logic should not be scattered across the code
- Different shapes (Brick, Mud, Leaf, Water) have different costs
- Centralized creation improves maintainability and consistency

Implementation:

- ShapeFactory is responsible for creating Shape objects based on type

Benefit:
This separates object creation from usage and makes it easy to add new shape types without modifying existing client code.\

```
class ShapeFactory {
    public static Shape createShape(String type) {
        if (type.equals("Brick")) return new Shape("Brick", 100);
        if (type.equals("Mud")) return new Shape("Mud", 60);
        if (type.equals("MangoLeaf")) return new
Shape("MangoLeaf", 20);
        if (type.equals("BananaLeaf")) return new
Shape("BananaLeaf", 15);
        if (type.equals("PoolWater")) return new
Shape("PoolWater", 50);
        if (type.equals("PondWater")) return new
Shape("PondWater", 40);
        return null;
    }
}
```

Builder Pattern

Purpose:
The Builder pattern is used to construct different village combinations step by step.

Justification:

- Villages can be built using different combinations of houses, trees, and water sources
- Construction involves multiple steps and components
- The same construction process can create different final villages

Implementation:

- VillageBuilder provides methods such as buildHouse, buildTree, and buildWaterSource
- Different combinations are created by calling these methods in different sequences

Benefit:
The Builder pattern allows flexible and readable construction of complex village objects while hiding construction details from the client.

```java
class VillageBuilder {
    private CompositeObject village;

    public VillageBuilder() {
        village = new CompositeObject("Village");
    }

    public VillageBuilder buildHouse(String type) {
        CompositeObject house = new CompositeObject(type + "
House");
        house.add(ShapeFactory.createShape(type));
        house.add(ShapeFactory.createShape(type));
        village.add(house);
        return this;
    }

    public VillageBuilder buildTree(String type) {
        CompositeObject tree = new CompositeObject(type + "
Tree");
        tree.add(ShapeFactory.createShape(type + "Leaf"));
        tree.add(ShapeFactory.createShape(type + "Leaf"));
        village.add(tree);
        return this;
    }

    public VillageBuilder buildWaterSource(String type) {
        CompositeObject water = new CompositeObject(type);
        water.add(ShapeFactory.createShape(type + "Water"));
        water.add(ShapeFactory.createShape(type + "Water"));
        village.add(water);
        return this;
    }

    public CompositeObject getVillage() {
        return village;
    }
}
```

Visitor Pattern

Purpose:
The Visitor pattern enables multiple operations on the village structure without modifying its classes.

Justification:

- The same village structure must support:
  - Cost calculation
  - Resource estimation
  - Report generation
- Adding these operations directly to the component classes would violate the Single Responsibility Principle

Implementation:

- Visitor interface defines visit methods for shapes and composite objects
- CostVisitor calculates total cost
- ResourceVisitor counts required resources
- ReportVisitor generates a textual report

Benefit:
New operations can be added easily by introducing new visitor classes without changing the village structure.

Visitor Interface

```java
interface Visitor {
    void visitShape(Shape shape);
    void visitComposite(CompositeObject composite);
}
```

Cost Calculation Visitor

```java
class CostVisitor implements Visitor {
    private int totalCost = 0;

    public void visitShape(Shape shape) {
        totalCost += shape.getCost();
    }

    public void visitComposite(CompositeObject composite) {}

    public int getTotalCost() {
        return totalCost;
    }
}
```

Resource Estimation Visitor

```java
class ResourceVisitor implements Visitor {
    private Map<String, Integer> resources = new HashMap<>();

    public void visitShape(Shape shape) {
        resources.put(shape.getName(),
                resources.getOrDefault(shape.getName(), 0) + 1);
    }

    public void visitComposite(CompositeObject composite) {}

    public Map<String, Integer> getResources() {
        return resources;
    }
}
```

Report Generation Visitor

```java
class ReportVisitor implements Visitor {
    private StringBuilder report = new StringBuilder();

    public void visitShape(Shape shape) {
        report.append("Shape:
").append(shape.getName()).append("\n");
    }

    public void visitComposite(CompositeObject composite) {
        report.append("Object:
").append(composite.getName()).append("\n");
    }

    public String getReport() {
        return report.toString();
    }
}
```

How the Patterns Work Together

- Composite defines the village structure
- Factory creates the basic building blocks (shapes)
- Builder assembles these blocks into different village configurations
- Visitor performs various operations on the completed structure

This combination ensures high cohesion, low coupling, and strong extensibility.

Fulfillment of the Original Scenario

Requirement	Solution
Village with multiple object types	Composite
Objects built from shapes	Composite + Factory
Multiple village combinations	Builder
Cost calculation	Visitor
Resource estimation	Visitor
Report generation	Visitor

All requirements of the *Age of Villagers* village creation scenario are fully met.