

# Composite

Definition

The Problem

Solution: Composite Design Pattern

Use cases

Class Diagram

Structure of Composite Pattern

Implementation

Example 1: File Structure

Example 2: Arithmetic Expressions

▼ Resources

- Video → [32. All Structural Design Patterns | Decorator, Proxy, Composite, Adapter, Bridge, Facade, FlyWeight](#)
- Video → [19. Design File System using Composite Design Pattern | Low Level Design Interview Question | LLD](#)

## Definition

*The Composite Design Pattern allows you to compose objects into tree structures to **represent a part-whole hierarchies** which lets **clients** manage and perform operations on **individual objects(leaf nodes)** and **compositions of objects(composite nodes)** **uniformly**(treating both objects the same way i.e., without knowing whether it's working with a component or a composite).*

## The Problem

Understanding the problems by implementing the File Structure example using a naive approach:

```
1 public class File {
2     String fileName;
3
4     public File(String name) {
5         this.fileName = name;
6     }
7
8     public void printContents() {
9         System.out.println("File name: " + fileName);
10    }
11 }
```

```
1 public class Directory {
2     String directoryName;
3     List<Object> objectList;
4
5     public Directory(String name) {
6         this.directoryName = name;
7         objectList = new ArrayList<>();
8     }
9
10    public void add(Object object) {
11        objectList.add(object);
12    }
```

```

13
14     public void remove(Object object) {
15         objectList.remove(object);
16     }
17
18     // Display full structure
19     // Breaks OCP - if we want to add a new file type, we need to
    modify this method to add another if/else condition
20     public void printContents() {
21         System.out.println("Directory Name: " + directoryName);
22         for (Object obj: objectList) {
23             if (obj instanceof File) {
24                 ((File) obj).printContents();
25             } else if (obj instanceof Directory) {
26                 ((Directory) obj).printContents();
27             }
28         }
29     }
30 }

```

```

1 // Client Code
2 public class Client {
3     public static void main(String[] args) {
4         Directory movieDirectory = new Directory("Movies");
5
6         File rentalReceipt = new File("RentalReceipt");
7         movieDirectory.add(rentalReceipt);
8
9         Directory comedyMovieDirectory = new
    Directory("ComedyMovies");
10        File dumbAndDumber = new File("DumbAndDumber");
11        comedyMovieDirectory.add(dumbAndDumber);
12        movieDirectory.add(comedyMovieDirectory);
13
14        movieDirectory.printContents();
15    }
16 }

```

## • No Common Abstraction

- `File` and `Directory` → Both are different types.
- If you write client code, you need to know whether **it is a File or a Folder** before calling methods.

```

public void printContents() {
    System.out.println("Directory Name: " + directoryName);
    for (Object obj : objectList) {
        if (obj instanceof File) {
            ((File) obj).printContents();
        } else if (obj instanceof Directory) {
            ((Directory) obj).printContents();
        }
    }
}

```

- This breaks the Open/Closed Principle. If we want to add a new file type, we need to modify this method to add another `if/else` condition (which is messy).

## • Scalability Issue

- If later you add a new type (say a `CompressedFolder`), you have to update everywhere you wrote those `if-else` checks.
- This makes the system rigid, tightly-coupled, and hard to extend.

## Solution: Composite Design Pattern

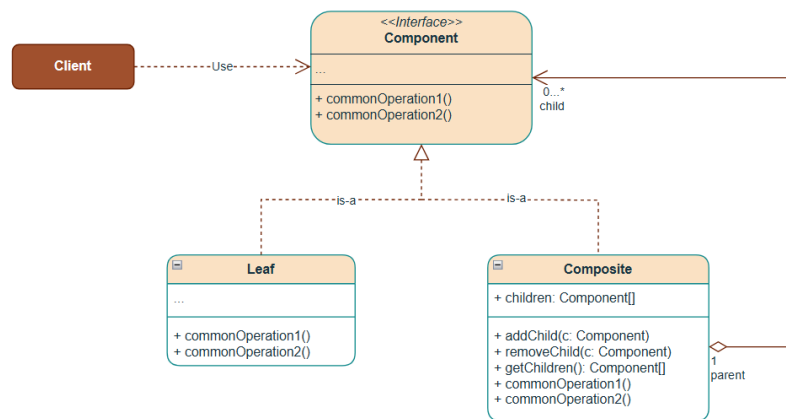
With the Composite Pattern:

- Both **File** and **Folder** implement the same Component interface.
- Client code doesn't have to know whether it's a file or a folder, and hence it can treat both as composite and perform operations uniformly (advantage of using abstraction).
- Adding new types (e.g., **Shortcut** or **ZipFolder**) doesn't require rewriting client logic, making it an extensible design.

## Use cases

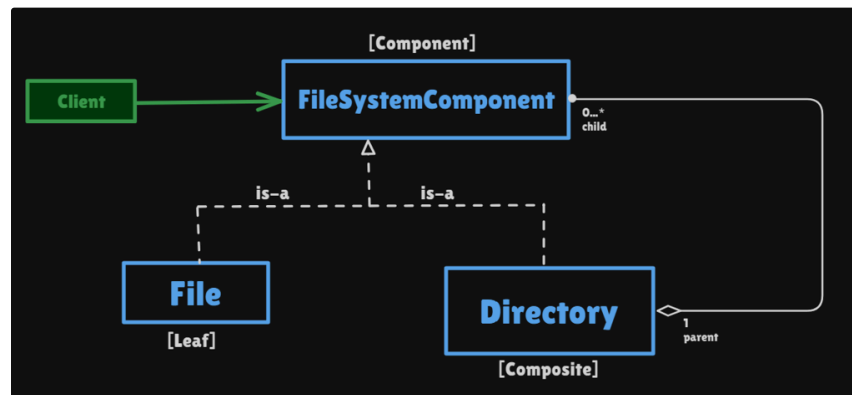
1. File Systems
2. Organizational Structures
3. Mathematical Expressions
4. Building Structures
5. Drawing applications and Computer-Aided Design (CAD)

## Class Diagram



## Structure of Composite Pattern

Understanding the Structure of Composite Pattern using File Structure Example:



- **Component ( FileSystemComponent )**: Abstract base class/Interface defines operations common to both simple (leaf) and complex (composite) objects.
- **Leaf ( File )**: Represents a leaf node(end objects) in the composition. A leaf has no children and implements the component interface directly.

- **Composite ( Directory )**: Defines the behavior for composites(components that can have children).
- **Client ( FileSystemDemo )**: Works with both component and composite objects uniformly through the Component interface.

## Implementation

### Example 1: File Structure

```

1 // Step 1: Component interface
2 public interface FileSystemComponent {
3     void printContents();
4 }

1 // Step 2: Leaf - File
2 public class File implements FileSystemComponent {
3     String fileName;
4
5     public File(String name) {
6         this.fileName = name;
7     }
8
9     @Override
10    public void printContents() {
11        System.out.println("File name: " + fileName);
12    }
13 }

1 // Step 3: Composite - Folder
2 public class Directory implements FileSystemComponent {
3     String directoryName;
4     List<FileSystemComponent> children;
5
6     public Directory(String name) {
7         this.directoryName = name;
8         children = new ArrayList<>();
9     }
10
11    public void add(FileSystemComponent fileSystemComponent) {
12        children.add(fileSystemComponent);
13    }
14
15    public void remove(FileSystemComponent fileSystemComponent) {
16        children.remove(fileSystemComponent);
17    }
18
19    @Override
20    public void printContents() {
21        System.out.println("Directory Name: " + directoryName);
22        for (FileSystemComponent child : children) {
23            child.printContents();
24        }
25    }
26 }

1 // Step 4: Client code with Composite Pattern
2 public class FileSystemDemo {
3     public static void main(String[] args) {
4         System.out.println("===== Composite Design Pattern =====");
5
6         // Create files
7         File receipt = new File("receipt.pdf");
8         File invoice = new File("invoice.pdf");
9         File torrentLinks = new File("torrentLinks.txt");
10        File tomCruise = new File("tomCruise.jpg");
11        File dumbAndDumber = new File("DumbAndDumber.mp4");
12        File hangoverI = new File("HangoverI.mp4");

```

```

13
14     // Create directories
15     Directory moviesDirectory = new Directory("Movies");
16     Directory comedyMovieDirectory = new
Directory("ComedyMovies");
17
18     // Build the tree structure hierarchically
19     moviesDirectory.add(receipt);
20     moviesDirectory.add(invoice);
21     moviesDirectory.add(torrentLinks);
22     moviesDirectory.add(tomCruise);
23     moviesDirectory.add(comedyMovieDirectory);
24     comedyMovieDirectory.add(dumbAndDumber);
25     comedyMovieDirectory.add(hangoverI);
26
27     // Display full structure
28     moviesDirectory.printContents();
29 }
30 }

```

## Example 2: Arithmetic Expressions

```

1 // Component Interface
2 public interface ArithmeticExpression {
3     int evaluate();
4 }

```

```

1 // Leaf
2 public class Number implements ArithmeticExpression {
3     int value;
4
5     public Number(int value) {
6         this.value = value;
7     }
8
9     public int evaluate() {
10         System.out.println("Number value is: " + value);
11         return value;
12     }
13 }

```

```

1 // Composite
2 public class Expression implements ArithmeticExpression {
3
4     ArithmeticExpression leftExpression;
5     ArithmeticExpression rightExpression;
6     OperationType operation;
7
8     public Expression(ArithmeticExpression leftPart,
ArithmeticExpression rightPart, OperationType operation) {
9         this.leftExpression = leftPart;
10        this.rightExpression = rightPart;
11        this.operation = operation;
12    }
13
14    public int evaluate() {
15        int value = 0;
16        switch (operation) {
17            case OperationType.ADD:
18                value = leftExpression.evaluate() +
rightExpression.evaluate();
19                break;
20            case OperationType.SUBTRACT:
21                value = leftExpression.evaluate() -
rightExpression.evaluate();
22                break;
23            case OperationType.DIVIDE:
24                value = leftExpression.evaluate() /
rightExpression.evaluate();
25                break;

```

```

26         case OperationType.MULTIPLY:
27             value = leftExpression.evaluate() *
rightExpression.evaluate();
28             break;
29         }
30
31         System.out.println("Expression value is:" + value);
32         return value;
33     }
34 }

```

```

1 // Client Code with Composite Pattern
2 public class MathExpressionEvaluator {
3     public static void main(String[] args) {
4         System.out.println("==== Composite Design Pattern =====");
5         // 2*(1+7) tree structure for evaluation
6         /*
7
8             *
9             / \
10            2  +
11               / \
12              1  7
13
14 */
15         ArithmeticExpression two = new Number(2);
16         ArithmeticExpression one = new Number(1);
17         ArithmeticExpression seven = new Number(7);
18
19         ArithmeticExpression addExpression = new Expression(one,
20             seven, OperationType.ADD);
21         ArithmeticExpression parentExpression = new Expression(two,
22             addExpression, OperationType.MULTIPLY);
23
24         System.out.println(parentExpression.evaluate());
25     }
26 }

```

```

1 public enum OperationType {
2     ADD, SUBTRACT, MULTIPLY, DIVIDE
3 }

```