Lecture 12

Dr. Umair Rehman

Agenda

- Structural Design Patterns
  - Composite Design Pattern
  - Decorator Design Pattern

Structural Design Patterns

- Adapter: Converts one interface to another compatible one.
- Facade: Simplifies a complex subsystem with a unified interface.
- Bridge: Separates abstraction from implementation.
- Composite: Treats individual and grouped objects uniformly.
- Decorator: Adds behavior to objects dynamically.
- Flyweight: Shares data to reduce memory usage.
- Proxy: Controls access or adds functionality as a substitute.

Task at Hand

- Implement customizable coffee order system
- Start with a base coffee
- Add milk, sugar, etc., dynamically
- Chain additions in any order
- Update description and cost

Inheritance

```
 1. #include <iostream>
 2. #include <string>
 3.
 4. // Base class
 5. class Coffee {
 6. public:
 7.     virtual std::string getDescription() const = 0;
 8.     virtual double getCost() const = 0;
 9. };
10.
11. // Concrete component
12. class SimpleCoffee : public Coffee {
13. public:
14.     std::string getDescription() const override {
15.         return "Simple Coffee";
16.     }
17.
18.     double getCost() const override {
19.         return 5.0;
20.     }
21. };
```

Inheritance

```
22.
23. // Concrete class for Coffee with Milk
24. class MilkCoffee : public SimpleCoffee {
25. public:
26.     std::string getDescription() const override {
27.         return SimpleCoffee::getDescription() + ", Milk";
28.     }
29.
30.     double getCost() const override {
31.         return SimpleCoffee::getCost() + 1.5;
32.     }
33. };
34.
```

Inheritance

```
34.
35. // Concrete class for Coffee with Sugar
36. class SugarCoffee : public SimpleCoffee {
37. public:
38.     std::string getDescription() const override {
39.         return SimpleCoffee::getDescription() + ", Sugar";
40.     }
41.
42.     double getCost() const override {
43.         return SimpleCoffee::getCost() + 0.5;
44.     }
45. };
46.
47. // Concrete class for Coffee with Milk and Sugar
48. class MilkAndSugarCoffee : public SimpleCoffee {
49. public:
50.     std::string getDescription() const override {
51.         return SimpleCoffee::getDescription() + ", Milk, Sugar";
52.     }
53.
54.     double getCost() const override {
55.         return SimpleCoffee::getCost() + 2.0;
56.     }
57. };
58.
```

Inheritance

```
59. int main() {
60.     SimpleCoffee simple;
61.     MilkCoffee milk;
62.     SugarCoffee sugar;
63.     MilkAndSugarCoffee milkSugar;
64.
65.     std::cout << simple.getDescription() << " : $" << simple.getCost() <<
"\n";
66.     std::cout << milk.getDescription() << " : $" << milk.getCost() << "\n";
67.     std::cout << sugar.getDescription() << " : $" << sugar.getCost() << "\n";
68.     std::cout << milkSugar.getDescription() << " : $" << milkSugar.getCost()
<< "\n";
69.
70.     return 0;
71. }
72.
```

Inheritance Cons

- Class Explosion: As combinations increase, the number of subclasses grows rapidly (e.g., MilkCoffee, SugarCoffee, MilkAndSugarCoffee).
- Inflexible: Hard to add new variations without creating new subclasses.
- Limited Composition: Behaviors are static and fixed during compile time.

Composition

```
 1. #include <iostream>
 2. #include <string>
 3.
 4. class Coffee {
 5.     std::string description = "Simple Coffee";
 6.     double cost = 5.0;
 7.
 8. public:
 9.     void addMilk() {
10.         description += ", Milk";
11.         cost += 1.5;
12.     }
13.
14.     void addSugar() {
15.         description += ", Sugar";
16.         cost += 0.5;
17.     }
18.
19.     std::string getDescription() const {
20.         return description;
21.     }
22.
23.     double getCost() const {
24.         return cost;
25.     }
26. };
27.
```

Composition

```
27.
28. int main() {
29.     Coffee coffee;
30.     coffee.addMilk();
31.     coffee.addSugar();
32.
33.     std::cout << coffee.getDescription() << " : $" << coffee.getCost() << "\n";
34.     return 0;
35. }
36.
```

Composition Cons

- Limited Reusability: Changes in behavior require altering the base class directly.
- Less Flexible: Hard to modify the composition dynamically.
- Not Scalable: Becomes cumbersome as more ingredients are added.

Switch Case or If-Else Statement

```
 1. #include <iostream>
 2. #include <string>
 3.
 4. class Coffee {
 5. public:
 6.     std::string getDescription(int type) {
 7.         switch (type) {
 8.             case 1: return "Simple Coffee";
 9.             case 2: return "Simple Coffee, Milk";
10.             case 3: return "Simple Coffee, Sugar";
11.             case 4: return "Simple Coffee, Milk, Sugar";
12.             default: return "Unknown Coffee";
13.         }
14.     }
15.
16.     double getCost(int type) {
17.         switch (type) {
18.             case 1: return 5.0;
19.             case 2: return 6.5;
20.             case 3: return 5.5;
21.             case 4: return 7.0;
22.             default: return 0.0;
23.         }
24.     }
25. };
```

Switch Case or If-Else Statement

```
27. int main() {
28.     Coffee coffee;
29.     int type = 4; // Example: Coffee with Milk and Sugar
30.
31.     std::cout << coffee.getDescription(type) << " : $" << coffee.getCost(type) << "\n";
32.     return 0;
33. }
34.
```

Switch Case or If-Else Statement

- Hard to Maintain: Adding new combinations requires modifying switch cases.

- Limited Flexibility: Cannot easily adapt to dynamic combinations at runtime.

Builder Pattern

```
#include <iostream>
#include <string>

class CoffeeBuilder {
    std::string description = "Simple Coffee";
    double cost = 5.0;

public:
    CoffeeBuilder& addMilk() {
        description += ", Milk";
        cost += 1.5;
        return *this;
    }
```

Builder Pattern

```
15.     CoffeeBuilder& addSugar() {
16.         description += ", Sugar";
17.         cost += 0.5;
18.         return *this;
19.     }
20.
21.     std::string getDescription() const {
22.         return description;
23.     }
24.
25.     double getCost() const {
26.         return cost;
27.     }
28. };
29.
30. int main() {
31.     CoffeeBuilder coffee;
32.     coffee.addMilk().addSugar();
33.
34.     std::cout << coffee.getDescription() << " : $" << coffee.getCost() << "\n";
35.     return 0;
36. }
37.
```

Builder Pattern Cons

- Limited Scalability: Works well for small numbers of additions but becomes less manageable as more options are added.
- Limited Extensibility: Adding new ingredients requires modifying the base class.
- Code Duplication: Can lead to duplicated code if multiple objects need similar additions.

Decorator Design Pattern

- Structural design pattern
- Adds behaviors dynamically
- Wraps objects for functionality
- No alteration of structure
- Alternative to subclassing

Decorator Design Pattern

```cpp
#include <iostream>
#include <string>

// Component Interface
class Coffee {
public:
    virtual ~Coffee() {}
    virtual std::string getDescription() const = 0;
    virtual double getCost() const = 0;
};

// Concrete Component
class SimpleCoffee : public Coffee {
public:
    std::string getDescription() const override {
        return "Simple Coffee";
    }

    double getCost() const override {
        return 5.0;
    }
};
```

Decorator Design Pattern

```
23.
24. // Base Decorator
25. class CoffeeDecorator : public Coffee {
26. protected:
27.     Coffee* coffee; // Traditional pointer to the wrapped object
28.
29. public:
30.     CoffeeDecorator(Coffee* coffee) : coffee(coffee) {}
31.
32.     ~CoffeeDecorator() {
33.         // Make sure to delete the wrapped coffee object
34.         delete coffee;
35.     }
36.
37.     std::string getDescription() const override {
38.         return coffee->getDescription();
39.     }
40.
41.     double getCost() const override {
42.         return coffee->getCost();
43.     }
44. };
```

Decorator Design Pattern

```
46. // Concrete Decorators
47. class MilkDecorator : public CoffeeDecorator {
48. public:
49.     MilkDecorator(Coffee* coffee) : CoffeeDecorator(coffee) {}
50.
51.     std::string getDescription() const override {
52.         return coffee->getDescription() + ", Milk";
53.     }
54.
55.     double getCost() const override {
56.         return coffee->getCost() + 1.5;
57.     }
58. };
59.
60. class SugarDecorator : public CoffeeDecorator {
61. public:
62.     SugarDecorator(Coffee* coffee) : CoffeeDecorator(coffee) {}
63.
64.     std::string getDescription() const override {
65.         return coffee->getDescription() + ", Sugar";
66.     }
67.
68.     double getCost() const override {
69.         return coffee->getCost() + 0.5;
70.     }
71. };
```

Decorator Design Pattern

```
73. // Main function to demonstrate the Decorator Pattern
74. int main() {
75.     // Create a simple coffee
76.     Coffee* coffee = new SimpleCoffee();
77.     std::cout << coffee->getDescription() << " : $" << coffee->getCost() << "\n";
78.
79.     // Add milk to the coffee
80.     coffee = new MilkDecorator(coffee);
81.     std::cout << coffee->getDescription() << " : $" << coffee->getCost() << "\n";
82.
83.     // Add sugar to the coffee
84.     coffee = new SugarDecorator(coffee);
85.     std::cout << coffee->getDescription() << " : $" << coffee->getCost() << "\n";
86.
87.     // Cleanup: delete the final coffee object, which will recursively delete its wrapped objects
88.     delete coffee;
89.
90.     return 0;
91. }
92.
```

Decorator Design Pattern Pros

- Dynamic behavior – Adds/removes features at runtime
- Open/Closed principle – Extend without modifying existing code
- Flexible combinations – Stacks decorators in any order
- Less duplication – Reusable, modular components
- Single responsibility – Each decorator is focused
- Runtime flexibility – Adapt to changing needs on the fly

Decorator Design Pattern Cons

- Complexity – Harder to understand/debug
- Order-sensitive – Behavior depends on decorator sequence
- Performance hit – Extra processing layers
- Verbose code – Nested, harder to read
- Memory management – Tricky with traditional pointers
- Interface-bound – Requires consistent interface

Composite Design Pattern

- Structural design pattern
- Treats objects and compositions uniformly
- Represents part-whole hierarchies
- Manages tree structures
- Uniform handling of leaves and composites

Composite Design Pattern

- Component – Defines common operations for composites and leaves
- Leaf – End object, no children
- Composite – Contains children, delegates operations

Composite Design Pattern

- Think of a folder structure on your computer
- Folders can contain files and/or other folders.
- Files are the leaf nodes that contain no other elements.
- Folders are the composite nodes that can contain both files and other folders.

Composite Design Pattern

- Implement file system hierarchy
- Treat files and folders uniformly
- Add, manage, display both
- Folders contain files/folders recursively
- Indented, structured output

Composite Design Pattern: 1. Define the Component Interface

```
1. #include <iostream>
 2. #include <vector>
 3. #include <string>
 4.
 5. // Component Interface
 6. // Defines the common interface for both files and folders
 7. class FileSystemComponent {
 8. public:
 9.     virtual ~FileSystemComponent() {}
10.
11.     // Common operation to display details of files or folders
12.     virtual void showDetails(int indent = 0) const = 0;
13. };
14.
15. // Helper function to print indentation
16. // Used for displaying hierarchical structure clearly
17. void printIndent(int indent) {
18.     for (int i = 0; i < indent; ++i) {
19.         std::cout << "  ";  // Print two spaces for each level of indentation
20.     }
21. }
22.
```

Composite Design Pattern: 2. Create the Leaf Class

```
23. // Leaf Class: Represents a file in the file system
24. class File : public FileSystemComponent {
25. private:
26.     std::string name;  // Name of the file
27. 
28. public:
29.     // Constructor to initialize the file name
30.     File(const std::string& name) : name(name) {}
31. 
32.     // Display the file details with indentation
33.     void showDetails(int indent = 0) const override {
34.         printIndent(indent);  // Print indentation
35.         std::cout << "File: " << name << "\n";  // Display file name
36.     }
37. };
38. 
```

Composite Design Pattern: 3. Create the Composite Class

```
39. // Composite Class: Represents a folder in the file system
40. class Folder : public FileSystemComponent {
41. private:
42.     std::string name;  // Name of the folder
43.     std::vector<FileSystemComponent*> children;  // Child components (files or folders)
44.
45. public:
46.     // Constructor to initialize the folder name
47.     Folder(const std::string& name) : name(name) {}
48.
49.     // Destructor to delete all child components and prevent memory leaks
50.     ~Folder() {
51.         for (auto* child : children) {
52.             delete child;  // Delete each child component
53.         }
54.     }
55.
```

Composite Design Pattern: 3. Create the Composite Class

```
56.     // Add a child component (file or folder) to the folder
57.     void add(FileSystemComponent* component) {
58.         children.push_back(component);  // Add to the list of children
59.     }
60.
61.     // Display the folder details and recursively display its children
62.     void showDetails(int indent = 0) const override {
63.         printIndent(indent);  // Print indentation
64.         std::cout << "Folder: " << name << "\n";  // Display folder name
65.
66.         // Iterate over all children and display their details
67.         for (const auto* child : children) {
68.             child->showDetails(indent + 1);  // Recursive call with increased
indentation
69.         }
70.     }
71. };
```

Composite Design Pattern: 4. Use the Composite Pattern

```
73. // Main function to demonstrate the Composite Pattern
74. int main() {
75.     // Create file objects (leaf nodes)
76.     File* file1 = new File("File1.txt");  // File1
77.     File* file2 = new File("File2.txt");  // File2
78.     File* file3 = new File("File3.txt");  // File3
79.
80.     // Create folder objects (composite nodes)
81.     Folder* folder1 = new Folder("Folder1");  // Folder1
82.     Folder* folder2 = new Folder("Folder2");  // Folder2
83.     Folder* rootFolder = new Folder("Root");  // Root folder
84.
85.     // Build the hierarchy
86.     folder1->add(file1);           // Add File1 to Folder1
87.     folder2->add(file2);           // Add File2 to Folder2
88.     folder1->add(folder2);         // Add Folder2 to Folder1
89.     rootFolder->add(folder1);      // Add Folder1 to Root folder
90.     rootFolder->add(file3);        // Add File3 to Root folder
91.
92.     // Display the entire hierarchy
93.     std::cout << "File System Structure:\n";
94.     rootFolder->showDetails();  // Display the root folder hierarchy
95.
96.     // Cleanup: Delete the root folder, which recursively deletes all children
97.     delete rootFolder;
98.
99.     return 0;
100. }
101.
```

Composite Design Pattern Uses

- GUI hierarchies: Window contains panels, panels contain buttons, etc.
- File systems: Folders contain files and other folders.
- Menus: A menu can contain menu items and submenus.
- Organizations: Departments contain employees and sub-departments.

Composite Design Pattern Pros

- Uniformity: Treats individual objects and composites the same way.
- Scalability: Allows easy addition or removal of leaf and composite objects.
- Flexibility: Simplifies client code by allowing it to interact with the structure in a consistent manner.

Composite Design Pattern Cons

- Complexity: Can make the code more complex, especially when the hierarchy becomes deep.
- Performance: Operations on large structures can be slow due to recursion over all nodes.
- Potential Overuse: If not used appropriately, it can lead to an unnecessarily complicated design.

Composite Design Pattern

- Ideal for tree structures
- Treats nodes uniformly
- Scalable, flexible design
- Abstracts part-whole hierarchies

Composite vs Decorator Design Pattern:

- Decorator – Adds behavior dynamically
- Composite – Manages tree hierarchies

- Decorator – Focuses on individual objects
- Composite – Handles groups and leaves uniformly

- Decorator – Works in flat structures
- Composite – Works in nested structures

- Decorator – Extends functionality
- Composite – Composes parts into wholes

Conclusion

- Structural Design Patterns
  - Composite Design Pattern
  - Decorator Design Pattern