Lecture 13

Dr. Umair Rehman

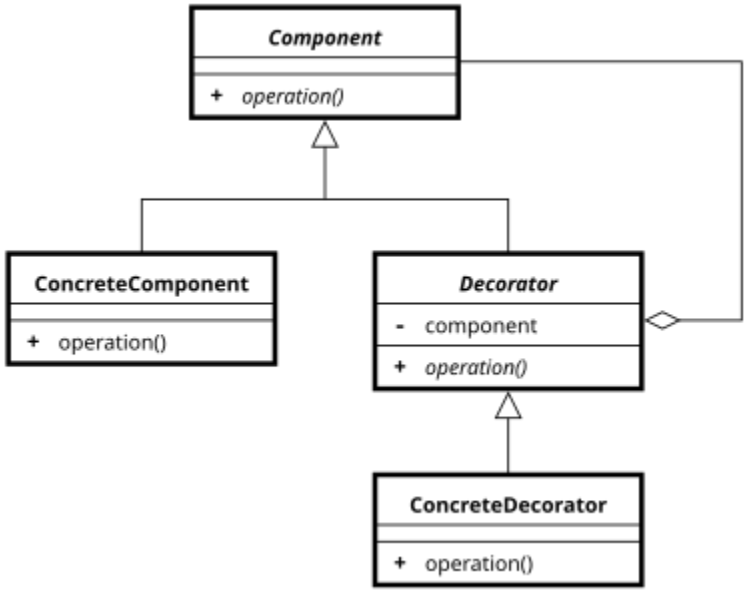
Agenda

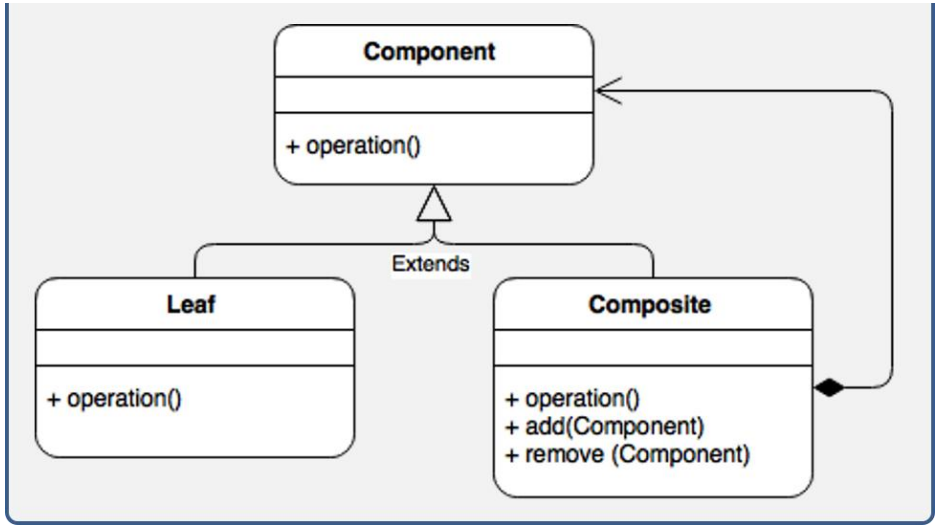
- Structural Design Patterns
  - Facade Design Pattern
  - Proxy Design Pattern

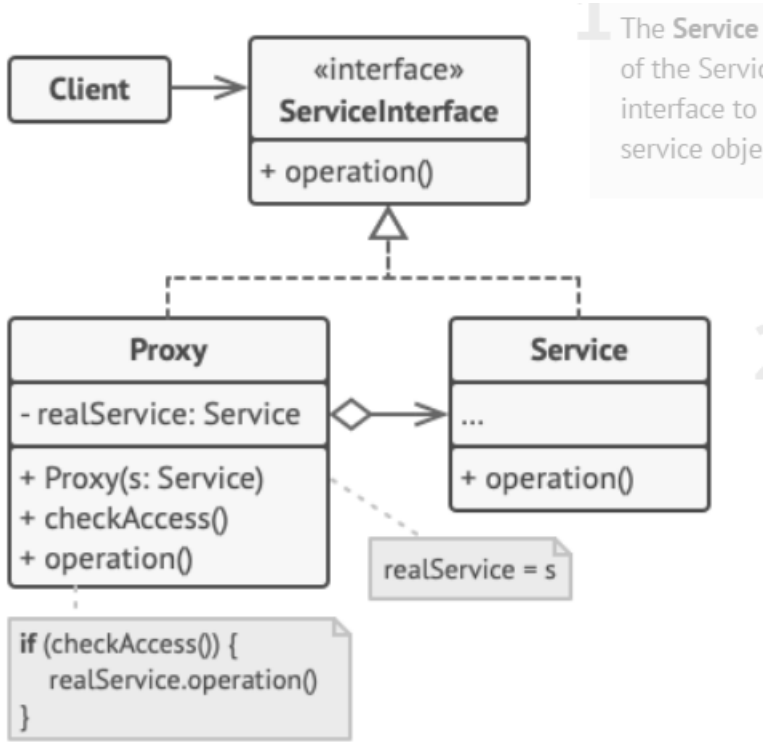
Structural Design Patterns

- 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.

Comparisons Pattern







https://refactoring.guru/design-patterns/proxy

Proxy Pattern

- Provide a placeholder or surrogate for another object to control access to it.

Types of Proxy

- Virtual Proxy: Manages lazy initialization.
- Protection Proxy: Controls access based on permissions.
- Remote Proxy: Represents an object on a different network/location.
- Caching Proxy: Stores results to improve performance.
- Logging Proxy: Logs interactions for debugging or monitoring.

Components

- Proxy: Controls access to the real object, adding features like caching, logging, or access control.
- Real Subject: Performs the actual work or holds the data.
- Client: Interacts with the proxy, as if it's the real object.

Task

- Suppose you have an image viewer that loads images from disk.
- Loading high-resolution images can be resource-intensive
- Use the proxy design pattern that can be used to delay the loading until the image is actually needed.

Structure of Proxy Pattern

- Client: Requests image display.
- Proxy: Checks if the image is in memory; loads from disk if not.
- Real Image: Loads and renders image data from disk.

Proxy Pattern

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

// Interface: Abstract base class for image operations
class Image {
public:
    virtual void display() const = 0;  // Pure virtual method
    virtual ~Image() = default;        // Virtual destructor
};

// Real Subject: Represents the high-resolution image
class RealImage : public Image {
private:
    std::string filename;

    void loadFromDisk() const {
        std::cout << "Loading image from disk: " << filename << std::endl;
    }

public:
    RealImage(const std::string& file) : filename(file) {
        loadFromDisk();  // Loads the image when the object is created
    }

    void display() const override {
        std::cout << "Displaying image: " << filename << std::endl;
    }
};

```

Proxy Pattern

```
29.
30. // Proxy: Controls access to the real image
31. class ProxyImage : public Image {
32. private:
33.     std::string filename;
34.     RealImage* realImage = nullptr;
35.
36. public:
37.     ProxyImage(const std::string& file) : filename(file) {}
38.
39.     ~ProxyImage() {
40.         delete realImage;
41.     }
42.
43.     void display() const override {
44.         if (realImage == nullptr) {
45.             realImage = new RealImage(filename);  // Load image if not already loaded
46.         }
47.         realImage->display();
48.     }
49. };
```

Proxy Pattern

```
50.
51. // Client Code
52. int main() {
53.     Image* image = new ProxyImage("high_resolution_photo.jpg");
54.     std::cout << "Image created, but not loaded from disk yet.\n";
55.
56.     std::cout << "Now displaying the image:\n";
57.     image->display();  // Loads from disk at this point
58.
59.     std::cout << "Displaying the image again:\n";
60.     image->display();  // Already loaded, no disk access
61.
62.     delete image;
63.     return 0;
64. }
65.
```

Lazy Initialization

- When the client calls `display()` on the proxy, it checks if `realImage` is null.

- If so, the proxy creates `realImage`
  - Ensuring it's only created when needed, conserving resources.

Façade Design Pattern

- Provides a simplified interface to a complex subsystem
  - A set of interfaces in a software system.

- Acts as a wrapper that hides the complexity of the subsystem
  - Exposes a more user-friendly interface to the client.

Purpose of the Facade Pattern

- The main purpose is to simplify interactions with complex systems by:
  - Providing a single-entry point for the client.
  - Hiding the complexities of subsystems.
  - Making the subsystem easier to use without revealing internal details.
  - Decoupling the client from the implementation details of the subsystem.

Facade Pattern

- Like a universal remote for a home theater
  - offers a simple interface (e.g., "Play Movie")
  - managing complex operations across devices (TV, speakers, etc.).

- The user (client) interacts only with the remote, not the individual devices.

Key Concepts

- Subsystem Classes: Handle specific functions, often requiring multiple steps or configurations.

- Facade Class: Simplifies access by wrapping subsystem classes
  - Offering easy-to-use methods that manage complexities internally.

Task

- Home Theater System: Includes DVD player, projector, sound system, and popcorn machine.
- Without Facade: Operate each subsystem individually to start movie night.
- Use the Façade Patten: Simplified interface starts everything in one step.

Façade Design Pattern

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

// Subsystem 1: DVD Player
class DVDPlayer {
public:
    void on() {
        std::cout << "DVD Player: On" << std::endl;
    }
    void play(const std::string& movie) {
        std::cout << "DVD Player: Playing movie " << movie << std::endl;
    }
};

// Subsystem 2: Projector
class Projector {
public:
    void on() {
        std::cout << "Projector: On" << std::endl;
    }
    void setInput(const std::string& input) {
        std::cout << "Projector: Set input to " << input << std::endl;
    }
};
```

Façade Design Pattern

```
25.
26. // Subsystem 3: Sound System
27. class SoundSystem {
28. public:
29.     void on() {
30.         std::cout << "Sound System: On" << std::endl;
31.     }
32.     void setVolume(int level) {
33.         std::cout << "Sound System: Set volume to " << level << std::endl;
34.     }
35. };
36.
37. // Subsystem 4: Popcorn Machine
38. class PopcornMachine {
39. public:
40.     void on() {
41.         std::cout << "Popcorn Machine: On" << std::endl;
42.     }
43.     void pop() {
44.         std::cout << "Popcorn Machine: Popping popcorn!" << std::endl;
45.     }
46. };
```

Façade Design Pattern

```
48. // Facade: Home Theater Facade
49. class HomeTheaterFacade {
50. private:
51.     DVDPlayer* dvdPlayer;
52.     Projector* projector;
53.     SoundSystem* soundSystem;
54.     PopcornMachine* popcornMachine;
55.
80.
```

Façade Design Pattern

```
56. public:
57. // Initializes all subsystems by taking pointers to each subsystem as
parameters.Assigns these pointers to its member variables.

58.     HomeTheaterFacade(DVDPlayer* dvd, Projector* proj, SoundSystem* sound,
PopcornMachine* popcorn)
59.         : dvdPlayer(dvd), projector(proj), soundSystem(sound),
popcornMachine(popcorn) {}
60.
```

Façade Design Pattern

```
60.  // Simplified method to watch a movie; invoking methods on objects
62.     void watchMovie(const std::string& movie) {
63.         std::cout << "Get ready to watch a movie..." << std::endl;
64.         popcornMachine->on();
65.         popcornMachine->pop();
66.         soundSystem->on();
67.         soundSystem->setVolume(10);
68.         projector->on();
69.         projector->setInput("DVD");
70.         dvdPlayer->on();
71.         dvdPlayer->play(movie);
72.     }

73.
74.     // Simplified method to end the movie
75.     void endMovie() {
76.         std::cout << "Shutting down the home theater..." << std::endl;
77.         // Turn off all subsystems (in real scenario, it would be more detailed)
78.     }
79. };
80.
```

Façade Design Pattern

```
81. // Client code
82. int main() {
83.     // Create instances of subsystems
84.     DVDPlayer dvdPlayer;
85.     Projector projector;
86.     SoundSystem soundSystem;
87.     PopcornMachine popcornMachine;
88.
89.     // Create a facade for the home theater
90.     HomeTheaterFacade homeTheater(&dvdPlayer, &projector, &soundSystem,
&popcornMachine);
91.
92.     // Use the facade to watch a movie
93.     homeTheater.watchMovie("Inception");
94.
95.     return 0;
96. }
97.
```

Key Point

- The client interacts only with the facade and is unaware of the complexity hidden behind it.

Key Benefits

- Simplification:
  - Single point of access to manage the home theater system.
- Encapsulation:
  - Hides the complexity of interacting with multiple subsystems.
- Convenience:
  - Allows users to start and stop a movie with simple method calls
  - Without worrying about the individual subsystems.

When to Use the Facade Pattern

- Simplify Complex Systems:
  - Need of a simpler interface for interacting with complex subsystems.
- Decoupling:
  - Separates the client from subsystems
  - Allowing subsystem changes without affecting the client.
- Reduce Dependencies:
  - Minimizes client-subsystem dependencies
  - Improving maintainability and extensibility.

Comparison

- Adapter vs. Facade: Adapter converts interfaces; Facade simplifies an existing one.

- Proxy vs. Facade: Proxy controls access; Facade focuses on ease of use.

Conclusion

- Structural Design Patterns
  - Facade Design Pattern
  - Proxy Design Pattern