Lecture 14

Dr. Umair Rehman

Agenda

- Structural Design Patterns
    - Bridge Design Pattern
    - Flyweight Design Pattern

- Combination of Patterns
    - Adapter + Composite
    - Decorator + Composite
    - Façade + Adapter
    - Proxy + Flyweight

Bridge Pattern

- The Bridge Pattern is a structural design pattern

- Decouples an abstraction from its implementation, allowing them to vary independently.

- This pattern is useful when you want to separate a class's interface from its implementation
  - So that either can be modified or extended without affecting the other.

The Bridge Pattern Components

- Abstraction: High-level interface for operations, unaware of concrete implementations.
- Implementor: Interface or base class defining the implementation layer.
- Concrete Implementor: Implements the Implementor interface, providing specific functionality.
- Refined Abstraction: Extends Abstraction and uses Implementor for operations.

Problem Setup

- Let's create a RemoteControl class that will act as the abstraction and a Device class that will act as the implementor.

- Then, we'll have specific devices like TV and Radio that are the concrete implementors.

Example Code

```cpp
// Implementor Interface
class Device {
public:
    virtual void powerOn() = 0;   // Pure virtual function for turning on the device
    virtual void powerOff() = 0;  // Pure virtual function for turning off the device
    virtual ~Device() {}          // Virtual destructor for safe deletion of derived objects
};
```

Example Code

```cpp
// Concrete Implementor 1: TV Device
class TV : public Device {
public:
    void powerOn() override {
        std::cout << "TV is now ON." << std::endl;
    }

    void powerOff() override {
        std::cout << "TV is now OFF." << std::endl;
    }
};

```

Example Code

```cpp
// Concrete Implementor 2: Radio Device
class Radio : public Device {
public:
    void powerOn() override {
        std::cout << "Radio is now ON." << std::endl;
    }

    void powerOff() override {
        std::cout << "Radio is now OFF." << std::endl;
    }
};
```

Example Code

```
// Abstraction
class RemoteControl {
protected:
    Device* device; // Pointer to the implementor

public:
    // Constructor accepts a pointer to a device implementor
    RemoteControl(Device* dev) : device(dev) {}

    // Virtual destructor for safe deletion of derived objects
    virtual ~RemoteControl() {}

    // High-level operation: turn the device on
    virtual void turnOn() {
        device->powerOn();
    }

    // High-level operation: turn the device off
    virtual void turnOff() {
        device->powerOff();
    }
};

```

Example Code

```cpp
// Refined Abstraction
class AdvancedRemoteControl : public RemoteControl {
public:
    // Constructor calls base class constructor with device pointer
    AdvancedRemoteControl(Device* dev) : RemoteControl(dev) {}

    // Additional operation for advanced remote
    void mute() {
        std::cout << "Muting the device." << std::endl;
    }
};

```

Example Code

```cpp
int main() {
    // Create TV and Radio devices
    Device* tv = new TV();
    Device* radio = new Radio();

    // Create a simple remote control for the TV
    RemoteControl* simpleRemote = new RemoteControl(tv);
    simpleRemote->turnOn();   // Output: TV is now ON.
    simpleRemote->turnOff();  // Output: TV is now OFF.

    // Create an advanced remote control for the radio
    AdvancedRemoteControl* advancedRemote = new AdvancedRemoteControl(radio);
    advancedRemote->turnOn();    // Output: Radio is now ON.
    advancedRemote->mute();      // Output: Muting the device.
    advancedRemote->turnOff();   // Output: Radio is now OFF.

    // Clean up dynamically allocated objects
    delete simpleRemote;
    delete advancedRemote;
    delete tv;
    delete radio;

    return 0;
}
```

Bridge Pattern Advantages

- Decouples abstraction & implementation – Independent evolution.
- Improves maintainability – Cleaner, modular code.
- Enhances flexibility – Easy to add new variations.
- Platform independence – Consistent interface across platforms.
- Supports Open/Closed Principle – Extend without modifying existing code.

Bridge Pattern Disadvantages

- Increased complexity – More classes and interfaces.
- Higher upfront design effort – Requires careful planning.
- May be overkill – Not ideal for simple systems.
- Harder to debug – More layers to trace through.
- Can affect performance – Multiple indirections may slow down execution.

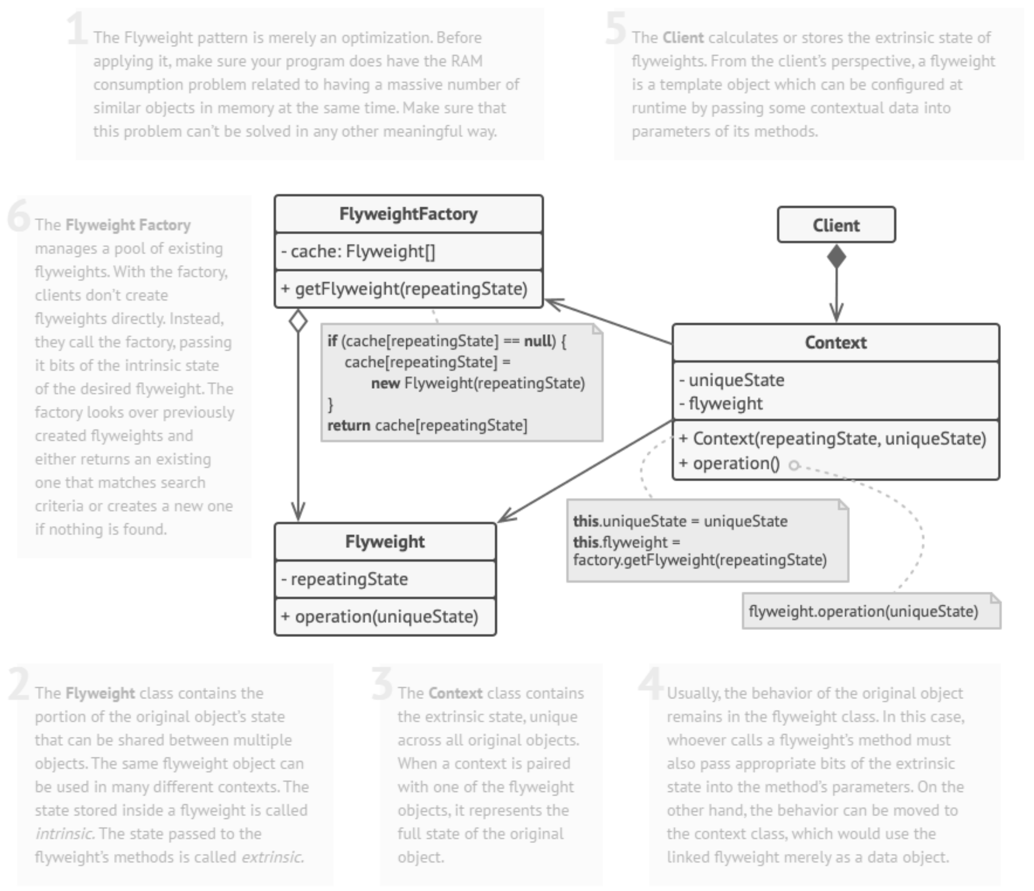
Flyweight Pattern

- Minimizes memory by sharing data among similar objects, ideal for large, similar objects with slight variations.

The Flyweight Pattern Components

- Flyweight: Interface for common operations using extrinsic state.
- Concrete Flyweight: Implements shared, reusable intrinsic state.
- Flyweight Factory: Manages and shares Flyweight instances.
- Client: Holds extrinsic state and uses Flyweights via the factory.

Flyweight Pattern



1 The Flyweight pattern is merely an optimization. Before applying it, make sure your program does have the RAM consumption problem related to having a massive number of similar objects in memory at the same time. Make sure that this problem can't be solved in any other meaningful way.

5 The **Client** calculates or stores the extrinsic state of flyweights. From the client's perspective, a flyweight is a template object which can be configured at runtime by passing some contextual data into parameters of its methods.

6 The **Flyweight Factory** manages a pool of existing flyweights. With the factory, clients don't create flyweights directly. Instead, they call the factory, passing it bits of the intrinsic state of the desired flyweight. The factory looks over previously created flyweights and either returns an existing one that matches search criteria or creates a new one if nothing is found.

```
if (cache[repeatingState] == null) {
    cache[repeatingState] =
        new Flyweight(repeatingState)
}
return cache[repeatingState]
```

```
this.uniqueState = uniqueState
this.flyweight =
factory.getFlyweight(repeatingState)
```

```
flyweight.operation(uniqueState)
```

2 The **Flyweight** class contains the portion of the original object's state that can be shared between multiple objects. The same flyweight object can be used in many different contexts. The state stored inside a flyweight is called *intrinsic*. The state passed to the flyweight's methods is called *extrinsic*.

3 The **Context** class contains the extrinsic state, unique across all original objects. When a context is paired with one of the flyweight objects, it represents the full state of the original object.

4 Usually, the behavior of the original object remains in the flyweight class. In this case, whoever calls a flyweight's method must also pass appropriate bits of the extrinsic state into the method's parameters. On the other hand, the behavior can be moved to the context class, which would use the linked flyweight merely as a data object.

Simpler Use Case: Problem Setup

- Lets use the Flyweight Pattern to represent characters in a document editor.
- Here, each character type (like 'A', 'B', etc.) shares properties such as the font and size, while the position on the document is unique for each character.

Example Code

```cpp
#include <iostream>
#include <unordered_map>

// Flyweight: Character Type, representing intrinsic state (shared properties like font and size)
class Character {
public:
    Character(char symbol, const std::string& font, int size)
        : symbol_(symbol), font_(font), size_(size) {}

    // Method to display the character at a given position
    void display(int x, int y) const {
        std::cout << "Displaying character '" << symbol_ << "' at position ("
                  << x << ", " << y << ") with font " << font_
                  << " and size " << size_ << "." << std::endl;
    }

private:
    char symbol_;         // Intrinsic state: character symbol
    std::string font_;    // Intrinsic state: font style
    int size_;            // Intrinsic state: font size
};

```

Example Code

```
// Flyweight Factory to manage Character instances
class CharacterFactory {
public:
    // Get an existing Character or create a new one if it doesn't exist
    Character* getCharacter(char symbol, const std::string& font, int size) {
        // Create a unique key based on the symbol, font, and size
        std::string key = std::string(1, symbol) + "_" + font + "_" + std::to_string(size);

        // Check if the character already exists in the map
        if (characters_.find(key) == characters_.end()) {
            // If not found, create a new Character and add it to the map
            characters_[key] = new Character(symbol, font, size);
            std::cout << "Creating new character '" << symbol << "' with font "
                      << font << " and size " << size << "." << std::endl;
        }

        // Return the existing or newly created Character
        return characters_[key];
    }

```

Example Code

```
    // Clean up all allocated Character objects
    ~CharacterFactory() {
        for (auto& pair : characters_) {
            delete pair.second;
        }
    }

private:
    std::unordered_map<std::string, Character*> characters_; // Map to store unique Characters
};

```

Example Code

```cpp
// Client code
int main() {
    CharacterFactory factory;

    // Request the same character with the same font and size multiple times
    Character* charA = factory.getCharacter('A', "Arial", 12);
    Character* charB = factory.getCharacter('B', "Arial", 12);
    Character* charA2 = factory.getCharacter('A', "Arial", 12); // Should reuse existing 'A' character

    // Display the characters at different positions
    charA->display(10, 20);   // Displaying character 'A' at position (10, 20) with font Arial and size 12.
    charB->display(15, 25);   // Displaying character 'B' at position (15, 25) with font Arial and size 12.
    charA2->display(30, 40);  // Displaying character 'A' at position (30, 40) with font Arial and size 12.

    return 0;
}

```

Advantages

- Memory Efficiency: Reduces memory by sharing objects.
- Performance Boost: Fewer objects improve speed.
- Consistency: Shared state ensures uniformity.
- Modularity: Separates intrinsic and extrinsic state.
- Scalability: Supports large volumes with minimal overhead.
- Flexibility: Easily vary extrinsic state without duplication.

Disadvantages

- Complexity: Adds layers for managing states.
- Intrusive: Requires separation of intrinsic/extrinsic states.
- Limited Use: Only beneficial for large, similar objects.
- Maintenance: Harder to track shared vs. unique data.
- Debugging: Extra indirection complicates tracing issues.

Agenda

- Structural Design Patterns
  - Bridge Design Pattern
  - Flyweight Design Pattern