Contents

Facade Pattern

What is the Facade Pattern?

The **Facade Pattern** is a **Structural Design Pattern** that provides a **simplified interface** to a complex system of classes, libraries, or subsystems.
Instead of interacting with multiple components individually, clients can interact with a **single facade class**, which acts as a gateway to the system.

Intent:

- Provide a **unified interface** to a set of interfaces in a subsystem.
- Simplify the use of a complex system by **hiding the complexity** behind a single facade.

Key Features:

1. Simplifies client interaction with a system.
2. Decouples the client code from the subsystem's implementation.
3. Provides a **high-level interface** while keeping the subsystem flexible.

Analogy: Hotel Reception

Imagine a **hotel reception**:

- Guests interact with **one receptionist** to:
  - Book a room.
  - Order food.
  - Request laundry service.
  - Ask for transportation.
- Guests don't need to know how individual departments like housekeeping, kitchen, or travel services operate.

Here, the **receptionist** is the **facade** that simplifies the interaction.

When to Use the Facade Pattern

1. **When a system is complex** and you need to simplify its usage.
2. **When you want to decouple clients** from subsystem implementations.
3. **When you need to provide a high-level entry point** for a large or evolving system.

Structure of Facade Pattern

Components:

1. **Facade Class**:
   - Provides a simplified interface to the subsystem.
2. **Subsystem Classes**:
   - Perform the actual work but are hidden from the client.
3. **Client**:
   - Uses the facade to interact with the system without dealing with its complexity.

Class Diagram:

```
mermaid
```

```
Copy code
classDiagram
    class Client {
        +operation()
    }

    class Facade {
        +simplifiedOperation()
    }

    class SubsystemA {
        +operationA()
    }

    class SubsystemB {
        +operationB()
    }

    class SubsystemC {
        +operationC()
    }

    Client --> Facade : Uses
    Facade --> SubsystemA : Delegates
    Facade --> SubsystemB : Delegates
    Facade --> SubsystemC : Delegates
```

Example of the Facade Pattern

Consider a **Home Theater System** with multiple components:

- DVD Player
- Amplifier
- Projector
- Lights
- Screen

Using these individually can be cumbersome for a user. Instead, a **HomeTheaterFacade** class simplifies the usage.

C++ Implementation

```
cpp
Copy code
#include <iostream>
using namespace std;

// Subsystem Classes
class DVDPlayer {
public:
    void on() { cout << "DVD Player is ON." << endl; }
    void play() { cout << "Playing DVD..." << endl; }
    void off() { cout << "DVD Player is OFF." << endl; }
};

class Amplifier {
public:
    void on() { cout << "Amplifier is ON." << endl; }
    void setVolume(int level) { cout << "Amplifier volume set to " << level
<< "." << endl; }
    void off() { cout << "Amplifier is OFF." << endl; }
};

class Projector {
public:
    void on() { cout << "Projector is ON." << endl; }
    void wideScreenMode() { cout << "Projector in widescreen mode." << endl;
}
    void off() { cout << "Projector is OFF." << endl; }
};

class Lights {
public:
    void dim() { cout << "Lights are dimmed." << endl; }
};

// Facade Class
class HomeTheaterFacade {
    DVDPlayer dvd;
    Amplifier amp;
    Projector proj;
    Lights lights;

public:
    void watchMovie() {
        cout << "Setting up your movie experience..." << endl;
        lights.dim();
        proj.on();
        proj.wideScreenMode();
        amp.on();
        amp.setVolume(5);
        dvd.on();
        dvd.play();
    }

    void endMovie() {
        cout << "Shutting down your movie experience..." << endl;
        dvd.off();
        amp.off();
        proj.off();
    }
};

// Client Code
int main() {
    HomeTheaterFacade homeTheater;
    homeTheater.watchMovie();
    cout << "Movie time!" << endl;
    homeTheater.endMovie();
    return 0;
}
```

Output:

```
vbnet
Copy code
Setting up your movie experience...
Lights are dimmed.
Projector is ON.
Projector in widescreen mode.
Amplifier is ON.
Amplifier volume set to 5.
DVD Player is ON.
Playing DVD...
Movie time!
Shutting down your movie experience...
DVD Player is OFF.
Amplifier is OFF.
Projector is OFF.
```

How Facade Simplifies the System

1. **Client Interaction**:
   - The client only interacts with the **HomeTheaterFacade** class.
2. **Subsystem Complexity**:
   - The client does not need to know the details of how the projector, amplifier, and DVD player work.

Advantages of Facade Pattern

1. **Simplifies Usage**:
   - Provides a single, high-level interface to a complex system.
2. **Decouples Client from Subsystems**:
   - The client does not directly interact with subsystems, making the system easier to maintain.
3. **Improves Maintainability**:
   - Changes to subsystem classes do not affect the client as long as the facade remains consistent.
4. **Improves Readability**:
   - Code becomes cleaner and more organized.

Disadvantages of Facade Pattern

1. **Limited Flexibility**:
   - The facade might restrict access to some advanced functionalities of the subsystems.
2. **Over-Simplification**:
   - If the subsystem is already simple, introducing a facade may add unnecessary overhead.

Real-World Software Examples

1. **Web Application Frameworks**:
   - Frameworks like **Spring Boot** in Java provide a facade to interact with complex systems like databases, security, and configurations.

2. **Operating System File Systems**:
    - The file explorer in Windows or macOS acts as a facade to interact with file operations like copying, renaming, or deleting files.
3. **Payment Gateways**:
    - Payment gateway libraries provide a single interface for processing payments while managing multiple underlying systems (e.g., card validation, fraud checks).

When to Use Facade Pattern

- When you want to **simplify client interaction** with a complex system.
- When you need to **decouple the client** from the subsystems.
- When the system evolves, and you want to provide a **consistent interface** to clients.

Conclusion

The **Facade Pattern**:

- Provides a simple interface to a complex system.
- Improves usability, maintainability, and code readability.
- Acts as a "front desk" for clients to access a system without getting bogged down by its complexity.

Iterator Pattern

What is the Iterator Pattern?

The **Iterator Pattern** is a **Behavioral Design Pattern** that provides a way to **access the elements of a collection** sequentially **without exposing the underlying representation** of the collection.

Intent:

- To provide a standard way of **iterating over a collection** (like arrays, lists, trees).
- Decouple the iteration logic from the collection class.
- Allow multiple ways of traversing a collection.

Key Features:

1. Provides a **uniform interface** to iterate over different types of collections.
2. Encapsulates the logic for **traversing** a collection.
3. Allows **multiple iterators** to work independently on the same collection.

Analogy: Reading a Book

- Think of **reading a book**:
    - You don't need to know how the book is structured (pages, binding, etc.).
    - You just **turn the pages one by one** using an iterator (like your hand or a bookmark).

Here, the **book** is the collection, and the **hand or bookmark** is the iterator.

When to Use the Iterator Pattern

1. When you need to traverse a collection **without exposing its structure**.
2. When multiple traversal mechanisms are needed (e.g., forward, backward).
3. When you want to standardize the way to access collections, regardless of their type.

Structure of the Iterator Pattern

Components:

1. **Iterator** (Interface):
   - Defines the standard methods for iteration (e.g., `next()`, `hasNext()`).
2. **Concrete Iterator**:
   - Implements the `Iterator` interface to traverse a specific collection.
3. **Aggregate** (Collection Interface):
   - Defines the interface for creating an iterator.
4. **Concrete Aggregate**:
   - Implements the aggregate interface and returns a specific iterator.
5. **Client**:
   - Uses the iterator to access elements of the collection.

Class Diagram:

```
mermaid
Copy code
classDiagram
    class Iterator {
        <<interface>>
        + hasNext() bool
        + next() Element
    }
    class ConcreteIterator {
        + hasNext() bool
        + next() Element
    }
    class Aggregate {
        <<interface>>
        + createIterator() Iterator
    }
    class ConcreteAggregate {
        + createIterator() Iterator
    }
    class Client {
        + operation()
    }

    Iterator <|-- ConcreteIterator
    Aggregate <|-- ConcreteAggregate
    Client --> Iterator
    ConcreteAggregate --> ConcreteIterator : Creates
```

Example Implementation in C++

Let’s implement the **Iterator Pattern** to iterate over a list of integers.

Code Example:

cpp
Copy code
```
#include <iostream>
#include <vector>
using namespace std;

// Iterator Interface
class Iterator {
public:
    virtual bool hasNext() = 0;
    virtual int next() = 0;
};

// Concrete Iterator
class ListIterator : public Iterator {
private:
    vector<int>& collection; // Reference to the collection
    int index;

public:
    ListIterator(vector<int>& col) : collection(col), index(0) {}

    bool hasNext() override {
        return index < collection.size();
    }

    int next() override {
        return collection[index++];
    }
};

// Aggregate Interface
class Aggregate {
public:
    virtual Iterator* createIterator() = 0;
};

// Concrete Aggregate (Collection)
class List : public Aggregate {
private:
    vector<int> collection;

public:
    void add(int value) {
        collection.push_back(value);
    }

    Iterator* createIterator() override {
        return new ListIterator(collection);
    }
};

// Client Code
int main() {
    List myList;
    myList.add(10);
    myList.add(20);
    myList.add(30);

    Iterator* it = myList.createIterator();

    cout << "Elements in the collection:" << endl;
    while (it->hasNext()) {
        cout << it->next() << " ";
    }

    delete it; // Free memory
    return 0;
}
```

Output:

```
yaml
Copy code
Elements in the collection:
10 20 30
```

Explanation:

1. **Iterator Interface**:
   - Defines methods `hasNext()` and `next()` for iteration.
2. **Concrete Iterator**:
   - `ListIterator` implements the interface and traverses the vector.
3. **Aggregate**:
   - Defines a method `createIterator()` to return an iterator.
4. **Concrete Aggregate**:
   - `List` implements `createIterator()` and holds the collection.
5. **Client**:
   - Uses the iterator to traverse the collection without knowing its internal structure.

Advantages of the Iterator Pattern

1. **Encapsulation**:
   - The client does not need to know the **internal structure** of the collection.
2. **Flexibility**:
   - Multiple iterators can be used for the same collection.
3. **Standardization**:
   - Provides a **common interface** for iterating different types of collections.
4. **Supports Polymorphism**:
   - Iterators work uniformly on lists, trees, graphs, etc.

Disadvantages of the Iterator Pattern

1. Adds **additional classes** and complexity for simple collections.
2. Requires additional memory for managing the iterator’s state.

Real-World Applications of the Iterator Pattern

1. **STL Iterators in C++**:

- The Standard Template Library (STL) in C++ uses iterators to traverse containers like `vector`, `list`, and `map`.

2. **Java Collections Framework**:
    - Java's `Iterator` interface provides methods like `hasNext()` and `next()` for traversing collections.
3. **File Systems**:
    - Iterators are used to traverse files in a directory structure.
4. **Database Cursors**:
    - Cursors in SQL allow you to traverse records in a result set one by one.

Key Points to Remember

- The **Iterator Pattern** provides a **standardized way** to traverse collections.
- It **decouples** the traversal logic from the collection's internal structure.
- Widely used in libraries, file systems, and databases.

Summary

The Iterator Pattern:

- Makes collections **easier to traverse**.
- Provides a **uniform interface** for iteration.
- Encourages encapsulation and flexibility in design.

Decorator Pattern

What is the Decorator Pattern?

The **Decorator Pattern** is a **Structural Design Pattern** that allows you to **dynamically add responsibilities or behaviors** to an object **at runtime** without modifying its structure.

Intent:

- Attach additional responsibilities to an object dynamically.
- Avoid subclassing by "decorating" the object with new functionality.

Key Features:

1. **Dynamic Behavior**:
    - You can add behavior **at runtime** without changing the class structure.
2. **Avoid Subclass Explosion**:
    - Instead of creating multiple subclasses, combine decorators to add different features.
3. **Flexible Composition**:
    - Decorators can be combined in different ways to achieve various results.

Analogy: Coffee Shop

Think of a **coffee shop**:

- You start with a basic coffee.
- You can add **milk**, **sugar**, or **chocolate syrup** to customize it.
- Each "add-on" is a **decorator** that adds functionality (flavor) to the base object.

Here:

- **Coffee** is the base object.
- **Milk, sugar, and syrup** are decorators.

When to Use the Decorator Pattern

1. When you need to add or change behavior **at runtime**.
2. When subclassing is impractical due to a large number of combinations of features.
3. When you want to keep classes **open for extension** but **closed for modification** (Open/Closed Principle).

Structure of the Decorator Pattern

Components:

1. **Component** (Interface or Abstract Class):
   - Defines the interface for objects that can have responsibilities added dynamically.
2. **Concrete Component**:
   - The base class to which additional behaviors will be added.
3. **Decorator**:
   - Implements the component interface and adds behavior to the object.
4. **Concrete Decorators**:
   - Specific classes that add new functionality to the component.

Class Diagram:

```
mermaid
Copy code
classDiagram
    class Component {
        <<interface>>
        +operation() void
    }

    class ConcreteComponent {
        +operation() void
    }

    class Decorator {
        +operation() void
    }

    class ConcreteDecoratorA {
        +operation() void
    }

    class ConcreteDecoratorB {
        +operation() void
    }

    Component <|-- ConcreteComponent
    Component <|-- Decorator
    Decorator <|-- ConcreteDecoratorA
    Decorator <|-- ConcreteDecoratorB
    Decorator --> Component : Wraps
```

Example: Decorator Pattern in C++

Let's implement a simple **coffee shop** example using the Decorator Pattern.

Code Example:

```
cpp
Copy code
#include <iostream>
#include <string>
using namespace std;

// Component Interface
class Coffee {
public:
    virtual string getDescription() = 0;
    virtual double cost() = 0;
    virtual ~Coffee() = default;
};

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

    double cost() override {
        return 2.0; // Basic coffee costs $2
    }
};

// Decorator Class
class CoffeeDecorator : public Coffee {
protected:
    Coffee* decoratedCoffee;

public:
    CoffeeDecorator(Coffee* coffee) : decoratedCoffee(coffee) {}

    string getDescription() override {
        return decoratedCoffee->getDescription();
    }

    double cost() override {
        return decoratedCoffee->cost();
    }
};

// Concrete Decorators
class MilkDecorator : public CoffeeDecorator {
public:
    MilkDecorator(Coffee* coffee) : CoffeeDecorator(coffee) {}

    string getDescription() override {
        return decoratedCoffee->getDescription() + ", Milk";
    }

    double cost() override {
        return decoratedCoffee->cost() + 0.5; // Add $0.5 for milk
    }
};

class SugarDecorator : public CoffeeDecorator {
public:
    SugarDecorator(Coffee* coffee) : CoffeeDecorator(coffee) {}

    string getDescription() override {
        return decoratedCoffee->getDescription() + ", Sugar";
    }

    double cost() override {
        return decoratedCoffee->cost() + 0.2; // Add $0.2 for sugar
    }
};

class ChocolateDecorator : public CoffeeDecorator {
public:
    ChocolateDecorator(Coffee* coffee) : CoffeeDecorator(coffee) {}

    string getDescription() override {
        return decoratedCoffee->getDescription() + ", Chocolate";
    }

    double cost() override {
        return decoratedCoffee->cost() + 0.7; // Add $0.7 for chocolate
    }
};

// Client Code
int main() {
    Coffee* myCoffee = new SimpleCoffee();
    cout << myCoffee->getDescription() << " -> Cost: $" << myCoffee->cost()
<< endl;

    myCoffee = new MilkDecorator(myCoffee);
    cout << myCoffee->getDescription() << " -> Cost: $" << myCoffee->cost()
<< endl;

    myCoffee = new SugarDecorator(myCoffee);
    cout << myCoffee->getDescription() << " -> Cost: $" << myCoffee->cost()
<< endl;

    myCoffee = new ChocolateDecorator(myCoffee);
    cout << myCoffee->getDescription() << " -> Cost: $" << myCoffee->cost()
<< endl;

    // Clean up memory
    delete myCoffee;

    return 0;
}
```

Output:

```bash
bash
Copy code
Simple Coffee -> Cost: $2
Simple Coffee, Milk -> Cost: $2.5
Simple Coffee, Milk, Sugar -> Cost: $2.7
Simple Coffee, Milk, Sugar, Chocolate -> Cost: $3.4
```

Explanation:

1. **SimpleCoffee**:
   - The base component providing basic coffee.
2. **Decorators**:
   - `MilkDecorator`, `SugarDecorator`, and `ChocolateDecorator` wrap around the coffee to add functionality (milk, sugar, chocolate) **dynamically**.
3. **Client**:
   - Combines decorators at runtime to build customized coffee.

Advantages of the Decorator Pattern

1. **Dynamic Behavior**:
   - Responsibilities can be added **at runtime** without altering existing code.
2. **Flexible Composition**:
   - Decorators can be **stacked** in different combinations.
3. **Open/Closed Principle**:
   - Existing classes are **closed for modification** but **open for extension**.

Disadvantages of the Decorator Pattern

1. **Complexity**:
   - Adding many decorators can lead to complex and difficult-to-debug code.
2. **Multiple Objects**:
   - Each decorator creates a new object, which increases object creation overhead.

Real-World Applications

1. **Java I/O Library**:
   - Java's `BufferedReader`, `InputStream`, and `DataInputStream` use the Decorator Pattern to add functionalities like buffering and reading different data types.
2. **UI Frameworks**:
   - Adding features like scrollbars, borders, or shadows to UI components.
3. **Text Processing**:
   - Wrapping plain text with decorators for formatting, encryption, or compression.

Summary

The **Decorator Pattern**:

- Adds **new behaviors** to an object dynamically.
- Avoids subclassing by using **composition**.

- Allows for flexible and reusable code while following the **Open/Closed Principle**.