Index

Problem No: 01
Problem Name: Implementation of Factory Method Design Pattern

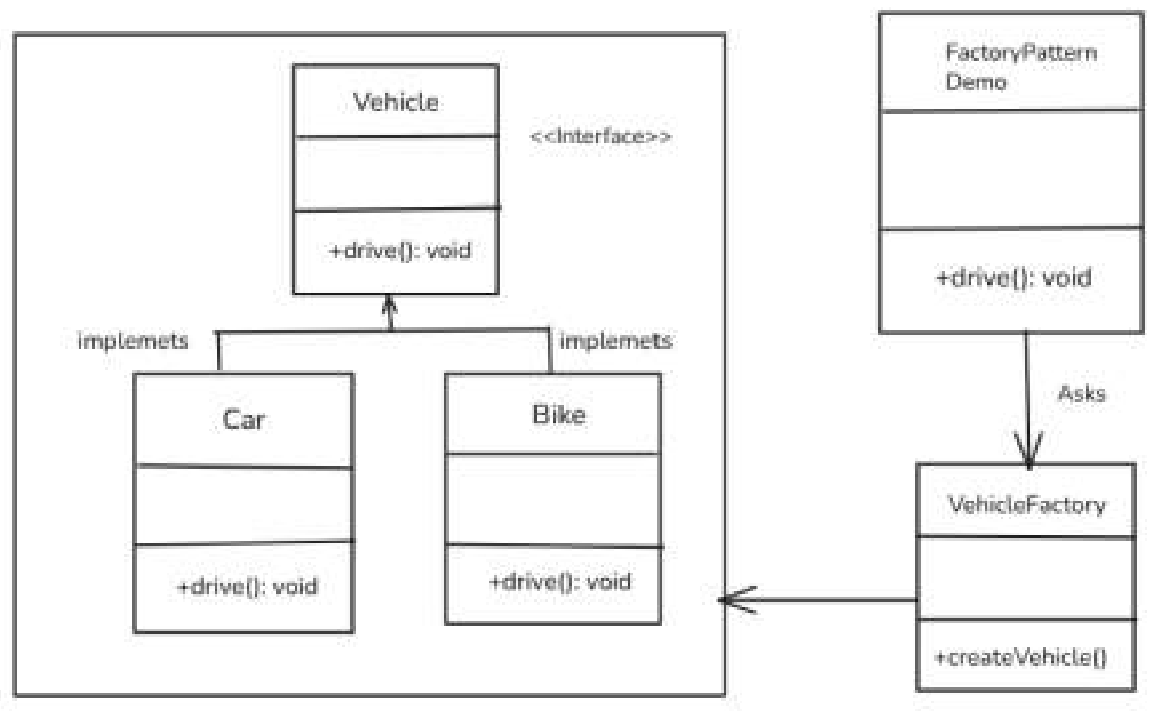
Objectives:

- To understand the Factory Method design pattern.
- To implement the Factory Method pattern using C++/Java.
- To design a class structure that allows object creation based on type without modifying existing code.

Theory:
The Factory Method Pattern is a creational design pattern that provides an interface for creating objects but allows subclasses to alter the type of objects that will be created. This helps in achieving loose coupling and better scalability.

Consider a vehicle manufacturing company that produces different types of vehicles like cars, bikes, and trucks. Instead of creating different objects manually, a Factory Method can be used to create vehicle objects dynamically based on a given type.

UML Diagram:



Programme Implementation (C++)

```
#include <iostream>
#include <memory>
using namespace std;

// Step 1: Abstract Product
class Vehicle {
public:
   virtual void drive() = 0;
   virtual ~Vehicle() {}
};

// Step 2: Concrete Products
class Car : public Vehicle {
public:
   void drive() override { cout << "Driving a Car!" << endl; }
};

class Bike : public Vehicle {
public:
   void drive() override { cout << "Riding a Bike!" << endl; }
};

// Step 3: Factory Class
class VehicleFactory {
public:
   static unique_ptr<Vehicle> createVehicle(const string& type) {
      if (type == "Car") return make_unique<Car>();
      else if (type == "Bike") return make_unique<Bike>();
      else return nullptr;
   }
};

// Step 4: Client Code
int main() {
   unique_ptr<Vehicle> v1 = VehicleFactory::createVehicle("Car");
   if (v1) v1->drive();

   unique_ptr<Vehicle> v2 = VehicleFactory::createVehicle("Bike");
   if (v2) v2->drive();

   return 0;
```

Result:

```
Driving a Car!
Riding a Bike!
```

Discussion:

1. The Factory Method successfully creates different objects dynamically based on the given type.
2. This implementation follows Open-Closed Principle, allowing us to add more vehicle types without modifying existing code.
3. It promotes code reusability and maintainability.

Applications:

- **Software Development:** Used in frameworks to create objects dynamically without exposing object creation logic.
- **Game Development:** Helps in creating different game characters, weapons, or enemies based on player choices.
- **Banking Systems:** Used to generate different types of bank accounts or transactions based on user input.
- **Operating Systems:** Used in GUI applications where different types of buttons, windows, or dialogs are created dynamically.

Conclusion:

The Factory Method Pattern is an efficient way to handle object creation dynamically. It provides flexibility, scalability, and better maintainability by ensuring that object creation logic is encapsulated within a factory class.

Problem No: 02
Problem Name: Implementation of Abstract Factory Method Design Pattern

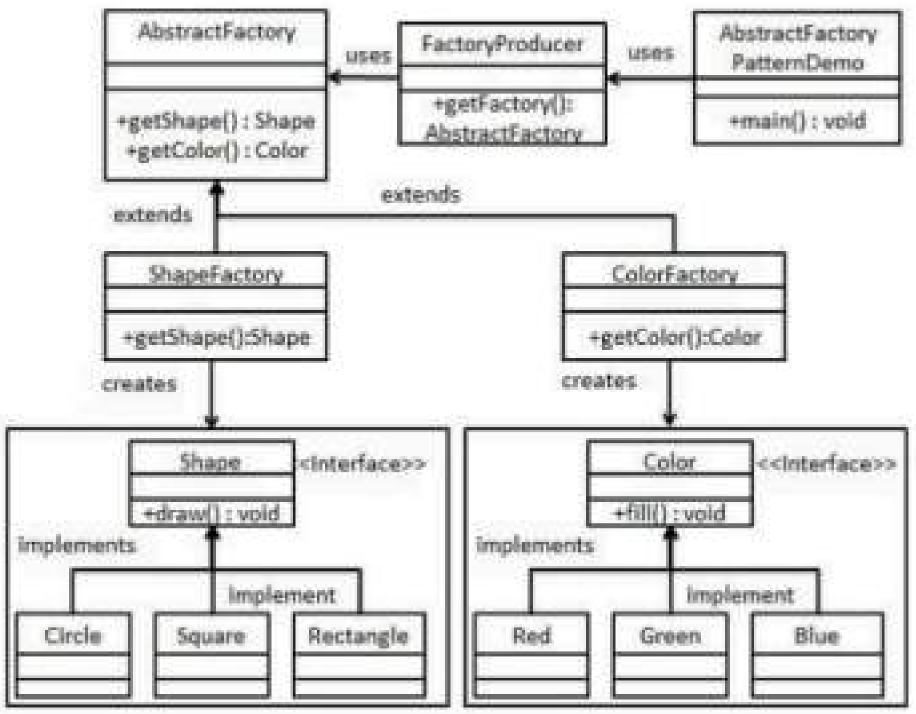
Objectives:

- To understand the Abstract Factory design pattern.
- To implement the Abstract Factory pattern using C++/Java.
- To design a class structure that allows the creation of families of related objects without specifying their concrete classes.

Theory:
The Abstract Factory Pattern is a creational design pattern that provides an interface for creating families of related or dependent objects without specifying their concrete classes. It helps in ensuring loose coupling and enhances scalability.

Consider a graphic design application where users can create shapes (like circles, squares, rectangles) and assign them colors (red, green, blue). Instead of hardcoding object creation, an Abstract Factory can dynamically generate shapes and colors based on the user's selection.

UML Diagram:



Program Implementation (C++)

```
#include <iostream>
#include <memory>
using namespace std;

// Step 1: Abstract Product Interfaces
```

```
class Shape {
public:
    virtual void draw() = 0;
    virtual ~Shape() {}
};

class Color {
public:
    virtual void fill() = 0;
    virtual ~Color() {}
};

// Step 2: Concrete Products
class Circle : public Shape {
public:
    void draw() override { cout << "Drawing a Circle!" << endl; }
};

class Square : public Shape {
public:
    void draw() override { cout << "Drawing a Square!" << endl; }
};

class Rectangle : public Shape {
public:
    void draw() override { cout << "Drawing a Rectangle!" << endl; }
};

class Red : public Color {
public:
    void fill() override { cout << "Filling with Red color!" << endl; }
};

class Green : public Color {
public:
    void fill() override { cout << "Filling with Green color!" << endl; }
};

class Blue : public Color {
public:
    void fill() override { cout << "Filling with Blue color!" << endl; }
};

// Step 3: Abstract Factory
class AbstractFactory {
public:
    virtual unique_ptr<Shape> getShape(const string& type) { return nullptr; }
```

```
  virtual unique_ptr<Color> getColor(const string& type) { return nullptr; }
  virtual ~AbstractFactory() {}
};

// Step 4: Concrete Factories
class ShapeFactory : public AbstractFactory {
public:
  unique_ptr<Shape> getShape(const string& type) override {
     if (type == "Circle") return make_unique<Circle>();
     else if (type == "Square") return make_unique<Square>();
     else if (type == "Rectangle") return make_unique<Rectangle>();
     else return nullptr;
  }
};

class ColorFactory : public AbstractFactory {
public:
  unique_ptr<Color> getColor(const string& type) override {
     if (type == "Red") return make_unique<Red>();
     else if (type == "Green") return make_unique<Green>();
     else if (type == "Blue") return make_unique<Blue>();
     else return nullptr;
  }
};

// Step 5: Factory Producer
class FactoryProducer {
public:
  static unique_ptr<AbstractFactory> getFactory(const string& choice) {
     if (choice == "Shape") return make_unique<ShapeFactory>();
     else if (choice == "Color") return make_unique<ColorFactory>();
     else return nullptr;
  }
};

// Step 6: Client Code
int main() {
  unique_ptr<AbstractFactory> shapeFactory =
  FactoryProducer::getFactory("Shape"); unique_ptr<Shape> shape1 =
  shapeFactory->getShape("Circle");
  shape1->draw();

  unique_ptr<AbstractFactory> colorFactory =
  FactoryProducer::getFactory("Color"); unique_ptr<Color> color1 =
  colorFactory->getColor("Red");
  color1->fill();

  return 0;
}
```

Result and Discussion:

```
Drawing a Circle!
Filling with Red color!
```

- The Abstract Factory Method successfully creates different objects dynamically based on the selected factory.
- This implementation follows the Open-Closed Principle, allowing easy addition of new shapes and colors without modifying existing code.
- It promotes code reusability and maintainability by separating object creation logic from the client code.

Applications:

- GUI Toolkits: Used to create platform-independent UI components like buttons and themes.
- Game Development: Helps in dynamically generating characters, backgrounds, and skins.
- Database Drivers: Allows applications to switch between different database engines without changing core logic.
- Cloud Services: Helps in selecting appropriate services (AWS, Azure, Google Cloud) dynamically.

Conclusion:

The Abstract Factory Pattern is useful when dealing with multiple families of related objects. It enhances code modularity, ensures scalability, and maintains flexibility in software development.

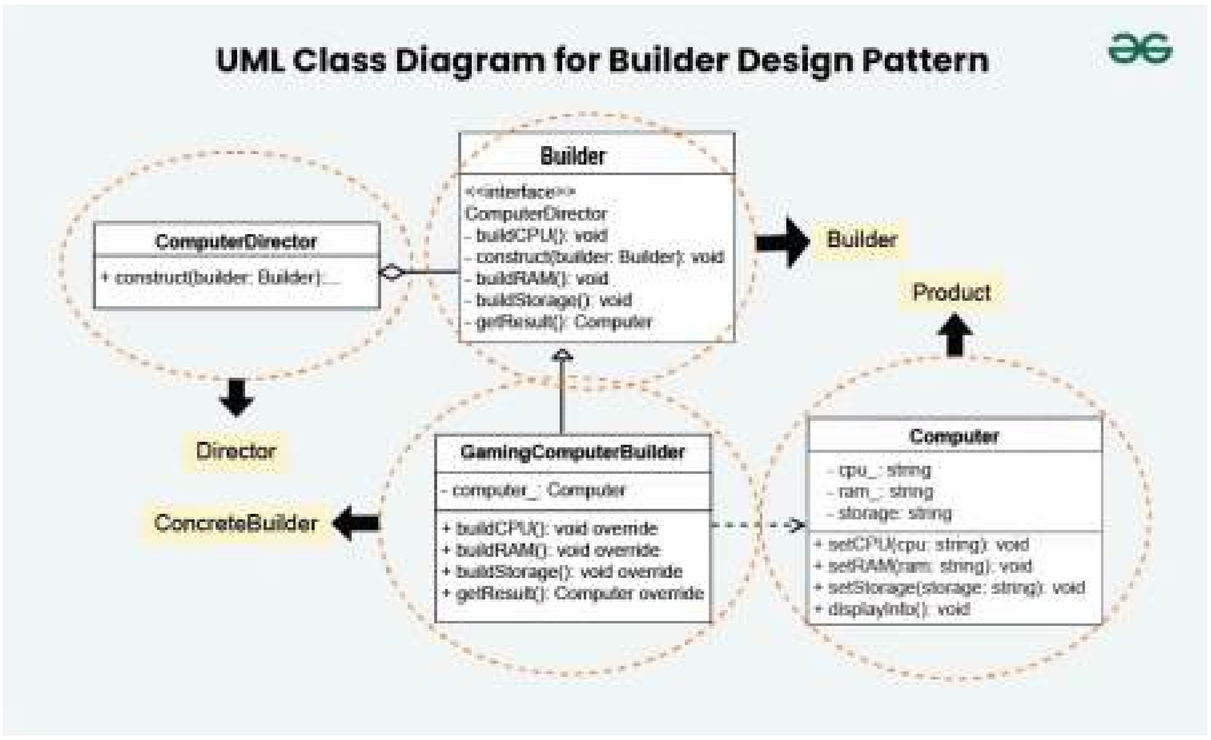
Problem No: 03
Problem Name: Implement and design Builder design pattern(for design use UML/user_defined class).

Objectives:

- To understand the Builder Design Pattern.
- To implement the Builder pattern using C++.
- To design a class structure that allows constructing complex objects step-by-step.

Theory:
The Builder Pattern is a creational design pattern used to construct complex objects step-by-step. It allows constructing objects by specifying only necessary attributes while keeping the construction process separate from the object representation. Consider a computer builder where users can choose different configurations of CPU, RAM, and Storage. Instead of creating multiple constructors, the Builder Pattern helps in constructing a customized computer configuration step-by-step.

UML Class Design:


UML Class Diagram for Builder Design Pattern
Builder
<<interface>>
ComputerDirector
- buildCPU(): void
- construct(builder: Builder): void
- buildRAM(): void
- buildStorage(): void
- getResult(): Computer
ComputerDirector
+ construct(builder: Builder):...
Builder
Product
Director
ConcreteBuilder
GamingComputerBuilder
- computer_: Computer
+ buildCPU(): void override
+ buildRAM(): void override
+ buildStorage(): void override
+ getResult(): Computer override
Computer
- cpu_: string
- ram_: string
- storage: string
+ setCPU(cpu: string): void
+ setRAM(ram: string): void
+ setStorage(storage: string): void
+ displayInfo(): void

Implementation(C++)

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

using namespace std;

// Product
class Computer {
public:
    void setCPU(const std::string& cpu) {
        cpu_ = cpu;
    }

    void setRAM(const std::string& ram) {
        ram_ = ram;
    }

    void setStorage(const std::string& storage) {
        storage_ = storage;
    }

    void displayInfo() const {
        std::cout << "Computer Configuration:"
                  << "\nCPU: " << cpu_
                  << "\nRAM: " << ram_
                  << "\nStorage: " << storage_ << "\n\n";
    }

private:
    string cpu_;
    string ram_;
    string storage_;
};

// Builder interface
class Builder {
public:
    virtual void buildCPU() = 0;
    virtual void buildRAM() = 0;
    virtual void buildStorage() = 0;
    virtual Computer getResult() = 0;
};

// ConcreteBuilder
class GamingComputerBuilder : public Builder {
private:
    Computer computer_;

public:
    void buildCPU() override {
```

```
        computer_.setCPU("Gaming CPU");
    }

    void buildRAM() override {
        computer_.setRAM("16GB DDR4");
    }

    void buildStorage() override {
        computer_.setStorage("1TB SSD");
    }

    Computer getResult() override {
        return computer_;
    }
};

// Director
class ComputerDirector {
public:
    void construct(Builder& builder) {
        builder.buildCPU();
        builder.buildRAM();
        builder.buildStorage();
    }
};

// Client
int main() {
    GamingComputerBuilder gamingBuilder;
    ComputerDirector director;

    director.construct(gamingBuilder);
    Computer gamingComputer = gamingBuilder.getResult();

    gamingComputer.displayInfo();

    return 0;
}
```

Result and Discussion:

Expected Output:

```
Computer Configuration:
CPU: Intel i9
RAM: 32GB
Storage: 1TB SSD
```

● The Builder Pattern successfully constructs a complex object step-by-step. ● It ensures flexibility and separation of concerns.
● The director takes responsibility for constructing objects without exposing object creation details.

Conclusion:

The Builder Design Pattern is useful when an object needs to be built in multiple steps. It provides a flexible, scalable approach to object creation, ensuring code maintainability and clarity.

Problem No: 04

Problem Name: Implement and design Singleton design pattern.

Objectives:

To understand the Singleton Design Pattern and its purpose. • To implement the Singleton Design Pattern using a class. • To demonstrate the benefits of the Singleton pattern, such as ensuring only one instance of a class exists throughout the program.

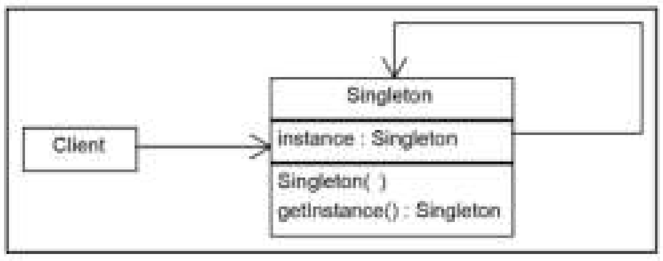
Theory:

The Singleton pattern restricts the instantiation of a class to one single object. It ensures that the class has only one instance and provides a global access point to that instance.

Real-life Example:

Database Connection: Often in applications, a database connection should be shared. If there were multiple instances of a database connection, it could lead to issues like resource wastage, inconsistent state, etc. The Singleton pattern ensures that only one database connection is created and used throughout the program.

UML/User-defined Class Design:



Program (Java):

```
public class Singleton {
 private static Singleton instance;
 private Singleton() {
 // Initialization code
 }
 public static Singleton getInstance() {
 if (instance == null) {
 instance = new Singleton();
 }
 return instance;
 }
 public void displayMessage() {
 System.out.println("Hello from Singleton!");
 }
 public static void main(String[] args) {
 // Getting the single instance of the Singleton class
 Singleton singleton = Singleton.getInstance();
 singleton.displayMessage();
Singleton anotherInstance = Singleton.getInstance();
System.out.println("Are both instances the same? " + (singleton == anotherInstance));
}
}
```

Result and Discussion:

- The output of the displayMessage() method confirms that the Singleton class is functioning correctly.
- It will also print a message confirming that both singleton and anotherInstance refer to the same object.
- The program demonstrates the Singleton Design Pattern, ensuring that only one instance of the Singleton class is created.
- The use of the getInstance() method prevents the creation of multiple objects of the class, which is the main goal of the Singleton pattern. • The key advantage of the Singleton pattern is that it provides a global

  point of access to the instance, which is particularly useful when managing shared resources like database connections, loggers, etc. • The use of the private constructor and static getInstance() method ensures that the object is lazily initialized and only created when needed.

Problem No: 05

Problem Name: Design and Implement the Adapter Design Pattern.

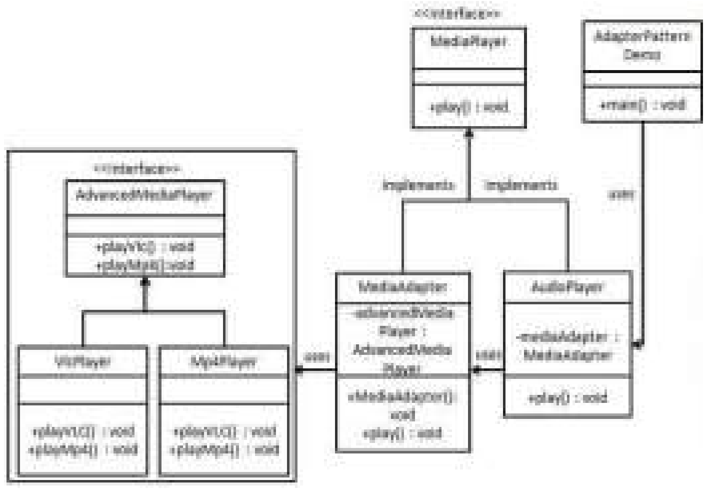
Objectives:

To understand the Adapter Design Pattern and its purpose. To implement the Adapter Design Pattern using a class in an object oriented language. To demonstrate how the Adapter pattern helps to convert one interface to another, making two incompatible interfaces compatible.

Theory:

The Adapter pattern is used when a class (or system) needs to interact with another class (or system) whose interface is not compatible. It provides a way to adapt the interface of one class so it can be used by another class without modifying the source code.

UML/User-defined Class Design:



Program (Java):

```
interface MediaPlayer {
 void play(String audioType, String fileName);
}
class AudioPlayer {
 public void playAudio(String fileName) {
 System.out.println("Playing audio file: " + fileName);
 }
}
class AudioAdapter implements MediaPlayer {
 private AudioPlayer audioPlayer;

 public AudioAdapter() {
 audioPlayer = new AudioPlayer();
 }

 @Override
 public void play(String audioType, String fileName) {
 if (audioType.equalsIgnoreCase("mp3")) {
 audioPlayer.playAudio(fileName);
 } else {
 System.out.println("Invalid audio type. Only MP3 supported.");
 }
 }
}
class MediaPlayerClient {
 public static void main(String[] args) {
 MediaPlayer player = new AudioAdapter();
 player.play("mp3", "song.mp3");
 player.play("mp4", "movie.mp4");
 }
}
```

Result and Discussion:

- The Adapter pattern allows us to make incompatible systems work together by converting the interface of one class to be compatible with the client's expectations.
- This pattern is useful when we want to integrate existing code or third party libraries without changing the original classes. Instead, an adapter class is created to translate method calls from one format to another.
- In this example, the AudioPlayer class could be an existing class that we cannot modify. By using the AudioAdapter, we can adapt it to the MediaPlayer interface without changing the AudioPlayer class itself.

Problem No. 6
Problem Name: Implement and design Bridge design pattern(for design use UML/user_defined class)

Objective:

The objective of this lab is to understand and implement the Bridge Design Pattern. The pattern is used to decouple abstraction from implementation so that both can vary independently. By doing so, we achieve flexibility, scalability, and reusability in software design.

Theory:

The Bridge Pattern is a structural design pattern that separates abstraction (high-level control classes) from implementation (low-level operations). It avoids creating multiple class hierarchies for every combination of abstraction and implementation.

Key Components:

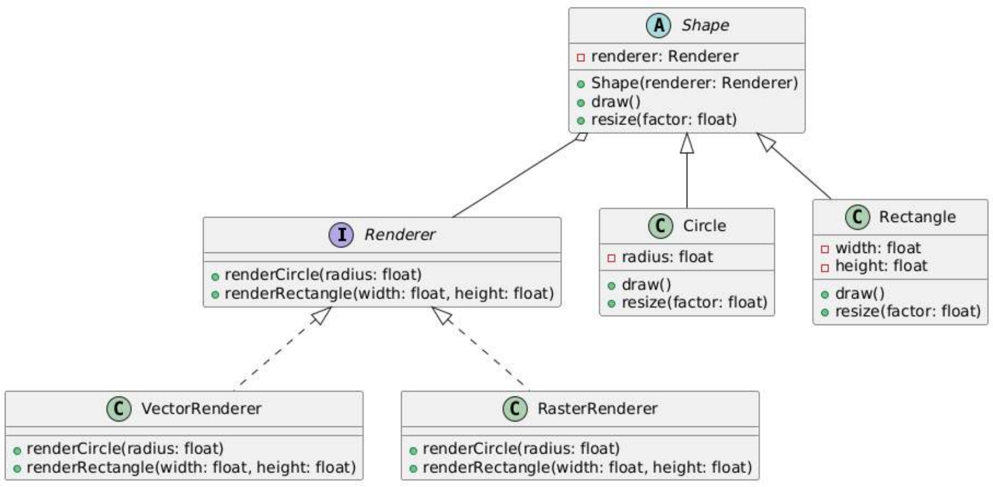
1. Abstraction: Defines the abstraction's interface and contains a reference to the implementor.
2. Refined Abstraction: Extends the abstraction with more specialized behavior.
3. Implementor: Defines the interface for implementation classes.
4. Concrete Implementor: Provides actual implementations.

Application:

Bridge pattern is widely used in scenarios where:

- Multiple orthogonal dimensions vary (e.g., Shape and Renderer).
- Platforms or devices differ (e.g., RemoteControl abstraction with TV or Radio implementation).
- You want to switch implementations at runtime without changing client code.

UML Design:


A Shape
renderer: Renderer
Shape(renderer: Renderer)
draw()
resize(factor: float)
I Renderer
renderCircle(radius: float)
renderRectangle(width: float, height: float)
C Circle
radius: float
draw()
resize(factor: float)
C Rectangle
width: float
height: float
draw()
resize(factor: float)
C VectorRenderer
renderCircle(radius: float)
renderRectangle(width: float, height: float)
C RasterRenderer
renderCircle(radius: float)
renderRectangle(width: float, height: float)

Implementation in C++:

```
#include <iostream>
protected:
Renderer& renderer;
public:
Shape(Renderer& r) : renderer(r) {}
virtual void draw() = 0;
virtual void resize(float factor) = 0;
virtual ~Shape() = default;
};

// Refined Abstraction - Circle
class Circle : public Shape {
float radius;
public:
Circle(Renderer& r, float rad) : Shape(r), radius(rad) {}
void draw() override {
renderer.renderCircle(radius);
}
void resize(float factor) override {
radius *= factor;
}
};

// Refined Abstraction - Rectangle
class Rectangle : public Shape {
float width, height;
public:
Rectangle(Renderer& r, float w, float h) : Shape(r), width(w), height(h) {}
void draw() override {
renderer.renderRectangle(width, height);
}
void resize(float factor) override {
width *= factor;
height *= factor;
}
};

// Demo
int main() {
VectorRenderer vr;
RasterRenderer rr;

Circle c(vr, 5.0f);
Rectangle rect(rr, 3.0f, 4.0f);

c.draw();
rect.draw();

c.resize(2.0f);
c.draw();
```

```
rect.resize(0.5f);
rect.draw();

return 0;
}
```

Result Discussion:

Sample Output:

```
Drawing a circle as vectors with radius 5
Drawing pixels for a rectangle with width=3, height=4
Drawing a circle as vectors with radius 10
Drawing pixels for a rectangle with width=1.5, height=2
```

Discussion:

- The circle first draws with radius 5 using a vector renderer.
- The rectangle is drawn with a raster renderer.
- After resizing, the circle doubles in size and the rectangle reduces to half.
- The abstraction (Shape) and implementation (Renderer) are fully decoupled, showing flexibility.

Conclusion:

The Bridge Design Pattern allows separation of concerns by decoupling abstraction from implementation. In this lab:

- We modeled shapes as abstractions and renderers as implementations.
- We demonstrated flexibility in mixing any shape with any renderer.
- We achieved runtime configurability and easier extensibility.

Thus, Bridge pattern is a powerful tool for creating scalable and maintainable designs where abstraction and implementation can evolve independently.

Problem No: 07

Problem Name: Implement and design Decorator design pattern (for design use UML/user_defined class)

Objective:

The objective of this lab is to understand and implement the Decorator Design Pattern. The pattern is used to add responsibilities to objects dynamically without modifying their base code. It promotes code reusability and adherence to the Open-Closed Principle.

Theory:

The Decorator Pattern is a structural design pattern that allows behavior to be added to individual objects, dynamically, without affecting the behavior of other objects from the same class. Instead of creating many subclasses, decorators provide a flexible alternative to extend functionality.

Key Components:

1. **Component:** Defines the interface for objects.
2. **ConcreteComponent:** The base implementation of the component.
3. **Decorator:** Maintains a reference to a component object and implements the component interface.
4. **ConcreteDecorator:** Adds new responsibilities dynamically to the component.

This pattern is useful for designing flexible and extensible systems.

Application:

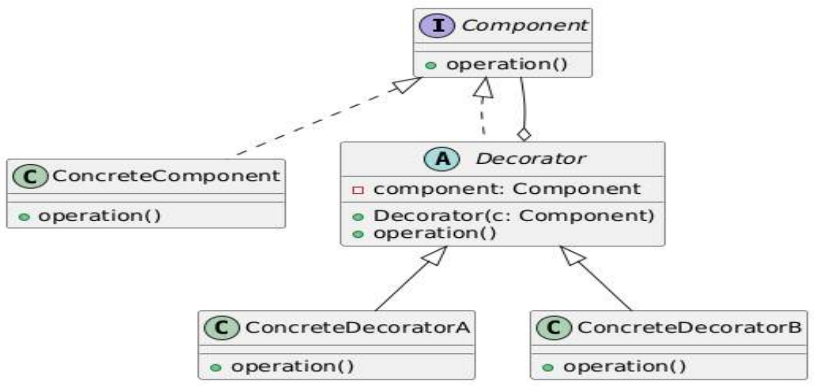
The decorator pattern is used when:

1. You need to add or remove responsibilities at runtime.
2. Subclassing would lead to an explosion of classes.
3. You want to combine multiple behaviors flexibly.

Examples:

1. Java I/O Streams (e.g., BufferedReader, InputStreamReader).
2. GUI frameworks (adding scrollbars, borders, colors dynamically).
3. Logging frameworks (adding timestamp, formatting, etc.).

UML Design:



Implementation in C++:

```
#include <iostream>
// Concrete Component
class ConcreteComponent : public Component {
public:
void operation() override {
cout << "Base operation" << endl;
}
};

// Decorator base class
class Decorator : public Component {
protected:
Component* component;
public:
Decorator(Component* c) : component(c) {}
void operation() override {
if (component)
component->operation();
}
};

// Concrete Decorator A
class ConcreteDecoratorA : public Decorator {
public:
ConcreteDecoratorA(Component* c) : Decorator(c) {}
void operation() override {
Decorator::operation();
cout << " + Added feature A" << endl;
}
};

// Concrete Decorator B
class ConcreteDecoratorB : public Decorator {
public:
ConcreteDecoratorB(Component* c) : Decorator(c) {}
void operation() override {
Decorator::operation();
cout << " + Added feature B" << endl;
}
};

// Demo
int main() {
ConcreteComponent base;
ConcreteDecoratorA decoratedA(&base);
ConcreteDecoratorB decoratedB(&decoratedA);

cout << "Final decorated operation:" << endl;
decoratedB.operation();

return 0;
}
```

Result Discussion:

Sample Output:

```
Final decorated operation:
Base operation
+ Added feature A
+ Added feature B
```

Discussion:

1. The ConcreteComponent performs the base operation.
2. ConcreteDecoratorA adds extra behavior A.
3. ConcreteDecoratorB further enhances the object with behavior B.
4. We can add responsibilities in different combinations dynamically.

Conclusion:

The Decorator Design Pattern provides a flexible alternative to subclassing for extending functionality. In this lab:

1. We designed a component and decorated it with multiple decorators.
2. We demonstrated a runtime combination of different behaviors.
3. This pattern ensures code remains extensible while adhering to the Open-Closed Principle.

Thus, the decorator pattern is essential for scalable, reusable, and maintainable object-oriented software design.

Problem No. 8
Problem Name: Implement and design Chain of Responsibility design pattern(for design use UML/user_defined class)

Objective:

The objective of this lab is to understand and implement the Chain of Responsibility (CoR) Design Pattern. The pattern is used to pass requests along a chain of handlers until one of them handles it. It promotes loose coupling between sender and receiver of a request.

Theory:

The Chain of Responsibility Pattern is a behavioral design pattern that lets you build a chain of objects to process a request. Each handler in the chain either processes the request or forwards it to the next handler.

Key Components:

1. **Handler:** Declares an interface for handling requests and optionally sets the next handler.
2. **ConcreteHandler:** Handles the request it is responsible for; otherwise forwards it to the next handler.
3. **Client:** Initiates and sends the request to the chain.

This pattern avoids coupling the sender to the receiver and allows adding/removing handlers dynamically.

Application:

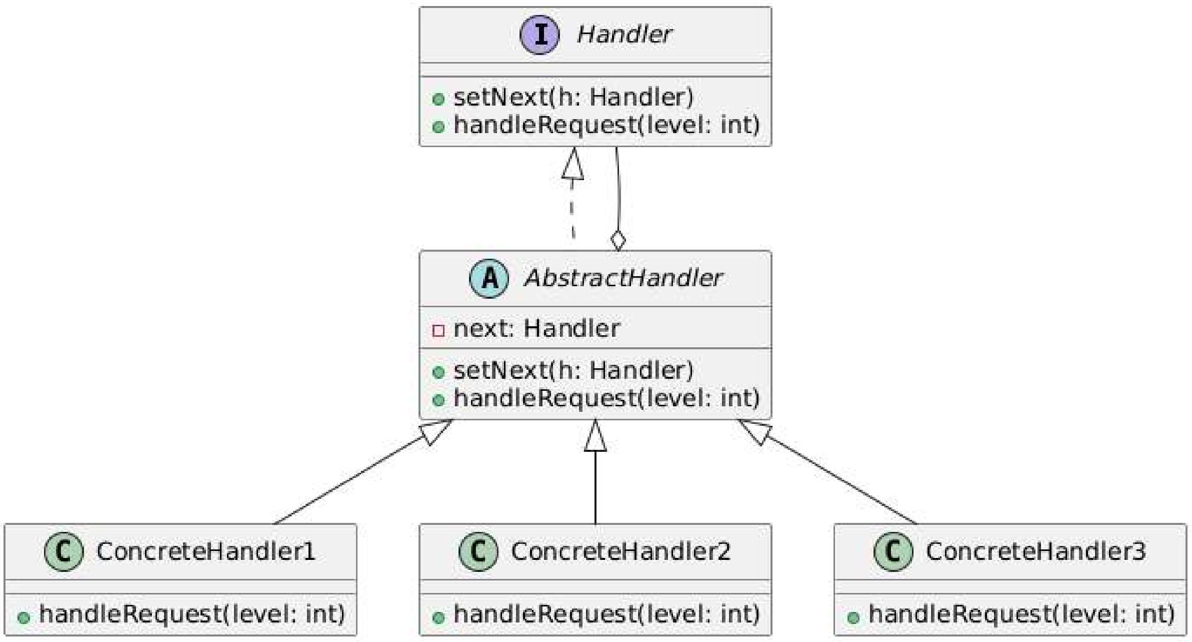
The Chain of Responsibility pattern is useful when:

1. Multiple objects can handle a request, and the handler is determined at runtime.
2. You want to decouple sender and receiver.
3. You want to specify a chain of processing objects dynamically.

Examples:

1. Exception handling systems.
2. Event handling in GUI frameworks.
3. Logging frameworks (debug → info → warning → error).
4. Customer support request escalation systems.

UML Design:



Implementation in C++:

```
#include <iostream>
using namespace std;

// Handler interface
class Handler {
public:
    virtual Handler* setNext(Handler* handler) = 0;
    virtual void handleRequest(int level) = 0;
    virtual ~Handler() = default;
};

// Abstract Handler
class AbstractHandler : public Handler {
protected:
    Handler* next = nullptr;
public:
    Handler* setNext(Handler* handler) override {
        next = handler;
        return handler;
    }
    void handleRequest(int level) override {
        if (next)
            next->handleRequest(level);
    }
};

// Concrete Handler 1
class ConcreteHandler1 : public AbstractHandler {
public:
    void handleRequest(int level) override {
        if (level == 1) {
            cout << "Handler1 processed request of level 1" << endl;
        } else if (next) {
            next->handleRequest(level);
        }
    }
};

// Concrete Handler 2
class ConcreteHandler2 : public AbstractHandler {
public:
    void handleRequest(int level) override {
        if (level == 2) {
            cout << "Handler2 processed request of level 2" << endl;
        } else if (next) {
            next->handleRequest(level);
        }
    }
};

// Concrete Handler 3
class ConcreteHandler3 : public AbstractHandler {
public:
    void handleRequest(int level) override {
        if (level == 3) {
            cout << "Handler3 processed request of level 3" << endl;
        } else if (next) {
            next->handleRequest(level);
        } else {
            cout << "No handler available for request of level " << level << endl;
        }
    }
};

// Demo
int main() {
    ConcreteHandler1 h1;
    ConcreteHandler2 h2;
    ConcreteHandler3 h3;

    // Build chain h1 -> h2 -> h3
    h1.setNext(&h2)->setNext(&h3);

    cout << "Sending request level 1:" << endl;
    h1.handleRequest(1);

    cout << "Sending request level 2:" << endl;
    h1.handleRequest(2);

    cout << "Sending request level 3:" << endl;
    h1.handleRequest(3);

    cout << "Sending request level 4:" << endl;
    h1.handleRequest(4);
    return 0;
}
```

Result Discussion

Sample Output:

```
Sending request level 1:
Handler1 processed request of level 1

Sending request level 2:
Handler2 processed request of level 2

Sending request level 3:
Handler3 processed request of level 3

Sending request level 4:
No handler available for request of level 4
```

Discussion:

1. Requests are passed along the chain until a suitable handler processes them.
2. Handler1 processed level 1, Handler2 processed level 2, Handler3 processed level 3.
3. If no handler exists for the request (level 4), a message is shown.

Conclusion:

The Chain of Responsibility Design Pattern provides a way to decouple senders and receivers of requests by allowing multiple handlers to process them dynamically. In this lab:

- We designed a handler chain with three concrete handlers.
- We demonstrated request processing at different levels.
- We showed how unhandled requests are propagated until the end of the chain.

This pattern increases flexibility and scalability by letting you add or remove handlers without changing client code.

Problem No.09

Problem Name: Implement and design Iterator design pattern(for design use UML/user_defined class)

Objective:

The objective of this lab is to understand and implement the Iterator Design Pattern. The pattern provides a way to sequentially access elements of a collection without exposing its underlying representation.

Theory:

The Iterator Pattern is a behavioral design pattern that allows clients to traverse elements in a collection (list, array, tree, etc.) without knowing how the collection is implemented. It promotes encapsulation and a standard way of iteration.

Key Components:

1. Iterator: Defines the interface for accessing elements.
2. ConcreteIterator: Implements the iterator interface.
3. Aggregate (Collection): Defines an interface for creating an iterator.
4. ConcreteAggregate: Implements the collection and returns a concrete iterator.

This pattern simplifies traversal logic and promotes reusability.

Application:

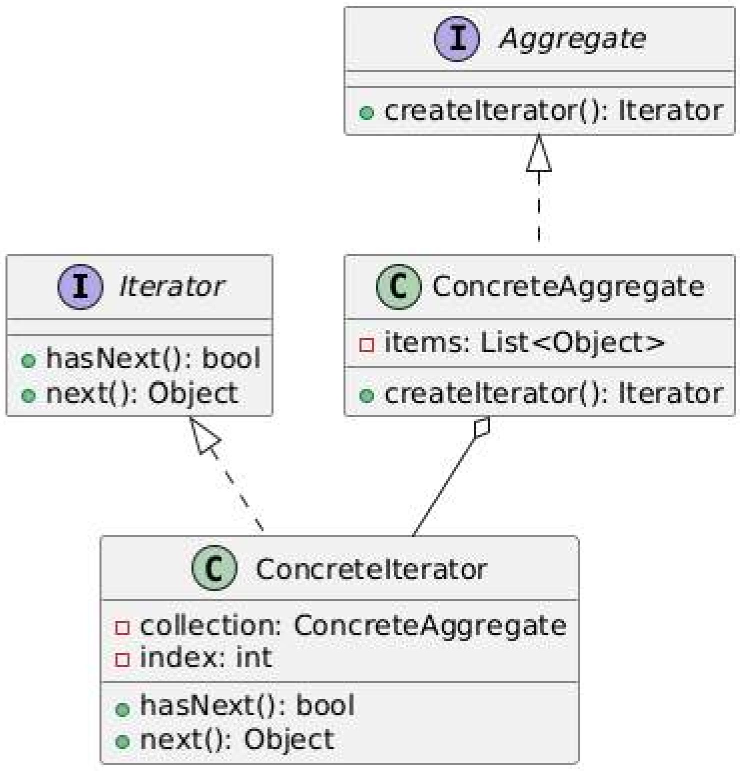
The Iterator pattern is useful when:

1. You want to access elements of a collection without exposing its structure.
2. You want multiple traversals of a collection.
3. You need a uniform way to iterate different types of collections.

Examples:

1. Standard Template Library (STL) iterators in C++.
2. Iterators in Java (Iterator, Iterable).
3. Iterators in Python (__iter__, __next__).

UML Design:



Implementation in C++:

```
#include <iostream>
#include <vector>
#include <memory>
using namespace std;

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

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

// Concrete Aggregate
class NumberCollection : public Aggregate {
private:
    vector<int> items;
public:
    void addItem(int item) { items.push_back(item); }
    vector<int>& getItems() { return items; }

    Iterator* createIterator() override;
};

// Concrete Iterator
class NumberIterator : public Iterator {
private:
    NumberCollection& collection;
    size_t index;
public:
    NumberIterator(NumberCollection& c) : collection(c), index(0) {}

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

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

// Implementation of createIterator
Iterator* NumberCollection::createIterator() {
    return new NumberIterator(*this);
}

// Demo
int main() {
    NumberCollection numbers;
    numbers.addItem(10);
    numbers.addItem(20);
    numbers.addItem(30);
    numbers.addItem(40);

    Iterator* it = numbers.createIterator();
    cout << "Iterating over collection:" << endl;
    while (it->hasNext()) {
        cout << it->next() << " ";
    }
    cout << endl;

    delete it;
    return 0;
}
```

Result Discussion

Sample Output:

```
Iterating over collection:
10 20 30 40
```

Discussion:

1. NumberCollection stores integers internally but hides its representation.
2. NumberIterator provides a sequential way to access collection elements.
3. Client code iterates through the collection without knowing it uses a vector.

Conclusion:

The **Iterator Design Pattern** provides a standard way to access elements of a collection sequentially without exposing its internal representation. In this lab:

- We designed a custom collection and iterator.
- We demonstrated sequential access to elements.
- We achieved encapsulation and flexibility in traversal.

This pattern is essential in object-oriented design for simplifying access to complex data structures.