Builder Design Pattern :

Story: Building a Smart Home Automation System

Requirements:

Imagine you're developing a smart home automation system. The system allows homeowners to control various aspects of their home, such as lighting, temperature, and security, through a mobile app. The system should have the following features:

Control multiple devices (e.g., lights, thermostats, cameras).
Set different configurations (e.g., turn on lights, set thermostat temperature, enable/disable cameras).
Easily switch between different configurations.

Naive Solution:

In the beginning, you decide to create classes for each type of device and configurations. You might create something like this in C++:

cpp
// Naive Solution

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

class Light {
public:
    void turnOn() {
        std::cout << "Light is on." << std::endl;
    }

    void turnOff() {
        std::cout << "Light is off." << std::endl;
    }
};

class Thermostat {
public:
    void setTemperature(int temperature) {
        std::cout << "Thermostat temperature set to " << temperature <<
" degrees." << std::endl;
    }
};

class Camera {
```

```
public:
    void enable() {
        std::cout << "Camera is enabled." << std::endl;
    }

    void disable() {
        std::cout << "Camera is disabled." << std::endl;
    }
};

class SmartHomeAutomation {
private:
    Light light;
    Thermostat thermostat;
    Camera camera;

public:
    void configureHome(std::string configuration) {
        if (configuration == "Home") {
            light.turnOn();
            thermostat.setTemperature(72);
            camera.enable();
        } else if (configuration == "Away") {
            light.turnOff();
            thermostat.setTemperature(60);
            camera.disable();
        }
        // Add more configurations...
    }
};
```

While the naive solution works, it has some limitations:

It becomes complex and less maintainable as you add more devices and configurations.
Users may find it cumbersome to manually configure each device and switch between configurations.
It lacks flexibility and scalability for future changes and additions.
Improving the Design:

To overcome these limitations, let's apply the Builder Design Pattern. The Builder pattern separates the construction of complex objects from their representation, allowing you to create different combinations of objects step by step.

cpp

// Improved Solution using Builder Design Pattern

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

// Device classes remain the same
class Light {
public:
    void turnOn() {
        std::cout << "Light is on." << std::endl;
    }

    void turnOff() {
        std::cout << "Light is off." << std::endl;
    }
};

class Thermostat {
public:
    void setTemperature(int temperature) {
        std::cout << "Thermostat temperature set to " << temperature <<
" degrees." << std::endl;
    }
};

class Camera {
public:
    void enable() {
        std::cout << "Camera is enabled." << std::endl;
    }

    void disable() {
        std::cout << "Camera is disabled." << std::endl;
    }
};

// Builder interface
class SmartHomeBuilder {
public:
    virtual void buildLights() = 0;
    virtual void buildThermostat() = 0;
    virtual void buildCameras() = 0;
};

// Concrete builder for 'Home' configuration
class HomeBuilder : public SmartHomeBuilder {
private:
    Light light;
```

```
    Thermostat thermostat;
    Camera camera;

public:
    void buildLights() override {
        light.turnOn();
    }

    void buildThermostat() override {
        thermostat.setTemperature(72);
    }

    void buildCameras() override {
        camera.enable();
    }
};

// Concrete builder for 'Away' configuration
class AwayBuilder : public SmartHomeBuilder {
private:
    Light light;
    Thermostat thermostat;
    Camera camera;

public:
    void buildLights() override {
        light.turnOff();
    }

    void buildThermostat() override {
        thermostat.setTemperature(60);
    }

    void buildCameras() override {
        camera.disable();
    }
};

// Director class
class SmartHomeDirector {
private:
    SmartHomeBuilder* builder;

public:
    SmartHomeDirector(SmartHomeBuilder* builder) : builder(builder) {}
```

```
    void configure() {
        builder->buildLights();
        builder->buildThermostat();
        builder->buildCameras();
    }
};

int main() {
    // Client code
    SmartHomeBuilder* homeBuilder = new HomeBuilder();
    SmartHomeBuilder* awayBuilder = new AwayBuilder();

    SmartHomeDirector homeDirector(homeBuilder);
    SmartHomeDirector awayDirector(awayBuilder);

    std::cout << "Home Configuration:" << std::endl;
    homeDirector.configure();

    std::cout << "\nAway Configuration:" << std::endl;
    awayDirector.configure();

    delete homeBuilder;
    delete awayBuilder;

    return 0;
}
```

In this improved design:

We have separated the construction of smart home configurations into builder classes (HomeBuilder and AwayBuilder), each responsible for creating a specific configuration.

The SmartHomeDirector class is responsible for directing the construction process using the appropriate builder. Users can easily switch between configurations by selecting the desired builder.

The code is more maintainable, flexible, and scalable. Adding new configurations is straightforward, and it avoids the complexity of configuring devices manually.

The Builder Design Pattern allows us to construct complex objects step by step while keeping the construction process separate from the representation, improving code organization and readability.

We can also use method chaining to make the code look more intuitive while configuring objects.

```
// Concrete builder for 'Home' configuration
class HomeBuilder : public SmartHomeBuilder {
private:
    Light light;
    Thermostat thermostat;
    Camera camera;

public:
    HomeBuilder& buildLights() {
        light.turnOn();
        return *this;
    }

    HomeBuilder& buildThermostat() {
        thermostat.setTemperature(72);
        return *this;
    }

    HomeBuilder& buildCameras() {
        camera.enable();
        return *this;
    }
};

// Concrete builder for 'Away' configuration
class AwayBuilder : public SmartHomeBuilder {
private:
    Light light;
    Thermostat thermostat;
    Camera camera;

public:
    AwayBuilder& buildLights() {
        light.turnOff();
        return *this;
    }

    AwayBuilder& buildThermostat() {
        thermostat.setTemperature(60);
        return *this;
    }

    AwayBuilder& buildCameras() {
        camera.disable();
        return *this;
    }
```

```
};
```

With these modifications, we can now use method chaining to configure different aspects of the smart home configuration more concisely:

cpp

```
int main() {
    // Client code
    SmartHomeBuilder* homeBuilder = new HomeBuilder(); //  Director class is optional
    SmartHomeBuilder* awayBuilder = new AwayBuilder();

    std::cout << "Home Configuration:" << std::endl;
    homeBuilder->buildLights().buildThermostat().buildCameras();

    std::cout << "\nAway Configuration:" << std::endl;
    awayBuilder->buildLights().buildThermostat().buildCameras();

    delete homeBuilder;
    delete awayBuilder;

    return 0;
}
```

Or we can also do the same thing in Director class

```
// Director class
class SmartHomeDirector {
private:
    SmartHomeBuilder* builder;

public:
    SmartHomeDirector(SmartHomeBuilder* builder) : builder(builder) {}

    SmartHomeDirector& configure() {
        builder->buildLights().buildThermostat().buildCameras();
        return *this;
    }
};
```

Another Example of Builder Pattern :

Requirements:

Imagine you're building a Meal Builder for a fast-food restaurant. A meal can consist of a burger, a drink, and optional sides. Customers should be able to build their meals with different components.

Using Method Chaining in Director:

In this example, we'll modify the Director class to support method chaining when configuring meals:

cpp

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

// Product: Burger
class Burger {
public:
    void addIngredient(const std::string& ingredient) {
        ingredients.push_back(ingredient);
    }

    void show() {
        std::cout << "Burger Ingredients: ";
        for (const auto& ingredient : ingredients) {
            std::cout << ingredient << ", ";
        }
        std::cout << std::endl;
    }

private:
    std::vector<std::string> ingredients;
};

// Product: Drink
class Drink {
public:
    void setType(const std::string& type) {
        this->type = type;
    }

    void show() {
        std::cout << "Drink Type: " << type << std::endl;
    }

private:
```

```
    std::string type;
};

// Product: Side
class Side {
public:
    void setName(const std::string& name) {
        this->name = name;
    }

    void show() {
        std::cout << "Side Name: " << name << std::endl;
    }

private:
    std::string name;
};

// Builder interface
class MealBuilder {
public:
    virtual MealBuilder& buildBurger() = 0;
    virtual MealBuilder& buildDrink() = 0;
    virtual MealBuilder& buildSide() = 0;
};

// Concrete builder for a specific meal
class VeggieMealBuilder : public MealBuilder {
public:
    VeggieMealBuilder() : meal(new Meal) {}

    ~VeggieMealBuilder() {
        delete meal;
    }

    MealBuilder& buildBurger() override {
        meal->burger.addIngredient("Veggie Patty");
        return *this;
    }

    MealBuilder& buildDrink() override {
        meal->drink.setType("Water");
        return *this;
    }

    MealBuilder& buildSide() override {
```

```
        meal->side.setName("Salad");
        return *this;
    }

    Meal* getMeal() {
        return meal;
    }

private:
    Meal* meal;
};

// Product: Meal
class Meal {
public:
    void show() {
        burger.show();
        drink.show();
        side.show();
    }

private:
    Burger burger;
    Drink drink;
    Side side;
};

int main() {
    VeggieMealBuilder veggieMealBuilder;
    MealBuilder& builder = veggieMealBuilder;

    Meal* meal =
builder.buildBurger().buildDrink().buildSide().getMeal();

    std::cout << "Veggie Meal Configuration:" << std::endl;
    meal->show();

    delete meal;

    return 0;
}
```

In this example:

The VeggieMealBuilder class represents a concrete builder for a veggie meal. It allows chaining the methods buildBurger(), buildDrink(), and buildSide() to configure the meal components.

The Meal class represents the final product, which includes a burger, a drink, and a side.

The main function demonstrates how to create a veggie meal using method chaining with the builder.

PROTOTYPE DESIGN PATTERN :
Story: Cloning Network Devices

Requirements:

Imagine you are an engineer working at a network equipment company. Your company manufactures various types of network devices, such as routers, switches, and firewalls. You want to create a system that allows for the efficient cloning of network devices. The requirements are as follows:

Create and configure network devices with different settings, including IP addresses, protocols, and security policies.
Enable network administrators to quickly replicate existing device configurations for scalability.
Ensure that the original device configurations remain unchanged for reference.
Intent of Prototype Design Pattern:

The Prototype Design Pattern allows you to create new objects by copying an existing object, known as the prototype. It's useful when creating objects directly is more complex or resource-intensive than copying them. In our story, we want to create replicas of network devices without modifying the original device configurations.

Naive Solution:

In your initial attempt to meet the requirements, you decide to create a class called NetworkDevice to represent these network devices. You implement a method to clone network devices. Here's a simplified version of what the code might look like in C++:

Naive Solution :

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

class NetworkDevice {
public:
    NetworkDevice(std::string name, std::string ip, std::string
protocol, std::string securityPolicy)
        : name(name), ip(ip), protocol(protocol),
securityPolicy(securityPolicy) {}

    // Copy constructor to clone a network device
    NetworkDevice(const NetworkDevice& other) {
        name = other.name;
        ip = other.ip;
        protocol = other.protocol;
        securityPolicy = other.securityPolicy;
    }

    // Method to clone a network device
    NetworkDevice clone() {
        return NetworkDevice(name, ip, protocol, securityPolicy);
    }

    void display() {
        std::cout << "Name: " << name << ", IP: " << ip << ", Protocol:
" << protocol
                  << ", Security Policy: " << securityPolicy <<
std::endl;
    }

private:
    std::string name;
    std::string ip;
    std::string protocol;
    std::string securityPolicy;
};

int main() {
```

```
    // Create an original network device
    NetworkDevice original("Router A", "192.168.1.1", "TCP/IP",
"Firewall Enabled");

    // Clone the network device
    NetworkDevice replica = original.clone();

    // Display the original and replica
    std::cout << "Original Network Device:" << std::endl;
    original.display();

    std::cout << "\nReplica Network Device:" << std::endl;
    replica.display();

    return 0;
}
```

However, this approach has limitations:

It works well for simple objects like NetworkDevice, but it becomes cumbersome when dealing with more complex network devices with deep configurations and dependencies.
If you need to create network devices with custom settings, the constructor and cloning methods become increasingly complex and error-prone.
If you have multiple types of network devices with varying configurations and behaviors, maintaining a single class becomes challenging.
Improving the Design with Prototype Pattern:

To address these limitations, you can apply the Prototype Design Pattern. The Prototype pattern separates the construction of complex objects from their representation by allowing objects to be cloned. Here's how you can refactor the code to use the Prototype pattern.

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

class NetworkDevice {
public:
    virtual NetworkDevice* clone() const = 0;
    virtual void display() const = 0;
    virtual void update(std::string newName) = 0; // Update the device's
name
};

class Router : public NetworkDevice {
public:
```

```
    Router(std::string name, std::string ip, std::string securityPolicy)
        : name(std::move(name)), ip(std::move(ip)),
securityPolicy(std::move(securityPolicy)) {}

    NetworkDevice* clone() const override {
        return new Router(*this);
    }

    void display() const override {
        std::cout << "Router - Name: " << name << ", IP: " << ip << ",
Security Policy: " << securityPolicy << std::endl;
    }

    void update(std::string newName) override {
        name = std::move(newName);
    }

private:
    std::string name;
    std::string ip;
    std::string securityPolicy;
};

class Switch : public NetworkDevice {
public:
    Switch(std::string name, std::string protocol)
        : name(std::move(name)), protocol(std::move(protocol)) {}

    NetworkDevice* clone() const override {
        return new Switch(*this);
    }

    void display() const override {
        std::cout << "Switch - Name: " << name << ", Protocol: " <<
protocol << std::endl;
    }

    void update(std::string newName) override {
        name = std::move(newName);
    }

private:
    std::string name;
    std::string protocol;
};
```

```
int main() {
    // Create prototype instances of a router and a switch
    NetworkDevice* routerPrototype = new Router("Router A",
"192.168.1.1", "Firewall Enabled");
    NetworkDevice* switchPrototype = new Switch("Switch X", "Ethernet");

    // Clone and display router and switch devices
    NetworkDevice* routerClone = routerPrototype->clone();
    NetworkDevice* switchClone = switchPrototype->clone();

    std::cout << "Router Clone:" << std::endl;
    routerClone->display();

    std::cout << "\nSwitch Clone:" << std::endl;
    switchClone->display();

    // Update the names of the clones
    routerClone->update("Router B");
    switchClone->update("Switch Y");

    std::cout << "\nUpdated Router Clone:" << std::endl;
    routerClone->display();

    std::cout << "\nUpdated Switch Clone:" << std::endl;
    switchClone->display();

    // Clean up
    delete routerPrototype;
    delete switchPrototype;
    delete routerClone;
    delete switchClone;

    return 0;
}
```

In this improved design:

We have separate classes for different types of network devices (Router and Switch) that inherit from the common NetworkDevice interface.

Each concrete network device class implements the clone method to create a deep copy of itself.

We create prototype instances of routers and switches, and we can easily clone these prototypes to create new network devices with the same configurations.

This design is more extensible and allows for custom settings, different types of network devices, and easier maintenance. It follows the Prototype Design Pattern, making it easier to manage complex object creation.

Summary of Prototype Design Pattern:

The Prototype Design Pattern allows you to create new objects by copying existing objects (prototypes). It's especially useful when:

Creating objects directly is more complex or resource-intensive than copying them.
You want to isolate the construction of complex objects from their representation.
You need to create objects with varying configurations or customizations based on existing prototypes.
In the context of networking or other domains, the Prototype pattern helps efficiently create and customize objects while maintaining the integrity of the original prototypes.

Comparison :

Comparing the Builder, Prototype, Abstract Factory, and Factory Design Patterns with examples for each:

Builder Design Pattern:
Intent:
The Builder Design Pattern separates the construction of a complex object from its representation. It allows the construction process to create different representations of an object.

Example:
Consider building a complex meal in a fast-food restaurant. The Builder pattern can be used to create a MealBuilder that constructs meals with various combinations of items (e.g., burger, fries, drink) based on customer preferences.

Prototype Design Pattern:
Intent:
The Prototype Design Pattern is used to create new objects by copying an existing object, known as the prototype. It's useful when creating objects directly is more complex or resource-intensive than copying them.

Example:

In a graphics editing software, you might have a shape library with predefined shapes (e.g., rectangles, circles). The Prototype pattern allows you to clone and customize these shapes to create new ones with different sizes and colors.

Abstract Factory Design Pattern:
Intent:
The Abstract Factory Design Pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes. It's used when a system should be independent of how its objects are created, composed, and represented.

Example:
Consider a user interface library. You might have different UI components like buttons and text fields. An abstract factory can create different UI component families (e.g., Windows or macOS) that consist of related objects, such as buttons and text fields, ensuring that these components work cohesively within a specific environment.

Factory Design Pattern:
Intent:
The Factory Design Pattern defines an interface for creating an object but allows subclasses to alter the type of objects that will be created. It's used when a class cannot anticipate the class of objects it must create.

Example:
In a document processing application, you might have a factory that creates different types of documents (e.g., PDF, Word, Excel). The Factory method allows you to create documents without specifying their exact class, making it easy to add new document types in the future.

Comparison:

Intent:
Builder: Separates the construction of complex objects from their representation.
Prototype: Creates new objects by copying existing prototypes.
Abstract Factory: Creates families of related objects without specifying their concrete classes.
Factory: Defines an interface for creating objects without specifying their exact class.

Object Creation:
Builder: Focuses on constructing complex objects with step-by-step processes.
Prototype: Creates objects by cloning existing instances.
Abstract Factory: Creates families of related objects.
Factory: Creates objects based on a factory method.

Complexity:
Builder: Suitable for complex object construction with multiple configuration options.

Prototype: Suitable for creating objects with complex configurations based on existing instances.
Abstract Factory: Suitable for environments where different families of objects need to be created.
Factory: Suitable for simpler object creation scenarios.

Usage:
Builder: Used when an object has a complex construction process with multiple steps.
Prototype: Used when creating objects by copying existing ones is efficient and practical.
Abstract Factory: Used when creating families of related objects with consistent interfaces.
Factory: Used when an object needs to be created without exposing the instantiation logic.

Examples:
Builder: Building complex meals in a fast-food restaurant.
Prototype: Cloning and customizing shapes in a graphics editing software.
Abstract Factory: Creating game elements in a video game.
Factory: Creating documents in a document processing application.
In summary, each design pattern addresses different concerns and use cases. The choice of pattern depends on the specific requirements and complexities of your application's object creation and construction processes.

Chain of Responsibility Design Pattern

Intent: The Chain of Responsibility design pattern is a behavioral design pattern that allows you to pass requests along a chain of handlers. Each handler decides whether to process the request or pass it to the next handler in the chain. This pattern promotes loose coupling and allows you to add, modify, or remove handlers without affecting the client code.

Story: Building an Expense Approval System

Let's illustrate the Chain of Responsibility design pattern with a story of building an Expense Approval System for a company. The system needs to handle expense approval requests where expenses pass through various levels of approval, including manager, department head, and CEO.

Requirements:

An expense approval request includes the expense amount and information about the requester.
There are three levels of approval: Manager, Department Head, and CEO.
The approval hierarchy is as follows: Manager can approve expenses up to $1000, Department Head can approve up to $5000, and CEO can approve any amount.
Each approver can either approve or reject the expense request. If they cannot approve, they pass it to the next level.

Naive Solution:

We could start with a naive solution that involves a single function for each level of approval.

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

class Expense {
public:
    std::string requester;
    double amount;

    Expense(const std::string& req, double amt) : requester(req), amount(amt) {}
};

class ExpenseApprovalSystem {
public:
    void approveExpenseByManager(const Expense& expense) {
        if (expense.amount <= 1000) {
            std::cout << "Manager approved the expense for " << expense.requester <<
"." << std::endl;
        } else {
            std::cout << "Manager cannot approve. Escalating to Department Head." <<
std::endl;
            approveExpenseByDepartmentHead(expense);
        }
    }

    void approveExpenseByDepartmentHead(const Expense& expense) {
        if (expense.amount <= 5000) {
            std::cout << "Department Head approved the expense for " <<
expense.requester << "." << std::endl;
        } else {
            std::cout << "Department Head cannot approve. Escalating to CEO." <<
std::endl;
            approveExpenseByCEO(expense);
        }
    }

    void approveExpenseByCEO(const Expense& expense) {
        std::cout << "CEO approved the expense for " << expense.requester << "." <<
std::endl;
```

```
    }
};

int main() {
    Expense expense1("Alice", 800);
    Expense expense2("Bob", 4500);
    Expense expense3("Charlie", 12000);

    ExpenseApprovalSystem approvalSystem;

    approvalSystem.approveExpenseByManager(expense1);
    approvalSystem.approveExpenseByManager(expense2);
    approvalSystem.approveExpenseByManager(expense3);

    return 0;
```

Issues with the Naive Solution:

> The code is tightly coupled with the approval logic. If we need to add or modify approval levels, it would require changing the `ExpenseApprovalSystem` class. The code doesn't follow the Single Responsibility Principle as it combines both expense approval and escalation logic.

Using the Chain of Responsibility Pattern:

Let's refactor the code to use the Chain of Responsibility design pattern to address these issues.

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

class Expense {
public:
    std::string requester;
    double amount;

    Expense(const std::string& req, double amt) : requester(req), amount(amt) {}
};

// Handler interface
class ExpenseHandler {
protected:
    ExpenseHandler* successor;
```

```
public:
    ExpenseHandler() : successor(nullptr) {}

    void setSuccessor(ExpenseHandler* next) {
        successor = next;
    }

    virtual void approveExpense(const Expense& expense) = 0;
};

// Concrete Handler: Manager
class Manager : public ExpenseHandler {
public:
    void approveExpense(const Expense& expense) override {
        if (expense.amount <= 1000) {
            std::cout << "Manager approved the expense for " << expense.requester <<
"." << std::endl;
        } else if (successor != nullptr) {
            std::cout << "Manager cannot approve. Escalating to " <<
typeid(*successor).name() << "." << std::endl;
            successor->approveExpense(expense);
        } else {
            std::cout << "Reached the end of the chain. Approval denied." <<
std::endl;
        }
    }
};

// Concrete Handler: Department Head
class DepartmentHead : public ExpenseHandler {
public:
    void approveExpense(const Expense& expense) override {
        if (expense.amount <= 5000) {
            std::cout << "Department Head approved the expense for " <<
expense.requester << "." << std::endl;
        } else if (successor != nullptr) {
            std::cout << "Department Head cannot approve. Escalating to " <<
typeid(*successor).name() << "." << std::endl;
            successor->approveExpense(expense);
        } else {
            std::cout << "Reached the end of the chain. Approval denied." <<
std::endl;
        }
    }
};

// Concrete Handler: CEO
class CEO : public ExpenseHandler {
public:
    void approveExpense(const Expense& expense) override {
        std::cout << "CEO approved the expense for " << expense.requester << "." <<
std::endl;
    }
```

```
};

int main() {
    Expense expense1("Alice", 800);
    Expense expense2("Bob", 4500);
    Expense expense3("Charlie", 12000);

    Manager manager;
    DepartmentHead departmentHead;
    CEO ceo;

    manager.setSuccessor(&departmentHead);
    departmentHead.setSuccessor(&ceo);

    manager.approveExpense(expense1);
    manager.approveExpense(expense2);
    manager.approveExpense(expense3);

    return 0;
```

Benefits of Using the Chain of Responsibility Pattern:

Each approver (Manager, Department Head, CEO) is responsible for its specific level of approval, making the code more modular and adhering to the Single Responsibility Principle.
The Chain of Responsibility pattern allows us to easily add, remove, or modify the chain of handlers without affecting the client code.
It promotes loose coupling between the client and the handlers, making the code more maintainable and flexible.

Summary:

The Chain of Responsibility design pattern allows you to pass requests through a chain of handlers, with each handler deciding whether to process the request or pass it to the next handler. This pattern promotes flexibility, modularity, and separation of concerns in your code, making it easier to maintain and extend. In the example, we illustrated how it can be used to implement an Expense Approval System with different approval levels.

Another Example :

Story: Building an E-commerce Order Processing System

Let's illustrate the Chain of Responsibility design pattern with a story of building an E-commerce Order Processing System. The system needs to handle various stages of order processing, including order validation, payment processing, and shipping.

Requirements:

1. An order consists of products and customer information.

2. Order processing consists of three stages: validation, payment processing, and shipping.

3. Each stage has specific responsibilities:

- Validation: Ensures the order is valid, contains valid products, and has customer details.

- Payment Processing: Handles payment authorization and processing.

- Shipping: Manages the shipment of the order.

4. If one stage fails, the subsequent stages should not execute.

Naive Solution:

We could start with a naive solution that involves a single function for each stage of order processing.

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

class Product {
public:
    std::string name;
    double price;

    Product(const std::string& n, double p) : name(n), price(p) {}
};
```

```
class Customer {
public:
    std::string name;
    std::string address;

    Customer(const std::string& n, const std::string& addr) : name(n),
address(addr) {}
};

class Order {
public:
    std::vector<Product> products;
    Customer customer;

    Order(const std::vector<Product>& prods, const Customer& cust) :
products(prods), customer(cust) {}
};

class OrderProcessor {
public:
    void processOrder(const Order& order) {
        if (validateOrder(order)) {
            if (processPayment(order)) {
                shipOrder(order);
            } else {
                std::cout << "Payment processing failed." << std::endl;
            }
        } else {
            std::cout << "Order validation failed." << std::endl;
        }
    }

    bool validateOrder(const Order& order) {
        // Perform order validation logic
        return true; // Validation always succeeds in the naive solution
    }

    bool processPayment(const Order& order) {
        // Perform payment processing logic
        return true; // Payment always succeeds in the naive solution
    }

    void shipOrder(const Order& order) {
        // Perform shipping logic
        std::cout << "Order shipped to " << order.customer.name << " at
```

```
" << order.customer.address << "." << std::endl;
    }
};

int main() {
    Product product1("Product A", 100.0);
    Product product2("Product B", 50.0);

    Customer customer("Alice", "123 Main St");

    std::vector<Product> products = {product1, product2};
    Order order(products, customer);

    OrderProcessor orderProcessor;
    orderProcessor.processOrder(order);

    return 0;
}
```

Issues with the Naive Solution:

1. The code is tightly coupled with the order processing logic. If we need to add or modify processing stages, it would require changing the `OrderProcessor` class.

2. The code doesn't follow the Single Responsibility Principle as it combines all order processing stages into one class.

Using the Chain of Responsibility Pattern:

Let's refactor the code to use the Chain of Responsibility design pattern to address these issues.

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

class Product {
public:
    std::string name;
    double price;

    Product(const std::string& n, double p) : name(n), price(p) {}
```

```
};

class Customer {
public:
    std::string name;
    std::string address;

    Customer(const std::string& n, const std::string& addr) : name(n),
address(addr) {}
};

class Order {
public:
    std::vector<Product> products;
    Customer customer;

    Order(const std::vector<Product>& prods, const Customer& cust) :
products(prods), customer(cust) {}
};

// Handler interface for order processing stages
class OrderProcessingStage {
protected:
    OrderProcessingStage* successor;

public:
    OrderProcessingStage() : successor(nullptr) {}

    void setSuccessor(OrderProcessingStage* next) {
        successor = next;
    }

    virtual bool process(const Order& order) = 0;
};

// Concrete Handler: Order Validation
class OrderValidation : public OrderProcessingStage {
public:
    bool process(const Order& order) override {
        // Perform order validation logic
        return true; // Validation always succeeds in this example
    }
};

// Concrete Handler: Payment Processing
class PaymentProcessing : public OrderProcessingStage {
```

```
public:
    bool process(const Order& order) override {
        // Perform payment processing logic
        return true; // Payment always succeeds in this example
    }
};

// Concrete Handler: Shipping
class Shipping : public OrderProcessingStage {
public:
    bool process(const Order& order) override {
        // Perform shipping logic
        std::cout << "Order shipped to " << order.customer.name << " at
" << order.customer.address << "." << std::endl;
        return true; // Shipping always succeeds in this example
    }
};

int main() {
    Product product1("Product A", 100.0);
    Product product2("Product B", 50.0);

    Customer customer("Alice", "123 Main St");

    std::vector<Product> products = {product1, product2};
    Order order(products, customer);

    OrderValidation orderValidation;
    PaymentProcessing paymentProcessing;
    Shipping shipping;

    orderValidation.setSuccessor(&paymentProcessing);
    paymentProcessing.setSuccessor(&shipping);

    if (orderValidation.process(order)) {
        std::cout << "Order processing completed successfully." <<
std::endl;
    } else {
        std::cout << "Order processing failed." << std::endl;
    }

    return 0;
}
```

Benefits of Using the Chain of Responsibility Pattern:

1. Each order processing stage (Order Validation, Payment Processing, Shipping) is responsible for its specific task, making the code more modular and adhering to the Single Responsibility Principle.

2. The Chain of Responsibility pattern allows us to easily add, remove, or modify the chain of handlers without affecting the client code.

3. It promotes loose coupling between the client and the handlers, making the code more maintainable and flexible.

Summary:

The Chain of Responsibility design pattern allows you to pass requests through a chain of handlers, with each handler deciding whether

---

Iterator Design Pattern : -

Intent:

The Iterator design pattern is a behavioral design pattern that provides a way to access the elements of an aggregate object (such as a collection) sequentially without exposing its underlying representation. It separates the traversal of a container from its contents, making it easier to add new traversal methods.

Story: Managing a Library Catalog

Let's illustrate the Iterator design pattern with a story of managing a Library Catalog. The library has a collection of books, and we want to provide different ways to browse through the catalog.

Requirements:

1. The library catalog contains a collection of books.

2. Users should be able to browse the catalog in different ways: by title, by author, and by genre.

3. The catalog should support adding, removing, and listing books.

Naive Solution:

We could start with a naive solution that involves a single class for managing the library catalog and providing methods for browsing by title, author, and genre.

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

class Book {
public:
    std::string title;
    std::string author;
    std::string genre;

    Book(const std::string& t, const std::string& a, const std::string&
g)
        : title(t), author(a), genre(g) {}
};

class LibraryCatalog {
private:
    std::vector<Book> books;

public:
```

```
    void addBook(const Book& book) {
        books.push_back(book);
    }

    void removeBook(const Book& book) {
        for (auto it = books.begin(); it != books.end(); ++it) {
            if (it->title == book.title && it->author == book.author) {
                books.erase(it);
                break;
            }
        }
    }

    void browseByTitle() {
        std::cout << "Browsing by Title:" << std::endl;
        for (const Book& book : books) {
            std::cout << "Title: " << book.title << ", Author: " <<
book.author << ", Genre: " << book.genre << std::endl;
        }
    }

    void browseByAuthor() {
        std::cout << "Browsing by Author:" << std::endl;
        for (const Book& book : books) {
            std::cout << "Author: " << book.author << ", Title: " <<
book.title << ", Genre: " << book.genre << std::endl;
        }
    }

    void browseByGenre() {
        std::cout << "Browsing by Genre:" << std::endl;
        for (const Book& book : books) {
            std::cout << "Genre: " << book.genre << ", Title: " <<
book.title << ", Author: " << book.author << std::endl;
        }
    }
};

int main() {
    LibraryCatalog catalog;

    catalog.addBook(Book("Book A", "Author X", "Fantasy"));
    catalog.addBook(Book("Book B", "Author Y", "Mystery"));
    catalog.addBook(Book("Book C", "Author X", "Science Fiction"));

    catalog.browseByTitle();
```

```
    catalog.browseByAuthor();
    catalog.browseByGenre();

    return 0;
}
```

Issues with the Naive Solution:

1. The code is tightly coupled with the specific traversal methods (browsing by title, author, and genre). Adding new traversal methods would require modifying the `LibraryCatalog` class.

2. The code doesn't follow the Single Responsibility Principle as it combines catalog management and traversal logic.

Using the Iterator Design Pattern:

Let's refactor the code to use the Iterator design pattern to address these issues.

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

class Book {
public:
    std::string title;
    std::string author;
    std::string genre;

    Book(const std::string& t, const std::string& a, const std::string&
g)
        : title(t), author(a), genre(g) {}
};

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

// Concrete Iterator: TitleIterator
```

```
class TitleIterator : public Iterator {
private:
    std::vector<Book> books;
    size_t position;

public:
    TitleIterator(const std::vector<Book>& bookList) : books(bookList),
position(0) {}

    bool hasNext() override {
        return position < books.size();
    }

    Book next() override {
        if (hasNext()) {
            return books[position++];
        }
        throw std::out_of_range("No more elements");
    }
};

// Concrete Iterator: AuthorIterator
class AuthorIterator : public Iterator {
private:
    std::vector<Book> books;
    size_t position;

public:
    AuthorIterator(const std::vector<Book>& bookList) : books(bookList),
position(0) {}

    bool hasNext() override {
        return position < books.size();
    }

    Book next() override {
        if (hasNext()) {
            return books[position++];
        }
        throw std::out_of_range("No more elements");
    }
};

// Concrete Iterator: GenreIterator
class GenreIterator : public Iterator {
private:
```

```
    std::vector<Book> books;
    size_t position;

public:
    GenreIterator(const std::vector<Book>& bookList) : books(bookList),
position(0) {}

    bool hasNext() override {
        return position < books.size();
    }

    Book next() override {
        if (hasNext()) {
            return books[position++];
        }
        throw std::out_of_range("No more elements");
    }
};

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

// Concrete Aggregate: LibraryCatalog
class LibraryCatalog : public Catalog {
private:
    std::vector<Book> books;

public:
    void addBook(const Book& book) {
        books.push_back(book);
    }

    void removeBook(const Book& book) {
        for (auto it = books.begin(); it != books.end(); ++it) {
            if (it->title == book.title && it->author == book.author) {
                books.erase(it);
                break;
            }
        }
    }

    Iterator* createIterator() override {
        return new TitleIterator(books);
```

```
    }

    Iterator* createAuthorIterator() {
        return new AuthorIterator(books);
    }

    Iterator* createGenreIterator() {
        return new GenreIterator(books);
    }
};

void printCatalog(Iterator* iterator, const std::string& label) {
    std::cout << "Browsing by " << label << ":" << std::endl;
    while (

iterator->hasNext()) {
        Book book = iterator->next();
        std::cout << label << ": " << book.title << ", Author: " <<
book.author << ", Genre: " << book.genre << std::endl;
    }
    delete iterator;
}

int main() {
    LibraryCatalog catalog;

    catalog.addBook(Book("Book A", "Author X", "Fantasy"));
    catalog.addBook(Book("Book B", "Author Y", "Mystery"));
    catalog.addBook(Book("Book C", "Author X", "Science Fiction"));

    Iterator* titleIterator = catalog.createIterator();
    Iterator* authorIterator = catalog.createAuthorIterator();
    Iterator* genreIterator = catalog.createGenreIterator();

    printCatalog(titleIterator, "Title");
    printCatalog(authorIterator, "Author");
    printCatalog(genreIterator, "Genre");

    return 0;
}
```

Benefits of Using the Iterator Design Pattern:

1. The Iterator design pattern separates the traversal logic from the catalog management, making the code more modular and adhering to the Single Responsibility Principle.

2. It allows us to add new traversal methods (iterators) without modifying the `LibraryCatalog` class.

3. The code follows the open-closed principle, as new iterators can be added without modifying existing code.

Summary:

The Iterator design pattern decouples the traversal of a container (library catalog) from its contents (books). It provides a way to access elements sequentially without exposing the underlying representation. This pattern promotes code reusability, modularity, and separation of concerns.

STRATEGY DESIGN PATTERN : (Behavioral)

Requirements:

Let's consider a scenario in which we are building a payment processing system for an online store. The system needs to support multiple payment methods such as credit card, PayPal, and cryptocurrency. Each payment method has its own unique processing logic. The system should allow users to select their preferred payment method at checkout and process payments accordingly.

Naive Solution:

A naive approach to solving this problem would be to create a monolithic payment processing class that handles all payment methods within a single class. Here's some pseudocode to illustrate the naive solution:

```
class PaymentProcessor {
public:
    void processPayment(string paymentMethod, double amount) {
        if (paymentMethod == "CreditCard") {
            // Process credit card payment logic
            // ...
        } else if (paymentMethod == "PayPal") {
            // Process PayPal payment logic
            // ...
        } else if (paymentMethod == "Cryptocurrency") {
            // Process cryptocurrency payment logic
            // ...
        }
        // Handle other payment methods...
    }
};
```

Issues with the Naive Solution:

1. The `PaymentProcessor` class becomes large and complex as we add more payment methods, violating the Single Responsibility Principle.

2. Adding new payment methods requires modifying the `PaymentProcessor` class, which can introduce bugs and make maintenance difficult.

3. It lacks flexibility, as we cannot easily add or remove payment methods at runtime.

Applying the Strategy Design Pattern:

The Strategy Design Pattern allows us to define a family of algorithms, encapsulate each one, and make them interchangeable. Let's refactor the payment processing system using this pattern.

Step 1: Define the Strategy Interface

Create an interface `PaymentStrategy` that declares a method `processPayment`.

```
class PaymentStrategy {
public:
    virtual void processPayment(double amount) = 0;
};
```

Step 2: Create Concrete Strategy Classes

Implement concrete strategy classes for each payment method, each of which implements the `processPayment` method.

```
class CreditCardPayment : public PaymentStrategy {
public:
    void processPayment(double amount) override {
        // Process credit card payment logic
        // ...
    }
};
```

```
class PayPalPayment : public PaymentStrategy {
public:
    void processPayment(double amount) override {
        // Process PayPal payment logic
        // ...
    }
```

```
};
```

```
class CryptocurrencyPayment : public PaymentStrategy {
public:
    void processPayment(double amount) override {
        // Process cryptocurrency payment logic
        // ...
    }
};
```

Step 3: Refactor the Context Class

The context class, in this case, is the `PaymentProcessor`. It should have a member variable to store the current payment strategy and a method to set the payment strategy.

```
class PaymentProcessor {
private:
    PaymentStrategy* paymentStrategy;

public:
    void setPaymentStrategy(PaymentStrategy* strategy) {
        paymentStrategy = strategy;
    }

    void processPayment(double amount) {
        paymentStrategy->processPayment(amount);
    }
};
```

Using the Strategy Pattern:

Now, we can use the strategy pattern to process payments in a flexible and maintainable way:

```
int main() {
    PaymentProcessor processor;
```

```
    // Select and set the payment strategy at runtime
    PaymentStrategy* strategy = new CreditCardPayment();
    processor.setPaymentStrategy(strategy);

    // Process the payment
    processor.processPayment(100.0);

    // Change the payment strategy
    strategy = new PayPalPayment();
    processor.setPaymentStrategy(strategy);

    // Process another payment using the new strategy
    processor.processPayment(50.0);

    // Clean up
    delete strategy;

    return 0;
}
```

Summary of the Strategy Design Pattern:

- Intent: The Strategy Design Pattern defines a family of algorithms, encapsulates each one as a separate class, and makes them interchangeable. It allows clients to choose the appropriate algorithm at runtime without altering the client code.

- Advantages:

- Promotes the Single Responsibility Principle.

- Encourages code reusability and maintainability.

- Allows for easy addition of new strategies.

- Provides flexibility to change algorithms dynamically.

In our example, we encapsulated the payment processing algorithms into separate strategy classes and allowed the client (PaymentProcessor) to select and use the appropriate strategy at runtime. This adheres to the principles of the Strategy Design Pattern.

Combining Strategy Pattern with Factory Pattern :

You can combine the Strategy Design Pattern with the Factory Design Pattern to make the code more flexible and extensible. The Factory Pattern can be used to create instances of concrete strategy classes, allowing you to decouple the creation of strategies from their usage. Here's how you can modify the code:

Step 1: Create a PaymentStrategyFactory

Create a factory class, `PaymentStrategyFactory`, responsible for creating instances of concrete payment strategies.

```
class PaymentStrategyFactory {
public:
    static PaymentStrategy* createPaymentStrategy(const std::string&
paymentMethod) {
        if (paymentMethod == "CreditCard") {
            return new CreditCardPayment();
        } else if (paymentMethod == "PayPal") {
            return new PayPalPayment();
        } else if (paymentMethod == "Cryptocurrency") {
            return new CryptocurrencyPayment();
        } else {
            throw std::invalid_argument("Invalid payment method");
        }
    }
};
```

Step 2: Modify the PaymentProcessor to Use the Factory

Modify the `PaymentProcessor` class to use the factory to create payment strategy instances.

```
class PaymentProcessor {
private:
    PaymentStrategy* paymentStrategy;

public:
    void setPaymentStrategy(const std::string& paymentMethod) {
        paymentStrategy =
PaymentStrategyFactory::createPaymentStrategy(paymentMethod);
    }

    void processPayment(double amount) {
        if (paymentStrategy) {
            paymentStrategy->processPayment(amount);
        } else {
            std::cerr << "Payment strategy not set." << std::endl;
        }
    }
};
```

Using the Combined Factory and Strategy Pattern:

Now, you can use both the Factory and Strategy patterns together:

```
int main() {
    PaymentProcessor processor;

    // Use the factory to create payment strategies
    processor.setPaymentStrategy("CreditCard");
    processor.processPayment(100.0);

    processor.setPaymentStrategy("PayPal");
    processor.processPayment(50.0);
```

```
    // Clean up (Note: You may need a destructor to release strategy
instances)
    delete processor.getPaymentStrategy();

    return 0;
}
```

Summary:

By combining the Factory Design Pattern with the Strategy Design Pattern, you achieve a more flexible and modular design. The Factory Pattern allows you to create strategy instances dynamically, making it easier to add new payment methods or change existing ones without modifying the `PaymentProcessor` class. This approach promotes the principles of separation of concerns, encapsulation, and flexibility in your design.

Template Design Pattern : (Behavioral)

Requirements:

Let's consider a scenario where we are building a reporting system for a company. The system needs to generate different types of reports, such as sales reports, expense reports, and employee performance reports. Each report has a common structure that includes a header, body, and footer, but the content and formatting are specific to each report type. We want to design a system that allows us to create these reports with consistent structure while allowing customization for each report type.

Naive Solution:

A naive approach would be to create separate classes for each type of report, duplicating the common structure code in each class. Here's a simplified example in C++:

```
class SalesReport {
public:
    void generateReport() {
        // Common report header
        std::cout << "Sales Report Header" << std::endl;

        // Report-specific content
        std::cout << "Sales data..." << std::endl;

        // Common report footer
        std::cout << "Sales Report Footer" << std::endl;
    }
};

class ExpenseReport {
public:
    void generateReport() {
        // Common report header
        std::cout << "Expense Report Header" << std::endl;

        // Report-specific content
        std::cout << "Expense data..." << std::endl;

        // Common report footer
        std::cout << "Expense Report Footer" << std::endl;
    }
};
```

// Similar classes for other report types...

Issues with the Naive Solution:

1. Code duplication: The common report structure is repeated in each report class, violating the DRY (Don't Repeat Yourself) principle.

2. Lack of flexibility: It's challenging to make changes to the common structure or add new report types without modifying multiple classes.

Applying the Template Design Pattern:

The Template Method Design Pattern defines the skeleton of an algorithm in the base class but lets subclasses override specific steps of the algorithm without changing its structure. Here's how we can refactor the reporting system using this pattern:

Step 1: Create an Abstract Report Class (Template)

Define an abstract base class called `Report` that provides the common structure for all reports. It includes template methods that represent the common steps of the report generation algorithm.

```
class Report {
public:
    void generateReport() {
        // Common report header
        generateHeader();

        // Report-specific content
        generateContent();

        // Common report footer
        generateFooter();
    }
```

```
    virtual void generateHeader() = 0;
    virtual void generateContent() = 0;
    virtual void generateFooter() = 0;
};
```

Step 2: Create Concrete Report Subclasses

Create concrete report subclasses that inherit from the `Report` class and override the template methods to provide report-specific implementations.

```
class SalesReport : public Report {
public:
    void generateHeader() override {
        std::cout << "Sales Report Header" << std::endl;
    }

    void generateContent() override {
        std::cout << "Sales data..." << std::endl;
    }

    void generateFooter() override {
        std::cout << "Sales Report Footer" << std::endl;
    }
};

class ExpenseReport : public Report {
public:
    void generateHeader() override {
        std::cout << "Expense Report Header" << std::endl;
    }

    void generateContent() override {
        std::cout << "Expense data..." << std::endl;
    }

    void generateFooter() override {
        std::cout << "Expense Report Footer" << std::endl;
    }
```

```
};
```

// Similar classes for other report types...

Using the Template Design Pattern:

Now, you can create reports with consistent structure and customized content:

```
int main() {
    Report* salesReport = new SalesReport();
    salesReport->generateReport();
    delete salesReport;

    Report* expenseReport = new ExpenseReport();
    expenseReport->generateReport();
    delete expenseReport;

    // Create and use other report types...

    return 0;
}
```

Summary of the Template Design Pattern:

- Intent: The Template Method Design Pattern defines the skeleton of an algorithm in a base class but allows subclasses to provide specific implementations for some of its steps. It enforces a common structure while allowing customization.

- Advantages:

- Promotes code reuse by encapsulating common behavior in a base class.

- Ensures a consistent structure across different subclasses.

- Allows subclasses to customize specific parts of the algorithm.

In our example, we created an abstract `Report` class that defined the common steps of report generation. Subclasses then provided their own implementations for the header, content, and footer, allowing us to generate different types of reports with a consistent structure. This adheres to the principles of the Template Method Design Pattern.

Another Example :

Requirements:

Imagine you are developing a simple web page rendering framework. You want to create a structure for rendering web pages with common elements like a header, body, and footer. However, the content and styling of each web page can vary.

Using Template Design Pattern :

```
class WebPageRenderer {
public:
    void render() {
        // Common rendering code for header
        renderHeader();

        // Render specific content
        renderContent();

        // Common rendering code for footer
        renderFooter();
    }
```

```
    virtual void renderContent() = 0;

    void renderHeader() {
        // Rendering code for header
    }

    void renderFooter() {
        // Rendering code for footer
    }
};

class HomePageRenderer : public WebPageRenderer {
public:
    void renderContent() override {
        // Rendering code for home page content
    }
};

class AboutPageRenderer : public WebPageRenderer {
public:
    void renderContent() override {
        // Rendering code for about page content
    }
};
int main() {
    WebPageRenderer* homePage = new HomePageRenderer();
    homePage->render();
    delete homePage;

    WebPageRenderer* aboutPage = new AboutPageRenderer();
    aboutPage->render();
    delete aboutPage;

    // Create and use other web page renderers...

    return 0;
}
```

Comparison of Behavioral Design Patterns :

Design Pattern	Description	Use-case
Observer Design Pattern	The Observer pattern allows one object (the subject) to notify multiple dependent objects (observers) of changes in its state, ensuring loose coupling between subjects and observers.	Real-time Stock Market Updates - Multiple investors (observers) subscribe to stock market updates from a brokerage (subject). When stock prices change, all subscribers receive real-time notifications
Command Design Pattern	The Command pattern encapsulates a request as an object, allowing for parameterization of clients with queues, requests, and operations. It supports undo and redo operations and decouples senders and receivers.	Text Editor Commands (Copy, Paste, Undo) - A text editor allows users to perform actions like copying, pasting, and undoing. These actions are encapsulated as command objects, making them easily queueable and undoable.
Chain of Responsibility Design Pattern	The Chain of Responsibility pattern passes a request along a chain of handlers, with each handler deciding whether to process the request or pass it to the next handler in the chain. It decouples senders and receivers.	In image processing, implementing a series of image filters or processing steps where each filter can handle a specific task, and the image passes through a chain of these filters sequentially. Each filter can choose to modify the image, pass it along to the next filter, or terminate the processing chain.
Iterator Design Pattern	The Iterator pattern provides a way to access elements of an aggregate object (e.g., a list or collection) sequentially without exposing its underlying representation.	developing a media player application that allows users to browse and play music. Users want the ability to browse their music library by various criteria, such as by artist, alphabetically by song title, or by genre.
Strategy Design Pattern	The Strategy pattern defines a family of algorithms, encapsulates each one, and makes them	Sorting Algorithms (e.g., Bubble, Quick, Merge) - A sorting algorithm strategy interface allows the client to

	interchangeable. Clients can choose the appropriate algorithm at runtime	select the desired sorting algorithm (strategy) dynamically, depending on the dataset's characteristics
Template Design Pattern	The Template pattern is used to define the skeleton of an algorithm in a method but allows subclasses to override specific steps of the algorithm without changing its structure. It promotes code reusability.	A report generation template provides a standardized structure for generating different types of reports (e.g., HTML or PDF), allowing subclasses to customize content while maintaining a consistent structure.

Adapter Design Pattern :

The Adapter Pattern is a **structural design** pattern that allows the interface of an existing class to be used as another interface. It lets classes work together that couldn't otherwise because of incompatible interfaces. The Adapter Pattern is particularly useful when you want to integrate existing classes into a new system without modifying their source code.

There are two common variations of the Adapter Pattern: the Class Adapter and the Object Adapter. I'll explain both with code examples in C++.

Class Adapter Pattern:

In the Class Adapter Pattern, you use multiple inheritance to adapt one interface into another. This approach is less common in C++ due to potential issues with ambiguity when inheriting from multiple classes.

Example:

Suppose you have an `OldSystem` class with a method called `legacyOperation()`, and you want to adapt it to a new interface `NewInterface` with a method called `newOperation()`:

// Old system class with the legacy operation

```
class OldSystem {
public:
    void legacyOperation() {
        std::cout << "Legacy operation" << std::endl;
    }
};
```

// New interface

```
class NewInterface {
public:
    virtual void newOperation() = 0;
};

// Class Adapter: Adapts OldSystem to NewInterface
class ClassAdapter : public OldSystem, public NewInterface {
public:
    void newOperation() override {
        legacyOperation();
    }
};
```

Usage:

```
int main() {
    NewInterface* adapter = new ClassAdapter();
    adapter->newOperation();
```

```
    delete adapter;
    return 0;
}
```

Object Adapter Pattern:

In the Object Adapter Pattern, you use composition to adapt one interface into another. This approach is more common in C++ and avoids the ambiguity issue associated with multiple inheritance.

Example:

Let's adapt `OldSystem` to `NewInterface` using the Object Adapter Pattern:

```
// Object Adapter: Adapts OldSystem to NewInterface
class ObjectAdapter : public NewInterface {
private:
    OldSystem oldSystem;

public:
    void newOperation() override {
        oldSystem.legacyOperation();
    }
};
```

Usage:

```
int main() {
    NewInterface* adapter = new ObjectAdapter();
    adapter->newOperation();
    delete adapter;
    return 0;
```

```
}
```

Key Differences:

1. Inheritance vs. Composition:

- Class Adapter: Uses multiple inheritance to adapt the class. It inherits both the old and new interfaces.

- Object Adapter: Uses composition to adapt the class. It holds an instance of the old class and delegates calls to it.

2. Ambiguity:

- Class Adapter: Can potentially suffer from ambiguity issues when multiple base classes have the same methods or members.

- Object Adapter: Avoids ambiguity issues because it doesn't inherit from the old class.

In most cases, the Object Adapter pattern is preferred because it avoids ambiguity problems and promotes a more flexible and maintainable design. It also adheres to the principle of "**favor composition over inheritance**."

Example :

A common real-world example where the Adapter Pattern can be applied is when integrating external APIs or libraries into an existing software system. These APIs or libraries may have different interfaces or conventions that need to be adapted to the system's requirements without modifying their source code.

Let's consider a scenario involving geographical data. Imagine you are building a mapping application that uses data from two different geographical data providers: Provider A and Provider B. Provider A represents geographical data in latitude and longitude coordinates, while Provider B uses a different format, such as longitude and latitude coordinates. Your application expects data in a consistent format.

Requirements:

1. The mapping application expects all geographical data in latitude and longitude coordinates (e.g., `double latitude, double longitude`).

2. Provider A offers data in latitude and longitude format.

3. Provider B offers data in longitude and latitude format.

Naive Solution:

A naive approach would be to modify the application's code to handle both formats directly. However, this would tightly couple the application to the two providers and make it challenging to switch or add new providers in the future.

Applying the Adapter Pattern:

To address this issue, you can create adapter classes for each provider to convert their data to the expected format in the application.

Step 1: Define the Target Interface:

Define an interface, `GeographicalDataProvider`, that represents the expected format for geographical data:

```
class GeographicalDataProvider {
public:
    virtual std::pair<double, double> getCoordinates() = 0;
};
```

Step 2: Create Provider Adapters:

Create adapter classes for Provider A and Provider B that implement the `GeographicalDataProvider` interface and adapt their data to the expected format:

```
class ProviderAAdapter : public GeographicalDataProvider {
private:
    ProviderA providerA;
public:
    ProviderAAdapter(ProviderA provider) : providerA(provider) {}

    std::pair<double, double> getCoordinates() override {
        auto data = providerA.getData();
        return std::make_pair(data.latitude, data.longitude);
    }
};

class ProviderBAdapter : public GeographicalDataProvider {
private:
    ProviderB providerB;

public:
    ProviderBAdapter(ProviderB provider) : providerB(provider) {}

    std::pair<double, double> getCoordinates() override {
        auto data = providerB.getData();
        return std::make_pair(data.longitude, data.latitude);
    }
};
```

Usage:

Now, you can use the adapters to seamlessly integrate data from both providers into your mapping application without worrying about the differences in data formats:

```
int main() {
    ProviderA providerAInstance;
    ProviderB providerBInstance;

    GeographicalDataProvider* adapterA = new
ProviderAAdapter(providerAInstance);
    GeographicalDataProvider* adapterB = new
ProviderBAdapter(providerBInstance);

    // Use the adapters to fetch and process data from both providers
    std::pair<double, double> coordinatesA = adapterA->getCoordinates();
    std::pair<double, double> coordinatesB = adapterB->getCoordinates();

    // Process the data in the expected format

    delete adapterA;
    delete adapterB;

    return 0;
}
```

Summary:

In this real-world example, the Adapter Pattern allows you to integrate data from two different geographical data providers into your mapping application seamlessly. By creating adapter classes, you can adapt the data from each provider to conform to the expected interface, promoting flexibility and maintainability in your application's design.

Decorator Design Pattern :

The Decorator Design Pattern is a structural design pattern that allows you to add behavior to objects dynamically without altering their class. It is often used to extend the functionalities of classes in a flexible and reusable way.

Intent of the Decorator Pattern:

The primary intent of the Decorator Pattern is to attach additional responsibilities to objects dynamically. It is a more flexible alternative to subclassing for extending behavior.

Requirements:

Let's illustrate the Decorator Design Pattern with a real-world example involving a coffee shop. Imagine you are building a system for ordering and customizing coffee drinks. Customers can order a basic coffee and then customize it with various toppings and extras such as milk, sugar, caramel, and whipped cream. The goal is to create a flexible and extensible system for creating customized coffee drinks.

Naive Solution:

A naive approach would be to create a separate class for each possible combination of coffee and toppings. For example, you might have classes like `BasicCoffee`, `CoffeeWithMilk`, `CoffeeWithSugar`, `CoffeeWithCaramel`, `CoffeeWithWhippedCream`, and so on.

```
class BasicCoffee {
public:
    std::string getDescription() {
        return "Basic Coffee";
    }
};

class CoffeeWithMilk {
```

```
public:
    std::string getDescription() {
        return "Coffee with Milk";
    }
};

// Similar classes for other coffee variations...
```

Issues with the Naive Solution:

1. The number of classes grows exponentially as you add more toppings and extras, leading to a maintenance nightmare.

2. Combinations like "Coffee with Milk and Sugar" would require additional classes, leading to code duplication.

Applying the Decorator Design Pattern:

The Decorator Design Pattern allows us to add responsibilities to objects dynamically, as opposed to static inheritance. Let's refactor the system using this pattern.

Step 1: Define the Component Interface:

Create an interface or abstract class that represents the basic component, in this case, a `Coffee`:

```
class Coffee {
public:
    virtual std::string getDescription() = 0;
    virtual double cost() = 0;
};
```

Step 2: Implement the Concrete Component:

Create a concrete class that implements the component interface, representing the basic coffee:

```
class BasicCoffee : public Coffee {
public:
    std::string getDescription() override {
        return "Basic Coffee";
    }

    double cost() override {
        return 2.0; // Base price for basic coffee
    }
};
```

Step 3: Create Decorators:

Define decorator classes that also implement the `Coffee` interface. Decorators will add new behavior (toppings) to the basic coffee. For example, here's a `MilkDecorator`:

```
class MilkDecorator : public Coffee {
private:
    Coffee* coffee;

public:
    MilkDecorator(Coffee* coffee) : coffee(coffee) {}

    std::string getDescription() override {
        return coffee->getDescription() + ", Milk";
    }

    double cost() override {
        return coffee->cost() + 0.5; // Add cost of milk
    }
};
```

Using the Decorator Pattern:

Now, you can create customized coffee drinks by stacking decorators:

```
int main() {
    Coffee* coffee = new BasicCoffee();
    coffee = new MilkDecorator(coffee);
    coffee = new SugarDecorator(coffee);
    coffee = new CaramelDecorator(coffee);

    std::cout << "Order: " << coffee->getDescription() << std::endl;
    std::cout << "Total Cost: $" << coffee->cost() << std::endl;

    delete coffee;

    return 0;
}
```

Summary of the Decorator Design Pattern:

- Intent: The Decorator Design Pattern allows behavior to be added to individual objects, either statically or dynamically, without affecting the behavior of other objects from the same class.

- Advantages:

  - Provides flexibility to add or remove responsibilities dynamically.

  - Promotes the principle of open/closed for extending classes.

  - Avoids the need for numerous subclasses for each combination.

In our coffee shop example, we created a flexible and extensible system that allows customers to customize their coffee drinks by stacking decorators. Each decorator adds a specific behavior (topping) to the basic coffee object, and the total cost and description are calculated dynamically. This adheres to the principles of the Decorator Design Pattern.

Another Example :

Requirements: Text Formatting

Imagine you're building a text editor, and you want to allow users to apply various formatting styles to their text, such as bold, italic, and underline. Instead of creating separate classes for each style, you can use the Decorator pattern to dynamically apply formatting to the text.

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

// Component Interface (Text)
class Text {
public:
    virtual std::string format() = 0;
};

// Concrete Component (PlainText)
class PlainText : public Text {
private:
    std::string content;

public:
    PlainText(const std::string& text) : content(text) {}

    std::string format() override {
        return content;
    }
};

// Decorator (TextDecorator)
class TextDecorator : public Text {
```

```
protected:
    Text* wrappedText;

public:
    TextDecorator(Text* text) : wrappedText(text) {}

    std::string format() override {
        return wrappedText->format();
    }
};

// Concrete Decorator (BoldText)
class BoldText : public TextDecorator {
public:
    BoldText(Text* text) : TextDecorator(text) {}

    std::string format() override {
        return "<b>" + TextDecorator::format() + "</b>";
    }
};

// Concrete Decorator (ItalicText)
class ItalicText : public TextDecorator {
public:
    ItalicText(Text* text) : TextDecorator(text) {}

    std::string format() override {
        return "<i>" + TextDecorator::format() + "</i>";
    }
};

int main() {
    Text* plainText = new PlainText("This is plain text.");
    Text* boldText = new BoldText(plainText);
    Text* italicText = new ItalicText(boldText);

    std::cout << "Formatted Text: " << italicText->format() <<
std::endl;

    delete italicText;
    delete boldText;
    delete plainText;

    return 0;
}
```

In this code:

1. We define a `Text` interface that represents the component (e.g., plain text).

2. The `PlainText` class is a concrete component that implements the `Text` interface.

3. The `TextDecorator` class is the decorator base class that also implements the `Text` interface. It contains a reference to a wrapped `Text` object.

4. `BoldText` and `ItalicText` are concrete decorator classes. They wrap a `Text` object and add `<b>` and `<i>` tags, respectively, to the formatted text.

5. In the `main` function, we create a chain of decorators (italic over bold over plain) to format the text.

The Decorator pattern allows you to add and combine formatting options dynamically without altering the original component (`PlainText` in this case).

Bridge Design Pattern :

Intent of the Bridge Design Pattern:

The Bridge Design Pattern is used to separate an object's abstraction from its implementation so that both can vary independently. It allows you to create a system with multiple dimensions of variability, making it easier to extend and maintain.

Story and Example: Remote Control for Electronic Devices

Requirements:

Imagine you are designing a universal remote control system for various electronic devices, including televisions and music players. Each electronic device can have multiple brands and models, and each brand/model can have different remote control features. Your goal is to

create a flexible remote control system that can support any electronic device and its specific features.

Naive Solution:

In a naive approach, you might create separate classes for each type of electronic device (e.g., `SonyTV`, `SamsungTV`, `SonyMusicPlayer`, `SamsungMusicPlayer`, etc.), each with its own set of remote control functions (e.g., `turnOn()`, `turnOff()`, `setVolume()`, `play()`, `pause()`, etc.). This approach can lead to a proliferation of classes and complex maintenance when new devices or features are introduced.

Better Design with Bridge Pattern:

To create a flexible and extensible remote control system, you can apply the Bridge Design Pattern. Here's how:

1. Identify Abstractions and Implementations:

- Define abstractions for remote control functions (e.g., `RemoteControl`) and for electronic devices (e.g., `Device`).

- Implementations represent the specific features of each electronic device (e.g., `SonyDevice`, `SamsungDevice`).

2. Create Abstraction and Implementor Base Classes:

// Abstraction for remote control functions

```
class RemoteControl {
public:
    RemoteControl(Device* device) : device(device) {}
```

```
    virtual void powerOn() = 0;
    virtual void powerOff() = 0;
    virtual void setVolume(int volume) = 0;
    virtual void play() = 0;

protected:
    Device* device;
};

// Abstraction for electronic devices
class Device {
public:
    virtual void turnOn() = 0;
    virtual void turnOff() = 0;
    virtual void setDeviceVolume(int volume) = 0;
    virtual void startPlaying() = 0;
    virtual void stopPlaying() = 0;
};
```

3. Implement Abstraction and Implementor Classes:

// Implementations for specific electronic devices

```
class SonyDevice : public Device {
public:
    void turnOn() override {
        // Implement the turnOn() for Sony devices
    }

    // Implement other device-specific functions
};

class SamsungDevice : public Device {
public:
    void turnOn() override {
        // Implement the turnOn() for Samsung devices
    }

    // Implement other device-specific functions
};
```

4. Create Refined Abstractions:

```
// Refined abstraction for remote control functions
class BasicRemoteControl : public RemoteControl {
public:
    BasicRemoteControl(Device* device) : RemoteControl(device) {}

    void powerOn() override {
        device->turnOn();
    }

    void powerOff() override {
        device->turnOff();
    }

    void setVolume(int volume) override {
        device->setDeviceVolume(volume);
    }

    void play() override {
        device->startPlaying();
    }
};
```

5. Client Code:

- Users can create remote controls and associate them with specific electronic devices.

```
int main() {
    Device* sonyTV = new SonyDevice();
    RemoteControl* sonyRemote = new BasicRemoteControl(sonyTV);

    sonyRemote->powerOn();
    sonyRemote->setVolume(20);
    sonyRemote->play();
```

```
    // Similar setup for Samsung devices

    delete sonyRemote;
    delete sonyTV;

    return 0;
}
```

Summary of the Bridge Design Pattern:

The Bridge Design Pattern separates the abstraction (e.g., remote control functions) from its implementation (e.g., electronic devices). This separation allows both the abstraction and implementation to vary independently, making the system more flexible and extensible. In this example, the Bridge Pattern enables the creation of a universal remote control system that can support various electronic devices with different features. It promotes code reuse, maintainability, and scalability.

Another Example :

Requirements:

Imagine you are designing a shape drawing system. You need to support various shapes like circles, squares, and triangles. Additionally, you want to draw these shapes using different rendering techniques, such as drawing them on a screen or printing them on paper.

Naive Solution:

In a naive approach, you might create separate classes for each combination of shape and rendering technique. For example, you could have classes like `ScreenCircle`, `ScreenSquare`, `ScreenTriangle`, `PrintCircle`, `PrintSquare`, and so on. This approach quickly becomes impractical as the number of shapes and rendering techniques increases.

Better Design with Bridge Design Pattern:

To achieve a more flexible and maintainable design, you can use the Bridge Design Pattern. Here's how:

1. Define Implementor Interfaces (Implementor): Create interfaces for both shapes and rendering techniques. These interfaces will define the methods that concrete implementations must implement.

```
// Shape Implementor Interface
class ShapeImplementor {
public:
    virtual void draw() = 0;
};

// Rendering Implementor Interface
class RenderingImplementor {
public:
    virtual void render() = 0;
};
```

2. Implement Concrete Implementations (Concrete Implementor): Implement concrete classes for both shapes and rendering techniques. Each class should inherit from its respective interface.

```
// Concrete Shape Implementations
class Circle : public ShapeImplementor {
public:
    void draw() override {
        std::cout << "Drawing Circle" << std::endl;
    }
};
```

```
class Square : public ShapeImplementor {
public:
    void draw() override {
        std::cout << "Drawing Square" << std::endl;
    }
};

// Concrete Rendering Implementations
class ScreenRenderer : public RenderingImplementor {
public:
    void render() override {
        std::cout << "Rendering on Screen" << std::endl;
    }
};

class PrintRenderer : public RenderingImplementor {
public:
    void render() override {
        std::cout << "Printing on Paper" << std::endl;
    }
};
```

3. Create Abstraction (Abstraction): Define an abstraction class that contains a reference to both the shape and rendering implementors. This abstraction can represent a shape drawing context.

```
class Shape {
protected:
    ShapeImplementor* shapeImpl;
    RenderingImplementor* renderingImpl;

public:
    Shape(ShapeImplementor* shape, RenderingImplementor* rendering) :
shapeImpl(shape), renderingImpl(rendering) {}

    virtual void drawShape() {
```

```
        shapeImpl->draw();
        renderingImpl->render();
    }
};
```

4. Create Refined Abstractions (Refined Abstraction): Implement specific shapes by extending the `Shape` class.

```
class CircleShape : public Shape {
public:
    CircleShape(ShapeImplementor* shape, RenderingImplementor*
rendering) : Shape(shape, rendering) {}

    void drawShape() override {
        std::cout << "Drawing a ";
        Shape::drawShape();
    }
};

class SquareShape : public Shape {
public:
    SquareShape(ShapeImplementor* shape, RenderingImplementor*
rendering) : Shape(shape, rendering) {}

    void drawShape() override {
        std::cout << "Drawing a ";
        Shape::drawShape();
    }
};
```

5. Client Code:

- Clients can create different shapes and rendering techniques.

- Clients can create specific shape objects and draw them using different rendering techniques.

```
int main() {
    ShapeImplementor* circle = new Circle();
    ShapeImplementor* square = new Square();

    RenderingImplementor* screenRenderer = new ScreenRenderer();
    RenderingImplementor* printRenderer = new PrintRenderer();

    Shape* circleOnScreen = new CircleShape(circle, screenRenderer);
    Shape* squareOnPaper = new SquareShape(square, printRenderer);

    circleOnScreen->drawShape();
    squareOnPaper->drawShape();

    delete circle;
    delete square;
    delete screenRenderer;
    delete printRenderer;
    delete circleOnScreen;
    delete squareOnPaper;

    return 0;
}
```

Summary of the Bridge Design Pattern:

The Bridge Design Pattern separates abstraction from implementation, allowing them to vary independently. It promotes flexibility and maintainability by avoiding a permanent binding between an abstraction and its implementation. In this example, the Bridge Pattern allows you to create different shapes and rendering techniques and combine them dynamically to achieve various drawing results. It also makes it easy to add new shapes or rendering techniques without modifying existing code.

Composite Design Pattern :

Intent of the Composite Design Pattern:

The Composite Design Pattern is used to compose objects into tree structures to represent part-whole hierarchies. It allows clients to treat individual objects and compositions of objects uniformly. The primary intent is to create a unified interface for both individual objects and compositions, making them interchangeable.

Story and Example: Building an Organization Hierarchy

Requirements:

Imagine you are developing an HR management system for an organization. The organization has a hierarchical structure consisting of departments, teams, and employees. You need to represent this hierarchy in your software to manage employees' information and access.

Naive Solution:

In a naive approach, you might create classes for departments, teams, and employees separately. For example, you could have classes like `Department`, `Team`, and `Employee`. However, this approach becomes challenging when you want to treat departments and

teams collectively and perform operations on them, such as calculating total salaries for a department.

Better Design with Composite Design Pattern:

To achieve a more flexible and maintainable design, you can use the Composite Design Pattern. Here's how:

1. Define a Component (EmployeeComponent): Create an abstract base class or interface that represents both individual objects (employees) and compositions (departments and teams). This component defines a common interface for all elements in the hierarchy.

```
class EmployeeComponent {
public:
    virtual void displayInfo() const = 0;
    virtual double calculateSalary() const = 0;
};
```

2. Implement Leaf (Employee): Create a leaf class that represents individual objects (employees). This class should inherit from the `EmployeeComponent` interface.

```
class Employee : public EmployeeComponent {
private:
    std::string name;
    double salary;

public:
    Employee(const std::string& empName, double empSalary) :
name(empName), salary(empSalary) {}

    void displayInfo() const override {
        std::cout << "Employee: " << name << " Salary: $" << salary <<
std::endl;
    }

    double calculateSalary() const override {
        return salary;
    }
};
```

3. Implement Composite (Department or Team): Create composite classes that represent compositions (departments and teams). These classes should also inherit from the `EmployeeComponent` interface.

```
#include <vector>

class Department : public EmployeeComponent {
private:
    std::string name;
    std::vector<EmployeeComponent*> members;

public:
    Department(const std::string& deptName) : name(deptName) {}

    void addMember(EmployeeComponent* member) {
```

```
        members.push_back(member);
    }

    void displayInfo() const override {
        std::cout << "Department: " << name << std::endl;
        for (const auto& member : members) {
            member->displayInfo();
        }
    }

    double calculateSalary() const override {
        double totalSalary = 0.0;
        for (const auto& member : members) {
            totalSalary += member->calculateSalary();
        }
        return totalSalary;
    }
};
```

4. Client Code:

- Clients can create individual employees, departments, and teams.

- Clients can treat both individual objects and compositions uniformly.

```
int main() {
    EmployeeComponent* john = new Employee("John", 50000.0);
    EmployeeComponent* jane = new Employee("Jane", 60000.0);

    Department* sales = new Department("Sales");
    sales->addMember(john);
    sales->addMember(jane);

    EmployeeComponent* bob = new Employee("Bob", 70000.0);
```

```
    Team* marketing = new Team("Marketing");
    marketing->addMember(bob);

    Department* headOffice = new Department("Head Office");
    headOffice->addMember(sales);
    headOffice->addMember(marketing);

    // Display and calculate total salary for the organization hierarchy
    headOffice->displayInfo();
    double totalSalary = headOffice->calculateSalary();
    std::cout << "Total Salary for the Organization: $" << totalSalary
<< std::endl;

    // Clean up memory (consider using smart pointers in a real
application)
    delete john;
    delete jane;
    delete sales;
    delete bob;
    delete marketing;
    delete headOffice;

    return 0;
}
```

Summary of the Composite Design Pattern:

The Composite Design Pattern allows you to compose objects into tree structures, making it possible to treat individual objects and compositions uniformly. It promotes flexibility and maintainability by creating a unified interface for both leaf objects and composite objects. In this example, the Composite Pattern enables you to represent the organization hierarchy and perform operations on it, such as displaying employee information and calculating total salaries, in a consistent manner.

Another Example :

Requirements:

Imagine you are designing a file management system where users can organize files and directories. Users should be able to create files, directories, and nested directories. They should also be able to perform operations like listing the contents of directories and calculating the total size of a directory.

Naive Solution:

In a naive approach, you might create separate classes for files and directories. For example, you could have classes like `File` and `Directory`. However, this approach becomes cumbersome when you want to treat directories and files collectively, such as calculating the total size of a directory containing subdirectories and files.

Better Design with Composite Design Pattern:

To achieve a more flexible and maintainable design, you can use the Composite Design Pattern. Here's how:

1. Define a Component (FileSystemComponent): Create an abstract base class or interface that represents both individual objects (files) and compositions (directories). This component defines a common interface for all elements in the hierarchy.

```
class FileSystemComponent {
public:
```

```
    virtual void listContents() const = 0;
    virtual int getSize() const = 0;
};
```

2. Implement Leaf (File): Create a leaf class that represents individual objects (files). This class should inherit from the `FileSystemComponent` interface.

```
class File : public FileSystemComponent {
private:
    std::string name;
    int size;

public:
    File(const std::string& fileName, int fileSize) : name(fileName),
size(fileSize) {}

    void listContents() const override {
        std::cout << "File: " << name << std::endl;
    }

    int getSize() const override {
        return size;
    }
};
```

3. Implement Composite (Directory): Create a composite class that represents compositions (directories). This class should also inherit from the `FileSystemComponent` interface.

```
#include <vector>

class Directory : public FileSystemComponent {
private:
    std::string name;
    std::vector<FileSystemComponent*> contents;

public:
    Directory(const std::string& dirName) : name(dirName) {}
```

```
    void addComponent(FileSystemComponent* component) {
        contents.push_back(component);
    }

    void listContents() const override {
        std::cout << "Directory: " << name << std::endl;
        for (const auto& component : contents) {
            component->listContents();
        }
    }

    int getSize() const override {
        int totalSize = 0;
        for (const auto& component : contents) {
            totalSize += component->getSize();
        }
        return totalSize;
    }
};
```

4. Client Code:

- Clients can create individual files, directories, and nested directories.

- Clients can treat both individual objects and compositions uniformly.

```
int main() {
    FileSystemComponent* root = new Directory("Root");

    FileSystemComponent* file1 = new File("Document.txt", 100);
    FileSystemComponent* file2 = new File("Image.jpg", 200);

    Directory* subDir = new Directory("Subdirectory");
    FileSystemComponent* file3 = new File("Data.csv", 150);

    subDir->addComponent(file3);

    root->addComponent(file1);
```

```
    root->addComponent(file2);
    root->addComponent(subDir);

    // List contents and calculate total size for the directory
structure
    root->listContents();
    int totalSize = root->getSize();
    std::cout << "Total Size: " << totalSize << " KB" << std::endl;

    // Clean up memory (consider using smart pointers in a real
application)
    delete root;
    delete file1;
    delete file2;
    delete file3;
    delete subDir;

    return 0;
}
```

Summary of the Composite Design Pattern:

The Composite Design Pattern allows you to compose objects into tree structures, making it possible to treat individual objects and compositions uniformly. It promotes flexibility and maintainability by creating a unified interface for both leaf objects and composite objects. In this example, the Composite Pattern enables you to represent a file and directory structure and perform operations like listing contents and calculating total sizes in a consistent manner.

Proxy Design Pattern:

The Proxy Design Pattern provides a surrogate or placeholder for another object to control access to it. It can be used to add an extra layer of control over the interaction with the real object. The primary intent is to manage access to the real object, which might involve lazy initialization, access control, logging, or monitoring.

Story and Example: Controlling Access to a Sensitive Document

Requirements:

Imagine you are building a document management system that stores sensitive documents. Users should be able to view these documents, but you want to add an extra layer of control to track and restrict access to them. Specifically, you need to ensure that only authorized users can view the documents, and you want to log each access attempt.

Naive Solution:

In a naive approach, you might implement document access directly in the application logic. For instance, you could add access control checks and logging code each time a document is accessed. However, this approach can lead to code duplication and lack of separation of concerns.

Better Design with Proxy Design Pattern:

To achieve a more organized and maintainable solution, you can use the Proxy Design Pattern. Here's how:

1. Define a Subject Interface (Document): Create an interface that both the real object (SensitiveDocument) and its proxy (DocumentProxy) will implement. This interface should include methods for viewing the document.

```
class Document {
public:
    virtual void view() = 0;
};
```

2. Implement the Real Object (SensitiveDocument): Create a class that represents the real object (sensitive document) and implements the Subject interface. This class contains the actual document content.

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

class SensitiveDocument : public Document {
private:
    std::string content;

public:
    SensitiveDocument(const std::string& docContent) :
content(docContent) {}

    void view() override {
        std::cout << "Viewing sensitive document content:\n" << content
<< std::endl;
```

```
    }
};
```

3. Implement the Proxy (DocumentProxy): Create a proxy class that also implements the Subject interface. This class will control access to the real object and add any extra functionality, such as access control and logging.

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

class DocumentProxy : public Document {
private:
    SensitiveDocument* realDocument;
    std::string username;

public:
    DocumentProxy(const std::string& docContent, const std::string&
user)
        : realDocument(new SensitiveDocument(docContent)),
username(user) {}

    void view() override {
        // Implement access control logic here
        if (username == "admin") {
            realDocument->view();
            std::cout << "Access granted for user: " << username <<
std::endl;
            // Implement logging here
            std::cout << "Logging access for user: " << username <<
std::endl;
        } else {
            std::cout << "Access denied for user: " << username <<
std::endl;
            // Implement logging here
            std::cout << "Logging access attempt for user: " << username
<< std::endl;
        }
    }
};
```

4. Client Code:

- Clients can create DocumentProxy objects and use them to view documents.

- Access control and logging are automatically handled by the proxy.

```
int main() {
    // Create a sensitive document proxy for an admin user
    Document* adminDocument = new DocumentProxy("Top-secret content",
"admin");

    // Attempt to view the document
    adminDocument->view();

    // Create a sensitive document proxy for a regular user
    Document* userDocument = new DocumentProxy("Confidential content",
"user123");

    // Attempt to view the document
    userDocument->view();

    delete adminDocument;
    delete userDocument;

    return 0;
}
```

Summary of the Proxy Design Pattern:

The Proxy Design Pattern provides a way to control access to a real object by introducing a proxy that implements the same interface as the real object. The proxy can perform additional tasks such as access control, logging, or lazy initialization while maintaining a

consistent interface. In this example, the Proxy Pattern is used to control access to sensitive documents, ensuring that only authorized users can view them and logging access attempts.

Another Example :

Controlling Access to a Movie Streaming Service

Requirements:

Imagine you are building a movie streaming service like Netflix. Users should be able to stream movies, but you want to add an extra layer of control to monitor user behavior and restrict access to certain content based on age ratings. Specifically, you need to ensure that only users of a certain age can watch restricted movies, and you want to log each movie viewing.

Naive Solution:

In a naive approach, you might implement movie streaming directly in the application logic. For instance, you could add age verification checks and logging code each time a movie is streamed. However, this approach can lead to code duplication and lack of separation of concerns.

Better Design with Proxy Design Pattern:

To achieve a more organized and maintainable solution, you can use the Proxy Design Pattern. Here's how:

1. Define a Subject Interface (Movie): Create an interface that both the real object (MovieImpl) and its proxy (MovieProxy) will implement. This interface should include methods for streaming the movie.

```
class Movie {
public:
    virtual void stream() = 0;
};
```

2. Implement the Real Object (MovieImpl): Create a class that represents the real object (movie) and implements the Subject interface. This class contains the actual movie content and age rating information.

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

class MovieImpl : public Movie {
private:
    std::string title;
    int ageRating;

public:
    MovieImpl(const std::string& movieTitle, int movieAgeRating)
        : title(movieTitle), ageRating(movieAgeRating) {}

    void stream() override {
        std::cout << "Streaming movie: " << title << std::endl;
    }

    int getAgeRating() const {
        return ageRating;
    }
};
```

3. Implement the Proxy (MovieProxy): Create a proxy class that also implements the Subject interface. This class will control access to the real object and add any extra functionality, such as age verification and logging.

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

class MovieProxy : public Movie {
private:
    MovieImpl* realMovie;
    int userAge;

public:
    MovieProxy(const std::string& movieTitle, int movieAgeRating, int
userAge)
        : realMovie(new MovieImpl(movieTitle, movieAgeRating)),
userAge(userAge) {}

    void stream() override {
        // Implement age verification logic here
        if (userAge >= realMovie->getAgeRating()) {
            realMovie->stream();
            std::cout << "Access granted for age " << userAge << " to
movie: " << realMovie->getTitle() << std::endl;
            // Implement logging here
            std::cout << "Logging movie stream for user with age " <<
userAge << std::endl;
        } else {
            std::cout << "Access denied for age " << userAge << " to
movie: " << realMovie->getTitle() << std::endl;
            // Implement logging here
            std::cout << "Logging access attempt for user with age " <<
userAge << std::endl;
        }
    }

    std::string getTitle() const {
        return realMovie->getTitle();
    }
};


4. Client Code:
   - Clients can create MovieProxy objects and use them to stream
movies.
   - Age verification and logging are automatically handled by the
```

```
proxy.


int main() {
    // Create a movie proxy for an adult user
    Movie* adultMovie = new MovieProxy("Action Movie", 18, 25);

    // Attempt to stream the movie
    adultMovie->stream();

    // Create a movie proxy for a child user
    Movie* childMovie = new MovieProxy("Animated Movie", 6, 8);

    // Attempt to stream the movie
    childMovie->stream();

    delete adultMovie;
    delete childMovie;

    return 0;
}
```

Facade Design Pattern :

Intent of the Facade Design Pattern:

The Facade Design Pattern provides a unified interface to a set of interfaces in a subsystem. It defines a higher-level interface that makes it easier to use the subsystem, reducing the complexity and dependencies for clients. The primary intent is to simplify the use of complex subsystems by providing a single entry point.

Story and Example: Booking a Flight

Requirements:

Imagine you are developing a flight booking system. Users should be able to search for flights, make reservations, and receive booking confirmations. Behind the scenes, the system interacts with various components, such as flight search engines, seat availability databases, and payment gateways. You want to simplify the process for users by providing a unified interface.

Naive Solution:

In a naive approach, you might expose all the underlying components directly to the users. For example, users would need to interact with flight search engines, check seat availability, and handle payment processing themselves. This approach would be complex and error-prone for users.

Better Design with Facade Design Pattern:

To simplify the user experience and hide the complexities of the subsystem, you can use the Facade Design Pattern. Here's how:

1. Identify Subsystem Components: Identify the various subsystem components that users need to interact with, such as flight search, seat availability, and payment processing.

2. Create a Facade (FlightBookingFacade): Design a facade class that provides a simplified interface for users to interact with the subsystem. This class should encapsulate the complex interactions with the subsystem components.

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

class FlightSearchEngine {
public:
    void searchFlights(const std::string& source, const std::string&
destination) {
        std::cout << "Searching for flights from " << source << " to "
<< destination << std::endl;
        // Implementation details for flight search
    }
};

class SeatAvailability {
public:
    bool checkAvailability(const std::string& flightNumber, int
numberOfSeats) {
        std::cout << "Checking seat availability for flight " <<
flightNumber << std::endl;
        // Implementation details for seat availability check
        return true; // Placeholder result
    }
};

class PaymentGateway {
public:
    bool makePayment(double amount, const std::string& cardNumber) {
        std::cout << "Processing payment of $" << amount << " with card
number " << cardNumber << std::endl;
        // Implementation details for payment processing
        return true; // Placeholder result
    }
};

class FlightBookingFacade {
private:
    FlightSearchEngine flightSearchEngine;
    SeatAvailability seatAvailability;
    PaymentGateway paymentGateway;

public:
    bool bookFlight(const std::string& source, const std::string&
destination, const std::string& flightNumber, int numberOfSeats, double
amount, const std::string& cardNumber) {
        // Simplify the booking process for users
        flightSearchEngine.searchFlights(source, destination);
```

```
        bool seatsAvailable =
seatAvailability.checkAvailability(flightNumber, numberOfSeats);
        if (seatsAvailable) {
            bool paymentSuccessful = paymentGateway.makePayment(amount,
cardNumber);
            if (paymentSuccessful) {
                std::cout << "Booking confirmed for flight " <<
flightNumber << std::endl;
                return true;
            } else {
                std::cout << "Payment failed. Booking canceled." <<
std::endl;
                return false;
            }
        } else {
            std::cout << "Seats not available. Booking canceled." <<
std::endl;
            return false;
        }
    }
};
```

3. Client Code:

- Clients can use the `FlightBookingFacade` to book flights without needing to interact directly with subsystem components.

```
int main() {
    FlightBookingFacade bookingSystem;

    // User books a flight
    bool bookingResult = bookingSystem.bookFlight("New York", "Los
Angeles", "ABC123", 2, 500.0, "1234-5678-9012-3456");

    if (bookingResult) {
        std::cout << "Booking was successful!" << std::endl;
    } else {
        std::cout << "Booking failed." << std::endl;
    }
```

```
    return 0;
}
```

Summary of the Facade Design Pattern:

The Facade Design Pattern simplifies the use of complex subsystems by providing a unified and simplified interface for clients. It hides the complexities of the subsystem and reduces dependencies, making it easier for clients to interact with the system. In this example, the Facade Pattern simplifies the flight booking process by encapsulating interactions with flight search, seat availability, and payment processing components.

Another Example :

Ordering a Fast Food Meal

Requirements:

Imagine you are building a fast food ordering system for a restaurant. Customers should be able to order meals, which consist of various food items (burgers, fries, drinks, etc.). Behind the scenes, the system interacts with multiple subsystems, such as the kitchen, inventory management, and billing. You want to simplify the process for customers by providing a user-friendly ordering experience.

Naive Solution:

In a naive approach, you might expose all the underlying subsystem components directly to customers. For example, customers would need to place individual orders for each food item, check availability, and manage payments themselves. This approach would be cumbersome and confusing for customers.

Better Design with Facade Design Pattern:

To simplify the user experience and hide the complexities of the subsystem, you can use the Facade Design Pattern. Here's how:

1. Identify Subsystem Components: Identify the various subsystem components that customers need to interact with, such as food items, inventory management, and billing.

2. Create a Facade (FastFoodFacade): Design a facade class that provides a simplified interface for customers to interact with the subsystem. This class should encapsulate the complex interactions with the subsystem components.

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

class FoodItem {
public:
    void prepare() {
        std::cout << "Preparing the food item..." << std::endl;
    }
};

class InventoryManagement {
public:
    bool checkAvailability(const std::string& item) {
        std::cout << "Checking availability of " << item << "..." <<
std::endl;
        // Implementation details for checking availability
        return true; // Placeholder result
    }
};

class BillingSystem {
public:
    void processPayment(double amount) {
        std::cout << "Processing payment of $" << amount << "..." <<
std::endl;
```

```
        // Implementation details for payment processing
    }
};

class FastFoodFacade {
private:
    FoodItem foodItem;
    InventoryManagement inventory;
    BillingSystem billing;

public:
    bool orderMeal(const std::string& item, double amount) {
        // Simplify the ordering process for customers
        if (inventory.checkAvailability(item)) {
            foodItem.prepare();
            billing.processPayment(amount);
            std::cout << "Enjoy your " << item << "!" << std::endl;
            return true;
        } else {
            std::cout << item << " is not available. Order canceled." <<
std::endl;
            return false;
        }
    }
};


3. Client Code:
   - Customers can use the `FastFoodFacade` to order meals without
needing to interact directly with subsystem components.


int main() {
    FastFoodFacade fastFoodService;

    // Customer orders a burger
    bool orderResult = fastFoodService.orderMeal("Burger", 5.99);

    if (orderResult) {
        std::cout << "Order was successful!" << std::endl;
    } else {
        std::cout << "Order failed." << std::endl;
    }

    return 0;
}
```

STATE PATTERN :

Story: Implementing a Vending Machine

Requirements:

1. The vending machine has different states: `Idle`, `PaymentPending`, `Dispensing`, and `OutOfStock`.

2. In the `Idle` state, users can view the available items and make selections.

3. When a user selects an item and initiates payment, the machine transitions to the `PaymentPending` state.

4. In the `PaymentPending` state, the machine awaits payment confirmation.

5. If payment is successful, the machine transitions to the `Dispensing` state, and the selected item is dispensed.

6. If payment fails, the machine remains in the `PaymentPending` state.

7. If an item is out of stock, the machine transitions to the `OutOfStock` state, and no further transactions are allowed.

Naive Solution:

One way to implement this is to use a simple switch statement to manage the state transitions and actions for each state. However, this approach can quickly become complex and hard to maintain as more states and transitions are added.

Here's a simplified example of a naive solution in C++:

cpp

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

enum class VendingMachineState { Idle, PaymentPending, Dispensing,
OutOfStock };

class VendingMachine {
public:
    VendingMachine() : state(VendingMachineState::Idle) {}

    void selectItem(const std::string& item) {
        if (state == VendingMachineState::Idle) {
            selectedItem = item;
            state = VendingMachineState::PaymentPending;
            std::cout << "Selected: " << item << std::endl;
        } else {
            std::cout << "Cannot select item in the current state." <<
std::endl;
        }
    }

    void confirmPayment(bool success) {
        if (state == VendingMachineState::PaymentPending) {
            if (success) {
                state = VendingMachineState::Dispensing;
                std::cout << "Payment successful. Dispensing item: " <<
selectedItem << std::endl;
            } else {
                state = VendingMachineState::PaymentPending;
                std::cout << "Payment failed. Please retry." <<
std::endl;
            }
        } else {
            std::cout << "Cannot confirm payment in the current state."
<< std::endl;
        }
```

```
    }

    void dispenseItem() {
        if (state == VendingMachineState::Dispensing) {
            state = VendingMachineState::Idle;
            std::cout << "Item dispensed. Vending machine is now idle."
<< std::endl;
        } else {
            std::cout << "Cannot dispense item in the current state." <<
std::endl;
        }
    }

    void reportOutOfStock() {
        state = VendingMachineState::OutOfStock;
        std::cout << "Item is out of stock. Vending machine is now out
of stock." << std::endl;
    }

private:
    VendingMachineState state;
    std::string selectedItem;
};

int main() {
    VendingMachine vendingMachine;
    vendingMachine.selectItem("Soda");
    vendingMachine.confirmPayment(true);
    vendingMachine.dispenseItem();
    return 0;
}
```

Better Design with State Design Pattern:

The State Design Pattern is a behavioral pattern that allows an object to alter its behavior when its internal state changes. It encapsulates states into separate classes, making it easier to add new states and transitions.

Let's redesign the vending machine using the State Design Pattern:

State.hpp:

cpp

```cpp
#ifndef STATE_HPP
#define STATE_HPP

class VendingMachine;

class State {
public:
    virtual void selectItem(VendingMachine* context, const std::string&
item) = 0;
    virtual void confirmPayment(VendingMachine* context, bool success) =
0;
    virtual void dispenseItem(VendingMachine* context) = 0;
};

class IdleState : public State {
public:
    static IdleState& instance() {
        static IdleState instance;
        return instance;
    }

    void selectItem(VendingMachine* context, const std::string& item)
override;
    void confirmPayment(VendingMachine* context, bool success) override;
    void dispenseItem(VendingMachine* context) override;
};

class PaymentPendingState : public State {
public:
    static PaymentPendingState& instance() {
        static PaymentPendingState instance;
        return instance;
    }

    void selectItem(VendingMachine* context, const std::string& item)
override;
    void confirmPayment(VendingMachine* context, bool success) override;
    void dispenseItem(VendingMachine* context) override;
};
```

```
class DispensingState : public State {
public:
    static DispensingState& instance() {
        static DispensingState instance;
        return instance;
    }

    void selectItem(VendingMachine* context, const std::string& item)
override;
    void confirmPayment(VendingMachine* context, bool success) override;
    void dispenseItem(VendingMachine* context) override;
};

class OutOfStockState : public State {
public:
    static OutOfStockState& instance() {
        static OutOfStockState instance;
        return instance;
    }

    void selectItem(VendingMachine* context, const std::string& item)
override;
    void confirmPayment(VendingMachine* context, bool success) override;
    void dispenseItem(VendingMachine* context) override;
};

#endif
```

State.cpp:

```
cpp
#include "State.hpp"
#include "VendingMachine.hpp"
#include <iostream>

void IdleState::selectItem(VendingMachine* context, const std::string&
item) {
    context->setSelectedItem(item);
    std::cout << "Selected: " << item << std::endl;
    context->transitionTo(PaymentPendingState::instance());
}

void IdleState::confirmPayment(VendingMachine* context, bool success) {
    std::cout << "Please select an item before confirming payment." <<
```

```
std::endl;
}

void IdleState::dispenseItem(VendingMachine* context) {
    std::cout << "No item selected for dispensing." << std::endl;
}

void PaymentPendingState::selectItem(VendingMachine* context, const
std::string& item) {
    std::cout << "An item is already selected. Please complete or cancel
the current transaction." << std::endl;
}

void PaymentPendingState::confirmPayment(VendingMachine* context, bool
success) {
    if (success) {
        context->dispenseItem();
    } else {
        std::cout << "Payment failed. Please retry." << std::endl;
    }
}

void PaymentPendingState::dispenseItem(VendingMachine* context) {
    std::cout << "Payment confirmation is pending. Please wait." <<
std::endl;
}

void DispensingState::selectItem(VendingMachine* context, const
std::string& item) {
    std::cout << "Dispensing in progress. Please wait." << std::endl;
}

void DispensingState::confirmPayment(VendingMachine* context, bool
success) {
    std::cout << "Payment confirmation is not required during item
dispensing." << std::endl;
}

void DispensingState::dispense

Item(VendingMachine* context) {
    context->completeTransaction();
}

void OutOfStockState::selectItem(VendingMachine* context, const
std::string& item) {
```

```
    std::cout << "The vending machine is out of stock. Please select
another item." << std::endl;
}

void OutOfStockState::confirmPayment(VendingMachine* context, bool
success) {
    std::cout << "The vending machine is out of stock. Payment is not
accepted." << std::endl;
}

void OutOfStockState::dispenseItem(VendingMachine* context) {
    std::cout << "The vending machine is out of stock. No items to
dispense." << std::endl;
}
```

VendingMachine.hpp:

```
cpp
#ifndef VENDINGMACHINE_HPP
#define VENDINGMACHINE_HPP

#include "State.hpp"
#include <string>

class VendingMachine {
public:
    VendingMachine();
    void selectItem(const std::string& item);
    void confirmPayment(bool success);
    void dispenseItem();
    void reportOutOfStock();

    void transitionTo(State& newState);
    void setSelectedItem(const std::string& item);
    void completeTransaction();

private:
    State* currentState;
    std::string selectedItem;
};

#endif
```

VendingMachine.cpp:

```
cpp
#include "VendingMachine.hpp"
#include "State.hpp"

VendingMachine::VendingMachine() : currentState(&IdleState::instance())
{}

void VendingMachine::selectItem(const std::string& item) {
    currentState->selectItem(this, item);
}

void VendingMachine::confirmPayment(bool success) {
    currentState->confirmPayment(this, success);
}

void VendingMachine::dispenseItem() {
    currentState->dispenseItem(this);
}

void VendingMachine::reportOutOfStock() {
    transitionTo(OutOfStockState::instance());
}

void VendingMachine::transitionTo(State& newState) {
    currentState = &newState;
}

void VendingMachine::setSelectedItem(const std::string& item) {
    selectedItem = item;
}

void VendingMachine::completeTransaction() {
    std::cout << "Item dispensed: " << selectedItem << std::endl;
    selectedItem.clear();
    transitionTo(IdleState::instance());
}


Main.cpp:

cpp
#include "VendingMachine.hpp"

int main() {
    VendingMachine vendingMachine;
```

```
    vendingMachine.selectItem("Soda");
    vendingMachine.confirmPayment(true);
    vendingMachine.dispenseItem();

    vendingMachine.selectItem("Chips");
    vendingMachine.reportOutOfStock();
    vendingMachine.selectItem("Candy");

    return 0;
}
```

Summary of the State Design Pattern:

The State Design Pattern allows an object to change its behavior when its internal state changes. It encapsulates each state into separate classes, making it easy to add new states and transitions without modifying the context class. In our vending machine example, the `VendingMachine` object can seamlessly switch between different states, such as `Idle`, `PaymentPending`, `Dispensing`, and `OutOfStock`, while maintaining a clean and maintainable code structure. This pattern promotes the Single Responsibility Principle and helps manage complex state-based behaviors.

Another Example :

Story: Implementing a TCP Connection Handler

Requirements:

1. The system must manage a TCP connection with various states, such as connecting, established, closed, and listening.

2. The system should allow the connection to transition between states appropriately.

3. Each state has different behaviors and operations.

Naive Solution:

In a naive implementation, you might use an enum or a set of flags to represent the state of the TCP connection and manage state transitions with if-else or switch-case statements. This approach can become complex and difficult to maintain as more states and transitions are added.

Here's a simplified example in C++:

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

enum class ConnectionState { Connecting, Established, Closed, Listening
};

class TCPConnection {
public:
    void setState(ConnectionState newState) {
        currentState = newState;
    }

    void open() {
        if (currentState == ConnectionState::Closed || currentState ==
ConnectionState::Listening) {
            setState(ConnectionState::Established);
            std::cout << "Connection established." << std::endl;
        } else {
            std::cout << "Invalid operation in the current state." <<
std::endl;
        }
    }

    void close() {
        if (currentState == ConnectionState::Established) {
            setState(ConnectionState::Closed);
            std::cout << "Connection closed." << std::endl;
```

```
        } else {
            std::cout << "Invalid operation in the current state." <<
std::endl;
        }
    }

    void listen() {
        if (currentState == ConnectionState::Closed) {
            setState(ConnectionState::Listening);
            std::cout << "Listening for incoming connections." <<
std::endl;
        } else {
            std::cout << "Invalid operation in the current state." <<
std::endl;
        }
    }

    void send(const std::string& data) {
        if (currentState == ConnectionState::Established) {
            std::cout << "Sent data: " << data << std::endl;
        } else {
            std::cout << "Invalid operation in the current state." <<
std::endl;
        }
    }
};

int main() {
    TCPConnection connection;
    connection.open();
    connection.send("Hello, Server!");
    connection.listen();
    connection.send("Hello, Client!");
    connection.close();

    return 0;
}
```

Challenges with the Naive Solution:

1. The code becomes complex and error-prone as the number of states and state transitions increases.

2. Adding a new state or managing complex state-related behaviors becomes difficult.

3. It violates the Open-Closed Principle, as you need to modify the existing code to accommodate new states or behaviors.

Improved Solution using the State Design Pattern:

The State Design Pattern helps address these challenges by encapsulating each state in a separate class, allowing you to manage state-specific behaviors more effectively.

State.hpp:

```
#ifndef STATE_HPP
#define STATE_HPP

class TCPConnection;

class ConnectionState {
public:
    virtual void open(TCPConnection* context) = 0;
    virtual void close(TCPConnection* context) = 0;
    virtual void listen(TCPConnection* context) = 0;
    virtual void send(TCPConnection* context, const std::string& data) =
0;
};

class ConnectingState : public ConnectionState {
public:
    static ConnectingState& instance() {
        static ConnectingState instance;
        return instance;
    }

    void open(TCPConnection* context) override;
    void close(TCPConnection* context) override;
    void listen(TCPConnection* context) override;
    void send(TCPConnection* context, const std::string& data) override;
};

class EstablishedState : public ConnectionState {
public:
    static EstablishedState& instance() {
        static EstablishedState instance;
```

```
        return instance;
    }

    void open(TCPConnection* context) override;
    void close(TCPConnection* context) override;
    void listen(TCPConnection* context) override;
    void send(TCPConnection* context, const std::string& data) override;
};

class ClosedState : public ConnectionState {
public:
    static ClosedState& instance() {
        static ClosedState instance;
        return instance;
    }

    void open(TCPConnection* context) override;
    void close(TCPConnection* context) override;
    void listen(TCPConnection* context) override;
    void send(TCPConnection* context, const std::string& data) override;
};

class ListeningState : public ConnectionState {
public:
    static ListeningState& instance() {
        static ListeningState instance;
        return instance;
    }

    void open(TCPConnection* context) override;
    void close(TCPConnection* context) override;
    void listen(TCPConnection* context) override;
    void send(TCPConnection* context, const std::string& data) override;
};

#endif
```

State.cpp:

```
#include "State.hpp"
#include "TCPConnection.hpp"
#include <iostream>

void ConnectingState::open(TCPConnection* context) {
    std::cout << "Already connecting..." << std::endl;
}
```

```
void ConnectingState::close(TCPConnection* context) {
    context->setState(ClosedState::instance());
    std::cout << "Connection closed." << std::endl;
}

void ConnectingState::listen(TCPConnection* context) {
    context->setState(ListeningState::instance());
    std::cout << "Now listening for incoming connections." << std::endl;
}

void ConnectingState::send(TCPConnection* context, const std::string&
data) {
    std::cout << "Cannot send data in connecting state." << std::endl;
}
```

TCPConnection.hpp:

```
#ifndef TCPCONNECTION_HPP
#define TCPCONNECTION_HPP

#include "State.hpp"

class TCPConnection {
public:
    TCPConnection();

    void open();
    void close();
    void listen();
    void send(const std::string& data);
    void setState(ConnectionState& newState);

private:
    ConnectionState* currentState;
};

#endif
```

```cpp
//TcpConnection.cpp file
#include "TCPConnection.hpp"
#include "State.hpp"

TCPConnection::TCPConnection() :
currentState(&ConnectingState::instance()) {}

void TCPConnection::open() {
    currentState->open(this);
}

void TCP

Connection::close() {
    currentState->close(this);
}

void TCPConnection::listen() {
    currentState->listen(this);
}

void TCPConnection::send(const std::string& data) {
    currentState->send(this, data);
}

void TCPConnection::setState(ConnectionState& newState) {
    currentState = &newState;
}

int main() {
    TCPConnection connection;
    connection.open();
    connection.send("Hello, Server!");
    connection.listen();
    connection.send("Hello, Client!");
    connection.close();

    return 0;
}
```

Summary of the State Design Pattern:

In our TCP connection handler example, we've effectively managed the connection states and state transitions by encapsulating state-specific behaviors in separate classes. This promotes the Single Responsibility Principle and makes it easier to add new states or behaviors in the future without modifying existing code. The State pattern helps maintain a clean and maintainable code structure for managing objects with multiple states and state transitions.
