Паттерны ood

Самые частые паттерны выделены цветом

Порождающие паттерны

проектирования сосредоточены на процессе создания объектов. Они помогают управлять созданием объектов в зависимости от ситуации. Вот список основных порождающих паттернов:

1. **Singleton (Одиночка)**: Обеспечивает, чтобы класс имел только один экземпляр и предоставляет глобальную точку доступа к этому экземпляру.
2. **Factory Method (Фабричный метод)**: Определяет интерфейс для создания объекта, но позволяет подклассам изменять тип создаваемого объекта.
3. **Abstract Factory (Абстрактная фабрика)**: Предоставляет интерфейс для создания семейств связанных или зависимых объектов без указания их конкретных классов.
4. **Builder (Строитель)**: Разделяет процесс создания сложного объекта на его представление и конструирование, позволяя создавать разные представления одного и того же типа объекта.
5. **Prototype (Прототип)**: Позволяет создавать новые объекты путем копирования существующих, что может быть полезно, когда создание нового экземпляра объекта является более затратным, чем его копирование.

Эти паттерны помогают разработчикам создавать гибкие и поддерживаемые системы, позволяя им управлять созданием объектов более эффективно.

Структурные паттерны

проектирования определяют, как классы и объекты могут составлять более крупные структуры. Они помогают обеспечить удобное взаимодействие между объектами и упрощают организацию кода. Вот список основных структурных паттернов:

1. **Adapter (Адаптер)**: Позволяет объектам с несовместимыми интерфейсами работать вместе, преобразуя интерфейс одного класса в интерфейс, ожидаемый клиентом.
2. **Bridge (Мост)**: Разделяет абстракцию и реализацию, позволяя им изменяться независимо друг от друга.
3. **Composite (Компоновщик)**: Позволяет объединять объекты в древовидные структуры для представления иерархий "часть-целое". Клиенты могут работать с отдельными объектами и их композициями одинаково.

4. **Decorator (Декоратор)**: Позволяет динамически добавлять новые обязанности объектам, оборачивая их в классы-декораторы. Это предоставляет гибкий способ расширения функциональности.
5. **Facade (Фасад)**: Предоставляет упрощенный интерфейс к сложной системе классов, скрывая ее сложность и делая ее более доступной для использования.
6. **Flyweight (Приспособленец)**: Позволяет экономить память, разделяя общие состояния между множеством объектов, вместо того чтобы хранить их в каждом объекте.
7. **Proxy (Заместитель)**: Предоставляет суррогат или заместителя для другого объекта, чтобы контролировать доступ к нему. Это может быть полезно для управления доступом, ленивой инициализации или ведения журнала.

Эти структурные паттерны помогают организовать код, улучшить его читаемость и упростить взаимодействие между различными компонентами системы.

Поведенческие паттерны

проектирования сосредоточены на взаимодействии между объектами и определяют, как они общаются друг с другом. Вот список основных поведенческих паттернов:

1. **Chain of Responsibility (Цепочка обязанностей)**: Позволяет передавать запросы по цепочке обработчиков, где каждый обработчик решает, обрабатывать запрос или передать его дальше.
2. **Command (Команда)**: Инкапсулирует запрос как объект, позволяя параметризовать клиентов с различными запросами, ставить запросы в очередь и поддерживать отмену операций.
3. **Interpreter (Интерпретатор)**: Определяет грамматику для языка и предоставляет интерпретатор, который использует эту грамматику для интерпретации предложений в языке.
4. **Iterator (Итератор)**: Предоставляет способ последовательного доступа к элементам агрегированного объекта без раскрытия его внутренней структуры.
5. **Mediator (Посредник)**: Определяет объект, который инкапсулирует взаимодействие между множеством объектов, уменьшая связанность между ними.
6. **Memento (Хранитель)**: Позволяет сохранять и восстанавливать состояние объекта без раскрытия его внутренней структуры.
7. **Observer (Наблюдатель)**: Определяет зависимость "один ко многим" между объектами, так что при изменении состояния одного объекта все его зависимые объекты уведомляются и обновляются автоматически.
8. **State (Состояние)**: Позволяет объекту изменять свое поведение в зависимости от его внутреннего состояния, что позволяет избежать большого количества условных

операторов.

9. **Strategy (Стратегия)**: Определяет семейство алгоритмов, инкапсулирует их и делает их взаимозаменяемыми, позволяя изменять алгоритм независимо от клиентов, которые его используют.
10. **Template Method (Шаблонный метод)**: Определяет скелет алгоритма в методе, оставляя некоторые шаги подклассам. Это позволяет подклассам переопределять определенные шаги алгоритма без изменения его структуры.
11. **Visitor (Посетитель)**: Позволяет добавлять новые операции к объектам, не изменяя их классы, путем определения нового класса, который реализует операции.

Эти поведенческие паттерны помогают организовать взаимодействие между объектами, улучшая читаемость и поддержку кода.

Примеры на cpp

Порождающие паттерны

1. Singleton

Паттерн **Singleton (Одиночка)** — это порождающий паттерн проектирования, который гарантирует, что класс имеет только один экземпляр, и предоставляет глобальную точку доступа к этому экземпляру. Этот паттерн часто используется для управления доступом к ресурсам, которые должны быть уникальными, например, к конфигурационным данным, логам или соединениям с базой данных.

```
#include <iostream>
#include <memory>

class Singleton {
public:
    // Удаляем конструктор, копирующий конструктор и оператор присваивания
    Singleton(const Singleton&) = delete;
    Singleton& operator=(const Singleton&) = delete;

    // Метод для получения единственного экземпляра
    static std::unique_ptr<Singleton>& getInstance() {
        static std::unique_ptr<Singleton> instance(new Singleton());
        return instance;
    }

    void someBusinessLogic() {
        std::cout << "Выполнение бизнес-логики" << std::endl;
```

```
    }

private:
    // Приватный конструктор
    Singleton() {
        std::cout << "Создание экземпляра Singleton" << std::endl;
    }
};

int main() {
    // Получаем единственный экземпляр Singleton
    Singleton::getInstance()->someBusinessLogic();

    // Получаем еще один экземпляр Singleton
    Singleton::getInstance()->someBusinessLogic();

    return 0;
}
}
```

2.Factory method

Паттерн **Factory Method** (Фабричный метод) — это порождающий паттерн проектирования, который определяет интерфейс для создания объектов, но позволяет подклассам изменять тип создаваемого объекта. Этот паттерн позволяет делегировать создание объектов подклассам, что делает код более гибким и расширяемым.

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

// Интерфейс продукта
class Car {
public:
    virtual void drive() = 0; // Чисто виртуальный метод
    virtual ~Car() {}
};
// Конкретный продукт: Спортивный автомобиль
class SportsCar : public Car {
public:
    void drive() override {
        std::cout << "Driving a sports car!" << std::endl;
    }
};
```

```cpp
// Конкретный продукт: Седан
class Sedan : public Car {
public:
    void drive() override {
        std::cout << "Driving a sedan!" << std::endl;
    }
};
// Интерфейс создателя
class CarFactory {
public:
    virtual std::unique_ptr<Car> createCar() = 0; // Чисто виртуальный метод
    virtual ~CarFactory() {}
};
// Конкретная фабрика для спортивных автомобилей
class SportsCarFactory : public CarFactory {
public:
    std::unique_ptr<Car> createCar() override {
        return std::make_unique<SportsCar>();
    }
};

// Конкретная фабрика для седанов
class SedanFactory : public CarFactory {
public:
    std::unique_ptr<Car> createCar() override {
        return std::make_unique<Sedan>();
    }
};
int main() {
    std::unique_ptr<CarFactory> factory;

    // Создаем спортивный автомобиль
    factory = std::make_unique<SportsCarFactory>();
    std::unique_ptr<Car> car1 = factory->createCar();
    car1->drive();

    // Создаем седан
    factory = std::make_unique<SedanFactory>();
    std::unique_ptr<Car> car2 = factory->createCar();
    car2->drive();

    return 0;
}
}
```

3.Abstract factory

Паттерн **Abstract Factory** (Абстрактная фабрика) — это порождающий паттерн проектирования, который предоставляет интерфейс для создания семейств связанных или зависимых объектов без указания их конкретных классов. Этот паттерн позволяет создавать различные продукты, которые могут быть взаимозаменяемыми, и обеспечивает возможность легко добавлять новые продукты без изменения существующего кода.

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

// Интерфейс для стульев
class Chair {
public:
    virtual void sitOn() = 0; // Чисто виртуальный метод
    virtual ~Chair() {}
};

// Интерфейс для столов
class Table {
public:
    virtual void use() = 0; // Чисто виртуальный метод
    virtual ~Table() {}
};
// Конкретный продукт: Современный стул
class ModernChair : public Chair {
public:
    void sitOn() override {
        std::cout << "Sitting on a modern chair!" << std::endl;
    }
};

// Конкретный продукт: Классический стул
class VictorianChair : public Chair {
public:
    void sitOn() override {
        std::cout << "Sitting on a Victorian chair!" << std::endl;
    }
};

// Конкретный продукт: Современный стол
class ModernTable : public Table {
public:
    void use() override {
        std::cout << "Using a modern table!" << std::endl;
```

```
    }
};

// Конкретный продукт: Классический стол
class VictorianTable : public Table {
public:
    void use() override {
        std::cout << "Using a Victorian table!" << std::endl;
    }
};
// Интерфейс абстрактной фабрики
class FurnitureFactory {
public:
    virtual Chair* createChair() = 0; // Чисто виртуальный метод
    virtual Table* createTable() = 0; // Чисто виртуальный метод
    virtual ~FurnitureFactory() {}
};
// Конкретная фабрика: Современная мебель
class ModernFurnitureFactory : public FurnitureFactory {
public:
    Chair* createChair() override {
        return new ModernChair();
    }

    Table* createTable() override {
        return new ModernTable();
    }
};

// Конкретная фабрика: Классическая мебель
class VictorianFurnitureFactory : public FurnitureFactory {
public:
    Chair* createChair() override {
        return new VictorianChair();
    }

    Table* createTable() override {
        return new VictorianTable();
    }
};
int main() {
    FurnitureFactory* factory;

    // Создаем современную мебель
    factory = new ModernFurnitureFactory();
    Chair* modernChair = factory->createChair();
```

```
    Table* modernTable = factory->createTable();

    modernChair->sitOn();
    modernTable->use();

    delete modernChair;
    delete modernTable;
    delete factory;

    // Создаем классическую мебель
    factory = new VictorianFurnitureFactory();
    Chair* victorianChair = factory->createChair();
    Table* victorianTable = factory->createTable();

    victorianChair->sitOn();
    victorianTable->use();

    delete victorianChair;
    delete victorianTable;
    delete factory;

    return 0;
}
```

4.Builder

Паттерн **Builder** (Строитель) — это порождающий паттерн проектирования, который используется для создания сложных объектов пошагово. Он позволяет разделить процесс создания объекта от его представления, что дает возможность создавать различные представления одного и того же типа объекта. Паттерн Builder особенно полезен, когда объект имеет множество параметров или когда его создание требует сложной логики.

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

// Класс Pizza, который будет строиться
class Pizza {
public:
    void setDough(const std::string& dough) {
        this->dough = dough;
    }

    void setSauce(const std::string& sauce) {
```

```
        this->sauce = sauce;
    }

    void addTopping(const std::string& topping) {
        toppings.push_back(topping);
    }

    void show() const {
        std::cout << "Pizza with " << dough << " dough, " << sauce << "
sauce and toppings: ";
        for (const auto& topping : toppings) {
            std::cout << topping << " ";
        }
        std::cout << std::endl;
    }

private:
    std::string dough;
    std::string sauce;
    std::vector<std::string> toppings;
};

// Интерфейс строителя
class PizzaBuilder {
public:
    virtual void buildDough() = 0;
    virtual void buildSauce() = 0;
    virtual void addToppings() = 0;
    virtual std::unique_ptr<Pizza> getPizza() = 0;
    virtual ~PizzaBuilder() {}
};

// Конкретный строитель для пиццы "Маргарита"
class MargheritaPizzaBuilder : public PizzaBuilder {
public:
    MargheritaPizzaBuilder() {
        pizza = std::make_unique<Pizza>();
    }

    void buildDough() override {
        pizza->setDough("Thin crust");
    }

    void buildSauce() override {
        pizza->setSauce("Tomato");
    }
```

```
    void addToppings() override {
        pizza->addTopping("Mozzarella");
        pizza->addTopping("Basil");
    }

    std::unique_ptr<Pizza> getPizza() override {
        return std::move(pizza);
    }

private:
    std::unique_ptr<Pizza> pizza;
};

// Конкретный строитель для пиццы "Пепперони"
class PepperoniPizzaBuilder : public PizzaBuilder {
public:
    PepperoniPizzaBuilder() {
        pizza = std::make_unique<Pizza>();
    }

    void buildDough() override {
        pizza->setDough("Thick crust");
    }

    void buildSauce() override {
        pizza->setSauce("Spicy");
    }

    void addToppings() override {
        pizza->addTopping("Pepperoni");
        pizza->addTopping("Cheese");
    }

    std::unique_ptr<Pizza> getPizza() override {
        return std::move(pizza);
    }

private:
    std::unique_ptr<Pizza> pizza;
};

// Класс директор, который управляет процессом строительства
class PizzaDirector {
public:
    void setBuilder(std::unique_ptr<PizzaBuilder> builder) {
```

```
        this->builder = std::move(builder);
    }

    std::unique_ptr<Pizza> makePizza() {
        builder->buildDough();
        builder->buildSauce();
        builder->addToppings();
        return builder->getPizza();
    }

private:
    std::unique_ptr<PizzaBuilder> builder;
};

int main() {
    PizzaDirector director;

    // Создаем пиццу "Маргарита"
    director.setBuilder(std::make_unique<MargheritaPizzaBuilder>());
    std::unique_ptr<Pizza> margherita = director.makePizza();
    margherita->show();

    // Создаем пиццу "Пепперони"
    director.setBuilder(std::make_unique<PepperoniPizzaBuilder>());
    std::unique_ptr<Pizza> pepperoni = director.makePizza();
    pepperoni->show();

    return 0;
}
```

5.Prototype

Паттерн **Прототип** — это порождающий паттерн проектирования, который позволяет создавать новые объекты путем копирования существующих объектов, известного как "прототип". Этот паттерн особенно полезен, когда создание нового объекта является более затратным по времени или ресурсам, чем копирование уже существующего.

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

// Интерфейс прототипа
class Shape {
public:
    virtual std::unique_ptr<Shape> clone() const = 0; // Метод клонирования
```

```
    virtual void draw() const = 0; // Метод рисования
    virtual ~Shape() {}
};

// Конкретный прототип: Круг
class Circle : public Shape {
public:
    Circle() {
        std::cout << "Circle created" << std::endl;
    }

    std::unique_ptr<Shape> clone() const override {
        return std::make_unique<Circle>(*this); // Клонирование круга
    }

    void draw() const override {
        std::cout << "Drawing a Circle" << std::endl;
    }
};

// Конкретный прототип: Квадрат
class Square : public Shape {
public:
    Square() {
        std::cout << "Square created" << std::endl;
    }

    std::unique_ptr<Shape> clone() const override {
        return std::make_unique<Square>(*this); // Клонирование квадрата
    }

    void draw() const override {
        std::cout << "Drawing a Square" << std::endl;
    }
};

// Клиентский код
int main() {
    // Создаем прототипы
    std::unique_ptr<Shape> circlePrototype = std::make_unique<Circle>();
    std::unique_ptr<Shape> squarePrototype = std::make_unique<Square>();

    // Клонируем фигуры
    std::unique_ptr<Shape> circleClone = circlePrototype->clone();
    std::unique_ptr<Shape> squareClone = squarePrototype->clone();
```

```
    // // Рисуем клонированные фигуры
    circleClone->draw();
    squareClone->draw();

    return 0;
}
```

Структурные паттерны

Adapter

Паттерн **Адаптер** (Adapter) — это структурный паттерн проектирования, который позволяет объектам с несовместимыми интерфейсами работать вместе. Он действует как "мост" между двумя интерфейсами, позволяя одному объекту использовать функциональность другого объекта, не изменяя его код.

```
#include <iostream>
#include <memory>

// Целевой интерфейс (NewSystem)
class NewSystem {
public:
    virtual void request() = 0; // Метод, который будет вызываться клиентом
    virtual ~NewSystem() {}
};

// Адаптируемый класс (OldSystem)
class OldSystem {
public:
    void specificRequest() {
        std::cout << "OldSystem: Specific request." << std::endl;
    }
};

// Адаптер (Adapter)
class Adapter : public NewSystem {
public:
    Adapter(std::shared_ptr<OldSystem> oldSystem) : oldSystem_(oldSystem) {}

    void request() override {
        std::cout << "Adapter: Adapting request to OldSystem." << std::endl;
        oldSystem_->specificRequest(); // Перенаправляем вызов к
адаптируемому классу
    }
```

```
private:
    std::shared_ptr<OldSystem> oldSystem_; // Указатель на адаптируемый
класс
};

// Клиентский код
int main() {
    // Создаем адаптируемый объект
    std::shared_ptr<OldSystem> oldSystem = std::make_shared<OldSystem>();

    // Создаем адаптер
    std::shared_ptr<NewSystem> adapter = std::make_shared<Adapter>
(oldSystem);

    // Используем адаптер
    adapter->request();

    return 0;
}
```

Bridge

Паттерн **Bridge** (Мост) — это структурный паттерн проектирования, который разделяет абстракцию и реализацию, позволяя им изменяться независимо друг от друга. Этот паттерн полезен, когда вы хотите избежать жесткой привязки между абстракцией и ее реализацией, что позволяет легко расширять и модифицировать систему.

```
#include <iostream>
#include <memory>

// Интерфейс реализации (Device)
class Device {
public:
    virtual void turnOn() = 0; // Включить устройство
    virtual void turnOff() = 0; // Выключить устройство
    virtual ~Device() {}
};

// Конкретная реализация 1 (Television)
class Television : public Device {
public:
    void turnOn() override {
        std::cout << "Television is now ON." << std::endl;
    }
```

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

// Конкретная реализация 2 (Radio)
class Radio : public Device {
public:
    void turnOn() override {
        std::cout << "Radio is now ON." << std::endl;
    }

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

// Абстракция (RemoteControl)
class RemoteControl {
public:
    RemoteControl(std::unique_ptr<Device> device) :
device_(std::move(device)) {}

    void pressPowerButton() {
        if (isOn) {
            device_->turnOff();
        } else {
            device_->turnOn();
        }
        isOn = !isOn; // Переключаем состояние
    }

private:
    std::unique_ptr<Device> device_; // Указатель на устройство
    bool isOn = false; // Состояние устройства
};

// Клиентский код
int main() {
    // Создаем телевизор и пульт
    std::unique_ptr<Device> tv = std::make_unique<Television>();
    RemoteControl remote1(std::move(tv));
    remote1.pressPowerButton(); // Включаем телевизор
    remote1.pressPowerButton(); // Выключаем телевизор

    // Создаем радио и пульт
```

```
    std::unique_ptr<Device> radio = std::make_unique<Radio>();
    RemoteControl remote2(std::move(radio));
    remote2.pressPowerButton(); // Включаем радио
    remote2.pressPowerButton(); // Выключаем радио

    return 0;
}
```

Composite

Паттерн **Composite** (Компоновщик) — это структурный паттерн проектирования, который позволяет объединять объекты в древовидные структуры для представления иерархий "часть-целое". Этот паттерн позволяет клиентам работать с отдельными объектами и составными объектами одинаково, что упрощает работу с сложными структурами.

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

// Компонент
class FileSystemComponent {
public:
    virtual void show(int indent = 0) const = 0; // Метод для отображения
    virtual ~FileSystemComponent() {}
};

// Лист (File)
class File : public FileSystemComponent {
public:
    File(const std::string& name) : name_(name) {}

    void show(int indent = 0) const override {
        std::cout << std::string(indent, ' ') << "File: " << name_ <<
std::endl;
    }

private:
    std::string name_;
};

// Композит (Directory)
class Directory : public FileSystemComponent {
public:
```

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

    void add(std::shared_ptr<FileSystemComponent> component) {
        components_.push_back(component);
    }

    void show(int indent = 0) const override {
        std::cout << std::string(indent, ' ') << "Directory: " << name_ <<
std::endl;
        for (const auto& component : components_) {
            component->show(indent + 2); // Увеличиваем отступ для дочерних
элементов
        }
    }

private:
    std::string name_;
    std::vector<std::shared_ptr<FileSystemComponent>> components_; //
Дочерние компоненты
};

// Клиентский код
int main() {
    // Создаем файлы
    auto file1 = std::make_shared<File>("file1.txt");
    auto file2 = std::make_shared<File>("file2.txt");

    // Создаем директории
    auto dir1 = std::make_shared<Directory>("dir1");
    auto dir2 = std::make_shared<Directory>("dir2");

    // Добавляем файлы в директории
    dir1->add(file1);
    dir2->add(file2);

    // Создаем корневую директорию
    auto root = std::make_shared<Directory>("root");
    root->add(dir1);
    root->add(dir2);

    // Отображаем файловую систему
    root->show();

    return 0;
}
```

Decorator

Паттерн Декоратор** (Decorator) — это структурный паттерн проектирования, который позволяет динамически добавлять новые функциональные возможности объектам, оборачивая их в другие объекты. Этот паттерн предоставляет гибкий способ расширения функциональности объектов без изменения их кода.

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

// Компонент
class Coffee {
public:
    virtual std::string getDescription() const = 0; // Метод для получения
описания
    virtual double cost() const = 0; // Метод для получения стоимости
    virtual ~Coffee() {}
};

// Конкретный компонент (Простое кофе)
class SimpleCoffee : public Coffee {
public:
    std::string getDescription() const override {
        return "Simple coffee";
    }

    double cost() const override {
        return 2.0; // Стоимость простого кофе
    }
};

// Декоратор
class CoffeeDecorator : public Coffee {
public:
    CoffeeDecorator(std::shared_ptr<Coffee> coffee) : coffee_(coffee) {}

    std::string getDescription() const override {
        return coffee_->getDescription(); // Передаем вызов декорируемому
объекту
    }

    double cost() const override {
        return coffee_->cost(); // Передаем вызов декорируемому объекту
    }
```

```
protected:
    std::shared_ptr<Coffee> coffee_; // Указатель на декорируемый объект
};

// Конкретный декоратор (Молоко)
class MilkDecorator : public CoffeeDecorator {
public:
    MilkDecorator(std::shared_ptr<Coffee> coffee) : CoffeeDecorator(coffee)
{}

    std::string getDescription() const override {
        return coffee_->getDescription() + ", with milk"; // Добавляем
описание
    }

    double cost() const override {
        return coffee_->cost() + 0.5; // Добавляем стоимость молока
    }
};

// Конкретный декоратор (Сахар)
class SugarDecorator : public CoffeeDecorator {
public:
    SugarDecorator(std::shared_ptr<Coffee> coffee) : CoffeeDecorator(coffee)
{}

    std::string getDescription() const override {
        return coffee_->getDescription() + ", with sugar"; // Добавляем
описание
    }

    double cost() const override {
        return coffee_->cost() + 0.2; // Добавляем стоимость сахара
    }
};

// Клиентский код
int main() {
    // Создаем простое кофе
    std::shared_ptr<Coffee> coffee = std::make_shared<SimpleCoffee>();
    std::cout << coffee->getDescription() << " costs $" << coffee->cost() <<
std::endl;

    // Добавляем молоко
    coffee = std::make_shared<MilkDecorator>(coffee);
```

```
    std::cout << coffee->getDescription() << " costs $" << coffee->cost() <<
std::endl;

    // Добавляем сахар
    coffee = std::make_shared<SugarDecorator>(coffee);
    std::cout << coffee->getDescription() << " costs $" << coffee->cost() <<
std::endl;

    return 0;
}
```

Facade

Паттерн **Facade** (Фасад) — это структурный паттерн проектирования, который предоставляет упрощенный интерфейс к сложной системе классов, библиотеке или фреймворку. Он скрывает сложность системы и предоставляет клиенту более простой способ взаимодействия с ней.

```
#include <iostream>

// Подсистема 1: Телевизор
class Television {
public:
    void turnOn() {
        std::cout << "Television is now ON." << std::endl;
    }

    void turnOff() {
        std::cout << "Television is now OFF." << std::endl;
    }
};

// Подсистема 2: Проигрыватель DVD
class DVDPlayer {
public:
    void turnOn() {
        std::cout << "DVD Player is now ON." << std::endl;
    }

    void turnOff() {
        std::cout << "DVD Player is now OFF." << std::endl;
    }

    void play() {
        std::cout << "Playing DVD." << std::endl;
```

```
    }
};

// Подсистема 3: Звуковая система
class SoundSystem {
public:
    void turnOn() {
        std::cout << "Sound System is now ON." << std::endl;
    }

    void turnOff() {
        std::cout << "Sound System is now OFF." << std::endl;
    }

    void setVolume(int level) {
        std::cout << "Setting volume to " << level << "." << std::endl;
    }
};

// Фасад
class HomeTheaterFacade {
public:
    HomeTheaterFacade()
        : tv_(new Television()), dvdPlayer_(new DVDPlayer()),
soundSystem_(new SoundSystem()) {}

    void watchMovie() {
        std::cout << "Getting ready to watch a movie..." << std::endl;
        tv_->turnOn();
        soundSystem_->turnOn();
        soundSystem_->setVolume(10);
        dvdPlayer_->turnOn();
        dvdPlayer_->play();
    }

    void endMovie() {
        std::cout << "Shutting down the home theater..." << std::endl;
        dvdPlayer_->turnOff();
        soundSystem_->turnOff();
        tv_->turnOff();
    }

private:
    Television* tv_;
    DVDPlayer* dvdPlayer_;
    SoundSystem* soundSystem_;
```

```
};

// Клиентский код
int main() {
    HomeTheaterFacade homeTheater;
    homeTheater.watchMovie(); // Подготовка к просмотру фильма
    homeTheater.endMovie();   // Завершение просмотра фильма

    return 0;
}
```

Flyweight

Паттерн **Flyweight** (Летучий вес) — это структурный паттерн проектирования, который позволяет уменьшить количество создаваемых объектов, используя совместное использование объектов, которые имеют общие состояния. Этот паттерн особенно полезен, когда необходимо создать большое количество объектов, которые имеют схожие характеристики, но различаются по некоторым параметрам.

```
#include <iostream>
#include <unordered_map>
#include <memory>

// Интерфейс Flyweight
class Character {
public:
    virtual void display(int x, int y) = 0; // Метод для отображения символа
    virtual ~Character() {}
};

// Конкретный Flyweight (символ)
class ConcreteCharacter : public Character {
public:
    ConcreteCharacter(char c, const std::string& font, const std::string&
color)
        : character_(c), font_(font), color_(color) {}

    void display(int x, int y) override {
        std::cout << "Character: " << character_
                  << " | Font: " << font_
                  << " | Color: " << color_
                  << " | Position: (" << x << ", " << y << ")" << std::endl;
    }

private:
```

```
    char character_; // Символ
    std::string font_; // Шрифт
    std::string color_; // Цвет
};

// Фабрика Flyweight
class CharacterFactory {
public:
    std::shared_ptr<Character> getCharacter(char c, const std::string& font,
const std::string& color) {
        std::string key = std::string(1, c) + font + color; // Уникальный
ключ для символа
        if (characters_.find(key) == characters_.end()) {
            characters_[key] = std::make_shared<ConcreteCharacter>(c, font,
color);
        }
        return characters_[key]; // Возвращаем существующий или новый символ
    }

private:
    std::unordered_map<std::string, std::shared_ptr<Character>> characters_;
// Хранит летучие веса
};

// Клиентский код
int main() {
    CharacterFactory factory;

    // Создаем символы
    auto a1 = factory.getCharacter('A', "Arial", "Red");
    auto a2 = factory.getCharacter('A', "Arial", "Red");
    auto b1 = factory.getCharacter('B', "Arial", "Blue");

    // Отображаем символы
    a1->display(10, 20);
    a2->display(30, 40); // a1 и a2 - это один и тот же объект
    b1->display(50, 60);

    std::cout << "a1 and a2 are the same object: " << (a1 == a2) <<
std::endl; // Проверка на совместное использование

    return 0;
}
```

Proxy

Паттерн **Proxy** (Заместитель) — это структурный паттерн проектирования, который предоставляет суррогат или заместитель другого объекта для контроля доступа к нему. Этот паттерн может использоваться для различных целей, таких как управление доступом, ленивое создание объектов, кэширование и логирование.

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

// Интерфейс субъекта
class Image {
public:
    virtual void display() = 0; // Метод для отображения изображения
    virtual ~Image() {}
};

// Реальный субъект (реальное изображение)
class RealImage : public Image {
public:
    RealImage(const std::string& filename) : filename_(filename) {
        loadImageFromDisk(); // Загружаем изображение из диска
    }

    void display() override {
        std::cout << "Displaying " << filename_ << std::endl;
    }

private:
    std::string filename_;

    void loadImageFromDisk() {
        std::cout << "Loading " << filename_ << std::endl;
    }
};

// Прокси
class ProxyImage : public Image {
public:
    ProxyImage(const std::string& filename) : filename_(filename),
realImage_(nullptr) {}

    void display() override {
        if (!realImage_) {
            realImage_ = std::make_shared<RealImage>(filename_); //
Загружаем реальное изображение при первом вызове
```

```
        }
        realImage_->display(); // Вызываем метод отображения реального
изображения
    }

private:
    std::string filename_;
    std::shared_ptr<RealImage> realImage_; // Указатель на реальное
изображение
};

// Клиентский код
int main() {
    std::shared_ptr<Image> image1 = std::make_shared<ProxyImage>
("image1.jpg");
    std::shared_ptr<Image> image2 = std::make_shared<ProxyImage>
("image2.jpg");

    // Изображение загружается только при первом вызове display()
    image1->display(); // Загружается и отображается
    image1->display(); // Отображается без загрузки

    image2->display(); // Загружается и отображается

    return 0;
}
```

Поведенческие паттерны

Chain of Responsibility (Цепочка обязанностей)

Паттерн **Chain of Responsibility** (Цепочка обязанностей) — это поведенческий паттерн проектирования, который позволяет передавать запросы по цепочке обработчиков. Каждый обработчик может обработать запрос или передать его следующему обработчику в цепочке. Этот паттерн позволяет избежать жесткой привязки между отправителем запроса и его получателем, а также упрощает добавление новых обработчиков.

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

// Интерфейс обработчика
class SupportHandler {
public:
    virtual void setNext(std::shared_ptr<SupportHandler> next) = 0; //
```

```
Устанавливает следующий обработчик
    virtual void handleRequest(const std::string& request) = 0; //
Обрабатывает запрос
    virtual ~SupportHandler() {}
};

// Конкретный обработчик 1 (Уровень 1 поддержки)
class SupportLevel1 : public SupportHandler {
public:
    void setNext(std::shared_ptr<SupportHandler> next) override {
        next_ = next;
    }

    void handleRequest(const std::string& request) override {
        if (request == "Level 1") {
            std::cout << "Support Level 1: Handling request: " << request <<
std::endl;
        } else if (next_) {
            next_->handleRequest(request); // Передаем запрос следующему
обработчику
        }
    }

private:
    std::shared_ptr<SupportHandler> next_;
};

// Конкретный обработчик 2 (Уровень 2 поддержки)
class SupportLevel2 : public SupportHandler {
public:
    void setNext(std::shared_ptr<SupportHandler> next) override {
        next_ = next;
    }

    void handleRequest(const std::string& request) override {
        if (request == "Level 2") {
            std::cout << "Support Level 2: Handling request: " << request <<
std::endl;
        } else if (next_) {
            next_->handleRequest(request); // Передаем запрос следующему
обработчику
        }
    }

private:
    std::shared_ptr<SupportHandler> next_;
```

```
};

// Конкретный обработчик 3 (Уровень 3 поддержки)
class SupportLevel3 : public SupportHandler {
public:
    void setNext(std::shared_ptr<SupportHandler> next) override {
        next_ = next;
    }

    void handleRequest(const std::string& request) override {
        if (request == "Level 3") {
            std::cout << "Support Level 3: Handling request: " << request <<
std::endl;
        } else {
            std::cout << "Support Level 3: No handler for request: " <<
request << std::endl;
        }
    }

private:
    std::shared_ptr<SupportHandler> next_;
};

// Клиентский код
int main() {
    // Создаем обработчики
    auto level1 = std::make_shared<SupportLevel1>();
    auto level2 = std::make_shared<SupportLevel2>();
    auto level3 = std::make_shared<SupportLevel3>();

    // Устанавливаем цепочку обработчиков
    level1->setNext(level2);
    level2->setNext(level3);

    // Отправляем запросы
    level1->handleRequest("Level 1"); // Обрабатывается на уровне 1
    level1->handleRequest("Level 2"); // Обрабатывается на уровне 2
}
```

Command

Паттерн **Command** (Команда) — это поведенческий паттерн проектирования, который инкапсулирует запрос как объект, позволяя параметризовать клиентов с различными запросами, ставить запросы в очередь и поддерживать отмену операций. Этот паттерн

позволяет отделить отправителя запроса от его получателя, что упрощает добавление новых команд и управление ими.

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

// Интерфейс команды
class Command {
public:
    virtual void execute() = 0; // Метод для выполнения команды
    virtual ~Command() {}
};

// Получатель (свет)
class Light {
public:
    void turnOn() {
        std::cout << "Light is ON." << std::endl;
    }

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

// Конкретная команда (включить свет)
class TurnOnLightCommand : public Command {
public:
    TurnOnLightCommand(std::shared_ptr<Light> light) : light_(light) {}

    void execute() override {
        light_->turnOn(); // Выполняем действие
    }

private:
    std::shared_ptr<Light> light_; // Указатель на получателя
};

// Конкретная команда (выключить свет)
class TurnOffLightCommand : public Command {
public:
    TurnOffLightCommand(std::shared_ptr<Light> light) : light_(light) {}

    void execute() override {
        light_->turnOff(); // Выполняем действие
```

```
    }

private:
    std::shared_ptr<Light> light_; // Указатель на получателя
};

// Инициатор (пульт)
class RemoteControl {
public:
    void setCommand(std::shared_ptr<Command> command) {
        command_ = command; // Устанавливаем команду
    }

    void pressButton() {
        if (command_) {
            command_->execute(); // Вызываем команду
        }
    }

private:
    std::shared_ptr<Command> command_; // Хранит текущую команду
};

// Клиентский код
int main() {
    // Создаем получателя
    auto light = std::make_shared<Light>();

    // Создаем команды
    auto turnOn = std::make_shared<TurnOnLightCommand>(light);
    auto turnOff = std::make_shared<TurnOffLightCommand>(light);

    // Создаем пульт
    RemoteControl remote;

    // Включаем свет
    remote.setCommand(turnOn);
    remote.pressButton();

    // Выключаем свет
    remote.setCommand(turnOff);
    remote.pressButton();

    return 0;
}
```

Interpreter

Паттерн **Interpreter** (Интерпретатор) — это поведенческий паттерн проектирования, который определяет грамматику для языка и предоставляет интерпретатор для обработки предложений на этом языке. Этот паттерн позволяет создавать системы, которые могут интерпретировать и выполнять команды, написанные на определенном языке.

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

// Абстрактное выражение
class Expression {
public:
    virtual int interpret() = 0; // Метод для интерпретации выражения
    virtual ~Expression() {}
};

// Конкретное выражение (число)
class Number : public Expression {
public:
    Number(int value) : value_(value) {}

    int interpret() override {
        return value_; // Возвращаем значение числа
    }

private:
    int value_;
};

// Конкретное выражение (операция сложения)
class Addition : public Expression {
public:
    Addition(std::shared_ptr<Expression> left, std::shared_ptr<Expression>
right)
        : left_(left), right_(right) {}

    int interpret() override {
        return left_->interpret() + right_->interpret(); // Сложение
    }

private:
```

```
    std::shared_ptr<Expression> left_; // Левый операнд
    std::shared_ptr<Expression> right_; // Правый операнд
};

// Конкретное выражение (операция вычитания)
class Subtraction : public Expression {
public:
    Subtraction(std::shared_ptr<Expression> left,
std::shared_ptr<Expression> right)
        : left_(left), right_(right) {}

    int interpret() override {
        return left_->interpret() - right_->interpret(); // Вычитание
    }

private:
    std::shared_ptr<Expression> left_; // Левый операнд
    std::shared_ptr<Expression> right_; // Правый операнд
};

// Фабрика для создания выражений
class ExpressionParser {
public:
    static std::shared_ptr<Expression> parse(const std::string& input) {
        std::istringstream stream(input);
        std::string token;
        std::vector<std::shared_ptr<Expression>> expressions;

        while (stream >> token) {
            if (isdigit(token[0])) {
                expressions.push_back(std::make_shared<Number>
(std::stoi(token))); // Число
            } else if (token == "+") {
                auto right = expressions.back(); expressions.pop_back();
                auto left = expressions.back(); expressions.pop_back();
                expressions.push_back(std::make_shared<Addition>(left,
right)); // Сложение
            } else if (token == "-") {
                auto right = expressions.back(); expressions.pop_back();
                auto left = expressions.back(); expressions.pop_back();
                expressions.push_back(std::make_shared<Subtraction>(left,
right)); // Вычитание
            }
        }

        return expressions.back(); // Возвращаем последнее выражение
```

```
    }
};

// Клиентский код
int main() {
    std::string expression = "5 3 - 2 +"; // (5 - 3) + 2
    auto parsedExpression = ExpressionParser::parse(expression);
    std::cout << "Result: " << parsedExpression->interpret() << std::endl;
// Выводим результат

    return 0;
}
```

Iterator

Паттерн **Iterator** (Итератор) — это поведенческий паттерн проектирования, который предоставляет способ последовательного доступа к элементам агрегированного объекта (например, коллекции) без раскрытия его внутренней структуры. Этот паттерн позволяет обходить элементы коллекции, не привязываясь к конкретной реализации коллекции.

```
#include <iostream>
#include <vector>
#include <memory>

// Интерфейс итератора
class Iterator {
public:
    virtual bool hasNext() = 0; // Проверяет, есть ли следующий элемент
    virtual std::string next() = 0; // Возвращает следующий элемент
    virtual ~Iterator() {}
};

// Интерфейс агрегата
class Aggregate {
public:
    virtual std::shared_ptr<Iterator> createIterator() = 0; // Создает
итератор
    virtual ~Aggregate() {}
};

// Конкретный итератор
class BookIterator : public Iterator {
public:
    BookIterator(const std::vector<std::string>& books) : books_(books),
index_(0) {}
```

```
    bool hasNext() override {
        return index_ < books_.size(); // Проверяем, есть ли следующий
элемент
    }

    std::string next() override {
        return books_[index_++]; // Возвращаем следующий элемент и
увеличиваем индекс
    }

private:
    const std::vector<std::string>& books_; // Ссылка на коллекцию книг
    size_t index_; // Текущий индекс
};

// Конкретный агрегат
class BookCollection : public Aggregate {
public:
    void addBook(const std::string& book) {
        books_.push_back(book); // Добавляем книгу в коллекцию
    }

    std::shared_ptr<Iterator> createIterator() override {
        return std::make_shared<BookIterator>(books_); // Создаем итератор
    }

private:
    std::vector<std::string> books_; // Коллекция книг
};

// Клиентский код
int main() {
    BookCollection collection;
    collection.addBook("The Catcher in the Rye");
    collection.addBook("To Kill a Mockingbird");
    collection.addBook("1984");

    // Создаем итератор для обхода коллекции
    auto iterator = collection.createIterator();

    // Обходим коллекцию и выводим книги
    while (iterator->hasNext()) {
        std::cout << iterator->next() << std::endl;
    }
```

```
    return 0;
}
```

Mediator

Паттерн **Mediator** (Посредник) — это поведенческий паттерн проектирования, который определяет объект, инкапсулирующий способ взаимодействия множества объектов. Паттерн позволяет уменьшить связанность между объектами, позволяя им взаимодействовать через посредника вместо прямого общения друг с другом.

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

// Впереди объявляем классы
class User;

// Интерфейс посредника
class ChatMediator {
public:
    virtual void sendMessage(const std::string& message, User* user) = 0; // Метод для отправки сообщения
    virtual void addUser (User* user) = 0; // Метод для добавления пользователя
    virtual ~ChatMediator() {}
};

// Коллега (Пользователь)
class User {
public:
    User(ChatMediator* mediator, const std::string& name) : mediator_(mediator), name_(name) {
        mediator_->addUser (this); // Добавляем пользователя в медиатор
    }

    void sendMessage(const std::string& message) {
        mediator_->sendMessage(message, this); // Отправляем сообщение через медиатор
    }

    void receiveMessage(const std::string& message) {
        std::cout << name_ << " received: " << message << std::endl; // Получаем сообщение
    }
```

```
private:
    ChatMediator* mediator_; // Ссылка на медиатор
    std::string name_; // Имя пользователя
};

// Конкретный посредник
class ChatRoom : public ChatMediator {
public:
    void sendMessage(const std::string& message, User* user) override {
        for (User * u : users_) {
            // Не отправляем сообщение отправителю
            if (u != user) {
                u->receiveMessage(message);
            }
        }
    }

    void addUser (User* user) override {
        users_.push_back(user); // Добавляем пользователя в список
    }

private:
    std::vector<User*> users_; // Список пользователей
};

// Клиентский код
int main() {
    ChatRoom chatRoom;

    User user1(&chatRoom, "Alice");
    User user2(&chatRoom, "Bob");
    User user3(&chatRoom, "Charlie");

    user1.sendMessage("Hello, everyone!");
    user2.sendMessage("Hi, Alice!");
    user3.sendMessage("Good morning!");

    return 0;
}
```

Memento

Паттерн **Memento** (Хранитель) — это поведенческий паттерн проектирования, который позволяет сохранять и восстанавливать состояние объекта без нарушения инкапсуляции.

Этот паттерн позволяет создавать снимки состояния объекта, которые могут быть использованы для восстановления объекта в определенный момент времени.

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

// Хранитель (Memento)
class Memento {
public:
    Memento(const std::string& state) : state_(state) {}

    std::string getState() const {
        return state_; // Возвращаем сохраненное состояние
    }

private:
    std::string state_; // Состояние объекта
};

// Производитель (Originator)
class TextEditor {
public:
    void setText(const std::string& text) {
        text_ = text; // Устанавливаем текст
    }

    std::string getText() const {
        return text_; // Возвращаем текущий текст
    }

    std::shared_ptr<Memento> save() {
        return std::make_shared<Memento>(text_); // Сохраняем текущее
состояние
    }

    void restore(const std::shared_ptr<Memento>& memento) {
        text_ = memento->getState(); // Восстанавливаем состояние из
хранителя
    }

private:
    std::string text_; // Текущий текст
};
```

```
// Управляющий (Caretaker)
class History {
public:
    void save(const std::shared_ptr<Memento>& memento) {
        history_.push_back(memento); // Сохраняем хранитель в истории
    }

    std::shared_ptr<Memento> undo() {
        if (history_.empty()) {
            return nullptr; // Если история пуста, возвращаем nullptr
        }
        auto memento = history_.back(); // Получаем последний сохраненный
хранитель
        history_.pop_back(); // Удаляем его из истории
        return memento; // Возвращаем хранитель
    }

private:
    std::vector<std::shared_ptr<Memento>> history_; // История сохраненных
состояний
};

// Клиентский код
int main() {
    TextEditor editor;
    History history;

    // Устанавливаем текст и сохраняем состояние
    editor.setText("Version 1");
    history.save(editor.save());

    editor.setText("Version 2");
    history.save(editor.save());

    editor.setText("Version 3");
    std::cout << "Current text: " << editor.getText() << std::endl;

    // Восстанавливаем предыдущее состояние
    editor.restore(history.undo());
    std::cout << "After undo: " << editor.getText() << std::endl;

    // Восстанавливаем еще одно предыдущее состояние
    editor.restore(history.undo());
    std::cout << "After another undo: " << editor.getText() << std::endl;
```

```
    return 0;
}
```

Observer

Паттерн **Observer** (Наблюдатель) — это поведенческий паттерн проектирования, который определяет зависимость "один ко многим" между объектами, так что при изменении состояния одного объекта все его зависимые объекты уведомляются и обновляются автоматически. Этот паттерн часто используется в системах событий и уведомлений.

```
#include <iostream>
#include <vector>
#include <memory>

// Интерфейс наблюдателя
class Observer {
public:
    virtual void update(float temperature) = 0; // Метод для обновления
состояния
    virtual ~Observer() {}
};

// Интерфейс субъекта
class Subject {
public:
    virtual void addObserver(std::shared_ptr<Observer> observer) = 0; //
Добавление наблюдателя
    virtual void removeObserver(std::shared_ptr<Observer> observer) = 0; //
Удаление наблюдателя
    virtual void notifyObservers() = 0; // Уведомление наблюдателей
    virtual ~Subject() {}
};

// Конкретный субъект (Метеостанция)
class WeatherStation : public Subject {
public:
    void addObserver(std::shared_ptr<Observer> observer) override {
        observers_.push_back(observer); // Добавляем наблюдателя
    }

    void removeObserver(std::shared_ptr<Observer> observer) override {
        observers_.erase(std::remove(observers_.begin(), observers_.end(),
observer), observers_.end()); // Удаляем наблюдателя
    }
```

```
    void notifyObservers() override {
        for (const auto& observer : observers_) {
            observer->update(temperature_); // Уведомляем всех наблюдателей
        }
    }

    void setTemperature(float temperature) {
        temperature_ = temperature; // Устанавливаем новую температуру
        notifyObservers(); // Уведомляем наблюдателей об изменении
    }

private:
    std::vector<std::shared_ptr<Observer>> observers_; // Список
наблюдателей
    float temperature_; // Текущая температура
};

// Конкретный наблюдатель (Дисплей)
class Display : public Observer {
public:
    void update(float temperature) override {
        std::cout << "Current temperature: " << temperature << "°C" <<
std::endl; // Обновляем состояние
    }
};

// Клиентский код
int main() {
    auto weatherStation = std::make_shared<WeatherStation>();
    auto display = std::make_shared<Display>();

    weatherStation->addObserver(display); // Добавляем дисплей как
наблюдателя

    // Изменяем температуру
    weatherStation->setTemperature(25.0f);
    weatherStation->setTemperature(30.0f);

    return 0;
}
```

State

Паттерн **State** (Состояние) — это поведенческий паттерн проектирования, который позволяет объекту изменять свое поведение в зависимости от его внутреннего состояния.

Этот паттерн позволяет избежать сложных условных операторов и делает код более чистым и понятным, так как каждое состояние представляется отдельным классом.

```
#include <iostream>
#include <memory>

// Интерфейс состояния
class State {
public:
    virtual void play() = 0; // Метод для воспроизведения
    virtual void pause() = 0; // Метод для паузы
    virtual void stop() = 0; // Метод для остановки
    virtual ~State() {}
};

// Контекст (Музыкальный плеер)
class MusicPlayer {
public:
    MusicPlayer() : state_(nullptr) {}

    void setState(std::shared_ptr<State> state) {
        state_ = state; // Устанавливаем текущее состояние
    }

    void play() {
        if (state_) {
            state_->play(); // Делегируем вызов текущему состоянию
        }
    }

    void pause() {
        if (state_) {
            state_->pause(); // Делегируем вызов текущему состоянию
        }
    }

    void stop() {
        if (state_) {
            state_->stop(); // Делегируем вызов текущему состоянию
        }
    }

private:
    std::shared_ptr<State> state_; // Текущее состояние
};
```

```
// Конкретное состояние (Воспроизведение)
class PlayingState : public State {
public:
    void play() override {
        std::cout << "Already playing." << std::endl; // Если уже
воспроизводится
    }

    void pause() override {
        std::cout << "Pausing the music." << std::endl; // Пауза
    }

    void stop() override {
        std::cout << "Stopping the music." << std::endl; // Остановка
    }
};

// Конкретное состояние (Пауза)
class PausedState : public State {
public:
    void play() override {
        std::cout << "Resuming the music." << std::endl; // Возобновление
    }

    void pause() override {
        std::cout << "Already paused." << std::endl; // Если уже на паузе
    }

    void stop() override {
        std::cout << "Stopping the music." << std::endl; // Остановка
    }
};

// Конкретное состояние (Остановлено)
class StoppedState : public State {
public:
    void play() override {
        std::cout << "Starting the music." << std::endl; // Начало
воспроизведения
    }

    void pause() override {
        std::cout << "Can't pause. Music is stopped." << std::endl; //
Нельзя поставить на паузу
    }
```

```
    void stop() override {
        std::cout << "Already stopped." << std::endl; // Если уже
остановлено
    }
};

// Клиентский код
int main() {
    MusicPlayer player;

    // Устанавливаем состояние "Остановлено"
    player.setState(std::make_shared<StoppedState>());
    player.play(); // Начинаем воспроизведение
    player.pause(); // Нельзя поставить на паузу
    player.stop(); // Уже остановлено

    // Устанавливаем состояние "Воспроизведение"
    player.setState(std::make_shared<PlayingState>());
    player.play(); // Уже воспроизводится
    player.pause(); // Ставим на паузу
    player.stop(); // Останавливаем
}
```

Strategy

Паттерн **Strategy** (Стратегия) — это поведенческий паттерн проектирования, который определяет семейство алгоритмов, инкапсулирует каждый из них и делает их взаимозаменяемыми. Этот паттерн позволяет изменять алгоритмы независимо от клиентов, которые их используют.

```
#include <iostream>
#include <vector>
#include <memory>

// Интерфейс стратегии
class SortStrategy {
public:
    virtual void sort(std::vector<int>& data) = 0; // Метод для сортировки
    virtual ~SortStrategy() {}
};

// Конкретная стратегия (Сортировка пузырьком)
class BubbleSort : public SortStrategy {
public:
    void sort(std::vector<int>& data) override {
```

```
        for (size_t i = 0; i < data.size() - 1; ++i) {
            for (size_t j = 0; j < data.size() - i - 1; ++j) {
                if (data[j] > data[j + 1]) {
                    std::swap(data[j], data[j + 1]); // Меняем местами
                }
            }
        }
        std::cout << "Sorted using Bubble Sort." << std::endl;
    }
};

// Конкретная стратегия (Сортировка выбором)
class SelectionSort : public SortStrategy {
public:
    void sort(std::vector<int>& data) override {
        for (size_t i = 0; i < data.size() - 1; ++i) {
            size_t minIndex = i;
            for (size_t j = i + 1; j < data.size(); ++j) {
                if (data[j] < data[minIndex]) {
                    minIndex = j; // Находим индекс минимального элемента
                }
            }
            std::swap(data[i], data[minIndex]); // Меняем местами
        }
        std::cout << "Sorted using Selection Sort." << std::endl;
    }
};

// Контекст
class Sorter {
public:
    Sorter(std::shared_ptr<SortStrategy> strategy) : strategy_(strategy) {}

    void setStrategy(std::shared_ptr<SortStrategy> strategy) {
        strategy_ = strategy; // Устанавливаем новую стратегию
    }

    void sort(std::vector<int>& data) {
        strategy_->sort(data); // Делегируем вызов стратегии
    }

private:
    std::shared_ptr<SortStrategy> strategy_; // Текущая стратегия
};

// Клиентский код
```

```
int main() {
    std::vector<int> data = {5, 3, 8, 6, 2};

    // Используем сортировку пузырьком
    Sorter sorter(std::make_shared<BubbleSort>());
    sorter.sort(data);

    // Выводим отсортированный массив
    for (int num : data) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    // Меняем стратегию на сортировку выбором
    data = {5, 3, 8, 6, 2}; // Сбрасываем данные
    sorter.setStrategy(std::make_shared<SelectionSort>());
    sorter.sort(data);

    // Выводим отсортированный массив
    for (int num : data) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Template Method (Шаблонный метод)

Паттерн **Template Method** (Шаблонный метод) — это поведенческий паттерн проектирования, который определяет скелет алгоритма в методе, оставляя некоторые шаги для реализации в подклассах. Этот паттерн позволяет переопределять определенные шаги алгоритма без изменения его структуры.

```
#include <iostream>

// Абстрактный класс (Шаблонный метод)
class CaffeineBeverage {
public:
    // Шаблонный метод
    void prepareRecipe() {
        boilWater();
        brew();
        pourInCup();
        addCondiments();
```

```
    }

protected:
    virtual void brew() = 0; // Абстрактный метод для заваривания
    virtual void addCondiments() = 0; // Абстрактный метод для добавления
приправ

private:
    void boilWater() {
        std::cout << "Boiling water." << std::endl; // Шаг 1: Кипятим воду
    }

    void pourInCup() {
        std::cout << "Pouring into cup." << std::endl; // Шаг 3: Наливаем в
чашку
    }
};

// Конкретный класс (Чай)
class Tea : public CaffeineBeverage {
protected:
    void brew() override {
        std::cout << "Steeping the tea." << std::endl; // Шаг 2: Завариваем
чай
    }

    void addCondiments() override {
        std::cout << "Adding lemon." << std::endl; // Добавляем лимон
    }
};

// Конкретный класс (Кофе)
class Coffee : public CaffeineBeverage {
protected:
    void brew() override {
        std::cout << "Dripping coffee through filter." << std::endl; // Шаг
2: Завариваем кофе
    }

    void addCondiments() override {
        std::cout << "Adding sugar and milk." << std::endl; // Добавляем
сахар и молоко
    }
};

// Клиентский код
```

```cpp
int main() {
    CaffeineBeverage* tea = new Tea();
    tea->prepareRecipe(); // Приготовление чая
    std::cout << std::endl;

    CaffeineBeverage* coffee = new Coffee();
    coffee->prepareRecipe(); // Приготовление кофе

    // Освобождаем память
    delete tea;
    delete coffee;

    return 0;
}
```

Visitor

Паттерн **Visitor** (Посетитель) — это поведенческий паттерн проектирования, который позволяет добавлять новые операции к объектам, не изменяя их классы. Этот паттерн позволяет отделить алгоритмы от объектов, над которыми они работают, что упрощает добавление новых операций и улучшает поддержку кода.

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

// Впереди объявляем классы
class WordDocument;
class PDFDocument;

// Интерфейс посетителя
class DocumentVisitor {
public:
    virtual void visit(WordDocument* wordDoc) = 0; // Метод для обработки
Word-документа
    virtual void visit(PDFDocument* pdfDoc) = 0;   // Метод для обработки
PDF-документа
    virtual ~DocumentVisitor() {}
};

// Интерфейс элемента
class Document {
public:
    virtual void accept(DocumentVisitor* visitor) = 0; // Метод для приема
посетителя
```

```
    virtual ~Document() {}
};

// Конкретный элемент (Word-документ)
class WordDocument : public Document {
public:
    void accept(DocumentVisitor* visitor) override {
        visitor->visit(this); // Передаем себя посетителю
    }

    void print() {
        std::cout << "Printing Word Document." << std::endl; // Операция для
Word-документа
    }

    void save() {
        std::cout << "Saving Word Document." << std::endl; // Операция для
Word-документа
    }
};

// Конкретный элемент (PDF-документ)
class PDFDocument : public Document {
public:
    void accept(DocumentVisitor* visitor) override {
        visitor->visit(this); // Передаем себя посетителю
    }

    void print() {
        std::cout << "Printing PDF Document." << std::endl; // Операция для
PDF-документа
    }

    void save() {
        std::cout << "Saving PDF Document." << std::endl; // Операция для
PDF-документа
    }
};

// Конкретный посетитель (Печать)
class PrintVisitor : public DocumentVisitor {
public:
    void visit(WordDocument* wordDoc) override {
        wordDoc->print(); // Вызываем операцию печати для Word-документа
    }
```

```
    void visit(PDFDocument* pdfDoc) override {
        pdfDoc->print(); // Вызываем операцию печати для PDF-документа
    }
};

// Конкретный посетитель (Сохранение)
class SaveVisitor : public DocumentVisitor {
public:
    void visit(WordDocument* wordDoc) override {
        wordDoc->save(); // Вызываем операцию сохранения для Word-документа
    }

    void visit(PDFDocument* pdfDoc) override {
        pdfDoc->save(); // Вызываем операцию сохранения для PDF-документа
    }
};

// Клиентский код
int main() {
    std::vector<std::shared_ptr<Document>> documents;
    documents.push_back(std::make_shared<WordDocument>());
    documents.push_back(std::make_shared<PDFDocument>());

    PrintVisitor printVisitor;
    SaveVisitor saveVisitor;

    // Печатаем все документы
    for (const auto& doc : documents) {
        doc->accept(&printVisitor);
    }

    std::cout << std::endl;

    // Сохраняем все документы
    for (const auto& doc : documents) {
        doc->accept
}
```