Оглавление

AbsractFactory 1
Адаптер....5
Builder 7
Chain of Responsibility (цепочка обязанностей) 11
Command 13
Composite (компоновщик) 16
Façade 19
Iterator 20
Mediator....22
Memento(Хранитель) 24
Observer(наблюдатель)....26
Prototype 28
Proxy(заместитель) 29
Singleton(одиночка)....30
State(состояние)....31
Strategy(стратегия)....34
Visitor 35
MVC....38

AbsractFactory

Абстрактная фабрика — это порождающий паттерн проектирования, который позволяет создавать семейства связанных объектов, не привязываясь к конкретным классам создаваемых объектов.

Структура:

1. Абстрактные продукты объявляют интерфейсы продуктов, которые связаны друг с другом по смыслу, но выполняют разные функции.
2. Конкретные продукты — большой набор классов, которые относятся к различным абстрактным продуктам (ферма1, ферма2), но имеют одни и те же вариации (коровы, овцы, куры).
3. Абстрактная фабрика объявляет методы создания различных абстрактных продуктов (ферма1, ферма2).
4. Конкретные фабрики относятся каждая к своей вариации продуктов (коровы, овцы, куры) и реализуют методы абстрактной фабрики, позволяя создавать все продукты определённой вариации.
5. Несмотря на то, что конкретные фабрики порождают конкретные продукты, сигнатуры их методов должны возвращать соответствующие абстрактные продукты. Это позволит клиентскому коду, использующему фабрику, не привязываться к конкретным классам продуктов. Клиент сможет работать с любыми вариациями продуктов через абстрактные интерфейсы.

Используйте паттерн Abstract Factory (абстрактная фабрика), если:

- Система должна оставаться независимой как от процесса создания новых объектов, так и от типов порождаемых объектов. Непосредственное использование выражения new в коде приложения нежелательно (подробнее об этом в разделе Порождающие паттерны).
- Необходимо создавать группы или семейства взаимосвязанных объектов, исключая возможность одновременного использования объектов из разных семейств в одном контексте.

Любое семейство или группа взаимосвязанных объектов характеризуется несколькими общими типами создаваемых продуктов, при этом сами продукты таких типов будут различными для разных семейств. Например, для случая соревнования между фермерами общими типами создаваемых продуктов будут коровы, куры и овцы, при этом каждый тип животных одной фермы может существенно отличаться по внешнему виду и характеристикам от второй.

Для того чтобы система оставалась независимой от специфики того или иного семейства продуктов необходимо использовать общие интерфейсы для всех основных типов продуктов. В данном случае это означает, что необходимо использовать три абстрактных базовых класса для каждого типа животных. Производные от них классы будут реализовывать специфику соответствующего типа воинов той или иной армии.

Для решения задачи по созданию семейств взаимосвязанных объектов паттерн Abstract Factory вводит понятие абстрактной фабрики. Абстрактная фабрика представляет собой некоторый полиморфный базовый класс, назначением которого является объявление интерфейсов фабричных методов, служащих для создания продуктов всех основных типов (один фабричный метод на каждый тип продукта).

Производные от него классы, реализующие эти интерфейсы, предназначены для создания продуктов всех типов внутри семейства или группы. В данным случае базовый класс абстрактной фабрики должен определять интерфейс фабричных методов для создания овец, кур и коров, а два производных от него класса будут реализовывать этот интерфейс, создавая животных всех видов для той или иной фермы.

По коду:

Создаем абстрактные базовые классы для всех предполагаемых видов животных (курица, корова, овечка), в них определяем виртуальные функции, которые переопределим в наследуемых классах. В следующих классах переопределяем поведение функций (определяем принадлежность той или иной ферме). Потом создаем абстрактную фабрику для производства животных, от которой наследуем классы для производства животных и их продуктов жизнедеятельности (т.е. яиц, молока и шерсти) на двух фермах. Далее, нам нужны еще несколько вспомогательных классов: класс, содержащий всех животных той или иной фермы и количество молока/яиц/шерсти, произведенных ими; и класс для производства животных на определенной ферме. В мэйн мы как раз создаем объекты производства и сами фермы, заполняем их животными, а далее устраиваем соревнование.

Код:

```
#include <iostream>
#include <vector>
#include <cstdlib>

using namespace std;

// Абстрактные базовые классы всех возможных видов животных
class Chicken
{
public:
	virtual void info() = 0;
	virtual int eggs() = 0;
	virtual ~Chicken() {}
};

class Cow
{
public:
	virtual void info() = 0;
	virtual int milk() = 0;
	virtual ~Cow() {}
};

class Ship
{
public:
	virtual void info() = 0;
	virtual int wool() = 0;
	virtual ~Ship() {}
};


// Классы всех видов животных "Веселой фермы"
class FunnyFarmChicken : public Chicken
{
public:
	string color = "brown";
	void info() {
		cout << "FunnyFarmChicken" << endl;
```

```
        }
        int eggs() {
                int eggs = rand() % rand() / 10;
                cout << eggs << " eggs " << endl;
                return eggs;
        }

};

class FunnyFarmCow : public Cow
{
public:
        void info() {
                cout << "FunnyFarmCow" << endl;
        }
        int milk() {
                int milk = rand() % rand() / 10;
                cout << milk << " liters of milk " << endl;
                return milk;
        }
};

class FunnyFarmShip : public Ship
{
public:
        void info() {
                cout << "FunnyFarmShip" << endl;
        }
        int wool() {
                int wool = rand() % rand() / 10;
                cout << wool << " pounds of wool " << endl;
                return wool;
        }
};


// Классы всех видов животных фермы "Рога и Копыта"
class HornsAndHoovesChicken : public Chicken
{
public:
        string color = "white";
        void info() {
                cout << "HornsAndHoovesChicken" << endl;
        }

        int eggs() {
                int eggs = rand() % rand() / 10;
                cout << eggs << " eggs " << endl;
                return eggs;
        }
};

class HornsAndHoovesCow : public Cow
{
public:
        void info() {
                cout << "HornsAndHoovesCow" << endl;
        }

        int milk() {
                int milk = rand() % rand() / 10;
                cout << milk << " liters of milk " << endl;
                return milk;
        }
};

class HornsAndHoovesShip : public Ship
{
public:
        void info() {
                cout << "HornsAndHoovesShip" << endl;
        }
        int wool() {
                int wool = rand() % rand() / 10;
                cout << wool << " pounds of wool " << endl;
                return wool;
        }
};
```

```
// Абстрактная фабрика для производства животных
class AnimalsFactory
{
public:
	virtual Chicken* createChicken() = 0;
	virtual Cow* createCow() = 0;
	virtual Ship* createShip() = 0;
	virtual ~AnimalsFactory() {}
};


// Фабрика для создания животных для "Веселая ферма"
class FunnyFarmFactory : public AnimalsFactory
{
public:
	Chicken* createChicken() {
		return new FunnyFarmChicken;
	}
	Cow* createCow() {
		return new FunnyFarmCow;
	}
	Ship* createShip() {
		return new FunnyFarmShip;
	}
};


// Фабрика для создания животных для «Рога и Копыта»
class HornesAndHoovesFactory : public AnimalsFactory
{
public:
	Chicken* createChicken() {
		return new HornsAndHoovesChicken;
	}
	Cow* createCow() {
		return new HornsAndHoovesCow;
	}
	Ship* createShip() {
		return new HornsAndHoovesShip;
	}
};


// Класс, содержащий всех животных той или иной фермы
class Farm
{
public:

	vector<Chicken*> vc;
	vector<Cow*> vco;
	vector<Ship*> vs;

	~Farm() {
		int i;
		for (i = 0; i < vc.size(); i++) delete vc[i];
		for (i = 0; i < vco.size(); i++) delete vco[i];
		for (i = 0; i < vs.size(); i++) delete vs[i];
	}
	void info() {
		int i;
		for (i = 0; i < vc.size(); i++) vc[i]->info();
		for (i = 0; i < vco.size(); i++) vco[i]->info();
		for (i = 0; i < vs.size(); i++) vs[i]->info();
	}

	int total() {
		int i;
		int sum = 0;
		for (i = 0; i < vc.size(); i++) sum += vc[i]->eggs();
		for (i = 0; i < vco.size(); i++) sum += vco[i]->milk();
		for (i = 0; i < vs.size(); i++) sum += vs[i]->wool();

		return sum;
	}
};
```

```
// Здесь создаются животные той или иной фермы
class Production
{
public:
	Farm* createFarm(AnimalsFactory& factory) {
		Farm* p = new Farm;
		p->vc.push_back(factory.createChicken());
		p->vco.push_back(factory.createCow());
		p->vs.push_back(factory.createShip());
		return p;
	}
};


int main()
{
	srand(time(NULL));
	Production prod;
	FunnyFarmFactory FFFactory;
	HornesAndHoovesFactory HHFactory;

	Farm* FF = prod.createFarm(FFFactory);
	Farm* HH = prod.createFarm(HHFactory);

	cout << "Funny farm: " << endl << endl;
	FF->info();
	int FFTotal = FF->total();
	cout << FFTotal << " - FF sum" << endl;

	cout << "\nHorns And Hooves: " << endl << endl;
	HH->info();
	int HHTotal = HH->total();
	cout << HHTotal << " - HH sum" << endl << endl;

	if (FFTotal > HHTotal)
		cout << "FF win" << endl;
	else if (FFTotal < HHTotal)
		cout << "HH win" << endl;
	else
		cout << "Dead heat!" << endl;

	return 0;
}
```

Адаптер - структурный

Часто в новом программном проекте не удается повторно использовать уже существующий код. Например, имеющиеся классы могут обладать нужной функциональностью, но иметь при этом несовместимые интерфейсы. В таких случаях следует использовать паттерн Adapter (адаптер).

Паттерн Adapter, представляющий собой программную обертку над существующими классами, преобразует их интерфейсы к виду, пригодному для последующего использования.

Адаптер — это структурный паттерн, который позволяет подружить несовместимые объекты.

Адаптер выступает прослойкой между двумя объектами, превращая вызовы одного в вызовы понятные другому.

Паттерн Адаптер (Adapter) предназначен для преобразования интерфейса одного класса в интерфейс другого. Благодаря реализации данного паттерна мы можем использовать вместе классы с несовместимыми интерфейсами.

Когда надо использовать Адаптер?

- Когда необходимо использовать имеющийся класс, но его интерфейс не соответствует потребностям
- Когда надо использовать уже существующий класс совместно с другими классами, интерфейсы которых не совместимы

Участники

- Target: представляет объекты, которые используются клиентом
- Client: использует объекты Target для реализации своих задач

- Adaptee: представляет адаптируемый класс, который мы хотели бы использовать у клиента вместо объектов Target
- Adapter: собственно адаптер, который позволяет работать с объектами Adaptee как с объектами Target.

То есть клиент ничего не знает об Adaptee, он знает и использует только объекты Target. И благодаря адаптеру мы можем на клиенте использовать объекты Adaptee как Target

+Отделяет и скрывает от клиента подробности преобразования различных интерфейсов.

-Усложняет код программы из-за введения дополнительных классов.

Структура:

Адаптер объектов

Эта реализация использует агрегацию: объект адаптера «оборачивает», то есть содержит ссылку на служебный объект. Такой подход работает во всех языках программирования.

1. Клиент — это класс, который содержит существующую бизнес-логику программы.
2. Клиентский интерфейс описывает протокол, через который клиент может работать с другими классами.
3. Сервис — это какой-то полезный класс, обычно сторонний. Клиент не может использовать этот класс напрямую, так как сервис имеет непонятный ему интерфейс.
4. Адаптер — это класс, который может одновременно работать и с клиентом, и с сервисом. Он реализует клиентский интерфейс и содержит ссылку на объект сервиса. Адаптер получает вызовы от клиента через методы клиентского интерфейса, а затем переводит их в вызовы методов обёрнутого объекта в правильном формате.
5. Работая с адаптером через интерфейс, клиент не привязывается к конкретному классу адаптера. Благодаря этому, вы можете добавлять в программу новые виды адаптеров, независимо от клиентского кода. Это может пригодиться, если интерфейс сервиса вдруг изменится, например, после выхода новой версии сторонней библиотеки.

Рассмотрим простой пример, когда следует применять паттерн Adapter. Пусть в нашем отеле разработали новую систему входа в номера для обеспечения безопасности гостей отеля. Но злоумышленники хотят взломать номера в отеле, чтобы выкрасть все драгоценности. Для этого они нашли сообщника, который выдал им подсказку в виде некоторого класса с соответствующим интерфейсом. Однако использовать этот класс непосредственно не удастся, так как им нужно понять сам шифр, и нужен адаптер.

Создадим три класса: абстрактный целевой – с которым может работать код клиента(в данном случае хакеры), адаптируемый – нуждающийся в доработке(подсказка) и класс адаптера, который поможет разгадать тайну хакерам.

В целевом – создаем виртуальный метод запроса, который переопределим в адаптере и оставляем его.

Адаптируемый класс содержит метод с подсказкой.

А в классе адаптера, который мы наследуем от целевого класса, загружаем объект адаптируемого класса и переопределяем метод запроса, декодируя шифр.

Код:

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

using namespace std;

/**
 * Целевой класс объявляет интерфейс, с которым может работать клиентский код.
 */
class Target {
public:
    virtual string Request() = 0;
    virtual ~Target() {};
};

/**
 * Адаптируемый класс содержит некоторое полезное поведение, но его интерфейс
 * несовместим с существующим клиентским кодом. Адаптируемый класс нуждается в
 * некоторой доработке, прежде чем клиентский код сможет его использовать.
 */
```

```
class Adaptee {
public:
    string SpecificRequest() const {
        return "Lw lv wkh Zxhvdu zlskhu";
    }
};

/**
 * Адаптер делает интерфейс Адаптируемого класса совместимым с целевым
 * интерфейсом.
 */
class Adapter : public Target {
private:
    Adaptee* adaptee_;
public:
    Adapter(Adaptee* adaptee) : adaptee_(adaptee) {}
    string Request() override {
        string decoded = this->adaptee_->SpecificRequest();
        for (int i = 0; i < decoded.size(); i++) {
            if (decoded[i] == ' ')
                continue;
            else if (decoded[i] == 'x')
                decoded[i] = 'a';
            else if (decoded[i] == 'X')
                decoded[i] = 'A';
            else if (decoded[i] == 'Y')
                decoded[i] = 'B';
            else if (decoded[i] == 'y')
                decoded[i] = 'b';
            else if (decoded[i] == 'z')
                decoded[i] = 'c';
            else if (decoded[i] == 'Z')
                decoded[i] = 'C';
            else
                decoded[i] -= 3;
        }
        return "Adapter: " + decoded;
    }
};

/**
 * Клиентский код поддерживает все классы, использующие целевой интерфейс.
 */
void ClientCode(Target* target) {
    cout << target->Request();
}

int main() {
    Adaptee* adaptee = new Adaptee;
    cout << "Client: I can't open the door, it's encrypted!" << endl;
    cout << "Adaptee: " << adaptee->SpecificRequest();
    cout << endl << endl;

    cout << "Admin: But you can work with it via the Adapter" << endl;
    Adapter* adapter = new Adapter(adaptee);
    ClientCode(adapter);
    cout << endl << endl;
    cout << "Client: Thanks" << endl;
    delete adaptee;
    delete adapter;

    return 0;
}
```

Builder

Строитель — это порождающий паттерн проектирования, который позволяет создавать объекты пошагово.

В отличие от других порождающих паттернов, Строитель позволяет производить различные продукты, используя один и тот же процесс строительства.

Паттерн Строитель предлагает вынести конструирование объекта за пределы его собственного класса, поручив это дело отдельным объектам, называемым строителями.

Вы можете пойти дальше и выделить вызовы методов строителя в отдельный класс, называемый директором. В этом случае директор будет задавать порядок шагов строительства, а строитель — выполнять их.

Структура

1. Интерфейс строителя объявляет шаги конструирования продуктов, общие для всех видов строителей.
2. Конкретные строители реализуют строительные шаги, каждый по-своему. Конкретные строители могут производить разнородные объекты, не имеющие общего интерфейса.
3. Продукт — создаваемый объект. Продукты, сделанные разными строителями, не обязаны иметь общий интерфейс.
4. Директор определяет порядок вызова строительных шагов для производства той или иной конфигурации продуктов.
5. Обычно Клиент подаёт в конструктор директора уже готовый объект-строитель, и в дальнейшем данный директор использует только его. Но возможен и другой вариант, когда клиент передаёт строителя через параметр строительного метода директора. В этом случае можно каждый раз применять разных строителей для производства различных представлений объектов.

Привер: прораб и рабочие

Другой пример: Пицца — это сложный объект, который может быть сконфигурирован сотней разных способов. Вместо того, чтобы настраивать ее через конструктор, мы вынесем ее сборку в отдельный класс-строитель, предусмотрев методы для добавления различных продуктов. Клиент может заказывать пиццу, работая со строителем(работником пиццерии) напрямую. Но представим, что у нас на работу пришел стажер, и поручим это дело директору(а точнее, более опытному коллеге). Это объект, который знает, какие шаги строителя нужно вызвать, чтобы получить несколько самых популярных конфигураций пиццы.

Создаем класс пиццы, где уже она собрана, и происходит передача заказа клиенту(вывод на экран).

Далее, приступаем к строителям: создаем 2 класса строителей (абстрактный и конкретный). Абстрактный определяет, какие ингредиенты мы можем добавлять в пиццу в целом, а конкретный определяет их реализацию. Также, поскольку все пиццы у нас разные по итогу, то после возвращения конечного результата клиенту, экземпляр строителя должен быть готов к началу производства следующего продукта. Поэтому обычной практикой является вызов метода сброса в конце тела метода getProduct. Конкретные Строители должны предоставить свои собственные методы получения результатов. Это связано с тем, что различные типы строителей могут создавать совершенно разные продукты с разными интерфейсами. Например, мы могли бы создать еще один класс строителя, но уже для пицца-роллов.

Создаем класс директора, который знает, какие пиццы из чего состоят и может их собрать. Он отвечает только за выполнение шагов построения в определённой последовательности и управляет строителем (т.е. стажером). У него в подчинении как раз есть стажер(строитель), поэтому создаем объект строителя и инициализируем его. И определяем также конечные вариации пицц, которые есть в нашей пиццерии.

Также у нас есть метод клиентского кода, который взаимодействует с директором, создаёт объект-строитель, передаёт его директору, а затем инициирует процесс построения. Конечный результат извлекается из объекта-строителя. В данном случае, это может быть, например, директор пиццерии, который принял на работу стажера, а затем пришел посмотреть, как у него идут дела. И дал задание стажеру(т.е. строителю), приготовить какую-нибудь новую, необычную пиццу(мы можем взаимодействовать со строителями напрямую, без класса директора).

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

using namespace std;

/*
 * Имеет смысл использовать паттерн Строитель только тогда, когда ваши продукты
 * достаточно сложны и требуют обширной конфигурации.
 *
 * В отличие от других порождающих паттернов, различные конкретные строители
 * могут производить несвязанные продукты. Другими словами, результаты различных
 * строителей могут не всегда следовать одному и тому же интерфейсу.
 */

class Product1 {
public:
    std::vector<std::string> parts_;
    void ListParts()const {
        std::cout << "Product parts: ";
        for (size_t i = 0; i < parts_.size(); i++) {
            if (parts_[i] == parts_.back()) {
                std::cout << parts_[i];
```

```
            }
            else {
                std::cout << parts_[i] << ", ";
            }
        }
        std::cout << "\n\n";
    }
};


/**
 * Интерфейс Строителя объявляет создающие методы для различных частей объектов
 * Продуктов.
 */
class Builder {
public:
    virtual ~Builder() {}
    virtual void Cheese() const = 0;
    virtual void Sauce() const = 0;
    virtual void Meat() const = 0;
    virtual void Peperoni() const = 0;
    virtual void Pineapple() const = 0;
    virtual void Ham() const = 0;
    virtual void Pepper() const = 0;
    virtual void Onion() const = 0;
};
/**
 * Классы Конкретного Строителя следуют интерфейсу Строителя и предоставляют
 * конкретные реализации шагов построения. Ваша программа может иметь несколько
 * вариантов Строителей, реализованных по-разному.
 */
class ConcreteBuilder1 : public Builder {
private:

    Product1* product;

    /**
     * Новый экземпляр строителя должен содержать пустой объект продукта,
     * который используется в дальнейшей сборке.
     */
public:

    ConcreteBuilder1() {
        this->Reset();
    }

    ~ConcreteBuilder1() {
        delete product;
    }

    void Reset() {
        this->product = new Product1();
    }
    /**
     * Все этапы производства работают с одним и тем же экземпляром продукта.
     */

    void Cheese()const override {
        this->product->parts_.push_back("Cheese");
    }

    void Sauce()const override {
        this->product->parts_.push_back("Sauce");
    }

    void Meat()const override {
        this->product->parts_.push_back("Meat");
    }

    void Peperoni() const override {
        this->product->parts_.push_back("Peperoni");
    }

    void Pineapple() const override {
        this->product->parts_.push_back("Pineapple");
    }

    void Ham() const override {
```

```
        this->product->parts_.push_back("Ham");
    }

    void Pepper() const override {
        this->product->parts_.push_back("Pepper");
    }

    void Onion() const override {
        this->product->parts_.push_back("Onion");
    }
    /**
     * Конкретные Строители должны предоставить свои собственные методы
     * получения результатов. Это связано с тем, что различные типы строителей
     * могут создавать совершенно разные продукты с разными интерфейсами.
     * Поэтому такие методы не могут быть объявлены в базовом интерфейсе
     * Строителя (по крайней мере, в статически типизированном языке
     * программирования). Обратите внимание, что PHP является динамически
     * типизированным языком, и этот метод может быть в базовом интерфейсе.
     * Однако мы не будем объявлять его здесь для ясности.
     *
     * Как правило, после возвращения конечного результата клиенту, экземпляр
     * строителя должен быть готов к началу производства следующего продукта.
     * Поэтому обычной практикой является вызов метода сброса в конце тела
     * метода getProduct. Однако такое поведение не является обязательным, вы
     * можете заставить своих строителей ждать явного запроса на сброс из кода
     * клиента, прежде чем избавиться от предыдущего результата.
     */

    Product1* GetProduct() {
        Product1* result = this->product;
        this->Reset();
        return result;
    }
};

/**
 * Директор отвечает только за выполнение шагов построения в определённой
 * последовательности. Это полезно при производстве продуктов в определённом
 * порядке или особой конфигурации. Строго говоря, класс Директор необязателен,
 * так как клиент может напрямую управлять строителями.
 */
class Director {
    /**
     * @var Builder
     */
private:
    Builder* builder;
    /**
     * Директор работает с любым экземпляром строителя, который передаётся ему
     * клиентским кодом. Таким образом, клиентский код может изменить конечный
     * тип вновь собираемого продукта.
     */

public:

    void set_builder(Builder* builder) {
        this->builder = builder;
    }

    /**
     * Директор может строить несколько вариаций продукта, используя одинаковые
     * шаги построения.
     */

    void BuildMinimalViableProduct() {
        this->builder->Cheese();
        this->builder->Sauce();
    }

    void BuildFullFeaturedProduct() {
        this->builder->Cheese();
        this->builder->Sauce();
        this->builder->Meat();
        this->builder->Peperoni();
        this->builder->Pineapple();
        this->builder->Ham();
        this->builder->Pepper();
        this->builder->Onion();
```

```
    }
};
/**
 * Клиентский код создаёт объект-строитель, передаёт его директору, а затем
 * инициирует процесс построения. Конечный результат извлекается из объекта-
 * строителя.
 */
 /**
  * I used raw pointers for simplicity however you may prefer to use smart
  * pointers here
  */
void ClientCode(Director& director)
{
    ConcreteBuilder1* builder = new ConcreteBuilder1();
    director.set_builder(builder);
    std::cout << "Standard basic product:\n";
    director.BuildMinimalViableProduct();

    Product1* p = builder->GetProduct();
    p->ListParts();
    delete p;

    std::cout << "Standard full featured product:\n";
    director.BuildFullFeaturedProduct();

    p = builder->GetProduct();
    p->ListParts();
    delete p;

    // Помните, что паттерн Строитель можно использовать без класса Директор.
    std::cout << "Custom product:\n";
    builder->Ham();
    builder->Pineapple();
    p = builder->GetProduct();
    p->ListParts();
    delete p;

    delete builder;
}

int main() {
    Director* director = new Director();
    ClientCode(*director);
    delete director;
    return 0;
}
```

Chain of Responsibility (цепочка обязанностей)

Цепочка обязанностей — это поведенческий паттерн проектирования, который позволяет передавать запросы последовательно по цепочке обработчиков. Каждый последующий обработчик решает, может ли он обработать запрос сам и стоит ли передавать запрос дальше по цепи.

Обработчики прерывают цепь только когда они могут обработать запрос. В этом случае запрос движется по цепи, пока не найдётся обработчик, который может его обработать. Очень часто такой подход используется для передачи событий, создаваемых классами графического интерфейса в результате взаимодействия с пользователем.

Структура

1. Обработчик определяет общий для всех конкретных обработчиков интерфейс. Обычно достаточно описать единственный метод обработки запросов, но иногда здесь может быть объявлен и метод выставления следующего обработчика.
2. Базовый обработчик — опциональный класс, который позволяет избавиться от дублирования одного и того же кода во всех конкретных обработчиках. Обычно этот класс имеет поле для хранения ссылки на следующий обработчик в цепочке. Клиент связывает обработчики в цепь, подавая ссылку на следующий обработчик через конструктор или сеттер поля. Также здесь можно реализовать базовый метод обработки, который бы просто перенаправлял запрос следующему обработчику, проверив его наличие.
3. Конкретные обработчики содержат код обработки запросов. При получении запроса каждый обработчик решает, может ли он обработать запрос, а также стоит ли передать его следующему объекту. В большинстве случаев обработчики могут работать сами по себе и быть неизменяемыми, получив все нужные детали через параметры конструктора.

4. Клиент может либо сформировать цепочку обработчиков единожды, либо перестраивать её динамически, в зависимости от логики программы. Клиент может отправлять запросы любому из объектов цепочки, не обязательно первому из них.

Цепочка образуется из звеньев цепи - объектов, и имеет следующие особенности:

- каждое из звеньев цепи имеет ссылку на следующее звено в цепи (одно звено знает только про одно звено, кроме последнего звена, которое не знает ничего о другом звене), таким образом, при получении запроса звено может что-то сделать с запросом, и обязано передать запрос следующему звену в цепочке.
- если звену не интересен запрос, звено не обрабатывает запрос и просто передает его следующему звену
- длина цепочки не имеет никакого значения
- передавая запрос в первое звено, Вы можете быть уверены, что все звенья в цепи смогут его обработать
- есть и подход, когда звено не обязательно должно передавать запрос дальше

Пример: звонок в техподдержку МГТС с вопросом о том, почему не работает интернет

Другой пример: мы подготовили подарки к празднику и сложили их в один большой мешок, но не подписали их, поэтому никто не знает, где чей подарок. Однако, сюрпризов не предусмотрено, и каждый знает, что ему должны подарить. А для эффекта мероприятия будем вынимать подарки по одному.

Третий пример: Юля, Саша и Олег участвуют в олимпиадном программировании и решают задачки. Допустим, что на контестах есть три типа задач: жадные алгоритмы(Саша), графы(Олег) и мат.задачи(Юля). Каждая задача подается по очереди, и задачи распределяются между участниками. Но создатели этого контеста решили пошутить и добавили новый тип задач.

Создаем интерфейс и класс для абстрактного обработчика(хэндлера), в нем определяем метод для определения следующего из цепочки, и переопределяем метод для обработки для общего случая.

Далее, создаем, наследуемые от абстрактного хэндлера, конкретные хэндлеры для каждого из участников команды. Все Конкретные обработчики либо обрабатывают запрос, либо передают его следующему обработчику в цепочке.

Ну и создаем метод взаимодействия с пользователем, в который передаем обработчик, в зависимости цепочки, заданной в мэйне.

Допустим, что Юля заболела, и не сможет решать задачи на ближайшем контесте. В этом случае мы также можем пользоваться данной конструкцией: можно отправлять запрос любому обработчику в цепочке, не только первому.

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

using namespace std;

class Handler {
public:
    virtual Handler* SetNext(Handler* handler) = 0;
    virtual string Handle(string request) = 0;
};
/**
 * Поведение цепочки по умолчанию может быть реализовано внутри базового класса
 * обработчика.
 */
class AbstractHandler : public Handler {
private:
    Handler* next_handler_;

public:
    AbstractHandler() : next_handler_(NULL) {
    }
    Handler* SetNext(Handler* handler) override {
        this->next_handler_ = handler;
        return handler;
    }
    string Handle(string request) override {
        if (this->next_handler_) {
            return this->next_handler_->Handle(request);
        }

        return {};
    }
};
```

```
class JuliHandler : public AbstractHandler {
public:
    string Handle(string request) override {
        if (request == "Math") {
            return "Juli: I'll take the " + request + ".\n\n";
        }
        else {
            cout << "Juli: No, this is not my task." << endl;
            return AbstractHandler::Handle(request);
        }
    }
};
class OlegHandler : public AbstractHandler {
public:
    string Handle(string request) override {
        if (request == "Graphs") {
            return "Oleg: I'll take the " + request + ".\n\n";
        }
        else {
            cout << "Oleg: No, this is not my task." << endl;
            return AbstractHandler::Handle(request);
        }
    }
};
class AlexHandler : public AbstractHandler {
public:
    string Handle(string request) override {
        if (request == "Greedy") {
            return "Alex: I'll take the " + request + ".\n\n";
        }
        else {
            cout << "Alex: No, this is not my task." << endl;
            return AbstractHandler::Handle(request);
        }
    }
};


void ClientCode(Handler& handler) {
    vector<string> tasks = { "Greedy", "Фывафывапролпрол", "Graphs", "Math"};
    for (string& t : tasks) {
        cout << "Who'll take the " << t << " one?\n";
        string result = handler.Handle(t);
        if (!result.empty()) {
            cout << "  " << result;
        }
        else {
            cout << "  " << t << " was left unsolved.\n\n";
        }
    }
}

int main() {
    JuliHandler* Juli = new JuliHandler;
    OlegHandler* Oleg = new OlegHandler;
    AlexHandler* Alex = new AlexHandler;
    Juli->SetNext(Oleg)->SetNext(Alex);

    //можно отправлять запрос любому обработчику в цепочке, не только первому
    cout << "Chain: Juli -> Oleg  -> Alex\n\n";
    ClientCode(*Juli);
    cout << "\n";
    cout << "Subchain: Oleg -> Alex\n\n";
    ClientCode(*Oleg);

    delete Juli;
    delete Oleg;
    delete Alex;

    return 0;
}
```

Command

Команда — это поведенческий паттерн, позволяющий заворачивать запросы или простые операции в отдельные объекты.

Паттерн Command преобразовывает запрос на выполнение действия в отдельный объект-команду. Такая инкапсуляция позволяет передавать эти действия другим функциям и объектам в качестве параметра, приказывая им выполнить запрошенную операцию. Команда – это объект, поэтому над ней допустимы любые операции, что и над объектом.

Интерфейс командного объекта определяется абстрактным базовым классом Command и в самом простом случае имеет единственный метод execute(). Производные классы определяют получателя запроса (указатель на объект-получатель) и необходимую для выполнения операцию (метод этого объекта). Метод execute() подклассов Command просто вызывает нужную операцию получателя.

В паттерне Command может быть до трех участников:

- Клиент, создающий экземпляр командного объекта.
- Инициатор запроса, использующий командный объект. – Invoker(Romote)
- Получатель запроса. – Light(Reciever)

Сначала клиент создает объект ConcreteCommand, конфигурируя его получателем запроса. Этот объект также доступен инициатору. Инициатор использует его при отправке запроса, вызывая метод execute(). Этот алгоритм напоминает работу функции обратного вызова в процедурном программировании – функция регистрируется, чтобы быть вызванной позднее.

Паттерн Command отделяет объект, инициирующий операцию, от объекта, который знает, как ее выполнить. Единственное, что должен знать инициатор, это как отправить команду. Это придает системе гибкость: позволяет осуществлять динамическую замену команд, использовать сложные составные команды, осуществлять отмену операций.

Шаблон команды инкапсулирует запрос в виде объекта, что позволяет нам параметризовать другие объекты с помощью различных запросов, ставить в очередь или регистрировать запросы, а также поддерживать операции, которые нельзя выполнить.

Пример:

Поначалу это определение немного сбивает с толку, но давайте рассмотрим его. По аналогии с нашей проблемой выше, дистанционное управление является клиентом, а свет приемником. В шаблоне команды есть объект Command, который инкапсулирует запрос, связывая вместе набор действий на конкретном получателе. Он делает это, предоставляя только один метод execute(), который вызывает некоторые действия для получателя.

Ставить в очередь или регистрировать запросы, а также поддерживать отмену операций означает, что операция Execute команды может сохранять состояние для отмены ее эффектов в самой команде. Команда может иметь добавленную операцию unExecute, которая отменяет эффекты предыдущего вызова для выполнения. Она также может поддерживать регистрацию изменений, чтобы их можно было повторно применить в случае сбоя системы.

Создаем класс получателя – света(ламп), в нем определяем основные операции.

Создаем интерфейс команды, связывая его с объектом света, и переопределяем его в дочерних классах для включения и выключения разного света.

Наконец, создаем класс удаленного управления(посыльщика запросов к свету)

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

using namespace std;

class Light
{
public:
    void CeilingOn() {
        cout << "Ceiling light is on " << endl;
    }
    void CeilingOff() {
        cout << "Ceiling light is off " << endl;
    }
    void DesktopOn() {
        cout << "Desktop light is on " << endl;
    }
    void DesktopOff() {
        cout << "Desktop light is off " << endl;
```

```
    }
};

// Базовый класс
class Command
{
public:
    virtual ~Command() {}
    virtual void execute() = 0;
protected:
    Command(Light* l) : light(l) {}
    Light* light;
};

class CeilingLightOnCommand : public Command
{
public:
    CeilingLightOnCommand(Light* l) : Command(l) {}
    void execute() override {
        light->CeilingOn();
    }
};

class CeilingLightOffCommand : public Command
{
public:
    CeilingLightOffCommand(Light* l) : Command(l) {}
    void execute() override {
        light->CeilingOff();
    }
};

class DesktopLightOnCommand : public Command
{
public:
    DesktopLightOnCommand(Light* l) : Command(l) {}
    void execute() override {
        light->DesktopOn();
    }
};

class DesktopLightOffCommand : public Command
{
public:
    DesktopLightOffCommand(Light* l) : Command(l) {}
    void execute() override {
        light->DesktopOff();
    }
};

class Remote {
private:
    Command* on_start_;

    Command* on_finish_;
public:
    ~Remote() {
        delete on_start_;
        delete on_finish_;
    }

    void SetOnStart(Command* command) {
        this->on_start_ = command;
    }
    void SetOnFinish(Command* command) {
        this->on_finish_ = command;
    }
    /**
     * Отправитель не зависит от классов конкретных команд и получателей.
     * Отправитель передаёт запрос получателю косвенно, выполняя команду.
     */
    void DoSomethingImportant() {
        cout << "Remote: Does anybody want to turn on the light before other tasks?\n";
        if (this->on_start_) {
            this->on_start_->execute();
        }
        cout << "Remote: ...doing something really important...\n";
        cout << "Remote: Does anybody want to turn off the light after I finish?\n";
```

```
        if (this->on_finish_) {
            this->on_finish_->execute();
        }
    }
};

int main()
{

    Remote* remote = new Remote; //Invoker
    Light* l = new Light; //Receiver
    string choose;

    cout << "You're at home. Would you like to turn on ceiling/desktop light?" << endl;
    cout << "Press c for ceiling, d for desktop or n if you don't wanna light" << endl;
    cin >> choose;
    if (choose == "c")
    {
        remote->SetOnStart(new CeilingLightOnCommand(l));
        remote->SetOnFinish(new CeilingLightOffCommand(l));
    }
    else if (choose == "d")
    {
        remote->SetOnStart(new DesktopLightOnCommand(l));
        remote->SetOnFinish(new DesktopLightOffCommand(l));
    }

    remote->DoSomethingImportant();

    return 0;

}
```

Composite (компоновщик)

Компоновщик — это структурный паттерн проектирования, который позволяет сгруппировать множество объектов в древовидную структуру, а затем работать с ней так, как будто это единичный объект.

Структура

1. Компонент(айтем) определяет общий интерфейс для простых и составных компонентов дерева.
2. Лист(элемент букета) — это простой компонент дерева, не имеющий ответвлений. Из-за того, что им некому больше передавать выполнение, классы листьев будут содержать большую часть полезного кода.
3. Контейнер (или композит) — это составной компонент дерева. Он содержит набор дочерних компонентов, но ничего не знает об их типах. Это могут быть как простые компоненты-листья, так и другие компоненты-контейнеры. Но это не является проблемой, если все дочерние компоненты следуют единому интерфейсу. Методы контейнера переадресуют основную работу своим дочерним компонентам, хотя и могут добавлять что-то своё к результату.
4. Клиент работает с деревом через общий интерфейс компонентов. Благодаря этому, клиенту не важно, что перед ним находится — простой или составной компонент дерева.

Пример: у нас есть магазин цветов, и приходят покупатели, чтобы сделать заказ. Создаем класс компонента, общий как для составляющих, так и для заказов. Затем создаем классы для составляющих букета (цветов, украшений, обертки), где определяем им стоимость. Создаем класс-контейнер – компоновщик, где собираем букеты, а затем и заказ и считаем их общую стоимость. Ну и вынесем создание букета в отдельный метод. Далее, клиент приходит, выбираем необходимые опции, и ему говорят итоговую стоимость его заказа.

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

using namespace std;

// Component
class Item
{
public:
    virtual int getCost() = 0;
    virtual void addItem(Item* p) {
    }
    virtual ~Item() {}
};

// Primitives
```

```
class Rose : public Item
{
public:
    virtual int getCost() {
        return 200;
    }
};

class Lily : public Item
{
public:
    virtual int getCost() {
        return 300;
    }
};

class Orchid : public Item
{
public:
    virtual int getCost() {
        return 600;
    }
};

class Tulip : public Item
{
public:
    virtual int getCost() {
        return 200;
    }
};

class Peony : public Item
{
public:
    virtual int getCost() {
        return 200;
    }
};

class Eustoma : public Item
{
public:
    virtual int getCost() {
        return 70;
    }
};

class Eucalyptus : public Item
{
public:
    virtual int getCost() {
        return 100;
    }
};

class Gypsofila : public Item
{
public:
    virtual int getCost() {
        return 50;
    }
};

class Paper : public Item
{
public:
    virtual int getCost() {
        return 250;
    }
};

class Plastic : public Item
{
public:
    virtual int getCost() {
        return 100;
    }
```

```
};

// Composite
class CompositeItem : public Item
{
public:
    int getCost() {
        int total = 0;
        for (int i = 0; i < c.size(); ++i)
            total += c[i]->getCost();
        return total;
    }
    void addItem(Item* p) {
        c.push_back(p);
    }
    ~CompositeItem() {
        for (int i = 0; i < c.size(); ++i)
            delete c[i];
    }
private:
    std::vector<Item*> c;
};


// Вспомогательная функция для создания букета
CompositeItem* createBouquet(int num, string type, string add, string pack)
{
    CompositeItem* b = new CompositeItem;

    for (int i = 0; i < num; i++)
    {
        if (type == "rose")
            b->addItem(new Rose);
        else if (type == "lily")
            b->addItem(new Lily);
        else if (type == "orchid")
            b->addItem(new Orchid);
        else if (type == "tulip")
            b->addItem(new Tulip);
        else if (type == "peony")
            b->addItem(new Peony);
        else if (type == "gypsofila")
            b->addItem(new Gypsofila);
        else
            cout << "These flowers are not available." << endl;
    }
    if (add == "eustoma")
        b->addItem(new Eustoma);
    else if(add == "eucalyptus")
        b->addItem(new Eucalyptus);
    else if(add == "gypsofila")
        b->addItem(new Gypsofila);

    if (pack == "paper")
        b->addItem(new Paper);
    else if (pack == "plastic")
        b->addItem(new Plastic);

    return b;
}

int main()
{
    CompositeItem* order = new CompositeItem;
    int numBouquet;
    cout << "Hello! How many bouquets you want to order? ";
    cin >> numBouquet;
    for (int i = 0; i < numBouquet; i++)
    {
        int num;
        string type;
        string add;
        string pack;
        cout << "Which flowers do you want in your " << (i + 1)
             << " bouquet? We have rose, lily, orchid, tulip, peony, gypsofila" << endl;
        cin >> type;
        cout << "How much flowers do you want in this bouquet?" << endl;
        cin >> num;
```

```
        cout << "Do you want to add something? We have eustoma, eucalyptus and gypsofila " << endl;
        cin >> add;
        cout << "And the last one. Do you want to pack this bouquet? We have paper and plactic
packaging " << endl;
        cin >> pack;
        order->addItem(createBouquet(num, type, add, pack));
    }
    cout << "Your order costs " << order->getCost() << endl;
    delete order;
    return 0;
}
```

Façade

Фасад (Facade) представляет собой структурный шаблон проектирования, который позволяет скрыть сложность системы с помощью предоставления упрощенного интерфейса для взаимодействия с ней.

Когда использовать фасад?

- Когда имеется сложная система, и необходимо упростить с ней работу. Фасад позволит определить одну точку взаимодействия между клиентом и системой.
- Когда надо уменьшить количество зависимостей между клиентом и сложной системой. Фасадные объекты позволяют отделить, изолировать компоненты системы от клиента и развивать и работать с ними независимо.
- Когда нужно определить подсистемы компонентов в сложной системе. Создание фасадов для компонентов каждой отдельной подсистемы позволит упростить взаимодействие между ними и повысить их независимость друг от друга.

Рассмотрим применение паттерна в реальной задаче. Думаю, большинство программистов согласятся со мной, что писать в Visual Studio код одно удовольствие по сравнению с тем, как писался код ранее до появления интегрированных сред разработки. Мы просто пишем код, нажимаем на кнопку и все - приложение готово. В данном случае интегрированная среда разработки представляет собой фасад, который скрывает всю сложность процесса компиляции и запуска приложения. В данном случае компонентами системы являются класс текстового редактора TextEditor, класс компилятора Compiller и класс общеязыковой среды выполнения CLR. Клиентом выступает класс программиста, фасадом - класс VisualStudioFacade, который через свои методы делегирует выполнение работы компонентам и их методам. При этом надо учитывать, что клиент может при необходимости обращаться напрямую к компонентам, например, отдельно от других компонентов использовать текстовый редактор. Но в виду сложности процесса создания приложения лучше использовать фасад. Также это не единственный возможный фасад для работы с данными компонентами. При необходимости можно создавать альтернативные фасады также, как в реальной жизни мы можем использовать альтернативные среды разработки.

```
#include<iostream>

using namespace std;

// текстовый редактор
class TextEditor
{
public:
    void CreateCode()
    {
        cout << "Working on code..." << endl;
        cout << "#include <iostream>" << endl << endl;
        cout << "int main() {" << endl;
        cout << "\tstd::cout << \"Hello world!\" << std::endl;" << endl;
        cout << "\treturn 0;" << endl;
        cout << "}" << endl;
    }
    void Save()
    {
        cout << "Saving code..." << endl;
    }
};

class Compiler
{
public:
    void Compile()
    {
        cout << "\nApplication compilation..." << endl;
    }
```

```
};
class CLR
{
public:
    void Execute()
    {
        int t;
        cout << "\nApplication executing..." << endl;
        cout << "Hello world!" << endl;
        cout << "\nPress e to exit" << endl;
        cin >> t;
    }
    void Finish()
    {
        cout << "\nApplication shutting dowm..." << endl;
    }
};

class VisualStudioFacade
{
    TextEditor textEditor;
    Compiler compiler;
    CLR clr;
public:
    VisualStudioFacade(TextEditor te, Compiler compil, CLR clr)
    {
        textEditor = te;
        compiler = compil;
        clr = clr;
    }
    void Start()
    {
        textEditor.CreateCode();
        textEditor.Save();
        compiler.Compile();
        clr.Execute();
    }
    void Stop()
    {
        clr.Finish();
    }
};

class Programmer
{
public:
    void CreateApplication(VisualStudioFacade facade)
    {
        facade.Start();
        facade.Stop();
    }
};

int main()
{
        TextEditor textEditor = TextEditor();
        Compiler compiler = Compiler();
        CLR clr = CLR();

        VisualStudioFacade ide = VisualStudioFacade(textEditor, compiler, clr);

        Programmer* programmer = new Programmer;
        programmer->CreateApplication(ide);
}
```

Iterator

Итератор — это поведенческий паттерн, позволяющий последовательно обходить сложную коллекцию, без раскрытия деталей её реализации.

Назначение: даёт возможность последовательно обходить элементы составных объектов, не раскрывая их внутреннего представления.

Когда использовать итераторы?

- Когда необходимо осуществить обход объекта без раскрытия его внутренней структуры
- Когда имеется набор составных объектов, и надо обеспечить единый интерфейс для их перебора
- Когда необходимо предоставить несколько альтернативных вариантов перебора одного и того же объекта

Идея паттерна Итератор состоит в том, чтобы вынести поведение обхода коллекции из самой коллекции в отдельный класс. Объект-итератор будет отслеживать состояние обхода, текущую позицию в коллекции и сколько элементов ещё осталось обойти. Одну и ту же коллекцию смогут одновременно обходить различные итераторы, а сама коллекция не будет даже знать об этом. К тому же, если вам понадобится добавить новый способ обхода, вы сможете создать отдельный класс итератора, не изменяя существующий код коллекции.

Итератор представляет собой объект, позволяющий получить последовательный доступ к элементам объекта агрегата без использования описаний каждого их агрегированных объектов.

```
#include <iostream>

using namespace std;

// Класс стека данных
class Stack {
  private:
    // содержит приватный массив интовых элементов
    int items[10];
    // длина массива данных в нашем стеке
    int length;

  public:
    // доверяем приватные поля классу StackIterator
    friend class StackIterator;
    Stack() {
      length = -1;
    }
    // через push в массив будут добавляться элементы и инкрементироваться length
    void push(int value) {
      items[++length] = value;
    }
    // pop возвращает последний элемент массива и дикреметрирующий length
    int pop() {
      return items[length--];
    }
};

// Класс итератора
class StackIterator {
  private:
    // стек
    const Stack& stack;
    // текущий индекс в стеке
    int index;
  public:
    StackIterator(const Stack& stack) : stack(stack) {
      index = 0;
    }

    // перегрузим оператор инкремента увеличивает индекс
    void operator++() {
      index++;
    }

    // перегрузим оператор получения значения стека через возвращение элемента по текущему индексу
    int operator*() {
      return stack.items[index];
    }

    // скобки возвращают true, если текущий индекс не достиг длины стека
    bool operator()() {
      return index != stack.length + 1;
    }
};

// перегрузим оператор сравнения 2 экземпляров класса Stack
bool operator==(const Stack& s1, const Stack& s2) {
  StackIterator it1(s1), it2(s2);

  // Через перебор их итераторов будут сравниваться их элементы,
  // и если не случилось прерывания из-за неравенства элементов, то true иначе false
```

```
  for (; it1(); ++it1, ++it2) {
    if (*it1 != *it2) break;
  }
  return !it1() && !it2();
}

int main() {
  Stack stack1;

  for (int i = 0; i < 5; ++i) {
    stack1.push(i);
  }

  Stack stack2(stack1);

  cout << boolalpha << (stack1 == stack2) << endl;
  // true

  stack2.push(10);

  cout << boolalpha << (stack1 == stack2) << endl;
  // false
}
```

Mediator

Паттерн Посредник (Mediator) представляет поведенческий шаблон проектирования, который обеспечивает взаимодействие множества объектов без необходимости ссылаться друг на друга. Тем самым достигается слабосвязанность взаимодействующих объектов.

Когда используется паттерн Посредник?

- Когда имеется множество взаимосвязаных объектов, связи между которыми сложны и запутаны.
- Когда необходимо повторно использовать объект, однако повторное использование затруднено в силу сильных связей с другими объектами.

Участники

- Mediator: представляет интерфейс для взаимодействия с объектами Colleague
- Colleague: представляет интерфейс для взаимодействия с объектом Mediator
- ConcreteColleague1 и ConcreteColleague2: конкретные классы коллег, которые обмениваются друг с другом через объект Mediator
- ConcreteMediator: конкретный посредник, реализующий интерфейс типа Mediator

Пример: у нас есть студия создания веб-сайтов, и к нам обратился заказчик, чтобы мы поменяли ему цвет кнопки.

Как и в стандартном команде, у нас есть фронтэндер, бэкэндер, дизайнер, тестер, аналитики и тимлид, который их всех пинает. Лид в данном случае является посредником. Его и создадим для начала. Интерфейс Посредника предоставляет метод, используемый компонентами(подчиненными) для уведомления посредника о различных событиях. Посредник может реагировать на эти события и передавать исполнение другим компонентам(подчиненным).

Далее, создаем базовый класс коллег, которые подчиняются какому-нибудь лиду, и указываем это. Потом создаем уже классы для разных типов сотрудников. Конкретные Компоненты реализуют различную функциональность. Они не зависят от других компонентов. Они также не зависят от каких-либо конкретных классов посредников.

Затем создаем класс нашего лида, конкретного, в котором указываем конкретных подчиненных ему сотрудников. А также переопределяем функцию реакции на события.

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

using namespace std;

class Colleague;
class Mediator { //lead
public:
    virtual void Notify(Colleague* sender, std::string event) = 0;
};


class Colleague {
protected:
```

```
    Mediator* mediator_;

public:
    Colleague(Mediator* mediator = NULL) : mediator_(mediator) {
    }
    void set_mediator(Mediator* mediator) {
        this->mediator_ = mediator;
    }
};

class FrontDev : public Colleague {
public:
    void DoNFFront() {
        std::cout << "Front dev made feature.\n";
        this->mediator_->Notify(this, "nf");
    }
    void DoOFFront() {
        std::cout << "Front dev finalized feature.\n";
        this->mediator_->Notify(this, "of");
    }
};

class BackDev : public Colleague {
public:
    void DoNFBack() {
        std::cout << "Back dev made feature.\n";
        this->mediator_->Notify(this, "nf");
    }
    void DoOFBack() {
        std::cout << "Back dev finalized feature.\n";
        this->mediator_->Notify(this, "of");
    }
};

class Tester : public Colleague {
public:
    void DoBugsBack() {
        std::cout << "Tester found bugs in backend.\n";
        this->mediator_->Notify(this, "bb");
    }
    void DoBugsFront() {
        std::cout << "Tester found bugs in frontend.\n";
        this->mediator_->Notify(this, "bf");
    }
};

class Designer : public Colleague {
public:
    void DoSmth1() {
        std::cout << "Designer made a scheme.\n";
        this->mediator_->Notify(this, "s");
    }
    void DoSmth2() {
        std::cout << "Designer painted the most red colour of button.\n";
        this->mediator_->Notify(this, "b");
    }
};

class Analytics : public Colleague {
public:
    void DoSmth1() {
        std::cout << "Analytics does something...\n";
        this->mediator_->Notify(this, "smth1");
    }
    void DoSmth2() {
        std::cout << "Analytics does something again...\n";
        this->mediator_->Notify(this, "smth2");
    }
};

class ConcreteMediator : public Mediator {
private:
    FrontDev* frontDev_;
    BackDev* backDev_;
    Tester* tester_;
    Designer* designer_;
    Analytics* analytics_;
```

```
public:
    ConcreteMediator(Tester* t, Designer* d, BackDev* b, FrontDev* f, Analytics* a)
                    : tester_(t), designer_(d), backDev_(b), frontDev_(f), analytics_(a) {
        this->backDev_->set_mediator(this);
        this->frontDev_->set_mediator(this);
        this->tester_->set_mediator(this);
        this->designer_->set_mediator(this);
        this->analytics_->set_mediator(this);
    }
    void Notify(Colleague* sender, std::string event) override {
        if (event == "bf") {
            std::cout << "Mediator(Lead) reacts on bugs in front and triggers on front dev:\n";
            cout << "Front dev: OMG, again... \n";
            this->frontDev_->DoOFFront();
        }
        if (event == "b") {
            std::cout << "Mediator(Lead) reacts on new feature in design and triggers on front
dev:\n";
            cout << "Front dev: OMG... \n";
            this->frontDev_->DoNFFront();
        }
    }
};

int main() {
    Tester* t = new Tester;
    Designer* d = new Designer;
    BackDev* b = new BackDev;
    FrontDev* f = new FrontDev;
    Analytics* a = new Analytics;

    ConcreteMediator* lead = new ConcreteMediator(t, d, b, f, a);
    cout << "Client: I wanna this button to be red, not blue" << endl;
    cout << "Lead: OK, we'll do this. Designer, come here." << endl;
    d->DoSmth2();
    cout << "\nClient: now smth is wrong with my application, it doesn't work correctly. Can you
find and fix bugs?" << endl;
    cout << "Tester: my time has come..." << endl;
    t->DoBugsFront();
    cout << endl;

    delete t;
    delete d;
    delete b;
    delete f;
    delete a;
    delete lead;
    return 0;
}
```

Memento(Хранитель)

Снимок(хранитель) — это поведенческий паттерн проектирования, который позволяет сохранять и восстанавливать прошлые состояния объектов, не раскрывая подробностей их реализации.

Паттерн Хранитель (Memento) позволяет выносить внутреннее состояние объекта за его пределы для последующего возможного восстановления объекта без нарушения принципа инкапсуляции.

Когда использовать Memento?

- Когда нужно сохранить состояние объекта для возможного последующего восстановления
- Когда сохранение состояния должно проходить без нарушения принципа инкапсуляции

То есть ключевыми понятиями для данного паттерна являются сохранение внутреннего состояния и инкапсуляция, и важно соблюсти баланс между ними. Ведь, как правило, если мы не нарушаем инкапсуляцию, то состояние объекта хранится в объекте в приватных переменных. И не всегда для доступа к этим переменным есть методы или свойства с сеттерами и геттерами. Например, в игре происходит управление героем, все состояние которого заключено в нем самом - оружие героя, показатель жизней, силы, какие-то другие показатели. И нередко может возникнуть ситуация, сохранить все эти показатели во вне, чтобы в будущем можно было откатиться к предыдущему уровню и начать игру заново. В этом случае как раз и может помочь паттерн Хранитель.

Участники

- Memento: хранитель, который сохраняет состояние объекта Originator и предоставляет полный доступ только этому объекту Originator
- Originator: создает объект хранителя для сохранения своего состояния
- Caretaker: выполняет только функцию хранения объекта Memento, в то же время у него нет полного доступа к хранителю и никаких других операций над хранителем, кроме собственно сохранения, он не производит

Теперь рассмотрим реальный пример: нам надо сохранять состояние игрового персонажа в игре:

Здесь в роли Originator выступает класс Hero, состояние которого описывается количество патронов и жизней. Для хранения состояния игрового персонажа предназначен класс HeroMemento. С помощью метода SaveState() объект Hero может сохранить свое состояние в HeroMemento, а с помощью метода RestoreState() - восстановить.

Для хранения состояний предназначен класс GameHistory, причем все состояния хранятся в стеке, что позволяет с легкостью извлекать последнее сохраненное состояние.

Использование паттерна Memento дает нам следующие преимущества:

Уменьшение связанности системы

Сохранение инкапсуляции информации

Определение простого интерфейса для сохранения и восстановления состояния

В то же время мы можем столкнуться с недостатками, в частности, если требуется сохранение большого объема информации, то возрастут издержки на хранение всего объема состояния.

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

using namespace std;

// Memento
class HeroMemento
{
public:
    int Patrons{};
    int Lives{};

    HeroMemento(int patrons, int lives)
    {
        Patrons = patrons;
        Lives = lives;
    }
};

// Originator
class Hero
{
private:
    int patrons = 10; // кол-во патронов
    int lives = 5; // кол-во жизней
public:
    void Shoot()
    {
        if (patrons > 0)
        {
            patrons--;
            cout << "Shoot. It's " << patrons << "patrons left." << endl;
        }
        else
            cout << "No more";
    }
    // сохранение состояния
    HeroMemento SaveState()
    {
        cout << "Saving game..." << endl << patrons << "patrons, " << lives << " lives\n";
        return HeroMemento(patrons, lives);
    }
```

```
    // восстановление состояния
    void RestoreState(HeroMemento memento)
    {
        patrons = memento.Patrons;
        lives = memento.Lives;
        cout << "Restore game..." << endl << patrons << "patrons, " << lives << " lives\n";
    }
};


// Caretaker
class GameHistory
{
public:
    stack<HeroMemento> History; //get; private set; }
    GameHistory()
    {
        History = stack<HeroMemento>();
    }
};

int main()
{
    Hero hero = Hero();
    hero.Shoot(); // делаем выстрел, осталось 9 патронов
    GameHistory game = GameHistory();

    game.History.push(hero.SaveState()); // сохраняем игру

    hero.Shoot(); // делаем выстрел, осталось 8 патронов

    game.History.push(hero.SaveState()); // сохраняем игру

    game.History.pop();
    hero.RestoreState(game.History.top());

    hero.Shoot(); //делаем выстрел, осталось 8 патронов

    return 0;
}
```

Observer(наблюдатель)

Наблюдатель — это поведенческий паттерн, который позволяет объектам оповещать другие объекты об изменениях своего состояния.

Назначение: создаёт механизм подписки, позволяющий одним объектам следить и реагировать на события, происходящие в других объектах.

Обратите внимание, что существует множество различных терминов с похожими значениями, связанных с этим паттерном. Просто помните, что Субъекта также называют Издателем, а Наблюдателя часто называют Подписчиком и наоборот.

Для реализации паттерна Вам понадобится:

- Event (Событие) - событие, которое может возникнуть (например пришло письмо)
- Subject (Издатель) - объект, который уведомляет о событии
- Observer (Подписчик/Слушатель) - объект, который подписывается на событие и ожидает уведомление

Паттерн "Наблюдатель" (Observer) представляет поведенческий шаблон проектирования, который использует отношение "один ко многим". В этом отношении есть один наблюдаемый объект и множество наблюдателей. И при изменении наблюдаемого объекта автоматически происходит оповещение всех наблюдателей.

Данный паттерн еще называют Publisher-Subscriber (издатель-подписчик), поскольку отношения издателя и подписчиков характеризуют действие данного паттерна: подписчики подписываются email-рассылку определенного сайта. Сайт-издатель с помощью email-рассылки уведомляет всех подписчиков о изменениях. А подписчики получают изменения и производят определенные действия: могут зайти на сайт, могут проигнорировать уведомления и т.д.

Когда использовать паттерн Наблюдатель?

- Когда система состоит из множества классов, объекты которых должны находиться в согласованных состояниях
- Когда общая схема взаимодействия объектов предполагает две стороны: одна рассылает сообщения и является главным, другая получает сообщения и реагирует на них. Отделение логики обеих сторон позволяет их рассматривать независимо и использовать отдельно друга от друга.
- Когда существует один объект, рассылающий сообщения, и множество подписчиков, которые получают сообщения. При этом точное число подписчиков заранее неизвестно и процессе работы программы может изменяться.

Пример: мы – известный в узких кругах блогер и хотим следить за количеством подписчиков на стримах.

Создадим абстрактного потенциального наблюдателя – подписчика, и создадим метод для отслеживания его состояния(на стриме он или нет). А также создадим абстрактный стрим, в котором будут методы присоединения, отсоединения и уведомления. Издатель(стрим) владеет некоторым важным состоянием и оповещает наблюдателей о его изменениях.

В классе данного конкретного стрима устанавливаем его настройки: методы для подписки/отписки наблюдателя, уведомление о том, сколько подписчиков на стриме после каждого нового действия стримера.

Теперь необходимо реализовать функционал наблюдателя/подписчика. Кастомизируем его под каждого подписчика, создадим ему приветствие, прощание, а также настроим уведомление о новых действиях на стриме.

```
#include <iostream>
#include <list>
#include <string>
#include<clocale>

using namespace std;

class IObserver {
public:
    virtual ~IObserver() {};
    virtual void Update(const string& message_from_stream) = 0;
};

class IStream { //Subject
public:
    virtual ~IStream() {};
    virtual void Attach(IObserver* observer) = 0;
    virtual void Detach(IObserver* observer) = 0;
    virtual void Notify() = 0;
};


class Stream : public IStream {
private:
    list<IObserver*> list_observer_;
    string message_;

public:
    virtual ~Stream() {}

    void Attach(IObserver* observer) override {
        list_observer_.push_back(observer);
    }
    void Detach(IObserver* observer) override {
        list_observer_.remove(observer);
    }
    void Notify() override {
        list<IObserver*>::iterator iterator = list_observer_.begin();
        HowManyObserver();
        while (iterator != list_observer_.end()) {
            (*iterator)->Update(message_);
            ++iterator;
        }
    }

    void CreateMessage(string message = "Empty") {
        this->message_ = message;
        Notify();
    }
    void HowManyObserver() {
        cout << "There are " << list_observer_.size() << " observers in the list.\n";
    }
```

```
};

class Observer : public IObserver {
private:
    string message_from_stream_;
    Stream& stream_;
    static int static_number_;
    int number_;

public:
    Observer(Stream& stream) : stream_(stream) {
        this->stream_.Attach(this);
        cout << "A new Observer attached: Hi, I'm the Observer \"" << ++Observer::static_number_ <<
"\".\n";
        this->number_ = Observer::static_number_;
    }
    virtual ~Observer() {}

    void Update(const string& message_from_stream) override {
        message_from_stream_ = message_from_stream;
        PrintInfo();
    }
    void RemoveMeFromTheList() {
        stream_.Detach(this);
        cout << "Observer \"" << number_ << "\" removed from the list.\n";
    }
    void PrintInfo() {
        cout << "Observer \"" << this->number_ << "\": a new message is available --> " << this-
>message_from_stream_ << "\n";
    }
};

int Observer::static_number_ = 0;

int main() {
    setlocale(LC_ALL, "RUSSIAN");
    Stream* stream = new Stream; //стрим
    Observer* observer1 = new Observer(*stream);
    Observer* observer2 = new Observer(*stream);
    Observer* observer3 = new Observer(*stream);
    Observer* observer4;
    Observer* observer5;
    cout << endl << endl;
    stream->CreateMessage("Сап, двач");
    observer3->RemoveMeFromTheList();
    cout << endl << endl;
    stream->CreateMessage("Я Елена Малышева, и я расскажу вам, почему жить - здорово");
    observer4 = new Observer(*stream);
    cout << endl << endl;
    observer2->RemoveMeFromTheList();
    observer5 = new Observer(*stream);
    cout << endl << endl;
    stream->CreateMessage("Хаю-хай, с вами Ивангай");
    observer5->RemoveMeFromTheList();
    observer4->RemoveMeFromTheList();
    observer1->RemoveMeFromTheList();

    delete observer5;
    delete observer4;
    delete observer3;
    delete observer2;
    delete observer1;
    delete stream;
    return 0;
}
```

Prototype

Прототип — это порождающий паттерн проектирования, который позволяет копировать объекты, не вдаваясь в подробности их реализации.

Паттерн Прототип (Prototype) позволяет создавать объекты на основе уже ранее созданных объектов-прототипов. То есть по сути данный паттерн предлагает технику клонирования объектов.

Когда использовать Прототип?

- Когда конкретный тип создаваемого объекта должен определяться динамически во время выполнения
- Когда нежелательно создание отдельной иерархии классов фабрик для создания объектов-продуктов из параллельной иерархии классов (как это делается, например, при использовании паттерна Абстрактная фабрика)
- Когда клонирование объекта является более предпочтительным вариантом нежели его создание и инициализация с помощью конструктора. Особенно когда известно, что объект может принимать небольшое ограниченное число возможных состояний.

Участники

- Prototype: определяет интерфейс для клонирования самого себя, который, как правило, представляет метод Clone()
- ConcretePrototype1 и ConcretePrototype2: конкретные реализации прототипа. Реализуют метод Clone()
- Client: создает объекты прототипов с помощью метода Clone()

Proxy(заместитель)

Заместитель — это объект, который выступает прослойкой между клиентом и реальным сервисным объектом. Заместитель получает вызовы от клиента, выполняет свою функцию (контроль доступа, кеширование, изменение запроса и прочее), а затем передаёт вызов сервисному объекту.

Заместитель имеет тот же интерфейс, что и реальный объект, поэтому для клиента нет разницы — работать через заместителя или напрямую.

Для чего вообще контролировать доступ к объектам? Рассмотрим такой пример: у вас есть внешний ресурсоёмкий объект, который нужен не все время, а изредка.

Паттерн Заместитель предлагает создать новый класс-дублёр, имеющий тот же интерфейс, что и оригинальный служебный объект. При получении запроса от клиента объект-заместитель сам бы создавал экземпляр служебного объекта и переадресовывал бы ему всю реальную работу.

Структура

1. Интерфейс сервиса(директора) определяет общий интерфейс для сервиса и заместителя. Благодаря этому, объект заместителя можно использовать там, где ожидается объект сервиса.
2. Сервис содержит полезную бизнес-логику.
3. Заместитель хранит ссылку на объект сервиса. После того как заместитель заканчивает свою работу (например, инициализацию, логирование, защиту или другое), он передаёт вызовы вложенному сервису. Заместитель может сам отвечать за создание и удаление объекта сервиса.
4. Клиент работает с объектами через интерфейс сервиса. Благодаря этому, его можно «одурачить», подменив объект сервиса объектом заместителя.

Пример: директор и его зам – сервис и заместитель. Создаем для них общий интерфейс, от которого наследуем классы, для директора и зама. Интерфейс объявляет общие операции как для директора, так и для Заместителя. Пока клиент работает с директором, используя этот интерфейс, вы сможете передать ему заместителя вместо директора.

Директор содержит некоторую базовую бизнес-логику. Как правило, класс директора способен выполнять некоторую полезную работу, которая к тому же может быть очень медленной или точной – например, коррекция входных данных. Заместитель может решить эти задачи без каких-либо изменений в коде директора.

Заместитель хранит ссылку на объект класса директора. Клиент может либо лениво загрузить его, либо передать Заместителю.

Клиентский код должен работать со всеми объектами (как с реальными, так и заместителями) через общий интерфейс, чтобы поддерживать как директоров, так и заместителей. В реальной жизни, однако, клиенты в основном работают с реальными субъектами напрямую. В этом случае, для более простой реализации паттерна, можно расширить заместителя из класса реального субъекта.

```
#include <iostream>

using namespace std;

class BoardOfDirectors { //Совет директоров
public:
    virtual void Request() = 0;
```

```
};

class Director : public BoardOfDirectors {
public:
    void Request() override {
        cout << "Director: I'll solve your problem.\n";
    }
};

class Proxy : public BoardOfDirectors {
private:
    Director* director_;

    bool CheckAccess() {
        cout << "Proxy: Checking access of my director.\n";
        string ans;
        cout << "Proxy: Are you busy, director? y/n\n";
        cin >> ans;
        if (ans == "y")
            return false;
        else
            return true;
    }

public:
    Proxy(Director* director) : director_(new Director(*director)) {
    }

    ~Proxy() {
        delete director_;
    }

    void Request() override {
        if (this->CheckAccess()) {
            this->director_->Request();
        }
        else
            cout << "Proxy: Tell me about your problem. May be I can help you myself." << endl;

    }
};


int main() {
    cout << "Client: Executing the client code with a real subject:\n";
    Director* director = new Director;
    director->Request();
    cout << "\n";
    cout << "Client: Executing the same client code with a proxy:\n";
    Proxy* proxy = new Proxy(director);
    proxy->Request();

    delete director;
    delete proxy;
    return 0;
}
```

Singleton(одиночка)

Одиночка (Singleton, Синглтон) - порождающий паттерн, который гарантирует, что для определенного класса будет создан только один объект, а также предоставит к этому объекту точку доступа.

Когда надо использовать Синглтон? Когда необходимо, чтобы для класса существовал только один экземпляр

Синглтон позволяет создать объект только при его необходимости. Если объект не нужен, то он не будет создан. В этом отличие синглтона от глобальных переменных.

В классе определяется статическая переменная - ссылка на конкретный экземпляр данного объекта и приватный конструктор. В статическом методе getInstance() этот конструктор вызывается для создания объекта, если, конечно, объект отсутствует и равен null.

Пример: в детском саду у каждого ребенка в один момент времени только один шкафчик, и 2 шкафа одновременно ему иметь нельзя. Создадим класс шкафчика(синглтона): он предоставляет метод `GetInstance`, который ведёт себя как альтернативный конструктор и позволяет клиентам получать один и тот же экземпляр класса при каждом вызове. Конструктор Одиночки всегда должен быть скрытым, чтобы предотвратить создание объекта через оператор new.

Это статический метод, управляющий доступом к экземпляру одиночки. При первом запуске, он создаёт экземпляр одиночки и помещает его в статическое поле. При последующих запусках, он возвращает клиенту объект, хранящийся в статическом поле.

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

using namespace std;

class Cupboard
{
protected:
    Cupboard(const std::string value) : value_(value){}

    static Cupboard* c;

    string value_;

public:

    //Одиночки не должны быть клонируемыми
    Cupboard(Cupboard& other) = delete;
    //Одиночкам не должны присваиваться значения
    void operator=(const Cupboard&) = delete;

    static Cupboard* GetInstance(const std::string& value);

    string value() const {
        return value_;
    }
};

Cupboard* Cupboard::c = nullptr;

//статик методы вне класса
Cupboard* Cupboard::GetInstance(const std::string& value)
{
    if (c == nullptr) {
        c = new Cupboard(value);
    }
    return c;
}

class Person
{

public:
    Person() {}
    Cupboard* c;
    void Launch(string cName)
    {
        c = Cupboard::GetInstance(cName);
        std::cout << c->value() << "\n";
    }

};
int main()
{
    Person p = Person();
    p.Launch("Raspberry");
    // у нас не получится изменить шкаф, так как объект уже создан
    p.c = Cupboard::GetInstance("Cranberry");
    cout << p.c->value() << endl;


}
```

State(состояние)

Состояние (State) – поведенческий шаблон проектирования, который позволяет объекту изменять свое поведение в зависимости от внутреннего состояния.

Когда применяется данный паттерн?

- Когда поведение объекта должно зависеть от его состояния и может изменяться динамически во время выполнения
- Когда в коде методов объекта используются многочисленные условные конструкции, выбор которых зависит от текущего состояния объекта

Участники паттерна

- State: определяет интерфейс состояния
- Классы StateA и StateB - конкретные реализации состояний
- Context: представляет объект, поведение которого должно динамически изменяться в соответствии с состоянием. Выполнение же конкретных действий делегируется объекту состояния

Паттерн Состояние предлагает создать отдельные классы для каждого состояния, в котором может пребывать объект, а затем вынести туда поведения, соответствующие этим состояниям.

Вместо того, чтобы хранить код всех состояний, первоначальный объект, называемый контекстом, будет содержать ссылку на один из объектов-состояний и делегировать ему работу, зависящую от состояния.

Например, вода может находиться в ряде состояний: твердое, жидкое, парообразное. Допустим, нам надо определить класс Вода, у которого бы имелись методы для нагревания и заморозки воды.

Пример: мы работаем на заводе светофоров, и нам выдают ими зарплату. Но мы не отчаиваемся, и хотим устроить с ними дискотеку. Для этого нам надо их запрограммировать так, чтобы они мигали как гирлянды. Для этого создадим класс LightState – он определяет интерфейс состояния – передачи управления тому или иному цвету. Пронаследуем от него классы красного, желтого и зеленого цветов – переопределим хэндл и добавим метод `GetInstance`, который ведёт себя как альтернативный конструктор и позволяет клиентам получать один и тот же экземпляр класса при каждом вызове. Ну и сам класс для светофоров, которые мы программируем, в нем прописываем действия для того, чтобы получить «дискотеку».

```
#include <iostream>
#include <windows.h>

using namespace std;

//state class
class LightState
{
public:
    virtual void Handle() = 0;
};

class RedLight : public LightState
{
public:
    static RedLight* GetInstance()
    {
        static RedLight redL;
        return &redL;
    }
    virtual void Handle()
    {
        cout << "Red is glowing for 1 sec" << endl;
    }


};

class YellowLight :public LightState
{
public:
    static YellowLight* GetInstance()
    {
        static YellowLight yellowL;
        return &yellowL;
    }

    void Handle()
    {
        cout << "Yellow is glowing for 1 sec" << endl;
    }
};

class GreenLight : public LightState
```

```
{
public:
    static GreenLight* GetInstance()
    {
        static GreenLight greenL;
        return &greenL;
    }


    void Handle()
    {
        cout << "Green is glowing for 1 sec" << endl;
    }

};

class Controller //светофор
{
    LightState* lState;
public:
    Controller()
    {
        lState = nullptr;
    }
    void run()
    {
        int i = 0;
        int l = 1;
        int j = 0;
        while (i < 3)
        {

            lState = RedLight::GetInstance();
            lState->Handle();
            Sleep(1000);
            system("CLS");
            for (j; j < l; j++)
            {
                cout << endl;
            }
            l++;
            j = 0;
            lState = YellowLight::GetInstance();
            lState->Handle();
            Sleep(1000);
            system("CLS");
            for (j; j < l; j++)
            {
                cout << endl;
            }
            l++;
            j = 0;
            lState = GreenLight::GetInstance();
            lState->Handle();
            Sleep(1000);
            system("CLS");
            for (j; j < l; j++)
            {
                cout << endl;
            }
            l++;
            j = 0;
            i++;

        }
        cout << "Party is over..." << endl;

    }

};


int main()
{
    cout << "Disco dance!" << endl;
    Sleep(2000);
    system("CLS");
    Controller c;
```

```
    c.run();
    return 0;
}
```

Strategy(стратегия)

Паттерн Стратегия (Strategy) представляет поведенческий шаблон проектирования, который определяет набор алгоритмов, инкапсулирует каждый из них и обеспечивает их взаимозаменяемость. В зависимости от ситуации мы можем легко заменить один используемый алгоритм другим. При этом замена алгоритма происходит независимо от объекта, который использует данный алгоритм.

Когда использовать стратегию?

- Когда есть несколько родственных классов, которые отличаются поведением. Можно задать один основной класс, а разные варианты поведения вынести в отдельные классы и при необходимости их применять
- Когда необходимо обеспечить выбор из нескольких вариантов алгоритмов, которые можно легко менять в зависимости от условий
- Когда необходимо менять поведение объектов на стадии выполнения программы
- Когда класс, применяющий определенную функциональность, ничего не должен знать о ее реализации

Участники

- Интерфейс IStrategy, который определяет метод Algorithm(). Это общий интерфейс для всех реализующих его алгоритмов. Вместо интерфейса здесь также можно было бы использовать абстрактный класс.
- Классы ConcreteStrategy1 и ConcreteStrategy, которые реализуют интерфейс IStrategy, предоставляя свою версию метода Algorithm(). Подобных классов-реализаций может быть множество.
- Класс Context хранит ссылку на объект IStrategy и связан с интерфейсом IStrategy отношением агрегации.

В данном случае объект IStrategy заключена в свойстве ContextStrategy, хотя также для нее можно было бы определить приватную переменную, а для динамической установки использовать специальный метод.

Теперь рассмотрим конкретный пример. Существуют различные легковые машины, которые используют разные источники энергии: электричество, бензин, газ и так далее. Есть гибридные автомобили. В целом они похожи и отличаются преимущественно видом источника энергии. Не говоря уже о том, что мы можем изменить применяемый источник энергии, модифицировав автомобиль. И в данном случае вполне можно применить паттерн стратегию.

Паттерн Стратегия предлагает определить семейство схожих алгоритмов, которые часто изменяются или расширяются, и вынести их в собственные классы, называемые стратегиями.

Вместо того, чтобы изначальный класс сам выполнял тот или иной алгоритм, он будет играть роль контекста, ссылаясь на одну из стратегий и делегируя ей выполнение работы. Чтобы сменить алгоритм, вам будет достаточно подставить в контекст другой объект-стратегию.

Важно, чтобы все стратегии имели общий интерфейс. Используя этот интерфейс, контекст будет независимым от конкретных классов стратегий. С другой стороны, вы сможете изменять и добавлять новые виды алгоритмов, не трогая код контекста.

Пример: Вам нужно добраться до аэропорта. Можно доехать на автобусе, такси или велосипеде. Здесь вид транспорта является стратегией. Вы выбираете конкретную стратегию в зависимости от контекста — наличия денег или времени до отлёта. Согдаем изначальный интерфейс стратегий, где определяем метод действия(move). Интерфейс Стратегии объявляет операции, общие для всех поддерживаемых версий некоторого алгоритма. Контекст использует этот интерфейс для вызова алгоритма, определённого Конкретными Стратегиями. Также создаем класс контекста, в данном случае транспортировки. Контекст определяет интерфейс, представляющий интерес для клиентов. Контекст хранит ссылку на один из объектов Стратегии. Контекст не знает конкретного класса стратегии. Он должен работать со всеми стратегиями через интерфейс Стратегии.

Обычно Контекст принимает стратегию через конструктор, а также предоставляет сеттер для её изменения во время выполнения. Обычно Контекст позволяет заменить объект Стратегии во время выполнения. – но в данном случае этого нет)))

Вместо того, чтобы самостоятельно реализовывать множественные версии алгоритма, Контекст делегирует некоторую работу объекту Стратегии.

Классы конкретных Стратегий (такси и автобус) реализуют алгоритм, следуя базовому интерфейсу Стратегии. Этот интерфейс делает их взаимозаменяемыми в Контексте(транспортировке). Клиентский код выбирает конкретную

стратегию и передаёт её в контекст. Клиент должен знать о различиях между стратегиями, чтобы сделать правильный выбор.

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

using namespace std;

class Strategy
{
public:
    virtual ~Strategy() {}
    virtual string move()  = 0;
};


class Transport //контекст
{
private:
    Strategy* strategy_;

public:
    Transport(Strategy* strategy = nullptr) : strategy_(strategy)
    {
    }
    ~Transport()
    {
        delete this->strategy_;
    }

    void set_strategy(Strategy* strategy)
    {
        delete this->strategy_;
        this->strategy_ = strategy;
    }
    void action()
    {
        cout << "Transport: Performing actions using the strategy\n";
        string result = this->strategy_->move();
        cout << result << "\n";
    }
};

class Taxi : public Strategy
{
public:
    string move()  override
    {
        return "Client: Moving by taxi...So expensive...";
    }
};
class Bus : public Strategy
{
    string move()  override
    {
        return "Client: Moving by bus...So long...";;
    }
};

int main()
{
    Transport* t = new Transport(new Taxi);
    cout << "Client: I have only 1.5 hours! Strategy is set to go by taxi.\n";
    t->action();
    cout << "\n";
    cout << "Client: Taxi is too expensive for me - high demand. Strategy is set to go by bus.\n";
    t->set_strategy(new Bus);
    t->action();
    delete t;
    return 0;
}
```

Visitor

Паттерн Посетитель (Visitor) позволяет определить операцию для объектов других классов без изменения этих классов.

При использовании паттерна Посетитель определяются две иерархии классов: одна для элементов, для которых надо определить новую операцию, и вторая иерархия для посетителей, описывающих данную операцию.

Когда использовать данный паттерн?

- Когда имеется много объектов разнородных классов с разными интерфейсами, и требуется выполнить ряд операций над каждым из этих объектов
- Когда классам необходимо добавить одинаковый набор операций без изменения этих классов
- Когда часто добавляются новые операции к классам, при этом общая структура классов стабильна и практически не изменяется

Участники

- Visitor: интерфейс посетителя, который определяет метод Visit() для каждого объекта Element
- ConcreteVisitor1 / ConcreteVisitor2: конкретные классы посетителей, реализуют интерфейс, определенный в Visitor.
- Element: определяет метод Accept(), в котором в качестве параметра принимается объект Visitor
- ElementA / ElementB: конкретные элементы, которые реализуют метод Accept()

ObjectStructure: некоторая структура, которая хранит объекты Element и предоставляет к ним доступ. Это могут быть и простые списки, и сложные составные структуры в виде деревьев

Сущность работы паттерна состоит в том, что вначале создает объект посетителя, который обходит или посещает все элементы в структуре ObjectStructure, у которой вызывается метод Accept():

Пример: вернемся в 2000-е, когда интернет не был так распространен. Во многие квартиры приходили продажники и пытались что-то продать, зачастую ненужное. Итак, у нас есть 2 класса посетителей(продажников) и 2 класса людей, к которым они приходят – из элитных ЖК и из обычных. В зависимости от того, к кому они приходят, они предлагают свои товары по той или иной цене.

Создадим интерфейс Посетителя, который объявляет набор методов посещения, соответствующих классам компонентов(жителей) – visitelite/visitusual. Сигнатура метода посещения позволяет посетителю определить конкретный класс компонента, с которым он имеет дело.

Объявим также интерфейс Жителей дома(Компонента), он объявляет метод accept, который в качестве аргумента может получать любой объект, реализующий интерфейс посетителя. Каждый Конкретный Компонент должен реализовать метод accept таким образом, чтобы он вызывал метод посетителя, соответствующий классу компонента.

Обратите внимание, мы вызываем visitEliteHouse, что соответствует названию текущего класса. Таким образом мы позволяем посетителю узнать, с каким классом компонента он работает.

Конкретные (жители разных домов) Компоненты могут иметь особые методы, не объявленные в их базовом классе или интерфейсе. Посетитель всё же может использовать эти методы, поскольку он знает о конкретном классе компонента.

Конкретные Посетители(продажники) реализуют несколько версий одного и того же алгоритма, которые могут работать со всеми классами конкретных компонентов. Максимальную выгоду от паттерна Посетитель вы почувствуете, используя его со сложной структурой объектов, такой как дерево Компоновщика. В этом случае было бы полезно хранить некоторое промежуточное состояние алгоритма при выполнении методов посетителя над различными объектами структуры.

Клиентский код может выполнять операции посетителя над любым набором элементов, не выясняя их конкретных классов. Операция принятия направляет вызов к соответствующей операции в объекте посетителя.

```
#include <iostream>
#include <algorithm>
#include <array>
#include <string>
#include <clocale>
using namespace std;

class EliteHouse;
class UsualHouse;

class Visitor { //продажники
public:
    virtual void VisitElite(const EliteHouse* element) const = 0;
    virtual void VisitUsual(const UsualHouse* element) const = 0;
};
```

```cpp
class House { //жители - компонент
public:
    virtual ~House() {}
    virtual void Accept(Visitor* visitor) const = 0;
};


class EliteHouse : public House { //элитные ЖК

public:
    void Accept(Visitor* visitor) const override {
        visitor->VisitElite(this);
    }

    string ExclusiveMethodOfEliteHouse() const {
        return "Для вас - предложение по специальной цене: скидка 50%";
    }
};

class UsualHouse : public House { //обычные ЖК

public:
    void Accept(Visitor* visitor) const override {
        visitor->VisitUsual(this);
    }
    string SpecialMethodOfUsualHouse() const {
        return "Для вас - предложение по специальной цене: скидка 50%";
    }
};


class HooverVisitor : public Visitor { //чел с пылесосами
public:
    void VisitElite(const EliteHouse* element) const override {
        cout << element->ExclusiveMethodOfEliteHouse() << " + \nНаш уникальный пылесос всего за
300000 \n";
    }

    void VisitUsual(const UsualHouse* element) const override {
        cout << element->SpecialMethodOfUsualHouse() << " + \nНаш уникальный пылесос всего за
150000 \n";
    }
};

class CosmeticsVisitor : public Visitor { //чел с косметикой
public:
    void VisitElite(const EliteHouse* element) const override {
        cout << element->ExclusiveMethodOfEliteHouse() << " \nСамый лучший набор косметики для
ухода за собой - всего за 7000\n";
    }
    void VisitUsual(const UsualHouse* element) const override {
        cout << element->SpecialMethodOfUsualHouse() << " \nСамый лучший набор косметики для ухода
за собой - всего за 3500\n";
    }
};


int main() {
    setlocale(LC_ALL, "Russian");
    array<const House*, 2> components = { new EliteHouse, new UsualHouse };
    cout << "The client code works with all visitors via the base Visitor interface:\n";
    HooverVisitor* visitor1 = new HooverVisitor;
    for (const House* comp : components) {
        comp->Accept(visitor1);
    }

    cout << "\n";
    cout << "It allows the same client code to work with different types of visitors:\n";
    CosmeticsVisitor* visitor2 = new CosmeticsVisitor;
    for (const House* comp : components) {
        comp->Accept(visitor2);
    }

    for (const House* comp : components) {
        delete comp;
    }
```

```
    delete visitor1;
    delete visitor2;

    return 0;
}
```

MVC