

# Порождающие паттерны

## Абстрактная фабрика

### *Abstract Factory*

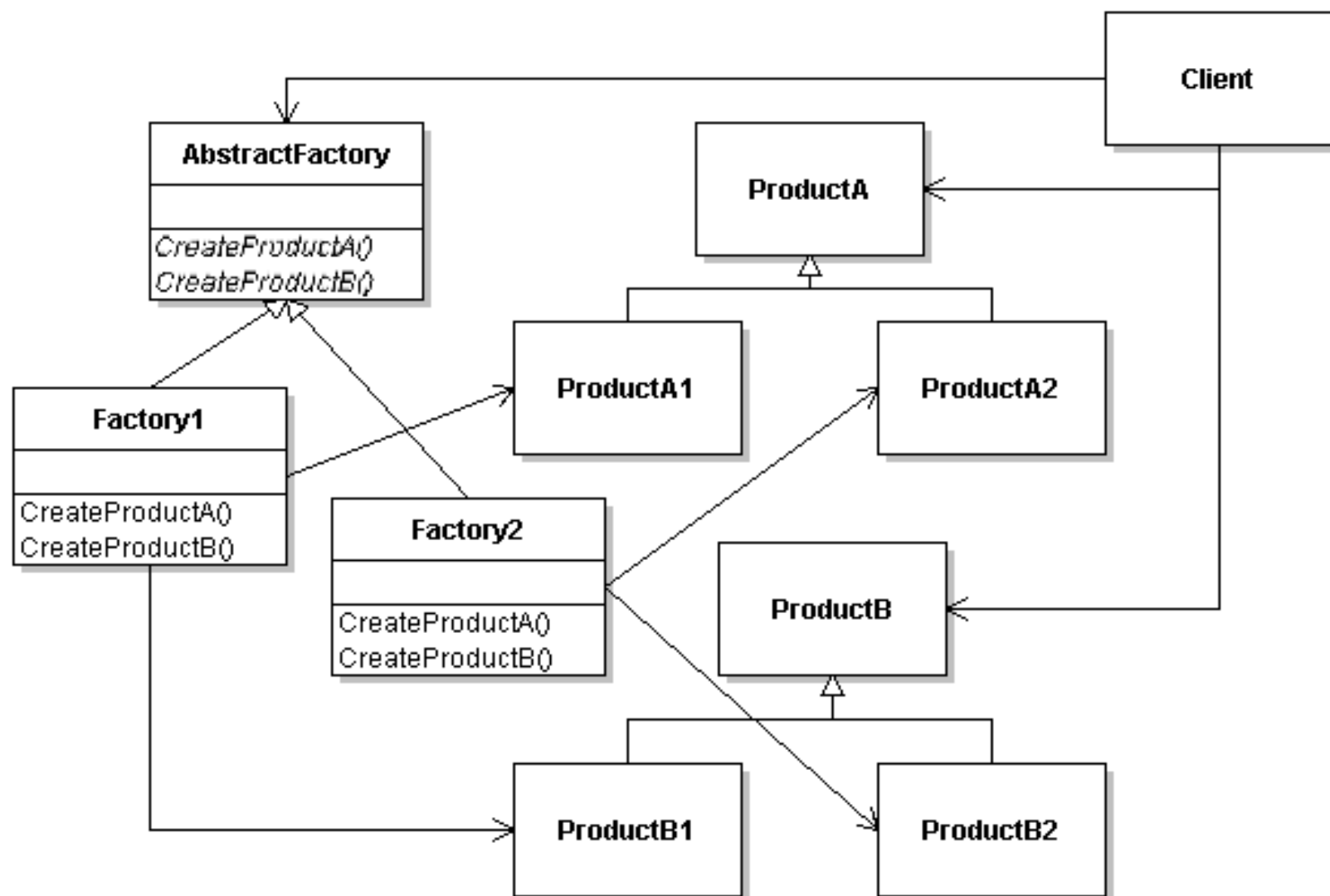
#### Назначение

Предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, не специфицируя их конкретных классов.

#### Результаты

1. Имена классов изготавливаемых объектов известны только конкретной фабрике, в коде клиента они не упоминаются. Клиент манипулируют объектами через их абстрактные интерфейсы.
2. Класс конкретной фабрики появляется в приложении только один раз при её создании. Это облегчает замену используемой приложением конкретной фабрики и семейства используемых продуктов.
3. Паттерн гарантирует сочетаемость создаваемых продуктов семейства.

# Структура



## Реализация

// абстрактные продукты

```
class ProductA {};
```

```
class ProductB {};
```

// Абстрактная фабрика – содержит методы для создания абстрактных объектов-продуктов

```
class AbstractFactory {
```

```
public:
```

```
    virtual ProductA *CreateProductA()=0;
```

```
    virtual ProductB *CreateProductB()=0;
```

```
};
```

// Конкретные продукты 1-го семейства

```
class ProductA1:public ProductA {};
```

```
class ProductB1:public ProductB {};
```

// Конкретная фабрика для создания продуктов 1-го семейства

```
class Factory1: public AbstractFactory {
```

```
public:
```

```
    ProductA1 *CreateProductA() { return new ProductA1(); }
```

```
    ProductB1 *CreateProductB() { return new ProductB1(); }
```

```
};
```

// Конкретные продукты 2-го семейства

```
class ProductA2:public ProductA {};
```

```
class ProductB2:public ProductB {};
```

```
// Конкретная фабрика для создания продуктов 1-го семейства
class Factory2: public AbstractFactory {
public:
    ProductA2 *CreateProductA() { return new ProductA2(); }
    ProductB2 *CreateProductB() { return new ProductB2(); }
};
// Создание фабрики
AbstractFactory *factory=new Factory1();
...
// Создание объектов в клиенте
ProductA *a=factory->CreateProductA();
```

## **Одиночка**

### *Singleton*

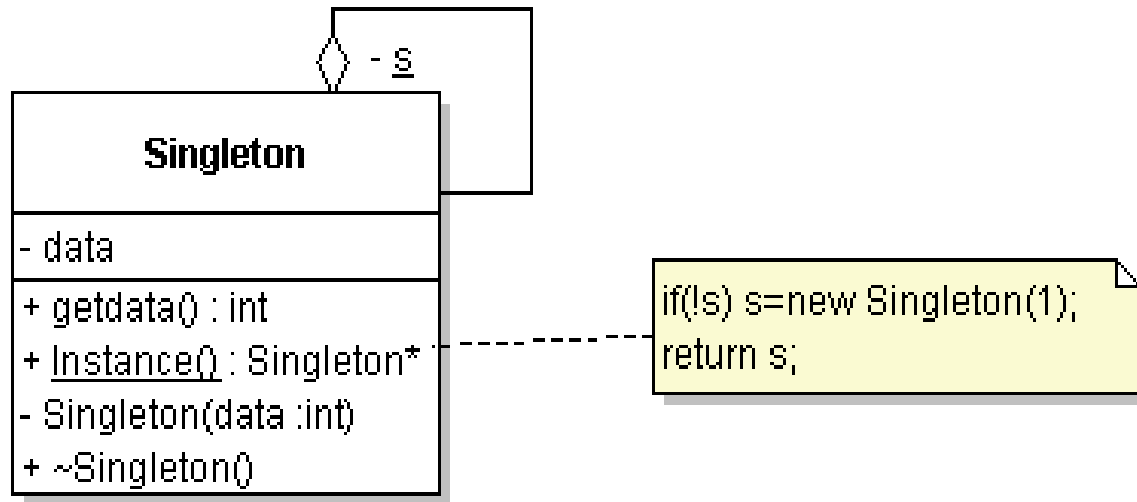
#### **Назначение**

Гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.

#### **Результаты**

1. Класс Singleton инкапсулирует свой единственный экземпляр, он полностью контролирует то, как и когда клиенты получают доступ к нему.
2. Паттерн одиночка позволяет избежать засорения пространства имен глобальными переменными, в которых хранятся уникальные экземпляры.
3. От класса Singleton можно порождать подклассы, а во время выполнения создавать экземпляр класса, заданного в конфигурации.
4. Паттерн позволяет использовать и более одного экземпляра класса Singleton. Для этого нужно изменить операцию доступа к экземпляру класса.

## Структура



## Реализация

// Интерфейс класса

```
class Singleton {
    int data; // какие-то данные
    static Singleton *s;
    // Создавать объекты могут только методы этого класса
    Singleton(int d):data(d){}
    // Запрещение создания копий и присваивания
    Singleton(const Singleton &);
    Singleton& operator=(const Singleton &);
public:
    // Метод для доступа к единственному экземпляру
    static Singleton *Instance();
```

```

~Singleton() { s=NULL; }
// метод для доступа к данным
int getdata() const { return data; }
};
// Реализация класса
Singleton *Singleton::s=NULL;
Singleton *Singleton::Instance()
{ if(!s)
    s=new Singleton(1);
  return s;
}
// Обращение к объекту в клиенте
int d=Singleton::Instance()->getdata();

```

Вариант реализации паттерна без возможности удаления клиентом экземпляра.

```

// Интерфейс класса
class Singleton {
    int data; // какие-то данные
    // Создавать объекты могут только методы этого класса
    Singleton(int d):data(d){}
    // Запрещение создания копий и присваивания
    Singleton(const Singleton &);
    Singleton& operator=(const Singleton &);

```

```
~Singleton() {}  
public:  
    // Метод для доступа к единственному экземпляру  
    static Singleton *Instance();  
    // метод для доступа к данным  
    int getdata() const { return data; }  
};  
// Реализация класса  
Singleton *Singleton::Instance()  
{ // Собственные объекты функции создаются только один раз  
    // при первом вызове этой функции  
    static Singleton s(1);  
    return &s;  
}  
// Обращение к объекту в клиенте  
int d=Singleton::Instance()->getdata();
```



## **Прототип**

*Prototype*

### **Назначение**

Задаёт виды создаваемых объектов с помощью экземпляра-прототипа и создаёт новые объекты путем копирования этого прототипа.

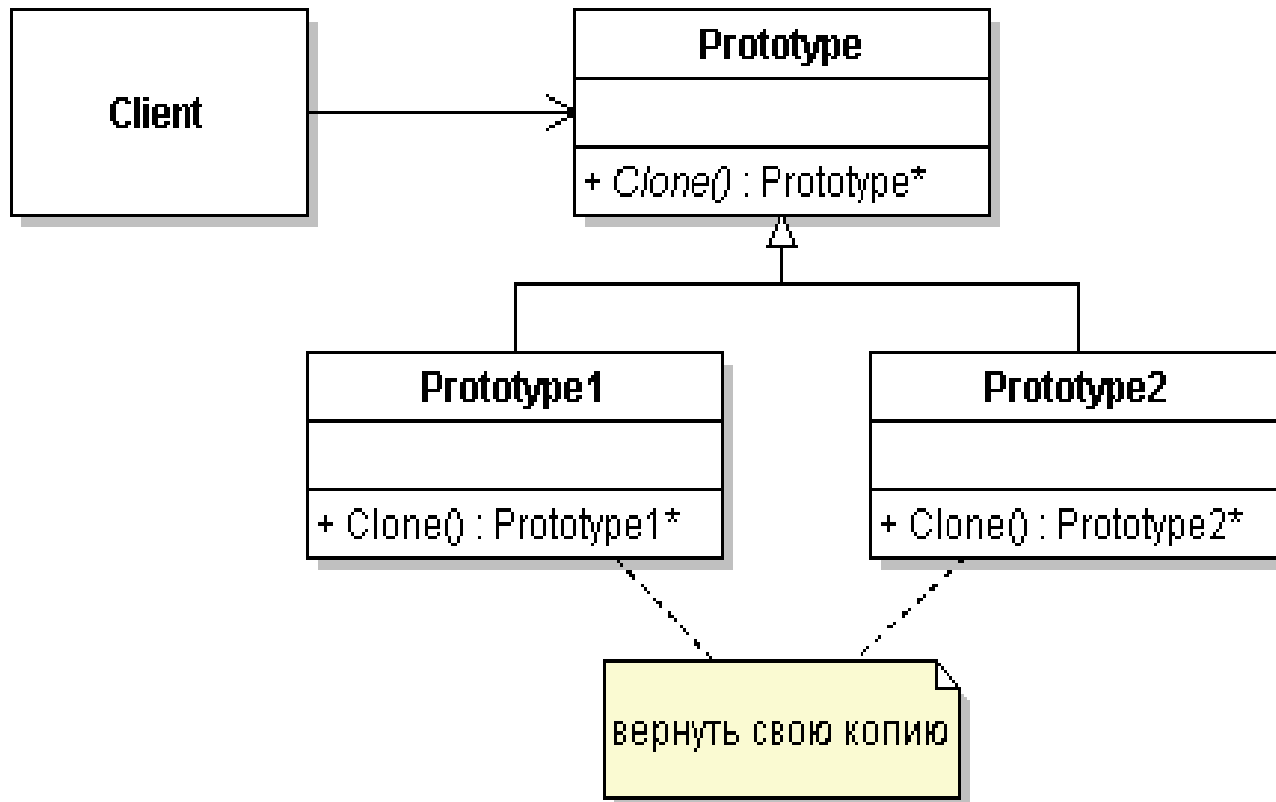
### **Результаты**

1. Паттерн прототип позволяет добавлять и удалять продукты во время выполнения.
2. Можно определять новые виды объектов, инстанцируя уже существующие классы, изменяя значений полей и регистрируя экземпляры как прототипы клиентских объектов. В результате число необходимых системе классов может быть уменьшено.
3. Многие приложения строят объекты из более мелких составляющих. В этом случае можно определять новые виды объектов путем изменения структуры.
4. Можно избавиться от иерархии класса Creator в паттерне фабричный метод, если не запрашивать фабричный метод создать новый объект, а клонировать прототип.
5. Возможно динамическое конфигурирование приложения классами, если исполняющая среда будет автоматически создавать экземпляр каждого класса и регистрировать экземпляр в диспетчере прототипов. Затем приложение может

запросить у диспетчера прототипов экземпляры вновь загруженных классов, которые изначально не были связаны с программой.

6. Каждый подкласс иерархии Prototype и их составляющие должны реализовывать операцию Clone, но ее добавление может оказаться затруднительным, если используются уже существующие классы.

## Структура



## Реализация

```
// Прототип объявляет интерфейс для клонирования самого себя
class Prototype {
public:
    virtual Prototype *Clone()=0;
};
// Конкретные прототипы реализуют операцию клонирования себя
class Prototype1 : public Prototype {
    Prototype1(const Prototype1 &)
public:
    Prototype1 *Clone() { return new Prototype1(*this); }
};
class Prototype2 : public Prototype {
    Prototype2(const Prototype2 &)
public:
    Prototype2 *Clone() { return new Prototype2(*this); }
};
// Образец
Prototype *proto;
// Создание объектов в клиенте
Prototype *p=proto->Clone();
```

# Строитель

*Builder*

## Назначение

Отделяет конструирование сложного объекта от его представления, так, что в результате одного и того же процесса конструирования могут получаться разные представления.

## Результаты

1. В отличие от порождающих паттернов, которые сразу конструируют весь объект целиком, строитель делает это шаг за шагом под управлением распорядителя. И лишь когда продукт завершен, распорядитель забирает его у строителя. Это позволяет обеспечить более тонкий контроль над процессом конструирования, а значит, и над внутренней структурой готового продукта.
2. Поскольку продукт конструируется через абстрактный интерфейс, то для изменения внутреннего представления достаточно всего лишь определить новый вид строителя.
3. Паттерн изолирует код, реализующий конструирование и представление. Клиентам ничего не надо знать о классах, определяющих внутреннюю структуру продукта, так как они отсутствуют в интерфейсе строителя. Разные распорядители могут строить разные варианты продукта из одних и тех же частей.

## Реализация

```
// Строитель задает абстрактный интерфейс для создания частей
// объекта Product
class Builder {
public:
    virtual void BuildPartA()=0;
    virtual void BuildPartB()=0;
};

// Распорядитель конструирует объект, пользуясь интерфейсом
// Builder
class Director {
    Builder *builder;
public:
    Director(Builder *b):builder(b) {}
    void BuildProduct() {
        builder->BuildPartA();
        builder->BuildPartB();
        ...
    }
};

// Продукт представляет собой сложный конструируемый объект
class Product {...};

// Конкретный строитель предоставляет интерфейс для доступа к
// продукту
```

```
class ConcreteBuilder : public Builder {  
    Product *product;  
public:  
    ConcreteBuilder():product(new Product()) {}  
    void BuildPartA();  
    void BuildPartB();  
    Product *GetProduct() { return product; }  
};
```

// Клиент создает конкретного строителя и распорядителя для  
создания продукта

```
ConcreteBuilder builder;  
Director director(&builder);  
director.BuildProduct();  
Product *product=builder.GetResult()
```

## **Фабричный метод**

### *Factory Method*

#### **Назначение**

Определяет интерфейс для создания объекта, но оставляет подклассам решение о том, какой класс инстанцировать.

#### **Результаты**

1. Фабричные методы избавляют проектировщика от необходимости встраивать в код зависящие от приложения классы. Код имеет дело только с интерфейсом класса Product, поэтому он может работать с любыми классами конкретных продуктов.
2. Паттерн может использоваться для соединения параллельных иерархий, например, основных и вспомогательных продуктов типа итераторов. С помощью фабричного метода локализуется знание о том, какие классы должны работать совместно.

## Реализация

```
// Продукт определяет интерфейс объектов, создаваемых фабричным
методом
class Product {};
// Создатель объявляет фабричный метод, возвращающий объект типа
Product
// и может вызывать этот метод для создания объектов
class Creator {
public:
    virtual Product *CreateProduct()=0;
    void Operation() {
        Product *product=CreateProduct();
        ...
    }
};
// Конкретный продукт
class Product1 : public Product {};
// Конкретный создатель замещает фабричный метод
class Creator1 : public Creator {
public:
    Product1 *CreateProduct() { return new Product1(); }
};
```



# Структурирующие паттерны

## Адаптер

### *Adapter*

#### **Назначение**

Преобразует интерфейс одного класса в интерфейс другого, который ожидают клиенты. Адаптер обеспечивает совместную работу классов с несовместимыми интерфейсами, которая без него была бы невозможна.

#### **Результаты**

1. Используя различные сменные адаптеры можно использовать один и тот же класс в разных системах, независимо от требуемого интерфейсом.
2. Адаптер объектов уже не обладает интерфейсом Adaptee, так что его нельзя использовать там, где Adaptee был применим. В тех случаях, когда клиенты должны видеть объект по-разному, применяются двусторонние адаптеры.

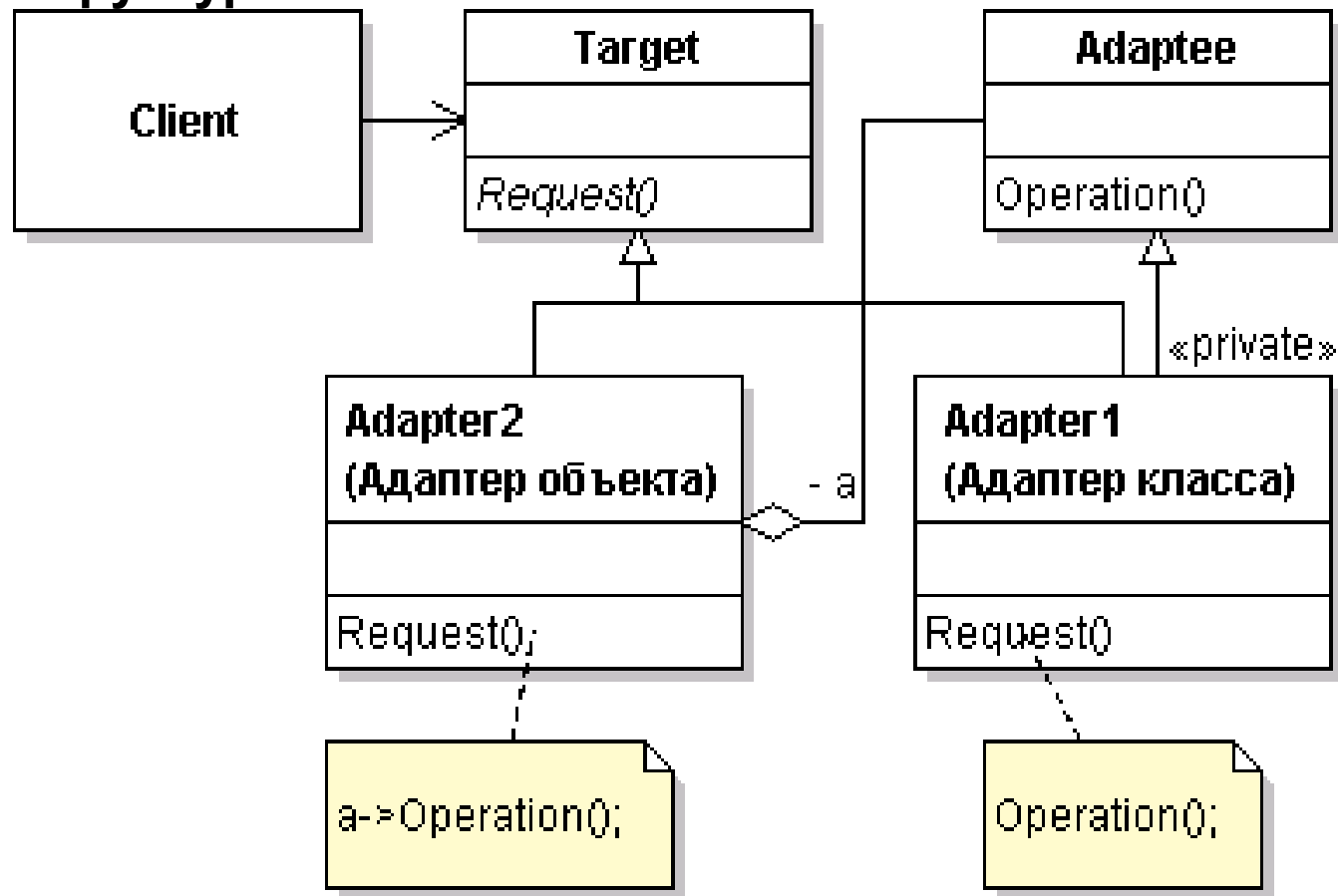
#### Адаптер класса:

1. Можно легко заместить некоторые виртуальные методы адаптируемого класса Adaptee, так как Adapter есть не что иное, как подкласс Adaptee.
2. Адаптер класса вводит только один новый объект. Чтобы добраться до адаптируемого класса, не нужно никакого дополнительного обращения по указателю.
3. Паттерн не применим, если требуется адаптирование и класса Adaptee и его подклассов.

Адаптер объектов:

1. Позволяет адаптеру работать и с самим классом Adaptee и его подклассами.
2. Для замещения виртуальных методов класса Adaptee требуется породить от Adaptee подкласс и инициализировать указатель адресом объекта этого подкласса.

## Структура



## Реализация

// Адаптируемый класс

```
class Adaptee {
```

```
public:
```

```
    void Operation();
```

```
};
```

// Требуемый интерфейс

```
class Target {
```

```
public:
```

```
    virtual void Request()=0;
```

```
};
```

// Адаптер класса

```
class Adapter1: public Target, private Adaptee {
```

```
public:
```

```
    void Request() { Operation(); }
```

```
};
```

// Адаптер объектов

```
class Adapter2: public Target {
```

```
    Adaptee *a;
```

```
public:
```

```
    void Request() { a->Operation(); }
```

```
};
```

# Декоратор

*Decorator*

## Назначение

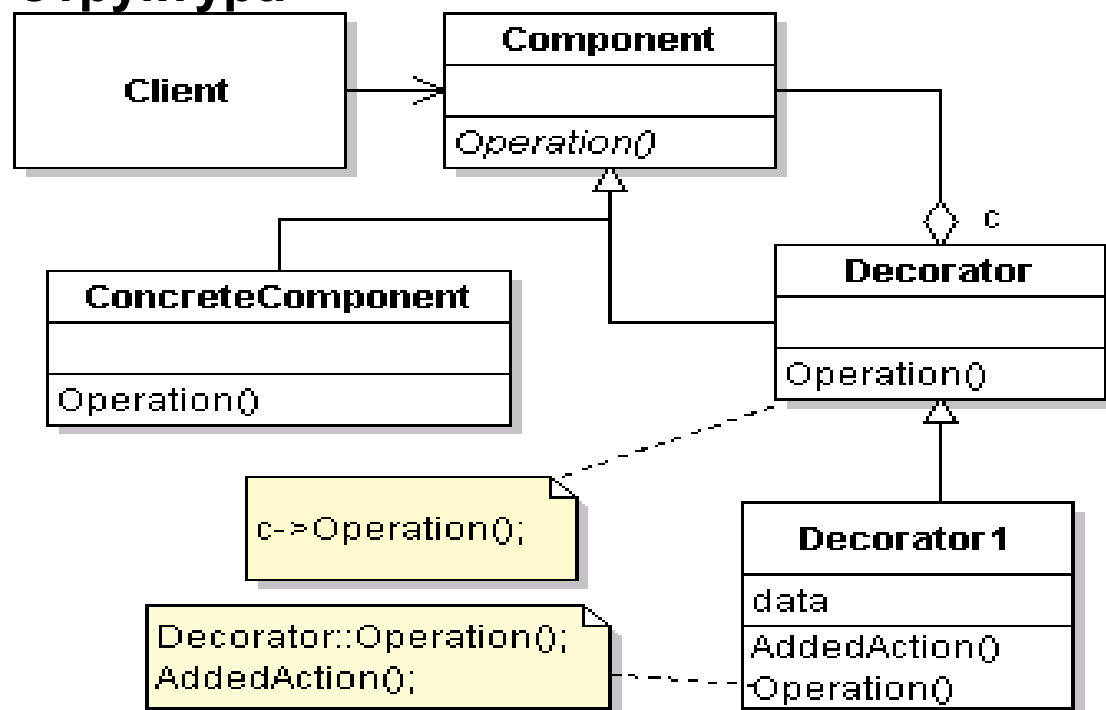
Динамически добавляет объекту новые обязанности. Является гибкой альтернативой порождению подклассов с целью расширения функциональности.

## Результаты

1. Паттерн декоратор позволяет более гибко добавлять объекту новые обязанности, чем было бы возможно в случае простого наследования. Декоратор может добавлять и удалять обязанности во время выполнения программы. Кроме того, применение нескольких декораторов к одному компоненту позволяет произвольным образом сочетать обязанности или добавить одно и то же свойство дважды.
2. Декоратор позволяет добавлять новые обязанности по мере необходимости и избежать перегруженных функциями классов на верхних уровнях иерархии. Нетрудно также определять новые виды декораторов независимо от классов, которые они расширяют, даже если первоначально такие расширения не планировались.
3. Декоратор действует как прозрачное обрамление. Но декорированный компонент все же не идентичен исходному.
4. При использовании в проекте паттерна декоратор нередко получается система, составленная из большого числа мелких объектов, которые похожи друг на друга и

различаются только способом взаимосвязи, а не классом и не значениями своих внутренних переменных. Хотя проектировщик, разбирающийся в устройстве такой системы, может легко настроить ее, но изучать и отлаживать ее очень тяжело.

## Структура



## Реализация

// Компонент определяет интерфейс для объектов

```
class Component {  
public:  
    virtual void Operation()=0;  
};
```

```
// Конкретный компонент определяет класс объектов,  
// на который возлагаются дополнительные обязанности  
class ConcreteComponent: public Component {  
public:  
    void Operation();  
};  
// Декоратор хранит указатель на объект Component  
// и определяет интерфейс, соответствующий интерфейсу Component  
class Decorator: public Component {  
protected:  
    Component *c;  
public:  
    void Operation() { c->Operation(); }  
};  
// Конкретный декоратор добавляет дополнительные обязанности  
компоненту  
class Decorator1: public Decorator {  
    int data;  
public:  
    void AddedAction();  
    void Operation() { Decorator::Operation(); AddedAction(); }  
};
```

## **Заместитель**

*Proxu*

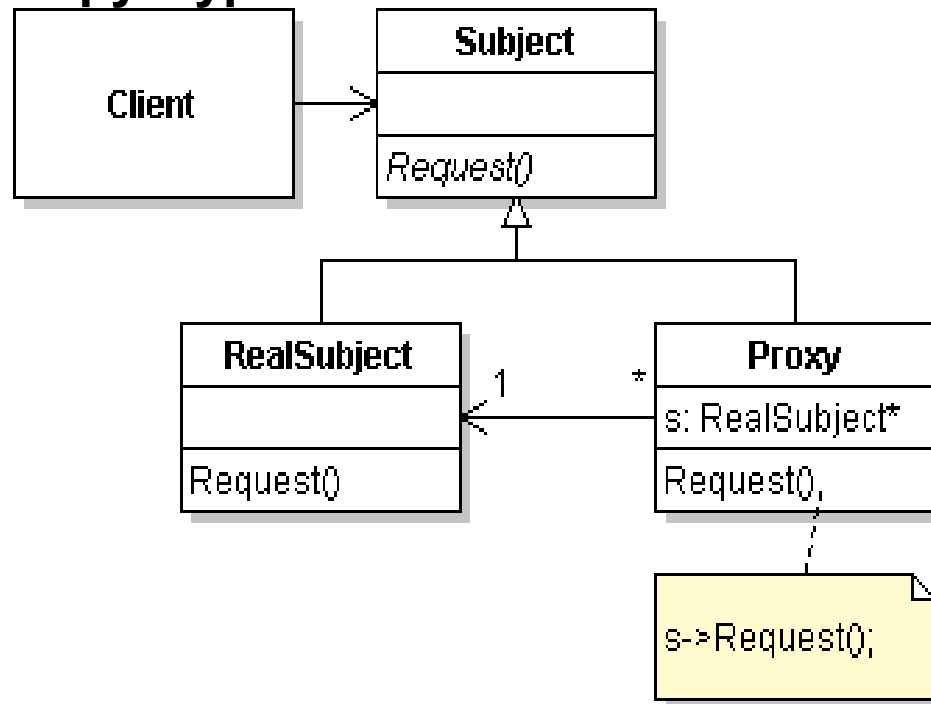
## **Назначение**

Является суррогатом другого объекта и контролирует доступ к нему.

## **Результаты**

1. Удаленный заместитель может скрыть тот факт, что объект находится в другом адресном пространстве.
2. Виртуальный заместитель может выполнять оптимизацию, например, создание объекта по требованию.
3. Защищающий заместитель и "умная" ссылка позволяют решать дополнительные задачи при доступе к объекту, например, контроль прав доступа или автоматическое уничтожение объекта при уменьшении ссылок на объект до 0.

## Структура



## Реализация

```
// Субъект определяет общий для RealSubject и Proxy интерфейс,  
// так что класс Proxy можно использовать везде, где ожидается  
RealSubject  
class Subject {  
public:  
    virtual void Request ()=0;  
};
```



```
// Реальный субъект
class RealSubject : public Subject {
public:
    void Request();
};

// Заместитель
class Proxy: public Subject {
    RealSubject *s;
public:
    void Request() { s->Request(); }
};
```

Умные указатели за счет перегрузки операций обеспечивают интерфейс как у обычного указателя и автоматическое уничтожение созданных объектов.

```
// Умный указатель
class SmartPtr {
    Data *ptr;
public:
    SmartPtr() ptr(0) {}
    SmartPtr(const SmartPtr& p);
    SmartPtr(Data *data);
    SmartPtr& operator=(const SmartPtr&p);
    ~SmartPtr();
};
```

```

Data *operator->() { return ptr; }
Data &operator*() { return *ptr; }
friend bool operator==(const SmartPtr &a, const SmartPtr &b)
{ return a.ptr==b.ptr; }
};
// Данные
class Data {
    int count; // счетчик указателей
    int data;  // данные
    friend class SmartPtr;
public:
    Data():count(0),data(1){}
    SmartPtr operator&() { return SmartPtr(*this); }
    int getData() const { return data; }
};
SmartPtr::SmartPtr(const SmartPtr& p)
{ if (ptr=p.ptr)
    ++(ptr->count);
}
SmartPtr::SmartPtr(Data *data)
{ ptr=data;
  ++(ptr->count);
}

```

```
SmartPtr& SmartPtr::operator=(const SmartPtr&p)
{ SmartPtr t(p);
  std::swap(t.ptr, ptr);
}
SmartPtr::~SmartPtr()
{ if(ptr && --(ptr->count)==0)
  delete ptr;
}
bool operator!=(const SmartPtr &a, const SmartPtr &b)
{ return !(a==b);
}
// Использование
SmartPtr p=new Data;
cout<<p->getData()<<"\n";
```

## **Компоновщик**

*Composite*

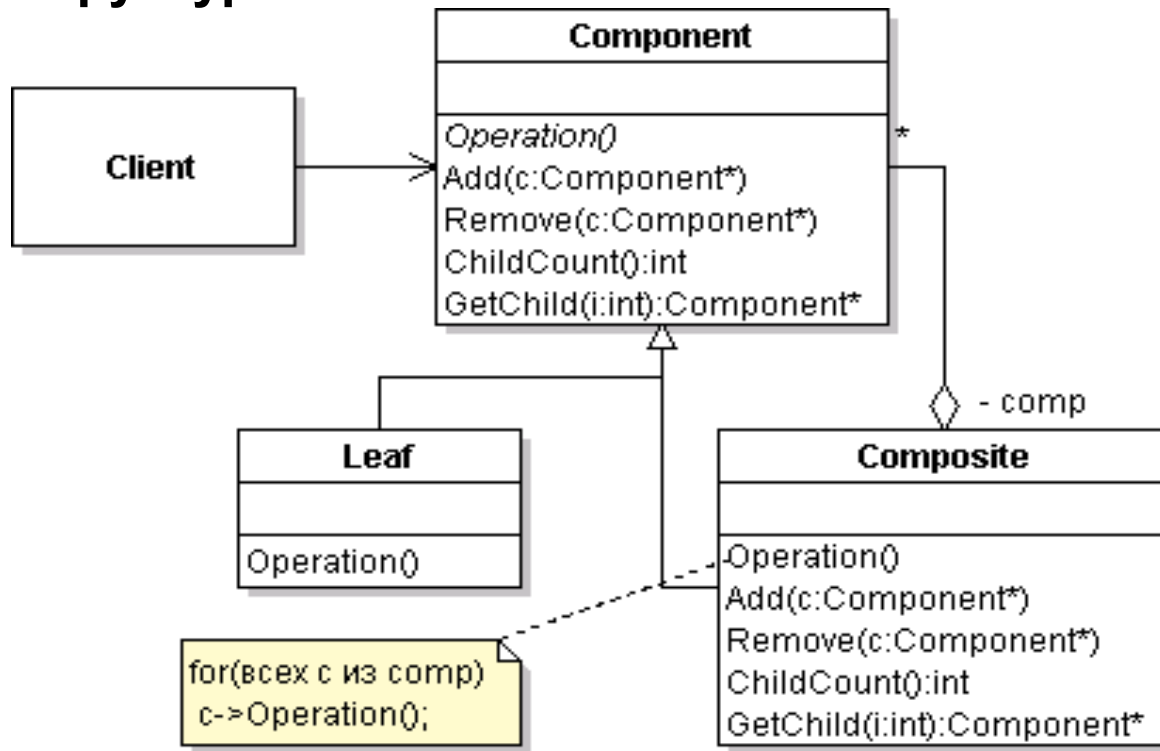
### **Назначение**

Компонуется объекты в древовидные структуры для представления иерархии часть-целое. Позволяет клиентам единообразно трактовать индивидуальные и составные объекты.

### **Результаты**

1. Паттерн компоновщик позволяет создавать иерархии классов, состоящие из примитивных и составных объектов. Из примитивных объектов составляются более сложные, которые, в свою очередь, участвуют в более сложных композициях и так далее.
2. Клиенты могут единообразно работать с индивидуальными объектами и с составными структурами. Обычно клиенту неизвестно, взаимодействует ли он с простым или составным объектом. Использование паттерна упрощает код клиента, поскольку нет необходимости писать функции, ветвящиеся в зависимости от того, с объектом какого класса они работают;
3. Паттерн компоновщик облегчает добавление новых видов компонентов. Существующие классы и клиенты будут автоматически работать с новыми подклассами.
4. Иногда желательно, чтобы составной объект мог включать только определенные виды компонентов. Проверку этих ограничений нужно проводить во время выполнения, так как паттерн не позволяет проверять ограничения при компиляции.

# Структура



## Реализация

// Компонент определяет интерфейс для всех объектов в иерархии,  
// объявляет интерфейс для доступа к потомкам и  
// определяет реализацию операций по умолчанию, общую для всех классов

```
class Component {  
public:  
    virtual void Operation()=0;  
    virtual void Add(Component *) { throw Error(); }  
    virtual void Remove(Component *) {}
```

```

    virtual int ChildCount() { return 0; }
    virtual Component *GetChild(int i) { throw Error(); }
};
// Простой компонент
class Leaf : public Component {
public:
    void Operation();
};
// Составной компонент
class Composite : public Component {
    vector<Component *> comp;
public:
    void Operation()
    { for(int i=0;i<comp.size();++i)
        comp[i]->Operation();

        ...
    }
    void Add(Component *c) { comp.push_back(c); }
    void Remove(Component *c)
{ comp.erase(find(comp.begin(), comp.end(), c)); }
    int ChildCount() { return comp.size(); }
    Component *GetChild(int i) { return comp[i]; }
};

```

# Мост

## Bridge

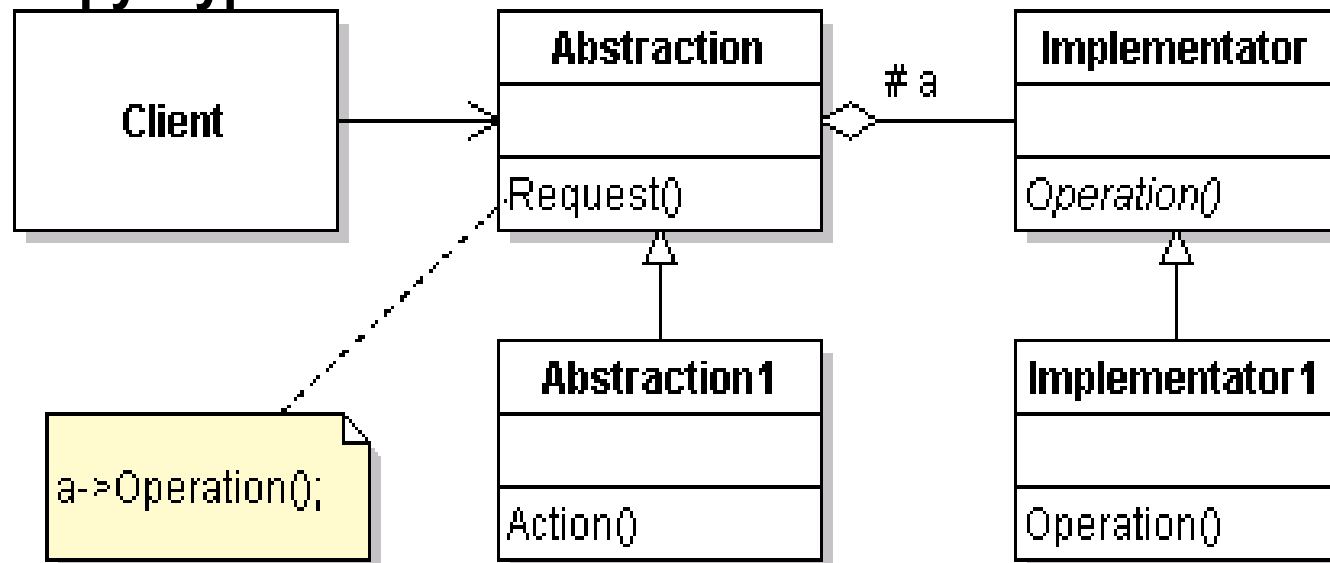
### Назначение

Отделить абстракцию от ее реализации так, чтобы то и другое можно было изменять независимо.

### Результаты

1. Реализация больше не имеет постоянной привязки к интерфейсу. Реализацию абстракции можно конфигурировать во время выполнения. Объект может даже динамически изменять свою реализацию.
2. Можно расширять независимо иерархии классов Abstraction и Implementor.
3. Можно скрыть детали реализации от клиентов, используя указатель типа `void*`.

### Структура



## Реализация

// Инструмент определяет интерфейс, содержащий примитивные операции

```
class Implementator {
```

```
public:
```

```
    virtual void Operation()=0;
```

```
};
```

// Конкретный инструмент определяет реализацию примитивных операций

```
class Implementator1 : public Implementator {
```

```
public:
```

```
    void Operation();
```

```
};
```

// Абстракция определяет интерфейс и реализацию по умолчанию для высокоуровневых операций

```
class Abstraction {
```

```
protected:
```

```
    Implementator *a;
```

```
public:
```

```
    virtual void Request() { a->Operation(); }
```

```
};
```

// Уточненная абстракция расширяет интерфейс



```
class Abstraction1 : public Abstraction {  
public:  
    void Action();  
};
```

## **Приспособленец**

*Flyweight*

## **Назначение**

Использует разделение для эффективной поддержки множества мелких объектов.

## **Результаты**

При использовании приспособленцев не исключены затраты на передачу, поиск или вычисление внутреннего состояния. Однако такие расходы с лихвой компенсируются экономией памяти за счет разделения объектов-приспособленцев, которая получается из-за:

- 1) уменьшение общего числа экземпляров;
- 2) сокращение объема памяти, необходимого для хранения внутреннего состояния;
- 3) вычисление, а не хранение внешнего состояния.

Чем выше степень разделения приспособленцев, тем существеннее экономия.

## Реализация

// Приспособленец, объявляет интерфейс, с помощью которого можно  
// получать внешнее состояние или как-то воздействовать на него

```
class Flyweight {
```

```
public:
```

```
    virtual char geta() const=0; // символ
```

```
    virtual int getb() const=0; // размер
```

```
};
```

// Фабрика приспособленцев создает объекты-приспособленцы и  
управляет ими

```
class FlyweightFactory {
```

```
    vector<char> a; // уникальное значение для каждого объекта
```

```
    map<int, int> b; // значение одинаково для нескольких
```

последовательных объектов

```
    typedef map<int, Flyweight *> fwmap;
```

```
    fwmap fw;
```

```
    char geta(int n) { return a[n]; }
```

```
    int getb(int n) { return (--b.upper_bound(n))->second; }
```

```
    void release(int n) { fw.erase(n); }
```

```
    friend class ConcreteFlyweight;
```

```
    friend class UnsharedFlyweight;
```

```
public:
```

```
    Flyweight *getFlyweight(int n);
```

```

}
// Конкретный приспособленец, хранит состояние внутри
class ConcreteFlyweight {
    char a;
    int b;
    FlyweightFactory *f;
    int n;
    ConcreteFlyweight(FlyweightFactory *f, int n):f(f),n(n)
    { a=f->geta(n); b=f->getb(n); }
    friend class FlyweightFactory;
public:
    ~ConcreteFlyweight() { f->release(n); }
    char geta() const { return a; }
    int getb() const { return b; }
};
// Неразделяемый конкретный приспособленец
class UnsharedFlyweight {
    FlyweightFactory *f;
    int n;
    UnsharedFlyweight(FlyweightFactory *f, int n):f(f),n(n) {}
    friend class FlyweightFactory;
public:
    ~UnsharedFlyweight() { f->release(n); }

```

```

    char geta() const { return f->geta(n); }
    int getb() const { return f->getb(n); }
};
Flyweight *FlyweightFactory::getFlyweight(int n);
{ fwmap::iterator f=fw.find(n);
  if(f!=fw.end())
    return f->second;
  return fw[n]=new ConcreteFlyweight(this,n);
}

```

Этот вариант можно использовать, когда не будет удаления и добавления элементов в набор. В случае частых добавлений и удалений для повышения эффективности для хранения состояний элементов набора вместо vector и map нужно использовать специальные структуры данных (дерево интервалов,  $\sqrt{}$ -декомпозицию).

# Фасад

*Facade*

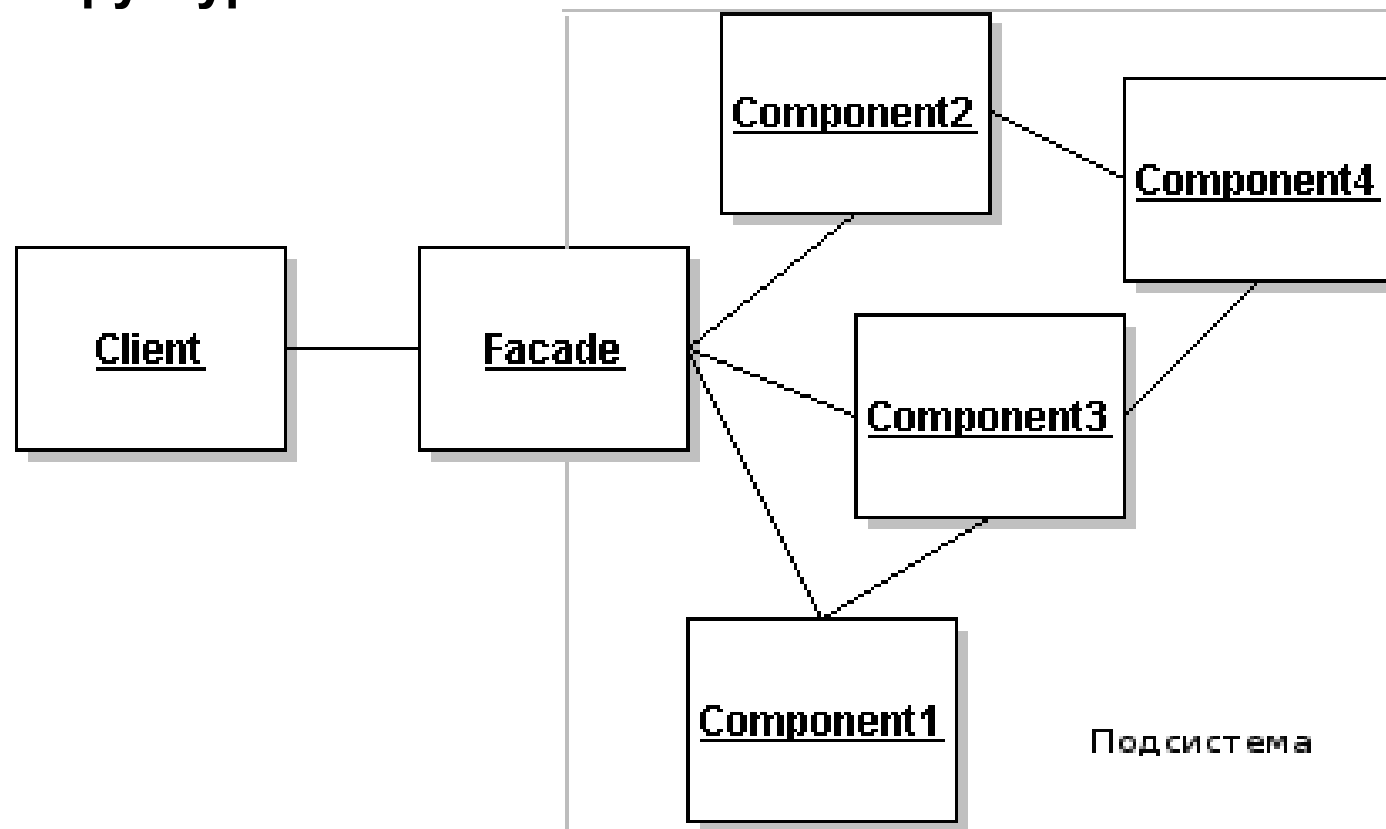
## Назначение

Предоставляет унифицированный интерфейс вместо набора интерфейсов некоторой подсистемы. Фасад определяет интерфейс более высокого уровня, который упрощает использование подсистемы.

## Результаты

1. Паттерн фасад изолирует клиентов от компонентов подсистемы, уменьшая тем самым число объектов, с которыми клиентам приходится иметь дело, и упрощая работу с подсистемой.
2. Зачастую компоненты подсистемы сильно связаны. Ослабление связей между подсистемой и ее клиентами позволяет видоизменять компоненты подсистемы, не затрагивая при этом клиентов. Также сокращение числа зависимостей за счет фасадов может уменьшить количество нуждающихся в повторной компиляции файлов после небольшой модификации какой-нибудь важной подсистемы. Фасад может упростить процесс переноса системы на другие платформы, поскольку уменьшается вероятность того, что в результате изменения одной подсистемы понадобится изменять и все остальные.
3. Фасад не препятствует приложениям напрямую обращаться к классам подсистемы, если это необходимо. Таким образом, у вас есть выбор между простотой и общностью.

## Структура



## Реализация

```
// Компоненты подсистемы реализуют функциональность подсистемы,  
// выполняют работу, порученную объектом Facade  
class Component1 {  
public:  
    void Operation1();  
} component1;  
class Component2 {  
public:  
    void Operation2();  
} component2;  
//Фасад делегирует запросы клиентов подходящим объектам внутри  
подсистемы  
class Facade {  
public:  
    void Operation1() { component1.Operation1(); }  
    void Operation2() { component2.Operation2(); }  
} facade;  
// Использование  
facade.Operation1();  
facade.Operation2();
```

# Паттерны поведения

## Интерпретатор

*Interpreter*

### Назначение

Для заданного языка определяет представление его грамматики, а также интерпретатор предложений этого языка.

### Результаты

1. Грамматику легко изменять и расширять.
2. Реализации классов, описывающих узлы абстрактного синтаксического дерева, достаточно тривиальна, их может автоматически создавать генератор синтаксических анализаторов.
3. Сложные грамматики трудно сопровождать, так как определяется по меньшей мере один класс для каждого правила грамматики.
4. Паттерн интерпретатор позволяет легко изменить способ вычисления выражений. При частом добавлении новых способов интерпретации выражений можно использовать паттерн посетитель.



## Реализация

// Контекст

```
class Context {
```

```
public:
```

```
    int curr(); // текущий символ
```

```
    void next(); // переход к следующему символу
```

```
    int getpos(); // запомнить позицию
```

```
    void setpos(int); // восстановить позицию
```

```
};
```

// Исключительная ситуация для ошибки разбора

```
class SyntaxError {};
```

// Абстрактное выражение

```
class Expression {
```

```
public:
```

```
    virtual void Interpret(Context&)=0;
```

```
};
```

// Терминальное выражение - один символ

```
class Terminal : public Expression {
```

```
    int ch;
```

```
public:
```

```
    Terminal(int ch) ch(ch) {}
```

```
    void Interpret(Context &c)
```

```
    { if(c.curr()!=ch) throw SyntaxError();
```

```

        c.next();
    }
};
// Нетерминальное выражение для правила грамматики R ::= R1 R2 ...
Rn
class Nonterminal : public Expression {
    vector <Expression *> rule;
public:
    void Add(Expression *e) { rule.push_back(e); }
    void Interpret(Context &c)
    { for(size_t i=0; i<rule.size(); ++i)
        rule[i]->Interpret(c);
    }
};
// Нетерминальное выражение для правила грамматики R ::= R1 | R2
| ... | Rn
class Select : public Expression {
    vector <Expression *> rule;
public:
    void Add(Expression *e) { rule.push_back(e); }
    void Interpret(Context &c)
    { int p=c.getpos();
        for(size_t i=0; i<rule.size(); ++i)

```

```
{
    try {
        rule[i] -> Interpret(c);
        return;
    }
    catch (SyntaxError &e)
    { c.setpos(p); }
}
throw SyntaxError();
}
```

```
};
```

# Итератор

*Iterator*

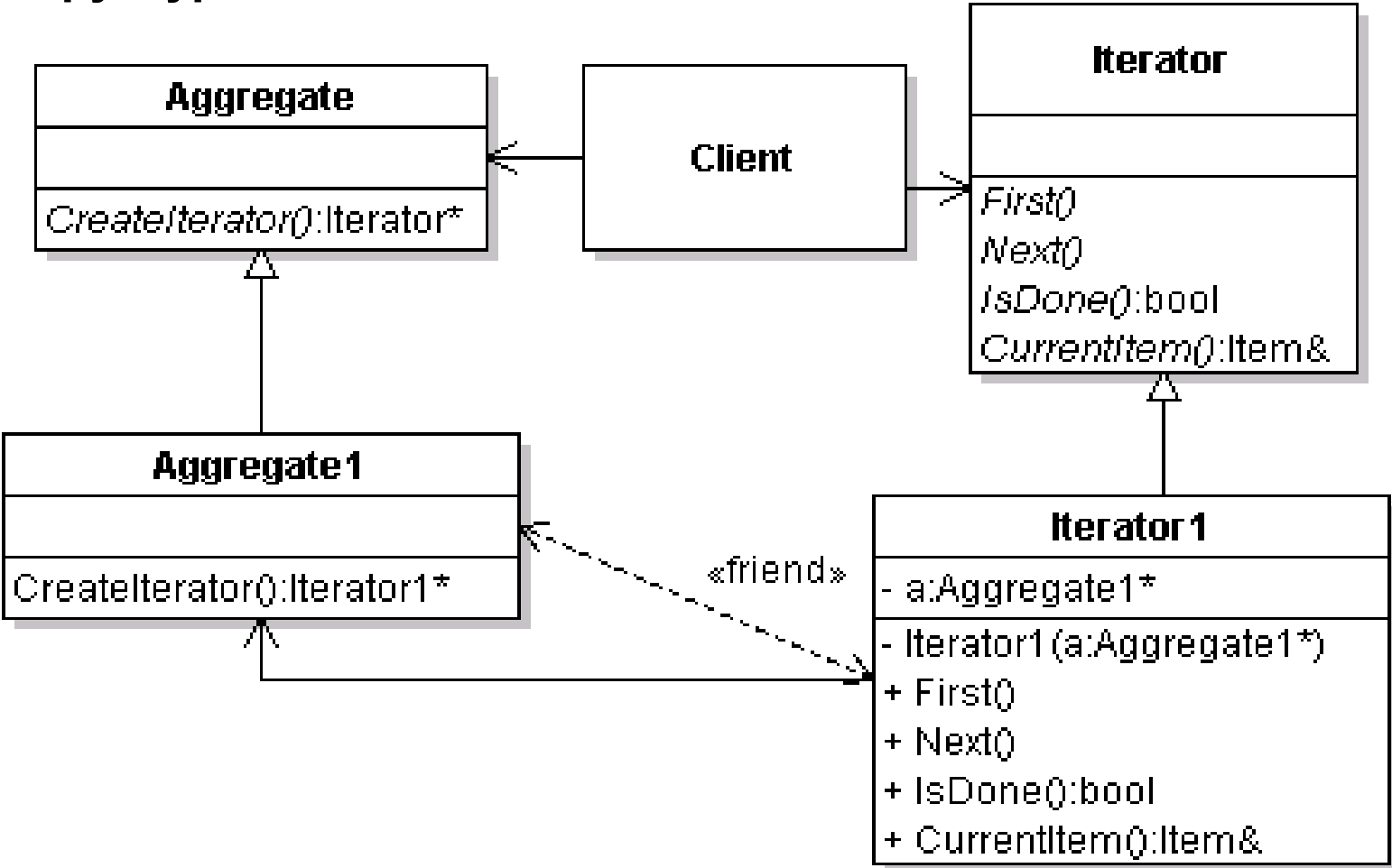
## Назначение

Предоставляет способ последовательного доступа ко всем элементам составного объекта, не раскрывая его внутреннего представления.

## Результаты

1. Сложные составные объекты можно обходить по-разному. Для изменения алгоритма обхода нужно определить подкласс класса `Iterator` и заменить один экземпляр итератора другим.
2. Итераторы упрощают интерфейс класса `Aggregate`.
3. Одновременно для данного агрегата может быть активно несколько обходов.
4. Модификация составного объекта в то время, как совершается его обход, может оказаться опасной. Для создания устойчивого к модификациям итератора необходимо изменять состояние всех созданных итераторов при вставке или удалении.
5. Если итерацией управляет клиент, то итератор называется внешним, в противном случае – внутренним. В случае внутреннего итератора клиент передает составному объекту некоторую операцию, а объект сам применяет эту операцию к каждому элементу. Внешние итераторы обладают большей гибкостью, чем внутренние. Например, сравнить две коллекции на равенство с помощью внешнего итератора очень легко, а с помощью внутреннего – практически невозможно.

Структура



## Реализация

// Составной объект

```
class Aggregate {
```

```
public:
```

```
    virtual Iterator *CreateIterator()=0;
```

```
};
```

// Элементы составного объекта

```
class Item;
```

// Конкретный объект реализует интерфейс создания итератора

```
class Aggregate1 : public Aggregate {
```

```
    friend class Iterator1;
```

```
    Item *items;
```

```
    int size;
```

```
public:
```

```
    Iterator1 *CreateIterator() { return new Iterator1(this); }
```

```
};
```

// Итератор определяет интерфейс для доступа и обхода элементов

```
class Iterator {
```

```
public:
```

```
    virtual void First()=0;
```

```
    virtual void Next()=0;
```

```
    virtual bool IsDone()=0;
```

```
    virtual Item &CurrentItem()=0;
```

```
};
// Конкретный итератор реализует интерфейс класса Iterator
class Iterator1 {
    friend class Agggregate1;
    Agggregate1 *a;
    int i;
    Iterator1(Agggregate1 *);
public:
    void First() { i=0; }
    void Next() { ++i; }
    bool IsDone() { return i>=a->size; }
    Item &CurrentItem() { return a->items[i]; }
};
// Использование
Agggregate *a;
Iterator *it=a->CreateIterator();
for(it->First(); !it->IsDone(); it->Next())
    ... действия с it->CurrentItem() ...
```

В STL для работы с итераторами используется перегрузка операций: для получения текущего элемента – операция разъадресации (\*), для перехода на следующий элемент – операция инкремента (++), для проверки завершения – сравнение с со специальным итератором контейнера `end()`.

## Внутренний итератор (см. также паттерн посетитель)

```
// Элементы составного объекта
typedef int Item;
// Внутренний итератор
class Iterator {
public:
    virtual void Operation(Item&)=0;
};
// Составной объект
class Aggregate {
public:
    virtual void ApplyIterator(Iterator &)=0;
};
// Конкретный объект реализует интерфейс создания итератора
class Aggreate1 : public Aggregate {
    Item *items;
    int size;
public:
    void ApplyIterator(Iterator &)
    { for(int i=0;i<size;++i)
        Iterator.Operation(items[i]);
    }
};
```



```
// Использование
class SumIterator : public Iterator {
    int sum;
public:
    SumIterator():sum(0) {}
    void Operation(Item &it) { sum+=it; }
    int getSum() const { return sum; }
} sumIt;

...
Aggregate *a;
a->ApplyIterator(sumIt);
cout<<sumIt->getSum();
```

## **Команда**

*Command*

### **Назначение**

Инкапсулирует запрос как объект, позволяя тем самым задавать параметры клиентов для обработки соответствующих запросов, ставить запросы в очередь или протоколировать их, а также поддерживать отмену операций.

### **Результаты**

1. Команда разрывает связь между объектом, инициирующим операцию, и объектом, имеющим информацию о том, как ее выполнить.
2. Команды – это самые настоящие объекты. Допускается манипулировать ими и расширять их точно так же, как в случае с любыми другими объектами.
3. Из простых команд можно собирать составные, например, макрокоманды, при реализации которых можно использовать паттерн компоновщик.
4. Добавлять новые команды легко, поскольку никакие существующие классы изменять не нужно.

## Реализация

// Команда объявляет интерфейс для выполнения операции

```
class Command {
```

```
public:
```

```
    virtual void Execute()=0;
```

```
};
```

// Получатель располагает информацией о способах выполнения операций

// Например, получателем может быть Окно, а операцией – закрытие окна

```
class Receiver {
```

```
public:
```

```
    void Action();
```

```
};
```

// Конкретная команда определяет связь между объектом-получателем и действием

```
class Command1 : public Command {
```

```
    Receiver *r;
```

```
public:
```

```
    Command1(Receiver *r) r(r) {}
```

```
    void Execute() { r->Action(); }
```

```
};
```

// Инициатор обращается к команде для выполнения запроса

// Например, инициатором может Кнопка, при нажатии на которую  
выполнится заданная команда

```
class Invoker {  
    Command *cmd;  
public:  
    void AddCommand(Command *c) { cmd=c; }  
    void Action() { cmd->execute(); }  
};  
// Использование  
Invoker inv;  
Receiver rec;  
inv.AddCommand(new Command1(&rec));  
...  
inv.Action();
```

## Наблюдатель

*Observer*

### Назначение

Определяет зависимость типа один ко многим между объектами таким образом, что при изменении состояния одного объекта все зависящие от него оповещаются об этом и автоматически обновляются.

### Результаты

1. Субъект имеет информацию лишь о том, что у него есть ряд наблюдателей, каждый из которых подчиняется простому интерфейсу абстрактного класса *Observer*. Субъекту неизвестны конкретные классы наблюдателей. Таким образом, связи между субъектами и наблюдателями носят абстрактный характер и сведены к минимуму. Это дает возможность добавлять новые виды наблюдателей без модификации субъекта или других наблюдателей.
2. Для уведомления, посылаемого субъектом, не нужно задавать определенного получателя. Уведомление автоматически поступает всем подписавшимся на него объектам. Субъекту не нужна информация о количестве таких объектов, от него требуется всего лишь уведомить своих наблюдателей. Поэтому мы можем в любое время добавлять и удалять наблюдателей. Наблюдатель сам решает, обработать полученное уведомление или игнорировать его.
3. Поскольку наблюдатели не располагают информацией друг о друге, им

неизвестно и о том, во что обходится изменение субъекта. Простая операция над субъектом может вызвать каскад обновлений наблюдателей и зависящих от них объектов. Если протокол обновления не содержит никаких сведений о том, что именно изменилось в субъекте, то наблюдатели будут вынуждены проделать сложную работу для косвенного получения такой информации.

## Реализация

```
// Наблюдатель определяет интерфейс уведомления
class Observer {
public:
    virtual void Update() {}
};

// Субъект предоставляет интерфейс для работы с наблюдателями
class Subject {
    vector <Observer *> obs;
public:
    void Attach(Observer *o) { obs.push_back(o); }
    void Detach(Observer *o)
    { obs.erase(find(obs.begin(), obs.end(), o)); }
    void Notify()
    { for(size_t i=0; i<obs.size(); ++i) o->Update(); }
};
```

// Конкретный субъект хранит состояние, представляющее интерес для наблюдателей

```
class Subject1 : public Subject {  
    int state;  
public:  
    int GetState() const { return state; }  
    void SetState(int s)  
    { if(s!=state)  
        { state=s;  
          Notify();  
        }  
    }  
};
```

// Конкретный наблюдатель хранит данные, которые должны быть согласованы с данными субъекта

```
class Observer1 : public Observer {  
    Subject1 *s;  
    int state;  
public:  
    Observer(Subject1 *s):s(s) { s->Attach(this); }  
    ~Observer() { s->Detach(this); }  
    void Update() { state=s->GetState(); }  
};
```

# Посетитель

*Visitor*

## Назначение

Описывает операцию, выполняемую с каждым объектом из некоторой структуры. Паттерн посетитель позволяет определить новую операцию, не изменяя классы этих объектов.

## Результаты

1. С помощью посетителей легко добавлять операции, зависящие от компонентов сложных объектов. Для определения новой операции над структурой объектов достаточно просто ввести нового посетителя.
2. Родственное поведение не разносится по всем классам, присутствующим в структуре объектов, оно локализовано в посетителе. Не связанные друг с другом функции распределяются по отдельным подклассам класса Visitor. Это способствует упрощению как классов, определяющих элементы, так и алгоритмов, инкапсулированных в посетителях. Все относящиеся к алгоритму структуры данных можно скрыть в посетителе.
3. Паттерн посетитель усложняет добавление новых подклассов класса Element. Каждый новый конкретный элемент требует объявления новой абстрактной операции в классе Visitor, которую нужно реализовать в каждом из существующих его подклассов. Поэтому при решении вопроса о том, стоит ли использовать паттерн посетитель, нужно прежде всего посмотреть, что будет изменяться чаще:



алгоритм, применяемый к объектам структуры, или классы объектов, составляющих эту структуру. Паттерн посетитель следует применять в случае, если иерархия классов `Element` стабильна, но постоянно расширяется набор операций или модифицируются алгоритмы.

4. Итератор может посещать объекты структуры по мере ее обхода, вызывая операции объектов. Но итератор не способен работать со структурами, состоящими из объектов разных типов. У посетителя таких ограничений нет. Ему разрешено посещать объекты, не имеющие общего родительского класса.

5. Посетители могут аккумулировать информацию о состоянии при посещении объектов структуры. Если не использовать этот паттерн, состояние придется передавать в виде дополнительных аргументов операций, выполняющих обход, или хранить в глобальных переменных.

6. Применение посетителей подразумевает, что у элементов достаточно развитый интерфейс для того, чтобы посетители могли справиться со своей работой. Поэтому при использовании данного паттерна приходится предоставлять открытые операции для доступа к внутреннему состоянию элементов, что нарушает инкапсуляцию.

## Реализация

```
class Visitor;
// Элемент определяет операцию Accept для приема посетителя
class Element {
public:
    virtual void Accept(Visitor &)=0;
};
// Конкретные элементы реализуют операцию Accept
class ElementA : public Element {
public:
    void Accept(Visitor &v) { v.Visit(*this); }
    void OperationA();
};
class ElementB : public Element {
public:
    void Accept(Visitor &v) { v.Visit(*this); }
    void OperationB();
};
// Посетитель объявляет операцию Visit для каждого подкласса
Element
class Visitor {
public:
    virtual void Visit(ElementA &) {}
```

```

    virtual void Visit(ElementB &) {}
};
// Конкретный посетитель реализует две операции, объявленные в
классе Visitor
// Может содержать поля для накопления результатов в процессе
обхода структуры
class Visitor1 : public Visitor {
public:
    void Visit(ElementA &e) {... e->OperationA(); ... }
    void Visit(ElementB &e) {... e->OperationB(); ... }
};
// Сложный объект предоставляет посетителю интерфейс для
посещения своих элементов
class Object {
    Element *els;
    int n;
public:
    void Accept(Visitor &v)
    { for(int i=0;i<n;++i)
        els[i]->Accept(v);
    }
};

```

# Посредник

## *Mediator*

### Назначение

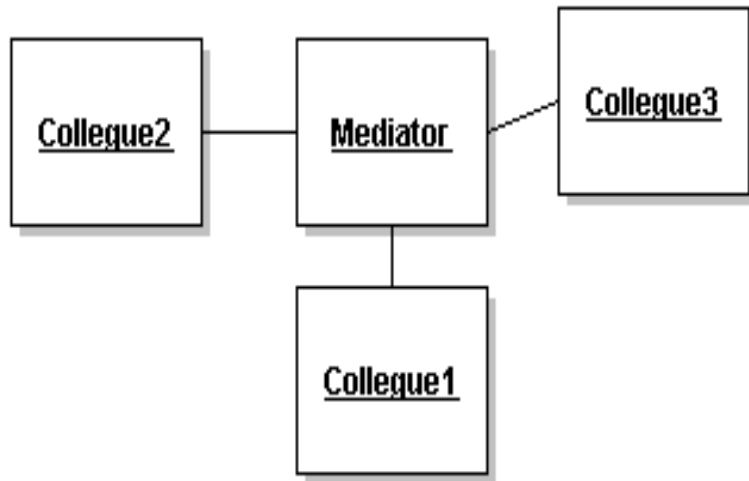
Определяет объект, инкапсулирующий способ взаимодействия множества объектов. Посредник обеспечивает слабую связанность системы, избавляя объекты от необходимости явно ссылаться друг на друга и позволяя тем самым независимо изменять взаимодействия между ними.

### Результаты

1. Посредник локализует поведение, которое в противном случае пришлось бы распределять между несколькими объектами. Для изменения поведения нужно породить подклассы только от класса посредника Mediator, классы коллег Colleague можно использовать повторно без каких бы то ни было изменений.
2. Посредник обеспечивает слабую связанность коллег. Можно изменять классы Colleague и Mediator независимо друг от друга.
3. Посредник заменяет способ взаимодействия «все со всеми» способом «один со всеми», то есть один посредник взаимодействует со всеми коллегами. Отношения вида «один ко многим» проще для понимания, сопровождения и расширения.
4. Выделение механизма посредничества в отдельную концепцию и инкапсуляция ее в одном объекте позволяет сосредоточиться именно на взаимодействии объектов, а не на их индивидуальном поведении. Это дает возможность прояснить имеющиеся в системе взаимодействия.

5. Паттерн посредник переносит сложность взаимодействия в класс-посредник. Поскольку посредник инкапсулирует протоколы, то он может быть сложнее отдельных коллег. В результате сам посредник становится монолитом, который трудно сопровождать.

## Структура



## Реализация

// Посредник реализует кооперативное поведение, координируя действия коллег

```
class Mediator {
```

```
public:
```

```
    void Operation1();
```

```
    void Operation2();
```

```
};
```

// Коллеги знают о своем посреднике и обмениваются информацией только через него

```
class Colleague {
```

```
protected:
```

```
    Mediator *m;
```

```
public:
```

```
    Colleague(Mediator *m) m(m) {}
```

```
};
```

```
class Colleague1 : public Colleague {
```

```
public:
```

```
    void Action1() { m->Operation1(); }
```

```
};
```

```
class Colleague2 : public Colleague {
```

```
public:
```

```
    void Action2() { m->Operation2(); }
```

```
};
```

## **Состояние**

### *State*

## **Назначение**

Позволяет объекту варьировать свое поведение в зависимости от внутреннего состояния. Извне создается впечатление, что изменился класс объекта.

## **Результаты**

1. Паттерн состояние помещает все поведение, ассоциированное с конкретным состоянием, в отдельный объект. Поскольку зависящий от состояния код целиком находится в одном из подклассов класса State, то добавлять новые состояния и переходы можно просто путем порождения новых подклассов. Вместо этого можно было бы использовать данные-члены для определения внутренних состояний, тогда операции объекта Context проверяли бы эти данные. Но в таком случае похожие условные операторы или операторы ветвления были бы разбросаны по всему коду класса Context. При этом добавление нового состояния потребовало бы изменения нескольких операций, что затруднило бы сопровождение. Паттерн состояние позволяет решить эту проблему, но одновременно порождает другую, поскольку поведение для различных состояний оказывается распределенным между несколькими подклассами State. Это увеличивает число классов. Конечно, один класс компактнее, но если состояний много, то такое распределение эффективнее, так как в противном случае пришлось бы иметь дело с громоздкими условными операторами. Наличие громоздких условных операторов в

нежелательно, равно как и наличие длинных процедур. Они слишком монолитны, вот почему модификация и расширение кода становится проблемой. Паттерн состояние предлагает более удачный способ структурирования зависящего от состояния кода. Логика, описывающая переходы между состояниями, больше не заключена в монолитные операторы `if` или `switch`, а распределена между подклассами `state`. При инкапсуляции каждого перехода и действия в класс состояние становится полноценным объектом. Это улучшает структуру кода и проясняет его назначение.

2. Если объект определяет свое текущее состояние исключительно в терминах внутренних данных, то переходы между состояниями не имеют явного представления; они проявляются лишь как присваивания некоторым переменным. Ввод отдельных объектов для различных состояний делает переходы более явными. Кроме того, объекты `State` могут защитить контекст `Context` от рассогласования внутренних переменных, поскольку переходы с точки зрения контекста – это атомарные действия. Для осуществления перехода надо изменить значение только одной переменной (объектной переменной `State` в классе `Context`), а не нескольких.

3. Если в объекте состояния `State` отсутствуют переменные экземпляра, то есть представляемое им состояние кодируется исключительно самим типом, то разные контексты могут разделять один и тот же объект `State`. Когда состояния разделяются таким образом, они являются, по сути дела, приспособленцами (см. описание паттерна приспособленец), у которых нет внутреннего состояния, а есть только поведение.



## Реализация

```
// Состояние определяет интерфейс для инкапсуляции поведения,  
// ассоциированного с конкретным состоянием  
class State {  
public:  
    virtual void Action() {}  
};  
// Каждый подкласс реализует поведение, ассоциированное с некоторым состоянием  
class State1 : public State {  
public:  
    void Action();  
};  
class State2 : public State {  
public:  
    void Action();  
};  
// Контекст определяет интерфейс, представляющий интерес для клиентов  
class Context {  
    State *state;  
public:  
    void Request() { state->Action(); }  
};
```

# Стратегия

## *Strategy*

### Назначение

Определяет семейство алгоритмов, инкапсулирует каждый из них и делает их взаимозаменяемыми. Стратегия позволяет изменять алгоритмы независимо от клиентов, которые ими пользуются.

### Результаты

1. Иерархия классов Strategy определяет семейство алгоритмов или поведений, которые можно повторно использовать в разных контекстах. Наследование позволяет вычленить общую для всех алгоритмов функциональность.
2. Наследование поддерживает многообразие алгоритмов или поведений. Можно напрямую породить от Context подклассы с различными поведением. Но при этом поведение жестко «зашивается» в класс Context. Вот почему реализации алгоритма и контекста смешиваются, что затрудняет понимание, сопровождение и расширение контекста. Кроме того, заменить алгоритм динамически уже не удастся. В результате вы получите множество родственных классов, отличающихся только алгоритмом или поведением. Инкапсуляции алгоритма в отдельный класс Strategy позволяют изменять его независимо от контекста.
3. Благодаря паттерну стратегия удастся отказаться от условных операторов при выборе нужного поведения. Когда различные поведения помещаются в один класс, трудно выбрать нужное без применения условных операторов. Инкапсуляция же каждого поведения в отдельный класс Strategy решает эту проблему. Если код содержит много условных операторов, то часто это признак того, что нужно

применить паттерн стратегия.

4. Стратегии могут предлагать различные реализации одного и того же поведения. Клиент вправе выбирать подходящую стратегию в зависимости от своих требований к быстродействию и памяти;

5. Потенциальный недостаток этого паттерна в том, что для выбора подходящей стратегии клиент должен понимать, чем отличаются разные стратегии. Поэтому наверняка придется раскрыть клиенту некоторые особенности реализации. Отсюда следует, что паттерн стратегия стоит применять лишь тогда, когда различия в поведении имеют значение для клиента.

6. Интерфейс класса Strategy разделяется всеми его подклассами – неважно, сложна или тривиальна их реализация. Поэтому вполне вероятно, что некоторые стратегии не будут пользоваться всей передаваемой им информацией, особенно простые. Это означает, что в отдельных случаях контекст создаст и проинициализирует параметры, которые никому не нужны. Если возникнет проблема, то между классами Strategy и Context придется установить более тесную связь.

7. Применение стратегий увеличивает число объектов в приложении. Иногда эти издержки можно сократить, если реализовать стратегии в виде объектов без состояния, которые могут разделяться несколькими контекстами. Остаточное состояние хранится в самом контексте и передается при каждом обращении к объекту-стратегии. Разделяемые стратегии не должны сохранять состояние между вызовами. В описании паттерна приспособленец этот подход обсуждается более подробно.

## Реализация

```
class Context;  
// Стратегия объявляет общий для всех поддерживаемых алгоритмов  
интерфейс  
class Strategy {  
public:  
    virtual void Interface(Context *)=0;  
};  
// Конкретная стратегия реализует алгоритм  
class Strategy1 : public Strategy {  
public:  
    void Interface(Context *);  
};  
// Контекст хранит все необходимые алгоритму данные и указатель  
на объект класса Strategy  
class Context {  
    Strategy *st;  
public:  
    Context(Strategy *st): st(st) {}  
    void Operation() { st->Interface(this); }  
};
```

# Хранитель

*Memento*

## Назначение

Не нарушая инкапсуляции, фиксирует и выносит за пределы объекта его внутреннее состояние так, чтобы позднее можно было восстановить в нем объект.

## Результаты

1. Хранитель позволяет избежать раскрытия информации, которой должен распоряжаться только хозяин, но которую тем не менее необходимо хранить вне последнего. Этот паттерн экранирует объекты от потенциально сложного внутреннего устройства хозяина, не изменяя границы инкапсуляции.
2. При других вариантах дизайна, направленного на сохранение границ инкапсуляции, хозяин хранит внутри себя версии внутреннего состояния, которое запрашивали клиенты. Таким образом, вся ответственность за управление памятью лежит на хозяине. При перекладывании заботы о запрошенном состоянии на клиентов упрощается структура хозяина, а клиентам дается возможность не информировать хозяина о том, что они закончили работу.
3. С хранителями могут быть связаны заметные издержки, если хозяин должен копировать большой объем информации для занесения в память хранителя или если клиенты создают и возвращают хранителей достаточно часто. Если плата за инкапсуляцию и восстановление состояния хозяина велика, то этот паттерн не всегда подходит.

4. В некоторых языках сложно гарантировать, что только хозяин имеет доступ к состоянию хранителя.
5. Посыльный отвечает за удаление хранителя, однако не располагает информацией о том, какой объем информации о состоянии скрыт в нем. Поэтому нетребовательный к ресурсам посыльный может расходовать очень много памяти при работе с хранителем.

## Реализация

```
// Хранитель сохраняет внутреннее состояние объекта Originator
// и запрещает доступ всем другим объектам, кроме хозяина
class Memento {
    int state;
    Memento(int s):state(s) {}
    friend class Originator;
public:
};

// Хозяин создает хранитель и использует его для восстановления
внутреннего состояния
class Originator {
    int state;
public:
    Memento *CreateMemento() { return new Memento(state); }
    void SetMemento(Memento *m) { state=m->state; }
```

```
};  
// Смотритель отвечает за сохранение хранителя  
class Caretaker {  
    Memento *m;  
    Originator *orig;  
public:  
    void Action()  
    { m=orig->CreateMemento();  
      ...  
    }  
    void Undo()  
    { orig->SetMemento(m);  
    }  
};
```

## Цепочка обязанностей

### *Chain Of Responsibility*

#### Назначение

Позволяет избежать привязки отправителя запроса к его получателю, давая шанс обработать запрос нескольким объектам. Связывает объекты-получатели в цепочку и передает запрос вдоль этой цепочки, пока его не обработают.

#### Результаты

1. Этот паттерн освобождает объект от необходимости «знать», кто конкретно обработает его запрос. Отправителю и получателю ничего неизвестно друг о друге, а включенному в цепочку объекту – о структуре цепочки. Таким образом, цепочка обязанностей помогает упростить взаимосвязи между объектами. Вместо того чтобы хранить ссылки на все объекты, которые могут стать получателями запроса, объект должен располагать информацией лишь о своем ближайшем преемнике.
2. Цепочка обязанностей позволяет повысить гибкость распределения обязанностей между объектами. Добавить или изменить обязанности по обработке запроса можно, включив в цепочку новых участников или изменив ее каким-то другим образом. Этот подход можно сочетать со статическим порождением подклассов для создания специализированных обработчиков.
3. Поскольку у запроса нет явного получателя, то нет и гарантий, что он вообще будет обработан: он может достичь конца цепочки и пропасть. Необработанным



запрос может оказаться и в случае неправильной конфигурации цепочки.

## Реализация

// Обработчик определяет интерфейс для обработки запросов

```
class Handler {
```

```
protected:
```

```
    Handler *next;
```

```
public:
```

```
    virtual void HandleRequest()=0;
```

```
};
```

```
class Handler1 : public Handler {
```

```
public:
```

```
    void HandleRequest()
```

```
    { // если конкретный обработчик способен обработать запрос, то  
      так и делает
```

```
        ...
```

```
        // иначе направляет его своему преемнику
```

```
        next->HandleRequest();
```

```
    }
```

```
};
```

```
// Использование
```

```
Handler *first;
```

```
first->HandleRequest();
```

# Шаблонный метод

## *Template Method*

### Назначение

Шаблонный метод определяет основу алгоритма и позволяет подклассам переопределить некоторые шага алгоритма, не изменяя его структуру в целом.

### Результаты

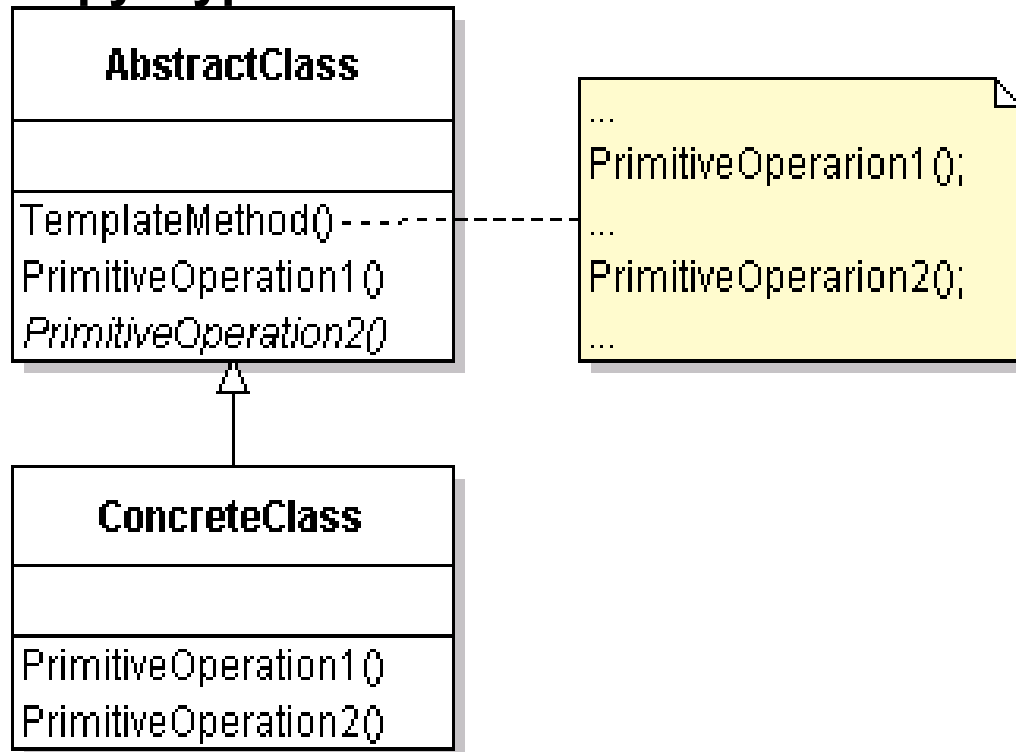
Шаблонные методы – один из фундаментальных приемов повторного использования кода. Они особенно важны в библиотеках классов, поскольку предоставляют возможность вынести общее поведение в библиотечные классы. Шаблонные методы приводят к инвертированной структуре кода, которую иногда называют принципом Голливуда, подразумевая часто употребляемую в этой киноимперии фразу «Не звоните нам, мы сами позвоним». В данном случае это означает, что родительский класс вызывает операции подкласса, а не наоборот. Шаблонные методы вызывают операции следующих видов:

- 1) конкретные операции из производных классов;
- 2) конкретные операции из базового класса, то есть операции, полезные всем подклассам;
- 3) примитивные операции, то есть абстрактные операции;
- 4) фабричные методы (см. паттерн фабричный метод);
- 5) операции-зацепки (hook operations), реализующие поведение по умолчанию, которое может быть расширено в подклассах. Часто такая операция по умолчанию

не делает ничего.

Важно, чтобы в шаблонном методе четко различались операции-зацепки, которые *можно* замещать, и абстрактные операции, которые *нужно* замещать. Чтобы повторно использовать абстрактный класс с максимальной эффективностью, авторы подклассов должны понимать, какие операции предназначены для замещения. Подкласс может расширить поведение некоторой операции, заместив ее и явно вызвав эту операцию из родительского класса.

## Структура



## Реализация

```
// Абстрактный класс определяет абстрактные примитивные операции,  
// замещаемые в конкретных подклассах для реализации шагов  
алгоритма
```

```
// и реализует шаблонный метод, определяющий скелет алгоритма
```

```
class AbstractClass {
```

```
public:
```

```
    void TemplateMethod()
```

```
    { ...
```

```
        PrimitiveOperation1();
```

```
        ...
```

```
        PrimitiveOperation2();
```

```
        ...
```

```
    }
```

```
    virtual void PrimitiveOperation1() {}
```

```
    virtual void PrimitiveOperation2()=0;
```

```
};
```

```
// Конкретный класс реализует примитивные операции
```

```
class ConcreteClass: public AbstractClass {
```

```
public:
```

```
    void PrimitiveOperation1();
```

```
    void PrimitiveOperation2();
```

```
};
```