<https://medium.com/@annacherenshchykova/-98ffef738f1d>

Принципы ООП

Главные принципы в ООП – абстрагирование, инкапсуляция, наследование и полиморфизм.

**Абстрагирование (абстракция)** – это выделение и представление существенных отличительных признаков, свойств, характеристик в терминах программирования.

**Инкапсуляция** – это «размещение в оболочке, капсуле», механизм объединения данных в единый компонент, дающий возможность защитить и спрятать их. То есть ограничить доступ одних компонентов программы к другим. Инкапсуляция позволяет над каждой частью программы работать изолированно. Это такая мантия-невидимка Гарри Поттера.

**Наследование** – это как в биологии схожесть детей, родителей и предков: передача свойств от одного объекта или класса другому, имеющему, помимо того, и собственные свойства. Похоже на смартфоны нового поколения: могут добавить камеры получше, но в целом наследуется то, чем мы уже пользуемся. Есть родительские/базовые классы и классы-потомки/наследники.

**Полиморфизм**: одно имя метода для класса – много внешне схожих, но технических различных действий, соответствующих особенностям наследников. Лаконичное определение: один интерфейс – множество реализаций.

# С++

## [Структура программы на языках С и С++. Функции С и С++. Перегрузка функций в С++. Ссылки.](https://github.com/RusinovaDaria/examOOP/wiki/№-1.-Структура-программы-на-языках-С-и-С--.-Функции-С-и-С--.-Перегрузка-функций-в-С--.-Ссылки.)

### Структура программы на языках С и С++

В программе обязательно присутствует функция int main(void). Она является управляющей, ее нельзя вызвать откуда-либо из программы (сама себя вызывать не может). Можно опустить return - функция будет возвращать 0.

Заголовочные файлы - описание типов, констант, библиотеки, функций, логически вынесенных в другой исходный файл.

Для получения исполняемого файла .exe:

* Препроцессирование - удаление комментариев, замена define, подключение заголовочных файлов .h (путем копирования содержимого). Для избежания повторного копирования используется #pragma once или #ifndef G\_H. Компиляция - перевод в код ассемблера
* Ассемблирование - перевод из кода ассемблера на машинный код. Получаем объектный файл.
* Компоновка, линковка объектных файлов в один исполняемый.

Некоторые нововведения с++:

* Классы
* New, delete
* Конструкторы и деструкторы
* Template
* Перегрузка функций и операторов
* throw/catch
* Умные указатели
* Ссылки

### Перегрузка функций в С++

Перегрузка – одно и то же имя может соответствовать нескольким функциям, если различаются их параметры.

unsigned abs(int);

double abs(double);

int sum(int, int);

double sum(double, double, double)

Параметры по-умолчанию: void sort (int \*A, int n, bool isAcs=true);

параметр isAcs будет тру, если не указано обратное. Допускается вызов sort(arr,5) и sort(arr,5,false), равно как и sort(arr,5,true). Параметры по умолчанию обязаны идти в конце списка параметров.

### Ссылки.

При передаче переменной по значению, происходит копирование памяти. При передаче объекта - создается новый объект - копия передаваемого.

int isNegative (int x); int a=3, b; b=a; isNegative(a); //происходит КОПИРОВАНИЕ памяти, передача по значению

В ООП удобно оперировать не копиями объектов, а оригиналами, поэтому вводится понятие ссылки:

Int& ref\_a = a; //ссылка должна быть проинициализирована и не может изменяться, указывать на другую переемнную;

реф\_а указывает на ту же память что и а.

Ссылка это не тип данных. Ссылка является «вторым именем» переменной а, ссылается на ту же память. В основном они применяются для передачи параметров в функции: void inc(int& a) { a++; } – фактически происходит передача адреса, по которому надо изменять значения – однако мы не используем \*разыменование.

Преимущество использования ссылок перед использованием указателя: безопасная работа с памятью. Ссылка позволяет проконтролировать выделение памяти и определение переменных.

Другое использование ссылок:

int& max(int \*A, int n)

{

int imax=0;

for

if (A[imax]>A[i])

imax = I;

return A[imax]

}

max(B,n) = 0;

Возврат по ссылке дает возможность сформировать левое значение.

(!) - Нельзя возвращать ссылки на локальные переменные.

Const int& refc\_a = a; //константная ссылка – по ней НЕЛЬЗЯ изменять значение, например через refc\_a++; или refc\_a =;

## [Классы и объекты в С++. Определение класса с помощью class, struct, union. Ограничение доступа к членам класса в С++. Члены класса и объекта. Методы. Константные члены. Схемы наследования.](https://github.com/RusinovaDaria/examOOP/wiki/№-2.-Классы-и-объекты-в-С--.-Определение-класса-с-помощью-class,-struct,-union.-Ограничение-доступа-к-членам-класса-в-С--.-Члены-класса-и-объекта.-Методы.-Константные-члены.-Схемы-наследования.)

**Классы и объекты в С++. Определение класса с помощью class, struct, union.**

Класс – тип данных, представляющий совокупность атрибутов объекта (переменные члены класса) и возможных операций над этими объектами (методов класса).

class <имя класса> [:<список базовых классов>]

{

private: //доступ к данным есть только внутри самого класса

int a; // член класса

protected: //доступ есть внутри класса и во всех его наследниках

int b;

public: //доступны для внешнего кода

int f(); // метод класса

}; //класс заканчивается ; как и структура

Объявления и определения классов располагаются в заголовочных файлах .h. Методы же определяются в .cpp

MyClass a, \*b = &a;

a.f();

b->f();

Размещение полей данных класса (атрибутов) публичными – плохой тон, поскольку это противоречит принципу инкапсуляции (скрывание реализации, предоставление пользователю только интерфейса для работы).

Объединения и структуры из си являются частными случаями классов.

Union - поля по одному адресу в памяти. Не может быть базовым и производным

union IPv4

{

int IntegerForm;

char ByteForm[4];

};

Однако юнион нельзя использовать в качестве базового класса и использовать для наследования.

struct -

struct Font

{

int Symbol:8;

int isItalic:1;

int isBold:1;

int color:3

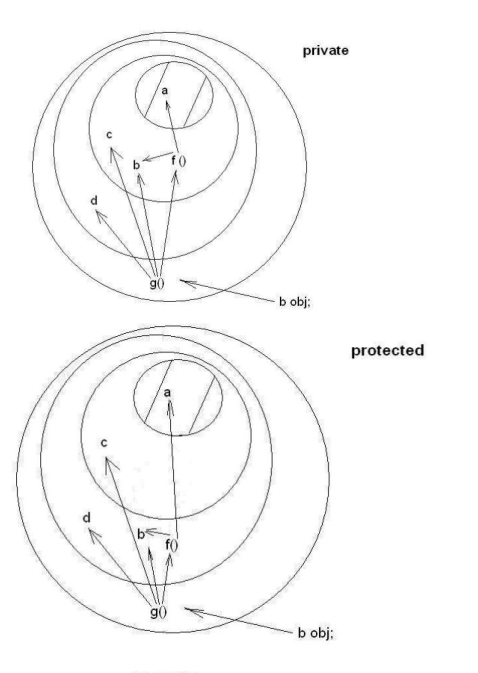
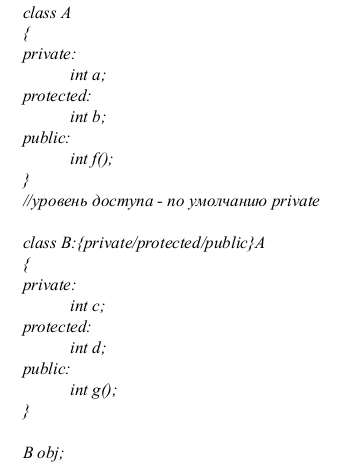
int :3 //необходимо для «закрытия» куска памяти – чтобы следующий юнион вдруг не попал на разделение байтов

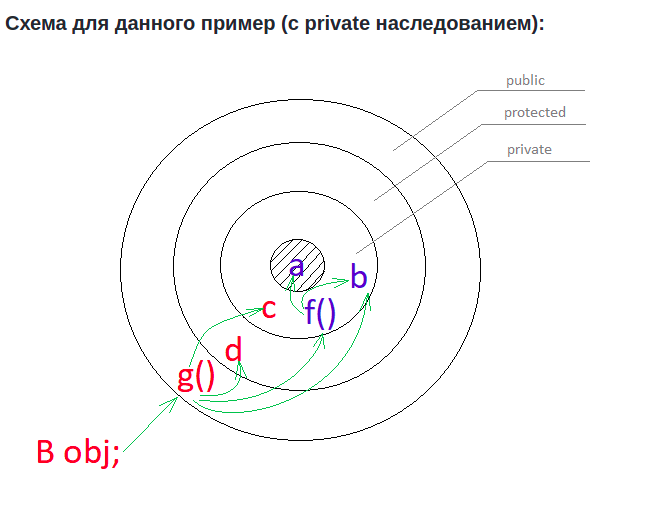
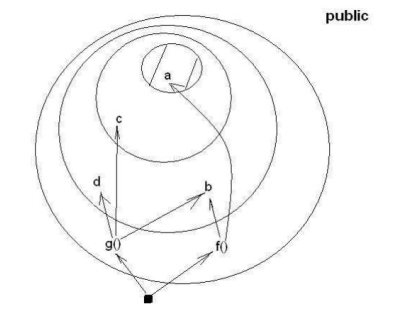
};

Допускается неполное объявление: class A; без определения класса.

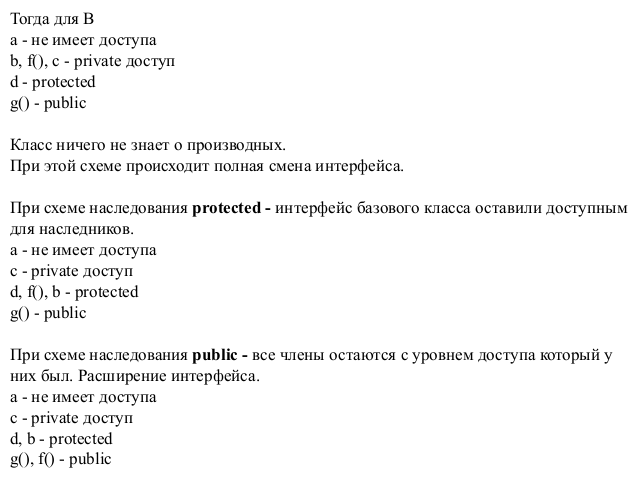
Создание объектов проходит как с помощью MyClass A; так и с помощью MyClass \*A = new MyClass;

**Ограничение доступа к членам класса в С++.**





При схеме наследования private - все члены класса наследуют с уровнем доступа private.



using a::f; - для методов private, protected - переопределять уровень доступа как у родительского.

Класс имеет 3 уровня доступа к своим атрибутам. По умолчанию - private; (в struct по умолчанию - public)

private - нет доступа извне protected - доступ только для потомков public - доступен извне

Порядок расположения: вначале приватные, потом протектные, потом публичные (метод белого ящика). Но в случае библиотечных классов инвертируется, паблик; протектед; приват.

**Схемы наследования.**

Класс может наследоваться от другого класса. Наследование - создание нового класса на основе старого.

В C ++ есть 3 схемы наследования:

* публичный (public)- публичные (public) и защищенные (protected) данные наследуются без изменения уровня доступа к ним; (доступ к унаследованным данным не меняется)
* защищенный (protected) — все унаследованные данные становятся защищенными;
* приватный (private) — все унаследованные данные становятся приватными.

**Члены класса и объекта. Методы.**

Статические члены или методы. Один общий для всех объектов этого класса. Доступен сразу из класса, без создания объекта. Вызывается для класса. Статические методы не имеют указателя this.

Class A

{

public:

static int a;

static void f();

};

A::a = 0;

A::f();

**Константные члены.**

Константное поле данных не меняет значение на протяжении всей жизни объекта. Инициализируется только в конструкторе.

Константный метод не изменяет объект, которым вызывается. Константный объект использует только константные методы.

class A {

public:

const int a;

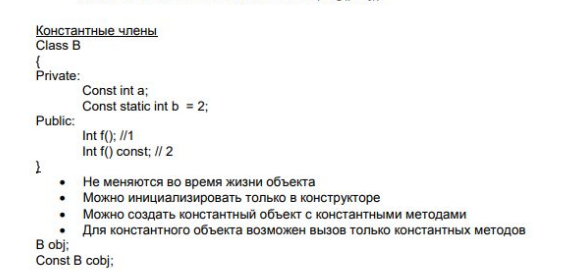
void f();

void f() const;

}

A b; b.f() // 1

Const A c; c.f() // 2

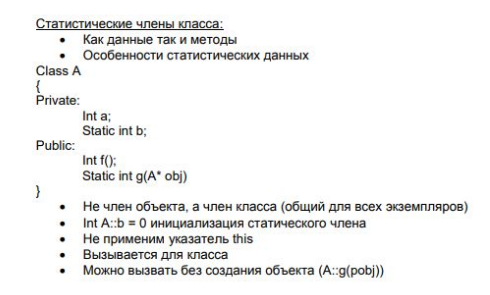


**Статические члены класса**

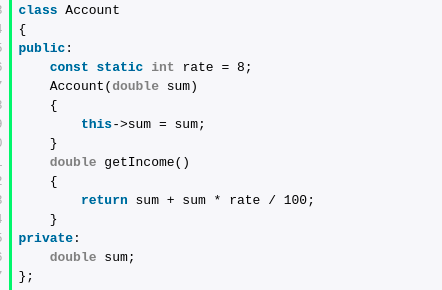
Статические данные относятся ко всем объектам класса. Такие данные используются, если

* требуется контроль общего количества объектов класса;
* требуется одновременный доступ ко всем объектам или части их;
* требуется разделение объектами общих ресурсов.

Статические поля **нельзя инициализировать в теле класса**, а также в методах. Статические поля должны инициализироваться аналогично глобальным переменным в области видимости файла:



**В отличие от статических переменных статические константы не нужно дополнительно определять вне класса.**



## [Создание и уничтожение объектов в С++. Конструкторы и деструкторы. Виды конструкторов. Раздел инициализации конструкторов. Способы создания объектов.](https://github.com/RusinovaDaria/examOOP/wiki/№-3.-Создание-и-уничтожение-объектов-в-С--.-Конструкторы-и-деструкторы.-Виды-конструкторов.-Раздел-инициализации-конструкторов.-Способы-создания-объектов.)

Когда выделили память под объект с помощью new, не нужно забывать освобождать ее с помощью delete... деструктор сам собой в этом случае не вызовется.

ТАК ЖЕ СТОИТ ПОЧИТАТЬ ПРО ИНИЦИАЛИЗАЦИЮ СТАТИК И КОНСТ ВЫШЕ!!!!!!!!!!

**Создание** происходит с помощью конструкторов, **уничтожение** - деструктора. Они существуют по умолчанию. Не имеют тип возврата.

**Конструктор -**  специальный метод – создаёт объект, не возвращает значения; имя совпадает с именем класса, могут принимать параметры.

В ЛЮБОМ классе автоматически создаются два конструктора – **конструктор по-умолчанию** (не принимающий параметра) и **конструктор копирования** (вызывается при передаче и возврате по значению).

Однако если в классе описан **хотя бы один не стандартный конструктор**, то конструктор-по-умолчанию не создаётся.

Аналогично, можно **явно задавать конструктор копирования**, в случае если под объект так или иначе выделяются ресурсы.

Если не нужна возможность передачи\возврата по значению (чтобы не создавались копии), то конструктор необходимо объявить приватным.

**Деструктор** также необходимо объявлять явно. Деструктор не принимает параметров.

Не может быть volatile (изменчивый), static, const. Не перегружается ( то есть только один для класса). Не наследуется.

*Для базовых классов вызывается в порядке обратном наследованию. Могут быть виртуальными.*

**Конструктор** также не может быть volatile, static, const, virtual. Возможна перегрузка. Не наследуется. Для базовых классов вызывается в порядке наследования.

class A

class B : public A

class C : public B

Для последнего: created A, created B, created C, destroyed C, destroyed B, destroyed A

**Проблема преобразования типов** – не даёт нормально обработать объект при преобразовании его из одного типа в другой. Может возникнуть ситуация, что при создании объекта возникнет ошибка, и будет неясно создан объект или нет. Чтобы не гадать о существовании объекта, используют оператор **explicit** – он запрещает преобразование типов и неявный вызов конструктора.

**Конструктор вызывается** при определении статических, локальных, временных объектов и при выполнении оператора new.

**Деструктор вызывается** при выполнении оператора delete, когда заканчивается область видимости локальных объектов, отпадает надобность во временных объектах, завершается программа.

Разница между конструктором перемещения и копирования. Когда они используются. <https://ravesli.com/urok-191-konstruktor-peremeshheniya-i-operator-prisvaivaniya-peremeshheniem/>

**Конструкторы копирования вызываются** при передачи или возврате объекта по значению, при инициализации одного объекта другим. принимают в качестве параметра константную ссылку l-value (в левой части операции = )

**Конструктор копирования** используется для инициализации **[класса](https://ravesli.com/urok-113-klassy-obekty-i-metody-klassov/)** путем создания копии необходимого объекта. **Оператор присваивания копированием** (или ***«копирующее присваивание»***) используется для копирования одного класса в другой (существующий) класс. По умолчанию язык C++ автоматически предоставляет конструктор копирования и оператор присваивания копированием, если вы не предоставили их сами.

**Конструктор перемещения** передает владение ресурсами новому объекту. в качестве параметра не константную ссылку r-value (в правой части =)

Class A {

Public:

A(); // : m\_a(0) {}; // A b;

A(int a); // A b(3); или A b = 3;

A(const A& b); // конструктор копирования // A b; A c = b;

A(A&& b); // конструктор переноса //

~A(); // деструктор

int m\_a;

};

Определение **конструктора перемещения и оператора присваивания перемещением** выполняется аналогично определению конструктора копирования и оператора присваивания копированием. Однако, в то время как функции с копированием принимают в качестве параметра константную ссылку l-value, функции с перемещением принимают в качестве параметра неконстантную **[ссылку r-value](https://ravesli.com/urok-190-ssylki-r-value/)**.

**Когда вызываются конструктор перемещения и оператор присваивания перемещением?**

Конструктор перемещения и оператор присваивания перемещением вызываются, когда аргументом для создания или присваивания является r-value. Чаще всего этим r-value будет литерал или временное значение (временный объект).

В большинстве случаев конструктор перемещения и оператор присваивания перемещением не предоставляются по умолчанию.

**Если вам нужен конструктор перемещения и оператор присваивания перемещением, которые выполняют перемещение (а не копирование), то вам их нужно предоставить (написать) самостоятельно.**

Рассмотрим на примере:

class Complex

{

private:

double re, im;

public:

Complex(); //1 //"неявный", "поумолчанию" конструктор

Complex(double r); //2

Complex(double r, double i); //3

Complex(Complex& C); //4 //копирование

};

int main()

{

Complex a(), //НЕ ОБЪЕКТ, функция!

b, //уже объект, НЕЯВНО вызывает конструктор 1

c=1., //вызывает констуктор 2

d(2.), //тоже вызывает конструктор 2

e(3.,4.), //конструктор 3

f=Complex(5.0,6.0), //тоже 3

g=f; //конструктор 4 - эквивалентно g(f), g=Complex(f)

return 0;

}

# **Виртуальный деструктор в C++**

В языке программирования C++ деструктор полиморфного базового класса должен объявляться виртуальным. Только так обеспечивается корректное разрушение объекта производного класса через указатель на соответствующий базовый класс.

Рассмотрим следующий пример.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | #include <iostream>    using namespace std;    // Вспомогательный класс  class Object  {  public:  Object() { cout << "Object::ctor()" << endl; }  ~Object() { cout << "Object::dtor()" << endl; }  };    // Базовый класс  class Base  {  public:  Base() { cout << "Base::ctor()" << endl; }  virtual ~Base() { cout << "Base::dtor()" << endl; }  virtual void print() = 0;  };    // Производный класс  class Derived: public Base  {  public:  Derived() { cout << "Derived::ctor()" << endl; }  ~Derived() { cout << "Derived::dtor()" << endl; }  void print() {}  Object obj;  };    int main ()  {  Base \* p = new Derived;  delete p;  return 0;  } |

В функции main указателю на базовый класс присваивается адрес динамически создаваемого объекта производного класса Derived. Затем через этот указатель объект разрушается. При этом наличие виртуального деструктора базового класса обеспечивает вызовы деструкторов всех классов в ожидаемом порядке, а именно, в порядке, обратном вызовам конструкторов соответствующих классов.

Вывод программы с использованием виртуального деструктора в базовом классе будет следующим:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | Base::ctor()  Object::ctor()  Derived::ctor()  Derived::dtor()  Object::dtor()  Base::dtor() |

Уничтожение объекта производного класса через указатель на базовый класс с невиртуальным деструктором дает неопределенный результат. На практике это выражается в том, что будет разрушена только часть объекта, соответствующая базовому классу. Если в коде выше убрать ключевое слово virtual перед деструктором базового класса, то вывод программы будет уже иным. Обратите внимание, что член данных obj класса Derived также не разрушается.

|  |  |
| --- | --- |
| 1  2  3  4 | Base::ctor()  Object::ctor()  Derived::ctor()  Base::dtor() |

**Когда же следует объявлять деструктор виртуальным?**

*Cуществует правило - если базовый класс предназначен для полиморфного использования, то его деструктор должен объявляться виртуальным.* Для реализации механизма виртуальных функций каждый объект класса хранит указатель на таблицу виртуальных функций vptr, что увеличивает его общий размер. Обычно, при объявлении виртуального деструктора такой класс уже имеет виртуальные функции, и увеличения размера соответствующего объекта не происходит.

Если же базовый класс не предназначен для полиморфного использования (не содержит виртуальных функций), то его деструктор не должен объявляться виртуальным.

[https://github.com/chrislvt/OOP/wiki/%D0%9B%D0%B5%D0%BA%D1%86%D0%B8%D1%8F-04.-%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%B8-%D1%81-%D0%BF%D0%B5%D1%80%D0%B5%D0%BC%D0%B5%D0%BD%D0%BD%D1%8B%D0%BC-%D1%87%D0%B8%D1%81%D0%BB%D0%BE%D0%BC-%D0%BF%D0%B0%D1%80%D0%B0%D0%BC%D0%B5%D1%82%D1%80%D0%BE%D0%B2.-%D0%9A%D0%BE%D0%BD%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%82%D0%BE%D1%80%D1%8B.#%D0%9C%D0%BE%D0%B4%D0%B8%D1%84%D0%B8%D0%BA%D0%B0%D1%82%D0%BE%D1%80%D1%8B-const-%D0%B8-static](https://github.com/chrislvt/OOP/wiki/Лекция-04.-Функции-с-переменным-числом-параметров.-Конструкторы.#Модификаторы-const-и-static)

## [Наследование в С++. Построение иерархии классов. Выделение общей части группы классов. Расщепление классов.](https://github.com/RusinovaDaria/examOOP/wiki/№-4.-Наследование-в-С--.-Построение-иерархии-классов.-Выделение-общей-части-группы-классов.-Расщепление-классов.)

**Наследование в С++.**

**Наследование** - создание нового класса на основе старого. Существует три схемы наследование: приватное, протектное и публичное. По умолчанию оно приватное.

class A

{

private:

int a;

protected:

int b;

public:

int f();

};

class B: public A //публичное наследование

{

private:

int c;

protected:

int d;

public:

int g();

};

B obj;

/\* приватное наследование

а не наследуется. c, b, f- приватные, d - протектная, g - публичная

извне можно получить доступ только к g (интерфейсу). Г имеет доступ к d,c,b,f. Ф имеет доступ к b,a

происходит смена интерфейса Ф на Г

протектное наследование

а не наследуется. c - приватная, b,d,f - протектные, g - публичная

-''-.

смена интерфейса, но в данном случае интерфейс Ф БУДЕТ ДОСТУПЕН к классам, наследующимся из класса Б

публичное наследование

все члены сохраняют свой уровень доступа базового класса

а не наследуется.

c - приватная, b,d - протектные, f,g - публичные.

интерфейс ДОПОЛНЯЕТСЯ.

\*/

**Private** – полная схема интерфейса.

**Public** – расширение интерфейса. Если нужно надстроить новый интерфейс поверх старого, закрыв что то старое и перекрыв что то новым, можно использовать кейворд using

class B: private A

{

public: using A::f;

}

В данном случае Б сможет использовать метод ф, несмотря на то что наследование – приватное. Тем не менее, ИЗВНЕ использовать pc->A::f() – это чрезвычайно не тот, нужно оборачивать в какой-нибудь другой метод.

**Выделение общей части:**

* Схожий набор свойств
* Общая схема использования
* Сходство реализации
* Общая схема исполнения разных объектов.
* В объектах один и тот же набор методов.
* Имеются два разных класса с разными методами. Если у методов похожая реализация, то выделяем базовый класс.
* У нас имеется два разных класса. Если в дальнейшем они будут участвовать вместе, лучше сделать для них базовый класс уже на этом этапе разработки. Это нужно сделать, чтобы в дальнейшем нам было легко модифицировать программу.

**Расщепление:**

* Разное использование
* Несвязная реализация
* Класс оперирует в двух несвязных обсуждениях проекта
* Если один объект исполняет разные роли.
* Два множества методов используются в разной манере.
* Методы между собой никак не связаны .
* Одна сущность, но используется в разных частях программы.

Набор классов, связанных отношением наследования, называют **иерархией.**

В производном классе можно определить тот же метод, что и в базовом. И Этот метод в производном классе будет доминировать (то есть при вызове будет выполняться именно он).

Если базовый класс наследуется как public, все открытые члены базового класса становятся открытыми членами производного класса, защищенные члены становятся защищенными членами, закрытые элементы базового класса остаются закрытыми в этом классе и недоступны для членов производного класса.

Если базовый класс наследуется как protected, все открытые и защищенные члены базового класса становятся защищенными членами производного класса.

Если базовый класс наследуется как private, открытые и защищенные члены базового класса становятся закрытыми членами производного класса. Они все еще доступны из членов производного класса, но из других частей программы к ним обратиться нельзя.

Во всех случаях наследования, закрытые члены базового класса остаются закрытыми и в этом классе.

**Конструкторы базовых объектов** вызываются в порядке наследования (от более базового к более специализированному), деструкторы в обратном.

С++ разрешает **множественное наследование**. Z наследует как от X, так и от Y class Z: public X, public Y

**Возникают неоднозначности при существовании одинаковых членов.** Стоит писать virtual при наследовании, что гарантирует переопределение и использование именно доминирующего члена.

Различают **прямое** ( А наследуется от В) и **косвенное** (А наследуется от В, которое наследуется от С; А и С косвенные) наследование.

--------------

ООП использует **рекурсивный дизайн** – постепенное разворачивание программы от базовых классов к более специализированным. С++ один из немногих языков с множественным наследованием. Оно может упростить граф наследования, но также создает пучок проблем для программиста: возникает неоднозначность, которую бывает тяжело контролировать.

class A

{

public:

int a; //арара! объявление публичного поля!

int (\*b)(); //указатель на функцию! тоже публичное поле!

//более того, указателей на ФУНКЦИИ ВООБЩЕ не должно быть! максимум - на методы

int f();

int f(int);

int g();

}

class B

{

int a; //допускается - приватные поля

int b;

public:

int f();

int g; //арара публичное свойство!

int h();

int h(int);

}

class C: public A, public B {};

//метод рандомного класса

X::f(C \*pc)

{

pc->a = 1; //еггог! ошибка доступа

pc->b(); //то же самое!

pc->f(); //йух пойми какой Ф вызывать, то ли из А, толи из Б

pc->f(1) //перегрузка не произойдёт

pc->g = 1; //функция или переменная!?

pc->h();

pc->h(1); //а вот в этих двух всё ок

ПРАВИЛА

**Наследование**

**Иерархия классов:**

1. Есть 2 или более классов и мы их объединяем общей базой (базовым классом).

Общая схема использования.

2. Два подмножества класса строятся в разной манере (создаем производные классы)

3. Класс фигурирует в двух или нескольких несвязанных между собой обсуждениях

проекта.

**Построение иерархии классов.**

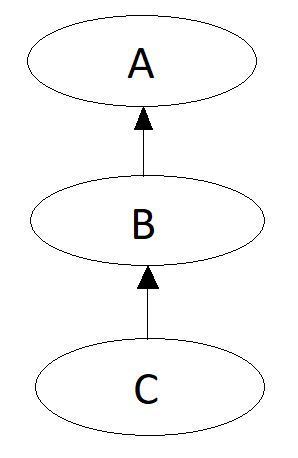
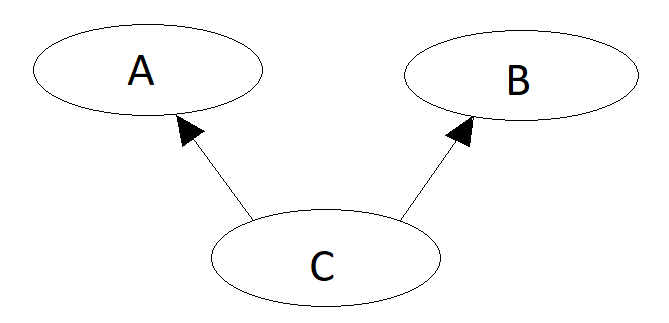
ну тут видимо ничего, так как это является не отдельным вопрос а его частью в совокупности в расщеплением и выделением части

## !!!![Множественное наследование. Прямая и косвенная базы. Виртуальное наследование. Понятие доминирования. Порядок создания и уничтожения объектов. Проблемы множественного наследования. Неоднозначности при множественном наследовании.](https://github.com/RusinovaDaria/examOOP/wiki/№-5.-Множественное-наследование.-Прямая-и-косвенная-базы.-Виртуальное-наследование.-Понятие-доминирования.-Порядок-создания-и-уничтожения-объектов.-Проблемы-множественного-наследования.-Неоднозначности-при-множественном-наследовании.)

**Множественное наследование**

Язык С++ – это практически единственный язык, в котором присутствует множественное наследование.

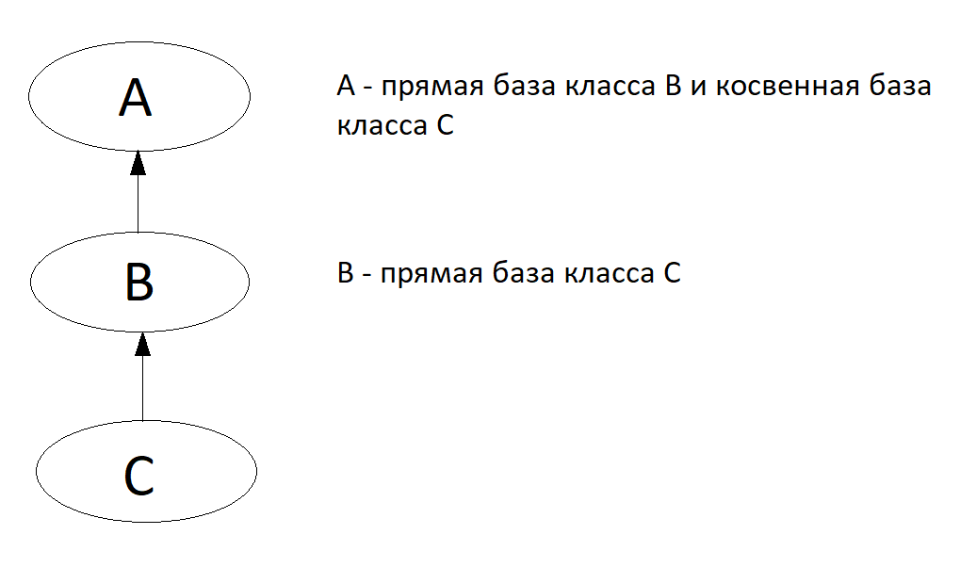
*Преимущества множественного наследования*

1. Представим такую ситуацию: выстраиваем вертикальную иерархию, класс C наследуется от класса B, а B наследуется от класса A. В этом случае, в класс A мы должны вынести много того, что к понятию класса А не относится. Не совсем логично.  
     
   В случае со множественным наследованием (рис. ниже), мы четко разделяем понятия A и B. Такой подход уменьшает иерархию.  
   
2. Второй момент (опять не используем множественное наследование). Мы можем не выносить что-то в базовый класс, а один из классов включить как подобъект, то есть не использовать наследование. Пусть С - производная от класса А и включает подобъект класса В. Тоже возникнет проблема - не будем иметь доступа к защищенным полям класса Б (нам придется делать это через методы класса Б) + придется протаскивать интерфейс для класса С класса Б.

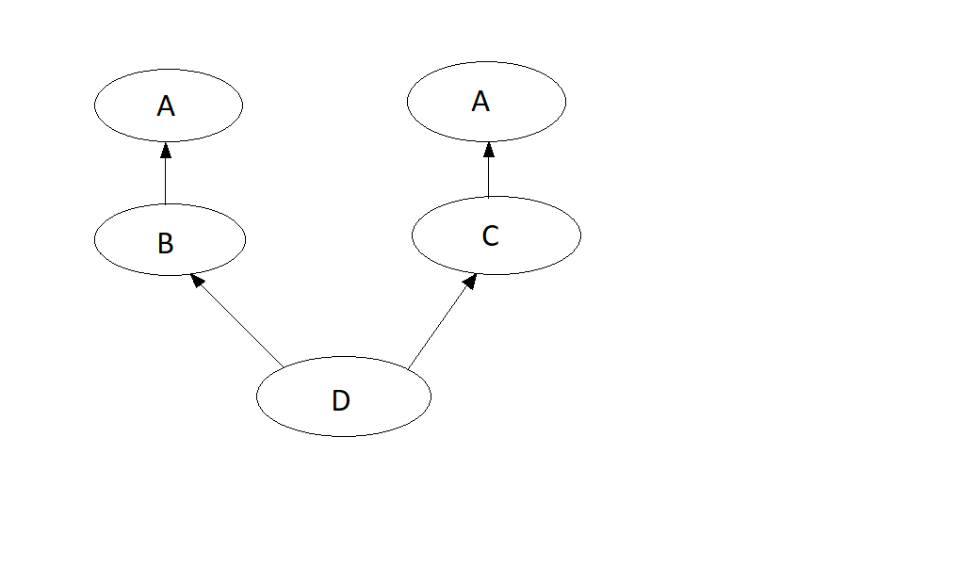
**Прямая и косвенная базы**

Непосредственная база для какого-либо класса является прямой базой. Может входить в производный класс только один раз. Прямой базовый класс явно перечисляется в заголовке при объявлении производного класса.

Косвенная база - прямая база прямой базы. Может входить в базовый класс сколько угодно раз.



Может возникнуть ситуация, когда в наш класс косвенная база входит два раза:



В большинстве случаев нам необходимо, чтобы базовый класс входил в производный только один раз. Рассмотрим пример ниже.

*Пример. Базовый класс входит в производный два раза*

В данном примере два раза отрабатывает конструктор класса А.

class A

{

public:

A (char\* s)

{

cout << "Creature A" << s << ";" << endl;

}

};

class B : public A

{

public:

B() : A (" from B")

{

cout << "Creature B;" << endl;

}

};

class C : public B, public A // В класс C подобъект класса А будет входить два раза.

{

public:

C() : A(" from C")

{

cout << "Creature C;" << endl;

}

};

void main()

{

C obj;

}

Вызовется конструктор класса С. Из С вызовется конструктор класса B (так как класс B наследуется раньше класса A). Вызовется конструктор класса A. Создастся объект класса А. Создастся объект класса B. Из С вызовется конструктор класса А. Создастся объект класса А. Создастся объект класса С.

*На экран в результате работы программы будет выведено следующее:*

Creature A from B;

Creature B;

Creature A from C;

Creature C;

Проблема решается с помощью **виртуального наследования.**

**Виртуальное наследование**

В большинстве случаев необходимо, чтобы базовый класс входил в производный только один раз. Для этого используется виртуальное наследование.

ВАЖНО! При виртуальном наследовании меняется порядок создания объекта: если в списке наследования есть виртуальное наследование (виртуальные базы), они отрабатывают в первую очередь слева направо, а потом всё остальные базы.

Чтобы сделать родительский (базовый) класс общим, используется ключевое слово virtual в строке объявления дочернего класса.

*Пример. Базовый класс входит в производный один раз*

Исправим предыдущий пример, добавив виртуальное наследование. Теперь, когда для класса С будет создаваться объект класса А, будет включаться механизм виртуальности. Когда будет создаваться подобъект класса С, для него не будет создан объект класса А.

class A

{

public:

A(char\* s)

{

cout << "Creature A" << s << ";" << endl;

}

};

class B : virtual public A

{

public:

B() : A(" from B")

{

cout << "Creature B;" << endl;

}

};

class C : public B, virtual public A

{

public:

C() : A(" from C")

{

cout << "Creature C;" << endl;

}

};

void main()

{

C obj;

}

На экран в результате работы программы будет выведено следующее:

Creature A from C;

Creature B;

Creature C;

*Проблемы виртуального наследования*

Рассмотрим следующий пример:

class A {};

class B : virtual public A{}; // Здесь virtual наследование

class C : public A {}; // Здесь не virtual наследование

class D : public B, public C{};

Порядок создания объекта класса D: сначала вызывается конструктор класса B. Для него вызывается конструктор класса А, будет выполняться механизм виртуальности. Создастся подобъект класса А, отработает конструктор класса B. Для C уже не будет выполняться A.

Но если поменять порядок наследования для класса D:

class D : public C, public B{};

Смена последовательности наследование приводит к тому, что класс А будет включен два раза, что не должно происходить при включении механизма виртуальности.

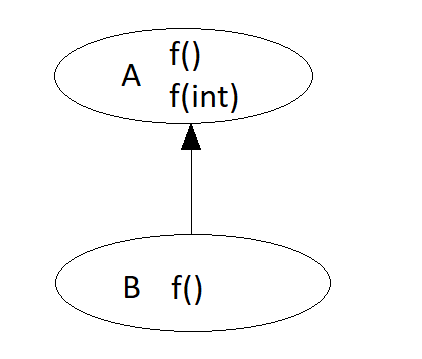
ВАЖНО! Используя множественное наследование, виртуально надо стараться виртуально наследоваться по всем ветвям, чтобы не зависеть от порядка наследования.

**Понятие доминирования**

В общем случае, диаграмма наследования представляет собой направленный ациклический граф.

ВАЖНО! Метод, находящийся в наге дальше, подменяет другие методы, находящиеся выше.

Рассмотрим нижеприведенную схему. В классе А есть перегруженный метод f: f() и f(int). В производном классе есть всего один метод f().



В данном случае в классе B подменяется метод f(). Для объектов класса B метод f(int) недоступен. Это сделано, чтобы при проектировании было корректное наследование.

ВАЖНО! Если мы подменяем один перегруженный метод, мы обязаны подменить все остальные. Если мы этого не сделаем, они будут перекрыты.

*Пример. Использование using*

Можно использовать using (но это плохо). В данном примере мы хоть и описали в классе B только один перегруженный метод, но с помощью using мы можем использовать f(int). Таким образом, объект класса B может использовать f() своего класса B и f(int) базового класса A.

class A

{

public:

void f() { cout<<"Executing f() from A;"<<endl; }

void f(int i) { cout<<" Executing f(int) from A;"<<endl; }

};

class B : virtual public A

{

public:

void f() { cout<<"Executing f from B;"<<endl; }

using A::f; // плохо!!!

};

class C : virtual public A

{

};

class D : virtual public C, virtual public B

{

};

void main()

{

D obj;

obj.f(); // вызвается метод f() класса B

obj.f(1); // Вызывается метод f(int) класса A

}

Программа выведет на экран:

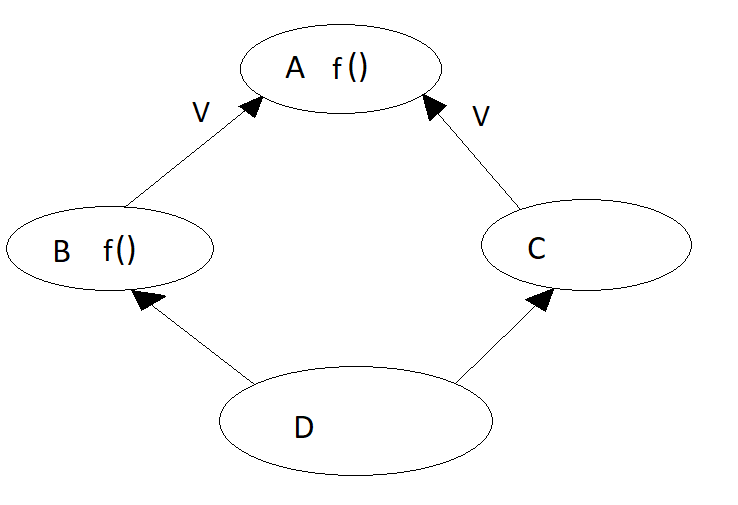
Executing f from B;

Executing f(int) from A;

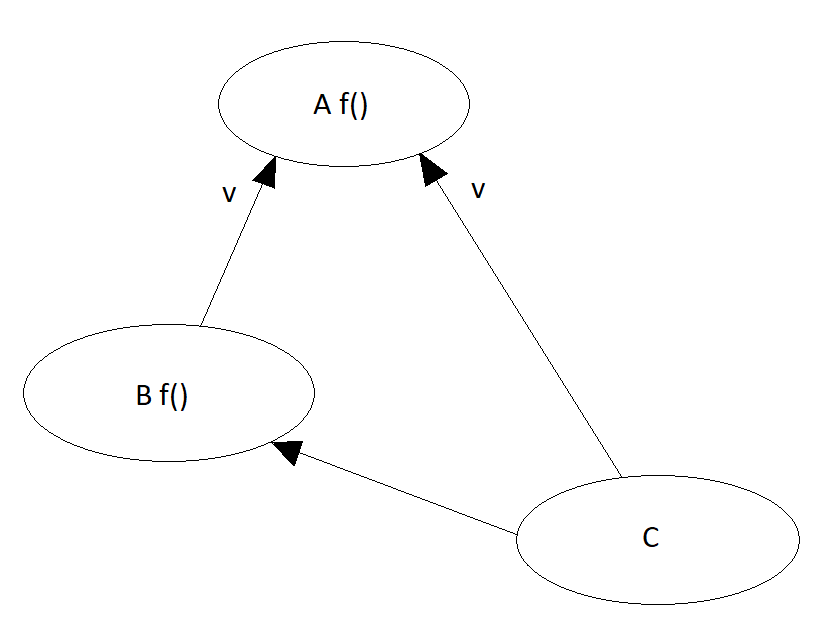
*Пара схем из мира доминирования*

Какие схемы возможны при множественном наследовании?

В данном случае для объектов класса D метод f() класса B подменят метод f() класса A.



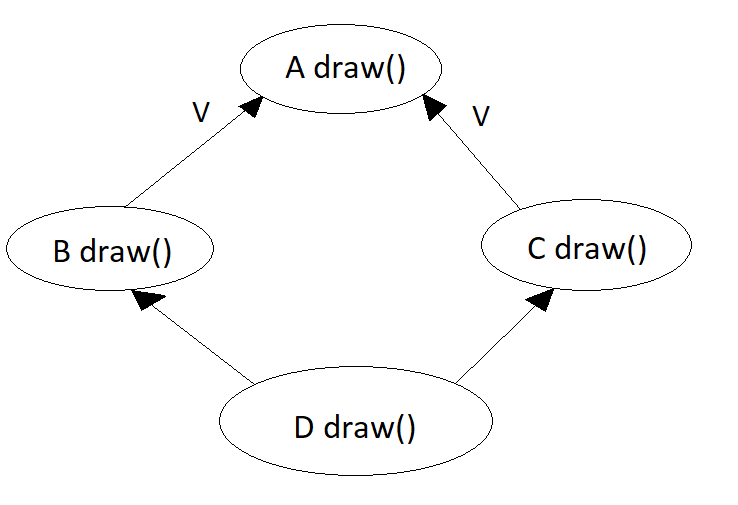
Аналогично для этой схемы:



**Порядок создания и уничтожения объектов (в каждом разделе что-то да написано про это)**

**Проблемы множественного наследования**

Рассмотрим следующую схему:



Предположим, есть объект класса А с методом draw(), который умеет себя нарисовать. Производные от него классы - B и C, тоже имеют метод draw() и тоже могут себя нарисовать, а так же они могут нарисовать подобъект базового класса. То есть, при рисовании объектов класса B (аналогично для C) вызывается метод draw() класса А.

Когда мы создаем объект класса D, в котором мы должны отрисовать объект класса B и объект класса C, draw() класса A вызывается два раза. Это называется проблема множественного вызова базового класса.

*Пример. Множественный вызов методов*

class A

{

public:

void f() { cout<<"Executing f from A;"<<endl; }

};

class B : virtual public A

{

public:

void f()

{

A::f();

cout<<"Executing f from B;"<<endl;

}

};

class C : virtual public A

{

public:

void f()

{

A::f();

cout<<"Executing f from C;"<<endl;

}

};

class D : virtual public C, virtual public B

{

public:

void f()

{

C::f();

B::f();

cout<<"Executing f from D;"<<endl;

}

};

void main()

{

D obj;

obj.f();

}

Метод f() класса А срабатывает дважды.

Программа выведет на экран:

Executing f from A;

Executing f from C;

Executing f from A;

Executing f from B;

Executing f from D;

Красивых способов борьбы с этой чумой, к сожалению, нет.

*Пример. Решение проблемы множественного вызова методов*

Идея: разделить метод на две части. Часть, которая относится непосредственно к самому классу, и часть, которая относится ко всему объекту.

В классе А мы разделили f() на две части, причем, то что относится к самому классу A - его собственное, мы делаем его защищенным, доступным только для методов класса А и производных классов. В производных классах мы тоже разделяем - метод, относящийся непосредственно к самому классу и ко всему объекту.

class A

{

protected:

void \_f() { cout<<"Executing f from A;"<<endl; }

public:

void f() { this->\_f(); }

};

class B : virtual public A

{

protected:

void \_f() { cout<<"Executing f from B;"<<endl; }

public:

void f()

{

A::\_f();

this->\_f();

}

};

class C : virtual public A

{

protected:

void \_f() { cout<<"Executing f from C;"<<endl; }

public:

void f()

{

A::\_f();

this->\_f();

}

};

class D : virtual public C, virtual public B

{

protected:

void \_f() { cout<<"Executing f from D;"<<endl; }

public:

void f()

{

A::\_f(); C::\_f(); B::\_f();

this->\_f();

}

};

void main()

{

D obj;

obj.f();

}

Программа выведет на экран:

Executing f from A;

Executing f from C;

Executing f from B;

Executing f from D;

Решение не самое красивое, но других, к сожалению, нет.

**Неоднозначности при множественном наследовании**

*Пример. Неоднозначности при множественном наследовании*

class A

{

public:

int a; //арара! объявление публичного поля!

int (\*b)(); //указатель на функцию! тоже публичное поле!

//более того, указателей на ФУНКЦИИ ВООБЩЕ не должно быть! максимум - на методы

int f();

int f(int);

int g();

}

class B

{

int a; //допускается - приватные поля

int b;

public:

int f();

int g; //арара публичное свойство!

int h();

int h(int);

}

class C: public A, public B {};

//метод рандомного класса

X::f(C \*pc)

{

pc->a = 1; //еггог! ошибка доступа

pc->b(); //то же самое!

pc->f(); //йух пойми какой Ф вызывать, то ли из А, толи из Б

pc->f(1) //перегрузка не произойдёт

pc->g = 1; //функция или переменная!?

pc->h();

pc->h(1); //а вот в этих двух всё ок

}

Есть два класса – класс А и класс В. Класс С – производная от классов А и В. В классе С мы получаем доступ к членам объекта класса C. Здесь играет следующее **правило проверки на неоднозначность: проверка на неоднозначность происходит до проверки на перегрузку, на тип и до проверки на уровень доступа.**

Решение проблемы (тоже некрасивое решение): Объединяем два класса, и в производном классе полностью подменяем то, что находится в базовых классах. Получается громоздкий производный класс (ведь мы всё, что относится к базовым классам - подменяем). Когда мы объединяем два класса одним классом, есть еще один недостаток.(см. пример ниже).

*Пример. Замена интерфейса*

Также есть еще один недостаток – когда мы программируем, для объекта какого-либо класса мы должны выделить цель - для чего мы создаем это понятие. Объекты данного класса должны выполнять определенную задачу, и она должна быть только одна. ***Правило: У одного объекта не должно быть несколько обязанностей.*** Когда мы объединяем два понятия, это чаще всего приводит к тому, что новое сформированное понятие имеет несколько обязанностей. Такого быть не должно.

Если один объект выступает в разных ролях, мы не должны объединять интерфейс, должны его разносить.

Исключением является ситуация, когда мы объединяем два разных понятия, формируя интерфейс одной обязанности. В этом случае используется следующая схема. У нас есть два класса, и мы формируем новое понятие, используя интерфейс только одного класса. В данном случае идёт наследование только по схеме public только от класса B, от класса A по схеме private. Таким образом, для объектов класса C интерфейс класса A невидим.

class A

{

public:

void f1() { cout<<"Executing f1 from A;"<<endl; }

void f2() { cout<<"Executing f2 from A;"<<endl; }

};

class B

{

public:

void f1() { cout<<"Executing f1 from B;"<<endl; }

void f3() { cout<<"Executing f3 from B;"<<endl; }

};

class C : private A, public B {};

class D

{

public:

void g1(A& obj)

{

obj.f1();

obj.f2();

}

void g2(B& obj)

{

obj.f1();

obj.f3();

}

};

void main()

{

C obj;

D d;

// obj.f1(); Error!!! Неоднозначность

// d.g1(obj); Error!!! Нет приведения к базовому классу при наследовании по схеме private

d.g2(obj);

}

Но здесь здесь есть проблема – проверка на неоднозначность происходит до проверки на схему наследования. Поэтому метод f() для объектов класса C мы вызвать не сможем - это неоднозначность, хотя наследуем по разной схеме.

Что нужно сделать: Нужно в классе С подменить те методы, которые идут по ветви public.

При такой схеме (когда в одном случае мы поддерживаем только один интерфейс) множественное наследование можно использовать.

ВАЖНО! Если в наге нет общей базы (общая база задает интерфейсные методы для производных классов), то подмена должны осуществляться только по одной ветке.

## [Полиморфизм в С++. Виртуальные методы. Виртуальные деструкторы. Чисто виртуальные методы. Понятие абстрактного класса. Ошибки возникающие при работе с указателем на базовый класс. Дружественные связи.](https://github.com/RusinovaDaria/examOOP/wiki/№6-Полиморфизм-в-С--.-Виртуальные-методы.-Виртуальные-деструкторы.-Чисто-виртуальные-методы.-Понятие-абстрактного-класса.-Ошибки-возникающие-при-работе-с-указателем-на-базовый-класс.-Дружественные-связи.)

**Полиморфизм в С++**

**Полиморфизм**: одно имя метода для класса – много внешне схожих, но технических различных действий, соответствующих особенностям наследников. Лаконичное определение: один интерфейс – множество реализаций.

***Полиморфизм*** (безразличие - не важно кто предоставляет функцонал) - использование объектов с одинаковым интерфейсом без информации о типе и внутренней структуре объекта. (“Один интерфейс, множество реализаций.”)

**Виртуальные методы**

**Виртуальная функция в языке С++** — это особый тип функции, которая, при её вызове, выполняет «наиболее» дочерний метод, который существует между родительским и дочерними классами. Это свойство еще известно, как **полиморфизм**.

***Идея виртуальных методов*** - когда создаются объекты, создавать в памяти специальные таблицы (виртуальные), которые для данного объекта содержат адреса методов, которые надо вызывать. Соответсвенно объект должен иметь указатель на эту таблицу. Всегда это использовать не удобно.

***Минусы***

1. увеличивается объём памяти. (В памяти хранятся таблицы для каждого объекта).
2. Увеличивается время вызова метода. (В таблице нужно найти метод, получить его адрес и по нему вызвать метод).

***Плюсы***

1. Лёгкость подмены одного понятия на другое

Если мы в класс добавляем, хотя бы один виртуальный метод, то в объекты этого класса добавляется указатель на виртуальную таблицу. Такие классы называются *полиморфными*.

*Правило полиморфных классов*

Базовый класс задаёт интерфейс, который в производных, мы не должны ни сужать, ни расширять

**Чисто Виртуальные методы. Абстрактные классы.**

*Чисто виртуальный метод* не имеет тела.

virtual Product CreateProduct() = 0;

Класс с хотя бы одной чисто виртуальным методом называется *абстрактным*. Объекты абстрактного класса создавать нельзя.

Если производные классы не будут подменять чисто виртуальные методы, они будут то же абстрактными.

Для абстрактного класса все методы, которые могут быть подменены в производном, определяем с модификатором virtual.

Для полиморфного класса мы всегда должны определять деструктор.

class A

{

public:

virtual void f() = 0;

virtual ~A() = 0;

};

Тут возникает проблема, объект уничтожается в обратном порядке (относительно порядка создания). Поэтому реализовать этот деструктор мы обязаны. (звучит, как костыль)

A::~A() = default;

**Понятие абстрактного класса.**

Базовый класс, объекты которого никогда не будут реализованы называется ***абстрактным* классом**. Такой класс может существовать с единственной целью – быть родительским классом к производным классом, объекты которых будут реализованы.

Для того чтобы сделать базовый класс абстрактным, достаточно ввести в класс хотя бы одну чисто виртуальную функцию.

**Чисто *виртуальная функция –*** это функция, после объявления которой добавлено выражение **=0**.

**Виртуальный деструктор в C++**

В языке программирования C++ деструктор полиморфного базового класса должен объявляться виртуальным. Только так обеспечивается корректное разрушение объекта производного класса через указатель на соответствующий базовый класс.

Рассмотрим следующий пример.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | #include <iostream>    using namespace std;    // Вспомогательный класс  class Object  {  public:  Object() { cout << "Object::ctor()" << endl; }  ~Object() { cout << "Object::dtor()" << endl; }  };    // Базовый класс  class Base  {  public:  Base() { cout << "Base::ctor()" << endl; }  virtual ~Base() { cout << "Base::dtor()" << endl; }  virtual void print() = 0;  };    // Производный класс  class Derived: public Base  {  public:  Derived() { cout << "Derived::ctor()" << endl; }  ~Derived() { cout << "Derived::dtor()" << endl; }  void print() {}  Object obj;  };    int main ()  {  Base \* p = new Derived;  delete p;  return 0;  } |

В функции main указателю на базовый класс присваивается адрес динамически создаваемого объекта производного класса Derived. Затем через этот указатель объект разрушается. При этом наличие виртуального деструктора базового класса обеспечивает вызовы деструкторов всех классов в ожидаемом порядке, а именно, в порядке, обратном вызовам конструкторов соответствующих классов.

Вывод программы с использованием виртуального деструктора в базовом классе будет следующим:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | Base::ctor()  Object::ctor()  Derived::ctor()  Derived::dtor()  Object::dtor()  Base::dtor() |

Уничтожение объекта производного класса через указатель на базовый класс с невиртуальным деструктором дает неопределенный результат. На практике это выражается в том, что будет разрушена только часть объекта, соответствующая базовому классу. Если в коде выше убрать ключевое слово virtual перед деструктором базового класса, то вывод программы будет уже иным. Обратите внимание, что член данных obj класса Derived также не разрушается.

|  |  |
| --- | --- |
| 1  2  3  4 | Base::ctor()  Object::ctor()  Derived::ctor()  Base::dtor() |

**Когда же следует объявлять деструктор виртуальным?**

*Cуществует правило - если базовый класс предназначен для полиморфного использования, то его деструктор должен объявляться виртуальным.* Для реализации механизма виртуальных функций каждый объект класса хранит указатель на таблицу виртуальных функций vptr, что увеличивает его общий размер. Обычно, при объявлении виртуального деструктора такой класс уже имеет виртуальные функции, и увеличения размера соответствующего объекта не происходит.

Если же базовый класс не предназначен для полиморфного использования (не содержит виртуальных функций), то его деструктор не должен объявляться виртуальным.

*Проблемы с виртуальными методами*.

**Мы не должны вызвать виртуальные методы в конструкторах и деструкторах.**

class A

{

public:

virtual ~A()

{

cout << "Class A destructor called;" << endl;

}

virtual void f()

{

cout << "Executing f from A" << endl;

}

};

class B : public A

{

public:

B()

{

// прикол в том, что класса C ещё нет

// поэтому вызовется метод класса A

this->f();

}

virtual ~B()

{

cout<<"Class B destructor called;"<<endl;

// прикол в том, что класса C уже нет

// поэтому вызовется метод класса A

this->f();

}

void g() { this->f(); }

};

class C : public B

{

public:

virtual ~C()

{

cout<<"Class C destructor called;"<<endl;

}

virtual void f() override

{

cout<<"Executing f from C;"<<endl;

}

};

void main()

{

C obj;

// this в методе B::g() будет = obj;

// т.е отработает штатно

obj.g();

}

**Дружба**

*Дружба* - это зло. Дружба даёт доступ ко всем членам класса методов других классов. В классе для объектов указываем, что есть «друг». Схема получается очень зависимая.

Она приводит к тому, что если нужно вносить изменения написанный код (фу позор!!!).

Необходимо, чтобы дружественных отношений было как можно меньше.

1. Не наследуется (сын друга - не друг)
2. Не транзитивна (друг моего друга - не друг).

### Дружба и наследование.

class C; // forward объявление

class A

{

private:

void f1() { cout<<"Executing f1;"<<endl; }

friend C;

};

class B : public A

{

private:

void f2() { cout<<"Executing f2;"<<endl; }

};

class C

{

public:

static void g1(A& obj) { obj.f1(); }

static void g2(B& obj)

{

obj.f1();

// obj.f2(); // Error!!! Имеет доступ только к членам A

}

};

class D : public C

{

public:

// static void g2(A& obj) ( obj.f1(); } // Error!!! Дружба не наследуется

};

### Дружба и виртуальные методы.

class C; // forward объявление

class A

{

protected:

virtual void f() { cout<<"Executing f from A;"<<endl; }

friend C;

};

class B : public A

{

protected:

virtual void f() override { cout<<"Executing f from B;"<<endl; }

};

class C

{

public:

static void g(A& obj) { obj.f(); }

};

void main()

{

B bobj;

C::g(bobj);

}

Пример из лекции

class C

{

friend void f(C& ac); // друг-функция

friend A::f(); // друг-метод

friend B; // друг-класс (самый плохой вариант)

};

**Ошибки возникающие при работе с указателем на базовый класс.**

Предположим, у нас есть класс B, производный от класса A.

А \*p = new В; // Создание объекта класса B. Мы вызываем конструктор класса B для создания конкретного объекта.

.

.

.

delete p;

Класс А – абстрактный, класс В – не абстрактный. Мы можем работать с классом В, вызывая метод. Вступает правило: для класса А мы все методы, которые могут быть подменены в классе В, должны определить с модификатором virtual, чтобы один объ ект можно было подменить другим.

Мы работаем с указателем на А. Мы подменили один объект другим, но правую часть мы вызываем конкретно. Напрашивается виртуальный конструктор, но конструктор - это не метод объекта, а метод класса, конструктор не может быть виртуальным. Получается проблема с подменой (спойлер: решается с помощью порождающих паттернов).

**Возникает еще одна проблема** – вызывается деструктор, а деструктор должен вызываться для объекта класса В. Но деструктор вызывается для объекта, поэтому деструктор может быть виртуальным. Соответственно, когда мы определяем какой-либо класс, в любом случае для базового полиморфного класса мы должны определить деструктор. Если мы не можем его определить, то мы делаем деструктор чисто виртуальным.

class A

{

public:

virtual void f() = 0;

virtual ~A() = 0;

};

Но возникает проблема – у нас создается объект какого-то производного класса, по цепочке отрабатывает конструктор, в обратном порядке отрабатывают деструкторы. А мы удалили этот деструктор! Поэтому реализовать этот деструктор мы обязаны. Реализовать как пустой.

class A

{

public:

virtual void f() = 0;

virtual ~A() = 0;

};

A::~A() {}

Задав виртуальный деструктор, каждый производный класс определяет для себя этот деструктор.

Итоги: если у нас полиморфный базовый класс, мы должны подменяемые методы определить, как виртуальные, а так же должны объявить и определить виртуальный деструктор. В дальнейшем мы будем говорить, что базовые классы ВСЕГДА должны быть абстрактными. Как раз механизм с виртуальным деструктором дает нам возможность формировать такой базовый класс. Если мы определили чисто виртуальный деструктор в классе, то этот класс тоже является абстрактным, хотя, казалось бы, есть его реализация.

## [Обработка исключительных ситуаций в С++. Проблемы с динамической памятью при обработке исключительных ситуаций. Перегрузка операторов в С++. Правила перегрузки операторов. Операторы .*, ->*. Перегрузка унарных и бинарных операторов. Перегрузка оператора = : копирование, перенос. Перегрузка операторов ->, \*, []. Перегрузка операторов ++, --. Операторы приведения типов.](https://github.com/RusinovaDaria/examOOP/wiki/№-7.-Обработка-исключительных-ситуаций-в-С--.-Проблемы-с-динамической-памятью-при-обработке-исключительных-ситуаций.-Перегрузка-операторов-в-С--.-Правила-перегрузки-операторов.-Операторы-.,---.-Перегрузка-унарных-и-бинарных-операторов.-Перегрузка-оператора-=-:-копирование,-перенос.-Перегрузка-операторов---,-*,-%5B%5D.-Перегрузка-операторов---,---.-Операторы-приведения-типов.)

**Обработка исключительных ситуаций в С++.**

Недостатки обработки ошибок в структурном программировании

Недостатки:

1. Если где-то возникает ошибка в коде, мы вынуждены "протащить" ее через все уровни абстракции/иерархии до того места, пока мы не сможем обработать эту ошибку.
2. Весь код насыщен непрерывными проверками. Обработка ошибки совмещена вместе с кодом.

Идея в том, чтобы при возникновении какой-либо ошибки передавать управление непосредственно в то место, где мы можем обработать ее, не "протаскивая" ее через много уровней.

Обработка ошибок в C++

Инструкции обработки ошибок:

* Инструкция try - заворачиваем в неё блок кода, в котором может произойти ошибка, и берём его под контроль.
* Если в блоке try возникает исключительная ситуация, мы можем перейти на обработчик catch. Обработчики идут непосредственно после блока try.
* Генерируем исключительную ситуацию, используя инструкцию throw.

try

{

// код, который может упасть

// где-то произошёл throw <объект>

}

catch (<тип1>& <объект>) // если объект приводится к тип1, то обработает этот обработчик

{

// TODO

}

catch (<тип2>& <объект>) // если предыдущие обработчики не могут обработать,

{ // то передаётся исключение след. обработчику

// TODO

}

catch (...) // перехватит любую исключительную ситуацию (порядок обработчиков важен)

{

}

Если ошибка ни кем не перехватилась, то ПО падает.

*задачи обработчика*

1. Выдать сообщение (пользователю или в лог)
   1. Информация об ошибке (в лог файл пишем)
   2. Время, когда она произошла
   3. Где произошла
   4. Данные, которые привели к этой ошибке
2. Обработать ситуацию (по возможности)
3. Если ошибка критическая - корректно завершить программу (например, корректно закрыть связь с бд)

*плюсы*

1. Не протаскиваем ошибку
2. Всю обработку искл ситуации вынесли в одно место

**Проблемы с динамической памятью при обработке исключительных ситуаций.**

В C++, перейдя на обработчик, мы не можем вернуться в место возникновения ошибки (все временные объекты будут уничтожены). Это проблема.

Предположим, у класса А есть метод f(). Если мы динамически выделили память:

try

{

A \*obj = new A;

pobj->f();

delete obj;

}

catch(...)

{

// TODO

}

*Если при вызове метода f() возникает исключительная ситуация и мы выходим на какой-то из обработчиков, объект obj не удаляется. Происходит утечка памяти.*

Что происходит:

1. Возникает искл ситуация
2. Удаляются все временные объекты, которые создавались в блоке try
3. Выходим на обработчик

Возникает утечка дин памяти.

Бросать исключение из деструкторов не стоит. Непредсказуемый результат. Если же в нём возникает исключение, то в нём мы должны её обработать.

Ловля ошибки в разделе инициализации.

Array::Array(int q) try: mas(new double[q]), cnt(q)

{}

catch(const std::bad\_alloc& exc)

{ cout<<exc.what()<<endl; }

Решение проблемы с помощью noexcept или throw()

Можно запретить методу обрабатывать исключительную ситуацию.

Два варианта решения проблемы:

void A::f() noexcept // Первый способ

void A::f() throw() // Второй способ

С **noexcept** при возникновении исключительной ситуации вызывается функция terminate(). Функция terminate() приводит к тому, что будут вызываться все деструкторы только временных объектов в порядке, обратном их созданию.

Со **throw()** результат непредсказуем, это старый синтаксис, который лучше не использовать.

noexcept без параметров аналогичен noexcept(True) - это говорит о том, что данный метод не должен обрабатывать исключительную ситуацию.

Если пишем noexcept(False) или throw(...), то этот метод может обрабатывать все исключительные ситуации, как и в случае если ничего не пишем.

Использование exception при обработке ошибок

Наша задача - cделать так, чтобы при модификации мы не исправляли написанный код. Хочется, чтобы при работе с ошибками был такой же подход. При модификации могут возникать новые ошибки и мы должных обрабатывать их, не изменяя написанный код. То есть, те обработчики, которые у нас написаны, должны обрабатывать новые ситуации.

Мы можем использовать механизм полиморфности. Дело в том, что обработчик перехватит исключительную ситуацию, если тип объекта является производным от заявленного в обработчике типа или им самим.

Таким образом, мы можем задавать базовый класс как ситуацию, а конкретная ситуация будет производной от этого типа.

Для универсальности и распространения на всю программы нам предоставляется базовый класс std::exception, от него можно порождать свои классы.

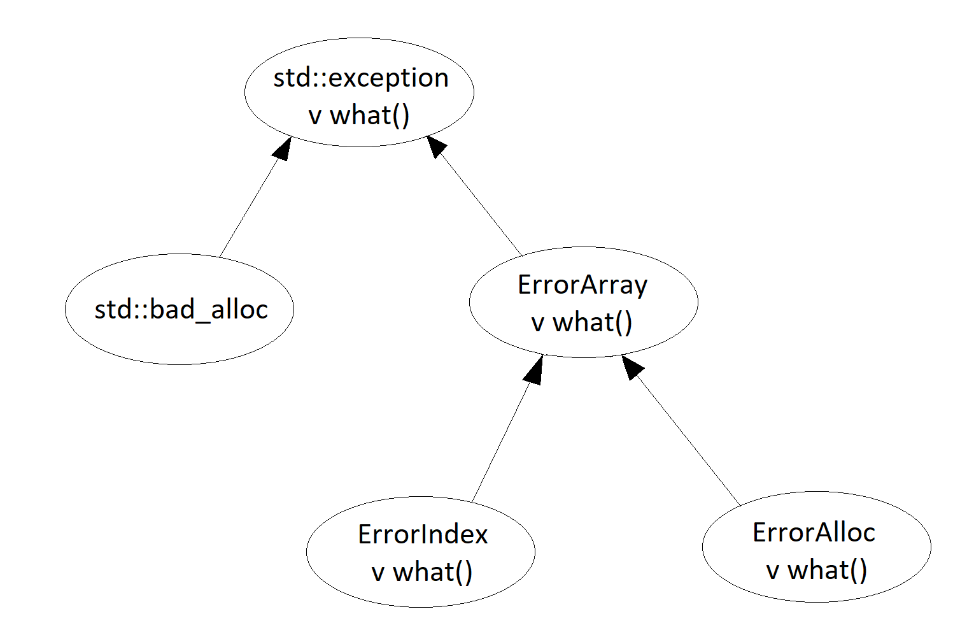
Для использования мы должны подключить заголовочный файл:

#include <exception>

Идея exception

У нас есть базовый класс std::exception. Этот базовый класс нам представляет виртуальный метод what(), возвращающий строку message. Стандартные ошибки являются производными от этого класса. Производные классы могут подменять метод what().

Например, есть стандартная ошибка bad\_alloc - ошибка, связанная с выделением памяти.



И да, мы свои классы можем тоже порождать от этого базового класса! Пусть у нас есть класс Array, мы для него хотим создать объекты, которые отвечают за определенные ситуации. Назовём класс ErrorArray. От него уже будем порождать конкретные ошибки: некорректный индекс - ErrorIndex, ErrorAlloc (перехватываем bad\_alloc на себя).

Любую ошибку с нашим классом мы можем перехватить. На уровне класса exp мы можем перехватить любую ошибку, связанную с нашим массивом.

Для нашего ПО таких уровней перехвата ошибок может быть много. Удобно модифицировать. Модифицируя наш класс, мы добавим новую ошибку, но она будет перехватываться на уровне базового класса, отвечающего за ошибки, связанные с объектом нашего класса.

Преимущества использования обработки исключительных ситуаций

Плюсы использования обработки исключительных ситуаций:

* Не нужно прокидывать ошибку через много уровней. Мы сразу переходим в обработчик.
* Мы разделяем саму логику (код нашей задачи) от обработки исключительных ситуаций, вынося обработчики отдельно, занося их в методы what().
* можем легко развивать ПО и модифицировать.

**Перегрузка операторов в С++. - это удобный механизм, но он не относится к объектно-ориентированной части. Эта вещь уже была реализована в необъектных языках.**

Шесть операторов, которые перегружать нельзя

1. Оператор . - доступ к члену объекта. Если мы перегрузим этот оператор, мы не сможем вызвать для объекта ни одного метода!
2. Оператор .\* - указатель на метод.
3. Оператор :: - оператор доступа к контексту. Он применяется для доступа к членам через имя класса или для доступа к глобальному контексту.
4. Оператор ? : - тернарный оператор. Разработчики просто не смогли придумать, как перегрузить этот оператор. Страуструп не пришел ни к одному из решений.
5. Оператор sizeof - определение размера объекта. Если мы перегрузим, мы такое вытворим в программе!
6. Оператор typeid - возвращает id типа объекта. Если мы перегрузим, мы не сможем идентифицировать объект и понять, какого он типа.

**Операторы .\* и ->\***

С++ добавляет два интересных оператора. Напомним, что оператор .\* перегружать нельзя, а оператор ->\* можно.

Посмотрим на очень интересный пример с функциями:

void f(); // Определили функцию f()

void (\*pf)(); // Определили указатель на функцию

pf = f; // Этот указатель инициализируем адресом функции(так как имя любой функции - это ее адрес в памяти)

pf(); // Через указатель на функцию вызываем функцию

Примечание: оператор () - оператор разыменования - вызов функции по адресу. Так же вызвать функцию f() можно и таким образом: (\*pf)() - синтаксис позволяет.

Что касается методов класса:

void A::f(); // Метод класса A

void (A::\*pf)(); // Указатель на метод класса A

// Хотелось бы проинициализировать этот указатель.

// pf = A::f; - Если мы таким образом напишем, мы получим не адрес этого метода

// Метод не находится в классе. Он вызывается по указателю, и чтобы получить этот адрес,

// было принято решение добавить вот такой синтаксис

pf = &A::f; // Вычисление адреса метода.

A obj;

(obj.\*pf)(); // Чтобы вызвать метод через указатель, используется оператор .\*

// Этот указатель имеет более низкий приоритет, чем (),

// поэтому, чтобы использовать (),

// надо повысить его приоритет, взяв obj.\*pf в круглые скобки.

A\* p = &obj;

(p->\*pf)(); // Оператор ->\* используется для указателя на объект

// В метод, на который указывавет этот указатель, будет передаваться

// указатель на объект.

Таким образом, мы разделяем вызов функции и вызов метода. Если мы вызываем метод класса через указатель для объекта, используется оператор .\*, а если работаем с указателем на объект, используется оператор ->\*.

Оператор ->\* перегружается как член класса и является бинарным (\*this и указатель на метод)

**Правила перегрузки операторов.**

1. Операторы, которые можно перегрузить только как члены классов:
   1. Оператор = - оператор присваивания (бинарный)
   2. Оператор() - функтуатор (бинарный)
   3. Оператор [] - индексация (бинарный)
   4. Оператор -> - унарный
   5. Оператор ->\* - бинарный, так как принимает указатель на метод и объект, метод которого вызываем
2. **Бинарные операторы** можно перегружать как члены класса или как внешние функции-операторы. Это зависит от ситуации. Конечно, надо отдавать предпочтение члену класса. Если мы перегружаем бинарный оператор, как член класса, он принимает 1 параметр (второй параметр он принимает неявно - \*this).
3. **Унарные операторы** перегружаем как члены класса.

**Перегрузка оператора = : копирование, перенос.**

Очевидное неочевидное: ***Если мы перегружаем оператор, то этот оператор не наследуется. Если он будет наследоваться, то начнется абсурд. В частности если он будет наследоваться, то производный класс будет вызвает оператор присваивания базового и произойдет неполное копирование объекта.***

Если мы динамически выделяем память под члены класса, мы обязаны явно определить оператор присваивания или запретить. Дело в том, что для любого типа неявно определяется оператор присваивания, который побайтно копирует данные. Может выйти так, что два объекта указывают на одну область памяти. В большинстве случаев это не нужно.

Разница оператора присваивания с копированием - создает копию объекта, а оператор присваивания с переносом - захватывает временный объект.

Копирование - выделяем новую память и копируем из одной области в другую, а при переносе захватываем область того объекта, который получаем. Параметр обнуляем, чтобы при деструкторе не произошло удаления области памяти, которой мы захватили.

**Перегрузка операторов ++, --.**

Идея: отделить постфиксную от префиксной записи.

Решение: унарный - префиксный, бинарный - постфиксный операторы.

Замечание: ***если мы бинарный не перегружаем, то и для постфиксного и префиксного будет вызываться перегруженный унарный оператор. Иначе мы четко разделяем их.***

// префиксная - унарный

Index& operator++() // ++obj

{

++ind;

return \*this;

}

// постфиксная - бинарный

Index operator++(int) // obj++

{

Index it(\*this); // временный объект с копией

++ind;

return it; // вернуть копию

}

operator int() const { return ind; } // Мы так же можем определить оператор приведения типа.

// оператор тип() и что мы возвращаем. В данном случае происходит неявное приведение типа int

**Операторы приведения типов.**

Если мы не хотим чтобы не было возможности неявно вызвать оператор, то точно так же как для конструктора записывается модификатор explicit перед оператором приведения типа. (выше пример)

### Перегрузка операторов new, delete.

Вот тут вот Тассов знаменитым жестом отпивает чай из бумажного стаканчика.

new и delete перегружаются как статические члены класса т.е. не для объкта т.е. они не принимают \*this. Можно перегрузить как глобальный, но это приведет к тому чо мы не сможем использовать глобально. Если перегружаем их, то для конкретного класса. Есть разные виды перегрузки new

*Важно! Глобальный вызов - ::operator new, ::operator delete*

## [Шаблоны функций и классов в С++. Параметры шаблонов. Специализация шаблонов частичная и полная. Параметры шаблона задаваемые по умолчанию. Шаблоны с переменным числом параметров. Пространства имен.](https://github.com/RusinovaDaria/examOOP/wiki/8.-Шаблоны-функций-и-классов-в-С--.-Параметры-шаблонов.-Специализация-шаблонов-частичная-и-полная.-Параметры-шаблона-задаваемые-по-умолчанию.-Шаблоны-с-переменным-числом-параметров.-Пространства-имен.)

**Шаблоны функций и классов в С++**

Если есть функции, которые выполняют одни и те же действия, но для разных типов данных, то их можно шаблонизировать.

В С надо либо плодить однотипные функции, что плохо по определению, либо использовать макросы с параметрами, что опасно, так как подстановка идет на этапе компиляции, а проверка - на этапе выполнения.

В С++ эта проблема решается шаблонами.

Шаблоны решают проблему копирования кода

Шаблон может быть для функции, класса, метода класса, своего типа. Во время компиляции будет подстановка параметров шаблона с проверкой шаблона.

Шаблон не является ни классом ни функцией. Функция и класс генерируется на основе шаблона во время использования.

Компилятор встречает вызов шаблона функции (создание класса) - смотрит, может ли он быть использован и если может, то по шаблону создается функция(класс).

**Шаблон** не является ни классом, ни функцией. Функция или класс генерируется на основе шаблона во время использования. Компилятор встречает вызов функции, смотрит, может ли использоваться шаблон, и, если может, создаёт по шаблону функцию или класс.

Синтаксис шаблона в общем виде:

template<[параметры шаблона]>

функция | класс | тип

**Параметры шаблонов**

Параметрами шаблона могут быть:

* Типы. Параметрами типа могут быть простые типы языка си, производные типы языка си и классы.
* Параметры значений. Параметры значения - только константные параметры целого типа или указатели с внешним связыванием.

Для параметра типа изначально использовалось ключевое слово class, но программисты стали возражать, так как в С++ простые типы изначально не были классами, и разработчик языка добавил еще одно ключевое слово - typename. Разницы между описанием class и typename нет, это сделано для того, чтобы лучше читался код, чтобы было понимание, что это тип.

**Специализация шаблонов частичная и полная**

Специализация может быть как частичная, так и полная.

Полная специализация

В данном случае описана полная специализация шаблона класса А:

template <typename T>

class A {...};

template <>

class A<float> {...};

Специализация является тоже шаблоном.

*По специализации также создается класс, если нужно*. У нас создаются классы по специализации. Специализация может иметь отличные параметры от шаблонов, вернее тело специализации может быть другим: другие члены, данные, методы, отличные от самого шаблоны.

Пример полной специализации

У нас есть шаблон класса А и специализация этого класса А. В специализации класса интерфейс, отличный от шаблона.

template <typename Type>

class A

{

public:

A() { cout << "constructor of template A;" << endl; }

void f() { cout << "metod f of template A;" << endl; }

};

template<>

void A<int>::f() { cout << "specialization of metod f of template A;" << endl;}

template <>

class A<float>

{

public:

A() { cout << "specialization constructor template A;" << endl; }

void f() { cout << "metod f specialization template A;" << endl; }

void g() { cout << "metod g specialization template A;" << endl; }

};

int main()

{

A<double> obj1;

obj1.f();

A<float> obj2;

obj2.f();

obj2.g();

A<int> obj3;

obj3.f();

return 0;

}

Частичная специализация

**Частичная специализация** - когда мы указываем не все значения параметра шаблона.

При частичной специализации шаблонов возможна неоднозначность при вызове. Здесь аналогичный подход с функциями: сначала идет выбор специализации, если невозможно создать класс по специализации, создается класс по шаблону.

Пример частичной специализации

У нас есть класс с двумя параметрами шаблонов. У шаблона класса два параметра class A<T, T>, но эти два параметра одного типа template <typename T>. Мы можем частично специализировать этот шаблон.

// Второй параметр - по умолчанию.

// То есть если компилятор генерит класс по шаблону, а в параметры передаётся

// только один параметр, второй будет приниматься за double.

template <typename T1, typename T2 = double>

class A

{

public:

A() { cout << "constructor of template A<T1, T2>;" << endl; }

};

// Частичная специализация, когда два параметра равны

template <typename T>

class A<T, T>

{

public:

A() { cout << "constructor of template A<T, T>;" << endl; }

};

// Частичная специализация, второй параметр - int

template <typename T>

class A<T, int>

{

public:

A() { cout << "constructor of template A<T, int>;" << endl; }

};

// Частичная специализация, когда два параметра, но они

// имеют конкретный тип, производный от параметров шаблона

template <typename T1, typename T2>

class A<T1\*, T2\*>

{

public:

A() { cout << "constructor of template A<T1\*, T2\*>;" << endl; }

};

int main()

{

A<int> a0; // Специализация, когда два параметра равны

A<int, float> a1; // Нет соответствующей специализации, класс будет создаваться по шаблону

A<float, float> a2; // Специализация, когда два параметра равны

A<float, int> a3; // Специализация со значением int

A<int\*, float\*> a4; // Специализация с двумя указателями

// A<int, int> a5; // Error!!! Неоднозначность. Мы можем создать класс

// A<int\*, int\*> a6; // по двум разным специализациям

// У этих специализаций нет приоритета

}

**Параметры шаблонов задаваемые по умолчанию**

Так же, как при определении функций, параметры шаблона могут быть **по умолчанию.**

В случае выше сам шаблон имеет один параметр по умолчанию - double. Если мы передаем только один параметр, будет вызываться этот шаблон (а второй параметр по умолчанию типа double):

template <typename T1, typename T2 = double>

class A

{

public:

A() { cout << "constructor of template A<T1, T2>;" << endl; }

};

**Шаблоны с переменным числом параметров**

Так же как функции, шаблоны мы можем создавать с переменным числом параметров. Это могут быть шаблоны функций, методов и классов.

Когда удобно использовать

Если мы рассматриваем для классов, чтобы в классах были инициализаторы с несколькими параметрами. В принципе, можно использовать initializer\_list, если параметры одного типа. А если разного?

И, если говорить о классах, используются так называемые кортежи, когда мы формируем какой-то тип из полей разных типов.

В python кортеж - альтернатива и массиву, и структуре. По существу, мы можем рассматривать как массив с элементами разного типа. А зачем это нужно? Мы не всегда знаем то данное, которое нам нужно возвращать, какие параметры могут входить в это данное. Мы можем возвращать список данных, формировать структуру, причем можем, конечно, формировать список данных разного типа. Кортеж - хорошая альтернатива, когда мы объединяем по надобности параметры разного типа в одно данное.

Пример шаблона функции с переменным числом параметров

Как для функции, так и для классов, будет создаваться набор функций с разным количеством параметров.

В данном примере вызываем функцию sum() и передаем в неё 5 параметров. По шаблону, будет вызываться функция с 5 параметрами. Но эта функция с 5 параметрами вызывает функцию sum(), в которую мы передаем 4 параметра. Будет вызываться функция с 4 параметрами, потом с 3, с 2... тут мы должны поставить шаблон/специализацию/перегрузку функций, которые ограничат нам формирование функций по шаблону. В данном случае с одним параметром.

template <typename Type>

Type sum(Type value)

{

return value;

}

template <typename Type, typename ...Args>

Type sum(Type value, Args... args)

{

return value + sum(args...);

}

int main()

{

cout << sum(1, 2, 3, 4, 5) << endl;

return 0;

}

Обязательно нужно поставить ограничение на создание функций по шаблону! В примере выше это реализовано за счёт шаблона с одним параметром.

Кортежи используются крайне редко.

С кортежами существуют две проблемы:

1. Рассматривать кортеж как объект нельзя, так как разные типы и определить перегруженные операции над ним невозможно.
2. Чёткое местоположение каждого элемента - нужно помнить, какие типы лежат внутри кортежа.

**ПРостранства имен**

Программа растет, она становится большой. Во время разрастания программы может возникнуть **проблема конфликта имён**.

Мы используем разные библиотеки, в одной библиотеке нужно что-то взять, в другой что-то взять... у некоторых библиотек может дублироваться функционал. Это типичная ситуация. Например, несколько реализаций функции с именем swap в различных библиотеках. Когда подключаем разные библиотеки, может возникнуть конфликт имен. Из какой библиотеки мы вызываем swap? Это так же может касаться имен классов, методов, которые мы используем.

Эту проблему пытались решить раньше. Например, в Fortran были так называемые "общие блоки памяти", когда мы использовали идентификатор, и помечали, из какого блока памяти мы его используем, чтобы решить этот конфликт.

В C++ мы можем задавать пространства имён.

Синтаксис такой:

namespace <имя>

{

<блок пространства имён>

}

// Доступ к пространству имён

<имя>::f(); // f() - член пространства имен

Или можно сделать так:

// или можно включить это пространство и использовать f

using namespace <имя>;

f(); // но таким образом можно вернуться к изначальной проблеме

В примере выше тоже можно натолкнуться на проблему конфликта имён, так как подключив несколько namespace, может возникнуть та же самая ситуация. Поэтому, когда у нас есть много пространств имён с пересекающимися параметрами, лучше использовать синтаксис :: - это поможет избежать конфликта.

Пространствами имён злоупотреблять не надо. Вложенных пространств имён надо избегать или сводить к минимуму.

Имя пространства имён может отсутствовать - это **анонимные пространства имён**. Их особенность в том, что из другого файла нельзя получить доступ к членам анонимного пространства имён.

Грубо говоря, если у нас часть, и мы не хотим, чтобы она была видна из других частей, мы можем определить это, как анонимное пространство имен.

## [Приведение типа в С++: static\_cast, dynamic\_cast, const\_cast, reinterpret\_cast. «Умные указатели» в С++: unique\_ptr, shared\_ptr, weak\_ptr. Контейнерные классы и итераторы. Работа с итераторами. Цикл for для работы с контейнерными объектами.](https://github.com/RusinovaDaria/examOOP/wiki/9.-Приведение-типа-в-С--:-static_cast,-dynamic_cast,-const_cast,-reinterpret_cast.-«Умные-указатели»-в-С--:-unique_ptr,-shared_ptr,-weak_ptr.-Контейнерные-классы-и-итераторы.-Работа-с-итераторами.-Цикл-for-для-работы-с-контейнерными-объектами.)

**Приведение типа** static\_cast, dynamic\_cast, const\_cast, reinterpret\_cast

**Приведение типов в Си**

В языке Си приведение типов: (<абстр. описатель>)<выражение>. Когда создавался язык Си, все говорили, что это великолепно. Когда мы указываем такое приведение типов, компилятор, не задумываясь об их совместимости, осуществляет такое преобразование. Что мы получим в результате - неизвесто. Для каких-то задач, например, системного применения, это можно использовать, это может быть даже удобно (разложить структуру по байтам и привести ее к массиву байтов). В общем случае, такое приведение не безопасно. Компилятор не предупреждает нас, возможно это или не возможно.

Пример (как правило, это не работает):

float A[10][10];

float\*\* P = (float\*\*) A; // это чистый бандитизм

// Каждый компилятор осуществит преобразование по-своему

// P - указатель на указатель на float, а A - адрес первой строки последовательности элементов.

// двойное разыменование для P = неопределённость.

В языке C++ решили использовать разные варианты приведения типов.

По умолчанию всегда указатель на производный класс приводится к указателю на базовый класс (или ссылка). А если нужно обратное преобразование?

Для обратного преобразования появились два оператора преобразования:

* Первый оператор преобразования выполняется на этапе компиляции - static\_cast
* Второй оператор преобразования на этапе выполнения - dynamic\_cast.

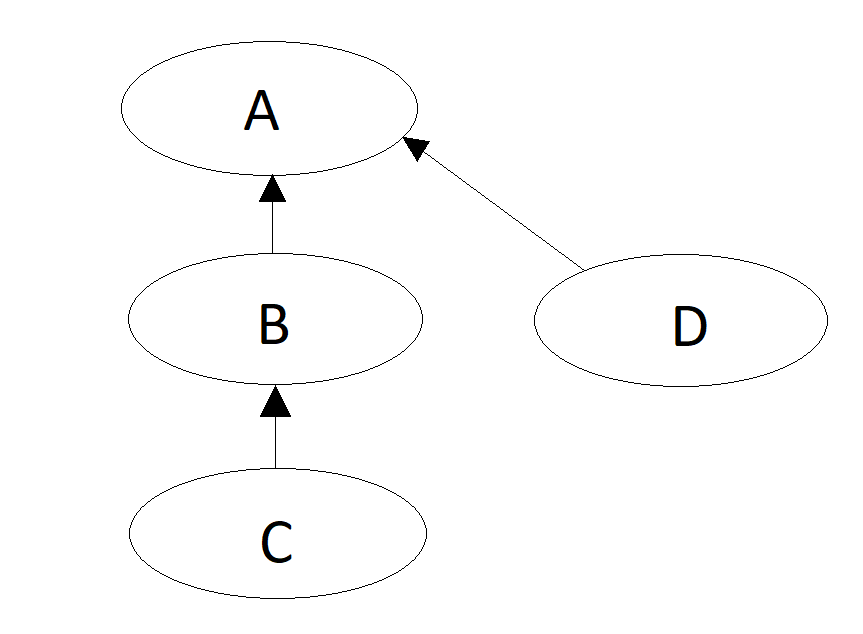
Для использование всех этих указателей надо

#include <memory>

***static\_cast***

*На этапе компиляции выполняется оператор static\_cast. Этот оператор используется для приведения родственных классов, находящихся по одной ветви наследования. Также используется для стандартных типов для которых определён механизм явного приведения.*

Рассмотрим следующую иерархию:



С помощью static\_cast мы можем от указателя A привестись к указателю B или к указателю на класс C, или от A к D. Он не позволит нам привестись от класса указателя на B к D. Они родственные, но находятся по разным ветвям.

**Проблема - это выполняется на этапе компиляции**. На этапе компиляции невозможно проверить, что это за объект, то есть приведение будет срабатывать, но как….

***dynamic\_cast***

Оператор dynamic\_cast делает проверку на этапе выполнения. Он приводит к типу, если реально указатель указывает на объект этого типа. Если нет - возвращает указатель на NULL. Приведение может быть не только указателем, но и ссылкой.

**Для dynamic\_cast есть жесткое требование - базовый классы должен быть полиморфными (то есть либо virtual метод, либо virtual деструктор).**

Пример:

pb = dynamic\_cast<B\*>(pa);

if (pb)

**Можно проверить**: если pb не равно нулю, приведение осуществилось, иначе не осуществилось. В данном случае всё будет нормально, так как pa указывает на объект класса B - приведение возможно. А приведение, например, к указателю на класс С невозможно из pa, так как pa указывает на объект другого класса.

Такое приведение типов удобно тем, что это удобно на этапе выполнение программ.

Если работаем со ссылкой в dynamic\_cast - вместо NULL возникает исключение bad\_cast, которое можно отловить. Исключения ловить неприятно, поэтому лучше работать с указателями.

***const\_cast***

Мы работаем с модификатором const. Мы контролируем, что объект может быть константными, контролируем методы.

Но есть проблема - **мы не можем менять поля константных объектов.** Иногда возникают ситуации, когда нам необходимо менять...

Предположим, у нас есть объект, который держит указатель на другой. Мы определили его, как константный, но этот указатель мы хотим отобрать от него. Чтобы отобрать, нам нужно это поле обнулить. А сделать это мы не можем, так как не можем обнулять поля константных объектов.

Чтобы убрать модификатор const, используется оператор **const\_cast**. Есть компиляторы, которые не позволяют изменять константность объектов. В MSVC такая возможность имеется.

class A

{

int a = 0;

public:

virtual ~A() = 0;

void f() { cout << "method f class A:"<< a << endl; }

};

A::~A() {}

class B : public A

{

int b = 1;

public:

void f() { cout << "method f class B;" << b << endl; }

void g1() { cout << "method g1 class B;" << endl; }

};

class C : public B

{

int c = 2;

public:

void f() { cout << "method f class C;" << c << endl; }

void g2() { cout << "method g2 class B;" << endl; }

};

class D : public A

{

int d = 3;

public:

void f() { cout << "method f class D;" << d << endl; }

};

int main()

{

A\* pa = new B;

B\* pb = static\_cast<B\*>(pa);; // Всё компилируется, но есть проблема. Так как выполняется на этапе компиляции, приведение будет срабатывать, но реально pb может не указывать на обьект класса B.

pb->f();

C\* pc = static\_cast<C\*>(pa); // Всё компилируется. Указатель на обьект класса C, но реально - на обьект класса B. Что будет, НЕИЗВЕСТНО.

pc->f();

D\* pd = static\_cast<D\*>(pa);

pd->f();

pb = dynamic\_cast<B\*>(pa);

if (!pb)

{

cout << "Error bad cast!" << endl;

}

else

{

pb->f();

pb->g1();

}

pc = dynamic\_cast<C\*>(pa); // НЕВОЗМОЖНО, т. к. pa -> обьект класса B

if (!pc)

{

cout << "Error bad cast!" << endl;

}

else

{

pc->f();

pc->g2();

}

const B obj;

const B\* p = &obj;

const\_cast<B\*>(p)->f();// У класса B метод f() не константный, то есть мы можем вызвать этот метод,

// если объект не константный

}

***reinterpret\_cast***

Также существует оператор, **эквивалентный С-му приведению -** reinterpret\_cast. Может приводить из любого типа.

Пример:

class A {...};

A\* p = new A;

char\* pbyte = reinterpret\_cast<char\*>(p); // Можем выполнить такой бандитизм

Мы поставили указатель типа char на первый байт объекта класса А.

Это то же самое, как преобразование, которое было в языке Си. Небезопасное преобразование. Неизвестно, к чему это приведет.

**Умные указатели**

**Проблема - утечка памяти.** Потеря указателя на объект. Не удаление объекта. Приводит к \Утечке памяти/

**Проблема - висящий указатель**. Указатель на объект, которого нет. Приводит к крашу системы.

Допустим на один объект держат ссылки разные объекты. **Проблема контроля за валидностью. (при копировании…. и переносе)**

Умные указатели решают проблемы с утечкой памяти и с висящим указателем. Первоначально эту проблему пытались решить одним указателем, но скоро поняли, что одним указателем решить проблему невозможно...

***unique\_ptr***

unique\_ptr - жестко держит один объект и не дает во владение другим. Отсутствует конструктор копирования и оператор присваивания. Может рассматриваться как временный объект, поэтому определяется конструктор переноса.

Есть метод get(), отдающий указатель. Если удалить из под одданого указателя память, то получится висящий unique\_ptr

Есть метод release(). Перестает владеть объектом и отдает указатель.

Метод reset() - сброс владения или замена на другой объект.

unique\_ptr - занимает мало места в памяти, хранится только указатель.

Простейшая оболочка над указателям. **Основная проблема, которую он решает - обработка исключительных ситуаций. Он освобождает память при выходе из области видимости.**

Подробнее про unique\_ptr. Мы можем работать не с одним объектом, а с несколькими объектами - оператор **[]. Есть проблема - мы не знаем, сколько объектов хранит unique\_ptr, и с помощью unique\_ptr мы это определить не можем. Должно быть данное, которое (если unique\_ptr хранит, например, массив объектов) мы должны тащить вместе с unique\_ptr. Проблема. Решение - можно сделать еще одну обертку, передавая unique\_ptr и количество элементов.**

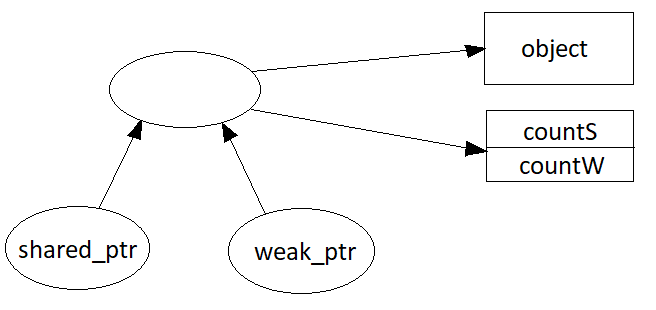
Указатель shared\_ptr содержит счетчик. Если shared\_ptr обеспечивает нас счетчиком, это так называемое совместное владение, то а паре с ним идет weak\_ptr - слабое владение. Этот указатель не отвечает за освобождение памяти из под объекта. Он может только проверить, есть объект или его нет. Эти два указателя связаны между собой.

У нас должен быть счетчик *countS*, определяющий, сколько объектов указывают на сберегаемый объект. Указателю weak\_ptr тоже надо знать об этом счетчике.

Пусть есть какой-то базовый класс, от которого порождаем два класса: shared\_ptr и weak\_ptr. И тому и другому нужен указатель на *object* и нужен счетчик *countS*. Базовый класс содержит указатель на объект, и счетчик *countS* тоже должен быть доступен всем shared\_ptr и weak\_ptr, следовательно, счетчик *countS* мы тоже должны вынести, как объект.

Так как память *object* вынесли, по счетчику *countS* weak\_ptr определяет, есть ли этот *object* или нет. Если счетчик равен нулю, то объекта нет. Когда создается новый shared\_ptr на область памяти *object*, счетчик *countS* увеличивается. Удаляется shared\_ptr - счетчик уменьшается. Если счетчик равен нулю - эта память должна быть освобождена.

А что будет отвечать за память счетчика *countS*, когда освободится память из под *object*? Здесь встает необходимость считать не только количество объектов shared\_ptr , но и количество объектов weak\_ptr. То есть, по существу у нас не один счетчик, а два. Счетчик weak\_ptr нужен для того, что если он станет равен нулю, и второй счетчик равен нулю, освободить эту память. Соответственно, общий класс для shared\_ptr и weak\_ptr может решать эту проблему. Он будет контролировать и память объектов, и область счетчика.



В общем, суть в том, что мы не можем удалить счетчик *countS*, так как он нужен weak\_ptr, чтобы понять, есть объект, или нет.

Мы можем удалить счетчик только в том случае, когда количество weak\_ptr + количество shared\_ptr будет равняться 0.

***shared\_ptr***

shared\_ptr - используется, если несколько объектов указывают на один объект.

Держит в себе помимо указателя кол-во ссылок (shared\_ptr'ов) на объект (совместное владение). Как только кол-во ссылок = 0, объект освобождается.

Методы для shared\_ptr

* get(), как у unique\_ptr
* reset() - уменьшает счетчик sharedov и становится nullptr
* use\_count() - кол-во sharedov
* unique() - true, если shared\_ptr один

Подробнее про shared\_ptr. Появляются методы, связанные с совместным владением:

* нам нужно знать количество указателей shared\_ptr на объект - метод use\_count.
* метод, который говорит, один shared\_ptr или нет - unique, возвращающий true, если всего один shared\_ptr и false в противном случае.

***weak\_ptr***

week\_ptr(слабое владение) идет в паре с shared\_ptr. Он не отвечает за освобождение памяти. Можно только проверить - есть объект или нет. Для использования weak\_ptr нужно сделать из weak\_ptr - shared\_ptr (метод lock());

Методы для weak\_ptr

* reset(),
* use\_count() - кол-во weakov,
* expired() - есть объект или нет.
* lock() - сделать из weak\_ptr shared\_ptr.

Они связаны, так как оба имеют доступ к счетчику ссылок на объект. ПОВОРОТ. Счетчиков два - один для weak\_ptr'ов, второй для shared\_ptr'ов. Объект удаляется, когда оба из них становятся 0.

shared\_ptr и weak\_ptr образованы от общей базы, которая держит счетчик sharedov и weakov

Если мы начнем через него работать с объектом, то возможно, что объект будет удален во время работы, а нам бы этого не хотелось.   
Поэтому было принято решение, что на основе weak\_ptr будет создаваться shared\_ptr, который будет захватывать объект, увеличивать счетчик. Когда нам нужно поработать с объектом, на который указывает weak\_ptr, то на его основе создаем shared\_ptr с помощью метода lock, работаем и удаляем shared\_ptr. Непосредственного доступа к объекту через weak\_ptr нет.

Метод expired говорит, есть объект, или нет (у нас же слабое владение). Проверяется счетчик countS для shared\_ptr, если он равен 0, то есть указателей shared\_ptr на объект нет, возвращается true, иначе false.



**Контейнерные классы и итераторы, работа с итераторами**

Контейнерный класс - класс, содержащий объекты другого класса.

Итератор - класс, отвечающий за просмотр содержимого других объектов.

Общий механизм работы итераторорв.

Есть стандартный шаблон iterator, у которого есть специализации под разные виды работы с итераторами.

Есть стандартный итератор - шаблон класса Iterator. У него есть специализации под разные виды работы с итераторами. **Итераторы делятся на итераторы ввода** (мы можем менять то, на что итератор указывает) и **итераторы вывода** (мы НЕ можем менять то, на что итератор указывает, то есть мы можем только читать, но не записывать).

Итератор может рассматривать контейнер как **направленную** последовательность, **двунаправленную** и последовательность **произвольного доступа.**

*С помощью итератора просматриваем содержимое контейнера.*

Мы должны иметь возможность сравнивать итераторы и получать доступ к данным, на которые он указывает (операторы \*, ->) В каждом итераторе надо минимум реализовать сравнение итераторов (!=), доступ к данным (\*, ->, bool) и увеличение итератора (++).

У контейнерного класса должны быть методы begin() и end(), возвращающие итератор на начало и конец, для универсального просмотра контейнерного класса.

Если они реализован, то можно применять цикл foreach:

for (<тип>& <имя> : <container>)

{}

Также могут быть реализованы в контейнере методы begin() const, end() const и cbegin(), cend()

возвращающие константный итератор для константного объекта и константный для неконстантного соответственно

**Цикл for для работы с контейнерными объектами**

выше….

# ООР

## [Структурное программирование: нисходящая разработка, использование базовых логических структур, сквозной структурный контроль](https://github.com/RusinovaDaria/examOOP/wiki/№-10.-Структурное-программирование:-нисходящая-разработка,-использование-базовых-логических-структур,-сквозной-структурный-контроль.)

## Дейкстра, Милдс выделили три идеи структурного программирования:

## нисходящая разработка

## использование базовых логических структур

## сквозной структурный контроль

## 

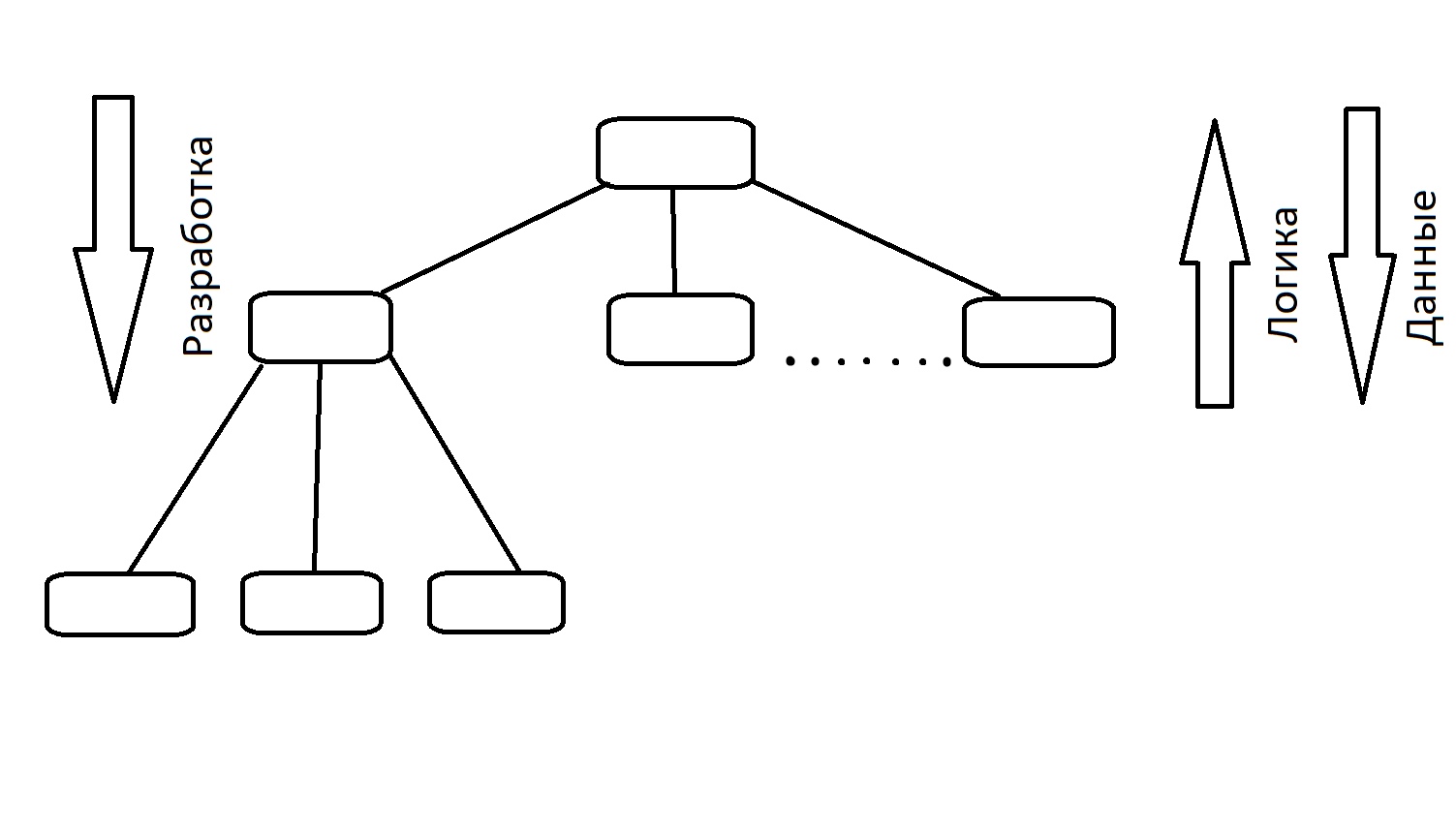
**нисходящая разработка**

Этапы создание программного продукта:

1. Анализ. (Оцениваем задачу, переработка ТЗ)
2. проектирование
3. кодирование
4. тестирование
5. сопровождение
6. модификация

2-4 используют нисходящий подход.

В нисходящей разработке используются **алгоритмы декомпозиции** – разбиение задачи на подзадач (из принципа "разделяй и властвуй"). Выделенные подзадачи разбиваются дальше на подзадачи. Таким образом, формируется иерархическая структура: данные нисходящие, логика восходящая, разработка нисходящая (см. рисунок ниже).



Нисходящая разработка используется в трех этапах разработки (проектировании, кодировании, тестировании).

*Правила структурного программирования:*

* Данные на низком уровне, на высшем логика.
* Для каждой полученной подзадачи создаем отладочный модуль. Готовятся тестирующие пакеты (до этапа кодирования). Принцип полного недоверия к данным.
* Возврат результата наверх и анализ последующего результата там.
* Явная передача данных через список параметров (не более 3х).
* Функция может возвращать не более одного параметра. Не более 7 подзадач у задачи.
* Глубина вложенности конструкций - не больше трёх.
* Иерархия уровня абстракции должна соответствовать иерархии данных [Нельзя работать с полями полей структур].
* Чем больше уровней абстракции, тем лучше.

*Принципы работы с кодом:*

1. Сегментирование (функция разбивается на логические куски).
2. Пошаговая реализация.
3. Вложенные конструкции (глубина не более 3х).

Заглушка - то, что должна выдавать функция при данных входных данных.

*Написание кода программы с использованием заглушек:*

Строится диаграмма иерархии алгоритма. Затем пишется код основной программы, в котором, вместо каждого блока диаграммы вставляется вызов подпрограммы, которая будет выполнять этот фрагмент. Вместо настоящих, работающих подпрограмм, в программу вставляются заглушки. Программа тестируется, после успешных тестов начинается написание подпрограмм с заглушками и их тестирование. На каждой стадии процесса реализации уже созданная программа должна правильно работать по отношению к более низкому уровню. Полученная программа проверяется и отлаживается. Разработка заканчивается тогда, когда не останется ни одной заглушки. Такая последовательность гарантирует, что на каждом этапе разработки программист одновременно имеет дело с обозримым и понятным ему множеством фрагментов, и может быть уверен, что общая структура всех более высоких уровней программы верна.

**использование базовых логических структур**

*Базовые структуры*

Майер: Любой алгоритм можно реализовать с помощью трех логических структур:

* Последовательность (однократное выполнение операций в том порядке, в котором они записаны)
* Ветвление (однократное выполнение одной из двух или более операций, в зависимости от выполнения заданного условия) [if, switch]
* Повторение (многократное исполнение одной и той же операции до тех пор, пока выполняется условие продолжения цикла) [while, until, for, loop - безусловный цикл]

Развилка (ветвление): выбор между двумя альтернативами, множественный выбор switch – ветви имеют const выражения.

Повторение: while, until , for, безусловный цикл loop

*Принципы структурного программирования:*

* Выход из цикла должен быть один.
* Не использовать оператора безусловного перехода goto.
* Любая программа строится из трёх базовых управляющих конструкций: последовательность, ветвление, цикл.
* В программе базовые управляющие конструкции могут быть вложены друг в друга произвольным образом.
* Повторяющиеся фрагменты программы можно оформить в виде подпрограмм (процедур и функций).
* Все перечисленные конструкции должны иметь один вход и один выход.

**сквозной структурный контроль**

IBM предложила идею организации контрольных сессий.

Суть контрольных сессий: перенос контролирующей функции на плечо самих программистов, чтобы они контролировали друг друга. Сколько было контрольных сессий - не влияет на оценку работы. А сколько недочетов найдут в коде - влияет. Участвуют только программисты (без руководства).

*Преимущества этой технологии:*

1. Самые серьезные логические ошибки исправляются на ранних стадиях разработки.
2. При таком подходе нет "кода в корзину".
3. Начиная с самых ранних стадий идет взаимодействие с заказчиком.
4. Объединение этапов кодирования, проектирования и тестирования (параллельно происходит).
5. Комплексная отладка - тесты пишутся до этапа проектирования на основе ТЗ.
6. Удобное распределение работы между программистами.
7. Из-за многоуровневой абстракции возникают естественные контрольные точки за наблюдением за проектом.
8. Локализация ошибок. (много уровней абстракции, легко выявить где)
9. Вероятность невыполнения проекта сводится к нулю.
10. Повторное исправление кода, выделяются библиотеки.
11. Плавное распределение ресурсов при разработке программного продукта. Нет аврала в конце проекта.

На начальном этапе используется иерархический подход (на этапе распределения ролей), а потом операционный(разработка).

Иерархический - порядок программирования и тестирования модулей определяется их расположением в схеме иерархии

Операционный - модули разрабатываются в порядке их выполнения при запуске готовой программы.

*Недостатки этой технологии:*

Сложно модифицировать код:

1. Понижение надежности за счет внесения изменений в написанный чужой код (плюс трата времени на разбор чужого кода).
2. Изменение данных, следовательно программа сыпется. Возникают моменты, когда легче написать свою программу с нуля.
3. Исключительные ситуации обрабатываются вперемешку с логикой кода - это приводит к большому количеству проверок и необходимости "протаскивать" ошибку чрез весь код до того места, где её можно будет обработать.

## [Преимущества и недостатки структурного и объектно-ориентированного программирования.](https://github.com/RusinovaDaria/examOOP/wiki/№-11-(2).-Преимущества-и-недостатки-структурного-и-объектно-ориентированного-программирования.)

#### **Преимущества и недостатки структурного программирования.**

* (+) Логически-связанные функции находятся визуально ближе, а слабо связанные — дальше, что позволяет обходиться без блок-схем и других графических форм изображения алгоритмов (по сути, сама программа является собственной блок-схемой).
* (+) Сильно упрощается процесс тестирования и отладки программ.
* (+) Нисходящая разработка даёт возможность на ранних этапах согласовывать с заказчиком прототип. В случае недовольства, придётся переписывать минимум кода.
* (+) Нисходящая декомпозиция позволяет совмещать проектирование и кодирование.
* (+) Начиная разработку с верхних уровней, избавляемся от логических ошибок.
* (-) Отсутствие гибкости системы. Сложно изменить формы данных и структур.
* (-) Исключительные ситуации обрабатываются вперемешку с логикой кода - это приводит к большому количеству проверок и необходимости "протаскивать" ошибку до того места, где ее можно будет обработать, через все уровни кода

#### **Преимущества и недостатки ООП.**

* (+) «Более естественная» декомпозиция программного обеспечения, которая существенно облегчает его разработку.
* (+) Сокращение количества межмодульных вызовов и уменьшение объемов информации, передаваемой между модулями.
* (+) Увеличивается показатель повторного использования код
* (+) Система более гибкая, код легче поддерживать. Эволюция ПО.
* (+) Можно обрабатывать ошибки отдельно от логики кода и сразу прокидывать их в место, где они могут быть обработаны
* (-) Производительность программ хуже.
* (-) Появление мертвого кода, кода "в корзину"
* (-) Легче совершить логическую ошибку

## [Основные понятия ООП: инкапсуляция, наследование, полиморфизм. Понятие объекта. Категории объектов. Отношения между объектами. Понятие класса. Отношения между классами. Понятие домена.](https://github.com/RusinovaDaria/examOOP/wiki/№-12-(3).-Основные-понятия-ООП:-инкапсуляция,-наследование,-полиморфизм.-Понятие-объекта.-Категории-объектов.-Отношения-между-объектами.-Понятие-класса.-Отношения-между-классами.-Понятие-домена.)

**Основные понятия ООП: инкапсуляция, наследование, полиморфизм**

Три слона, на которых держится наша плоская планета:

1. Инкапсуляция — объединение данных и методов, работающих с ними в классе, скрывая детали реализации от пользователя.
2. Наследование — это свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью. Класс, от которого производится наследование, называется базовым, родительским или суперклассом. Новый класс — потомком, наследником или производным классом.
3. Полиморфизм — это свойство системы использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта. Переопределение методов. Один интерфейс - много реализаций.

**Понятие объекта**

Объект - экземпляр класса. реализация абстрактного понятия, обладающего:

* Состояние - один из возможных вариантов существования объекта (спать, есть).
* Поведение - описание объектах в терминах изменения его состояния и передачи сообщений, данных под воздействием других объектов.
* Индивидуальность - сущность объекта, отличающая его от других объектов.

**Категория объектов**

1. Реальные объекты - абстракция реальных объектов физического мира: дом, стул, стол.
2. Роли - абстракция цели или назначения оборудования или человека.
3. Инциденты - абстракции чего-либо произошедшего или случившегося: наводнение, выборы.
4. Взаимодействие - объекты, полученные из отношения других объектов: перекресток, взятка, дом.
5. Спецификации - представление правил, стандартов, критерий, качеств: распределение занятий, правил дорожного движения.

**Отношения между объектами**

1. Отношение использования (старшинства) - каждый объект включается в такие отношения. Может играть 3 роли.
   * Воздействие - объект только воздействует на другие объекты, но сами воздействия не подтверждаются - Активные объекты.
   * Исполнение - объект подвержен воздействию другого объекта, но сам ни на кого не влияет. Пассивные объекты.
   * Посредничество - объект может и воздействовать и исполнять.
2. Отношение включения - один объект может включать другие объекты (делает объекты более зависимыми друг от друга).

**Понятие класса**

**Класс** - абстракция множества предметов, которые обладают одними и теми же характеристиками и подчиняются и согласовываются одними и теми же правилами и линиями поведения.

**Отношение между классами**

1. Наследование (на основе одного класса строится новый)
2. Использование (один класс вызывает методы другого)
3. Наполнение (один класс содержит другие)
4. Метаклассы (используется для создание других классов)

**Понятие домена**

**Домен** - отдельный реальный гипотетический мир, который наполнен отчетливым набором объектов, которые ведут себя в соответствии с определенными доменом правилами и линиями поведения. Домен - связанное единое целое. Программа разбивается на домены в процессе создания.Класс определяется в одном домене.

**Активный домен** - единое целое. Наличие класса в одном домене не требует наличия классов в другом.

## [Цикл разработки ПО с использованием ООП: анализ, проектирование, эволюция, модификация. Рабочие продукты объектно-ориентированного анализа.](https://github.com/RusinovaDaria/examOOP/wiki/№-13-(4).-Цикл-разработки-ПО-с-использованием-ООП:-анализ,-проектирование,-эволюция,-модификация.-Рабочие-продукты-объектно-ориентированного-анализа.)

**Цикл разработки ПО с использованием ООП: анализ, проектирование, эволюция, модификация**

ООП направлено на то, чтобы в дальнейшем легко модифицировать программу. В ООП этапы:

1. Этап анализа - построение модели задачи. Этап анализа проходит вместе с заказчиком. Анализ строится на основе физического мира.

Требования к модели:

* Полная
* Понятная всем заинтересованным лицам

В структурном программировании анализ - это формализация задач. В ООП на этап анализа приходится гораздо больше, это чуть ли не основной этап из всех этапов. Этап анализа - построение модели нашей задачи.

Этап анализа проходит вместе с заказчиком. Для нас, как программистов, это плюс - мы четко понимаем, что представляет из себя задача. Для заказчика это тоже плюс, так как на этом этапе он сам разбирается в модели той задачи, которую мы решаем.

Анализ строится на основе физического мира. Мы строим модель и как бы делаем "слепок" физического мира.

Требования к модели:

* Модель должна быть полная.
* Модель должна быть понятная всем заинтересованным лицам.

1. Этап проектирования. На современном уровне как правильно автоматизирован. Создаем проектные документы.

Этап проектирования - очень важный этап, так как нам нужна документация к нашему ПП.

Проектирование - перевод из документов, которые нам помогали проводить анализ, в проектные документы, на основе которых мы уже непосредственно пишем код.

1. Этап эволюции. Эволюция = кодирование + тестирование + рекурсивный дизайн (от простого состояния системы развиваем до сложного). Разница между эволюцией и модификацией:

* Эволюция - в рамках разработки продукта. До этапа сдачи продукта.
* Модификация - изменения, вносимые уже после сдачи продукта заказчику. Этапы эволюции и модификации выполняют разные люди.

Преимущества выделения этапа эволюции:

* Предоставляется обширная обратная связь
* Получаются разные версии системы - дает механизм отката и можно выпускать альтернативные версии продукта.
* Можно рассматривать каждую версию для демонстрации и обсуждения с заказчиком. Уже на начальных этапах есть результаты.
* Интерфейс разрабатывается отдельно от модели.

Какие изменения могут быть реализованы при эволюции:

* Добавление новых классов.
* Изменение реализации класса.
* Изменение представления класса.
* Реорганизация структуры класса.
* Самое страшное - изменение интерфейса базового класса. (До такой ситуации лучше не доводить)

В структурном программировании было кодирование + тестирование.

В ООП эволюция включает в себя кодирование + тестирование + рекурсивный дизайн. Мы начинаем с простого, начального состояния модели нашей системы и потом её развиваем. Если у нас возникли какие-то проблемы, мы переходим или к анализу, либо перепроектируем какие-то части системы (рд-подход).

### Преимущества эволюции:

1. Предоставляется обширная обратная связь.
2. Появляются различные версии нашей системы. РД дает очень удобный механизм отката. Если мы понимаем, что текущая версия нам не подходит, мы можем вернуться на предыдущую. Возможна реализация нескольких версий нашей системы, мы можем разработать альтернативные версии (как правило, это связано с тем, когда мы делаем систему для нескольких пользователей - у них может быть разный функционал). Быстро и легко мы можем перейти от одного решения к другому.
3. За счет большого количества версий системы появляется возможность плотно взаимодействовать с заказчиком.
4. Как в структурном программировании, мы разворачиваем систему и уже на начальных этапах видны результаты.
5. Интерфейс проектируется отдельно, заранее, до основного этапа разработки, так как на интерфейсе отрабатывается взаимодействие с заказчиком, как с пользователем системы.

### Какие изменения могут быть реализованы на этом этапе:

1. Добавление нового класса - вещь совершенно безболезненная, которая при нормальном анализе не приводит к изменению написанного кода.
2. Изменения реализации класса. Это происходит на основе наследования, мы добавляем новый класс, подменяющий другой класс. Мы не переписываем существующий код.  
   Добавление нового класса - имеется ввиду добавление своей ветви. Реализация класса - на основе наследования.
3. Изменение представления класса. Мы можем с классом работать по другому и это не должно повлиять на переписывание кода.
4. Реорганизация структуры классов - болезненная часть, затрагивающая минимальное переписывание (использование паттернов, например, появление компоновщика, адаптера для расширения функционала, прокси для добавления функционала или реализации доступа к объекту).
5. Самое страшное - изменение интерфейса базового класса. До такой ситуации лучше не доводить, как правило, это переделывание большого кода.

4. Модификация

Эволюция - непосредственно разработка ПП. Мы сдали заказчику готовый продукт, а следующие изменения, когда мы уже сдали, будет модификацией. Модификация затрагивает анализ и затрагивает проектирование

Разница между эволюцией и модификацией:

1. До этапа сдачи - эволюция, после - модификация.
2. Эти этапы выполняют совершенно разные люди.

**Рабочие продукты ОО анализа**

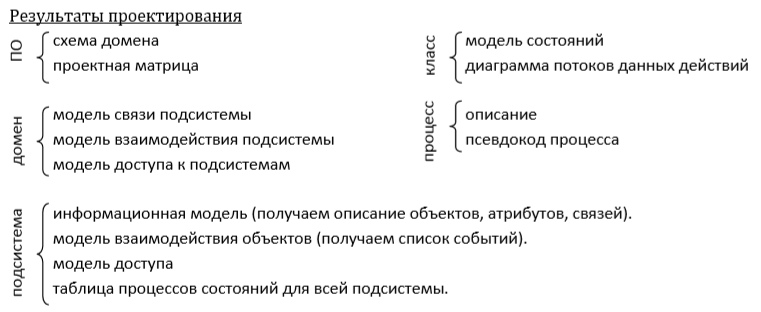
Когда мы проводим анализ, мы разрабатываем рабочие документы. Они лягут в основу проектных документов (на этапе проектирования).

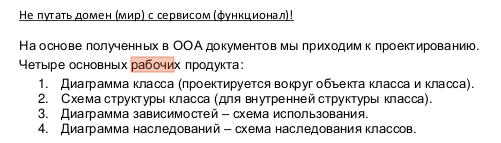
Какие документы создаются при анализе:

1. **Для всей программы:**  
   1.1. Схема доменов  
   1.2. Проектная матрица
2. **Для каждого домена:**Если в домене больше 30-50ти классов, разбиваем домен на подсистемы. Разбиваем граф связности "по слоям" (по минимуму связей) - внутри подсистемы связей много, между подсистемами связей мало.  
   2.1. Модель связей подсистем  
   2.2. Модель доступа подсистем  
   2.3. Модель взаимодействия подсистем
3. **Для каждой подсистемы:**3.1. Информационная модель (диаграмма-сущность-связь)  
   Описание классов и из атрибутов (членов данных)  
   Описание связей  
   3.2. Модель (диаграмма) взаимодействxия объектов  
   Список событий в подсистеме  
   3.3. Модель доступа к объекту  
   3.4. Таблица процессов состояний
4. **Для каждого класса:**4.1. Модель состояний (диаграмма переходов состояний)  
   4.2. Алгоритм действий состояний
5. **Для каждого состояния:**5.1 Диаграмма потоков данных действий
6. **Для каждого действия:**  
   6.1. Описание процессов

С чего начинать разработку проекта: разбили задачу на домены, рассматриваем прикладной домен (наш). Разработка прикладного домена начинается с информационного моделирования.

Информационное моделирование начинаем всегда с физических объектов. Смотрим, какие объекты существуют. Пытаемся эти объекты сгруппировать по принципу одних и тех же характеристик. Выделяем, чем характеризуется объект - выделяем атрибуты объектов.





## [Концепции информационного моделирования. Понятие атрибута. Типы атрибутов. Правила атрибутов. Понятие связи. Типы связей. Формализация связей. Композиция связей. Подтипы и супертипы. Диаграмма сущность-связь.](https://github.com/RusinovaDaria/examOOP/wiki/14-(5).-Концепции-информационного-моделирования.-Понятие-атрибута.-Типы-атрибутов.-Правила-атрибутов.-Понятие-связи.-Типы-связей.-Формализация-связей.-Композиция-связей.-Подтипы-и-супертипы.-Диаграмма-сущность-связь.)

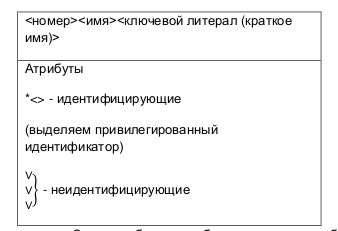
С чего начинать разработку проекта: разбили задачу на домены, рассматриваем прикладной домен (наш). Разработка прикладного домена начинается с информационного моделирования. Мы начинаем всегда с физических объектов. Смотрим, какие объекты существуют. Пытаемся эти объекты сгруппировать по принципу одних и тех же характеристик. Выделяем, чем характеризуется объект - выделяем атрибуты объектов.

**Концепции информационного моделирования**

Разработка прикладного домена начинается с информационного моделирования.

Информационное моделирование включает в себя:

* Выделение физических объектов
* Короткое описание классов (чтобы установить, является ли объект экземпляром класса)
* Выделение характеристик объектов и идентификаторов (множество из одного или несколькихатрибутов, однозначно определяющих экземпляр класса)
* Графическое обозначение класса на информационной модели:



Строим таблицу атрибутов: для каждого объекта должен быть определен однозначно атрибут

(<имя>(<иден.>,<список атрибутов>) - текстовое описание)

**Понятие атрибута**

Атрибуты - характеристики сущностей, переменные-члены объектов (данные). Каждая характеристика, которая является общей для всех экземпляров класса, выделяется как отдельный атрибут.

**Типы атрибутов**

* Описательные – факты, внутренне присущие каждому объекту (если меняется, то только какой-то аспект, сам объект остается прежним)
* Указывающие (идентифицирующие) – использующиеся как идентификатор или часть идентификатора (если меняется, то объект сменил имя, но сам остался)
* Вспомогательные – выделяемые из состояния объекта или его отношений, связей с другими объектами (при изменении соответственно меняются связи или состояние объекта)

Для каждого атрибута выделяем, какие значения может принимать атрибут. (Чтобы в дальнейшем определить тип для атрибута).

* Если описательный, то описание – информационная строка, которая показывает реальную характеристику, как определяется характеристика и почему она уместна для данного объекта.
* Если указывающий, то описание – форма указания, кто назначает атрибуты использующиеся в идентификаторе.
* Если вспомогательный, то описание – какое отношение или состояние сохраняются.
* Для описательного: показываем, какую характеристику хранит атрибут, как определяется и кто задает этот атрибут.
* Для идентифицирующего: показываем форму указания, кто назначает указание и степень, в которой идентифицирующий атрибут используется как идентификатор.  
  Из группы идентифицирующих атрибутов выделяется привилегированный. (Если имя и фамилия - идентифицирующие атрибуты, из них фамилия - привилегированный).
* Для вспомогательного: показываем, какую связь формализует атрибут и почему мы так ее формализуем.

**Правила атрибутов**

1. Один объект класса имеет одно единственное значение для каждого атрибута в любой момент времени. Не может быть атрибутов, значения которых не определены в какой-то момент жизни объекта.
2. Атрибут не должен содержать никакой внутренней структуры.
3. Когда объект имеет составной идентификатор, каждый атрибут являющийся частью идентификатора, представляет характеристику всего объекта, а не его части.
4. Каждый атрибут не являющийся частью идентификатора, представляет собой характеристику объекта указанного идентификатором, а не характеристику другого атрибута.

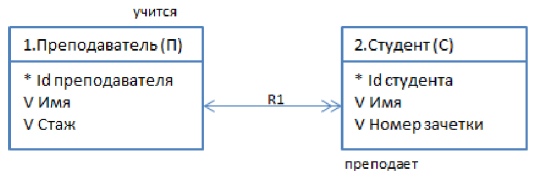
(!) - Изменение атрибута не приводит к изменению объекта - это тот же объект с другим атрибутом. Даже идентифицирующего.

**Понятие связи**

Связь – это абстракция отношений, которые возникают между объектами.

Задаём связь из перспективы каждого участвующего объекта.

Каждой связи присваивается уникальный идентификатор, который состоит из буквы и номера.



**типы связи**

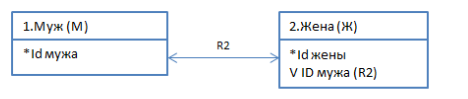
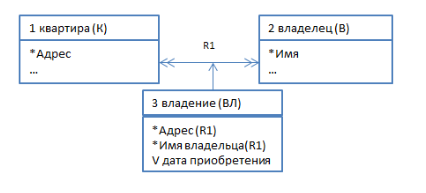
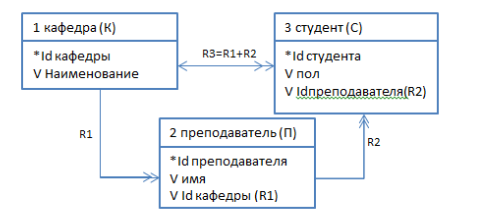
Типы связей по множественности:

* Один к одному <--------->
* Один ко многим <--------->>
* Многие ко многим <<--------->>

Типы связей по условности:

* Условные (один может не участвовать)
* Безусловные (оба участвуют)
* Биусловные (оба могут не участвовать) Если со стороны объекта связь условная, то у стрелочки с его стороны ставится У.

**Формализация связей**

* Один к одному (атрибут связи добавляется в главный из объектов, но если у связи динамическое поведение - с помощью ассоциативного объекта). Главный - наиболее осведомленный о всей системе.
* Один ко многим (атрибут связи добавляется со стороны многих, но если у связи динамическое поведение или объекты со стороны многих стоят рангом ниже, чем со стороны одного - с помощью ассоциативного объекта)
* Многие ко многим - с помощью ассоциативного объекта
* 1) Безусловная один к одному: добавляем атрибут связи в любой из объектов (тот, который более осведомлен о системе)
* 
* 2) Безусловная один ко многим: формализуется со стороны многих (добавляем им атрибут связи)
* 3)Многие ко многим: формализуется через ассоциативный объект:
* 
* 4) Если связь условна, биусловна или имеет динамическое поведение, то она формализуется ассоциативным объектом
* 

**Композиция связей (видимо выше…)**

**Подтипы и супертипы**

Если для объектов разных классов существует некий общий атрибут - объединяем их **суперклассом**. Общие атрибуты для разных классов выносим в их **суперкласс**.

В объектно-ориентированном анализе суперкласс - всегда абстрактное понятие, мы не рассматриваем возможность создания объектов суперкласса.

Связь суперкласса с подклассами обозначается номером, начинающемся со 100 (101, 102, 103...).

**Диаграмма сущность-связь**

На диаграмме сущность-связь сущности располагаются в прямоугольнике. Каждой сущности присваиваем и уникальный для домена номер. Выделяем имя сущности (желательно существительное). Для имени указываем ключевой литерал (29:20 часть 2 лекция 10). Указываем атрибуты: Привилегированный указывающий атрибут обозначаем '\*' Все остальные атрибуты - просто перечисляем.

Между сущностями графически указываем связи.

Остальные правила и термины приведены выше.

## [Модель поведения объектов. Жизненный цикл и диаграмма перехода в состояния (ДПС). Виды состояний. События, данные событий. Действия состояний. Таблица перехода в состояния (ТПС). Правила переходов.](https://github.com/RusinovaDaria/examOOP/wiki/№-15-(6).-Модель-поведения-объектов.-Жизненный-цикл-и-диаграмма-перехода-в-состояния-(ДПС).-Виды-состояний.-События,-данные-событий.-Действия-состояний.-Таблица-перехода-в-состояния-(ТПС).-Правила-переходов.)

**Модель поведения объектов**

Выделяем поведение объектов. Отталкиваемся от реального физического мира. Объекты в течении времени жизни проходят некоторые стадии. То, как эволюционирует объект между стадиям, характеризует черту поведения объекта. Любой объект в данный момент времени находится в какой-то стадии.

Объект переходит из одной стадии в другую скачкообразно.

Не все переходы из одной стадии в другую возможны.

В физическом мире происходят инциденты, которые заставляют объекты переходить из одной стадии в другую. Или инциденты являются следствием перехода объекта из одной стадии в другую.

Для описания поведения объектов используем модель Мура:

* Множество состояний объекта (стадий),
* Множество событий, которые переводят объект из одной стадии в другую,
* Множество действий состояний
* Правила переходов
* 1. Из множества состояний: каждое состояние представляет стадию в жизненном цикле объекта.
* 2. Из множества событий: каждое событие означает инцидент, указывающий на эволюционирование.
* 3. Из правил перехода: правило определяет, какое новое состояние достигается объектом под воздействием события.
* 4.Из действий: операции, которые должны быть выполнены, чтобы объект перешел в какое-то состояние.

Модель состояний можно формализовать таблицей и диаграммой.

**Жизненный цикл и диаграмма перехода в состояния ДПС**

Формы жизненных циклов:

1. Циркуляционный
2. Рождение-смерть

Когда формируются жизненные циклы:

1. Создание или уничтожение во время выполнения
2. Миграция между подклассами
3. Объект производится или возникает поэтапно.
4. Объект – задача или запрос
5. Динамическая связь Для пассивных объектов мы не выделяем жизненные циклы. Но иногда мы это делаем в интересах активных объектов.

Жизненные циклы объектов:

* В каждый момент объект находится в какой-то одной стадии
* Переход из одной стадии в другую происходит скачкообразно и является реакцией на какой-то инцидент
* Переходы возможны не из всех состояний

Состояние – положение объекта, в котором применяется особый набор правил и линий поведения, предписаний физических законов (для состояния ставятся в соответствие уникальные в рамках данной модели состояний им и номер)

Состояние – это положение объектов, в котором определяются определённый набор правил, линий поведений, предписаний, определённых законов.

ДПС - диаграмма переходов состояний.

Каждое состояние рисуем прямоугольником.

Каждому состоянию присваиваем номер и имя состояния.

**Виды состояний**

1. Состояние создания - в эти состояния происходим переход не из стостояния
2. Заключительные состояния: 2.1. Объект уничтожается - состояние рисуем пунктирной линией  
   2.2. Состояние, из которого объект больше не переходит в другие состояния, но не уничтожается - рисуется как обычное состояние из которого нет переходов
3. Текущее состояние (не создания и не заключительное)

Чтобы определять текущее состояние в класс добавляется вспомогательный атрибут - статут - хранит текущее состояние. Состояния контекста - промежуточные состояния, которые определяются предыдущим и следующим состояниями. Их цель - организовать этот переход.

Если для объекта выделяется модель состояний, то на информационной модели для класса этого объекта нужно добавить атрибут «статус», хранящий состояние

Событие – это абстракция инцидента или сигнала в реальном мире, сообщающего о перемещении чего-либо в новое состояние.

**События, данные событий Действия состояний**

**Таблица перехода в состояния ТПС**

**Правила переходов**

## [Модель взаимодействия объектов (МВО). Диаграмма взаимодействия объектов в подсистеме. Типы событий. Схемы управления. Имитирование. Каналы управления.](https://github.com/RusinovaDaria/examOOP/wiki/16(7).-Модель-взаимодействия-объектов-(МВО).-Диаграмма-взаимодействия-объектов-в-подсистеме.-Типы-событий.-Схемы-управления.-Имитирование.-Каналы-управления.)

## [Диаграмма потоков данных действий (ДПДД). Типы процессов: аксессоры, генераторы событий, преобразования, проверки. Таблица процессов (ТП). Модель доступа к объектам (МДО).](https://github.com/RusinovaDaria/examOOP/wiki/17(8).-Диаграмма-потоков-данных-действий-(ДПДД).-Типы-процессов:-аксессоры,-генераторы-событий,-преобразования,-проверки.-Таблица-процессов-(ТП).-Модель-доступа-к-объектам-(МДО).)

## [Домены. Модели доменного уровня. Типы доменов. Мосты, клиенты, сервера.](https://github.com/RusinovaDaria/examOOP/wiki/№-18(9).-Домены.-Модели-доменного-уровня.-Типы-доменов.-Мосты,-клиенты,-сервера.)

## ДС[Объектно-ориентированное проектирование. Диаграмма класса. Структура класса. Диаграмма зависимостей. Диаграмма наследования.](https://github.com/RusinovaDaria/examOOP/wiki/№-19(10).-Объектно-ориентированное-проектирование.-Диаграмма-класса.-Структура-класса.-Диаграмма-зависимостей.-Диаграмма-наследования.)

## [Архитектурный домен. Паттерн КМС. Шаблоны для создания прикладных классов.](https://github.com/RusinovaDaria/examOOP/wiki/20(11).-Архитектурный-домен.-Паттерн-КМС.-Шаблоны-для-создания-прикладных-классов.)

## [Структурные паттерны: адаптер (Adapter), декоратор (Decorator), компоновщик (Composite), заместитель (Proxy), мост (Bridge), фасад (Facade).](https://github.com/RusinovaDaria/examOOP/wiki/№-21-(12).-Структурные-паттерны:-адаптер-(Adapter),-декоратор-(Decorator),-компоновщик-(Composite),-заместитель-(Proxy),-мост-(Bridge),-фасад-(Facade).)

### АДАПТЕР - паттерн, структурирующий классы и объекты. <https://refactoring.guru/ru/design-patterns/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.

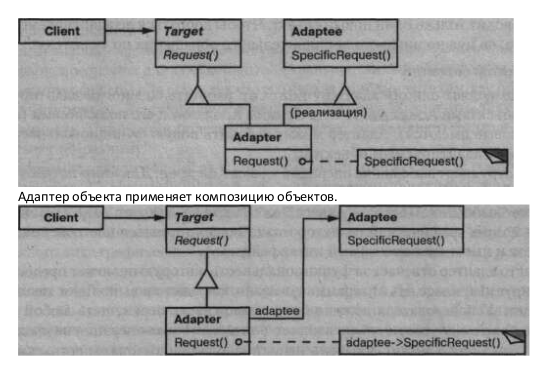
**Применимость**

Применяйте паттерн адаптер, когда:

* Хотите использовать существующий класс, но его интерфейс не соответствует вашим потребностям;
* Собираетесь создать повторно используемый класс, который должен взаимодействовать с заранее неизвестными или не связанными с ним классами, имеющими несовместимые интерфейсы;
* (только для адаптера объектов!) Нужно использовать несколько существующих подклассов, но непрактично адаптировать их интерфейсы путем порождения новых подклассов от каждого. В этом случае адаптер объектов может приспосабливать интерфейс их общего родительского класса.

**Структура**

Адаптер класса использует множественное наследование для адаптации одного интерфейса к другому.



**Участники**

* Target - целевой: определяет зависящий от предметной области интерфейс, которым пользуется
* Client;
* Client - клиент: вступает во взаимоотношения с объектами, удовлетворяющими интерфейсу Target;
* Adaptee - адаптируемый: определяет существующий интерфейс, который нуждается в адаптации;
* Adapter - адаптер: адаптирует интерфейс Adaptee к интерфейсу Target.

**Результаты**

Результаты применения адаптеров объектов и классов различны.

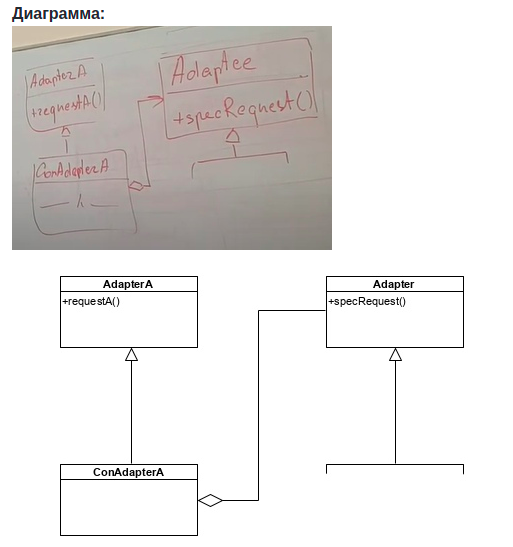
*Адаптер класса:*

* Адаптирует Adaptee к Target, перепоручая действия конкретному классу Adaptee. Поэтому данный паттерн не будет работать, если мы захотим одновременно адаптировать класс и его подклассы;
* Позволяет адаптеру Adapter заместить некоторые операции адаптируемого класса Adaptee, так как Adapter есть не что иное, как подкласс Adaptee;
* Вводит только один новый объект. Чтобы добраться до адаптируемого класса, не нужно никакого дополнительного обращения по указателю.

*Адаптер объектов:*

* Позволяет одному адаптеру Adapter работать со многим адаптируемыми объектами Adaptee, то есть с самим Adaptee и его подклассами (если таковые имеются). Адаптер может добавить новую функциональность сразу всем адаптируемым объектам;
* Затрудняет замещение операций класса Adaptee. Для этого потребуется породить от Adaptee подкласс и заставить Adapter ссылаться на этот подкласс, а не на сам Adaptee.

-----



**Проблема**: объект в разных местах программы играет разные роли.

Это плохо, потому что:

* Для каждой роли разрабатывается свой интерфейс. Несколько ролей для одного объекта - значит, избыточный интерфейс.
* Роль объекта - это возложение на него определенной ответственности. Несколько ролей - это несколько ответственностей. Недопустимо с точки зрения принципов ООП.

**Идея решения**: У объекта был один интерфейс. Подменяем этот интерфейс другим - соответственно той роли, в которой мы хотим использовать объект. Таким образом, в зависимости от ситуации мы можем использовать этот объект в разных ролях. С этим объектом работаем через объект другого класса. Объект, через класс которого мы работаем, имеет интерфейс необходимой нам роли.

**Использование паттерна:**

1. Один объект может выступать в нескольких ролях.
2. Нам нужно встроить в систему сторонние классы, имеющие другой интерфейс. Класс с любым интерфейсом можем встроить в нашу программу.
3. Мы, используя полиморфизм, сформировали интерфейс для базового. Определенные сущности, наследуемые от базового класса, должны поддерживать еще какой-то функционал. Мы не можем расширить этот функционал и изменять написанный код. Решаем эту проблему за счет адаптера, который предоставляет расширенный интерфейс.

**Плюсы и минусы**

### 

### ДЕКОРАТОР

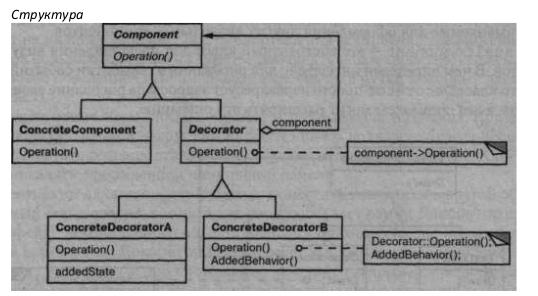
**Назначение**

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

**Применимость**

Используйте паттерн декоратор:

* Для динамического, прозрачного для клиентов добавления обязанностей объектам;
* Для реализации обязанностей, которые могут быть сняты с объекта;
* Когда расширение путем порождения подклассов по каким-то причинам неудобно или невозможно.
* Иногда приходится реализовывать много независимых расширений, так что порождение подклассов для поддержки всех возможных комбинаций приведет к комбинаторному росту их числа. В других случаях определение класса может быть скрыто или почему-либо еще недоступно, так что породить от него подкласс нельзя.



**Участники**

* Component - компонент: определяет интерфейс для объектов, на которые могут быть динамически возложены дополнительные обязанности;
* ConcreteComponent - конкретный компонент: определяет объект, на который возлагаются дополнительные обязанности;
* Decorator - декоратор: хранит ссылку на объект Component и определяет интерфейс, соответствующий интерфейсу Component;
* ConcreteDecorator (BorderDecorator, ScrollDecorator) - конкретный декоратор: возлагает
* дополнительные обязанности на компонент.

**Результаты**

У паттерна декоратор есть, по крайней мере, два плюса и два минуса:

* Большая гибкость, нежели у статического наследования.
* Позволяет избежать перегруженных функциями классов на верхних уровнях иерархии.
* Декоратор и его компонент не идентичны.
* Множество мелких объектов. (При использовании в проекте паттерна декоратор нередко получается система, составленная из большого числа мелких объектов, которые похожи друг на друга 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.)

-----

**Проблема**: нам надо добавить/подменить классам функционал. Причем одинаковый для нескольких классов. Если мы подменим функционал в производных для каждого из классов, которые мы хотим изменить - разрастается иерархия, приходится дублировать код.

**Идея решения:** вынести это добавление в отдельный класс - декоратор.

**Использование паттерна**: добавление/подмена небольшой части функционала, одинаковой для разных классов.

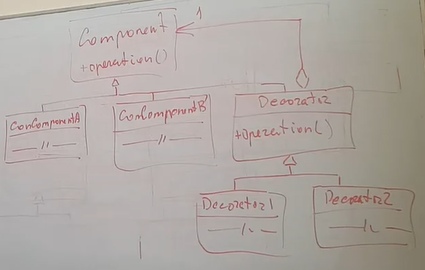
**Преимущества:**

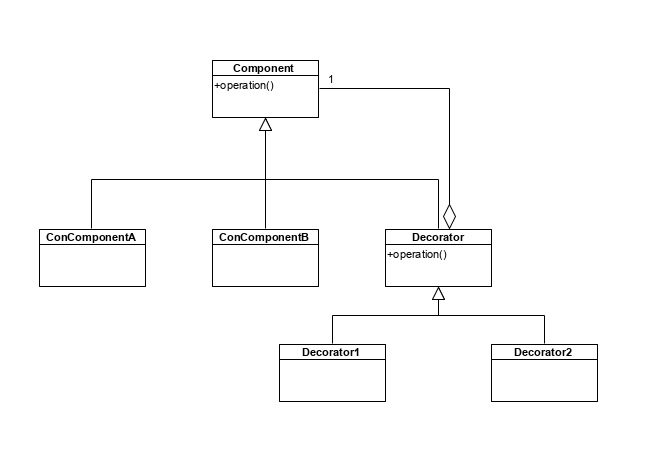
* Гибкая система. Уменьшается иерархия.
* Декорировать можем во время выполнения работы программы.
* Избавляемся от дублирования кода. Этот код уходит в конкретный декоратор, не дублируясь.

**Недостатки**:

Не можем убрать обертку из иерархии оберток. Придется заново создавать компонент с обёртками.

**Диаграмма:**





--- С САЙТА <https://refactoring.guru/ru/design-patterns/decorator>

### КОМПОНОВЩИК

**Назначение**

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

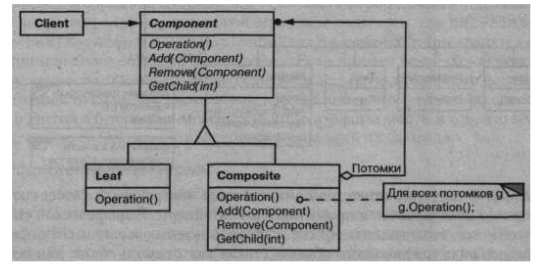
**Применимость**

Используйте паттерн компоновщик, когда:

Нужно представить иерархию объектов вида часть-целое;

Хотите, чтобы клиенты единообразно трактовали составные и индивидуальные объекты.

**Структура**



**Участники**

* Component - компонент:объявляет интерфейс для компонуемых объектов; предоставляет подходящую реализацию операций по умолчанию, общую для всех классов; объявляет интерфейс для доступа к потомкам и управления ими; определяет интерфейс для доступа к родителю компонента в рекурсивной структуре и при необходимости реализует его. Описанная возможность необязательна;
* Leuf - лист: представляет листовые узлы композиции и не имеет потомков; определяет поведение примитивных объектов в композиции;
* Composite - составной объект: определяет поведение компонентов, у которых есть потомки; хранит компоненты-потомки; реализует относящиеся к управлению потомками операции в интерфейсе
* класса Component;
* Client - клиент: манипулирует объектами композиции через интерфейс Component.

**Результаты**

Паттерн компоновщик:

* Определяет иерархии классов, состоящие из примитивных и составных объектов.
* Упрощает архитектуру клиента.
* Облегчает добавление новых видов компонентов.
* Способствует созданию общего дизайна.

### ЗАМЕСТИТЕЛЬ

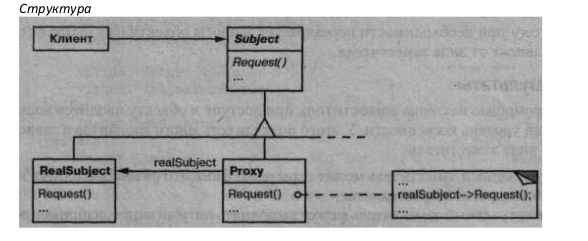
**Назначение**

Является суррогатом другого объекта и контролирует доступ к нему.

**Применимость**

* Удаленный заместитель предоставляет локального представителя вместо объекта, находящегося в другом адресном пространстве;
* Виртуальный заместитель создает «тяжелые» объекты по требованию.
* Защищающий заместитель контролирует доступ к исходному объекту.
* «Умная» ссылка - это замена обычного указателя.

**Структура**



**Участники**

* Proxy - заместитель: хранит ссылку, которая позволяет заместителю обратиться к реальному субъекту.
* Объект класса Proxy может обращаться к объекту класса Subject, если интерфейсы классов RealSubject и Subject одинаковы; предоставляет интерфейс, идентичный интерфейсу Subject, так что заместитель всегда может быть подставлен вместо реального субъекта; контролирует доступ к реальному субъекту и может отвечать за его создание и удаление;
* Subject - субъект: определяет общий для RealSubject и Proxy интерфейс, так что класс Proxy можно использовать везде, где ожидается RealSubject;
* RealSubject - реальный субъект: определяет реальный объект, представленный заместителем.

**Результаты**

С помощью паттерна заместитель при доступе к объекту вводится дополнительный уровень косвенности. У этого подхода есть много вариантов в зависимости от вида заместителя:

* Удаленный заместитель может скрыть тот факт, что объект находится в другом адресном пространстве;
* Виртуальный заместитель может выполнять оптимизацию, например создание объекта по требованию;
* Защищающий заместитель и «умная» ссылка позволяют решать дополнительные задачи при доступе к объекту.

### 

### МОСТ

**Назначение**

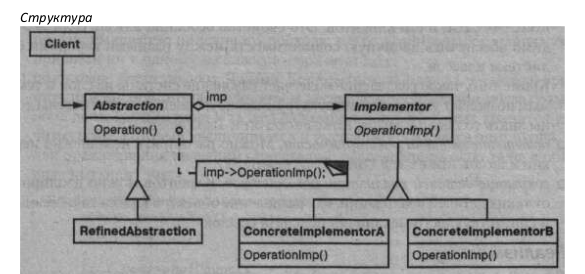
Отделить абстракцию от ее реализации так, чтобы то и другое можно было изменять независимо.

**Применимость**

Используйте паттерн мост, когда:

* Хотите избежать постоянной привязки абстракции к реализации. Так, например, бывает, когда реализацию необходимо выбирать во время выполнения программы;
* И абстракции, и реализации должны расширяться новыми подклассами.
* В таком случае паттерн мост позволяет комбинировать разные абстракции и реализации и изменять их независимо;
* Изменения в реализации абстракции не должны сказываться на клиентах, то есть клиентский код не должен перекомпилироваться;
* Вы хотите полностью скрыть от клиентов реализацию абстракции. В C++ представление класса видимо через его интерфейс;

**Структура**



**Участники**

* Abstraction - абстракция: определяет интерфейс абстракции; Хранит ссылку на объект типа Implementor;
* RefinedAbstraction - уточненная абстракция: расширяет интерфейс, определенный абстракцией Abstraction;
* Implementor - реализатор: определяет интерфейс для классов реализации. Он не обязан точно соответствовать интерфейсу класса Abstraction. На самом деле оба интерфейса могут быть совершенно различны. Обычно интерфейс класса Implementor предоставляет только примитивные операции, а класс Abstraction определяет операции более высокого уровня, базирующиеся на этих примитивах;
* Concretelmplementor - конкретный реализатор: содержит конкретную реализацию интерфейса класса Implementor.

**Результаты**

Результаты применения паттерна мост таковы:

* Отделение реализации от интерфейса.
* Повышение степени расширяемости.
* Сокрытие деталей реализации от клиентов.

### ФАСАД

**Назначение**

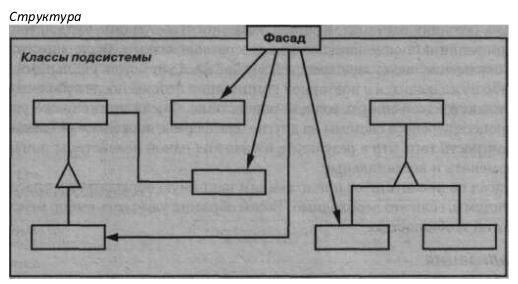
Предоставляет унифицированный интерфейс вместо набора интерфейсов некоторой подсистемы. Фасад определяет интерфейс более высокого уровня, который упрощает использование подсистемы.

**Применимость**

Используйте паттерн фасад, когда:

* Хотите предоставить простой интерфейс к сложной подсистеме.
* Между клиентами и классами реализации абстракции существует много зависимостей.
* Вы хотите разложить подсистему на отдельные слои.

**Структура**



**Участники**

* Facade - фасад: «знает», каким классам подсистемы адресовать запрос; делегирует запросы клиентов подходящим объектам внутри подсистемы;
* Классы подсистемы: реализуют функциональность подсистемы; выполняют работу, порученную объектом Facade; ничего не «знают» о существовании фасада, то есть не хранят ссылок на него.

**Результаты**

У паттерна фасад есть следующие преимущества:

* Изолирует клиентов от компонентов подсистемы, уменьшая тем самым число объектов, с которыми клиентам приходится иметь дело, и упрощая работу с подсистемой;
* Позволяет ослабить связанность между подсистемой и ее клиентами;
* Фасад не препятствует приложениям напрямую обращаться к классам подсистемы, если это необходимо.

## [Порождающие паттерны: одиночка (Singleton), фабричный метод (Factory Method), абстрактная фабрика (Abstract Factory), строитель (Builder), прототип (Prototype), пул объектов (Object Pool).](https://github.com/RusinovaDaria/examOOP/wiki/№-22-(13).-Порождающие-паттерны:-одиночка-(Singleton),-фабричный-метод-(Factory-Method),-абстрактная-фабрика-(Abstract-Factory),-строитель-(Builder),-прототип-(Prototype),-пул-объектов-(Object-Pool).)

### ОДИНОЧКА <https://refactoring.guru/ru/design-patterns/singleton>

**Назначение**

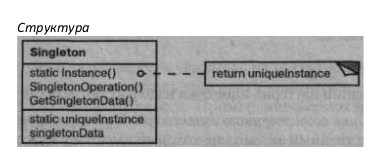
Гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.

**Применимость**

Используйте паттерн одиночка, когда:

* Должен быть ровно один экземпляр некоторого класса, легко доступный всем клиентам;
* Единственный экземпляр должен расширяться путем порождения подклассов, и клиентам нужно иметь возможность работать с расширенным экземпляром без модификации своего кода.

**Структура**



**Участники**

* Singleton - одиночка: определяет операцию Instance, которая позволяет клиентам получать доступ к единственному экземпляру. Instance – это метод класса и статическая функция-член в C++; Может нести ответственность за создание собственного уникального экземпляра.

**Результаты**

У паттерна одиночка есть определенные достоинства:

* Контролируемый доступ к единственному экземпляру.
* Уменьшение числа имен.
* Допускает уточнение операций и представления.
* Допускает переменное число экземпляров.
* Большая гибкость, чем у операций класса.
* Гарантирует наличие единственного экземпляра класса.
* Предоставляет к нему глобальную точку доступа.
* Реализует отложенную инициализацию объекта-одиночки.
* Нарушает *принцип единственной ответственности класса*.
* Маскирует плохой дизайн.
* Проблемы мультипоточности.
* Требует постоянного создания Mock-объектов при юнит-тестировании.

### 

### ФАБРИЧНЫЙ МЕТОД <https://refactoring.guru/ru/design-patterns/factory-method>

* Избавляет класс от привязки к конкретным классам продуктов.
* Выделяет код производства продуктов в одно место, упрощая поддержку кода.
* Упрощает добавление новых продуктов в программу.
* Реализует *принцип открытости/закрытости*.
* Может привести к созданию больших **[параллельных иерархий классов](https://refactoring.guru/ru/smells/parallel-inheritance-hierarchies)**, так как для каждого класса продукта надо создать свой подкласс создателя.

### 

### АБСТРАКТНАЯ ФАБРИКА <https://refactoring.guru/ru/design-patterns/abstract-factory>

Абстрактная фабрика - паттерн, порождающий объекты.

**Назначение:**

Предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, не

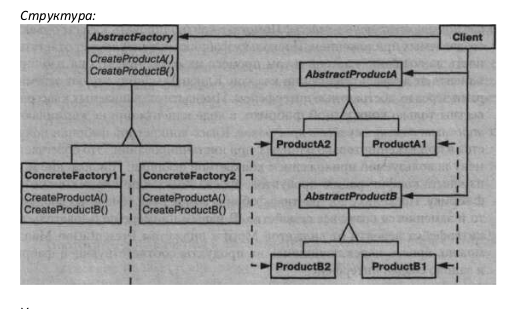
специфицируя их конкретных классов.

**Применимость:**

Используйте паттерн абстрактная фабрика, когда:

* a) Cистема не должна зависеть от того, как создаются, компонуются и представляются входящие в нее
* объекты;
* b) Входящие в семейство взаимосвязанные объекты должны использоваться вместе и вам необходимо
* обеспечить выполнение этого ограничения;
* c) Система должна конфигурироваться одним из семейств составляющих ее объектов;
* d) Вы хотите предоставить библиотеку объектов, раскрывая только их интерфейсы, но не реализацию.

**Структура:**



**Участники:**

* AbstractFactory - абстрактная фабрика: объявляет интерфейс для операций, создающих абстрактные объекты-продукты;
* ConcreteFactory - конкретная фабрика: реализует операции, создающие конкретные объекты- продукты;
* AbstractProduct - абстрактный продукт: объявляет интерфейс для типа объекта-продукта;
* ConcreteProduct - конкретный продукт: определяет объект - продукт, создаваемый соответствующей конкретной фабрикой, реализует интерфейс Abstract Product;
* Client - клиент: пользуется исключительно интерфейсами, которые объявлены в классах AbstractFactory и AbstractProduct.

**Результаты:**

Паттерн абстрактная фабрика обладает следующими плюсами и минусами:

* Гарантирует сочетаемость создаваемых продуктов.
* Избавляет клиентский код от привязки к конкретным классам продуктов.
* Выделяет код производства продуктов в одно место, упрощая поддержку кода.
* Упрощает добавление новых продуктов в программу.
* Реализует *принцип открытости/закрытости*.
* Усложняет код программы из-за введения множества дополнительных классов.
* Требует наличия всех типов продуктов в каждой вариации.

### СТРОИТЕЛЬ <https://refactoring.guru/ru/design-patterns/builder>

**Назначение:**

Отделяет конструирование сложного объекта от его представления, так что в результате одного и того же

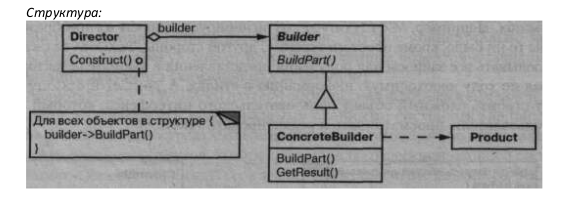
процесса конструирования могут получаться разные представления.

**Применимость:**

Используйте паттерн строитель, когда:

* Алгоритм создания сложного объекта не должен зависеть от того, из каких частей состоит объект и как они стыкуются между собой;
* Процесс конструирования должен обеспечивать различные представления конструируемого объекта.

**Структура:**



**Участники:**

* Builder - строитель: задает абстрактный интерфейс для создания частей объекта Product;
* ConcreteBuilder- конкретный строитель: конструирует и собирает вместе части продукта посредством реализации интерфейса Builder; Определяет создаваемое представление и следит за ним; Предоставляет интерфейс для доступа к продукту;
* Director - распорядитель: конструирует объект, пользуясь интерфейсом Builder;
* Product - продукт: представляет сложный конструируемый объект. ConcreteBuilder строит внутреннее представление продукта и определяет процесс его сборки; Включает классы, которые определяют составные части, в том числе интерфейсы для сборки конечного результата из частей.

**Результаты:**

Плюсы и минусы паттерна строитель и его применения:

* Позволяет изменять внутреннее представление продукта;
* Изолирует код, реализующий конструирование и представление;
* Дает более тонкий контроль над процессом конструирования.
* Позволяет создавать продукты пошагово.
* Позволяет использовать один и тот же код для создания различных продуктов.
* Изолирует сложный код сборки продукта от его основной бизнес-логики.
* Усложняет код программы из-за введения дополнительных классов.
* Клиент будет привязан к конкретным классам строителей, так как в интерфейсе директора может не быть метода получения результата.

### ПРОТОТИП <https://refactoring.guru/ru/design-patterns/prototype>

Прототип - паттерн, порождающий объекты.

**Назначение:**

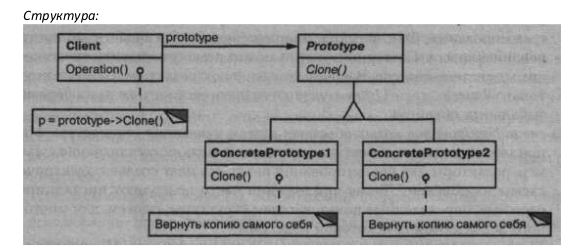
Задает виды создаваемых объектов с помощью экземпляра-прототипа и создает новые объекты путем

копирования этого прототипа.

**Применимость:**

* Используйте паттерн прототип, когда система не должна зависеть от того, как в ней создаются, компонуются и представляются продукты:
* Инстанцируемые классы определяются во время выполнения, например с помощью динамической загрузки;
* Для того чтобы избежать построения иерархий классов или фабрик, параллельных иерархии классов продуктов;
* Экземпляры класса могут находиться в одном из не очень большого числа различных состояний. Может оказаться удобнее установить соответствующее число прототипов и клонировать их, а не инстанцировать каждый раз класс вручную в подходящем состоянии.

**Структура:**



**Участники:**

* Prototype - прототип: объявляет интерфейс для клонирования самого себя;
* ConcretePrototype - конкретный прототип: реализует операцию клонирования себя;
* Client - клиент: создает новый объект, обращаясь к прототипу с запросом клонировать себя.

**Результаты**

У прототипа те же самые результаты, что у абстрактной фабрики и строителя: он скрывает от клиента конкретные классы продуктов, уменьшая тем самым число известных клиенту имен. Кроме того, все эти паттерны позволяют клиентам работать со специфичными для приложения классами без модификаций.

Ниже перечислены дополнительные преимущества паттерна прототип:

* Добавление и удаление продуктов во время выполнения.
* Спецификация новых объектов путем изменения значений.
* Специфицирование новых объектов путем изменения структуры
* Уменьшение числа подклассов.
* Динамическое конфигурирование приложения классами.
* Позволяет клонировать объекты, не привязываясь к их конкретным классам.
* Меньше повторяющегося кода инициализации объектов.
* Ускоряет создание объектов.
* Альтернатива созданию подклассов для конструирования сложных объектов.
* Сложно клонировать составные объекты, имеющие ссылки на другие объекты.

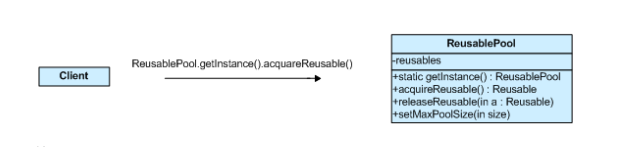
### ПУЛ ОБЪЕКТОВ

**Применимость**

Объектный пул применяется для повышения производительности, когда создание объекта в начале работы и

уничтожение его в конце приводит к большим затратам. Особенно заметно повышение производительности,

когда объекты часто создаются-уничтожаются, но одновременно существует лишь небольшое их число.



**Участники**

* Reusable - экземпляры классов в этой роли взаимодействуют с другими объектами в течение ограниченного времени, а затем они больше не нужны для этого взаимодействия.
* Client - экземпляры классов в этой роли используют объекты Reusable.
* ReusablePool - экземпляры классов в этой роли управляют объектами Reusable для использования объектами Client.

## [Паттерны поведения: стратегия (Strategy), команда (Command), цепочка обязанностей (Chain of Responsibility), подписчик-издатель (Publish-Subscribe), посредник (Mediator), посетитель (Visitor), опекун (Memento), шаблонный метод (Template Method), хранитель (Holder), итератор (Iterator), свойство (Property).](https://github.com/RusinovaDaria/examOOP/wiki/№-23-(14).-Паттерны-поведения:-стратегия-(Strategy),-команда-(Command),-цепочка-обязанностей-(Chain-of-Responsibility),-подписчик-издатель-(Publish-Subscribe),-посредник-(Mediator),-посетитель-(Visitor),-опекун-(Memento),-шаблонный-метод-(Template-Method),-хранитель-(Holder),-итератор-(Iterator),-свойство-(Property).)

### СТРАТЕГИЯ <https://refactoring.guru/ru/design-patterns/strategy>

**Назначение**

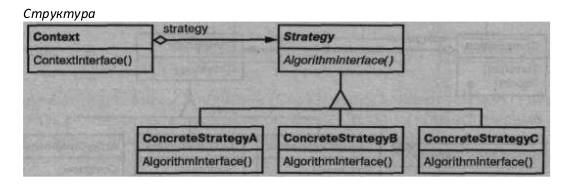
Определяет семейство алгоритмов, инкапсулирует каждый из них и делает их взаимозаменяемыми.

Стратегия позволяет изменять алгоритмы независимо от клиентов, которые ими пользуются.

Используйте паттерн стратегия, когда:

* Имеется много родственных классов, отличающихся только поведением.
* Вам нужно иметь несколько разных вариантов алгоритма.
* В алгоритме содержатся данные, о которых клиент не должен «знать». Используйте паттерн стратегия, чтобы не раскрывать сложные, специфичные для алгоритма структуры данных;
* В классе определено много поведений, что представлено разветвленными условными операторами. В этом случае проще перенести код из ветвей в отдельные классы стратегий.

**Структура**



**Участники**

* Strategy - стратегия: объявляет общий для всех поддерживаемых алгоритмов интерфейс. Класс Context пользуется этим интерфейсом для вызова конкретного алгоритма, определенного в классе ConcreteStrategy;
* ConcreteStrategy - конкретная стратегия: реализует алгоритм, использующий интерфейс, объявленный
* в классе Strategy;
* Context - контекст: конфигурируется объектом класса ConcreteStrategy; хранит ссылку на объект класса Strategy; может определять интерфейс, который позволяет объекту Strategy получить доступ к данным контекста.

Результаты

У паттерна стратегия есть следующие достоинства и недостатки:

* Семейства родственных алгоритмов.
* Альтернатива порождению подклассов.
* С помощью стратегий можно избавиться от условных операторов. Благодаря паттерну стратегия удается отказаться от условных операторов при выборе нужного поведения.
* Выбор реализации. Стратегии могут предлагать различные реализации одного и того же поведения.
* Клиенты должны знать о различных стратегиях.
* Обмен информацией между стратегией и контекстом.
* Увеличение числа объектов.
* Горячая замена алгоритмов на лету.
* Изолирует код и данные алгоритмов от остальных классов.
* Уход от наследования к делегированию.
* Реализует *принцип открытости/закрытости*.
* Усложняет программу за счёт дополнительных классов.
* Клиент должен знать, в чём состоит разница между стратегиями, чтобы выбрать подходящую.

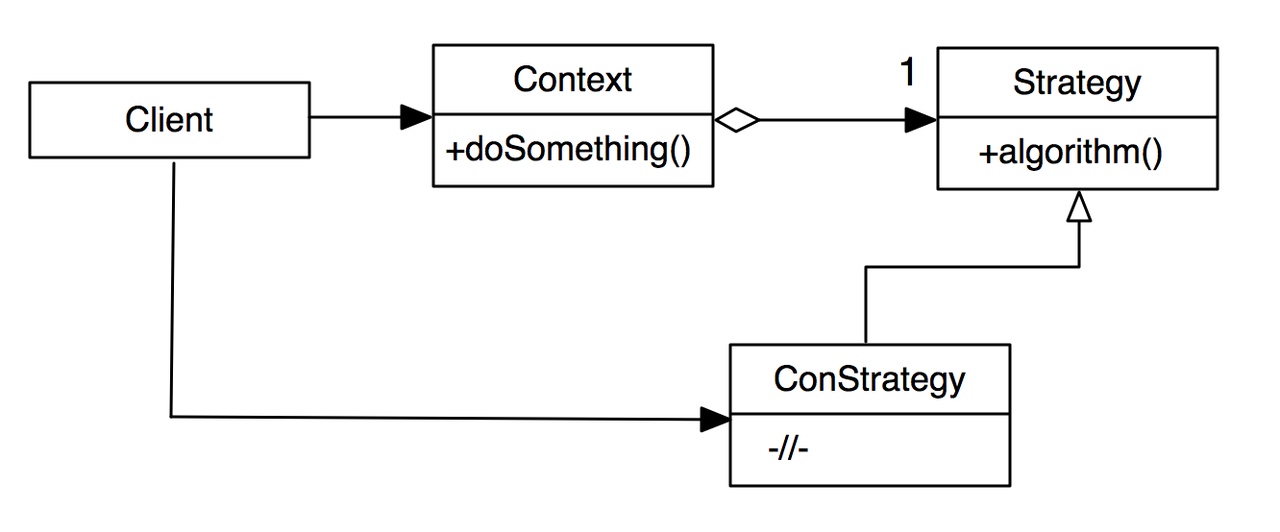
-----

**Стратегия**

Во время выполнения надо менять реализацию конкретного метода.можем делать производные классы с разными реализациями и осуществлять "миграцию" между классами во врем выполнения. Это неудобно.

Идея: изменяемый метод вынести в отдельный класс, который выполняет только это действие. От этого класса создаем производные, реализующие разные вариации этого действия. Паттерн стратегия определяет семейство всевозможных алгоритмов действия.

Клиент устанавливает для нашего класса конкретную стратегию (алгоритм). Работая с объектом, он будет вызывать этот конкретный алгоритм. Во время работы мы можем этот алгоритм поменять.



### КОМАНДА <https://refactoring.guru/ru/design-patterns/command>

**Назначение**

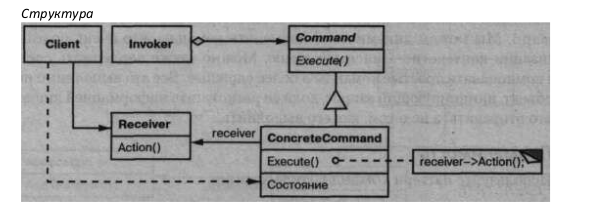
Инкапсулирует запрос как объект, позволяя тем самым задавать параметры клиентов для обработки соответствующих запросов, ставить запросы в очередь или протоколировать их, а также поддерживать отмену операций.

**Применимость**

Используйте паттерн команда, когда хотите:

* Параметризовать объекты выполняемым действием, как в случае с пунктами меню.
* Определять, ставить в очередь и выполнять запросы в разное время.
* Поддержать отмену операций. Операция Execute объекта Command может сохранить состояние, необходимое для отката действий, выполненных командой.
* Поддержать протоколирование изменений, чтобы их можно было выполнить повторно после аварийной остановки системы.
* Структурировать систему на основе высокоуровневых операций, построенных из примитивных.

**Структура**



**Участники**

* Command - команда: объявляет интерфейс для выполнения операции;
* ConcreteCommand - конкретная команда: определяет связь между объектом-получателем Receiver и действием; реализует операцию Execute путем вызова соответствующих операций объекта Receiver;
* Client - клиент: создает объект класса ConcreteCommand и устанавливает его получателя;
* Invoker - инициатор: обращается к команде для выполнения запроса;
* Receiver (Document, Application) - получатель: располагает информацией о способах выполнения операций, необходимых для удовлетворения запроса. В роли получателя может выступать любой класс.

**Результаты**

Результаты применения паттерна команда таковы:

* Команда разрывает связь между объектом, инициирующим операцию, и объектом, имеющим информацию о том, как ее выполнить;
* Команды - это самые настоящие объекты. Допускается манипулировать ими и расширять их точно так же, как в случае с любыми другими объектами;
* Из простых команд можно собирать составные. В общем случае составные команды описываются паттерном компоновщик;
* Добавлять новые команды легко, поскольку никакие существующие классы изменять не нужно.
* Убирает прямую зависимость между объектами, вызывающими операции, и объектами, которые их непосредственно выполняют.
* Позволяет реализовать простую отмену и повтор операций.
* Позволяет реализовать отложенный запуск операций.
* Позволяет собирать сложные команды из простых.
* Реализует *принцип открытости/закрытости*.
* Усложняет код программы из-за введения множества дополнительных классов.

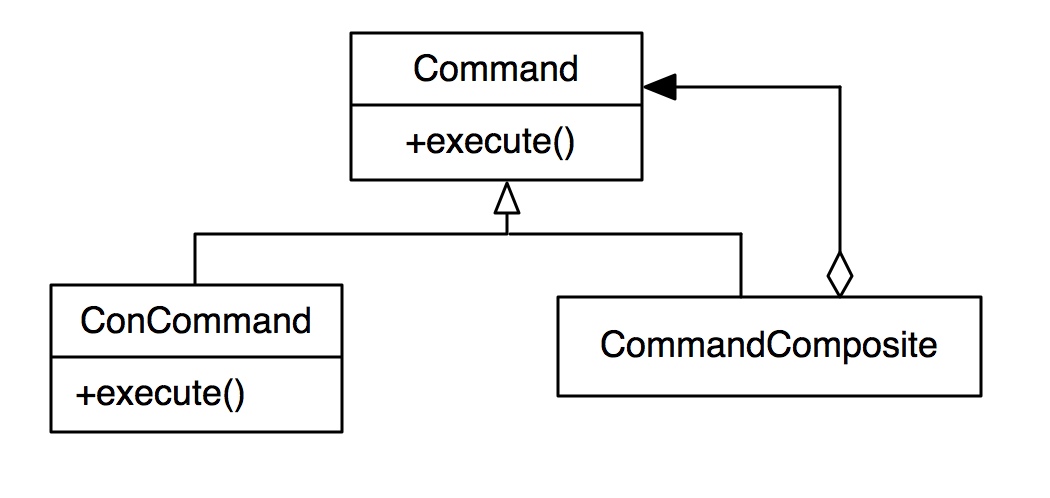
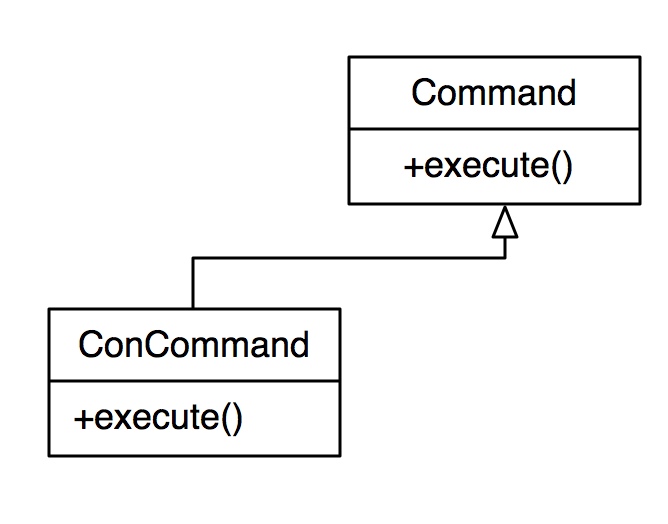
------

Возможны разные запросы. Формализуем каждый запрос, как класс команды. Команда несет данные для выполнения команды и действие, которое необходимо выполнить – например указатель на метод. Конкретная команда наследуется от абстрактной.

(+) Уменьшается связность между объектами за счет их взаимодействия через команды.

(+) Если мы сформировали команду, можем выполнить ее не сейчас, а через какое-то время. Можем сформировать очередь из команд.

(+) Если добавить композит команд, то можно формировать сложные составные команды.



### ЦЕПОЧКА ОБЯЗАННОСТЕЙ <https://refactoring.guru/ru/design-patterns/chain-of-responsibility>

**Назначение**

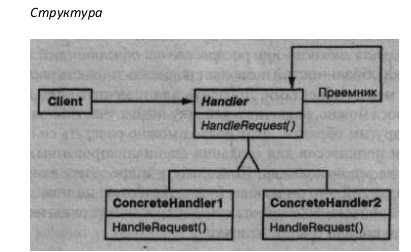
Позволяет избежать привязки отправителя запроса к его получателю, давая шанс обработать запрос нескольким объектам. Связывает объекты-получатели в цепочку и передает запрос вдоль этой цепочки, пока его не обработают.

**Применимость**

Используйте цепочку обязанностей, когда:

* Есть более одного объекта, способного обработать запрос, причем настоящий обработчик заранее неизвестен и должен быть найден автоматически;
* Вы хотите отправить запрос одному из нескольких объектов, не указывая явно, какому именно;
* Набор объектов, способных обработать запрос, должен задаваться динамически.

**Структура**



**Участники**

* Handler - обработчик: определяет интерфейс для обработки запросов; (необязательно) реализует связь с преемником;
* ConcreteHandler - конкретный обработчик: обрабатывает запрос, за который отвечает; имеет доступ к своему преемнику; если ConcreteHandler способен обработать запрос, то так и делает, если не может, то направляет его - его своему преемнику;
* a Client - клиент: отправляет запрос некоторому объекту ConcreteHandler в цепочке.

**Результаты**

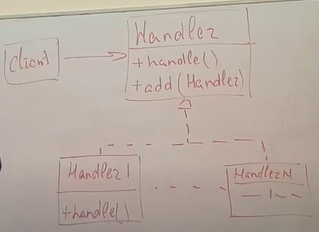
Паттерн цепочка обязанностей имеет следующие достоинства и недостатки:

* Ослабление связанности. 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
* Дополнительная гибкость при распределении обязанностей между объектами.
* Получение не гарантировано. Поскольку у запроса нет явного получателя, то нет и гарантий, что он вообще будет обработан.
* Уменьшает зависимость между клиентом и обработчиками.
* Реализует *принцип единственной обязанности*.
* Реализует *принцип открытости/закрытости*.
* Запрос может остаться никем не обработанным.

----

## Часто задача сводится к тому, что запрос должен обрабатывать не один объект, а несколько объектов.

Идея: создать цепочку обработчиков. Предавать запрос по цепочке обработчиков. Каждый из обработчиков может обработать запрос или, не обрабатывая, передать дальше. Это ложится на обработчика в цепочке.



Варианты реализации - внутренний список (храним контейнер) или рекуррентный вариант.

Когда мы используем цепочку обязанностей:

1. Когда один и тот же запрос может быть выполнен разными способами.
2. Когда у нас есть четкая последовательность в обработчиках.
3. Когда на этапе выполнения мы решаем, какие объекты будут в этой цепочке – на этапе выполнения формируем цепочку обработчиков.

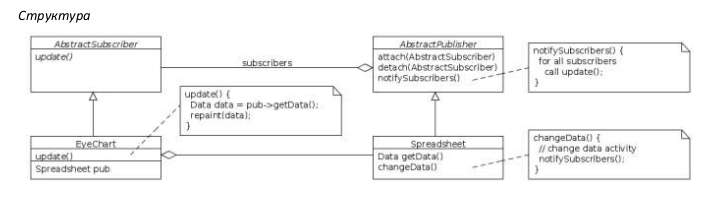
\*В примерах – идем по цепочке, пока не наткнемся на обработчик с состоянием тру, дальше по цепочке не пойдем.

### ПОДПИСЧИК-ИЗДАТЕЛЬ

**Назначение**

Если один или несколько объектов в Вашем проекте должны отслеживать изменения другого объекта – гибким решением будет применение шаблона проектирования Publish-Subscribe.

**Структура**



**Участники:**

* AbstractSubscriber, задает интерфейс подписчиков;
* EyeChart, один из реальных подписчиков (легко могут быть добавлены другие виды диаграмм). Хранит ссылку на издателя, при помощи которой получает его обновленное состояние после получения сигнала об обновлении данных;
* AbstractPublisher, задает интерфейс издателей. Может реализовывать этот интерфейс (не являться абстрактным), т.к. все издатели должны одинаково добавлять/удалять подписчика и уведомлять их об обновлении;
* SpreadSheet, таблица с данными, уведомляющая подписчиков об изменении своих данных.

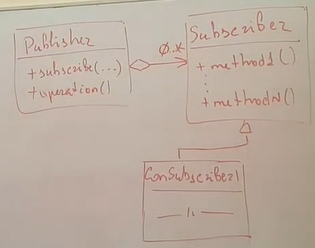
**Результат:**

* Ослабляется зависимость издателя от подписчика. Остается зависимость лишь от абстрактного класса, что позволяет без модификации кода издателя добавлять новые типы подписчиков;
* Появляется возможность добавлять и удалять подписчиков во время выполнения программы. Издатель не располагает информацией не только о конкретных типах Подписчиков, но и об их количестве. Так, например, во многих играх наряду с картой, отображающей состояние игрового мира присутствует мини-карта. Обе карты являются подписчиками, а игровой мир – издателем, при этом для отключения мини-карты достаточно удалить соответствующий объект и снять его с подписки.

-----

Часто запрос надо передавать не одному объекту, а многим. Решать, каким, необходимо во время выполнения.

Паттерн подписчик-издатель задает механизм: тот, кто хочет принять сообщение – подписывается на издателя. Если он подписался – он получает запрос, не подписался – не получает запрос. Получаем механизм, когда группа объектов реагирует на один запрос/изменение одного объекта. Изменение одного объекта-издателя вызывает методы всех подписчиков.



Издатель держит список объектов, которые на него подписались.

Вектор пар: подписчик и метод подписчика, который необходимо вызвать. У издателя должны быть методы подписаться() и отписаться().

Отправленный сигнал может обрабатываться подписчиками синхронно и асинхронно.

(+) не делаем зависимыми издателя и подписчика.

(-) издателю надо держать список подписчиков.

(-) нет никакого порядка в оповещении подписчиков.

(-) если издателей-подписчиков много, связей очень много, каша из связей, как мы их вообще разорвем если нужно будет.

### ПОСРЕДНИК <https://refactoring.guru/ru/design-patterns/mediator>

**Назначение**

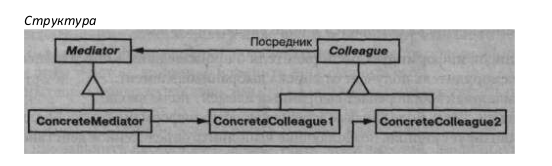
Определяет объект, инкапсулирующий способ взаимодействия множества объектов. Посредник обеспечивает слабую связанность системы, избавляя объекты от необходимости явно ссылаться друг на друга и позволяя тем самым независимо изменять взаимодействия между ними.

**Применимость**

Используйте паттерн посредник, когда

* Имеются объекты, связи между которыми сложны и четко определены. Получающиеся при этом взаимозависимости не структурированы и трудны для понимания;
* Нельзя повторно использовать объект, поскольку он обменивается информацией со многими другими объектами;
* Поведение, распределенное между несколькими классами, должно поддаваться настройке без порождения множества подклассов.

**Структура**



**Участники**

* Mediator (DialogDirector) - посредник; определяет интерфейс для обмена информацией с объектами Colleague;
* ConcreteMediator (FontDialogDirector) - конкретный посредник: реализует кооперативное поведение, координируя действия объектов Colleague; владеет информацией о коллегах и подсчитывает их;
* Классы Colleague (ListBox, EntryField) - коллеги: каждый класс Colleague «знает» о своем объекте Mediator; все коллеги обмениваются информацией только с посредником, так как при его отсутствии им пришлось бы общаться между собой напрямую.

**Результаты**

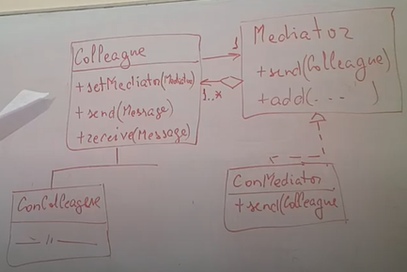
У паттерна посредник есть следующие достоинства и недостатки:

* Снижает число порождаемых подклассов.
* Устраняет связанность между коллегами.
* Упрощает протоколы взаимодействия объектов.
* Абстрагирует способ кооперирования объектов.
* Централизует управление.
* Устраняет зависимости между компонентами, позволяя повторно их использовать.
* Упрощает взаимодействие между компонентами.
* Централизует управление в одном месте.
* Посредник может **[сильно раздуться](https://refactoring.guru/ru/smells/large-class)**.

-----

Есть много связей между объектами, к примеру: издателей-подписчиков много, связей очень много, каша из связей, как мы их вообще разорвем если нужно будет.

Идея: Отдельный класс, ответственный за связи. Связи можно держать как матрицу связности или по-другому. Каждый объект будет обращаться к этому отдельному объекту, а он в свою очередь будет искать нужные связи (выполняет адресацию).



Логику связей реализует конкретный медиатор. Он устанавливает, кому передается сообщение.

### ОПЕКУН <https://refactoring.guru/ru/design-patterns/memento>

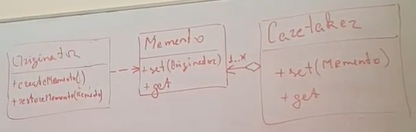
Мы изменили объект, а потом хотим вернуть его к предыдущему состоянию – сделать откат.

Вариант: хранить предыдущие состояния. Если мы возложим ответственность за хранение предыдущих состояний объекта на сам объект – он будет тяжелым.

Идея: выделить отдельный объект, который будет хранить историю состояний другого объекта и, если нужно – он позволит нам вернуться к предыдущему состоянию.

Задача опекуна: сохранение предыдущих состояний другого объекта.

Объект может сделать свой снимок и может на основе снимка восстановить свое состояние. Снимки хранит опекун.



Очевидно, используется, когда нам нужно иметь возможность делать откаты.

(+) Объекту не надо хранить свои состояния.

(-) Опекуном надо управлять. Опекун наделал снимки – какие-то из них больше не нужны. Кто-то должен их очищать.

* Не нарушает инкапсуляции исходного объекта.
* Упрощает структуру исходного объекта. Ему не нужно хранить историю версий своего состояния.
* Требует много памяти, если клиенты слишком часто создают снимки.
* Может повлечь дополнительные издержки памяти, если объекты, хранящие историю, не освобождают ресурсы, занятые устаревшими снимками.
* В некоторых языках (например, PHP, Python, JavaScript) сложно гарантировать, чтобы только исходный объект имел доступ к состоянию снимка.

### 

### ШАБЛОННЫЙ МЕТОД <https://refactoring.guru/ru/design-patterns/template-method>

**Назначение**

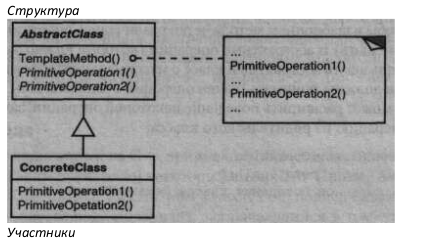
Шаблонный метод определяет основу алгоритма и позволяет подклассам переопределить некоторые шаги алгоритма, не изменяя его структуру в целом.

**Применимость**

Паттерн шаблонный метод следует использовать:

* Чтобы однократно использовать инвариантные части алгоритма, оставляя реализацию изменяющегося поведения на усмотрение подклассов;
* Когда нужно вычленить и локализовать в одном классе поведение, общее для всех подклассов, дабы избежать дублирования кода.
* Для управления расширениями подклассов. Можно определить шаблонный метод так, что он будет вызывать операции-зацепки в определенных точках, разрешив тем самым расширение только в этих точках.

**Структура**



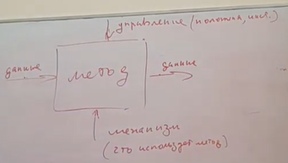
**Участники**

* AbstractClass (Application) - абстрактный класс: определяет абстрактные примитивные операции, замещаемые в конкретных подклассах для реализации шагов алгоритма; реализует шаблонный метод, определяющий скелет алгоритма. Шаблонный метод вызывает примитивные операции, а также операции, определенные в классе AbstractClass или в других объектах;
* ConcreteClass (MyApplication) - конкретный класс: реализует примитивные операции, выполняющие шаги алгоритма способом, который зависит от подкласса.
* Облегчает повторное использование кода.
* Вы жёстко ограничены скелетом существующего алгоритма.
* Вы можете нарушить *принцип подстановки Барбары Лисков*, изменяя базовое поведение одного из шагов алгоритма через подкласс.
* С ростом количества шагов шаблонный метод становится слишком сложно поддерживать.

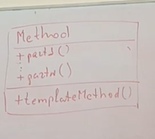
-------

Скелет какого-либо метода. Любую задачу мы разбиваем на шаги.

Есть диаграмма DR0, которая задает нам абстракцию метода.



Задача: реализовать класс, который будет реализовывать это понятие. Храним этапы выполнения метода. И интерфейс его вызова.



### ХРАНИТЕЛЬ

**Назначение**

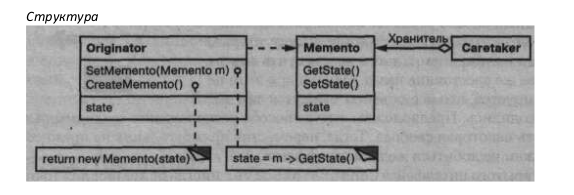
Не нарушая инкапсуляции, фиксирует и выносит за пределы объекта его внутреннее состояние так, чтобы позднее можно было восстановить в нем объект.

**Применимость**

Используйте паттерн хранитель, когда:

* Необходимо сохранить мгновенный снимок состояния объекта (или его части) чтобы впоследствии объект можно было восстановить в том же состоянии;
* Прямое получение этого состояния раскрывает детали реализации и нарушает инкапсуляцию объекта.

**Структура**



**Участники**

* Memento - хранитель: сохраняет внутреннее состояние объекта Originator. Объем сохраняемой информации может быть различным и определяется потребностями хозяина; запрещает доступ всем другим объектам, кроме хозяина.
* Originator - хозяин: создает хранитель, содержащего снимок текущего внутреннего состояния; использует хранитель для восстановления внутреннего состояния;
* Caretaker- посыльный: отвечает за сохранение хранителя; не производит никаких операций над хранителем и не исследует его внутреннее содержимое.

**Результаты**

* Характерные особенности паттерна хранитель:
* Сохранение границ инкапсуляции.
* Упрощение структуры хозяина.
* Значительные издержки при использовании хранителей. 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.
* Скрытая плата за содержание хранителя. Посыльный отвечает за удаление хранителя, однако не располагает информацией о том, какой объем информации о состоянии скрыт в нем.

--------------

*Используется в умных указателях.*

Напишем шаблон хранителя. Необходимо организовать передачу объектов в холдер. Если передавать объект по ссылке, то будет не торт; если по значению – будет создаваться копия, причем может произойти попытка дважды уничтожить один и тот же объект. Создаётся специальная оболочка для передачи – трансфер-капсула. Реализуем сначала её, потом холдер.

### ИТЕРАТОР <https://refactoring.guru/ru/design-patterns/iterator>

Предоставляет способ доступа к элементам контейнера, независимо от его внутреннего устройства. в с++ введён новый цикл for each for (auto& elem: container) { // TODO elem }

это аналогично записи for (auto it = container.begin(); it != container.end(); ++it) { // TODO \*it;

**Суть паттерна**

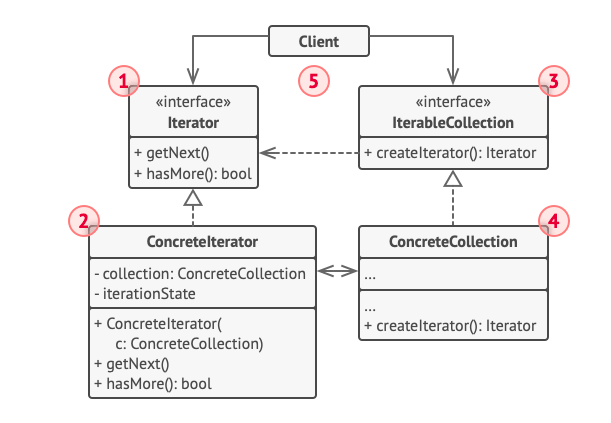
Итератор — это поведенческий паттерн проектирования, который даёт возможность последовательно обходить элементы составных объектов, не раскрывая их внутреннего представления.

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.

**Структура**



1. Итератор описывает интерфейс для доступа и обхода элементов коллекции.
2. Конкретный итератор реализует алгоритм обхода какой-то конкретной коллекции. Объект итератора должен сам отслеживать текущую позицию при обходе коллекции, чтобы отдельные итераторы могли обходить одну и ту же коллекцию независимо.
3. Коллекция описывает интерфейс получения итератора из коллекции. Как мы уже говорили, коллекции не всегда являются списком. Это может быть и база данных, и удалённое API, и даже дерево [**Компоновщика**](https://refactoring.guru/ru/design-patterns/composite). Поэтому сама коллекция может создавать итераторы, так как она знает, какие именно итераторы способны с ней работать.
4. Конкретная коллекция возвращает новый экземпляр определённого конкретного итератора, связав его с текущим объектом коллекции. Обратите внимание, что сигнатура метода возвращает интерфейс итератора. Это позволяет клиенту не зависеть от конкретных классов итераторов.
5. Клиент работает со всеми объектами через интерфейсы коллекции и итератора. Так клиентский код не зависит от конкретных классов, что позволяет применять различные итераторы, не изменяя существующий код программы.  
   В общем случае клиенты не создают объекты итераторов, а получают их из коллекций. Тем не менее, если клиенту требуется специальный итератор, он всегда может создать его самостоятельно.

* Упрощает классы хранения данных.
* Позволяет реализовать различные способы обхода структуры данных.
* Позволяет одновременно перемещаться по структуре данных в разные стороны.
* Не оправдан, если можно обойтись простым циклом.

### СВОЙСТВО

Формализуем понятие свойства – Property – в отдельном классе Удобно создать шаблон свойства (см. примеры). Храним указатель на сеттер и указатель на геттер.

Перегружаем оператор=.

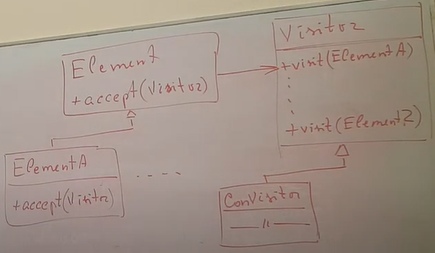
Этот паттерн не любят, потому что при некорректном использовании свойство может отделиться копировании. Это проблема, поэтому мы используем его крайне аккуратно.

### ПОСЕТИТЕЛь <https://refactoring.guru/ru/design-patterns/visitor>

Если мы используем полиморфизм, то в производном классе мы не можем ни сузить, ни расширить интерфейс – производные должны полностью поддерживать интерфейс базы, возможно подменять.

Если нам нужно добавить функционал до этапа выполнения мы можем использовать адаптер. Если необходимо расширить функционал во время выполнения программа - используем визитер.

Чтобы можно было подменять/расширять функционал, в базовом классе добавляем метод accept(Visiter). Соответственно, все производные классы могут подменять этот метод под себя.



Visitor собирает один функционал для разных классов в одно место. Для каждого такого класса в визитере есть свой метод, который посещает именно этот класс. Конкретный визитер реализует эти методы.

(+) В один класс сводим методы, относящиеся к одному функционалу.

(-) В визитере есть методы для конкретного множества классов. А если добавились новые классы – проблема.

(-) Может меняться иерархия – не можем использовать визитер.

* Упрощает добавление операций, работающих со сложными структурами объектов.
* Объединяет родственные операции в одном классе.
* Посетитель может накапливать состояние при обходе структуры элементов.
* Паттерн не оправдан, если иерархия элементов часто меняется.
* Может привести к нарушению инкапсуляции элементов.