Объектно-ориентированное программирование
(Тассов Кирилл Леонидович)

2 модуля, рубежный контроль, экзамен

Лекция 1
Структурное программирование

Идеи программирования:
- Нисходящая разработка
- Использование базовых логических структур
- Сквозной структурный контроль

Подход черного ящика
Вход → f → Выход
Функции: Мы думаем, что они делают, а не как
Разбиваем на подзадачи

Разработка - нисходящая ↓
Логика - восходящая ↑
Данные - нисходящая ↓

Глубина вложенности функции не больше 3, дальше выделяем подфункцию.
Функция 63 строки.

Что дает эта технология:
- Логические ошибки исправляются на ранних стадиях
- За счет выделения абстракции упрощается написание кода
- Повышается надежность программного продукта
- Повторное использование кода в других проектах (библиотеки)
- Плавное распределение ресурсов при разработке ПО
- Упрощается сопровождение кода
- Возникает много естественных контрольных точек
- Сквозной структурный контроль

Лекция 2
ООП

Принципы ООП:
1. Инкапсуляция
2. Наследование

Типы взаимодействий по Хоару:
- синхронное взаимодействие (процессорное)
- Событийное взаимодействие (асинхронное)

Объект - конкретная реализация какого-либо понятия, обладающая характеристиками состояния, поведения, индивидуальности.
Состояние - один из возможных вариантов существования объекта.
Поведение - описание объекта в терминах изменения его состояния и передачи сообщений, данных в процессе воздействия или под воздействием другого или других объектов.

Модель состояния Мура:
Формировать жизненный цикл объекта
1. Выделяем множество состояний, в котором может находиться я объект
2. Выделяем множество событий на которые может реагировать объект
3. Правило перехода (когда происходит определенное действие, в какое состояние объект перейдет)
4. Действия состояния (то что должно выполниться в результате возникновения события)

Роли объектов
1. Реальные
2. Роли (абстракции целей обозначенных человекам)
3. Инциденты (абстракции чего-то произошедшего, случившегося)
4. Объекты взаимодействия (объекты получаемые из отношений между другими объектами)
5. Объекты спецификации (для представления правил, стандартов, критериев)

Отношения между объектами:
1. Использование (старшинства)

3 роли:
- Воздействие (активный объект)
- Исполнение (объект подвержен воздействию, пассивный объект)
- Посредничество

2. Отношение включения
- Один объект включает в себя другой

Классы - такая абстракция множества предметов реального мира, что все эти предметы обладают одними и теми же характеристиками и подчинены и согласовываются с одним и тем же набором правил и поведений.
Отношения:
1. Наследование
2. Использование
3. Включение
4. Метакласс

Домен - отдельный реальный гипотетический или абстрактный мир, населенный отчетливым набором объектов, которые ведут себя в соотвествии с характерными правилами для домена.
Класс в одном домене не требует нахождения класса в другом домене.

Лекция 3

Ссылка - не тип данных, не данное. Еще одно имя какого-то данного.
int I;
int& ai = i;
ai =1; // i= 1
f(a) = 2;
// не использовать!
Перегрузка функций - несколько функций с одним и тем же именем. Тип возвращаемого значения не влияет на перегрузку.
Void sort(Array& Ar, int key = 0); // key- значение по умолчанию
sort(Mas);
Параметры по умолчанию
Методы и функции с переменным числом параметров

Лабораторная работа 1
Структурное программирование!!!
Вьювер (просмотр) каркасной модели (есть фигура 3д). Вершины соединяем ребрами, получает каркасную модель. Отображение, перенос, масштабирование. Модель считывать из файла.
Оконный интерфейс. Графика - любые библиотеки. Интерфейс - ООП. Задача - структура.

Лекция 4

struct, union, class.
Union - поля по одному адресу в памяти. Не может быть базовым и производным

class (имя (имя типа)) [:(список баз)]
{
    //члены класса, 3 уровня доступа
private: // частные, доступ только из других членов этого класса

protected: // защищенные, доступ членам этого класса и производным этого класс

public: // общий доступ, только методы

}

struct - уровень доступа по умолчанию public

class - уровень доступа по умолчанию private
Объявляем в хедере. Реализации в cpp.

```
class A
{
private:
	int a;
public:
	int f();
}

int A::f() // можно рекомендацию inline
{
	return this->a;
}

A obj, *p;
obj.f();
p->f();
```

Еще .* u ->*

```
class A
{
private:
	int a;
protected:
	int b;
public:
	int f();
}
//уровень доступа - по умолчанию private

class B:{private/protected/public}A
{
private:
	int c;
protected:
	int d;
public:
	int g();
}

B obj;
```

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

Тогда для B
a - не имеет доступа
b, f(), c - private доступ
d - protected
g() - public

Класс ничего не знает о производных.
При этой схеме происходит полная смена интерфейса.

При схеме наследования **protected -** интерфейс базового класса оставили доступным для наследников.
a - не имеет доступа
c - private доступ
d, f(), b - protected
g() - public

При схеме наследования **public -** все члены остаются с уровнем доступа который у них был. Расширение интерфейса.
a - не имеет доступа
c - private доступ
d, b - protected
g(), f() - public

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

Конструктор
имя::имя([параметры])
[:раздел инициализации]
{
}
Конструктор вызывается не для объекта. Для инициализации. Не используется указатель *this*.

```
class A
{
private:
	int a;
	const int cb;
	static int sc; // член класса
	const static int csd = 1; // инициализация только при определении
public:
	A(int ia)
		:a(ia), // ok
		cb(ia), // ok
```

```
        sc(ia); // не нужно
        csd(ia); // error
{
    a = ia; // ok
    cb = ia; // error
    sc = ia; // не нужно
};
```
Член класса вне инициализация: int A::sc=0; // с типом считается объявление, можно инициализировать

```
class B
{
 public:
    static int f(); // вызов B::f(), работает с полями класса.
    int y() const; //  метод вызывается как для константных так и нет
    int y(); // можно перегрузить
};
```

Явное приведение типа
(тип)выражение
тип(выражение)

Конструктор можно перегружать. Не может быть const, static, volatile. Конструктор не наследуется.

Если не определить: то появляются по умолчанию, конструктор без параметров и конструктор копирования (принимает ссылку на себе подобного).
Если хоть один конструктор определи, конструктор по умолчанию не создается.

Конструктор копирования
Потайное копирование объекта. Если есть динамическое или потоки. Обязательно реализовать этот конструктор. Принимает константную ссылку на себя.
Конструктор переноса

Делегирующий конструктор
В одном конструкторе можем вызвать другой этого же класса.

```
class A
{
public:
    A(int i) {};
    A():A(0){}
};
```

С 11 версии возможно наследование конструкторов

```
class A
{
public:
        A(int i);
};

class B:public A
{
public:
        using A::A;
};

B obj(2);
```

Конструктор с переменным числом параметров

```
Class Complex
{
private:
        double re, im;
public:
        Complex():Complex(0,r){}
        Complex(double r):Complex(r,0){}
        Complex(double r, double I);
        Complex(const Complex &C); //конструктор копирования

        //explicit - запрет неявного вызова конструктора
        //Complex(const Complex &C)=delete; // удаление конструктора копирования
                                                    //(запрет копирования)
}
Complex a(), //функция
          v // вызывается 1 конструктор
          b1=Complex(1.),  // вызывается 2 конструктор, явный вызов конструктора
          b2(2.), // вызывается 2 конструктор, явный вызов конструктора
          b3=3., // вызывается 2 конструктор, неявный вызов конструктора
          b4 = {4.}, // вызывается 2 конструктор, неявный вызов конструктора C++11
          b5{5.} // вызывается 2 конструктор, неявный вызов конструктора C++11
          c1=Complex(6.,7), // вызывается 3 конструктор
          c2(8.,9), // вызывается 3 конструктор
          c3={10.,11}, // вызывается 3 конструктор
          c4{12.,13}, // вызывается 3 конструктор
          d1 = Complex(c1), // вызывается конструктор копирования, явный
          d2(c2), // вызывается конструктор копирования, явный
          d3=c3; // вызывается конструктор копирования, неявный
```

```
class A
{
Public:
        void f(int);
        void f(double)=delete;
        A(int);
        A()=default;
};
```

Деструктор

~ИмяКласса(); //не принимает параметров!

Не может быть константным. Можно (но не надо) вызывать явно. Может быть виртуальным.

Оператор new

new <имяТипа>() // выделение памяти и создание объекта

new <имяТипа>[размерность] //для нескольких объектов

Оператор delete

delete <указатель> //удаление одного

delete [] <указатель> //удаление нескольких

<тип>*<идентификатор>=new <имя>[(<пер>)|[<размерность>]]

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.

Лекция 6

```
class B. public V
{
protected:
        void _draw();
public:
        void draw()
        {
                v ::_draw();
                _draw();
         }
}
```

Используя указатель на класс, можно обратиться к любому полю.

ООП - программирование без возможности выбора (switch)

```
v *p = new A;
p->draw(); //вызовется класс B
```

```
class A
{
public:
        virtual void f(); //деструктор может быть виртуальным (при наследовании
                                                                должен)
};
class B
{
public:
        void f() override/*обязательно писать*/ final/*в производных методах
         не сможем подменить этот метод*/; //подмена базового метода
};
```

Если класс абстрактный явно пишем модификатор abstruct.

```
A& index(A * vec, int i)
{
        return vec[i];
}
```

Объекты класса A имеют доступ ко всм членам класса B:

```
class A
{
}
class B
{
friend class A;
}
```

Дружба не наследуется (сын моего друга, мне не друг). Дружба не транзитивна. Можем сделать другом не весь класс, а только какой-то метод.

```
class A
{
public:
        int f();
}
class B
{
friend int A::f();
}
```

Нельзя!

```
{
        A* p = new A;
        p->f(); // не выполнится все после этого при исключительной ситуации
        delete p;
}
```

Перегрузка операторов

Нельзя: . , * , :: , sizeof , typeid

Операторы:

= , () , [] , -> , ->* , <знак>=

Явное и неявное приведение типа это тоже оператор

a+b:

```
class Complex
{
private:
        double re, im;
public:
        Complex operator + (const Complex & C) const
        {
                return (this->re + C.re, this->im + C.im);
        }
};
```

( ) - бинарный оператор, можно перегружать только один раз

[] - смещение относительно адреса, при перегрузке данное любого типа, создание ассоциативных массивов

-> - унарный оператор, возвращает указатель на объект

```
class A
{
public:
        int f();
};
class B
{
public:
        A*operator ->();
};

B obj;
obj->f(); //(obj.operator->())->f();
```

```
int f();
int (*pf)();
pf = f // Имя любой функции - ее адрес в памяти
pf(); // Вызов функции по адресу

int A::f();
int(A::*pf)();
pf = &A::f; // вычисление адреса для метода
A obj, *p = &obj;
(obj.*pf)();
(p->*pf)();
```

Лекция 7

```
class MyInt
{
private:
	int i = 0;
public:
	MyInt& operator++() //префиксный ++оператор
	{
		++i;
		return *this;
	}

	MyInt operator++(int) // постфиксный оператор ++
	{
		MyInt temp(i);
		i++;
		return temp;
	}
};
```

Приведение типов

```
class A
{
private:
	int i = 0;
public:
	operator bool()
	{
		return i == 0;
	}
};
A obj;
if (obj)
```

```
class A
{
 private:
	int I = 0;
public:
	explicit operator bool()
	{
		return i == 0;
	}
};
A obj;
if (obj) // нельзя
if (bool(obj)) // только явно
```

Временный объект

```
Vector v1, v2, v3;
v1 = v2 + v3; // Создается временный объект справа (результат действия)
```

Копирование и перенос

```
class Vector
{
private:
	double *ar;
	int count;
	static void copy(Vector& v1, const Vector& v2);
public:
	Vector(const Vector& v); // копирование
	Vector(Vector&& v); // перенос
	Vector& operator = (const Vector& v); //копирование
	Vector& operator = (Vector& v); //перенос
};

Vector::Vector(const Vector & v)
{
	copy(*this, v);
}

Vector::Vector(Vector && v)
{
	this->count = v.count;
	this->ar = v.ar;
	v.ar = nullptr;
}
```

```
//a=b копируем объект b в a
Vector& Vector::operator = (const Vector & v)
{
	delete[] this->ar;
	copy(*this, v);
}

//a=b+c переносим временный объект b+c в a
Vector& Vector::operator = (Vector&& v)
{
	delete[] this->ar;
	this->count = v.count;
	this->ar = v.ar;
	v.ar = nullptr;
}

void Vector::copy(Vector& v1, const Vector& v2)
{
	v1.count = v2.count;
	v1.ar = new double[v1.count];
	for (int i = 0; i < v1.count; i++)
		v1.ar[i] = v2.ar[i];
}
```

Обработка исключительных ситуаций

```
 try
{
	throw <тип>(<параметр>); //создание исключительной ситуации
}
catch(<тип>& <переменная>)
{

}

class A
{
public:
	void f() noexcept
	//noexcept(true)
	//noexcept(false)
	//throw(<тип>) // функция может сгенерировать исключительную ситуацию
};
```

std::exception

```
class MyException : public std::exception
{
public:
	virtual const char* what(const char *msg) const noexcept;
}
```

Шаблоны
template<class | typename <параметр>>

```
template <typename Type>
unsigned length(FILE *stream)
{
	return filelength(fileno(stream)) / sizeof(Type);
}

size = length<double>(stream); //вариант вызова шаблона
```

Лекция 8

Можно также создавать шаблоны классов.

```
// Первый аргумент тип данных, второй его размер
template <typename T, size_t n>
class Vector
{
private:
	T ar[n];
	size_t size = n;
public:
	Vector();
	size_t count();
	...
};
```

Все шаблонные методы мы выписываем в заголовочном файле, так как мы не можем скомпилировать их.
Базовый класс - нешаблонный.

```
template <typename T, size_t n>
Vector<T, n>::Vector() {};
```

Полная или частичная специализация

- **полная**

template <>
class Vector<double, 5> {}; // пример специализации

- **создание объекта**

Vector <int, 2> v;
Шаблон не накладывает ограничения на параметры.

- **частичная**

template <typename T1, typename T2>
class A {};
template<typename T>
class A<T, T> {};
template<typename T>
class A<T, int> {};
template<typename T1, typename T2>
class A<T1, T2*> {};*

A<int, float> a1; // ни одна специализация не подходит, создается класс по шаблону
A<float, float> a2; // используется специализации class A<T, T> {};
A<float, int> a3; // класс создается по специализации class A<T, int> {};
*A<int *, float *> a4; // class A<T1*, T2*> {};*
A<int, int> a5; // неоднозначность по выбору специализации, которая приводит к ошибке class A<T, T> {}; или class A<T, int> {};
*A<int *, int *> a6; // неоднозначность по выбору специализации, которая приводит к ошибке class A<T, T> {}; или class A<T1*, T2*> {};*

Параметры шаблона по умолчанию

template <typename T1, typename T2 = float> // параметры по умолчанию последние в списке шаблонов
class A{};
A<float> a1;
A<float, int> a2,

Шаблоны с перечным числом параметров

Не знаем изначально что нужно, какие поля данного
Такой тип создается при использовании! Смысла в таком кортеже нет.

```
template <typename T1, typename ...Args>
T sum(T v1, T v2, Args… args)
{
        return v1 + sum(v2, args…);
}

template <typename T>
T sum(T, v)
{ return v; }
```

Сложение 2 чисел разного типа

```
template <typename T1, typename T2>
auto sum (T1 x, T2 y) -> decltype(x + y)
{
	return x + y;
}
```

auto obj = <выражение>;

Пространство имени

```
namespace <имя>
{
	int f();
}
```

Обращение

<>::f();

Определение пространства имен

```
using namespace <имя>;
f();

{
	A *p = new A;
	p->f();
	delete p;
}
```

Умный указатель, оболочка над указателем

```
{
	holder <A> obj (new A);
	obj -> f();
}
```

3 умных указателя в с++

Хранители(умные указатели)

unique_ptr <mun> // легкая прозрачная оболочка -

unique_ptr обеспечивает строгое владение указателя (он за него отвечает).

Можно передать указатель другому хранителю (не обязательно *unique_ptr).*

std:: unique_ptr<A> obj(new A)

shared_ptr<A> obj1(new A), **- совместное владение**

obj2(make_shared<A>());

shared_ptr<A> obj3(obg1);

obj1.use_count() obj3 = obj1;

obj1.reset()

obj1.unique()

weak_ptr<A> **- мягкое владение**
obj = make_shared<A>
shared_ptr<A> obj1(new A);
obj = obj1;
shared_ptr<A> a = obj.lock();
obj.use_count()
obj.expired()
obj.lock()

Лабораторная работа
Создание контейнерного библиотечного класса (шаблонного)
1. Вектор математический (не массив), складывание, вычитание, коллинеарность и тд
2. Множества (каждый элемент уникален)
3. **Список прямого доступа (односвязный)**
4. Матрица

Требования:
1. Контейнерный класс
2. Операции
3. Просматривать содержимое контейнера по итератору
4. Интерфейс библиотечного класса избыточен

Список 2 указателя: 1 - начало, 2 - конец, и еще несколько для работы со списком.
```
struct List
{
	Node *first, *last;
};
struct Iterator
{
	Node *Current;
};
```

Шаблоны и паттерны проектирования
- порождающий
- поведение
- структурный

Порождающие паттерны

Методы решения проблем(паттерны)
- **Фабричный метод** - создать объект

```
class AbstractCreator
{
	virtual unique_ptr<Product>;
```

```
        CreateProduct() = 0;
};

template <typename TProd>
class ConCreator : public AbstractCreator
{
public:
        virtual unique_ptr<Product> CreateProduct() override
        {
                return unique_ptr<Product>(new Tprod());
};

class Creator
{
public:
        shared_ptr<Product> GetProduct();
protected:
        virtual shared_ptr<Product> CreateProduct() = 0;
private:
        shared_ptr<Product> product;
};

shared_ptr<Product> Creator::GetProduct()
{
        if (product)
                product = CreateProduct();
        return product;
}
```

- **Абстрактная фабрика -** создавать объекты разных классов, но связанных между собой
- **Прототип** - реализуют метод, который называется *clone* , на основе которого создают новый объект

```
class Prototype
{
public:
        virtual unique_ptr<Prototype> Clone() = 0;
};
```

- **Строитель**

```
template<typename Type>
class Singleton
{
public:
```

```
        static Type& instance()
        {
                return *(myInstance ?myInstance:(myInstance = new Type()));
        }
private:
        static Type* myInstance = nullptr;
};
```

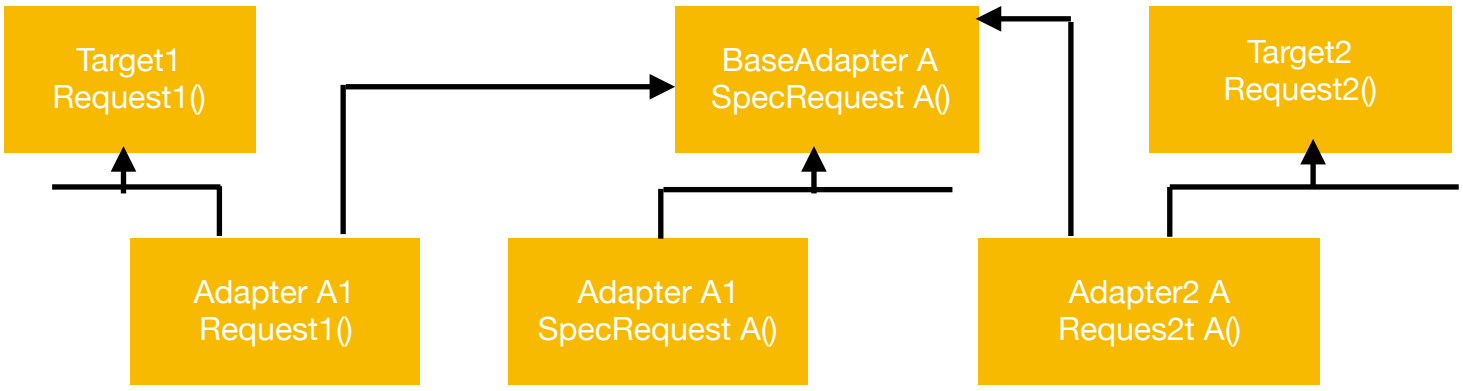
-Singlenton - антипаттерн
Singleton() = delete;
Singleton<Type>& operator=(const Singleton<Type>&) = delete;
Singleton(const Singleton<Type> &) = delete;

На основе второй реализации фабричного метода пул-объект.

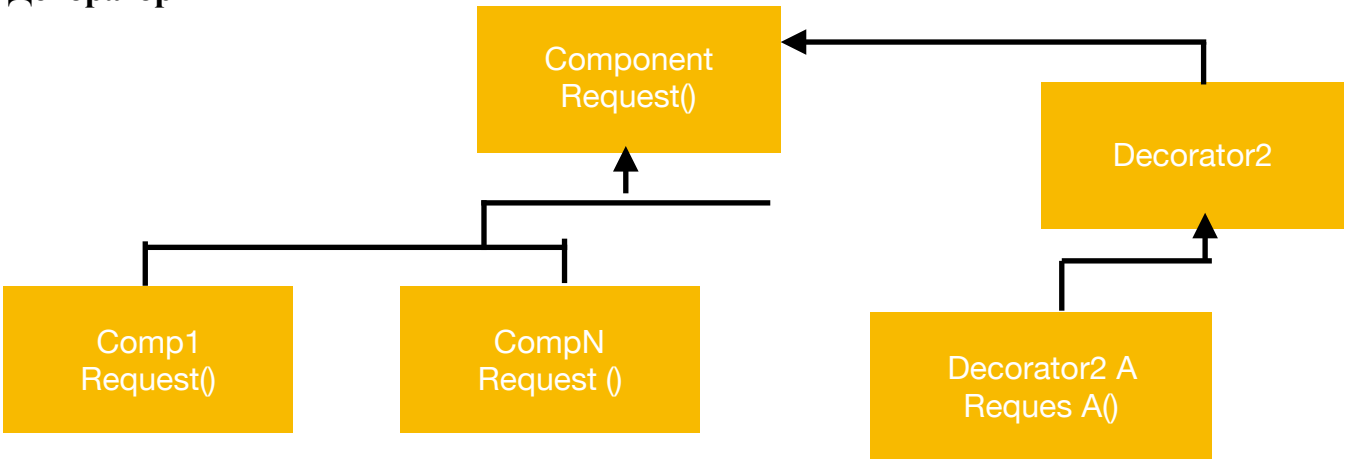
- Полуобъект
Проблемы:

1. Невозможно всем угодить
2. Расширение: изменить все существующие методы
    1. Создание просто класс с ограниченным набором методов
    2. Пользователь сам создает интерфейс (оболочку для этого класса)
    3. Упрощается модернизация

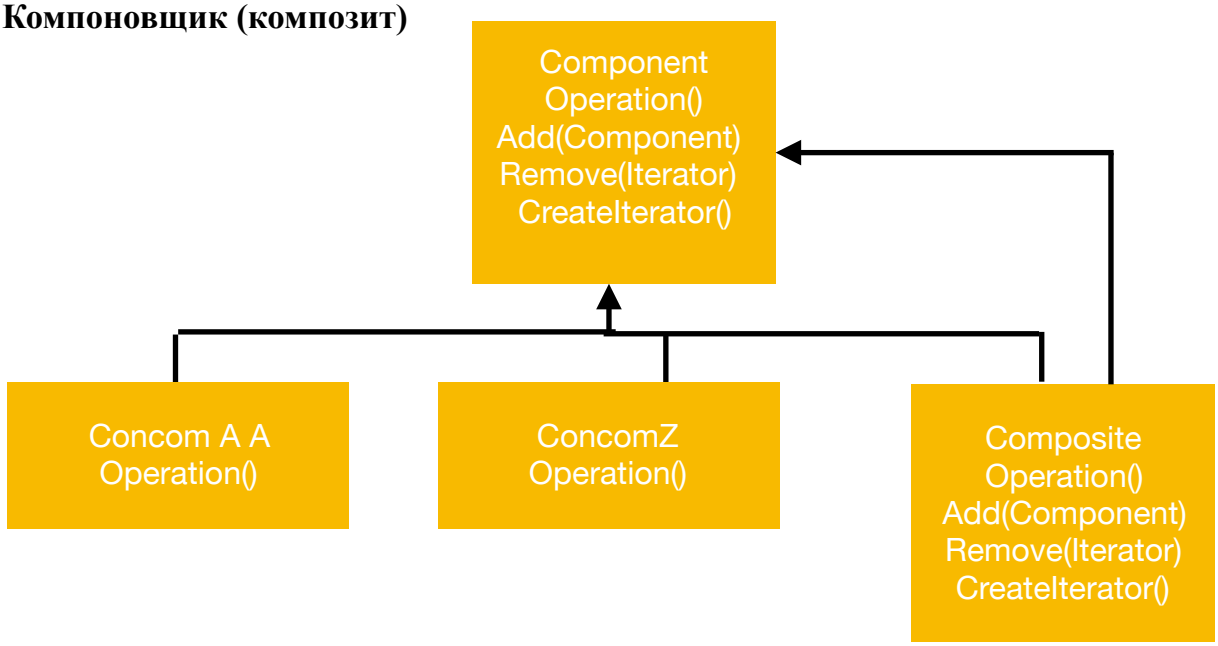
Паттерн адаптер



- **Декоратор**

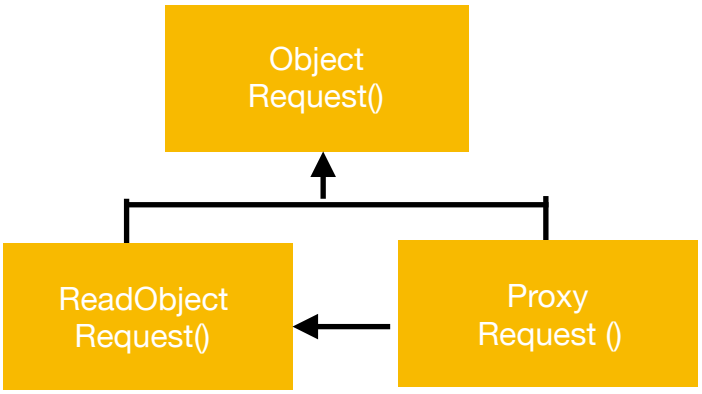


- Компоновщик (композит)



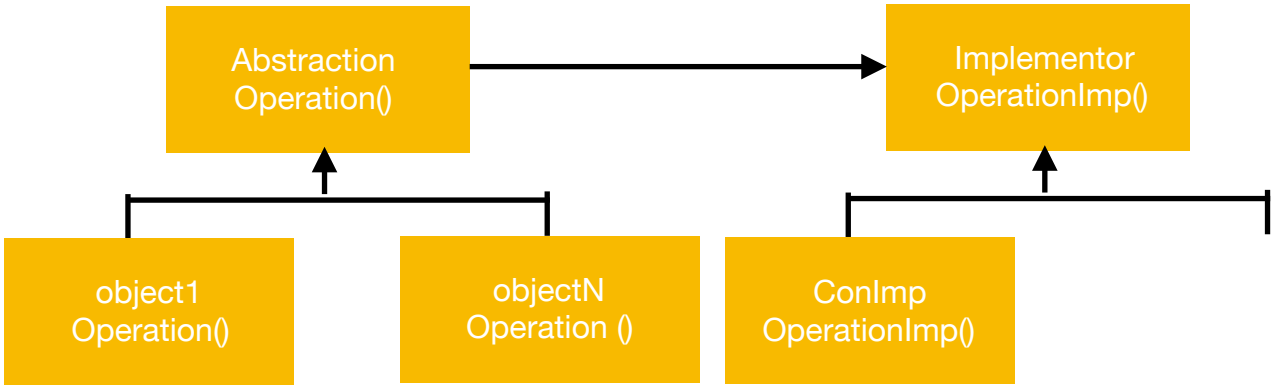

```
class Component
{
public:
        virtual ~Component();
        virtual bool Operation() = 0;
        virtual bool Add(Component *) {return false;}
        virtual bool Remove(Iterator &) {return false;}
        virtual Iterator CreateIterator() {return Iterator();}
};
```

- **Заместитель (или прокси)**



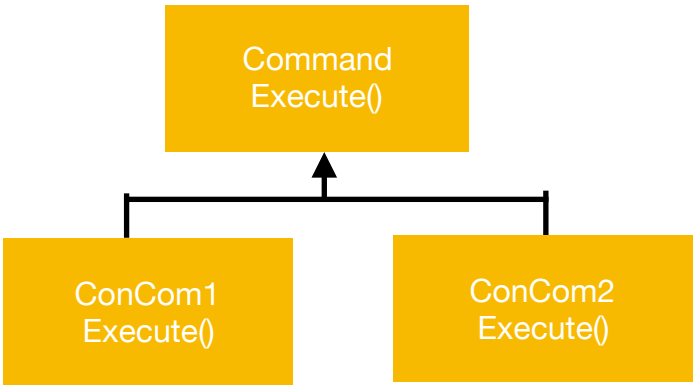
- **Мост**

2 иерархии: сущность и реализация



- **Приспособленец**

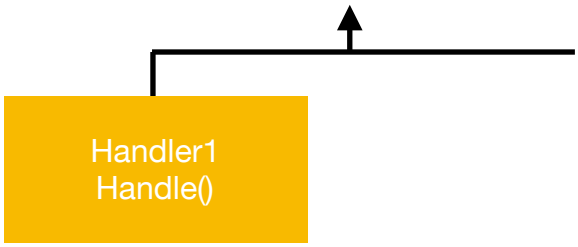
Могут быть объекты похожий друг на друга(порождение и храним одно и тоже) хранится везде и много.
Не хранить для каждого объекта одно и тоже.
Если подобный объект был порожден, используем копию или сам объект с помощью различных преобразований.




```
class Command
{
public:
	virtual void execute() = 0;
};

template <typename Receiver>
class SimpleCom : public Command
{
public:
	typedef void (Receiver::*Action)();
	SimpleCom(shared_ptr<Receiver> r, Actions);
	void execute() override
	{
		((*_r).*_a)(); // вызов функции по указателю
	}
private:
	shared_ptr<Receiver> _r;
	Action _a;
};
```

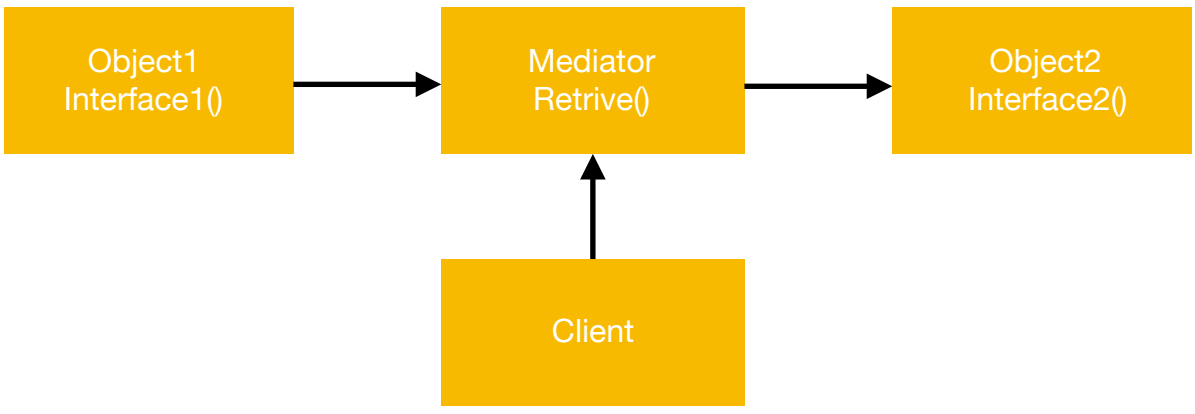
- **Цепочка обязанностей**

Что-то выполнили и перешли по цепочки к следующему. Есть базовый класс, задача которого вызвать действия производного класса, который выполняет действия и следующий переход. (список в структуре класса)



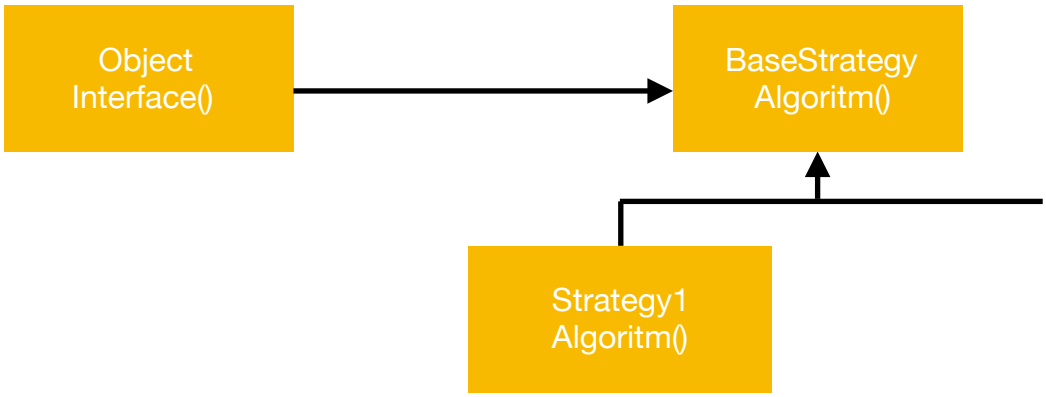

```
class Handler()
{
public:
	void add(shared_ptr <Handler> elem)
	{
		if (next)
			next->add(elem);
		else
			next = elem;
	}
	virtual void handle()
	{
		if (next)
			next->handle();
	}
private:
	shared_ptr<Handler> next;
};

class Handler1 : public Handler
{
public:
	void handle() override;
	{
		if ()
			Handler::handle();
	}
};
```

3 лабораторная - паттерн подписчик/издатель



Паттер опекун

Паттерн стратегия



Паттерн состояния
Держит состояние объекта, используется в 3 лр. Входит в КМС - конечная модель состояний - паттерн проектирования.
Паттер визитер
Посетитель
Паттерн итератор
Паттерн индексатор
Паттерн интерпретатор

Объектно-ориентированный анализ

Этапы:
1. Анализ
2. Проектирование - перевод документов из модели в проект
3. Эволюция
4. Модификация

Всегда можно вернуться назад!

РД подход - рекурсивный дизайн, система разворачивается постепенно, от ядра.

Этапы
1. Добавление нового класса
2. Изменение реализации класса
3. Изменение представления класса
4. Реорганизация структуры класса
5. Изменение интерфейса класса

1. Схема доменов
2. Проектная матрица
3. Модель связей подсистем

   ———————————————-

   Для каждого домена

4. Модель взаимодействия подсистем
5. Модель доступа к подсистемам
6. Информационная модель

   ———————————————-
7. Описание классов и их атрибутов
8. Описание и формализация связей
9. Модель взаимодействия объектов
10. Состояние объекта
11. Список событий
12. Модель доступа к объектам
13. Таблица процессов состояний

    ———————————————

    Для каждого класса и может быть связи (связь можем формировать как класс)
14. Модель состояний

    ———————————————
15.

4 лабораторная - 1 работа, реализовать объектами, построить модель (выделить сущности, формализация (отношения), посредники). Задача рассмотрения одного объекта, но возможно появление нескольких или даже групп. Изменение положения наблюдателей, несколько наблюдателей (с разных ракурсов). Связь наблюдателя с объектом.

Объектно-ориентированный анализ

Подход белого ящика
1. Выделение сущностей физических объектов
2. Выделение атрибутов

Любая характеристика, которую мы выделяем, абстрагируется как **отдельный атрибут.**
Идентификатор - множество из одного или нескольких атрибутов, с помощью которых можно честно идентифицировать объект.
Каждая сущность на информационной модели.

<Номер> <Имя> (<Короткая запись имени>)
- идентефикатор - - - - - -

Атрибуты
1. Описательные - представляет факты, присущие каждому объекту данной выделанной сущности.
2. Указывающие - идентификатор или его часть

3. Вспомогательные
    - Атрибуты возникающие в результате формализации связей между сущностями
    - Атрибуты состояния

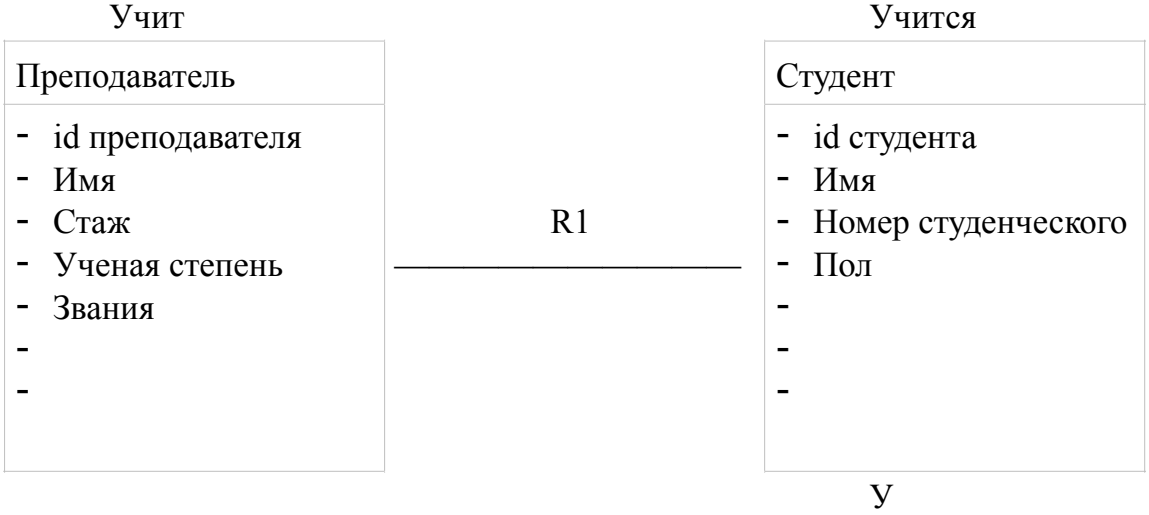
Описательный
1. Описать причину, по которой мы выделили этот атрибут
2. Важно кто задает или использует этот атрибут

Указывающий
1. Важно кто задает значение

Вспомогательный
1.

На атрибуты накладываются **правила**.
1. Любой объект в любой момент времени для каждого атрибута должен иметь значение и оно должно быть одно.
2. Атрибут должен быть простым, не содержать внутренних структур
3. Если объект имеет составной идентификатор, каждый атрибут, который является часть идентификатора должен быть характеристикой всего объекта, а не его части
4. Каждый атрибут, не являющийся частью идентификатора представляет характеристику объекта указанного идентификатора

Связи - абстракция набора отношений, которая возникает между предметами или объектами в реальном мире.



1. **Безусловная связь** - и один, и другой объекты участвуют в связи
2. **Условная связь** - объект, может не участвовать в связи
3. **Биусловная связь** - оба объекта могут не участвовать

Множественная связь:
1 к 1
1 ко многим
Многие ко многим

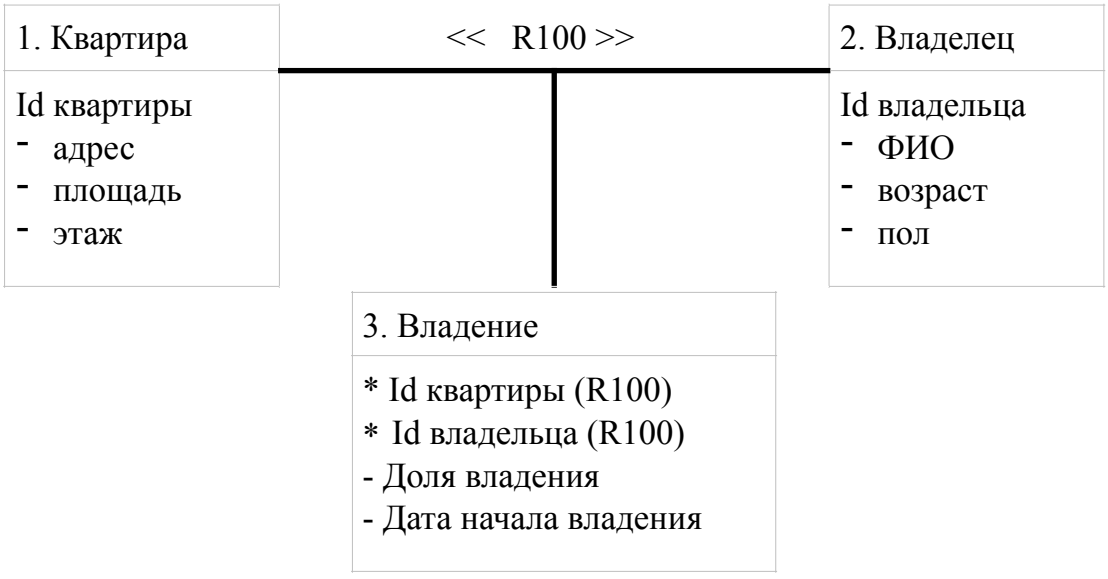
В результате перестановок получаем 10 видов связи.

1. Идентификатор
2. Формулировка связи каждому объекту
3. Вид связи (Условность, множественность)
4. Обосновать основание связи
5. Как была связь формализована

Если связь 1 к 1, то кто-то должен держать эту связь, добавляем вспомогательный атрибут.

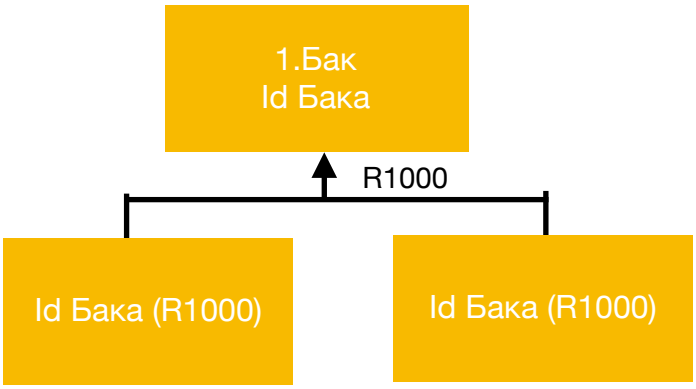
1. Муж		2. Жена
… Идентификатор связи ФИО жены(R10)	R10	ФИО

Связь 1 ко многим нормализуется со стороны многих.
Многие ко многим - добавляется ассоциативный объект, который отвечает за связь.



Композиция связей — связь является следствием других связей.
R3 = R1 + R2

Подклассы и суперклассы.



Модель Мура

- Выделяем множество состояний, в которых он может находится
- Множество событий, при котором изменяются состояний
- Правила перехода, в какой новое состояние он перейдет при возникновении в данном состоянии события
- Вспомогательный атрибут состояние
- Действие состояния, каждому состоянию ставим в соответствие действие, которое выполняется при переходе объекта в состояние

Диаграмма переходов состояний

Состояние объекта - набор правил оиний поведения.

- Состояние создания
- Заключительное
  - Объект становится подвижным
  - Объект удаляется

Выделяем значения события — короткая фраза о том, что происходит с объектом
Не все переходы возможны.

Те события, которые переводят объект из одного состояния в другое, должны в себе держать **идентификатор объекта**. (и нести одни и те же данные)
Действие - это деятельность или операция, которая должна быть выполнена когда объект перешил в состояние.
Что может делать:

- выполнять вычисления
- порождать события, как для себя, объектов класса, так и для любых других объектов; порождать события вне области анализа
- выполнять все действия с таймером (создавать, удалять, устанавливать, запускать и сбрасывать, считывать время)
- работать с атрибутами своего объекта, класса или других классов

Обязательно:

- любое действие не должно оставлять противоречивости атрибутов и связей (если связь 1к1 и удаляется объект, то удаляется и другой или переходит в другой класс)
- задача изменения атрибута состояния

Описание действий:
- нарисовать схему или алгоритм, если оно большое
- если небольшое, то можем написать на псевдокоде на модели состояния под состоянием

Событие никогда не теряется, оно им принимается, когда объект не выполняет никакое действие. После того как событие принимается каким-то действие оно перестает быть событием.

В каждый момент времени одно действие для объекта.
Действия разных объектов могут выполняться одновременно.

Асинхронная схема взаимодействия. Подход к проектированию другой.

При проектировании свойства, состоящее из двух методов гетер и сетер.

Каждому состоянию ставится действие. Сколько состояний, столько и действий (обработчиков).
ЛР3. C++ CLI

Правило перехода
Строим таблицу переходов состояний.
Каждая строка - возможное состояние объектов.
Столбец - событие.
Как объект в данном состоянии (каждое состояние нумеруем, уникальным номером) реагирует на событие.
3 возможных перехода:
- переход в какое-то другое состояние
- если объект не реагирует, то ставится -
- если выполниться не может, ставим крестик
    - можем обработать как исключительную ситуацию - удаляются объекты
    - лучше нетипичную работу добавлять к состоянию (обработка и переход в ошибочное состояние, из него можно попытаться вывести объект)

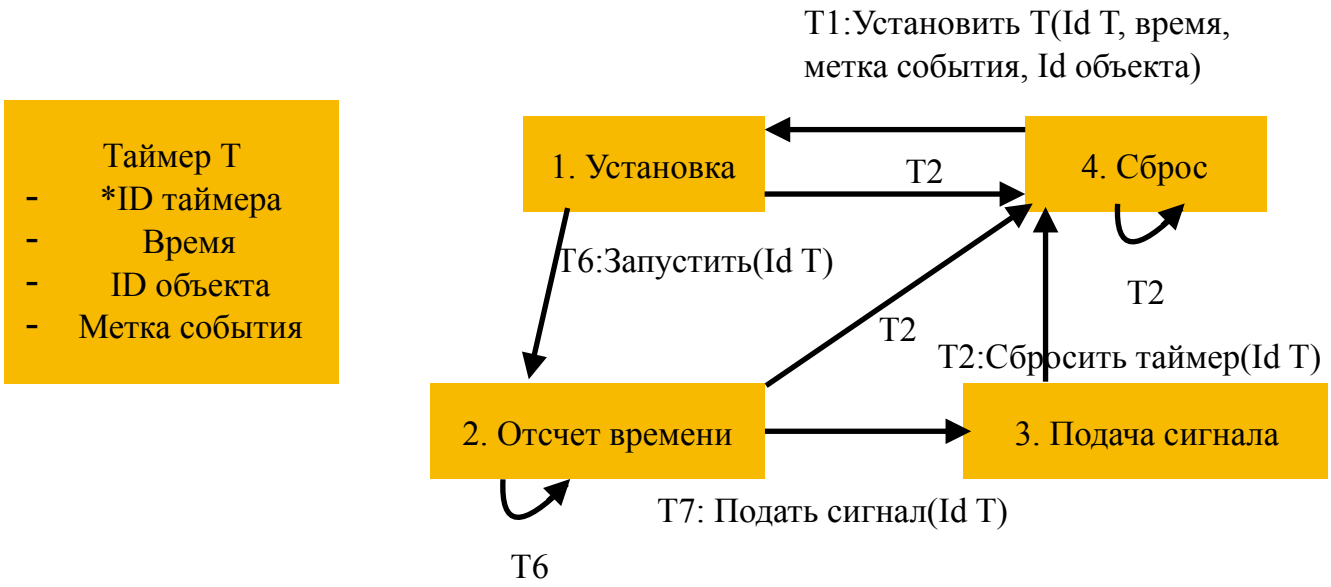
3 лабораторная
Паттерн подписчик-издатель. Смоделировать работу лифта. Выделяем несколько объектов: кабина, двери, сам лифт, блок управления, умный лифт (знает кол-во пассажиров (объект)). Есть разные объекты и возникают связанные жизненные циклы. Состояние одного объекта связано с состоянием другого (но соответствие неполное). Состояния связаны и определяют друг друга.
- выделяем несколько объектов
- строи модели состояний
- смотрим какие события приводят к состояниям этого объекта или других (связь моделей состояния)

Использовать:
- c++ qt на сигналах и слотах (псевдо асинхронность)
- c++ cli

Паттерн подписчик-издатель
Делегат - прототип функции, указатель на функцию.
ref - динамическое размещение.

```
delegate void Eventhandler(Object^ source, double d);
//Издатель
public ref class Manager
{
public:
	event Eventhandler^ OnHandler;
	void Method()
	{
		onHandler(this, 100);
	}
};
//Подписчик
public ref class Watcher
{
public:
	Watcher(Manager^ m)
	{
		m->OnHandler += gcnew Eventhandler(this, &Watcher::f);
	}
	void f(Object^ source, double d) {...}//обработчик того события (после подписки)
};
//Main
{
	Manager^ m = gcnew Manager();
	Watcher^ w = gcnew Watcher(m);
	m->Method();
}
```

^ - указатель, управляемый код
Подписываемся на событие и событие держит объект



	T1	T2	T6	T7
1. Установка	—	4	6	X
2. Отсчет времени	—	4	2	3
3. Подача сигнала	—	4	X	X
4. Сброс	1	4	X	X

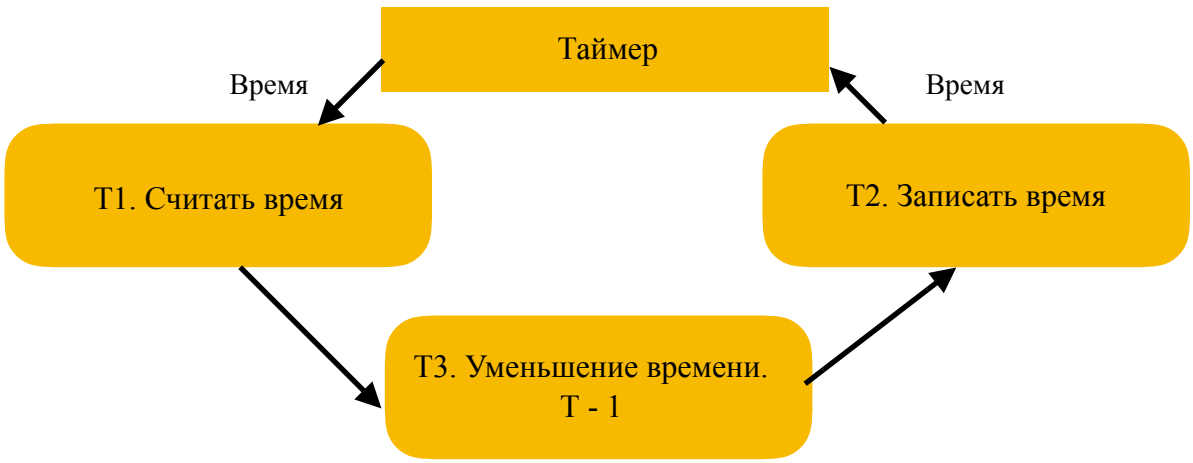
```
if это_время = 0 then Порождаем T7(Id T)
else
        это_время := это_время - 1
        if это_время = 0 then Порождаем T7(Id T)
        else Порождаем T6(Id T)
```

Объекты спецификации тоже могут иметь жизненный цикл
Принципы при формировании жизненного цикла:

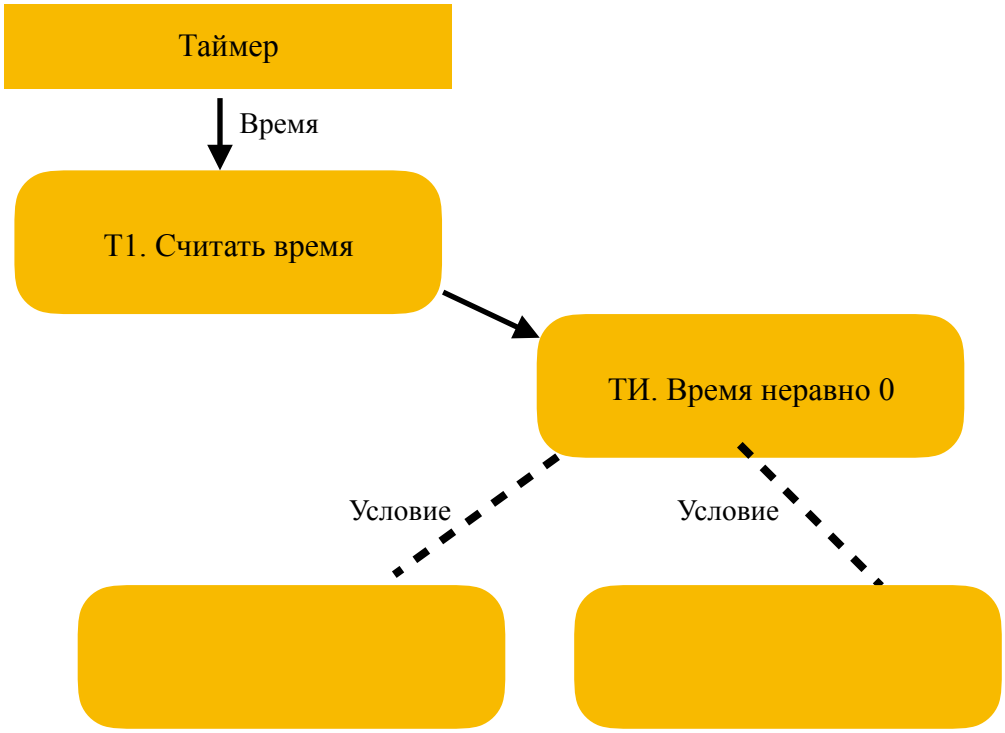
1. Цель
2. Контекст

Процессы:

1. Аксессоры - читает атрибуты, записывает атрибут, создает и уничтожает архив данных (объекты). Любой доступ к архиву. Помечаем именем, что они делают. ДПДД:



2. Генераторы события - четко выделяем процесс, единственное действие которого сгенерировать событие. Имя и метка события. Можем породить событие только когда входы доступны.
3. Преобразования - что делает процесс (вычисления).
4. Проверка - проверка архива данных.

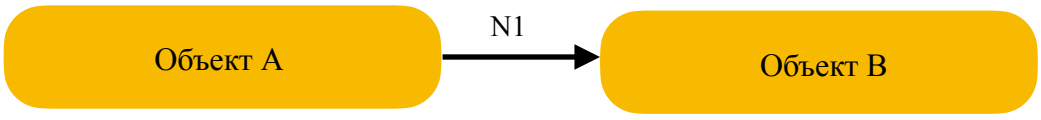


Данные события всегда доступны. Данные архива всегда доступны.
Повторяющиеся действия расписывать используя условные процессы. (циклы)

Создают одни и те же атрибуты, события, выводы условного управления.

Id процесса	Тип	Название процесса	Где использовать	
			М. С.	Дейст
	А (аксессорный)			

Выделив аксессор строим диаграмму: **модель доступа к объекту.**
Каждый объект рисуется в вытянутом овале



Объект А использует аксессуар объекта В.

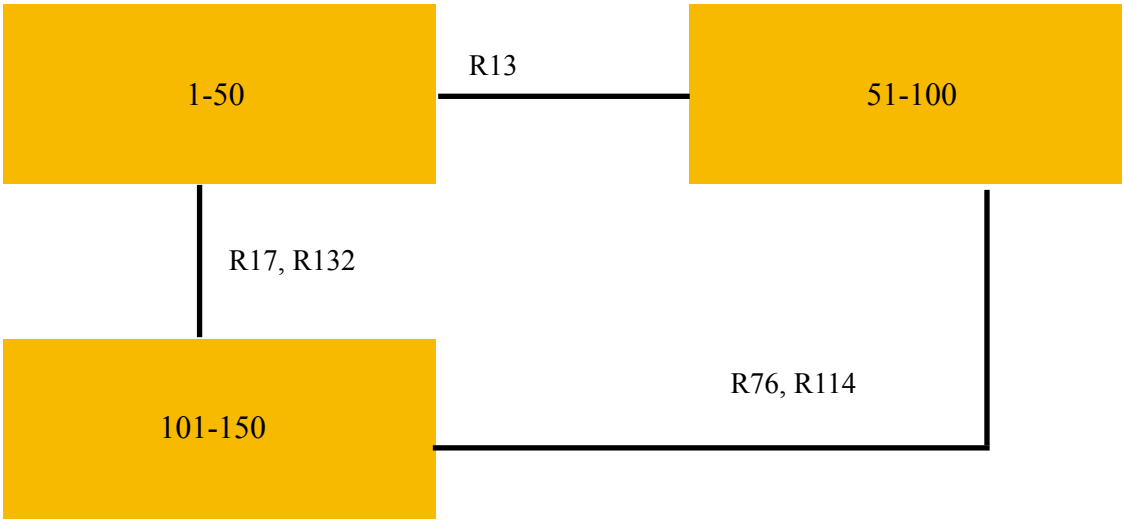
Домены и подсистемы.

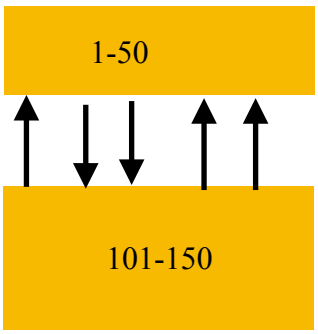
Домен - отдельный реальный гипотетический или абстрактный мир, населенный отчетливым набором объектов, которые ведут себя в соотвествии с характерными правилами для домена.

Типы:
1. Прикладной домен - задача, предмета я область того что мы решаем.
2. Сервисный - набор средств, которые мы используем: библиотеки, то что нам предоставляет какой-то функционал. Как сторонние так и наши библиотеки.
3. Архитектурный - задает общий механизм и структуру управления в нашем ПО. Как правило в основу включается паттерн шаблонный метод. Входят таймеры. Паттерн КМС.
4. Реализация - функционал, которы обеспечивается ОС.

Если один домен использует механизмы другого домена, то тогда между доменами существует мост Тот домен, который используется - клиент.
Клиент рассматривает мост, как набор предложений, ему безразлично кто их обеспечивает.
Сервер - рассматривает мост, как набор требований к себе.
Если количество классов в домене больше 50. То домен развивается на подсистемы.

№ диаграммы:
1. моделей связей подсистем - диаграмма сущность связь
2. модель доступа к подсистеме



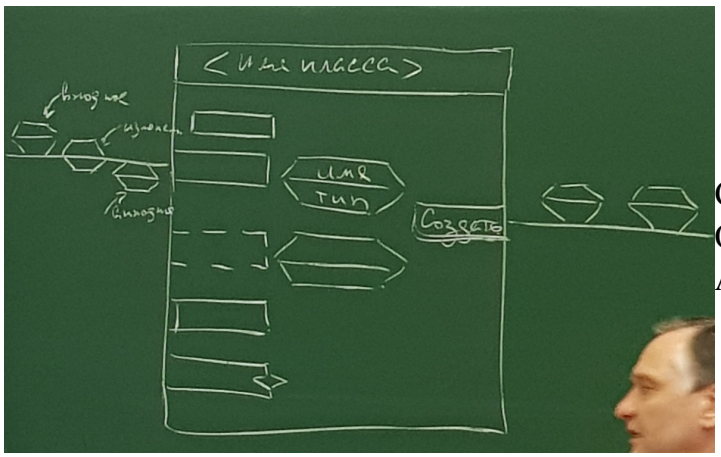


5 лабораторная:
Диаграммы, моделирование.
Выделить сущности, атрибуты, связи.
Выделить жизненный циклы для объектов, построить модель и канал управления.
Расписать действия состояний (алгоритм). Строим ДПС, выделяем аксессорные связяи.

Бытовой прибор - стиральная машина.

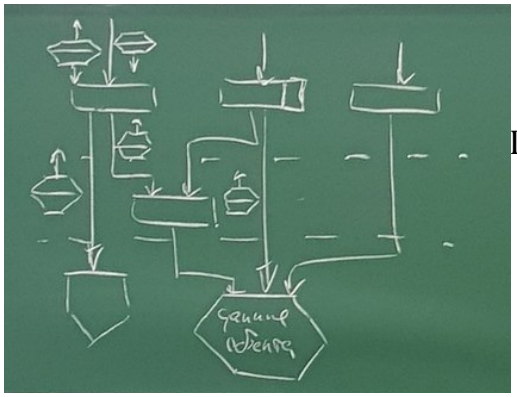
Диаграмма класса, диаграмма зависимости, диаграмма наследования

Диаграмма класса



Сущность связь - атрибуты
Обработчики состояний
Аксессорные процессы

Диаграмма структуры класса



Потоки данных и внутренне взаимодействие

Диаграмма зависимости
Взаимодействие сущность связь
Схема включения
Схема использования

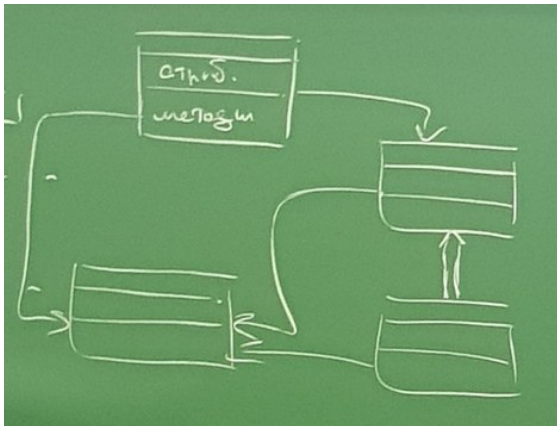
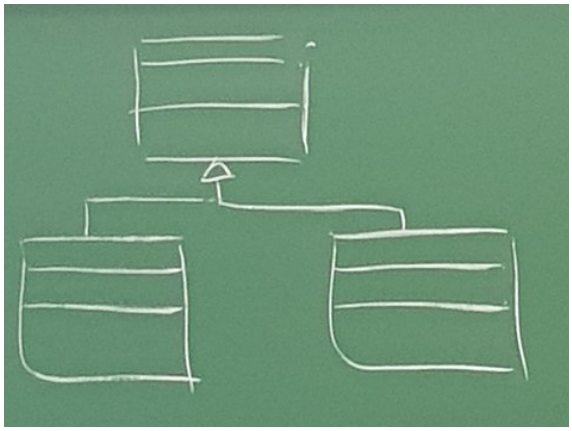


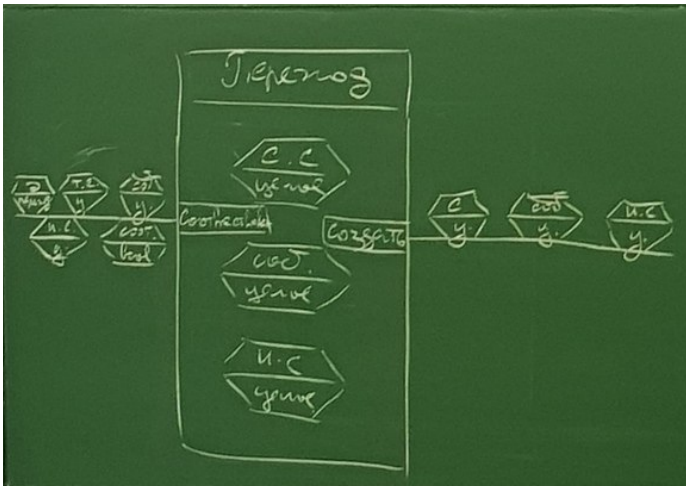
Диаграмма наследования



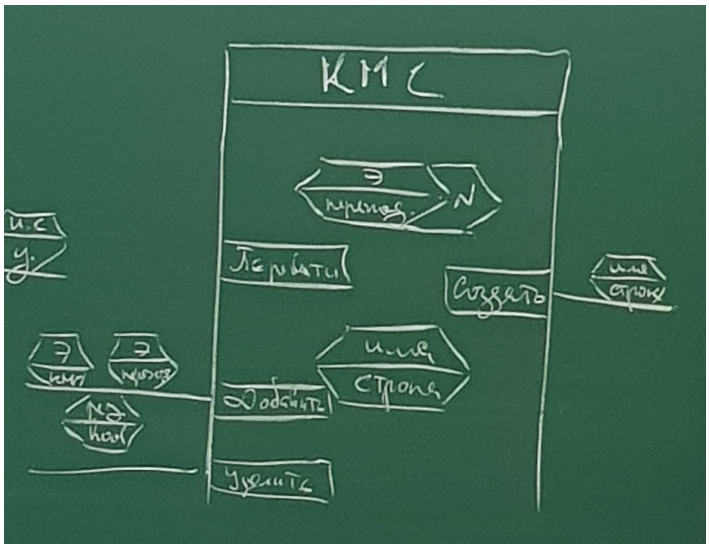
Архитектурный домен
Задача - задать архитектуру системы.
Сущности:

- Активный экземпляр
- КМС
- Переход

Переход
Диаграмма:



КМС - конечная модель состояний, хранит все переходы
Диаграмма:



Активный экземпляр

Диаграмма: