07.02.2014----------

Тассов Кирилл Леонидович

3 модуля, 5 лабораторных работ

1й лабы не будет, только через неделю

Появление технологии

Структурное программирование, автор – IBM, 1961 год. «Усовершенствованные методы программирования.

В основе любой технологии лежит декомпозиция, приведение чего-то сложного к простым действиям, разбитие задачи на подзадачи. В основе структурного программирования лежит алгоритмическая декомпозиция – разбиение задачи на подзадачи по ДЕЙСТВИЮ, отвечая на вопросы «что нужно делать».

IBM выделили три основные идеи технологии:

1. Нисходящая разработка программы
2. Использование базовых логических структур
3. Сквозной структурный контроль

При разбиении на подзадачи выделять нужно не более 7 подзадач. Однако каждую из них в свою очередь можно разбивать дальше, пока не придем к решению простейших задач.

Глубина вложности конструкции не должна превышать 3. Оптимальный размер подзадачи - <200 строк. Количество передаваемых на модуль данных также не должно превышать 3 элементов. Все данные передаются явно через список параметров, не важно что за данные – переменные или константы. Для этого данные структурируются – структура данных должна соответствовать уровням абстракции (при разбиении задач).

IBM предложили на всех этапах (за исключением анализа – начального этапа) использовать нисходящий подход: проектирование, кодирование, тестирование. На низшем уровни логика должна отсутсвовать; она концентрируется на более высоких уровнях. Отлаживая верхние модули, мы избавляемся от логических ошибок.

В структурном программировании существует жёсткое требование: тесты подготавливаются ДО реализации. В современных условиях это достаточно тяжело.

Разработка: сверху-вниз, логика: снизу-вверх, данные: сверху-вниз

+ логические ошибки исправляются на ранней стадии
+ ставящий и реализующий задачу – как правило разные люди
+ структурный подход даёт жёсткую связь между заказчиком и исполнителем
+ легко распределять задачи между программистами
- зависимые модули не принимают решения за более высокие
- важно чётко формулировать действия – они не должны пересекаться, чтобы одно и то же не выполнялось два раза
* модули не должны прерывать работы программи; исключение – самый низкий уровень
* выделение и освобождение ресурсов должно происходить на одном уровне

Алгоритм любой сложности можно реализовать с использованием трёх базовых логических структур: условие, следование, цикл.

Использование сквозного структурного контроля

Размер рабочий группы желательно не должен превышать 7; однако группой необходимо руководить, уровень руководителя должен быть выше чем у подчиненных. Однако руководитель не вовлечен в написание кода. IBM предложили перенести это руководство (контроль) на уровень программиста. Производится формализация задачи и устраивается «контрольная сессия» с коллегами.

14.02.14----------

Изменение функционала приводит к изменению структуры данных. Это приводит к изменению уровней абстракции, и в конце концов нарушается стройность иерархической структуры. Наступает момент, когда дешевле написать программу заново, чем модифицировать старую. На сопровождение и модификацию требуется гораздо больше средств, чем на написание программного кода.

Хоар: совместное использование записей. При изменении данных нужно выявлять места, где мы работаем с этим данным. Хоар предложил: четко выделить, что можно делать с этим данным (выделить функции для работы с данным); извне доступа к данным нет, доступ есть только у этих функций. Следовательно, не нужно будет перелопачивать всю программу.
В дальнейшем эта идея получила название инкапсуляции – объединение данных с действиями над этими данными в одно целое, доступ к данным только через эти действия.

Наследование – «надстройка» нового функционала над старым, не изменяя его.

Хоар выделил два вида взаимодействия объектов в физическом мире:
* «аксессорное взаимодействие», синхронное взаимодействие (один объект синхронно взаимодействует с другим). Безразлично, с каким объектом взаимодействовать – полиморфизм.
* «событийное взаимодействие», асинхронное. Происходят события, которые приводят к изменению состояния объекта.

Основные понятия ООП

Объект – реализация конкретного типа, обладающая характеристиками состояния, поведения и индивидуальности.

Состояние – один из возможных вариантов условий существования объекта.

Поведение – описание объекта в терминах изменения его состояния и передачи данных (сообщений) в процессе воздействия или под воздействием других объектов.

Индивидуальность – сущность объекта, отличающая его от других объектов.

На основе выделяемых состояний объекта строится так называемая модель Мура. Она состоит из множества состояний – для объекта формируется «жизненный цикл» состоящий из состояний и событий, переводящих его из одного состояния в другое. Каждое событие представляет инцинденту, указание на то, что должен произойти переход из состояния в состояние. Правила перехода определяют, в какое новое состояние перейдет объект, при возникновении с ним в данном состоянии данного события.

Действие – деятельность или операция, которые должны быть выполнены над объектом.

Категории объектов:
- реальные (физические) – дом, стул и т.д.
- роли – абстракции цели\назначения, (части оборудования, человека) – студент, преподаватель, дипломат
- инцинденты – абстракции произошедшего или случившегося – наводнение, выборы
- объекты взаимодействия – полученные из отношений между другими объектами – перекресток, порог, взятка
- спецификации – используются для представления правил, стандартов, критериев, протоколов – ПДД, распорядок дня, расписание поездов

Категории отношений:
- отношение использования, старшинства – каждый объект может играть три роли: воздействия (может воздействовать на другие, сам не подвержен воздействию других объектов; активные объекты), исполнения (может только подвергаться воздействию со стороны других; пассивные объекты), посредничества (выступает как в роли воздействуемого, так и в роли исполнителя)
- отношение включения – объект может включать в себя другие объекты

Класс – абстракция множества предметов реального мира. Все объекты имеют одни и те же характеристики, и подчинены и согласовываются с одним набором правил и линий поведения.

Отношения между классами:
- наследование – на основе одного класса выделяется другой, обладающий дополнительными свойствами

- использование – объект может использовать методы и атрибуты другого объекта
- наполнение – класс может включать в себя другие классы
- метакласс – класс над классом

Домен – отдельный реальный\гипотетический мир, населенный определенным набором объектов, которые ведут себя в соответствии с характерными для домена правилами и линиями поведения. Каждый домен образует отдельное связанное единое целое.

На этапе разработки хедеры подключаются к .cpp а не к другим .x. Однако на момент окончательный сборки можно переподключить .x к другим .x, чтобы было понятнее.

~~~~~~~~~~~~~~~~~~~~

Задача: написание вьювера каркасной модели
~~~~~~~~~~~~~~~~~~~~

21.02.14----------

(Глеб Андреевич ...)

Нововведения C++

Перегрузка – одно и то же имя может соответствовать нескольким функциям, если различаются их параметры (по типу)

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

```
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 является «вторым именем» переменной a, 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.

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;
```

Операция возвращения ссылки достаточно опасна – нельзя возвращать ссылки на локальные переменные. В основном это используется только при перегрузке присваивания.

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

Классы

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;
protected: //доступ есть внутри класса и во всех его наследниках
	Int b;
public: //доступны для внешнего кода
	Int f();
}; //класс заканчивается ; как и структура

//Классы описываются в заголовочных файлах .h. Методы же определяются в .cpp
```

```
MyClass a, *b = &a;
a.f();
b->f();
```

Размещение полей класса (переменных) 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 IPv4
{
	int IntegerForm;
	char ByteForm[4];
};
```

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

```
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;

– – – – – – – – – – – –

<class.h>

```
class B
{
private:
	const int a; //поле должно быть инициализировано вместе с экземпляром класса, и не может изменяться.
public:
	B(int a); //конструктор
	void f() const; //константный метод – может вызываться для константных объектов; не может работать с константными полями
	void g();
};
```

<class.cpp>

```
#include “class.h”
B::B(int val): a(val) //, c(7) – всё это инициализация
{
}

const B obj; //в константном экземпляре можно вызывать ТОЛЬКО константные методы: obj.f() НО НЕ obj.g()
```

– – – – – – – – – – – –

Инлайновые функции

Родственник макросов. Объявление функции через кейворд inline – компилятор будет заменять все встречающиеся использования на НЕПОСРЕДСТВЕННО КОД, как define.

Статические члены классов

```
class A
{
public:
	static void func();
private:
	static int b;
	static const char* text;
};
```

Инициализация проходит через

```
int A::b = 0;
const char* text = "abc";
```

С ОБЯЗАТЕЛЬНЫМ УКАЗАНИЕМ ТИПА. Без этого компилятор ругнётся на недоступность переменных.

Статический метод не привязан к конкретному экземпляру.

28.02.2014----------

Требование: поля класса не должны быть публичными, только приватные или протектные. Порядок расположения: вначале приватные, потом протектные, потом публичные (метод белого ящика). Но в случае библиотечных классов инвертируется, паблик; протектед; приват.

По возможности компилятор делает методы класса инлайновыми.

Конструкторы и деструкторы. Создание и уничтожение объектов

Конструкторы отрабатываются в порядке появления в коде объектов.

У каждого объекта существует жизненный цикл. В начале этого цикла (при создании объекта через classname object; или object = new classname) вызывается конструктор. Деструктор всегда вызывается неявно, при удалении объекта через delete object; также деструкция происходит при завершении выполнения функции (или при выходе из области видимости), в порядке, обратном созданию объектов – но только «явно» заданных объектов, не через *a = new.

Локальные статические объекты – конструктор вызывается при первом определении объекта, деструктор вызывается после выполнения функции main но до вызова деструкторов внешних и внешних статических переменных.

Возврат объекта – объект уничтожается когда «необходимость в объекте отпадает». Зависит полностью от компилятора.

Метод класса – static, не передаётся указатель this на объект. 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. Деструктор также необходимо объявлять явно.

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

Конструктор не может быть константным, volatile, virtual, статичным. Не наследуется.

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.

```
class A: /*<список баз>*/
{
private:
	int a;
	const int cb;
	B obj;
public:
	A(int param) : cb = 123 //:раздел инициализации; здесь можно дописать вызов парент-конструкторов и инициализацию конст-параметров
	{	//тело конструктора выполняется, когда под объект выделена память
		//объекты базы создаются в том же порядке, в котором они перечислены в списке
		//аналогично с инициализацией членов класса
	}
}
```

Рассмотрим работу с классами – на примере комплексных чисел.

```
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); //3 //копирование
};

Complex::Complex()
{
	re = 0.0;
	im = 0.0;
}
Complex::Complex(double r, double i) : re(r), im(i)
{}
Complex::Complex(double r)
{
	re = r;
	im = 0.0;
}
Complex::Complex(Complex& C)
{
	re = C.re;
	im = C.im;
}


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;
}
```

Деструкторы – работают непосредственно с экземпляром класса, this передаётся. НЕ МОГУТ быть константными или статически, могут быть виртуальными. Если выделялись ресурсы под объект, то обязан быть описан деструктор, его задача – очищение ресурсов. Также на деструкторы возлагается функция разрыва связей.

Схемы наследования

Наследование – создание нового класса на основе старого. Три схемы наследования – приватное, публичное, протектное. По умолчанию, наследование происходит приватным.

```
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
a 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
a 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
a не наследуется.
c - приватная, b,d - протектные, f,g - публичные.
интерфейс ДОПОЛНЯЕТСЯ.
*/
```

07.03.2014----------

Private – полная схема интерфейса. Public – расширение интерфейса.

Если нужно надстроить новый интерфейс поверх старого, закрыв чтото старое и перекрыв чтото 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 торт, нужно оборачивать в какой-нибудь другой метод.

В производном классе можно определить такой же метод как в базовом – одно имя, один список параметров, один тип возврата. Метод в производном классе будет доминировать над методом базового класса.

В каких случаях использовать наследование? Два варианта: наследование когда выделяем базовый класс из нескольких; когда есть класс, его нужно расщепить на несколько разных.

Выделение общей базы (в порядке приоритета): ВСЕГДА выделяется класс, если есть общая схема использования. Сходство между набором операций (над объектами разных классов выполняются одинаковые операции). Совершенно разный функционал и разное использование, но могут быть выделены операции, имеющие единую реализацию. Желательно выделять базу даже в том случае, когда объекты разных классов фигурируют вместе в «дискуссиях по проекту» (когда возможно в дальнейшем может проявиться общность классов).

Расщепление класса: два подмножества операций класса используются в разной манере (в разных областях программы, домена). Группы операций имеют несвязанную реализацию. Один и тот же класс фигурирует в разных, не связанных между собой частях – лучше этот класс разделить.

Множественное наследование

ООП использует рекурсивный дизайн – постепенное разворачивание программы от базовых классов к более специализированным. C++ 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
}
```

Патовая ситуация – если не мы писали А и В, но пытаемся объединить их в один свой. В С можно было бы доминантно определить новые методы взамен старых, но члены классов всё равно будут пересекаться.

Также, иерархия наследования при МН может быть достаточно сложной. Введем понятие «прямой базы» - от которой НЕПОСРЕДСТВЕННО наследуется новый класс, и «косвенной базы» - прямая база прямой базы и так далее, we need to go deeper. Прямая база может использоваться один раз (: public A, public A низя!), а вот косвенная – сколько угодно.

Тем не менее, желательно, чтобы база входила только один раз. Для этого используется «виртуальное наследование». Если подобъект класса был создан, идет проверка на это, и ещё раз он не создается.

```
class A {};
class B: virtual public A {};
class C:         public A {};
class D: public B, public C {};
```

Всё ок.

```
class A {};
class B:       public A {};
class C: virtual public A {};
class D: public B, public C {};
```

Не всё ок: для подобъекта класса В будет создан подобъект А. Если в В и С будут функции, меняющие что-то в А, то эти изменения будут происходить независимо друг от друга. Поэтому лучше писать виртуал и там и тут – предохранения ради.
Если в А будет f() и в В будет f(); то при вызове f() из Д вызовется доминирующий – из В.

```
class A {f();};
class B: virtual public A {f();};
class C: public B, virtual public A {};
```

Интерпретируется по-разному в зависимости от компилятора (неопределенной поведение). В билдере для С будет вызываться B::f(); в визуале – неоднозначность.

Предположим, есть некоторые визуализированные объекты которые можно нарисовать.

```
class W
{
public:
	void draw();
};

class B: virtual public W
{
public:
	void draw() {W::draw(); /*...*/};
};

class M: virtual public W
{
public:
	void draw() {W::draw(); /*...*/};
};

class MB: virtual public W
{
public:
	void draw() {M::draw(); B::draw(); /*...*/};
```

```
};
```

При отрисовке МБ ДВАЖДЫ отрисуется W – что не есть торт. Для этого в раздел протектеда добавляют _draw(); - то же имя но с подчерком. W::_draw будет рисовать ТОЛЬКО W, B::_draw – ТОЛЬКО B. Соответственно, в MB надо будет по отдельности вызывать W::_draw, B::_draw, M::_draw. Неудобно но безвыходно.

Виртуальные методы

Базовый класс может выполнять объединяющую функцию – обладать набором свойств, присущих объектам производных классам. Функционал общий, однако выполняется по разному (трамвай ездит только по рельсам, а автобус – не только). Методы должны реализовывать производные классы, а базовый задает только их интерфейс. Было решено, что указатель на элемент ЛЮБОГО класса преобразуется к указателю на элемент его базового; в базовом классе метод описывается через virtual – будут создаваться «таблицы соответствия». Жрёт время и ресурсы (нужно спуститься по таблице), но мы получаем свободу подменить одно понятие другим. НЕ ИДЕНТИЧНО доминированию – при доминировании могут вызываться методы базовых классов, а при виртуальности – методы производного.

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

A *pa = NULL;
B *pb = NULL;
if (pa==pb); //неопределено!

A* index(A* p, int i)
{ //категорически запрещено для C++!!1! создание МАССИВОВ ОБЪЕКТОВ - критически атата! максимум - массив указателей на объекты
	return &p[i];
  }
```

14.03.14----------

При выделении базового понятия, не всегда можно определить методы для него. Более того, технология подразумевает, что базовое понятие должно всегда быть абстрактным. Чисто виртуальный метод записывается как обычный, но перед ; пишут =0. Производные от абстрактного класса должны определять чисто абстрактные методы; в противном случае они также будут абстрактными.

Дружественные классы

Взаимодействие объектов – один объект даёт доступ к всем своим членам другому, но не наоборот. Используется, когда один класс должен отвечать за всю работу другого класса.

```
class B; //форвард-обьявление
class A {};
class B
{
	friend class A;
private:
	int abs;
public:
	int getabs();
};
```

Все члены класса А имеют доступ ко всем членам класса Б. Можно ограничить доступ: чтобы только определенный метод был другом класса Б. Этот вариант более правилен – при изменении класса Б в А нужно изменять только один метод.

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

Дружба – не наследуется и не транзитивна. Если от класса Б породить другой класс, то для него класс А уже не будет другом. Если класс А дружит с классом С, то класс Б не дружит с С.

Перегрузка операторов

При задаче «нового» оператора необходимо учесть несколько аспектов. Приоритет (* раньше +); арность (со сколькими операндами работаем); порядок выполнения (слеванаправо или справаналево). В цыплюсе – новые операторы задаются только на множестве тех что уже есть, без добавления новых.

Запрещены к перегрузке: . :: .* (вызов метода через указатель в СИ) ?: (тернарный).

Рассмотрим новые операторы в цыплюсе.

Оператор .*

Void f(); - функция. Указатель на функцию: void (*pf)()

Void A::f(); - метод класса А. Указатель на метод: void (A::*pf)();

Pf(); - вызов функции по указателю.

A obj, *pobj; - объект и указатель на объект. (obj.*pf)() – вызов метода по указателю из объекта. Аналогично используется (pobj->*pf)()

Оператор new [(область размещения)] тип [(список параметров для конструктора) | [количество]]

```
int *pa = new int; //подразумевает наличие конструктора по умолчанию
int *pb = new int(2); //выделяется память и инициализируется; вызов конструктора с одним параметром
int *A = new int[10]; //выделяется память под 10 объектов и для каждого вызывается конструктор по умолчанию
```

Простейший пример утечки памяти:

```
A* obj = new A;
obj->f();
delete obj; //арара! в ф() могло чтонибудь выделиться!
```

Как вариант: написать типа менеджер памяти, когда память аллоцируется в указанном нами куске

```
int buff[10];
```

```
int *p = new (buff) int[10]; //10 объектов размещаются в области памяти buff
```

Оператор delete [[]] указатель
Вызывает деструктор и освобождает память. Если указаны [], то происходит очищение памяти из-под массива. Для объектов массива вызываются деструкторы.

Перегрузка операторов
[тип] operator<знак> ([список параметров]) //longint operator * (longint &a, longint &b)
Операторы можно перегружать как члены или как внешные функции (дружественные или нет).
Оператор нью перегружается как статический метод класса (он вызывается чтобы создать объект – когда самого объекта нет).
Существует три формы его перегрузки.

```
//объявляется методом класса
void* operator new(size_t Size[,  void *buff]); //модификатор статик можно написать, но здесь он используется автоматически
void* operator new[](size_t C); //нью служит ТОЛЬКО для выделения памяти; после возвращения области начинает работать конструктор
//delete возвращает void а не указатель
A* obj = new A;
```

Вернёмся к бинарным операторам + - * /. Принято: если бинарный оператор изменяет существующие объекты, перегружать его надо как член класса, к примеру A+B == A += B.

```
Complex operator +(const Complex& C1, const Complex& C2)
{
	return complex(C1.rl+C2.rl, C1.im+C2.im);
}
```

Оператор при этом должен быть другом класса.
Присваивание перегружается как член класса; выполняется справа налево. Перегрузка присваивания не влияет на инициализацию.

() перегружаются как оператор-член класса, НЕ статический. Получит как минимум один параметр – this, адрес объекта для которого был вызван. () нельзя перегружать для одного класса несколько раз – только единожды.

[] индексация. Позволяет создавать ассоциативные массивы. После перегрузки теряется транзитивность – вызываться будет всегда для левого объекта.

->

```
class A
{
public:
	void f();
};

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

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

Так называемый «умный указатель» - можно создавать объекты, выполняющие чисто транзитные операции. Через них вызываются методы нужного объекта. Позволяет работать со всеми объектами класса A, включая и производные, и даже если A абстрактный (virtual void f()=0;)

++ -- инкремент и декремент. Постфиксный вариант выполняется после того, как объект отработал в выражении; учесть это никак нельзя.

```
class A
{
public:
	A& operator++(); //++a
```

```
        A& operator++(int); //a++0
        operator int(); //оператор неявного приведения типа
}
```

21.03.14----------

Шаблоны

В Си есть возможность задавать макросы с параметрами. В цыплюсе появились шаблоны. Проверка шаблона на корректность происходит только при его использовании. В зависимости от передаваемых параметров шаблон может работать корректно или некорректно. Однако шаблоны достаточно полезны – если есть несколько функций, совершающих одни и те же действия над разными объектами, можно написать шаблон функции.

Шаблон задает функцию для работы с разными типами данных. При создании есть параметр типа.

Пример – определение размера файла.

```
template <typename/*или class*/ T> unsigned length(FILE *fv)
{
	return filelength(fileno(fv)) / sizeof(T); //возвращает размер в байтах, делённый на количество байт в типе
}

FILE *f;
...
unsigned Count = length<double>(f);
```

Если мы явно передаём в список параметров значения с параметрами шаблона, то указывать специализацию (..<double>) не нужно.

Функция может иметь параметры по умолчанию того же типа что и шаблон – тогда специализация нужна. Кроме того, специализацию можно указывать КОНСТАНТНОЙ переменной – написать const double P и писать length<P>.

```
template<typename T> T inc(T &i, T di)
{
	return i += di;
}

int i=0;
inc(i,5);
double d=3.2;
inc(d,1.1);
```

Более того – можно в ШАБЛОНЕ задавать параметры по умолчанию, <typename T=int>

При работе с классами процедура аналогична. В данном случае шаблон НЕ ЕСТЬ класс – по шаблону мы будем создавать какой-либо класс.

```
template <typename T1, typename T2>
class A //<int, double> - конкретный класс, <int, T2> - шаблон с частичной специализацией
{
	/*...*/
};
```

Примеры частичных специализаций:

```
template <typename T>
class A<T,T>
{};

template <typename T1, typename T2>
class A<T1*,T2*> //если при создании класса будем специализировать указатель адресами, то будет использоваться этот шаблон, а не верхний
{};
```

В случае функций – мы получали перегруженные функции. Здесь мы получаем совершенно разные классы.

Параметры-значения можно инициализировать константами (целого типа) или указателями на объекты с внешним связыванием (определенные вне данного файла).

Использование специализации при определении:

```
template <typename T, size_t Size> //сайз - константа //пример того чего не надо делать
class Vector
{
	T v[Size];
	/*...*/
};
//при определении обязательно специализировать значение параметра
```

```
//частичной специализации быть не может - должен создаться объект конкретного класса
Vector<double,10> V1;
Vector<double,20> V2;
```

Два объекта разных классов. Что не есть торт. Если использование шаблона идёт локально то всё нормально, но если его передавать куда-то – то методы других классов должны быть опять-таки шаблонными.
Таким образом, пока мы не создадим чего-нибудь, мы не сможем проверить. Ошибки в шаблонах могут сидеть чрезвычайно долго. Шаблоны хороши только при локальном использовании.

Допускается выделять базовый класс вне шаблона; шаблон же будет производной от него. На уровне же шаблона необходимо будет реализовывать только операции для передачи данного, транзитные операции (данное принимается, куда-то передаётся, а там где начинаем работать – работаем уже конкретное приведение типа со специализацией). Таким образом решается проблема классов-посредников.

```
class A {};
template <typename T> //использование в шаблоне базового класса
class B: public A
{
public:
	int f();
};
template<typename T>
int B<T>::f()
{
}
```

Методы шаблонного класса являются опять-таки шаблонами; определять их надо как шаблоны, но можно указывать специализацию – полную или частичную – под конкретные значения параметры.

Обработка исключительных ситуаций

Главный недостаток структурного программирования – если на каком-то нижнем уровне возникает ошибка, её необходимо наверх наверх наверх где её обработают, спустя N уровней абстракции. Возникает идея передавать ошибку СРАЗУ туда, где её можно обработать. Реализация совершенно разная в зависимости от языка.

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

Страх и ужас – в Ф могла выделиться память, а потом произойти исключение, и управление передаётся выше. Остаётся и память внутри ф(), и память под объект А.

```
try
{
}
catch (<тип> & <идентификатор>)
{
}
```

Блок трай – под контролем. Если возникает исключительная ситуация НА ЛЮБОМ УРОВНЕ, то управление передаётся в обработчик кетч. Если обработать её кетч не может – то исключение передаётся ещё выше. Возникает слоёный пирог из обработчиков исключений. Вызов исключения происходит с помощью

```
if (целое) throw тип (...);
```

Создаётся иерархия «объектов ошибок». Есть базовое понятие ошибки, остальные классы – производные от этого базового класса ошибки.

Пространства имён

В разных кусках кода могут использоваться разные библиотеки – нет гарантии, что в разных библиотеках не будет классов с одинаковыми именами. Поэтому каждой библиотеке нужно задавать своё пространство имён. При работе необходимо указывать, с каким именном неймспейсом мы будем работать.

28.03.14----------

Шаблоны и паттерны проектирования

Шаблон проектирования – нечто готовое, на основе чего можно создавать своё. Предоставляется, например, шаблон класса, и на его основе можно создавать свой класс.

Паттерн – проектное решение, которое можно использовать в своей задаче (как алгоритм).

Были рассмотрены .* и ->* - вызов метода через указатель.

Функтуатор – позволяет вызвать по указателю метод любого класса. «Делегаты» в современных языках.

```
class Callcc;
class Caller
{
	typedef int (Callcc::*FnPtr)(int); //имя типа
private:
	Callcc *pobj; //указатель на объект
	FnPtr ptr; //указатель на метод
public:
	Caller(Callcc *p, FnPtr pf) //конструируем класс, задаем побж и птр
		: pobj(p), ptr(pf) {}
	int operator()(int d) //перегружаем оператор () для класса
	{
		return (pobj->*ptr)(d);
	}
};

class Callcc
{
private:
	int index;
public:
	Callcc(int i=0) : index(i){}
	int inc(int di)
	{
		return index += di;
	}
	int dec(int di)
	{
		return index -= di;
	}
};

Callcc obj;
Caller cl1(&obj, &Callcc::inc); //& мастхэв - потому что это метод класса, а не просто функция
Caller cl2(&obj, &Callcc::dec);

printf("%d  %d\n", cl1(3), cl2(5)); //3, -2; создаётся объект; инкается на 3; декается на -5
```

– – – – – – –

Для создания уникального класса используется «**синглтон**». Его можно рассматривать как шаблон, так и как паттерн. Это неразделяемый ресурс – приводится самый простой пример. В многопоточной реализации данный пример будет непригоден.

```
template <typename T>
class Singletone
{
private: //должен создаваться один объект. Особенность статического данного - будет общим для всех объектов данного класса. Плясать можно от статики - выделять память, а затем контролировать, была выделена память или нет. Если была - возвращать указатель на существующий объект, если нет - создавать заново
	static T* inst;
protected: //в принципе, создавать объекты этого класса пользователю не нужно - нужен только указатель. Конструктор может быть протектынм или приватным
	Singletone(){}
	//так же можно определить конструктор копирования
	Singletone(const Singletone&) {}
	Singletone& operator = (const Singletone&)
	{
```

```
		return *this;
	}
public: //собственно ради чего создавалось - метод, возвращающий указатель на синглтон
	static T& instance()
	{
		if (!inst)
			inst = new T;
		return *inst;
	}
};
template <typename T>
T* Singletone::inst = 0;

Canvas &obj = Singletone::instance<Canvas>();
```

– – – – – – –

Рассмотрим какой-нибудь класс,

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

Как обычно, если в ф() возникает исключение, то делет обж не сможет вызваться; память будет утекать. Используется шаблон-хранитель, контролирующий указатель обж.

```
{
	Holder<A> obj(new A); //хранитель держит указатель на объект - при выходе за границы видимости произойдёт вызов деструктора объекта А
	obj->f();
}
```

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 тотже объект. Создаётся специальная оболочка для передачи – трансфер-капсула. Реализуем сначала её, потом холдер.

```
using namespace std;

template <typename T>
class Holder;

template <typename T>
class Trule //TRansfer capsULE //используется только для передачи; чтобы при передачи в метод функции или возврата функции из метода, не возникло утечки
{
private:
	T* ptr;
public:
	Trule(Holder<T>&h)
	{
		ptr = h.release();
	}
	~Trule() //при передаче параметров также возможна отработка исключений; необходимо позаботиться об уничтожении труля
	{
		delete ptr;
	}
private:
	Trule(Trule<T>&);
	Trule<T>& operator = (Trule<T>&);
	friend class Holder<T>; //для простоты обращения
};

template <typename T>
class Holder
{
private:
	T* ptr;
public:
```

```
	Holder(): ptr(0) {} //когда создаётся холдер указателя, он не должен принимать указатель откуда-то извне
- это может привести к тому, что холдер может держать указатель на уже освобождённую память
	explicit Holder(T* p): ptr(p) {} //нужен явный вызов конструктора, явное создание объекта
	~Holder()
	{
		delete ptr;
	}
	//холдер должен быть прозрачен: черзе него мы должны мочь обратиться ко всему что он держит
	T* operator ->() const
	{return ptr;}

	//также нужна и работа со ссылкой на объект
	T& opetator *() const
	{return *ptr;}

	//обмен
	void exchange(Holder<T>& h)
	{
		swap(ptr, h.ptr); }

	}
	//холдер берёт на себя указатель капсулы
	Holder(Trule<T> const& t)
	{
		ptr = t.ptr;
		//т.птр - константа, поэтому приходится извращаться
		const_cast<Trule<T>&>(t).ptr = 0;
	}

	//присваивание
	Holder<T>& operator = (Trule<T> const& t)
	{
		delete ptr;
		ptr = t.ptr;
		const_cast<Trule<T>&>(t).ptr = 0;
		return *this;
	}
	T* release()
	{
		T* p = ptr;
		ptr = 0;
		return p;
	}
	//также нужны ещё конструктор копирования и оператор присванивания(холдер холдер)
	//при передаче объектов в функцию мы будем использовать трул
};
```

04.04.14----------

Паттерны проектирования

Руководство к действию – можно брать паттерн и решать определенный круг задач.

Паттерны делят на три группы – структурные, поведения, порождающие. В свою очередь, для каждого паттерна существуют различные реализации.

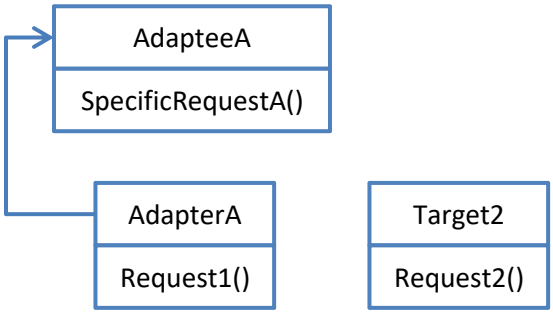
Структурные паттерны

Паттерны, позволяющие каким-либо образом сложную реализацию дать просто. При решении задач возникает много посредников – СП позволяют добавлять объекты-посредники.

Паттерн **адаптер**.

Употребляется достаточно часто. Идея: большая программа; когда мы выделяем объект, может возникнуть ситуация что в разных частях программы мы работаем с этим объектом по-разному. Каркасная модель – формирование, преобразования, отображение и т.д.. Если программу необходимо развивать (модель – физический объект, сделать прочностной расчет) – объекту должны соответствовать колоссальное число методов для работы с ним. При формировании модели мы всё равно тащим с собой методы расчета прочности.

Предлагаемое решение: для объекта реализуется простейший интерфейс для получения доступа к членам и возможности изменять\инициализировать.



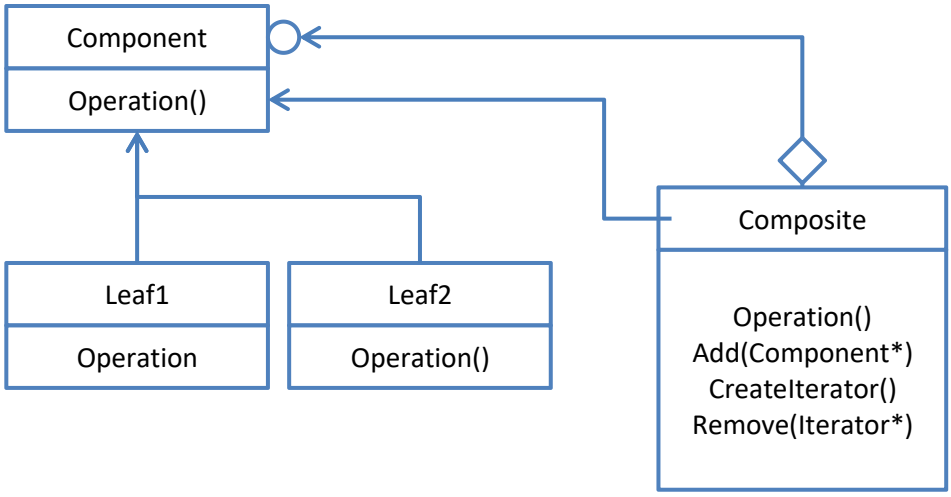
Можно породить объект-посредник, который будет предоставлять интерфейс – мы связываем его с ^ объектом, и работаем теперь с посредником.

Может быть также и другая ветвь, в которой другой посредник работает с другим посредником. В другом месте, для этого объекта можно создать нового посредника с новым интерфейсом. Это можно делать, передавая в метод объект (по ссылке); или при создании объекта мы в конструкторе держим указатель на объект.

Таким образом, в adaptee содержится только set,get; а в adapter – вся работа с объектом в данной области применения.

Паттерн **композит**.

Модель может быть не одиночной; модель может состоять из нескольких.
Задается базовый класс:



Создаем объект список компонентов:

Когда мы передаем команду поворота в этот список, поворачиваются все компоненты, из которых он состоит. Получается контейнер.

Можно рассматривать этот список как сцену. Для того, чтобы получить доступ к конкретному элементу списка, нужно реализовать возможность получения компоненты (через итератор). Функция CreateIterator() возвращает этот итератор, с помощью которого можно обойти список, добраться до нужного объекта, и произвести необходимое действие.

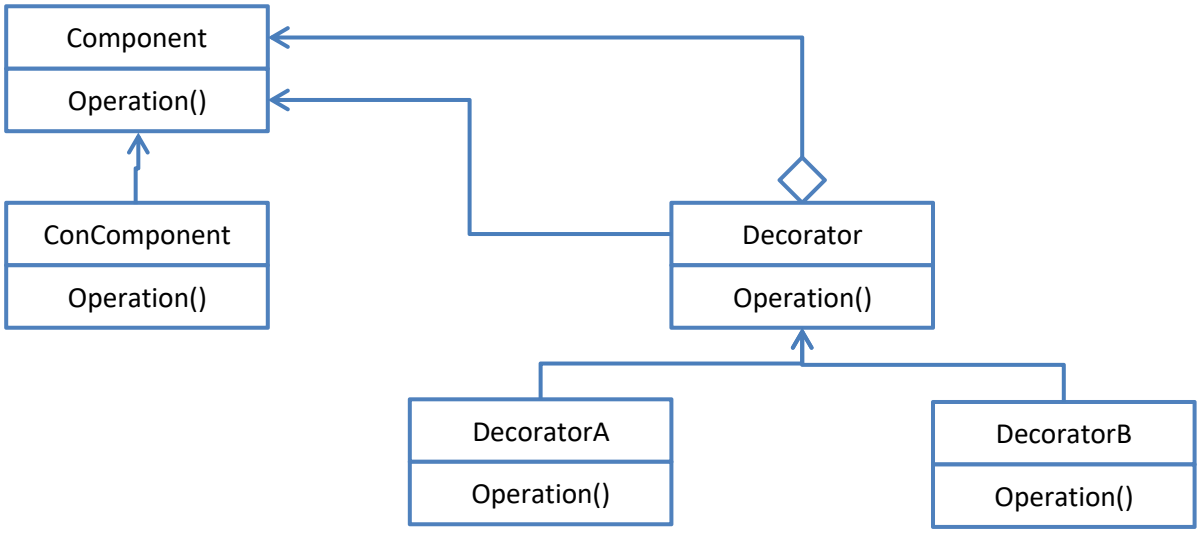
Нужно унифицировать происходящее. Тогда add,create,remove должно быть в компоненте. Возникает вопрос – как быть с простыми элементами лиф1, лиф2. Если эти методы будут абстрактными, то их нужно будет реализовывать и для листьев. Можно реализовать пустые методы, возвращающие false/ничто, символизирующее не выполненность. Появляется возможность проверки – если create возвращает 0, то вызвавший его объект – просто компонент.

```
class Component
{
public:
	virtual bool operation() = 0;
	virtual bool add(Component*) { return false; }
	virtual bool Remove(Iterator*) { return false; }
	virtual Iterator* CreateIterator() {return 0 ; }
//... - конструктор деструктор и так далее
};
```

Для разных компонентов можно использовать совершенно разные адаптеры – действия над одним объектом нельзя выполнить над другим. Добавляется идентификация. Композит наследуется из компонента и переопределяет его виртуальные методы.

Паттерн **декорат.**

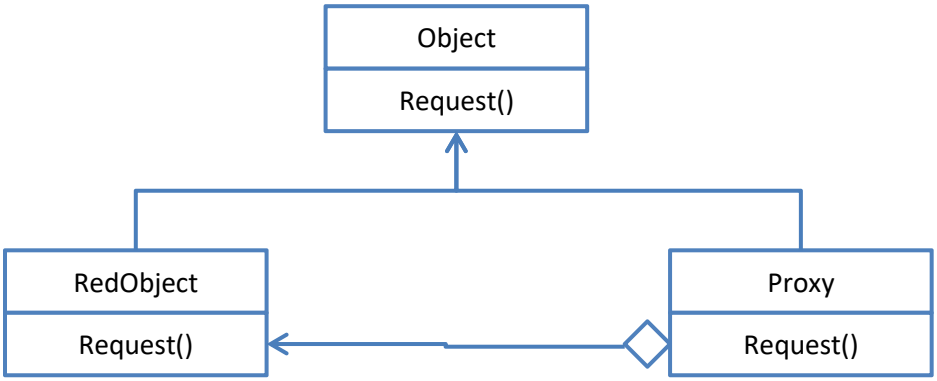
Любое развитие происходит за счет наследования; расширяя понятие мы создаем производные объекты. Есть визуализируемые компоненты – кнопки, текстбоксы и так далее. Их решили по-другому отображать – причём все. Создаются наследники от каждой компоненты. Это приводит к громоздкой иерархии наследования.



Мы выносим класс декоратор, который содержит в себе указатель на компоненту.

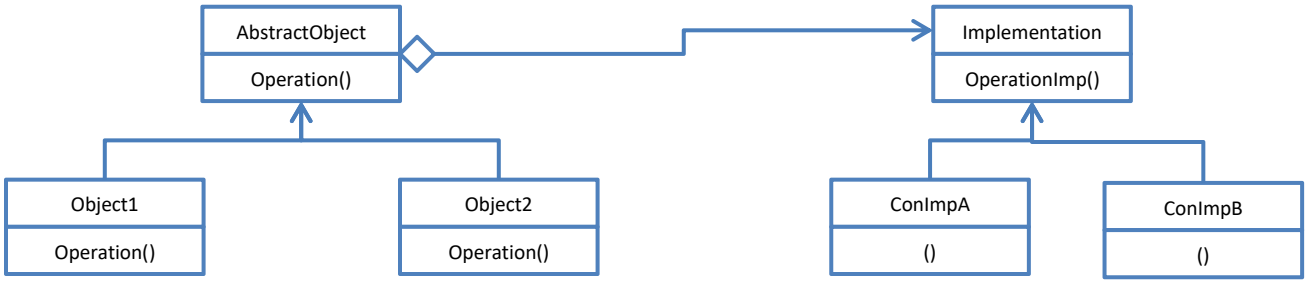
Паттерн **прокси** (заместитель).

Компонент может не существовать, или непонятно нужно с ним работать в данный момент или нет. Прокси выполняет некоторые функции без самого объекта, и только по надобности создаёт нужный объект.



Паттерн **мост.**

Отделяется сущность от реализации.



Отделяется сущность от реализации.

Паттерн **фасад.**

Много классов. Иерархия разваливается на куски со связями между собой. Чтобы не работать с большим числом объектов, создают классы, являющимися фасадами системы. Фасад отвечает за взаимодействие одного «мирка» с другим. По аналогии со структурным программированием, когда функция получала информацию из другого домена. Всю задачу можно рассматривать как единый объект.
Фасадом можно скрыть большое число разных объектов и классов. Однако возникают задачи, когда нужно работать со множеством мелких объектов одного класса.

Паттерн **приспособленец.**

Приспособленец как и фасад даёт интерфейс для мелких объектов. Однако приспособленец является не только структурным, но и порождающим паттерном. Если данного объекта нет, то приспособленец создаёт его. Прокси не создаёт объект – он лишь делает запросы, получая от кого-то указатели на объект, в то время как приспособленец непосредственно создаёт объекты.

Есть иерархия примитивных объектов; есть класс, содержащий список этих объектов или кэш. Не нужно держать большое количество объектов – приспособленец может удалять объекты, после того как отработаны.

```
//работа с символами
#include <map>
//базовый класс:
class Character
{
protected:
	char Symbol;
	int Size;
public:
	virtual void view() = 0; //если 0 не написать, то метод нужно будет реализовать хотя бы где-нибудь
};

class CharacterFactory
{
private:
	std::map<char,Character*> Characters; //связка ключ_символ - значение_ОбъектСимвола
	int size;
public:
	Character& GetCharacter(char key)
	{
		/*std::map<char,Character*>::iterator*/Characters::iterator it = Characters.find(key);
		if (Characters.end() == it)
		{
			Character* ch = new ConCharacter(key,size);
			Characters[key] = ch;
			return *ch;
		}
		else
		{
			return *it->second; //приспособленец или возвращает новый объект, или указатель на
существующий
		}
	}
};
```

11.04.14----------

- - -

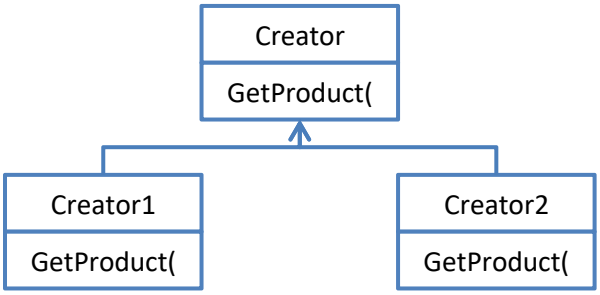
ЛР №3 – реализовать первую с использованием ООП. ООП используется для того, чтобы в дальнейшем программу можно было расширять и модифицировать – проектировать надо в этом направлении. Использовать паттерны ^. При проектировать предполагать, что в дальнейшем помимо вьювера модели потребуется редактор, просмотр нескольких моделей, что модель может состоять из подмоделей и так далее. Возможно использование шаблонных классов. Обработка исключений, с использованием стандартных классов и производных от них. Для реализации списков использовать не стандарты, а руки, с применением паттерна итерации.
КРИТИЧЕСКИ СЛЕДИТЬ ЗА ПАМЯТЬЮ, шаблоном-хранителем. При формализации получается порядка тридцати классов. Никаких праздношатающихся функций, всё привязано к объектам.

- - -

Порождающие паттерны

Часто нужно порождать классы. Однако было бы удобно определять в самом начале, объект какого класса нам нужен, и создавать соответствующе. Соответствующие классы должны быть родственниками, поддерживающими единый интерфейс.

Фабричный метод.



При создании креатор1 и креатор2, передаётся указатель на базовый класс. Можно вызывать виртуальный метод гетпродукт который вернет именно то, что нужно в данный момент.

```
class Creator
{
public:
	Product* GetProduct();
protected:
	virtual Product* CreateProduct() = 0;
private:
	//два варианта реализации - нужно в разных местах использовать один объект. креатор должен содержать указатель на объект, который надо порождать. если же нужно в создавать объект в разных местах, то указатель можно не держать - метод будет выглядеть просто как создание объекта
	//продукт - базовый тип для продуктов разных классов. недостаток - для использования фабричного метода нужно использовать одну и ту же базу.
	//используем один объект в разных местах; порождать нужно только один раз - держим указатель
	Product* prod;
};

//реализуем гетпродукт
Product* Creator::GetProduct()
{
	if (!prod)
		prod = CreateProduct(); //передаётся зыс - будет создаваться объект того типа, из которого вызвали
	return prod;
}

//создаем шаблон
template<typename Tprod> //Тпрод - производный от продукта!! низя будет передавать то что с продуктом не связано, иначе атата!
class ConCreator: public Creator
{
protected:
```

```
	virtual Product* CreateProduct() //можно было бы и тпрод*, но указатель на производный класс автоматически преобразуется к базовому
	{
		return new Tprod;
	}
};
```

Предположим что используется среда или библиотека, например графическая. В зависимости от того, какая библиотека, будут использоваться разные методы для отрисовки с разными названиями. Указание цветов, инициализация рисовалок (пен, браш) везде реализуется по своему. Таким образом, нужно создавать разные объекты. В ПРИНЦИПЕ, можно создавать кучу фабричных методов, но будет неудобно. Кроме того это не защищает от ошибки.

Идея – задать класс, который будет создавать набор объектов для выполнения общих действий. При работе с графикой – свои собственные объекты для рисования, и в зависимости от использования библиотеки будут создаваться конкретные объекты. По существу, будет не один метод ГетПродукт, а несколько, например ГетПен, ГетБраш, ГетГрафикс и так далее.

Называется это всё абстрактной фабрикой. Шаблоном в этом случае не обойтись – нужны конкретные производные классы, создаются конкретные объекты.

Существует много задач, в которых объект создаётся поэтапно. ООП говорит о том, что при создании класса (формализации понятия) надо придерживаться жёсткого правила: все атрибуты класса должны для любого объекта иметь значения, не может быть неинициализированных полей; при этом значение должно быть одно.

Возникают ситуации, когда полностью все для объекта инициализировать нельзя. Пример – камера на дороге, с которой получается фотография; далее она обрабатывается, выделяется машина, номерной знак, знак распознается, пытаемся распознать марку машины, получается инфомрация из баз данных и так далее. Идёт поток фотографий, уходит время на операции и дополнительную обработку. Данное «информация о машине» появляется постепенно.

Создаются специальные классы, занимающиеся построением целостного объекта.

Паттерн **строитель**.

Строитель может задавать проекты создания объектов – нельзя вначале заложить фундамент и сразу же начать строить крышу. Что-то может зависеть от порядка, что-то нет; какие-то элементы могут иметь связи между собой. Задача строителя – построение объекта, и только после построения строитель может уже вернуть объект (Если объект не создался – ничего не вернётся).

Строитель – набор методов поэтапного создания объекта и его частей + метод GetProduct возвращающий окончательно созданный объект.

Иногда нужно создавать похожие объекты. Создали объект, попользовались, затем нужен подобный ему. Мы не создаём целиком новый продукт, а создаем новый на основе существующего. Единственное требование – у него должен быть метод клонирования (создания копии). Диаграмма такая же, паттерн – прототип.

Строитель часто включает в себя интерпретатор – мы на каком-то языке записываем, как создавать объект, а затем интерпретатор по этому «коду» создаётся нужный нам объект.

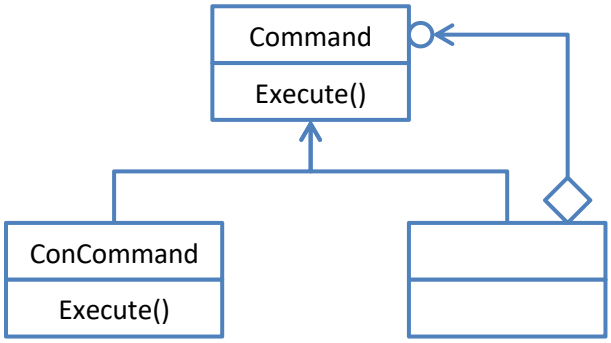
Паттерны поведения

Чрезвычайно объёмная группа. Наиболее часто используется аж 11 штук.

Паттерн **команда**.

При событии (нажатии на кнопку, выделении компоненты) можно реализовывать событийную схему или действовать по принципу вызова и передачи команды. Команда может использоваться даже в асинхронном взаимодействии, но в целом появилась именно в синхронном. В первой лабе – имя команды (номер) данные (юнион структур). Можно подойти и более широко.

При передаче команды должно выполниться какое-то действие – можно передавать и действие, которое надо выполнить. Команда будет вызываться для нужного нам объекта.



Одна команда может разбиваться на целую цепочку действий (повернуть объект – повернуть все его подобъекты). Можно использовать паттерн композицирования. //добавляется вотэта ячейка-композит со стрелочкой с ромбиком и точкой

```
class Command //базовый класс команды
{
public:
	virtual bool execute() = 0;
};

template<typename Reciever>
class SimpleCommand: public Command
{
private:
	typedef bool (Reciever::*Action)();
	Reciever* rec; //указатель на объект
	Action act; //указатель на метод

public:
	SimpleCommand(Reciever* r, Action a); //конструктор
	bool execute() //экзекутор
	{
		return (rec->*act)();
	}
};
```

При создании команды происходит связывание с объектом, однако оно может происходить и при приеме команды. Экзекут может принимать объект, для которого выполнить данную команду.

Здесь реализовывается простая команда, дальше композируем – выполняется тоже самое но, например, обходя список команд (объект для которого выполняется список команд должен быть один).

Если команда несёт какие-то данные, то в конкоманде (НЕ В ШАБЛОНЕ СИМПЛКОМАНДЕ!) будут дополнительные инты флоаты и так далее.

Паттерн **стратегия**.

В ^ экзекуторе мы по укзаателю вызываем метод объекта. Реализовывать можно и по-другому – рассматривая алгоритм обработки чего-либо. Для реализации поиска по контейнеру (который может быть поразному организован) - могут быть разные алгоритмы поиска. Идея: обернут алгоритм в объект. Связывать будем наш объект с алгоритмом обработки – передавать алгоритм, по которому обрабатывать.

Пример – разные алгоритмы для обработки изображения. Выделение характеристических точек, сёрф-алгоритм. Один устойчив к освещению, другой неустойчив. В зависимости от условий используются разные алгоритмы.

Патерн **цепочка обязаностей**.

Похож на команду, однако для одного объекта можно задать цепочку или последовательность обработки в виде списка. Организовать конвейер обработки – один этап, потом другой, этцетера. Аналогия с композицией команд – поскольку используется компонент, то допускается древовидная и даже рекуррентная структура.

Паттерн **интерпретатор**.

Используется для того, чтобы можно было код на псевдоязыке обрабатывать во что-то. Может использоваться для строителя, для формирования порядка команд, цепочки обязанностей, формирования сложного графа для обхода.

Паттерн **посредник**.

Определяет интерфейс обмена информацией с объектами. Похож на стратегию – однако вызывает не алгоритм, а посредник один метод преобразует в другой. Обращаемся к посреднику, а тот вызывает метод связанного объекта. Не путать с адаптером – адаптер полностью заменяет интерфейс, посредник осуществляет обёртку интерфейса.
Получаем посредника и работаем с ним, неважно какой именно связанный объект будет выполнять требуемое нам действие.

Посредник может быть сложным – опять можно реализовывать подобную схему компоновки. Однако посредник может вызывать какой-то определённый объект на котором «стоит фокус».

Паттерн **подписчик-издатель**.

Может реализовываться как синхронно так и асинхронно.

Есть объект, с которым происходят события, и можно «подписаться» на конкретные события. Для реального объекта выделяются какие-либо его состояния и изучаем, как объект эти состояния меняет. Состояния надо описывать – поведенческие паттерны именно этим и занимаются.

18.04.14----------

Объектно-ориентированный анализ и проектирование

Циклы разработки ПО с использованием ОО проектирования

Все почти так же как и в структурном, но идет перераспределение и изменение названий этапов.
Основной этап: ОО анализ, больше всего времени. Задача ООА – построение модели системы. На основе построенной модели выполняется проектирование.
Проектирование – перевод модели в проектные документы; с помощью CASE-средств.
Эволюция. Написание кода, тестирование, итерирование - ОО подход за счет рекурсивного дизайна, система разворачивается и усложняется.
Модификация – эволюционирование на готовом продукте, сданном заказчику.

Преимущества рекурсивного дизайна при эволюционировании:
система постоянно развивается, но всегда готова – обеспечивается обратная связь с заказчиком
предоставляются разные версии системы – можно откатываться на шаг назад; можно пускать разные версионные ветки
заказчик постоянно видит результат
серьёзных ошибок не возникает за счет постоянного взаимодействия с заказчиком
хорошо распределяется время при проектировании
размеры системы (кода) в принципе не ограничиваются; при структурном же подходе чрезвычайно большая программа при большом количестве разработчиков вызывает проблемы с взаимодействием оных

Изменения в порядке ухудшения сценария
проектировать надо так, чтобы добавление классов было безболезненно
изменение реализации класса
изменение представления класса
реорганизация структуры классов
изменение интерфейса класса – самое страшное, тянет за собой кучу изменений в основном коде

Задача анализа – довести модель до такого состояния, чтобы дальше не понадобилось изменять интерфейс.

ОО анализ

В структурном программировании используется принцип черного ящика – мы не думаем о конкретных действиях и данных, а о том что нужно сделать. В ООП – белый ящик, первее всего – данные. Мы выделяем характеристики объектов, и потом уже приходим к тому, как их обрабатывать.

Система разбивается на домены – интерфейс, реализация, сама задача, сервисные функции (например коммуникация) и так далее. Рисуется схема доменов для доменного уровня всей задачи. Для выполнения же этапов анализа рисуется проектная матрица – мы контролируем, какие этапы анализа нами пройдены.
С доменом в котором более 150 классов работать тяжело, а на деле – уже даже с более 50. В домене четко выделяются группы классов, между которыми много связей, в то время как между группами связей мало – домен делится на подсистемы, и проектируем вначале подсистемы, а потом реализуем связи.

Для домена строится модель связи подсистем. Выделяются синхронная и асинхронная схемы взаимодействия, соответственно модель доступа к подсистемам (синхронная) и модель взаимодействия подсистем (асинхронная).

Информационное моделирование – выделяет сущности, с которыми надо работать, их характеристики, описать и выявить связь между сущностями, попытаться её реализовать.

Для каждой подсистемы:

- Строится описание классов, атрибутов, связей между классами.
- Также строится модель взаимодействия объектов (событийная).

- Модель доступа к объектам.
- Таблица процессов состояний.

Для каждого класса:

- Выделяем модель состояний (модель переходов состояний).

Для каждого состояния каждой модели состояний:

- Строится диаграмма потоков данных действий (ДПДД).

Для каждого действия (процесса):

- Делается описание процесса.

Начнем с главного.

Выделили домен. На подсистемы делить бессмысленно – нужно вначале наполнить его сущностями-классами. Идея информационного моделирования: мы начинаем работать с физическими объектами, которые есть в реальном физическом мире; всё остальное появится в результате их анализа. Выделяются физические объекты.

Мы пытаемся посмотреть, какими объектами населён домен, и разбить их на группы. Пытаемся выделить характеристики сущностей. Если все объекты имеют одинаковые характеристики, то их можно отнести к одному классу – формируется класс.

При выделении характеристик (атрибутов) классов. Атрибут – каждая отдельная характеристика, являющаяся общей для всех возможных экземпляров классов. Каждый атрибут задаётся множеством значений, которое важно определить при анализе для определения в дальнейшем типе атрибута.

Каждый объект нужно идентифицировать. Идентификатор – то, что чётко определяет конкретный объект. Может состоять из одного или нескольких атрибутов. Изменение атрибута не приводит к изменению самого объекта – объект просто меняет «имя», но остаётся тем же самым.

Выделяют три типа атрибутов:

- Описательные – характеристика, присущая каждому экземпляру класса (вес студента).
- Указывающие – идентификатор или его часть (имя студента).
- Вспомогательные – используются для формализации связей; атрибуты состояния объектов.

При проведении информационного моделирования нужно четко проанализировать характеристику. Мы смотрим, какое множество значений имеет атрибут и стараемся описать этот атрибут.
Если он описательный, то мы определяем, какую характеристику физического объекта этот атрибут абстрагирует.
Если он указывающий, то нужно установить форму указания (если уместно), кто назначает указание (если уместно), степень с которой атрибут используется как часть идентификатора (ФИО – фамилия основной атрибут, потом имя, потом отчество; не хватило – дополняем чем-то ещё).
Если он вспомогательный, возник при формализации связи – надо описать, какое отношение сберегает данный вспомогательный атрибут (муж-жена). Если объект имеет динамическое поведение, то мы записываем состояние, в котором он находится.

Выделяют четыре правила атрибутов:

• Один экземпляр имеет единственное значения для любого атрибута в любое данное время – не может быть чтобы значений было два, или не было. Если появляется объект с неопределенным значением атрибута, то относить его к этому классу мы не можем – это уже объект другого класса.

• Атрибут не должен содержать никакой внутренней структуры.

• Когда объект имеет составной идентификатор, каждый атрибут-часть идентификатора представляет характеристику всего объекта, а не его части (и тем более не характеристику чего-либо другого).

• Атрибут, не являющийся частью идентификатора, представляет характеристику экземпляра

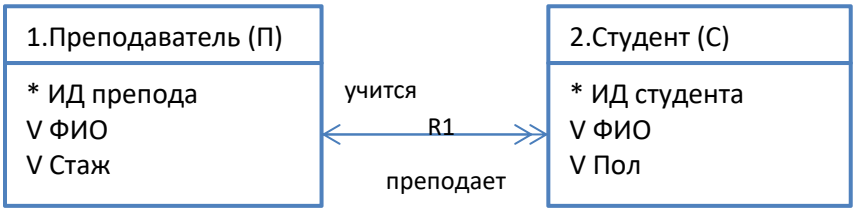
указанного идентификатором, а не характеристику некоторого другого атрибута-неидентификатора. Не должно быть атрибутов, описывающих только часть объекта (атрибут описывающий другой атрибут).

На информационной модели каждый класс описывается прямоугольником с заголовком: номером в домене, именем класса, ключевым литералом. Далее перечисляются характеристики.

<№><имя>(<к.л>)
* часть_идентификатора
V Атрибут
V Атрибут

Объекты вступают во взаимодействия друг с другом, что нужно формализовать.

Связь – абстракция набора отношений, которая систематически возникает между различными предметами в реальном мире – внутренняя (R) и внешняя (E).

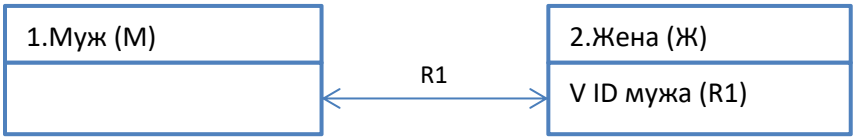


Дальше необходимо выявить множественность связей. С каждой стороны может участвовать один объект, или один с многими, или многие со многими: один к одному, один к многим, многие к многим – виды связей. Учебный курс – студент, МкМ; Преподаватель – Студент, ОкМ.

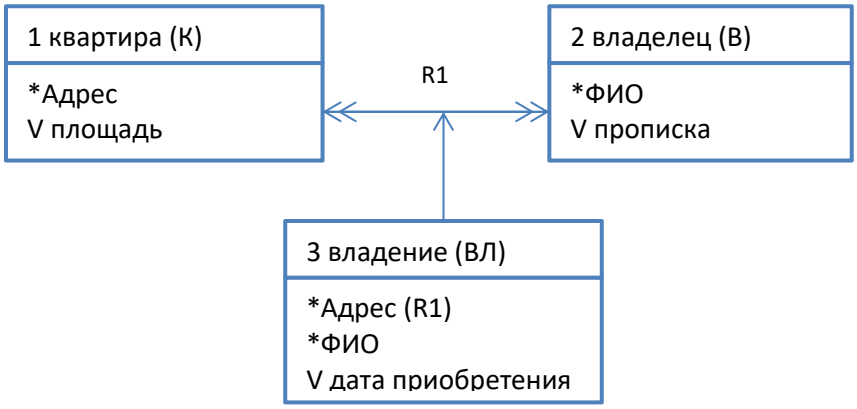
Иногда с какой-то стороны объект МОЖЕТ не участвовать в связи, тогда связь – условная (преподаватель не руководит студентами – условная связь со стороны преподавателя, т.к. у студента должен быть руководитель, а преподаватель не обязан руководить). Если с обоих сторон объекты МОГУТ не участвовать в связи, то связь – биусловная (УчебныйКурс – Студент; курс не читается в этом семестре, или студент не выбрал этот курс).
Итого – 3 безусловных (ОКО, ОКМ, МКМ), 4 условных ОуКМ, ОКМу, МуКМ, МкМу), 3 биусловных(ОукОу, ОуКМу, МуКМу).

Множественность связи указывается стрелочками. Одна стрелочка – единичная, две стрелочки – множественная.

При формализации связи, вспомогательный атрибут добавляется к одному из объектов в связи.



При связи ОКМ добавляется атрибут *со стороны многих* – поскольку атрибут не может содержать внутреннюю структуру. Однако при связи МКМ добавляется так называемый ассоциативный класс, который формализует связь. Собственник может владеть многими квартирами, у квартиры может быть много собственников.



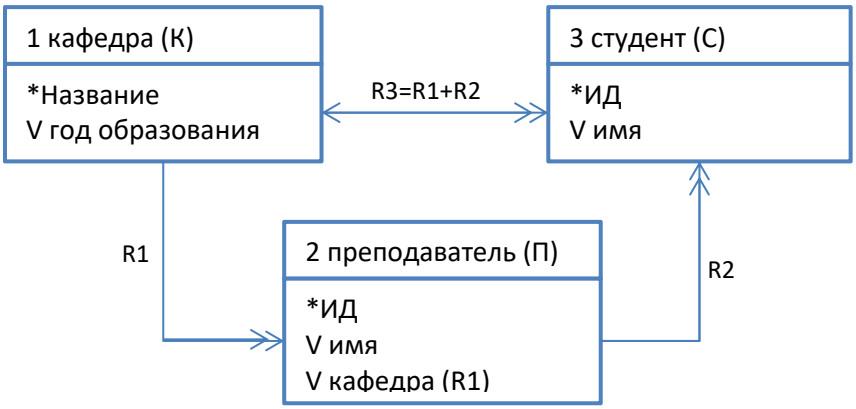
Владение формализует связь. Его адрес и фио четко идентифицируют два объекта, находящихся в связи R1.

Если связь ОКО и один объект удаляется, то второй объект, участвующий в связи, тоже должен быть удалён, либо изменен – если жены не будет, то муж станец вдовцом или холостяком. Это будет другой объект другого класса. Если есть условность связи (связь имеет динамичекое поведение), то такая связь должна формализовываться ассоциативным объектом. При этом условность необязательна, достаточно динамики – была одна связь, потом поменялась.

25.04.14----------

Любую связь, имеющую динамическое поведение, нужно формализовывать ассоциативным объектом.

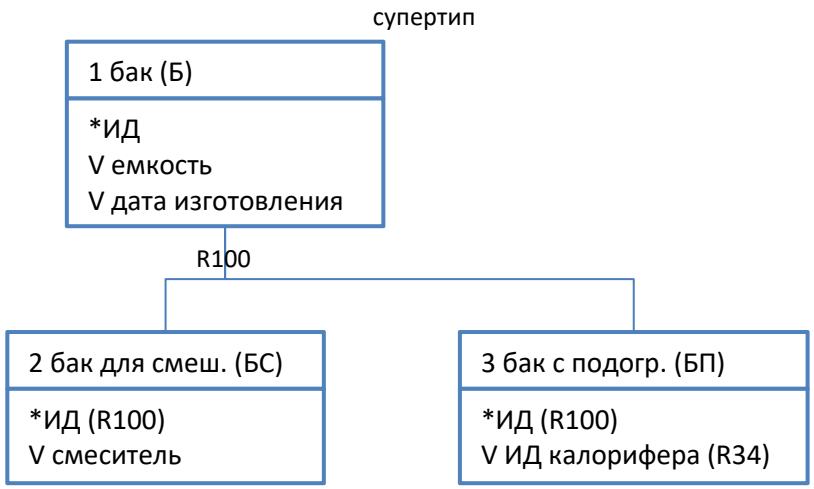
Могут возникать ситуации, когда некоторые связи могут являться результатом других связей. Студент – куратор – кафедра. Куратор выбирается с той же кафедры, что и студент.



Жена – муж, ассоциативный объект – свидетельство о браке, может иметь связь с ЗАГСом в котором выдано.

Подтипы и супертипы

Возникают ситуации, когда какие-то объекты имеют данные атрибуты, а какие-то не имеют. Объекты похожи, но отличаются небольшим набором атрибутов. Реализуется выделение супертипа, в который выносятся атрибуты, являющиеся общими для всех объектов. В подтипы добавляем атрибуты, свойственные конкретной группе объектов.



^ смеситель – состояние, калорифер – произвольная связь.

В ОО анализе есть ограничение – мы не можем создавать объекты супертипа, только объекты подтипов.

Модели поведения в реальном мире

Можно чётко выделить сущности, которые появляются, проходят через отчетливые стадии и прекращают существование. Соответственно, можно представить мир через изменение состояния объектов.

1 Многие предметы на протяжение своего существования проходят через отчетливые стадии.

2 Порядок перехода из одной стадии в другую (эволюционирование) формирует характерную черту поведения объекта

3 Реальный объект находится в единственной стади модели поведния в любой момент времени

4 Предметы эволюционируют от одной стадии к другой скачкообразно

5 В схеме поведения разрешены не все эволюции между стадиями (самолёт стоящий должен вначале взлететь а потом убрать шасси)

6 В реальном мире существуют инциденты, которые заставляют объекты переходить из одной стадии в другую – эволюционировать

Для описания динамического поведения используется модель состояний Мура. Она состоит из множества состояний, множества событий (инциндентов), правил перехода и из действий состояний. Для правил перехода используется либо ДиаграммаПереходовСостояний, либо ТаблицаПереходовСостояний.

В реальном мире, состояния разных сущностей (относящихся к разным классам) скоординированы друг с другом. Изменение одного состояния одного объекта приводит к изменению состояний других объектов. Как правило, мы рассматриваем объекты во взаимосвязи друг с другом.

Определение состояния: состояние – положение объекта, в котором применяется определенный набор правил, линий поведения, физических законов и т.д..

На ДПС состояния задаются прямоугольниками. Каждому состоянию присваивается имя (уникальное в рамках модели – но может быть таким же, как в модели состояний другого класса), номер (опять же в рамках модели состояний).

Можно выделить три вида состояний:

• создание - объект появляется в первый раз (может быть несколько). Возникает событие, но не происходит перехода.

• заключительное – 1) объект переходит через стадии и приходит в состояние, из которого других переходов нет; 2) объект уничтожается при переходе в это состояние (рисуется пунктирной линией).

• текущее

Событие – абстракция инцидента или сигнала в реальном мире, которая сообщает о том, что что-либо переходит в новое состояние. Выделяют четыре аспекта событий:

• значение – короткая фраза, которая сообщает что происходит с объектом.

• предназначение – модель состояний, которую принимает событие. Приёмник должен быть один.

• уникальная метка – как правило буква (ключевой литерал сущности/МС) и номер. Если событие внешнее, то помечается буквой Е

• данные события – данные, сопровождающие событие. Каждому состоянию ставится в соответствие действие, которое должно произойти по событию. Событие должно перевести данные для выполнения этого действия.

- идентифицирующие данные
- дополнительные данные

Существует ряд правил, связывающих события и данные.

1 Правило тех же данных – все события, которые вызывают переход в определенное состояние, должны нести одни и те же данные.

2 Правило состояний несоздания – если событие переводит объект из состояния в состояние, то оно должно переносить идентификатор объекта.

3 Правило состояния создания – событие, переводящее объект в состояние создания, не несёт его идентификатора.

Модель скачкообразная. Плавный переход из состояния в состояние можно формализовать несколькими промежуточными состояниями в модели.

Действие – операция, которая должна быть выполнена объектом когда он достигает некоего состояния. С каждым состоянием связано только одно действие. При формировании ДПС мы

стараемся описать действия. Для этого как правило используется псевдокод. Если действие небольшое, то его можно располагать непосредственно под состоянием на диаграмме, если большое – то выносим.

Что может выполнять действие: любые вычисления; порождать событие для любого объекта любого класса; порождать событие для чего-либо вне области анализа; выполнять все работы с таймером – создавать\удалять\устанавливать\считывать.

Действие может иметь доступ к любым атрибутам любого объекта, как своего так и всех других – читать, записывать. Однако на выполнение действия накладываются ограничения:

• действие не должно оставить данные противоречивыми.

• действие не должно оставлять противоречивыми связи (если был удалён объект, то нужно позаботиться и об удалении\изменении объектов с ним связанных).

• только одно действие может выполняться в данный момент для конкретного объекта; но для разных – действия могут выполняться одновременно.

События могут откладываться до тех пор, пока не будет выполнено действие. Каждое событие прекращается, когда оно «перестаёт быть» событием.

К выделению МоделиСостояний мы подходим как к формализации асинхронного взаимодействия – происходит инцидент, в результате него объекты меняют состояния. Смена состояния может происходить не мгновенно, а через какое-то время после.

Для моделирования чрезвычайно важным является объект таймер. Мы устанавливаем какое-то время и запускаем обратный отсчёт. Задача таймера – подать сигнал, когда время станет равным нулю. Требуется событие, которое он будет порождать, и объект, которому подаётся сигнал. Реализуем циклический жизненный цикл – для упрощения и показания реализации.

Действие - <Событие>:<действие>(<данные>;<данные)

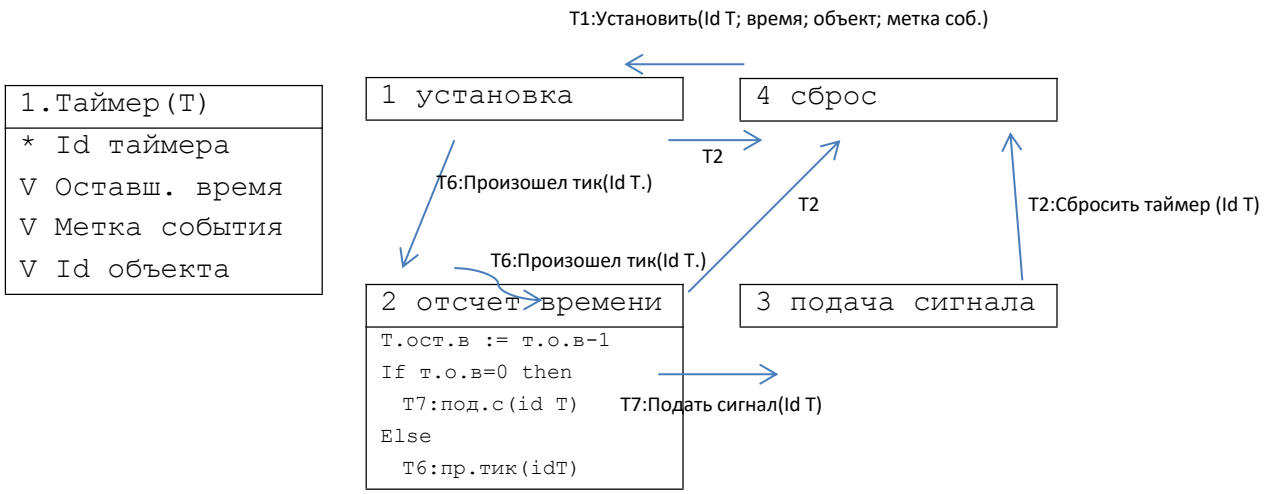


Таблица переходов состояний – каждая строка есть состояние, каждый столбец есть событие. Нужно заполнить все ячейки – какое новое состояние достигается объектом, при возникновении события.

	T1	T2	T6	T7
1	-	4	2	x
2	-	4	2	3
3	-	4		
4	1			

Пустые ячейки – событие игнорируется. Ячейки заполняются прочерками. А если событие не может произойти в этом состоянии, то крестиками.

Паттерн подписчик-издатель

{

Один объект меняет свои состояния. Он генерирует событие – так как весь мир связан, то другие объекты других классов могут среагировать на это событие. Один объект генерит события, а другие подписываются на возникновения этих событий – будет вызываться мето данного объекта. Жмётся кнопка – лифт начинает двигаться по этажам, на каждом этаже проверяя не дошёл ли. Когда дойдет – произойдёт событие остановки лифта, сгенерит дверям действие открыться. Если мы подписались на событие «Двери открыты» то можно изменить своё состояние на «войти в лифт».

Существует понятие делегата – отсроченный вызов, или вызов по указателю, callback. Однако калбак – синхронное взаимодействие, мы же реализуем асинхронне. В языки добавляют события events; делегат предоставляет интерфейс метода, который мы будем вызывать, а событие – переносит данные для вызова метода.

```
delegate void EventHandler(Object^ source, double a);
public ref class Manager //издатель
{
public:
	event EventHandler^ OnHandler;
	void method()
	{
		OnHandler(this, 10.);
	}
}
public ref class Watcher //подписчик
{
public:
	Watcher(Manager^ m)
	{
		m->OnHandler += gcnew EventHandler(this, &Watcher::f); //& - потомучто метод класса
	}
	void f(Object^ source, double a) //подписали метод f на возникновение события. Когда начнёт выполняться
method, будут выполняться методы классов, подписанных на возникновение события
	{
	//...
	}
}
```

}

16.05.14----------

Подписчик издатель – выделили объект, выделили его атрибуты, строим модель состояний (модель мура – состояние, событие, действия состояний, правила перехода). Каждому состоянию соответствует действие – обработчик при переходе состояния. Управление при обработке передаётся обработчику. Действие – метод объекта.

Объект отвечает только за себя, принцип инкапсуляции; строитель может давать связи объекта с объектом, но за внутреннее состояние отвечает только сам объект.

Правило перехода: когда объект переходит из состояния в состояние, это фиксируется; строится таблица переходов состояний. Все ячейки заполняются: новое состояние, игнор, недопустимость. В объекте вводится атрибут состояния – в простом варианте енум; в обработчике необходимо проверить, а может ли объект из текущего состояния перейти в заданное. Если не может – то по какой причине; либо ничего не происходит, либо кидается исключение. Действие обязательно меняет атрибут состояния объекта.

Выделяются суперклассы по атрибутам объектов. Если разные объекты имеют общие атрибуты то выделяется суперкласс и реализуются подклассы. Выделяются жизненные циклы для объектов – здесь также необходим анализ. Разные объекты могут иметь схожие жизненные циклы, либо различные состояния у разных объектов (с совершенно разными атриубтами) могут иметь общие части жизненных циклов. Тогда нам необходимо вернуться к информационному моделированию – нужно продолжить формализацию, выделить базовый класс и производные от него, и уже базовый класс не будет задавать общие атрибуты, он будет лишь отвечать за общее поведение производных классов.

В каких случаях надо выделять жизненные циклы (совокупности состояний)?

- создание и уничтожение объекта во время выполнения
- миграция между подклассами
- накопление атрибутов. С изменением значения атрибута меняется поведение объекта (человек в зависимости от возраста изменяет поведение)
- операционный цикл оборудования (лифт, станок)
- объект производится поэтапно (сборка машины на конвеере)
- если возникают объекты с жизненным циклом, и мы хотим реализовать асинхронную схему взаимодействия, то задача или запрос – тоже имеют жизненный цикл.
- динамические связи. Формализуется ассоциативным объектом, и для него выделяется свой жизненный цикл. Объект, являющийся таким «результатом» связи, стоит на более высоком уровне, лучше осведомлен о состоянии системы.
- если для какого-то объекта мы выделили цикл, но он имеет безусловную связь с другим объектом (пассивным), то жизненный цикл нужно выделить и для этого пассивного объекта.

Что касается суперклассов и подклассов, то используются разные подходы формирования диаграммы переходов состояний. Если все подклассы имеют единый жизненный цикл, то мы задаем его на уровне суперкласса. Иначе на уровне суперкласса задается только часть жизненного цикла – выполняется операция сращивания; на диаграмме указывается общая часть присущая всем подклассам, и та часть, которая отличается в зависимости от подкласса.

Анализ отказов

Любое аномальное поведение необходимо реализовывать также как и нормальное. В принципе, можно формировать состояния\ситуации отказов при проектировании – состояние, в которое переходит объект, вместо того чтобы кидать исключение.

Выделение отказов посредством ООМ.

1 какой инцидент в реальном мире мог бы вынудить объект к переходу в некорректное состояние.

2 гарантируется ли наступление каждого события.

3 если событие происходит, то нормально ли это.

4 гарантируется ли что событие происходит своевременно (когда объект находится в нужном

состоянии).

Не всегда переход из состояния в состояние происходит мгновенно – для учёта этого перехода приходится моделировать состояния, которых в реальном мире нет.

Таблица переходов: имеет описание действий и список событий, происходящих в системе.

Один объект может порождать события для других – лифт и двери, лифт остановился -> двери начинают открываться. Получаем связанные жизненные циклы.

Динамика систем. Модель взаимодействия объектов (МВО)

МВО обеспечивает краткое графическое изложение событий взаимодействия между моделями состояний и внешними сущностями, происходящими в подсистеме. МВО рисуется в овале, внешняя сущность – прямоугольник (именуемый терминатором). События, которые порождаются одной моделью для другой, рисуются стрелкой – так же могут приходить события от внешних сущностей. События могут быть направлены к терминаторам.

МВО формируется иерархически – объекты, наиболее осведомленные о всей системе (активные) располагаются вверху диаграммы. Если событие приходит извне к МВО, находящимся вверху, терминаторы рисуются вверху – терминаторы верхнего уровня. Если события уходят или приходят к МВО нижнего уровня, терминаторы рисуются снизу. Может быть схема верхнего и нижнего управления – система ограничена терминаторами сверху или снизу.

Надо стремиться к тому, чтобы на верхнем уровне взаимодействие терминатора сводилось к одной модели. С нижним этого ограничения нет – может быть сколько угодно МВО, взаимодействующих с терминаторами нижнего уровня. Как правило на нижнем уровне терминатор (внешняя сущность) может быть физическим объектом, с которым мы работаем.

С точки зрения организации МВО, есть более и менее осведомленные объекты; терминаторы в этой иерархии взаимодействуют чётко: верхние терминаторы взаимодействуют с осведомлёнными объектами, нижние с неосведомлёнными. Противных схем лучше избегать – нужно продолжать формализацию, искусственно выделяя объекты меньшей осведомленности.

Все события на МВО делятся на две группы: внешние (приходят или уходят к терминаторам) и внутренние (соединяют одну модель состояний с другой).

Внешние также делятся на две группы: незапрашиваемые события и запрашиваемые. Незапрашиваемые – не являются результатом предыдущих действий подсистемы (управляющие); запрашиваемые – являются результатом действий.

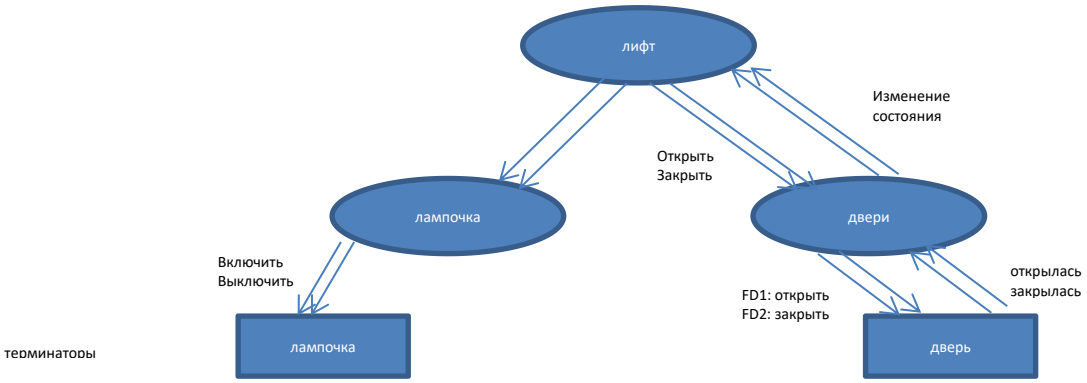
Каналы управления

Последовательность действий и событий, которые происходят в ответ на поступление незапрашиваемого события, когда подсистема находится в определенном состоянии. Если терминаторы реагируют на какую-то предыдущую деятельность системы, то это тоже включается канал управления – есть событие к терминатору, и ответное от терминатора.

КУ должен иметь конец. Не должно быть к постоянному запросу события от терминатора для изменения системы для запроса...

После построения МВО надо выполнить процесс имитирования (тесты) – понять, насколько правильно была выбрана модель, нет ли ошибок. Необходимо рассмотреть всевозможные начальные состояния объектов подсистемы. Далее, в любом состоянии надо проверить, как подсистема будет реагировать на все незапрашиваемые события, и построить соответствующие каналы управления.

Рассмотрим пример: МВО лифта.



В свою очередь, над лифтом может быть терминатор блока управления, с которого будут приходить команды. Переход состояний может происходить таким образом: лифт с открытыми дверями -> лифт с закрытыми -> лифт движется -> лифт с закрытыми -> двери открыты. Подача сигнала на переезд на этаж – незапрашиваемое событие; приезд на этаж – тоже; дверь закрылась – запрашиваемое, потому что изменяет состояние. Если система расширяется пользователями, то уже пользователь будет взаимодействовать с лифтом; пользователь принимает незапрашиваемые события, а события лифта – запрашиваемые.

Есть начальные состояния объектов, являющиеся непротиворечивыми. Каждая модель состояний рисуется в виде вертикальной последовательности событий.

Для переходов из состояния в состояние выделяют время задержки – должно пройти время нахождения в состоянии. Также выполняют время выполнения действия. Вводится объект таймера, выполняющий задержку по времени на то время, которое объект должен находиться в этом состоянии.

К примеру, объект не должен стоять с открытыми дверьми больше времени T – по истечению времени T объект переходит в закрытое состояние.

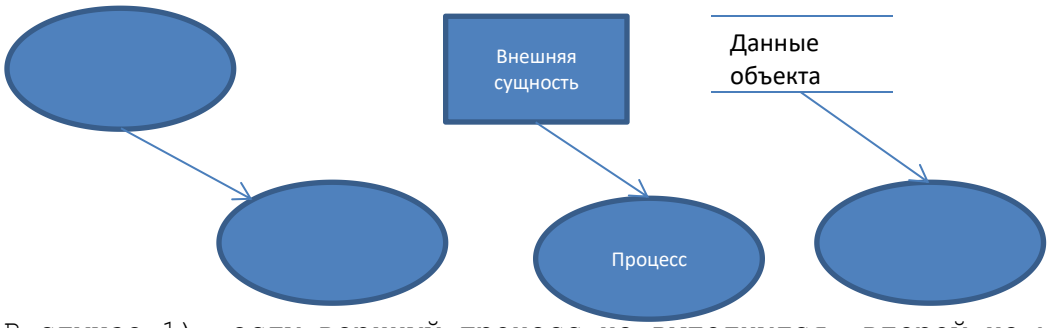
23.05.14----------

Диаграммы потоков данных действия (ДПДД)

Выделили действия, состояния, сформировали ДПС. Существует модель де Марка – действия расписываются в виде процессов, формируемых в системе. Любое действие рассматривается как процесс. ДПДД строится для каждого действия каждого состояния каждой модели состояний.

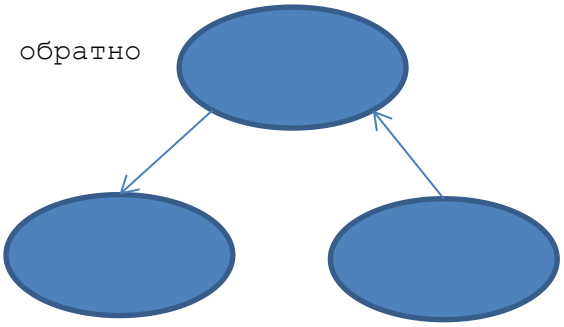
На диаграмме каждый процесс рисуется овалом. При написании псевдокода выделяется последовательность действий – здесь мы отходим от этого принципа; процесс может выполняться, когда будут доступны все данные, необходимые для его выполнения.

Процессы могут получать данные от других процессов и от каких-либо внешних сущностей.



В случае 1) ,если верхний процесс не выполнился, второй не может выполниться. В 2), процесс может выполниться, поскольку данные внешних сущностей всегда доступно. То же самое касается атрибутов самого себя в 3)

Результатом процесса могут быть данные, возвращающиеся обратно



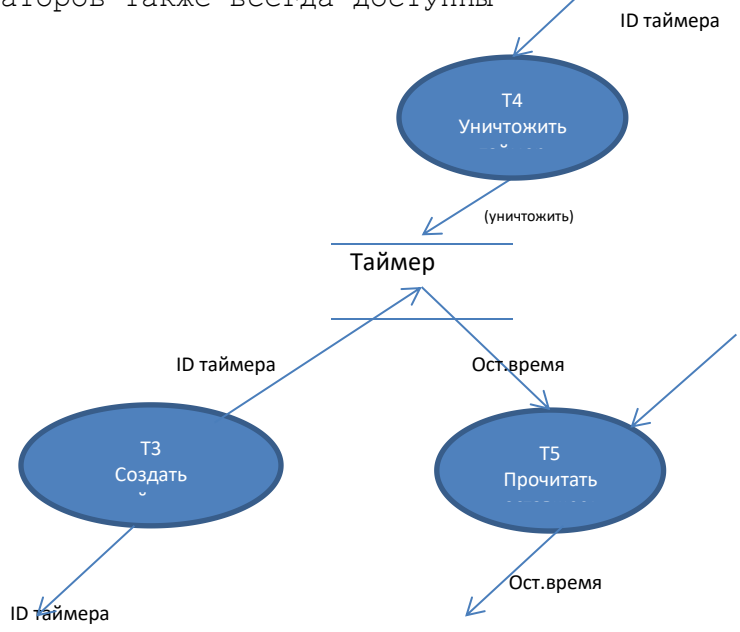
Возможно условное выполнение – процесс выполняется в зависимости от условий. При этом нет передачи данных, а есть условность выполнения – от «условного» процесса рисуется пунктирная стрелочка с указанием условия выполнения, для каждого перехода.

Правила выполнения для ДПДД:

1 процес может выполняться, когда всех входы доступны.

2 выводы процесса доступны, когда он завершает своё выполнение.

3 данные событий (^ просто стрелка сверху) всегда доступны; данные из архивов данных и терминаторов также всегда доступны



Все процессы можно разбить на четыре типа.

Аксесоры – процессы, которые читают какой-либо атрибут, записывают, создают или уничтожают объекты.

Генераторы событий – стрелочка наружу процесса. (// ^ их нет)

Процессы преобразования – выполняют каки-лбо вычисления.

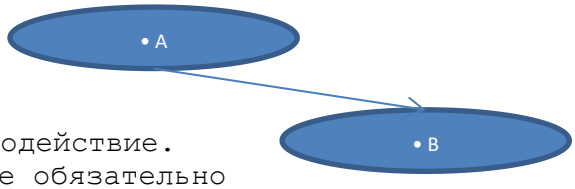
Процессы проверки – условные переходы.

Каждый процесс нужно чётко именовать и описывать. Аксессоры – какие атрибуты считывают или записывют, какие объекты создают или уничтожают. Генераторы событий – результат-событие, метка события. Преобразования – что делают. Проверки – «проверить, что...»

Все процессы в подсистеме объединяются в единую таблицу. В разных действиях могут происходить одни и те же процессы - они будут общими. Общие процессы могут выполнять одну и ту же функцию, читать и записывать и создавать и уничтожать одни и те же объекты, и т.д..

ID процесса	Тип	Название	Где используется	
			Модель состояний	действие

На основе выдленных аксессорных процессов строится модель доступа к объектам. На модели доступа, модели состояний (объектов) рисуются вытянутыми овалами.
Если А использует аксессор модели состояний В,
то рисуется стрелка, А будет аксессором.



Аксессоры реализуются добавлением в объект действий по записи и чтению атрибутов.

Данная диаграмма представляет синхронное взаимодействие. Может использоваться совместно с асинхронной – не обязательно данные переносят события. Автобусная остановка: событие «пришел автобус», говорит лишь о том что «пришел транспорт», подходящий по функции; а может и нести информацию «пришел автобус №». Если данное не переносится, то с объектом надо вступить в аксессорное взаимодействия.

Диаграмма построена для подсистемы. Переходим на доменный уровень.

В любом случае, когд имеется большая система, мы разбиваем задачу на домены.
Прикладные - основные домены, которые решают непосредственно задачу.
Сервисные домены – обеспечивают функции, необходимые для поддержания прикладного домена, всевозможные сервисные функции.
Архитектурный – домен, отвечающий за архитектуру построения системы (один домен на систему); обеспечивает общие механизмы для управления системы.
Домены реализации – стандартные, библиотечные функции и так далее. Дают возможность лёгкой замены одной реализации на другую.

Мосты, клиенты, сервера

Один домен использует возможности и механизмы другого – между этими доменами есть мост. Тот, который предоставляет возможности, называют сервером; использующий – клиентом. Клиент рассматривает мост как набор каких-то предложений, которые кто-то ему представляет. Сервер – набор трбований. Диаграмма доменного уровня как правило содержит в верхней части – домены, наиболее осведомленне о системе (прикладные), внизу – сервисные и реализации.

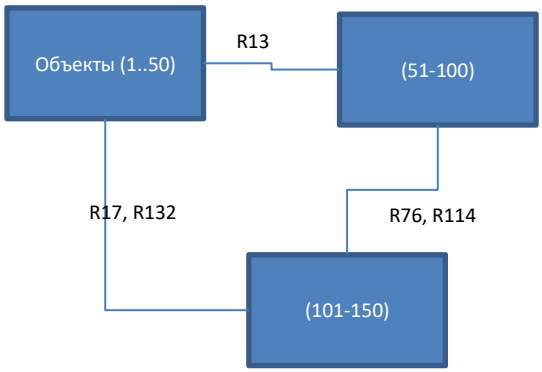
При разбиении задачи для проектирования каждого домена можно использовать разные технологии. Например в задаче отрисовки, непоредственно рисование – структурный подход, а различные взаимодействия на сцене – объектный.

Доменный подход позволяет в дальнейшем легко заменить один домен на другой; сервер рассматривается как набор предложений.

Прикладной домен разбивается на подсистемы, вто время как сервисный – просто набор

функций. Для домена, так же как и для подсистемы, рисуется три диаграммы.

Модель связей подсистем (аналог – информационная модель для подсистем). В рамке рисуется подсистема, перечисляются связи.



Модель доступа подсистем. Количество взаимодействий между подсистемами должно быть минимальным.

В ЛР№5 – не нужно брать большую задачу, нужно просто пройти технологический этап создания модели. Документы: информационная модель, на её основе выделяются сущноти и формируются жизненные циклы этих сущностей (начиная с наиболее осведомленных). На основе их сроится диаграмма переходов состояний; для проверки их корректности – таблица (событие X __). Модель взаимодейстия объектов в подсистеме – анализируются все возможные начальны состояния; формируется список событий извне; анализируется какие запрашиваемые и незапрашиваемые – на основе незапрашиваемых и начальных состояний строятся каналы управления.

Не обязательно строить ВСЕ начальные состояния и ВСЕ события, надо показать корректно построенные каналы управления.

Рассматриваются действия состояний – расписываются с использованием процессов, строится ДПДД. Для какой-то одной модели состояний нужно расписать все ДПДД (для каждого действи своя); выделяются аксессорные процессы и рисуется модель доступа.

Задача небольшая, домен содержит одну подсистему, диаграмма доменного уровня не нужна.

Задача – любое оборудование, имеющее технологический цикл. Чайник, самолёт, машина.

Объектно Ориентированный Дизайн (проектирование)

На основе полученных в ООА документов мы приходим к проектированию. Роль ему отводится достаточно скромная – из документов ООА получаем документы ООД.

Одна из нотаций ООД: нотация Буча и Бухра. Четыре основных диаграммы.

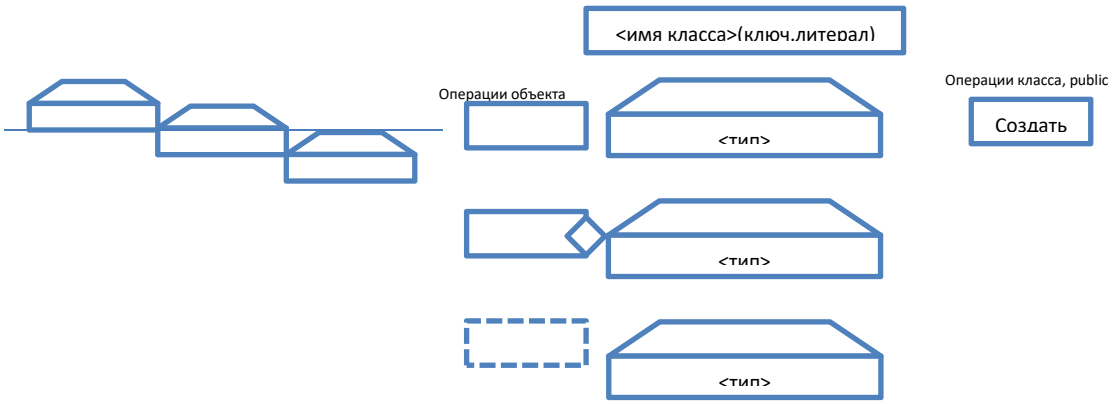
Диаграмма класса – описывает внешне представление одного класса.

Схема класса – внутреннее представление, структура класса.

Диаграмма зависимостей – схема использования.

Диаграмма наследований – схема наследования классов.

Диаграмма класса: (v – бока, «гробы»)

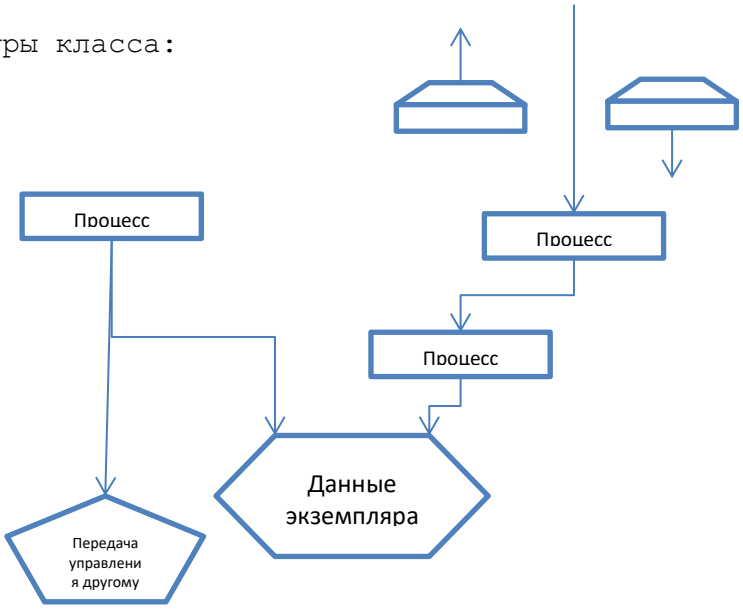


Слева: принимаемые, возвращаемые, обрабатываемые данные

Ромб – обрабатывает исключительную ситуацию.

Пунктир – отсрочиваемая операция, callback выполняемый в результате события

Схема структуры класса:



Класс реализуется по слоям. Есть публичные методы, есть приватные; доступ к нижним слоям напрямую снаружи недопустим, как и вызов пабликом паблика.

Диаграмма зависимостей: схема использования. Двойная стрелка – дружественная связь.

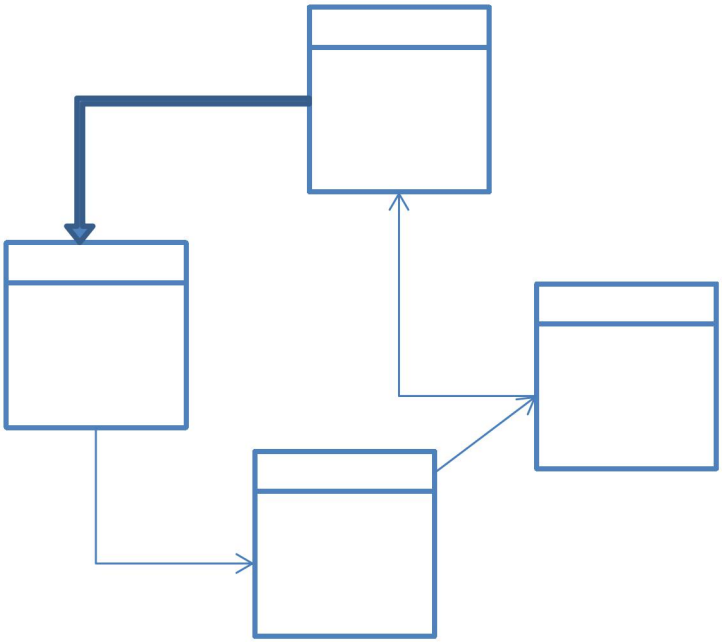
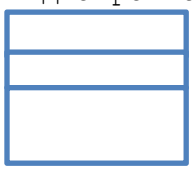


Диаграмма наследования. Класс разбивается на методы и атрибуты.

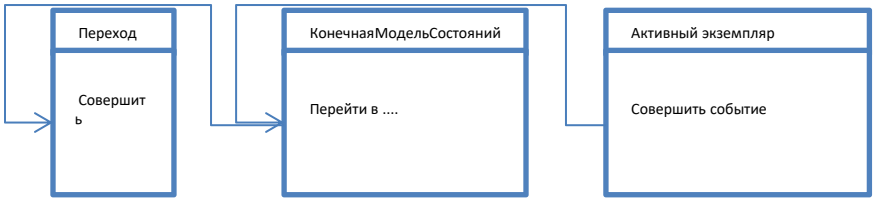


30.05.2014----------

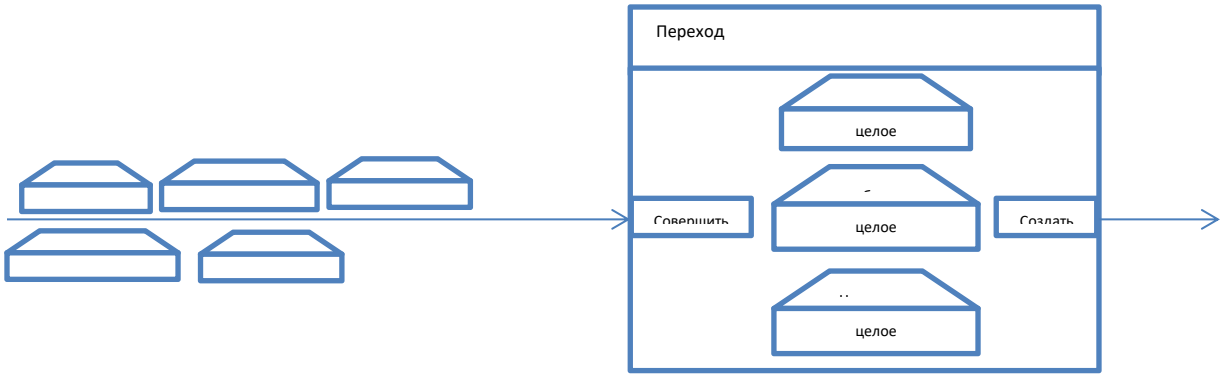
Из документов ООА довольно легко получить ^ диаграммы.

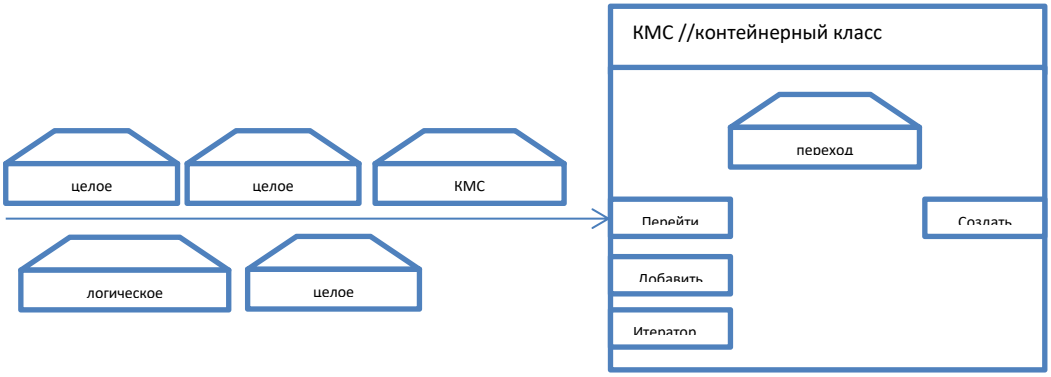
Создаются несколько классов для архитектурного домена, задача которых – задать правила перехода из состояния в состояние при возникновении событий. Эти классы представляют активный экземпляр; все остальные будут производными от активного.

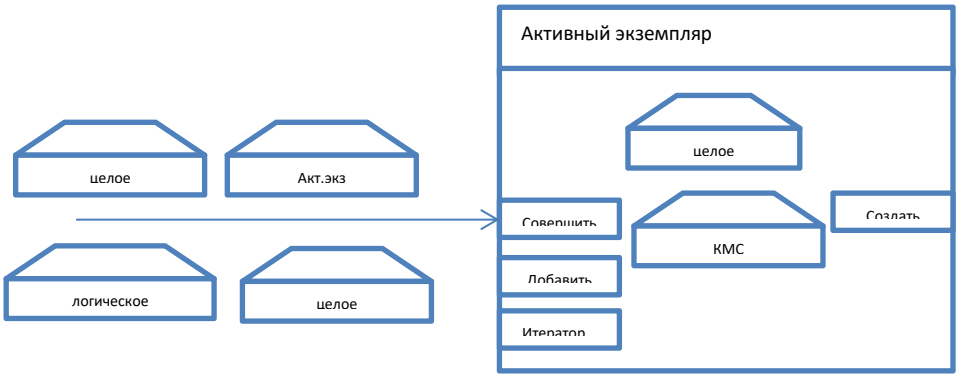
Классы, выделяемые для архитектурного домена:



Задаются переходы: КМС будет состоять из возможных переходов.







КМС делается шаблонным классом – позволяет сделать систему гибкой.

Все активные классы будут производными от «акт экз». Соответственно у АЭ должен быть конструктор, в который мы передаём начальное состояние и КМС.

В схему можно добавить параметр игнорирования перехода; возвращается флаг. Либо проверка происходит на уровне формирования активного класса.

Выделены пассивные и активные объекты. Для активных выделяются жизненные циклы.

Диаграмма пассивного класса должна включать в себя: атрибуты (компоненты экземпляра), аксессоры (сет и гет) (как объекта так и класса).

Диаграмма активного: атрибуты, аксессоры (как правило), методы объекта (тейкеры событий) – действия, которые соответствуют состояниям в модели состояний объекта. «Затребовать событие» - реакция на событие.
Кроме этого, активный класс включает инициализатор КМС.

Выделяются объекты, осуществляющие только связь между другими. Если для таких объектов не выделять атрибутов, то их можно рассматривать как объекты определители связи. Такие объекты не имеют аксессоров, только тейкеры и инициализаторы.

ОО нотаций достаточно много, ^ рассмотрена нотация Буча и Бухра – можно использовать и другие, например UML. УМЛ предлагает только набор всевозможных диаграмм и крайне избыточна – одно и тоже можно представить разными диаграммами.