# Билет 1

**1. Эволюция языков программирования. Жизненный цикл программы**

В первых электронных машинах программы вводились покомандно в двоичном формате с пульта управления оператором. Очевидным шагом вперед стало использование псевдомашинного языка (Ассемблера), где вместо двоичных кодов стали писать текстовые названия, например,

ADD A,5

По этой команде к переменной А добавляется 5, т. к. ADD обозначает прибавить.

В создании компиляторов решающую роль сыграла разработка Ноэмом Хомским формальной теории языков. Хомский выделяет четыре класса языков: автоматные (регулярные), контекстно-свободные (КС-языки), контекстно-зависимые и языки со свободной структурой. Для целей программирования наибольшее значение имеют первые два класса. Для каждого языка используются определенные математические средства их разбора. Для автоматных языков такими средствами являются детерминированные автоматы без памяти, для КС-языков – недетерминированные автоматы с памятью. Применяются также деревья разбора (например, деревья нисходящего синтаксического разбора) и другие формализмы.

По сути, эра автоматического программирования началась с появления первого компилятора – программы, которая переводит код с языка высокого уровня в машинный код. В конце 50-х годов прошлого века первый компилятор был создан для языка Фортран американцами Бэкусом и Науром. В 1960-х годах число языков стремительно растет: ПЛ/1, Алгол, Бейсик, Кобол, РПГ и др. Никлаус Вирт (Швейцария) создает язык Паскаль. В 1970-е годы появляется язык Си (авторы Кен Томпсон и Денис Ритчи). Все эти языки относились к категории процедурно-ориентированных языков. Минимальной единицей кода была процедура (функция, модуль). В языках Паскаль и Си использовались сложные типы данных: записи в Паскале, структуры – в Си. Преимуществами Си стали наличие указателей (на ячейки памяти), возможность динамически выделять память под переменные и освобождать ее.

Этапы разработки образуют жизненный цикл программы. Он включает в себя:

• анализ требований: определяются и уточняются требования заказчика, разрабатываются соответствующие документы (техническое задание, описание применения и др.);

• проектирование: формируются модели будущей системы или другие спецификации, определяются компоненты архитектуры, условия взаимодействия компонентов, представление данных, алгоритмы работы;

• реализацию (кодирование): осуществляется выбор и обоснование языка(ов) программирования, разработка программных модулей и интерфейса, интеграция модулей;

• тестирование и отладку;

• документирование: разрабатываются документы на программное обеспечение, включающие описание применения, руководство пользователя, руководство системного программиста, тексты программ и др.;

• внедрение;

• эксплуатацию;

• сопровождение.

Основным нормативным документом, регламентирующим жизненный цикл ПО, является международный стандарт ISO/IEC 12207 (ISO – International Organization of Standardization – Международная организация по стандартизации, IEC – International Electrotechnical Commission – Международная комиссия по электротехнике).

Соответственно этапам жизненного цикла программного обеспечения разрабатывались технологии, которые их поддерживали. Известны следующие основные модели жизненного цикла программы: водопадная (каскадная), водопадная с возвратами и спиральная [1].

Каждый этап водопадной модели завершается выпуском полного комплекта документации, достаточной для продолжения разработки другой командой разработчиков. Недостаток водопадной модели состоит в возможности непредусмотренных возвратов на ранние этапы с целью коррекции или исправления части проекта. Кроме того, такая модель не предусматривает возможность изменения в ходе разработки концепции системы, что существенно при создании сложных программных комплексов, для которых трудно предусмотреть все аспекты работы. В связи с этим появляется водопадная модель с возвратами (рисунок 1.1).

Основной минус водопадной модели с возвратами – проект выполняется по-прежнему достаточно медленно в связи с его этапностью (переход на следующий этап нельзя осуществить до полного завершения предыдущего этапа). Поэтому появляется более прогрессивная форма жизненного цикла – спиральная модель (рисунок 1.2) [1].

В спиральной модели сначала создается первая версия системы, затем выполняется доводка компонентов разработанной архитектуры, модернизация и повышение эффективности программных модулей, т. е. реализуется следующий виток, и т. д. Каждый виток спирали соответствует поэтапной модели создания фрагмента или версии программного изделия, на нем уточняются цели и характеристики проекта, определяется его качество, планируются работы следующего витка спирали. Таким образом, углубляются и последовательно конкретизируются детали проекта, и в результате выбирается обоснованный вариант, который доводится до реализации

**2.** **Пример простого делегата в С++**

Класс делегат расширяет понятие указателя на функцию. Используя этот класс Вы можете передать ссылку на метод объекту-делегату. Далее объект-делегат может быть передан клиентскому коду, который может косвенно вызвать метод. При этом код, который обращается с делегатом не знает что за метод скрывается за ним. Преимущество делегата состоит в том, что Вы можете динамически передавать ему указатели на методы и тем самым изменять поведение программы без перекомпиляции клиентского кода. Программа, использующая делегат будет оставаться в счастливом неведении о ваших манипуляциях с методами и их обладателями

#include <iostream>

#include <stdio.h>

using namespace std;

class Delegate

{

struct S {};

typedef void (S::\* Func)();

S\* \_ptr;

Func \_func;

public:

Delegate()

: \_ptr(0), \_func(0)

{

}

template <class Zap>

Delegate(Zap\* zap, void (Zap::\* func)())

: \_ptr(0), \_func(0)

{

\_func = reinterpret\_cast<Func>(func);

\_ptr = reinterpret\_cast<S\*>(zap);

}

void operator () ();

};

inline void Delegate::operator () ()

{

(\_ptr->\*\_func)();

}

class Testing

{

public:

Testing()

{

//Передаем делегату указатель на функцию

\_delegate = Delegate(this, &Testing::Alkash);

}

void Alkash()

{

cout << "\nAlkash needs bottle " << endl;

}

void MAN()

{

cout << "\nMAN needs book " << endl;

}

void Start()

{

//Делаем вызов делегата первый раз

\_delegate();

//Теперь самое интересное. Лёгким движением рук подставляем

// в делегат другую функцию

\_delegate = Delegate(this, &Testing::MAN);

//Второй раз вызываем делегат

\_delegate();

}

private:

Delegate \_delegate;

};

int main()

{

Testing test;

test.Start();

getchar();

return 0;

}

**3.** **Создание Unit-теста в Visual C++**

Модульное тестирование, иногда блочное тестирование или юнит-тестирование — процесс в программировании, позволяющий проверить на корректность отдельные модули исходного кода программы, наборы из одного или более программных модулей вместе с соответствующими управляющими данными, процедурами использования и обработки

Используйте Visual Studio для определения и выполнения модульных тестов, чтобы поддерживать работоспособность кода, обеспечивать покрытие кода и находить ошибки и сбои раньше, чем это сделают ваши клиенты. Чаще запускайте модульные тесты, чтобы убедиться, что ваш код работает правильно.

**Программа**

#include <math.h>

#include <iostream>

#include <stdio.h>

#include <conio.h>

using namespace std;

int Fib(int);

int Fib\_R(int);

//int k;

int main()

{

int n;

do{

do {

cout << "\nInput n" << endl;

cin >> n;

} while (n < 0);

cout <<"\nRecursive "<<Fib\_R(n);

cout << "\nNOT Recursive " << Fib(n);

//cout<<"\nk= "<<k;

cout << "\nRepeat y,Y,else - EXIT" << endl;

} while (tolower(\_getch()) == 'y');

}

int Fib\_R(int n)

{

//k++;

//if (n == 1) return 1;

if (n < 2) return n;

return Fib\_R(n - 1) + Fib\_R(n - 2);

}

int Fib(int n)

{

int a = 1, b = 1, c;

//if (n == 1) return 1;

if (n < 2) return n;

for (int i = 3; i <= n; i++)

{

c = a + b;

a = b; b = c;

}

return b;

}

**ТЕСТ**

#include "pch.h"

#include "CppUnitTest.h"

#include "C:\Users\btr48\source\repos\laba1sem2\laba1sem2\laba1sem2.cpp"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace LabaTest

{

TEST\_CLASS(LabaTest)

{

public:

TEST\_METHOD(TestMethod1)

{

int t;

t = Fib\_R(6);

Assert::AreEqual(t, 7);

}

};

}

# Билет 2

**1.Требования к программному обеспечению**

**Функциональные и нефункциональные требования** (Functional and Non-functional Requirements) – **функциональные** требования задают “что” система должна делать; **нефункциональные** – с соблюдением “каких условий” (например, скорость отклика при выполнении заданной операции); часто функциональные требования представляют в виде сценариев (вариантов использования) Use Сase.

**Потребности** (needs) – отражают проблемы бизнеса, персоналии или процесса, которые должны быть соотнесены с использованием или приобретением системы.

Группа функциональных требований

**Бизнес-требования** (Business Requirements) – определяют высокоуровневые цели организации или клиента (потребителя) – заказчика разрабатываемого программного обеспечения.

**Пользовательские требования** (User Requirements) – описывают цели/задачи пользователей системы, которые должны достигаться/выполняться пользователями при помощи создаваемой программной системы. Эти требования часто представляют в виде вариантов использования (Use Cases).

**Функциональные требования** (Functional Requirements) – определяют функциональность (поведение) программной системы, которая должна быть создана разработчиками для предоставления возможности выполнения пользователями своих обязанностей в рамках бизнес-требований и в контексте пользовательских требований.

**Группа нефункциональных требований** (Non-Functional Requirements)

**Бизнес-правила** (Business Rules) – включают или связаны с корпоративными регламентами, политиками, стандартами, законодательными актами, внутрикорпоративными инициативами (например, стремление достичь зрелости процессов по CMMI 4-го уровня), учетными практиками, алгоритмами вычислений и т.д. На самом деле, достаточно часто можно видеть недостаточное внимание такого рода требованиям со стороны сотрудников ИТ-департаментов и, в частности, технических специалистов, вовлеченных в проект. Business Rules Group дает понимание бизнес-правила, как “положения, которые определяют или ограничивают некоторые аспекты бизнеса. Они подразумевают организацию структуры бизнеса, контролируют или влияют на поведение бизнеса”. Бизнес-правила часто определяют распределение ответственности в системе, отвечая на вопрос “кто будет осуществлять конкретный вариант, сценарий использования” или диктуют появление некоторых функциональных требований.   
  
В контексте дисциплины управления проектами (уже вне проекта разработки программного обеспечения, но выполнения бизнес-проектов и бизнес-процессов) такие правила обладают высокой значимостью и, именно они, часто определяют ограничения бизнес-проектов, для автоматизации которых создается соответствующее программное обеспечение.

**Внешние интерфейсы** (External Interfaces) – часто подменяются “пользовательским интерфейсом”. На самом деле вопросы организации пользовательского интерфейса безусловно важны в данной категории требований, однако, конкретизация аспектов взаимодействия с другими системами, операционной средой (например, запись в журнал событий операционной системы), возможностями мониторинга при эксплуатации – все это не столько функциональные требования (к которым ошибочно приписывают иногда такие характеристики), сколько вопросы интерфейсов, так как функциональные требования связаны непосредственно с функциональностью системы, направленной на решение бизнес-потребностей.

**Атрибуты качества** (Quality Attributes) – описывают дополнительные характеристики продукта в различных “измерениях”, важных для пользователей и/или разработчиков. Атрибуты касаются вопросов портируемости, интероперабельности (прозрачности взаимодействия с другими системами), целостности, устойчивости и т.п.

**Ограничения** (Constraints) – формулировки условий, модифицирующих требования или наборы требований, сужая выбор возможных решений по их реализации. В частности, к ним могут относиться параметры производительности, влияющие на выбор платформы реализации и/или развертывания (протоколы, серверы приложений, баз данных, ...), которые, в свою очередь, могут относиться, например, к внешним интерфейсам.

**Системные требования** (System Requirements) – иногда классифицируются как составная часть группы функциональных требований (не путайте с как таковыми “функциональными требованиями”). Описывают высокоуровневые требования к программному обеспечению, содержащему несколько или много взаимосвязанных подсистем и приложений. При этом, система может быть как целиком программной, так и состоять из программной и аппаратной частей. В общем случае, частью системы может быть персонал, выполняющий определенные функции системы, например, авторизация выполнения определенных операций с использованием программно-аппаратных подсистем.

**2.** **Шаблоны классов и функций**

**Шабло́н (template)** — средство языка C++, предназначенное для создания обобщённых алгоритмов, без привязки к некоторым параметрам (например, типам данных, размерам буферов).

Точное значение параметра передается на этапе компиляции. Таким образом, создается функция, которая автоматически перегружает себя.

template <class *список\_параметров*>

*тип* *имя\_функции*(*список\_параметров*)

{

// тело функции

}

Виды параметров шаблона:

* **типизированный параметр** – ключевое слово class и идентификатор;
* **нетипизированный параметр** – имя типа и идентификатор, определяющие константу.

Описанные параметры могут быть использованы в заголовке и теле функции

Шаблон с несколькими параметрами

**template** < **typename** параметр1, **typename** параметр2 >

void f1 (параметр1 *имя\_переменной1*, параметр2 *имя\_переменной2*)

{

// тело функции

}

*Порядок перечисления параметров в списке шаблона значения не имеет .*

Объявление шаблона не приводит к генерации кода.

**Генерация** (**реализация шаблона**) происходит в **месте вызова функции**.

Код генерируется с используемыми в данном вызове типами аргументов. При повторном вызове с аналогичными аргументами генерации не происходит.

Если встречается другой, разрешенный тип аргументов, то происходит еще одна реализация шаблона

Корме параметров – типов допускается использование параметров – констант. В этом случае вместо имени параметра подставляется значение константы из определения шаблона.

Для определения фактического типа и значения константы, исследуются фактические аргументы, переданные при вызове функции.

Процесс определения типов и значений аргументов шаблона по известным фактическим аргументам функции называется **выведением (deduction) аргументов шаблона**

Если для некоторых типов данных требуется отличная от шаблона реализация, то можно описать обычную функцию, список типов аргументов и возвращаемого значения которой соответствуют объявлению шаблона. Такая, перегружающая шаблон, функция, называется **специализацией шаблонной функции**

**Шаблоны классов**

**template** < **typename** параметр>

**class** имя\_класса

{

// тело класса

};

Каждая версия класса, создаваемая по шаблону, содержит одинаковый базовый код с изменёнными параметрами шаблона.

Для шаблона может использоваться частичная или полная специализация:

**template** <> **class** ***имя\_класса*** <***имя\_специализированного\_типа***>

{

}

Допустимо использование шаблонов классов и функций с произвольным числом аргументов.

Шаблон **функции** с переменным числом параметров объявляется:

template <typename **...** *Имя\_пакета\_аргументов*>

*тип\_результата*

*имя\_функции*(*Имя\_пакета\_аргументов* **...** args);

Шаблон **класса** с переменным числом параметров объявляется:

template <typename **...** *Имя\_пакета\_аргументов* >

class *Имя\_класса*;

**3.Создание DLL**

**DLL** (англ. *Dynamic-link library* - динамически загружаемая библиотека) - реализованные компанией Microsoft общие библиотеки в ОС Windows и OS / 2. Как правило, библиотеки имеют расширение файла \*. dll, \*. ocx (для библиотек, содержащих элементы управления ActiveX) или \*. drv (драйверы старых версий ОС). DLL может содержать код, данные и ресурсы в любой комбинации.

Под Windows динамические библиотеки хранятся в файлах с расширением \*. dll. Кроме подпрограмм, там могут также храниться другие ресурсы, например, иконки или битмапы. В коде приложения, использующего функции из динамической библиотеки, загрузки и выгрузки библиотеки должны быть прописаны непосредственно. Компилятору не требуется код функций, содержащихся в динамических библиотеках. При запуске программы, она, однако, будет искать соответствующую dll. Если такая dll не найдена на компьютере, то программа выдаст сообщение об отсутствии динамической библиотеки

# Билет 3

**1. Принципы структурного программирования**

Значительная часть технологий ориентировалась на этап проектирования. Никлаус Вирт предложил концепцию структурного программирования. Оператор goto «изгонялся» из языка программирования. Принципы структурного программирования включали нисходящее проектирование, модульность и пошаговую детализацию. Идеи структурного программирования так или иначе проявились в таких технологиях проектирования программ, как SADT, SREM, методике Джексона и др.

Графический язык системы SADT − это иерархический структурированный набор диаграмм, причем каждый блок диаграммы раскрывается более детально с помощью другой диаграммы. Структура проекта представляется со все большей степенью детализации по мере его разработки

Система SREM использовалась для автоматизации создания и анализа требований к программному проекту и содержала язык определения требований (RSL) к данным и операциям их обработки, посредством которого устанавливались связи между объектами. Предложения RSL обрабатывались с помощью специализированного процессора

**2.Паттерн строитель**

Шаблон Строитель позволяет отделить процесс создания сложного объекта от его реализации. При этом результатом одних и тех же операций могут быть различные объекты. Данный шаблон используется в случае, если процесс создания объекта можно разделить на стадии (шаги). При этом конструирование должно обеспечивать возможность создавать разные объекты. Шаблон Строитель включает двух участников процесса .Строитель (Builder) предоставляет методы для сборки частей объекта, при необходимости преобразовывает исходные данные в нужный вид, создаёт и выдаёт объект. Распорядитель (Director) определяет стратегию сборки: собирает данные и определяет порядок вызовов методов строителя. Задача распорядителя – сокрытие стратегии сборки. Это позволит, при необходимости, модифицировать или даже полностью менять её, не затрагивая остальной код

#include <iostream>

#include <vector>

using namespace std;

class House1 {

public:

vector<string> parts\_;

void ListParts()const {

cout << "House parts: ";

for (size\_t i = 0; i < parts\_.size(); i++) {

if (parts\_[i] == parts\_.back()) {

cout << parts\_[i];

}

else {

cout << parts\_[i] << ", ";

}

}

cout << "\n\n";

}

};

/\*

Интерфейс Строителя объявляет создающие методы для различных частей объектов

Продуктов.

\*/

class Builder {

public:

virtual ~Builder() {}

virtual void Walls() const = 0;

virtual void Roof() const = 0;

virtual void Windows() const = 0;

};

/\*

Классы Конкретного Строителя следуют интерфейсу Строителя и предоставляют

конкретные реализации шагов построения.

\*/

class ConcreteBuilder1 : public Builder {

private:

House1\* house;

/\*

Новый экземпляр строителя должен содержать пустой объект продукта,

который используется в дальнейшей сборке.

\*/

public:

ConcreteBuilder1() {

this->Reset();

}

~ConcreteBuilder1() {

delete house;

}

void Reset() {

this->house= new House1();

}

/\*\*

\* Все этапы производства работают с одним и тем же экземпляром продукта.

\*/

void Walls()const override {

this->house->parts\_.push\_back("Walls");

}

void Roof()const override {

this->house->parts\_.push\_back("Roof");

}

void Windows()const override {

this->house->parts\_.push\_back("Windows");

}

/\*

Конкретные Строители должны предоставить свои собственные методы

получения результатов. Это связано с тем, что различные типы строителей

могут создавать совершенно разные продукты с разными интерфейсами.

Поэтому такие методы не могут быть объявлены в базовом интерфейсе

Строителя (по крайней мере, в статически типизированном языке

программирования). Обратите внимание, что PHP является динамически

типизированным языком, и этот метод может быть в базовом интерфейсе.

Однако мы не будем объявлять его здесь для ясности.

Как правило, после возвращения конечного результата клиенту, экземпляр

строителя должен быть готов к началу производства следующего продукта.

Поэтому обычной практикой является вызов метода сброса в конце тела

метода getHouse. Однако такое поведение не является обязательным, вы

можете заставить своих строителей ждать явного запроса на сброс из кода

клиента, прежде чем избавиться от предыдущего результата.

\*/

House1\* GetHouse() {

House1\* result = this->house;

this->Reset();

return result;

}

};

/\*

Директор отвечает только за выполнение шагов построения в определённой

последовательности. Это полезно при производстве продуктов в определённом

порядке или особой конфигурации.

\*/

class Director {

private:

Builder\* builder;

/\*

Директор работает с любым экземпляром строителя, который передаётся ему

клиентским кодом. Таким образом, клиентский код может изменить конечный

тип вновь собираемого продукта.

\*/

public:

void set\_builder(Builder\* builder) {

this->builder = builder;

}

/\*

Директор может строить несколько вариаций продукта, используя одинаковые

шаги построения.

\*/

void BuildMinimalHouse() {

this->builder-> Walls();

}

void FullHouse() {

this->builder-> Walls();

this->builder-> Roof();

this->builder-> Windows();

}

};

/\*

Клиентский код создаёт объект-строитель, передаёт его директору, а затем

инициирует процесс построения. Конечный результат извлекается из объекта-

строителя.

\*/

void ClientCode(Director& director)

{

ConcreteBuilder1\* builder = new ConcreteBuilder1();

director.set\_builder(builder);

cout << "Minimal house:\n";

director.BuildMinimalHouse();

House1\* p = builder->GetHouse();

p->ListParts();

delete p;

cout << "Full house:\n";

director.FullHouse();

p = builder->GetHouse();

p->ListParts();

delete p;

// Помните, что паттерн Строитель можно использовать без класса Директор.

cout << "Custom house:\n";

builder-> Walls();

builder-> Roof();

p = builder->GetHouse();

p->ListParts();

delete p;

delete builder;

}

int main() {

Director\* director = new Director();

ClientCode(\*director);

delete director;

return 0;

}

**3.Использование директивы Assert**

[Assert](http://en.wikipedia.org/wiki/Assertion_(computing)) — это специальная конструкция, позволяющая проверять предположения о значениях произвольных данных в произвольном месте программы. Эта конструкция может автоматически сигнализировать при обнаружении некорректных данных, что обычно приводит к аварийному завершению программы с указанием места обнаружения некорректных данных.

#include <iostream>

#include <conio.h>

#include <assert.h>

using namespace std;

int main()

{

do{

double a, b;

double c,x;

cout << "\nEnter two numbers. Positive and not zero . b>a" << endl;

cout << "\nInput a" << endl;

cin >> a;

cout << "\nInput b" << endl;

cin>> b;

assert(a > 0);

assert(b> 0);

assert(b > a);

c = sqrt(b - a);

x = a / b;

cout << "\nSquare root from (b-a)= " << c << endl;

cout << "\nDivision = " <<x<< endl;

cout << "\nPress [Y,y] to continue" << endl;

} while (tolower(\_getch()) == 'y');

}

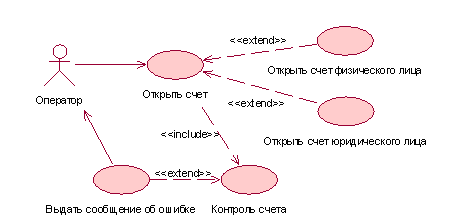
# Билет 4

1. **Диаграммы UML. Диаграмма вариантов использования. Пример**

Важной составляющей технологий проектирования программного обеспечения является язык диаграмм. Так, один из ведущих языков проектирования – UML (Universal Modeling Language) содержит следующие **типы диаграмм**: вариантов использования; классов; объектов; активности (деятельности/activity); последовательности; коммуникаций и др.; графы состояний (state-diagram);

Каждый тип диаграмм «схватывает» некоторую характеристику проектируемой системы, но в общем случае диаграммы используют объекты, классы, сообщения, отношения, условия и методы. Диаграммы не содержат способа реализации метода или класса. Они являются графическими моделями. По замыслу разработчиков (прежде всего, отметим Гради Буча), диаграммы нужны для целостного восприятия системы, ее архитектуры и внутренних связей. «Исключение» составляют диаграммы классов, которые используются для создания программных классов при конвертации UML-диаграмм в языки типа C++.

Диаграммы вариантов использования показывают, как разрабатываемая система взаимодействует с ее пользователями. Пользователи называются акторами (actors). Каждый актор использует систему по-своему (реализует вариант использования). Данная диаграмма носит достаточно общий характер, но она уже содержит будущие классы. Рассмотрим пример диаграммы вариантов использования



**2.Паттерн Синглетон**

Шаблон Одиночка гарантирует создание единственного экземпляра объекта некоторого класса и предоставляет точку доступа для получения этого экземпляра. Существование единственного объекта часто требуется при организации доступа к аппаратному обеспечению, реализации кэша, систем ведения отладочной информации и т. п. Общая схема шаблона достаточно проста Конструктор класса объявляется закрытым – создавать объекты могут только методы самого класса. Для хранения единственного экземпляра используется закрытое статическое поле. Для получения экземпляра служит статический метод

#include <iostream>

using namespace std;

class MySingleton {

public:

// Функция-член для доступа к единственному экземпляру

static MySingleton\* getInstance() {

static MySingleton instance;

return &instance;

}

// Наполняем полезным функционалом, как и любой другой класс

void test() {

cout << "Singleton test" << endl;

}

private:

// Объявляем конструктор закрытым, чтобы нельзя было

// создавать экземпляры класса извне

MySingleton() { }

};

int main() {

// Используем Синглтон

MySingleton::getInstance()->test();

// А это работать не будет, поскольку конструктор - закрытый

// MySingleton singleton;

return 0;

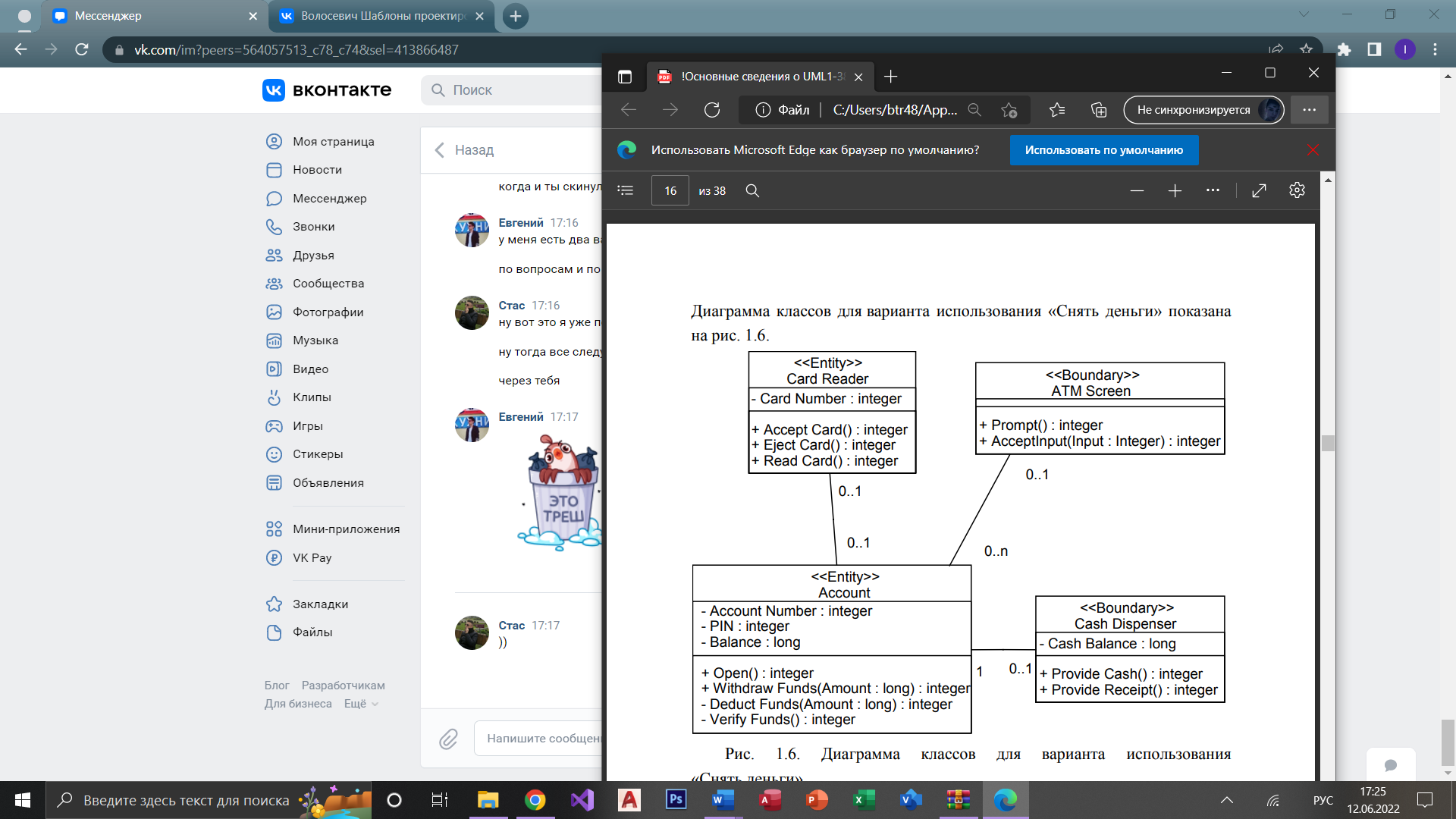
}

**3.Технология ActiveX**

ActiveX - технология Microsoft, предназначенная для написания сетевых приложений. Стандарт ActiveX позволяет программным компонентам взаимодействовать друг с другом по сети независимо от языка программирования, на котором они написаны. С помощью ActiveX можно "оживить" страницы Web эффектами мультимедиа, интерактивными объектами или сложными приложениями. Программные компоненты ActiveX могут быть установлены автоматически на компьютер пользователя по сети с удаленного сервера, причем будет загружен код, подходящий для конкретной платформы клиента, будь то Macintosh, Windows или Unix. ActiveX включает в себя клиентскую и серверную части, а также библиотеки для разработчика. Клиентская и серверная части могут находиться на одном и том же компьютере. Имеется сравнительно большой арсенал компонентов ActiveX, зарегистрированных в реестре Windows. В программах и документах эти компоненты доступны по номерам классов (CLSID).

# Билет 5

1. **Диаграммы классов. Виды зависимостей между классами.**



**Диаграммы классов** являются важнейшим типом диаграмм UML. На этой диаграмме отображаются один или несколько классов и связей между ними

Каждый класс на диаграмме выглядит в виде прямоугольника, разделенного на три части. В первой содержится имя класса, во второй – его атрибуты. В последней части содержатся операции класса, отражающие его поведение (действия, выполняемые классом).

Стереотипы – это механизм, позволяющий разделять классы на категории. В языке UML определены три основных стереотипа классов: Boundary (граница), Entity (сущность) и Control (управление). Граничные классы Граничными классами (boundary classes) называются такие классы, которые расположены на границе системы и всей окружающей среды. Это экранные формы, отчеты, интерфейсы с аппаратурой (такой как принтеры или сканеры) и интерфейсы с другими системами. Классы-сущности (entity classes) содержат хранимую информацию. Они имеют наибольшее значение для пользователя, и потому в их названиях часто используют термины из предметной области. Управляющие классы (control classes) отвечают за координацию действий других классов. Обычно у каждого варианта использования имеется один управляющий класс, контролирующий последовательность событий этого варианта использования.

Атрибут – это элемент информации, связанный с классом

Public (общий, открытый). Это значение видимости предполагает, что атрибут будет виден всеми остальными классами. Любой класс может просмотреть или изменить значение атрибута. В соответствии с нотацией UML общему атрибуту предшествует знак « + ».

Private (закрытый, секретный). Соответствующий атрибут не виден никаким другим классом. Закрытый атрибут обозначается знаком « – » в соответствии с нотацией UML.

Protected (защищенный). Такой атрибут доступен только самому классу и его потомкам. Нотация UML для защищенного атрибута – это знак « # ».

Package or Implementation (пакетный). Предполагает, что данный атрибут является общим, но только в пределах его пакета. Этот тип видимости не обозначается никаким специальным значком

Важнейшими отношениями являются **обобщение** (быть родовым классом для) и ассоциация (какие-то объекты одного класса могут входить в множество объектов другого класса или иметь с ними какую-то общую характеристику).

Для передачи отношения обобщения используется стрелка с пустым трегольником. Стрелка направлена к родительскому (общему) классу.

Ассоциация (association) – это семантическая связь между классами. Их рисуют на диаграмме классов в виде обыкновенной линии

Связи зависимости (dependency) также отражают связь между классами, но они всегда однонаправлены и показывают, что один класс зависит от определений, сделанных в другом. Зависимости изображают в виде стрелки, проведенной пунктирной линией

Агрегации (aggregations) представляют собой более тесную форму ассоциации. Агрегация – это связь между целым и его частью. Агрегации визуализируют в виде линии с ромбиком у класса, являющегося целым

В дополнение к простой агрегации UML вводит более сильную разновидность агрегации, называемую композицией. Согласно композиции, объект-часть может принадлежать только единственному целому, и, кроме того, как правило, жизненный цикл частей совпадает с циклом целого: они живут и умирают вместе с ним. Любое удаление целого распространяется на его части

С помощью обобщений (generalization) показывают связи наследования между двумя классами. На языке UML связи наследования называют обобщениями и изображают в виде стрелок от класса-потомка к классу-предку

Суммируем связи между классами:

 **Агрегация (Aggregation)** − тип связи «часть-целое», в котором «часть» может существовать отдельно от «целого».

 **Композиция (Composition)** − тип агрегации, в котором «части» не существуют вне «целого».

 **Зависимость (Dependency)** − изменение одной сущности (независимой) может влиять на поведение другой сущности (зависимой). Стрелка направлена на независимую сущность.

**2.Запись и использование класса в DLL**

**3.Охрана try-catch**

***Исключительной ситуацией*** или **исключением (exception)** называется прерывание нормального хода выполнения программы в случае возникновения непредвиденного или аварийного события

Для реализации механизма обработки исключений в язык С++ введены три ключевых слова:

* **try** - начало блока исключений;
* **catch** - начало блока, "ловящего" исключение
* **throw** - генерация исключения

Контролируемый блок:

**try**

{

*Операторы*

}

Контролируемые блоки могут быть вложенными.

Среди операторов, могут быть обычные операторы языка Си++ и специальные операторы генерации исключений.

Генерация исключения:

**throw** *выражение\_генерации\_исключения*;

Исключение создается как статический объект, тип которого определяется типом значения *выражения\_генерации\_исключения*.

Местоположение оператора throw называют **точкой выброса исключения**.

Среди операторов контролируемого блока может быть любое количество операторов throw.

После формирования исключения исполняемый оператор **throw** передает управление непосредственно за пределы контролируемого блока, где обязательно должны находятся один или несколько обработчиков исключений:

**catch** (*тип\_исключения имя\_исключения*)

{

*//Операторы блока обработки исключений*

}

Если используется несколько обработчиков исключений, то они должны отличаться друг от друга типами исключений

Формы обработчика исключений:

**1. catch** (тип\_параметра имя\_параметра)

Содержит тип и имя параметра. Наличие имени позволяет передавать информацию из обрабатываемого исключения.

**2. catch** (тип параметра)

Содержит только тип параметра. Используются в случаях, когда важен только тип исключения и сам факт его получения.

**3. catch** (...)

Обработчик реагирует на любое исключение независимо от его типа. Как правило помещается в конце списка обработчиков для «перехвата» оставшихся необработанными исключений.

*Блоки catch обрабатываются последовательно. После нахождения первого соответствующего catch последующие обработчики не проверяются. Нельзя, например, помещать обработчик для базового класса перед обработчиком для производного класса.*

Если тип объекта исключения не сооствествует указанному типу то проверяются типы, в которые можно преобразовать объект при помощи стандартных правил преобразования.

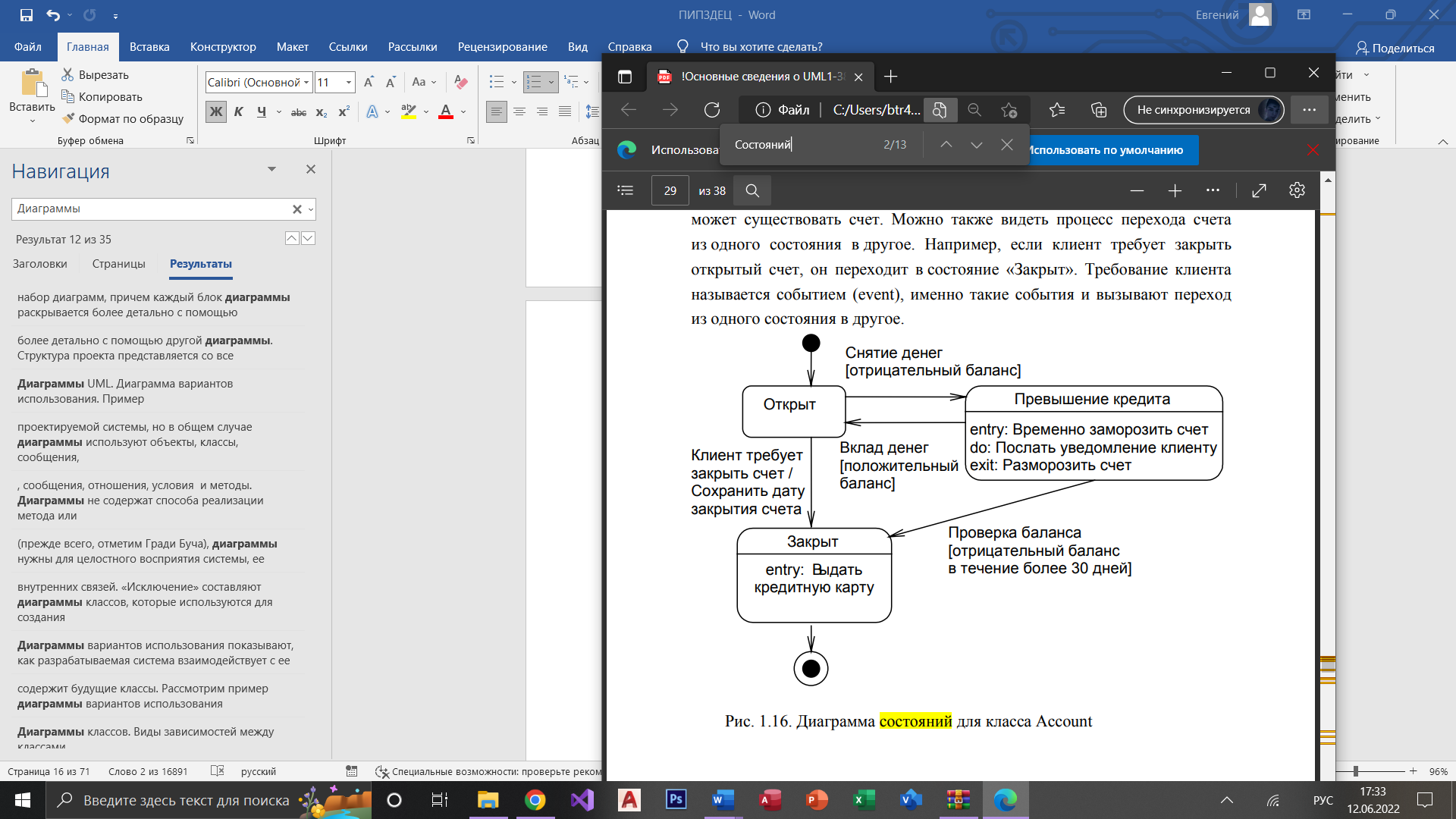
Если в контролируемом блоке **исключение не порождено**, то обработчики исключений обходятся

Если поиск подходящего блока обработки исключения в контролируемом блоке окончился неудачей, то поиск подходящего блока осуществляется в блоках верхнего уровня вложенности.

Для перехода на предшествующий уровень вложенности используется оператор **throw** без параметров.

# Билет 6

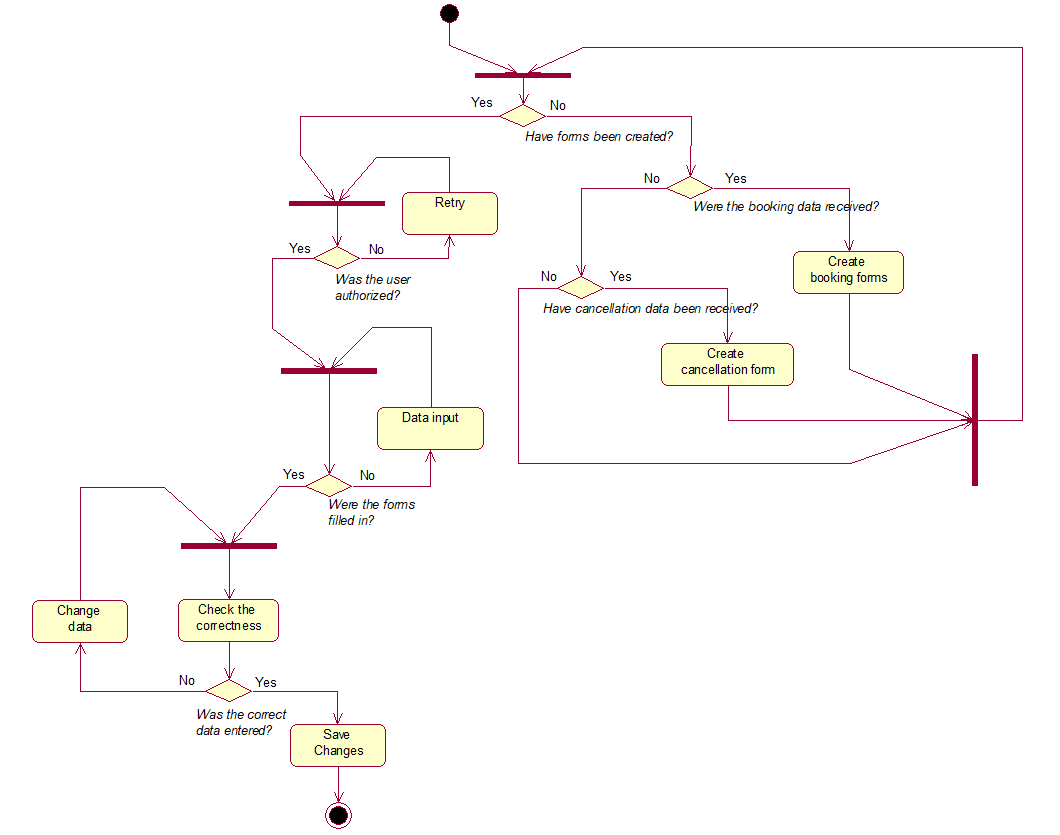
1. **Диаграммы состояний и диаграммы деятельности. Связь между ними.**



**Диаграммы состояний** определяют все возможные состояния, в которых может находиться конкретный объект, а также процесс смены состояний объекта в результате наступления некоторых событий.

На диаграмме имеются **два специальных состояния** – начальное (start) и конечное (stop). Начальное состояние выделено черной точкой, оно соответствует состоянию объекта, когда он только что был создан. Конечное состояние обозначается черной точкой в белом кружке, оно соответствует состоянию объекта непосредственно перед его уничтожением. На диаграмме состояний может быть одно и только одно начальное состояние. В то же время, может быть столько конечных состояний, сколько вам нужно, или их может не быть вообще. Когда объект находится в каком-то конкретном состоянии, могут выполняться различные процессы

С состоянием можно связывать данные пяти типов: деятельность, входное действие, выходное действие, событие и история состояния. **Деятельностью** (activity) называется поведение, реализуемое объектом, пока он находится в данном состоянии. Деятельность – это прерываемое поведение. Оно может выполняться до своего завершения, пока объект находится в данном состоянии, или может быть прервано переходом объекта в другое состояние. Деятельность изображают внутри самого состояния, ей должно предшествовать слово do (делать) и двоеточие. **Входным действием** (entry action) называется поведение, которое выполняется, когда объект переходит в данное состояние. **Выходное действие** (exit action) подобно входному. Однако, оно осуществляется как составная часть процесса выхода из данного состояния. **Переходом** (Transition) называется перемещение из одного состояния в другое. Совокупность переходов диаграммы показывает, как объект может перемещаться между своими состояниями. На диаграмме все переходы изображают в виде стрелки, начинающейся на первоначальном состоянии и заканчивающейся последующим. Переходы могут быть рефлексивными. Объект может перейти в то же состояние, в котором он в настоящий момент находится. Рефлексивные переходы изображают в виде стрелки, начинающейся и завершающейся на одном и том же состоянии. У перехода существует несколько спецификаций. Они включают события, аргументы, ограждающие условия, действия и посылаемые события. **Событие** (event) – это то, что вызывает переход из одного состояния в другое



**Диаграммы активностей** (Activity Diagrams) являются представлением алгоритмов неких действий (активностей), выполняющихся в системе. Именно на диаграмме деятельности представлены переходы потока управления от одной деятельности к другой. Это, по сути, разновидность диаграммы состояний, где все или большая часть состояний являются некоторыми деятельностями, а все или большая часть переходов срабатывают при завершении определенной деятельности и позволяют перейти к выполнению следующей. Диаграммы деятельности позволяют моделировать сложный жизненный цикл объекта, с переходами из одного состояния (деятельности) в другое. В отличие от блок-схем, диаграммы деятельности могут отображать одновременно выполняемые действия

**2.Паттерн Фасад**

Шаблон Фасад позволяет скрыть сложность системы путём сведения всех возможных внешних вызовов к одному объекту, делегирующему их соответствующим объектам системы. Шаблон Фасад строит над набором подсистем интерфейс, причём этот интерфейс может быть абсолютно любым. Допускается существование нескольких фасадов для одного набора подсистем. Клиент вместо прямой работы с подсистемами использует предложенный фасадом интерфейс.

Без применения фасада клиент должен взаимодействовать со всем перечисленным набором объектов, используя их свойства и методы. Фасад берет работу по взаимодействию на себя – теперь клиент использует только объект-фасад, который делегирует работу нужным подсистемам

Из возможных вариантов отметим прозрачные фасады (в этом случае компоненты подсистемы могут быть доступны и через фасад, и в обход его) и статические фасады (фасад является статическим классом – скрываемые объекты не агрегируются, а создаются в методах фасада по необходимости).

#include <iostream>

using namespace std;

class Subsystem1 {

public:

string Operation1() const {

return "\nSubsystem1 is ready ";

}

string Operation2() const {

return "\nFirst rocket is launched ";

}

};

/\*

Некоторые фасады могут работать с разными подсистемами одновременно.

\*/

class Subsystem2 {

public:

string Operation3() const {

return "\nSubsystem2 is ready ";

}

string Operation4() const {

return "\nSecond rocket is launched ";

}

};

/\*\*

\* Класс Фасада предоставляет простой интерфейс для сложной логики одной или

\* нескольких подсистем. Фасад делегирует запросы клиентов соответствующим

\* объектам внутри подсистемы. Фасад также отвечает за управление их жизненным

\* циклом. Все это защищает клиента от нежелательной сложности подсистемы.

\*/

class Facade {

protected:

Subsystem1\* subsystem1\_;

Subsystem2\* subsystem2\_;

/\*

В зависимости от потребностей вашего приложения вы можете предоставить

Фасаду существующие объекты подсистемы или заставить Фасад создать их

самостоятельно.

\*/

public:

Facade(Subsystem1\* subsystem1 = nullptr, Subsystem2\* subsystem2 = nullptr) {

this->subsystem1\_ = new Subsystem1;

this->subsystem2\_ = new Subsystem2;

}

~Facade() {

delete subsystem1\_;

delete subsystem2\_;

}

/\*\*

\* Методы Фасада удобны для быстрого доступа к сложной функциональности

\* подсистем. Однако клиенты получают только часть возможностей подсистемы.

\*/

string Operation() {

string result = "\nFacade initializes subsystems:";

result += this->subsystem1\_->Operation1();

result += this->subsystem2\_->Operation3();

result += "\nFacade orders subsystems to perform the action:";

result += this->subsystem1\_->Operation2();

result += this->subsystem2\_->Operation4();

return result;

}

};

/\*\*

\* Клиентский код работает со сложными подсистемами через простой интерфейс,

\* предоставляемый Фасадом. Когда фасад управляет жизненным циклом подсистемы,

\* клиент может даже не знать о существовании подсистемы. Такой подход позволяет

\* держать сложность под контролем.

\*/

void ClientCode(Facade\* facade) {

cout << facade->Operation();

}

/\*

В клиентском коде могут быть уже созданы некоторые объекты подсистемы. В этом

случае может оказаться целесообразным инициализировать Фасад с этими

объектами вместо того, чтобы позволить Фасаду создавать новые экземпляры.

\*/

int main() {

Subsystem1\* subsystem1 = new Subsystem1;

Subsystem2\* subsystem2 = new Subsystem2;

Facade\* facade = new Facade(subsystem1, subsystem2);

ClientCode(facade);

delete facade;

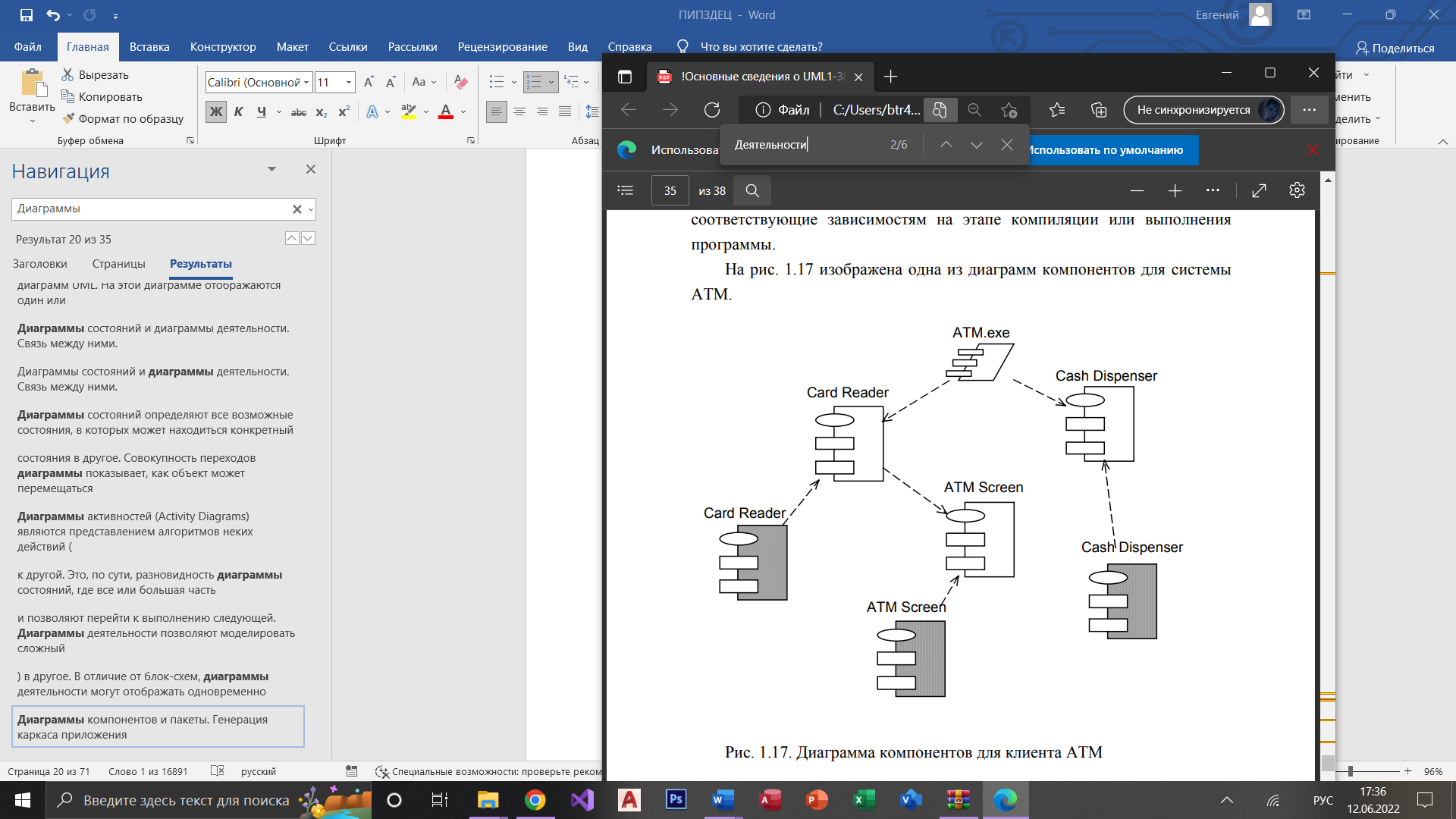
return 0;

}

**3.Многопроектные приложения**

# Билет 7

1. **Диаграммы компонентов и пакеты. Генерация каркаса приложения**



**Диаграммы компонентов** показывают, как выглядит модель на физическом уровне. На них изображены компоненты программного обеспечения и связи между ними. При этом на такой диаграмме выделяют два типа компонентов: исполняемые компоненты и библиотеки кода. Каждый класс модели (или подсистема) преобразуется в компонент исходного кода. После создания они сразу добавляются к диаграмме компонентов. Между отдельными компонентами изображают зависимости, соответствующие зависимостям на этапе компиляции или выполнения программы

|  |  |  |
| --- | --- | --- |
| Таблица 12.2. Графическое изображение *стереотипов компонентов* и их характеристика | | |
| **Графическое изображение и имя по умолчанию** | **Название стереотипа** | **Характеристика стереотипа *компонента*** |
| https://intuit.ru/EDI/12_08_14_3/1407852794-22209/tutorial/50/objects/12/files/12_18.png | Subprogram Specification | **Спецификация подпрограммы**. Содержит описание переменных, процедур и функций и не содержит определений классов |
| https://intuit.ru/EDI/12_08_14_3/1407852794-22209/tutorial/50/objects/12/files/12_19.png | Subprogram Body | **Тело подпрограммы**. Содержит реализацию процедур и функций, не относящихся к каким-то классам, при этом не содержит определений классов или реализаций операций других классов |
| https://intuit.ru/EDI/12_08_14_3/1407852794-22209/tutorial/50/objects/12/files/12_20.png | Main Program | **Главная программа**. Реализует базовую логику работы программного приложения и содержит ссылки на другие компоненты модели |
| https://intuit.ru/EDI/12_08_14_3/1407852794-22209/tutorial/50/objects/12/files/12_21.png | Package Specification | **Спецификация пакета**. Содержит определение класса, его атрибутов и операций. В языке программирования С++ спецификации пакета соответствует отдельный файл с расширением "h" |
| https://intuit.ru/EDI/12_08_14_3/1407852794-22209/tutorial/50/objects/12/files/12_22.png | Package Body | **Тело пакета**. Содержит код реализации операций класса. В языке программирования С++ спецификации пакета соответствует отдельный файл с расширением "cpp" |
| https://intuit.ru/EDI/12_08_14_3/1407852794-22209/tutorial/50/objects/12/files/12_23.png | Task Specification | **Спецификация задачи**. Может содержать определение класса, его атрибутов и операций, которые предполагается использовать в независимом потоке управления |
| https://intuit.ru/EDI/12_08_14_3/1407852794-22209/tutorial/50/objects/12/files/12_24.png | Task Body | **Тело задачи**. Может содержать реализацию операций класса, которые имеют независимый поток управления. |
| https://intuit.ru/EDI/12_08_14_3/1407852794-22209/tutorial/50/objects/12/files/12_25.png | Generic Subprogram | **Типовая подпрограмма**. Содержит описание переменных, процедур и функций, которые могут быть использованы в нескольких программных приложениях. При этом типовая подпрограмма не содержит определений классов |
| https://intuit.ru/EDI/12_08_14_3/1407852794-22209/tutorial/50/objects/12/files/12_26.png | Generic Package | **Типовой пакет**. Содержит определение класса, его атрибутов и операций, которое может быть использовано в нескольких программных приложениях |
| https://intuit.ru/EDI/12_08_14_3/1407852794-22209/tutorial/50/objects/12/files/12_27.png | Database | **База данных**. Содержит определение одного или нескольких классов, их атрибутов и, возможно, операций. При этом соответствующие классы могут быть реализованы в форме одной или нескольких таблиц базы данных |

**Процесс генерации кода** состоит из четырех основных шагов:

1) проверка корректности модели;

2) установка свойств генерации кода;

3) выбор класса, компонента или пакета;

4) генерация кода.

Можно установить несколько параметров генерации кода для классов, атрибутов, компонентов и других элементов модели. Этими свойствами определяется способ генерации программ. Для каждого языка в Rose предусмотрен ряд определенных свойств генерации кода

При генерации кода за один раз можно создать класс, компонент или целый пакет. Код генерируется с помощью диаграммы или браузера. При генерации кода из пакета можно выбрать или пакет логического представления на диаграмме классов, или пакет представления компонентов на диаграмме компонентов. При выборе пакета логического представления генерируются все классы этого пакета. При выборе пакета представления компонентов генерируются все компоненты этого пакета.

Во время генерации кода Rose выбирает информацию из логического и компонентного представлений модели и генерирует большой объем «скелетного» (skeletal) кода:

1) классы - генерируются все классы модели;

2) атрибуты - код включает атрибуты каждого класса, в том числе видимость, тип данных и значение по умолчанию;

3) сигнатуры операций - код содержит определения операций со всеми параметрами, типами данных параметров и типом возвращаемого значения операции;

4) связи - некоторые из связей модели вызывают создание атрибутов при генерации кода;

5) компоненты - каждый компонент реализуется в виде соответствующего файла с исходным кодом.

**2.Паттерн Адаптер**

Шаблон Адаптер нужен для обеспечения коллективной работы классов, которые изначально не были созданы для совместного использования. Такие ситуации часто возникают, когда идёт работа со сторонними библиотеками кода. Нередко их интерфейс не отвечает требованиям клиента , но изменить библиотеку нет возможности. Следовательно, появляется задача адаптации библиотеки для клиента

Адаптируемый класс Adaptee обладает нужной функциональностью, но неподходящим интерфейсом. Класс Adapter реализует интерфейс Target и перенаправляет вызовы к Adaptee, изменяя при необходимости аргументы и возвращаемые значения

#include <iostream>

using namespace std;

class Target {

public:

virtual ~Target() = default;

virtual string Request() const {

return "\n The default target's behavior.";

}

};

/\*\*

\* Адаптируемый класс содержит некоторое полезное поведение, но его интерфейс

\* несовместим с существующим клиентским кодом. Адаптируемый класс нуждается в

\* некоторой доработке, прежде чем клиентский код сможет его использовать.

\*/

class Adaptee {

public:

string SpecificRequest() const {

return "\n eetpadA eht fo roivaheb laicepS";

}

};

/\*\*

\* Адаптер делает интерфейс Адаптируемого класса совместимым с целевым

\* интерфейсом.

\*/

class Adapter : public Target {

private:

Adaptee\* adaptee\_;

public:

Adapter(Adaptee\* adaptee) : adaptee\_(adaptee) {}

string Request() const override {

string to\_reverse = this->adaptee\_->SpecificRequest();

reverse(to\_reverse.begin(), to\_reverse.end());

return "\n Translated " + to\_reverse;

}

};

/\*\*

\* Клиентский код поддерживает все классы, использующие целевой интерфейс.

\*/

void ClientCode(const Target\* target) {

cout << target->Request();

}

int main() {

cout << "\n Work with the Target :\n";

Target\* target = new Target;

ClientCode(target);

cout << "\n\n";

Adaptee\* adaptee = new Adaptee;

cout << "\n Reversed text :( ";

cout << "\nAdaptee: " << adaptee->SpecificRequest();

cout << "\n\n";

cout << "\n Adapter";

Adapter\* adapter = new Adapter(adaptee);

ClientCode(adapter);

cout << "\n";

delete target;

delete adaptee;

delete adapter;

return 0;

}

**3.Интерфейс IUnknown**

#include <conio.h>

#include <stdio.h>

#include <iostream>

#include <objbase.h>

using namespace std;

void trace(const char\* msg)

{

cout << msg << endl;

}

//Интерфейсы

interface IX :IUnknown

{

virtual void \_\_stdcall Fx() = 0;

};

interface IY :IUnknown

{

virtual void \_\_stdcall Fy() = 0;

};

//Предварительное объявление GUID

extern const IID IID\_IX;

extern const IID IID\_IY;

//Компонент

class CA : public IX, public IY

{

virtual HRESULT \_\_stdcall QueryInterface(const IID& iid,

void\*\* ppv);

virtual ULONG \_\_stdcall AddRef() { return 0; }

virtual ULONG \_\_stdcall Release() { return 0; }

virtual void \_\_stdcall Fx() { cout << "Hi, from FX" << endl; };

virtual void \_\_stdcall Fy() { cout << "Hi, from FY" << endl; };

};

HRESULT \_\_stdcall CA::QueryInterface(const IID& iid, void\*\* ppv)

{

if (iid == IID\_IUnknown)

{

trace("return a pointer to IUnknown");

\*ppv = static\_cast<IX\*>(this);

}

else

if (iid == IID\_IX)

{

trace("return a pointer to IX");

\*ppv = static\_cast<IX\*>(this);

}

else

if (iid == IID\_IY)

{

trace("return a pointer to IY");

\*ppv = static\_cast<IY\*>(this);

}

else

{

trace("Interface not supported");

\*ppv = NULL;

return E\_NOINTERFACE;

}

reinterpret\_cast<IUnknown\*>(\*ppv)->AddRef();

return S\_OK;

}

//Функция создания

IUnknown\* CreateInstance()

{

IUnknown\* pI = static\_cast<IY\*> (new CA);

pI->AddRef();

return pI;

}

// IID

// {32bb8320-b41b-11cf-a6bb-0080c7b2d682}

static const IID IID\_IX =

{ 0x32bb8320, 0xb41b, 0x11cf,

{0xa6, 0xbb,0x0,0x80,0xc7,0xb2,0xd6,0x82} };

// {32bb8321-b41b-11cf-a6bb-0080c7b2d682}

static const IID IID\_IY =

{ 0x32bb8321, 0xb41b, 0x11cf,

{0xa6, 0xbb,0x0,0x80,0xc7,0xb2,0xd6,0x82} };

//Клиент

int main(int argc, char\* argv[])

{

HRESULT hr;

trace("Get a pointer to Iunknown");

IUnknown\* pIUnknown = CreateInstance();

trace("Get a pointer to IX");

IX\* pIX = NULL;

hr = pIUnknown->QueryInterface(IID\_IX, (void\*\*)&pIX);

if (SUCCEEDED(hr))

{

trace("IX gotcha successfully!");

pIX->Fx();

}

cout << "Press any key" << endl;

getchar();

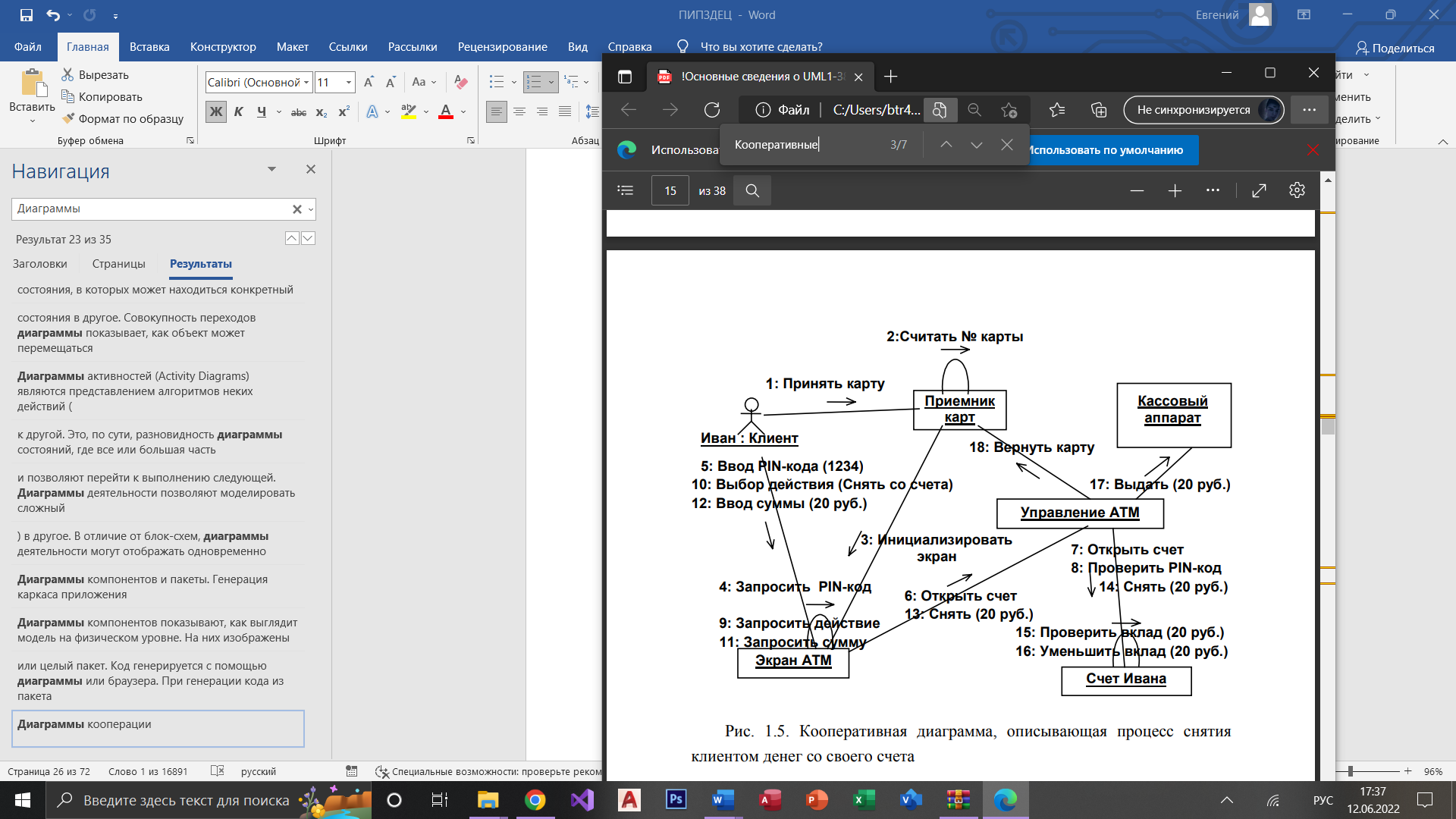
delete pIUnknown;

return 0;

}

# Билет 8

**1. Диаграммы кооперации**



Вторым видом диаграммы взаимодействия является кооперативная диаграмма. Подобно диаграммам последовательности, кооперативные диаграммы (collaborations) отображают поток событий через конкретный сценарий варианта использования. Диаграммы последовательности упорядочены по времени, а кооперативные диаграммы больше внимания заостряют на связях между объектами

Из нее легче понять связи между объектами, однако, труднее уяснить последовательность событий

По этой причине часто для какого-либо сценария создают диаграммы обоих типов. Хотя они служат одной и той же цели и содержат одну и ту же информацию, но представляют ее с различных точек зрения. На кооперативной диаграмме так же, как и на диаграмме последовательности, стрелки обозначают сообщения, обмен которыми осуществляется в рамках данного варианта использования. Их временнáя последовательность, однако, указывается путем нумерации сообщений

**2.Паттерн Декоратор**

Шаблон Декоратор предназначен для динамического добавления к объекту новой функциональности. Он является гибкой альтернативой наследованию (в том числе множественному). Декоратор служит обёрткой для декорируемого объекта и реализует тот же интерфейс, что и класс декорируемого объекта. Однако декоратор добавляет свой код до, после или вместо вызовов методов декорируемого объекта. Это приводит к модификации исходного поведения и появлению новых возможностей

IComponent определяет общие функции исходных компонентов и декораторов1 . Класс ConcreteComponent – это компонент, функциональность которого необходимо модифицировать. Класс Decorator – базовый декоратор (как правило, он включает механизм хранения компонента и реализует простую переадресацию). Конкретные декораторы ConcreteDecoratorA и ConcreteDecoratorB наследуются от базового декоратора, добавляя определённое состояние и (или) поведение

#include <iostream>

using namespace std;

class Component {

public:

virtual ~Component() {}

virtual string Operation() const = 0;

};

/\*\*

\* Конкретные Компоненты предоставляют реализации поведения по умолчанию. Может

\* быть несколько вариаций этих классов.

\*/

class ConcreteComponent : public Component {

public:

string Operation() const override {

return "ConcreteComponent";

}

};

/\*\*

\* Базовый класс Декоратора следует тому же интерфейсу, что и другие компоненты.

\* Основная цель этого класса - определить интерфейс обёртки для всех конкретных

\* декораторов. Реализация кода обёртки по умолчанию может включать в себя поле

\* для хранения завёрнутого компонента и средства его инициализации.

\*/

class Decorator : public Component {

protected:

Component\* component\_;

public:

Decorator(Component\* component) : component\_(component) {

}

/\*\*

\* Декоратор делегирует всю работу обёрнутому компоненту.

\*/

string Operation() const override {

return this->component\_->Operation();

}

};

/\*\*

\* Конкретные Декораторы вызывают обёрнутый объект и изменяют его результат

\* некоторым образом.

\*/

class ConcreteDecoratorA : public Decorator {

/\*\*

\* Декораторы могут вызывать родительскую реализацию операции, вместо того,

\* чтобы вызвать обёрнутый объект напрямую. Такой подход упрощает расширение

\* классов декораторов.

\*/

public:

ConcreteDecoratorA(Component\* component) : Decorator(component) {

}

string Operation() const override {

return "ConcreteDecoratorA(" + Decorator::Operation() + ")";

}

};

/\*\*

\* Декораторы могут выполнять своё поведение до или после вызова обёрнутого

\* объекта.

\*/

class ConcreteDecoratorB : public Decorator {

public:

ConcreteDecoratorB(Component\* component) : Decorator(component) {

}

string Operation() const override {

return "ConcreteDecoratorB(" + Decorator::Operation() + ")";

}

};

/\*\*

\* Клиентский код работает со всеми объектами, используя интерфейс Компонента.

\* Таким образом, он остаётся независимым от конкретных классов компонентов, с

\* которыми работает.

\*/

void ClientCode(Component\* component) {

// ...

cout << "RESULT: " << component->Operation();

// ...

}

int main() {

/\*\*

\* Таким образом, клиентский код может поддерживать как простые компоненты...

\*/

Component\* simple = new ConcreteComponent;

cout << "Client: I've got a simple component:\n";

ClientCode(simple);

cout << "\n\n";

/\*\*

\* ...так и декорированные.

\*

\* Обратите внимание, что декораторы могут обёртывать не только простые

\* компоненты, но и другие декораторы.

\*/

Component\* decorator1 = new ConcreteDecoratorA(simple);

Component\* decorator2 = new ConcreteDecoratorB(decorator1);

cout << "Client: Now I've got a decorated component:\n";

ClientCode(decorator2);

cout << "\n";

delete simple;

delete decorator1;

delete decorator2;

return 0;

}

**3.Компоненты COM**

COM – это

1)Выполняемая программная единица

2)Двоичный файл, загружаемый в память как dll

3)Составной объект, содержащий интерфейс и тело.

Интерфейс «отделен» от тела, так что COM можно по замыслу создателей использовать в других языках. Т.е. создавать, например, в С++, а использовать в С# или Visual Basic.

COM должен наследовать интерфейс IUnknown. C помощью этого средства можно узнать, какие еще есть интерфейсы в COM

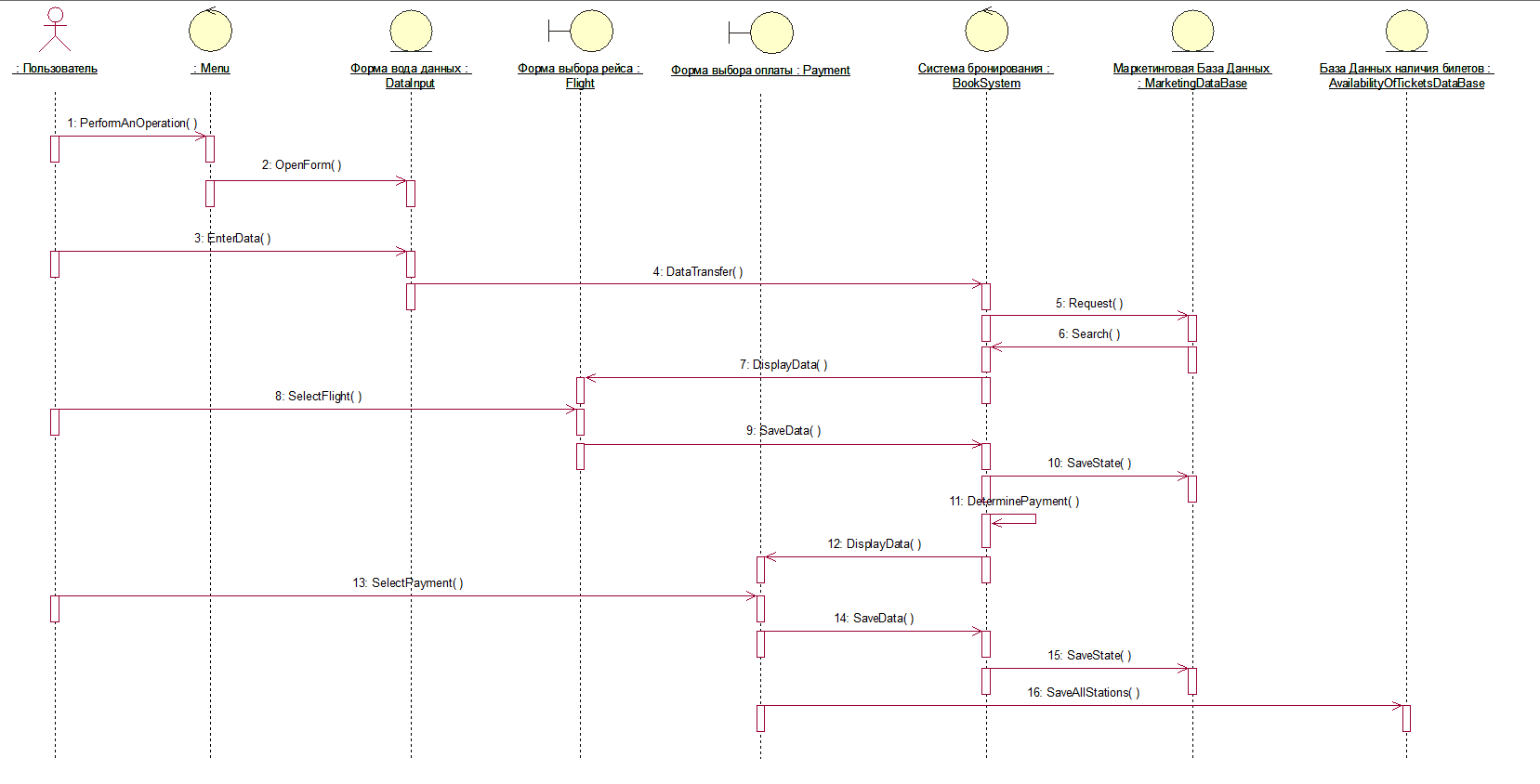
интерфейс – это вид абстрактного класса, где методы объявлены, но не реализованы. COM (интерфейс COM) должен иметь уникальный системный номер. Такие номера называются GUID (global universal identifier). GUID присваивается специальной утилитой

тело COM – т.е. класс, где реализован интерфейс COM

Метод QueryInterface возвращает нам ссылку на объект интерфейса.. Аргументами являются идентификатор интерфейса – большой номер, называемый GUID и указатель на объект интерфейса

# Билет 9

1. **Диаграммы последовательности действий**



**Диаграммы последовательности** отражают поток событий, происходящих в рамках варианта использования. Все действующие лица показаны в верхней части диаграммы

Стрелки соответствуют сообщениям, передаваемым между действующим лицом и объектом или между объектами для выполнения требуемых функций

На диаграмме последовательности объект изображается в виде прямоугольника, от которого вниз проведена пунктирная вертикальная линия. Эта линия называется линией жизни (lifeline) объекта. Она представляет собой фрагмент жизненного цикла объекта в процессе взаимодействия

Каждое сообщение представляется в виде стрелки между линиями жизни двух объектов. Сообщения появляются в том порядке, как они показаны на странице сверху вниз. Каждое сообщение помечается как минимум именем сообщения; при желании можно добавить также аргументы и некоторую управляющую информацию, и, кроме того, можно показать само-делегирование (self-delegation) – сообщение, которое объект посылает самому себе, при этом стрелка сообщения указывает на ту же самую линию жизни

1. **Паттерн Мост**

Шаблон Мост позволяет разделить объект на абстракцию и реализацию так, чтобы они могли изменяться независимо друг от друга. При этом основной код, необходимый для функционирования объекта, переносится в реализацию. Всё остальное, включая взаимодействие с клиентом, содержится в абстракции. Методы абстракции при необходимости могут быть изменены или дополнены. Абстракция содержит экземпляр реализации и использует его для обработки поступающих от клиента запросов.

Выделим следующих участников шаблона Мост (рис. 5). Класс Abstraction определяет базовую абстракцию. Класс ExtendedAbstraction наследуется от Abstraction и представляет уточнённый функционал взаимодействия. Implementation является интерфейсом реализации. Конкретная реализация описывается классом Platform

#include <iostream>

using namespace std;

class Implementation {

public:

virtual ~Implementation() {}

virtual string OperationImplementation() const = 0;

};

/\*\*

\* Каждая Конкретная Реализация соответствует определённой платформе и реализует

\* интерфейс Реализации с использованием API этой платформы.

\*/

class PlatformA : public Implementation {

public:

string OperationImplementation() const override {

return "PlatformA: Train arrived on the platform A.\n";

}

};

class PlatformB : public Implementation {

public:

string OperationImplementation() const override {

return "PlatformB: Train arrived on the platform B.\n";

}

};

/\*\*

\* Абстракция устанавливает интерфейс для «управляющей» части двух иерархий

\* классов. Она содержит ссылку на объект из иерархии Реализации и делегирует

\* ему всю настоящую работу.

\*/

class Abstraction {

protected:

Implementation\* implementation\_;

public:

Abstraction(Implementation\* implementation) : implementation\_(implementation) {

}

virtual ~Abstraction() {

}

virtual string Operation() const {

return "Base operation with:\n" +

this->implementation\_->OperationImplementation();

}

};

/\*\*

\* Можно расширить Абстракцию без изменения классов Реализации.

\*/

class ExtendedAbstraction : public Abstraction {

public:

ExtendedAbstraction(Implementation\* implementation) : Abstraction(implementation) {

}

string Operation() const override {

return "Extended operation with:\n" +

this->implementation\_->OperationImplementation();

}

};

/\*\*

\* За исключением этапа инициализации, когда объект Абстракции связывается с

\* определённым объектом Реализации, клиентский код должен зависеть только от

\* класса Абстракции. Таким образом, клиентский код может поддерживать любую

\* комбинацию абстракции и реализации.

\*/

void ClientCode(const Abstraction& abstraction) {

// ...

cout << abstraction.Operation();

// ...

}

/\*\*

\* Клиентский код должен работать с любой предварительно сконфигурированной

\* комбинацией абстракции и реализации.

\*/

int main() {

Implementation\* implementation = new PlatformA;

Abstraction\* abstraction = new Abstraction(implementation);

ClientCode(\*abstraction);

cout << endl;

delete implementation;

delete abstraction;

implementation = new PlatformB;

abstraction = new ExtendedAbstraction(implementation);

ClientCode(\*abstraction);

delete implementation;

delete abstraction;

return 0;

}

1. **Образование функций на основе Лямбда-выражений. Примеры**

***Лямбда-выражение*** – способ определения анонимной функции непосредственно в месте вызова.

Лямбда-выражения имеют несколько вариантов синтаксиса, например:

[*список\_захвата*] (*список\_параметров*) -> *возвращаемый\_тип* {*тело\_выражения*}

Список захвата определяет, к каким переменным из внешнего окружения получает доступ лямбда-выражение.

[] – доступ к переменным из внешней области видимости запрещен;

[=] – доступ ко всем внешним переменным осуществляется по значению;

[&] – доступ ко всем внешним переменным осуществляется по ссылке.

Доступ к переменным с префиксом с амперсандом (&) осуществляется по ссылке, а к переменным без префикса — по значению.

Элементы списка отделяются друг от друга запятыми.

Пример списков захвата:

[x, &y] – x захвачено по значению, y – по ссылке;

[&, x] – x захвачено по значению, остальные по ссылке;

[=, \*this] – захват переменных и объекта по значению.

#include <iostream>

using namespace std;

int main(){

double x = 7, y = 3;

double s = 0;

[=, &s]() {return s = x / y; }();

cout << s << endl; // Выводит: 2.33333

[&](double x, double y) {return s = x / y; }(15, 3);

cout << s << endl; // Выводит: 5

auto w = [=, &s](double x) {return x / y; };

cout << w(9) << endl; // Выводит: 3

cout << w(14) << endl; // Выводит: 4.66667

}

# Билет 10

**1.** В результате развития концепций БД выделены три уровня представления информации: инфологический, даталогический и физический. На каждом уровне проводится структуризация информации таким образом, чтобы на третьем уровне информация могла быть представлена в виде структур данных, реализуемых в памяти ЭВМ. На инфологическом уровне определяется какая информация о предметной области будет хранится и обрабатываться в компьютере, а в результате исследования предметной области строится ее инфологическая модель, которая описывается в терминах классов объектов и их взаимодействий. В инфологической модели информация представляется вне зависимости от того, что представляют собой данные и какие технические средства будут использовании в дальнейшем для ее хранения и обработки. Даталогическая и физическая модели непосредственно реализуются в системах управления БД (СУБД), а физическая модель в свою очередь определяет структуру хранения данных на физических носителях. Цель инфологического проектирования заключается в представлении семантики (смысла) предметной области.\*/ Для описания предметной области наиболее часто используется модель “сущность–связь”, которую сокращенно называют ER–моделью от английского названия “Entity – Relationship” (“Сущность – связь”). ER–диаграмма модели имеет лексикографическую структуру и включает в себя текст и элементы графики

ER–модель опирается на понятия сущность, атрибут, связь и предметная область должна быть представлена как совокупность сущностей с атрибутами, между которыми установлены связи. С объектами ER–модели связаны понятия: тип – набор однородных предметов, явлений, которые выступают как единое целое; экземпляр – конкретный элемент набора, который (набор) означает некоторый тип; множество – конкретный набор экземпляров типа. Сущность – это объект, который может быть идентифицирован некоторым способом, который отличает его от других объектов. Тип сущности означает набор однородных сущностей некоторого типа. Набор сущностей – множество сущностей одного типа. Сущность фактически является множеством атрибутов, которые описывают свойства всех членов данного набора сущности. Тип сущности означает множество однородных объектов, а “экземпляр сущности” – конкретный объект из множества объектов .Связь – это поименованное отношение, имеющее место между двумя сущностями

На сегодняшний день существуют несколько нотаций для представления ER– диаграмм:

1. **Нотация Чена**: сущность изображается прямоугольником, атрибут – овалом, соединенным со своей сущностью (идентифицирующий атрибут подчеркнут), а связь – ромбом, соединенным со связываемыми сущностями. Вид линии в месте соединения с сущностью определяет кардинальность связи («воронья лапка» – М, «крест» – 1). Имена сущности, атрибута и связи располагаются внутри их изображений

2. **Нотация Мартина**: сущность изображается прямоугольником, внутри которого указано ее имя жирным шрифтом и список ее атрибутов (идентифицирующий атрибут подчеркнут), а связь – линией, название которой располагается над ней и ее вид в месте соединения с сущностью определяет кардинальность связи («воронья лапка» – М, «крест»– 1)

3. **Нотация Баркера**: сущность изображается прямоугольником, внутри которого указано ее имя жирным и список ее атрибутов (перед идентифицирующим атрибутом стоит #), а связь – линией, название которой располагается над ней и ее вид в месте соединения с сущностью определяет кардинальность связи («воронья лапка» – М, отсутствие – 1)

4. **Нотация IDEFIX**: сущность изображается прямоугольником, атрибут – овалом, соединенным со своей сущностью, а связь – ромбом, соединенным со связываемыми сущностями. Имена сущности, атрибута и связи располагаются внутри их изображений

5. **Нотация Бахмана**: сущность изображается таблицей из одного столбца, столбцы которой являются атрибутами сущности (идентифицирующий атрибут выделен жирным шрифтом), а связь – стрелкой, соединяющей таблицы, направление которой указано на стороне М

**2.Паттерн Посредник**

Шаблон Посредник служит для обеспечения коммуникации между объектами. Этот шаблон также инкапсулирует протокол, которому должна удовлетворять процедура коммуникации

Дизайн шаблона Посредник предполагает наличие двух выделенных классов, использующих сообщения для взаимного обмена информацией: BaseComponent и Mediator (посредник). Объект BaseComponent регистрирует объект Mediator и сохраняет его в своём внутреннем поле. В свою очередь, посредник поддерживает список зарегистрировавших его объектов. Как только один из объектов BaseComponent вызывает свой метод , посредник вызывает метод у остальных зарегистрированных объектов

#include <iostream>

#include <string>

using namespace std;

/\*\*

\* Интерфейс Посредника предоставляет метод, используемый компонентами для

\* уведомления посредника о различных событиях. Посредник может реагировать на

\* эти события и передавать исполнение другим компонентам.

\*/

class BaseComponent;

class Mediator {

public:

virtual void Notify(BaseComponent\* sender, string event) const = 0;

};

/\*\*

\* Базовый Компонент обеспечивает базовую функциональность хранения экземпляра

\* посредника внутри объектов компонентов.

\*/

class BaseComponent {

protected:

Mediator\* mediator\_;

public:

BaseComponent(Mediator\* mediator = nullptr) : mediator\_(mediator) {

}

void set\_mediator(Mediator\* mediator) {

this->mediator\_ = mediator;

}

};

/\*\*

\* Конкретные Компоненты реализуют различную функциональность. Они не зависят от

\* других компонентов. Они также не зависят от каких-либо конкретных классов

\* посредников.

\*/

class Component1 : public BaseComponent {

public:

void DoA() {

cout << "Component 1 does A.\n";

this->mediator\_->Notify(this, "A");

}

void DoB() {

cout << "Component 1 does B.\n";

this->mediator\_->Notify(this, "B");

}

};

class Component2 : public BaseComponent {

public:

void DoC() {

cout << "Component 2 does C.\n";

this->mediator\_->Notify(this, "C");

}

void DoD() {

cout << "Component 2 does D.\n";

this->mediator\_->Notify(this, "D");

}

};

/\*\*

\* Конкретные Посредники реализуют совместное поведение, координируя отдельные

\* компоненты.

\*/

class ConcreteMediator : public Mediator {

private:

Component1\* component1\_;

Component2\* component2\_;

public:

ConcreteMediator(Component1\* c1, Component2\* c2) : component1\_(c1), component2\_(c2) {

this->component1\_->set\_mediator(this);

this->component2\_->set\_mediator(this);

}

void Notify(BaseComponent\* sender, string event) const override {

if (event == "A") {

cout << "Mediator reacts on A and triggers following operations:\n";

this->component2\_->DoC();

}

if (event == "D") {

cout << "Mediator reacts on D and triggers following operations:\n";

this->component1\_->DoB();

this->component2\_->DoC();

}

}

};

void ClientCode() {

Component1\* c1 = new Component1;

Component2\* c2 = new Component2;

ConcreteMediator\* mediator = new ConcreteMediator(c1, c2);

cout << "Client triggers operation A.\n";

c1->DoA();

cout << "\n";

cout << "Client triggers operation D.\n";

c2->DoD();

delete c1;

delete c2;

delete mediator;

}

int main() {

ClientCode();

return 0;

}

**3. Использование указателей на функции, классы и методы классов**

Мы можем легко получить **адрес функции**. Нам просто нужно упомянуть имя функции, нам не нужно вызывать функцию.

**Пример.**

#include <iostream>

using namespace std;

int main()

{

std::cout << "Address of a main() function is : " <<&main<< std::endl;

  return 0;

}

Мы можем вызвать функцию с помощью указателя функции, просто используя имя указателя функции. Синтаксис вызова функции через указатель функции будет аналогичен обычному вызову функции

**Пример.**

#include <iostream>

using namespace std;

int add(int a , int b)

{

    return a+b;

}

int main()

{

 int (\*funcptr)(int,int);  // function pointer declaration

 funcptr=add; // funcptr is pointing to the add function

 int sum=funcptr(5,5);

 std::cout << "value of sum is :" <<sum<< std::endl;

  return 0;

}

**Указатели на структуры C++**

Как и другие указатели, указатели структур объявляются путем размещения звездочки (∗) перед именем указателя структуры.

Он принимает следующий общий вид:

struct-name ∗struct-pointer ;

**Обращение к членам структуры**

struct-pointer -> structure-member

Указатель на класс C++ делается точно так же, как указатель на структуру, и для доступа к членам указателя на класс вы используете оператор доступа к члену -> оператор, точно так же, как и с указателями на структуры. Также, как и в случае со всеми указателями, вы должны инициализировать указатель перед его использованием

**Указатели на поля и методы класса**

#include <iostream>

using namespace std;

class X {

public:

int a;

void f(int b) {

cout << "The value of b is "<< b << endl;

}

};

int main() {

// declare pointer to data member

int X::\*ptiptr = &X::a;

// declare a pointer to member function

void (X::\* ptfptr) (int) = &X::f;

// create an object of class type X

X xobject;

// initialize data member

xobject.\*ptiptr = 10;

cout << "The value of a is " << xobject.\*ptiptr << endl;

// call member function

(xobject.\*ptfptr) (20);

}

Результат:

The value of a is 10

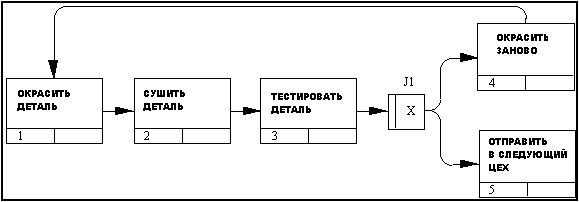
The value of b is 20

# Билет 11

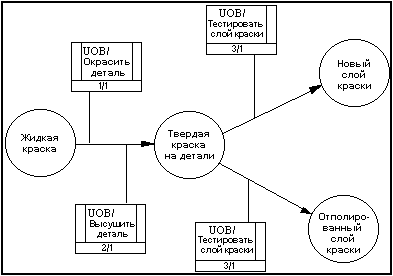
**1. Диаграммы IDEF3**

IDEF3 является стандартом документирования технологических процессов, происходящих на предприятии, и предоставляет инструментарий для наглядного исследования и моделирования их сценариев. **Сценарием (Scenario)** мы называем описание последовательности изменений свойств объекта, в рамках рассматриваемого процесса (например, описание последовательности этапов обработки детали в цеху и изменение её свойств после прохождения каждого этапа). Исполнение каждого сценария сопровождается соответствующим документооборотом, который состоит из двух основных потоков: документов, определяющих структуру и последовательность процесса (технологических указаний, описаний стандартов и т.д.), и документов, отображающих ход его выполнения (результатов тестов и экспертиз, отчетов о браке, и т.д.

Существуют два типа диаграмм в стандарте IDEF3, представляющие описание одного и того же сценария технологического процесса в разных ракурсах. Диаграммы относящиеся к первому типу называются диаграммами Описания Последовательности Этапов Процесса (Process Flow Description Diagrams, PFDD), а ко второму - диаграммами Состояния Объекта в и его Трансформаций Процессе (Object State Transition Network, OSTN). Предположим, требуется описать процесс окраски детали в производственном цеху на предприятии. С помощью диаграмм PFDD документируется последовательность и описание стадий обработки детали в рамках исследуемого технологического процесса.



Диаграммы OSTN используются для иллюстрации трансформаций детали, которые происходят на каждой стадии обработки



**2.Паттерн Прокси**

Шаблон Заместитель позволяет контролировать доступ к заданному объекту, перехватывая все вызовы к этому объекту и прозрачно замещая его. Ни интерфейс, ни функциональность замещённого объекта с точки зрения клиента не меняются. Данный шаблон часто используется, если необходимо упростить или оптимизировать взаимодействие с объектом, скрывая несущественные для конкретной задачи подробности реализации

Перечислим некоторые разновидности шаблона Заместитель: Удалённый заместитель (remote proxy) обеспечивает связь с замещаемым объектом, который находится в другом адресном пространстве или на удалённой машине; – Виртуальный заместитель (virtual proxy) реализует создание замещаемого объекта только тогда, когда он действительно необходим; – Защищающий заместитель (protection proxy) проверяет, имеет ли вызывающий объект необходимые для выполнения запроса права

#include <iostream>

using namespace std;

/\*\*

\* Интерфейс Субъекта объявляет общие операции как для Реального Субъекта, так и

\* для Заместителя. Пока клиент работает с Реальным Субъектом, используя этот

\* интерфейс, вы сможете передать ему заместителя вместо реального субъекта.

\*/

class Subject {

public:

virtual void Request() const = 0;

};

/\*\*

\* Реальный Субъект содержит некоторую базовую бизнес-логику. Как правило,

\* Реальные Субъекты способны выполнять некоторую полезную работу, которая к

\* тому же может быть очень медленной или точной – например, коррекция входных

\* данных. Заместитель может решить эти задачи без каких-либо изменений в коде

\* Реального Субъекта.

\*/

class RealSubject : public Subject {

public:

void Request() const override {

cout << "RealSubject: Handling request.\n";

}

};

/\*\*

\* Интерфейс Заместителя идентичен интерфейсу Реального Субъекта.

\*/

class Proxy : public Subject {

private:

RealSubject\* real\_subject\_;

bool CheckAccess() const {

// Некоторые реальные проверки должны проходить здесь.

cout << "Proxy: Checking access prior to firing a real request.\n";

return true;

}

void LogAccess() const {

cout << "Proxy: Logging the time of request.\n";

}

/\*\*

\* Заместитель хранит ссылку на объект класса РеальныйСубъект. Клиент может

\* либо лениво загрузить его, либо передать Заместителю.

\*/

public:

Proxy(RealSubject\* real\_subject) : real\_subject\_(new RealSubject(\*real\_subject)) {

}

~Proxy() {

delete real\_subject\_;

}

/\*\*

\* Наиболее распространёнными областями применения паттерна Заместитель

\* являются ленивая загрузка, кэширование, контроль доступа, ведение журнала и

\* т.д. Заместитель может выполнить одну из этих задач, а затем, в зависимости

\* от результата, передать выполнение одноимённому методу в связанном объекте

\* класса Реального Субъект.

\*/

void Request() const override {

if (this->CheckAccess()) {

this->real\_subject\_->Request();

this->LogAccess();

}

}

};

/\*\*

\* Клиентский код должен работать со всеми объектами (как с реальными, так и

\* заместителями) через интерфейс Субъекта, чтобы поддерживать как реальные

\* субъекты, так и заместителей. В реальной жизни, однако, клиенты в основном

\* работают с реальными субъектами напрямую. В этом случае, для более простой

\* реализации паттерна, можно расширить заместителя из класса реального

\* субъекта.

\*/

void ClientCode(const Subject& subject) {

// ...

subject.Request();

// ...

}

int main() {

cout << "Client: Executing the client code with a real subject:\n";

RealSubject\* real\_subject = new RealSubject;

ClientCode(\*real\_subject);

cout << "\n";

cout << "Client: Executing the same client code with a proxy:\n";

Proxy\* proxy = new Proxy(real\_subject);

ClientCode(\*proxy);

delete real\_subject;

delete proxy;

return 0;

}

**3.Создание процессов в API Windows**

Функция **[CreateProcess](https://docs.microsoft.com/ru-RU/windows/win32/api/processthreadsapi/nf-processthreadsapi-createprocessa)** создает новый процесс, который выполняется независимо от процесса создания

**BOOL CreateProcess(**

**LPCTSTR** *lpApplicationName* **,** // имя исполняемого модуля

**LPTSTR** *lpCommandLine***,** // командная строка

**LPSECURITY\_ATTRIBUTES** *lpProcessAttributes* **,** // SD (дескриптор безопасности)

**LPSECURITY\_ATTRIBUTES** *lpThreadAttributes***,** // SD

**BOOL** *bInheritHandles***,** // дескриптор параметра наследования

**DWORD** *dwCreationFlags***,** // флажки создания

**LPVOID** *lpEnvironment***,** // новый блок конфигурации

**LPCTSTR** *lpCurrentDirectory***,** // имя текущего каталога

**LPSTARTUPINFO** *lpStartupInfo***,** // информация предустановки

**LPPROCESS\_INFORMATION** *lpProcessInformation* // информация о процессе

**);**

Возвращаемое значение: в случае успешного создания процесса и потока — TRUE, иначе — FALSE.

Параметры

lpApplicationName и lpCommandLine (последний указатель имеет тип LPTSTR, а не LPCTSTR) — используются вместе для указания исполняемой программы и аргументов командной строки, о чем говорится в следующем разделе.

lpsaProcess и lpsaThread — указатели на структуры атрибутов защиты процесса и потока. Значениям NULL соответствует использование атрибутов защиты, заданных по умолчанию, и именно эти значения будут использоваться нами вплоть до главы 15, посвященной рассмотрению средств безопасности Windows.

bInheritHandles — показывает, наследует ли новый процесс наследуемые открытые дескрипторы (файлов, отображений файлов и так далее) из вызывающего процесса. Наследуемые дескрипторы имеют те же атрибуты, что и исходные, и их обсуждение будет продолжено в одном из следующих разделов.

dwCreationFlags — может объединять в себе несколько флаговых значений, включая следующие:

• CREATE\_SUSPENDED — указывает на то, что основной поток будет создан в приостановленном состоянии и начнет выполняться лишь после вызова функция ResumeThread.

• DETACHED\_PROCESS и CREATE\_NEW\_CONSOLE — взаимоисключающие значения, которые не должны устанавливаться оба одновременно. Первый флаг означает создание нового процесса, у которого консоль отсутствует, а второй — процесса, у которого имеется собственная консоль. Если ни один из этих флагов не указан, то новый процесс наследует консоль родительского процесса.

• Create\_New\_Process\_Group — указывает на то, что создаваемый процесс является корневым для новой группы процессов. Если все процессы, принадлежащие данной группе, разделяют общую консоль, то все они будут получать управляющие сигналы консоли (Ctrl-C или Ctrl-break)

lpEnvironment — указывает на блок параметров настройки окружения нового процесса. Если задано значение NULL, то новый процесс будет использовать значения параметров окружения родительского процесса. Блок параметров содержит строки, в которых заданы пары "имя-значение", определяющие, например, пути доступа к файлам.

lpCurDir — указатель на строку, содержащую путь к текущему каталогу нового процесса. Если задано значение NULL, то в качестве текущего каталога будет использоваться рабочий каталог родительского процесса.

lpStartupInfo — указатель на структуру, которая описывает внешний вид основного окна и содержит дескрипторы стандартных устройств нового процесса.

lpProInfо — указатель на структуру, в которую будут помещены возвращаемые функцией значения дескрипторов и глобальных идентификаторов процесса и потока

# Билет 12

1. **1. Функциональное моделирование IDEF0**

В основе *IDEF0*-методологии лежат **4 основных понятия**:

1) функциональный блок;

2) интерфейсная дуга (стрелка);

3) декомпозиция;

4) глоссарий.

**Функциональный блок**

Олицетворяет некоторую конкретную функцию или работу в рамках рассматриваемой системы

РД IDEF0 – 2000: прямоугольник, содержащий имя и номер и используемый для описания функции

Наименование осуществляется оборотом глагола или существительного

Каждый блок в рамках единой системы имеет уникальный номер

Каждая сторона функционального блока имеет свое назначение

Управлять предприятием

А0

управление

вход

выход

механизм

**Интерфейсная дуга**

Интерфейсная дуга отображает элемент системы, который обрабатывается функциональным блоком или оказывает иное влияние на функцию, отображаемую функциональным блоком.

Графически изображается в виде однонаправленной стрелки.

Каждая дуга должна иметь свое уникальное название, сформулированное оборотом существительного (должно отвечать на вопросы кто?, что?). Примеры: информация, разработчик, документ, обработанная заявка.

В зависимости от того, к какой стороне блока она подходит, интерфейсная дуга будет являться входящей, выходящей, управления, механизма

**Стрелки выхода и управления – обязательны**

**Декомпозиция**

Принцип декомпозиции применяется при разбиении сложных процессов на составляющие его функции. При этом уровень детализации определяется непосредственно разработчиком модели.

Модель IDEF0 всегда начинается с рассмотрения системы как единого целого, т.е. одного функционального блока с интерфейсными дугами, простирающимися за пределы рассматриваемой области. Такая диаграмма называется *контекстной*, она обозначается идентификатором А-0.

Для определения границ системы на контекстной диаграмме обязательно должны быть цель и точка зрения.

**Цель моделирования** должна отвечать на следующие вопросы:

Почему процесс должен быть замоделирован?

Что должна показывать модель?

Что может получить читатель?

**Точка зрения** – позиция, с которой будет строиться модель. В качестве точки зрения берется взгляд человека, который видит систему в нужном для моделирования аспекте.

Как правило, выбирается точка зрения человека, ответственного за выполнение моделируемой работы.

**Между целью и точкой зрения должно быть жесткое соответствие.**

**Глоссарий**

Для каждого из элементов в IDEF0 существует стандарт, подразумевающий создание и поддержку набора соответствующих определений, ключевых слов, повествований, изложений и т.д, которые характеризуют объект, отраженный данным элементом. Этот набор – **глоссарий**, являющийся описанием сущности данного элемента.

**FEO-диаграмма** (*For Exposition Only*) – это диаграмма, которая поясняет особо интересные и тонкие аспекты диаграмм. Эти диаграммы не ограничены синтаксисом IDEF0. В них может быть текстовая, графическая информация, схемы, альтернативная точка зрения на процесс и т.п

Формат номера блока:

1.Префикс

2.Номер родительской работы

3. Собственный порядковый номер

**Основные правила построения диаграмм**

1. На одной диаграмме рекомендуется рисовать от 3 до 6 блоков. Иначе диаграмма будет плохо читаемой.

2. Функциональные блоки должны располагаться слева направо сверху вниз в порядке доминирования.

3. Следует избегать излишнего пересечения стрелок.

4. Выход одного блока может являться входом (управлением) для другого. Могут быть и обратные связи по входу и управлению.

5. Стрелки могут быть сливающимися и разветвляющимися

**Обратная связь по входу**, как правило, используется для описания циклов.

**Обратная связь по управлению** – выход нижестоящей работы передается на управление вышестоящей

**Обратная связь по механизму** – выход нижестоящей работы создает ресурсы, выполняющие вышестоящую работу

**2.Паттерн Стратегия**

При помощи шаблона Стратегия из клиента выделяется алгоритм, который затем инкапсулируется в типах, наследуемых от общего класса (реализующих общий интерфейс). Это позволяет клиенту выбирать нужный алгоритм путём создания объектов необходимых типов. Кроме этого, шаблон допускает изменение набора доступных алгоритмов со временем

#include<iostream>

#include<stdio.h>

#include<math.h>

#include<vector>

#include <string>

using namespace std;

// Иерархия классов, определяющая алгоритмы сжатия файлов

class Compression

{

public:

virtual ~Compression() {}

virtual void compress(const string& file) = 0;

};

class ZIP\_Compression : public Compression

{

public:

void compress(const string& file) {

cout << "ZIP compression" << endl;

}

};

class ARJ\_Compression : public Compression

{

public:

void compress(const string& file) {

cout << "ARJ compression" << endl;

}

};

class RAR\_Compression : public Compression

{

public:

void compress(const string& file) {

cout << "RAR compression" << endl;

}

};

// Класс для использования

class Compressor

{

public:

Compressor(Compression\* comp) : p(comp) {}

~Compressor() { delete p; }

void compress(const string& file) {

p->compress(file);

}

private:

Compression\* p;

};

int main()

{

Compressor\* p = new Compressor(new ZIP\_Compression);

p->compress("file.txt");

delete p;

Compressor\* t = new Compressor(new RAR\_Compression);

t->compress("file.txt");

delete t;

Compressor\* z = new Compressor(new ARJ\_Compression);

z->compress("file.txt");

delete z;

return 0;

}

**3. Динамически создаваемые объекты без объявлений классов (на примере с#)**

# Билет 14

1. **Модель RDF**

RDF — это универсальный способ разложения любых знаний на маленькие кусочки. Он задаёт определённые правила касательно семантики, т.е. смысла этих кусочков. Идея состоит в том, чтобы одним простым способом можно было бы описать любой факт, притом в таком структурированном виде, чтобы его могли обрабатывать компьютерные программы. RDF — это не просто формат записи; это формат представления информации, организованной в виде графов или сетей.

То, в чём [RDF](http://xmlhack.ru/) сильнее всего отличается от XML и других технологий — это то, что RDF предназначен для представления знаний в распределённом мире. Другими словами, для RDF особенно важен смысл. Всё, с чем работает RDF, имеет определённый смысл, — ссылается на какой-то конкретный объект, или на абстрактное понятие, или на некий факт. Стандарты, основанные на RDF, описывают логические выводы, связывающие эти факты, и указывают, как можно найти сами факты в огромной базе данных всех знаний, представленных в RDF.

В RDF предусматривается два процесса, благодаря которым это осуществляется, — во-первых, объединяются документы, использующие общие языки, и во-вторых, допускается использовать любые языки в каждом из документов. Эта гибкость — одна из отличительных черт RDF

RDF можно определить как совокупность трёх простых правил:

1.Факт выражается тройкой вида (Подлежащее, Сказуемое, Дополнение) — похожей на простое предложение на естественном языке.

2.Подлежащие, сказуемые и дополнения — это имена сущностей реального мира, конкретных или абстрактных. Имя может быть 1) глобальным, ссылающимся на одну и ту же сущность во всех RDF-документах, где оно используется, либо 2) локальным, и тогда на сущность, на которую ссылается это имя, нельзя ссылаться из-за пределов RDF-документа непосредственно.

3.Дополнения, кроме этого, могут быть текстовыми строками — «литералами».

Имена бывают двух типов. Глобальные имена, имеющие всюду один и тот же смысл, всегда имеют вид URI (Универсальных Идентификаторов Ресурсов). Синтаксис (формат) URI может быть таким же, как у адреса веб-сайта. Так как URI могут быть довольно длинными, то в форматах, используемых для представления RDF, они обычно сокращаются, используя перенятый из XML механизм «пространств имён». Именно поэтому в именах :john, :hasMother и других сущностей в приведённых примерах стоят двоеточия — они означают, что используются сокращённые имена

Есть два взаимодополняющих способа рассматривать информацию, представленную в RDF. Первый способ — считать её набором утверждений, как в примерах выше: каждое утверждение представляет собой факт. Второй способ — считать её графом. В RDF узлы — это имена (но не сами сущности), а рёбра — утверждения.

**2.Паттерн абстрактная фабрика классов**

Шаблон Абстрактная фабрика предназначен для создания объектов, принадлежащих одному набору классов и используемых совместно. Абстрактная фабрика переопределяется в конкретных классах-фабриках, создающих объекты одного набора. Этот шаблон изолирует имена классов и их определения от клиента: единственный способ для клиента получить необходимый объект – это воспользоваться одной из реализаций абстрактной фабрики.

Клиент работает с определённым набором объектов, но использует для этого только реализуемые классами объектов интерфейсы (Infantryman,Archer,Horseman). Клиент хранит ссылку на конкретную фабрику, реализующую интерфейс ArmyFactory. Для получения нужного объекта клиент вызывает один из методов фабрики

#include <iostream>

#include <stdio.h>

#include <string>

#include<vector>

using namespace std;

//интерфейсы

class Infantryman

{

public:

virtual void info() = 0;

virtual ~Infantryman() {}

};

class Archer

{

public:

virtual void info() = 0;

virtual ~Archer() {}

};

class Horseman

{

public:

virtual void info() = 0;

virtual ~Horseman() {}

};

// Классы всех видов воинов Римской армии – наборы классов, реализующих интерфейсы

class RomanInfantryman : public Infantryman

{

public:

void info() {

cout << "RomanInfantryman" << endl;

}

};

class RomanArcher : public Archer

{

public:

void info() {

cout << "RomanArcher" << endl;

}

};

class RomanHorseman : public Horseman

{

public:

void info() {

cout << "RomanHorseman" << endl;

}

};

// Классы всех видов воинов армии Карфагена - аналогично

class CarthaginianInfantryman : public Infantryman

{

public:

void info() {

cout << "CarthaginianInfantryman" << endl;

}

};

class CarthaginianArcher : public Archer

{

public:

void info() {

cout << "CarthaginianArcher" << endl;

}

};

class CarthaginianHorseman : public Horseman

{

public:

void info() {

cout << "CarthaginianHorseman" << endl;

}

};

// Абстрактная фабрика для производства воинов- интерфейс

class ArmyFactory

{

public:

virtual Infantryman\* createInfantryman() = 0;

virtual Archer\* createArcher() = 0;

virtual Horseman\* createHorseman() = 0;

virtual ~ArmyFactory() {}

};

// Фабрика для создания воинов Римской армии – конкретные фабрики

class RomanArmyFactory : public ArmyFactory

{

public:

Infantryman\* createInfantryman() {

return new RomanInfantryman;

}

Archer\* createArcher() {

return new RomanArcher;

}

Horseman\* createHorseman() {

return new RomanHorseman;

}

};

// Фабрика для создания воинов армии Карфагена

class CarthaginianArmyFactory : public ArmyFactory

{

public:

Infantryman\* createInfantryman() {

return new CarthaginianInfantryman;

}

Archer\* createArcher() {

return new CarthaginianArcher;

}

Horseman\* createHorseman() {

return new CarthaginianHorseman;

}

};

// Класс, содержащий всех воинов той или иной армии

class Army

{

public:

~Army() {

int i;

for (i = 0; i < vi.size(); ++i) delete vi[i];

for (i = 0; i < va.size(); ++i) delete va[i];

for (i = 0; i < vh.size(); ++i) delete vh[i];

}

void info() {

int i;

for (i = 0; i < vi.size(); ++i) vi[i]->info();

for (i = 0; i < va.size(); ++i) va[i]->info();

for (i = 0; i < vh.size(); ++i) vh[i]->info();

}

vector<Infantryman\*> vi;

vector<Archer\*> va;

vector<Horseman\*> vh;

};

// Здесь создается армия той или иной стороны

class Game

{

public:

Army\* createArmy(ArmyFactory& factory) {

Army\* p = new Army;

p->vi.push\_back(factory.createInfantryman());

p->va.push\_back(factory.createArcher());

p->vh.push\_back(factory.createHorseman());

return p;

}

};

int main()

{

Game game;

RomanArmyFactory ra\_factory;

CarthaginianArmyFactory ca\_factory;

Army\* ra = game.createArmy(ra\_factory);

Army\* ca = game.createArmy(ca\_factory);

cout << "Roman army:" << endl;

ra->info();

cout << "\nCarthaginian army:" << endl;

ca->info();

}

**3.Создание потоков в API Windows**

Функция CreateThread создает поток, который выполняется в пределах виртуального адресного пространства вызывающего процесса.

Чтобы создавать поток, который запускается в виртуальном адресном пространстве другого процесса, используется функция [CreateRemoteThread](http://narovol.narod.ru/_tbkp/New_MSDN_API/fn_createremotethread.htm).

Синтаксис

|  |
| --- |
| HANDLE CreateThread(  LPSECURITY\_ATTRIBUTES lpThreadAttributes, // дескриптор защиты SIZE\_T dwStackSize,                       // начальный размер стека LPTHREAD\_START\_ROUTINE lpStartAddress,    // функция потока LPVOID lpParameter,                       // параметр потока DWORD dwCreationFlags,                    // опции создания LPDWORD lpThreadId                        // идентификатор потока  ); |

Параметры

lpThreadAttributes

[in] Указатель на структуру SECURITY\_ATTRIBUTES, которая обуславливает, может ли возвращенный дескриптор быть унаследован дочерними процессами. Если lpThreadAttributes является значением ПУСТО (NULL), дескриптор не может быть унаследован.

dwStackSize

[in] Начальный размер стека, в байтах. Система округляет это значение до самой близкой страницы памяти. Если это значение нулевое, новый поток использует по умолчанию размер стека исполняемой программы

lpStartAddress

[in] Указатель на определяемую программой функцию типа LPTHREAD\_START\_ROUTINE, код которой исполняется потоком и обозначает начальный адрес потока.

lpParameter

[in] Указатель на переменную, которая передается в поток.

dwCreationFlags

[in] Флажки, которые управляют созданием потока. Если установлен флажок CREATE\_SUSPENDED, создается поток в состоянии ожидания и не запускается до тех пор, пока не будет вызвана функция [ResumeThread](http://narovol.narod.ru/_tbkp/New_MSDN_API/fn_resumethread.htm). Если это значение нулевое, поток запускается немедленно после создания. В это время, никакие другие значения не поддерживаются.

lpThreadId

[out] Указатель на переменную, которая принимает идентификатор потока.

Возвращаемые значения

Если функция завершается успешно, величина возвращаемого значения - дескриптор нового потока.

Если функция завершается с ошибкой, величина возвращаемого значения - ПУСТО (NULL). Чтобы получать дополнительные данные об ошибках, вызовите [GetLastError.](http://narovol.narod.ru/_tbkp/New_MSDN_API/Debbag_error/fn_getlasterror.htm)

Одной из важнейших функций Windows, обеспечивающих управление процессами, является функция CreateProcess, которая создает новый процесс с единственным потоком. При вызове этой функции требуется указать имя файла исполняемой программы.

# Билет 13

**1.**

**2.**

**3.**

# Билет 15

1. **Паттерны, их виды и назначение**

Паттерн проектирования — это часто встречающееся решение определённой проблемы при проектировании архитектуры программ.

В отличие от готовых функций или библиотек, паттерн нельзя просто взять и скопировать в программу. Паттерн представляет собой не какой-то конкретный код, а общую концепцию решения той или иной проблемы, которую нужно будет ещё подстроить под нужды вашей программы.

Паттерны часто путают с алгоритмами, ведь оба понятия описывают типовые решения каких-то известных проблем. Но если алгоритм — это чёткий набор действий, то паттерн — это высокоуровневое описание решения, реализация которого может отличаться в двух разных программах

Паттерны отличаются по уровню сложности, детализации и охвата проектируемой системы. Проводя аналогию со строительством, вы можете повысить безопасность перекрёстка, поставив светофор, а можете заменить перекрёсток целой автомобильной развязкой с подземными переходами.

Самые низкоуровневые и простые паттерны — *идиомы*. Они не универсальны, поскольку применимы только в рамках одного языка программирования.

Самые универсальные — *архитектурные паттерны*, которые можно реализовать практически на любом языке. Они нужны для проектирования всей программы, а не отдельных её элементов.

Кроме того, паттерны отличаются и предназначением. В этой книге будут рассмотрены три основные группы паттернов:

**Порождающие паттерны** беспокоятся о гибком создании объектов без внесения в программу лишних зависимостей.

**Структурные паттерны** показывают различные способы построения связей между объектами.

**Поведенческие паттерны** заботятся об эффективной коммуникации между объектами.

**2.Модель MVC**

Шаблон **Model-View-Controller** — это методология разделения структуры приложения на специализированные компоненты

Схема MVC предполагает разделение всей системы на 3 взаимосвязанных компонента (подсистемы): так называемую **модель (Model)**, **представление (View)** и **контроллер (Controller)**

Преимущества такого подхода очевидны: модульность, расширяемость, простота поддержки и тестирования.

***Представление***  отвечает за отображение информации, поступающей из системы или в систему.

***Модель*** является сутью системы и отвечает за непосредственные алгоритмы, расчёты и тому подобное внутреннее устройство системы.

***Контроллер*** является связующим звеном между представлением и моделью системы, посредством которого и существует возможность произвести разделение между ними. Контроллер получает данные от пользователя и передает их в модель. Кроме того, он получает сообщения от модели, а также может изменять текущий режим представления.

#include <iostream>

#include <vector>

using namespace std;

class Observer

{

public:

virtual void update() = 0;

};

class Observable

{

public:

void addObserver(Observer\* observer)

{

\_observers.push\_back(observer);

}

void notifyUpdate()

{

int size = \_observers.size();

for (int i = 0; i < size; i++)

{

\_observers[i]->update();

}

}

private:

vector<Observer\*> \_observers;

};

class TemperatureModel : public Observable

{

public:

float getF()

{

return \_temperatureF;

}

float getC()

{

return (\_temperatureF - 32.0) \* 5.0 / 9.0;

}

void setF(float tempF)

{

\_temperatureF = tempF;

notifyUpdate();

}

void setC(float tempC)

{

\_temperatureF = tempC \* 9.0 / 5.0 + 32.0;

notifyUpdate();

}

private:

float \_temperatureF;

};

class ConsoleView : public Observer

{

public:

ConsoleView(TemperatureModel\* model)

{

\_model = model;

\_model->addObserver(this);

}

virtual void update()

{

system("cls");

printf("Temperature in Celsius: %.2f\n", \_model->getC());

printf("Temperature in Farenheit: %.2f\n", \_model->getF());

printf("Input temperature in Celsius: ");

}

private:

TemperatureModel\* \_model;

};

class Controller

{

public:

Controller(TemperatureModel\* model)

{

\_model = model;

}

void start()

{

\_model->setC(0);

float temp;

do

{

scanf\_s("%f", &temp);

\_model->setC(temp);

} while (temp != 0);

}

private:

TemperatureModel\* \_model;

};

int main()

{

TemperatureModel model;

ConsoleView view(&model);

Controller controller(&model);

controller.start();

return 0;

}

**3.Охрана try-catch**

***Исключительной ситуацией*** или **исключением (exception)** называется прерывание нормального хода выполнения программы в случае возникновения непредвиденного или аварийного события

Для реализации механизма обработки исключений в язык С++ введены три ключевых слова:

* **try** - начало блока исключений;
* **catch** - начало блока, "ловящего" исключение
* **throw** - генерация исключения

Контролируемый блок:

**try**

{

*Операторы*

}

Контролируемые блоки могут быть вложенными.

Среди операторов, могут быть обычные операторы языка Си++ и специальные операторы генерации исключений.

Генерация исключения:

**throw** *выражение\_генерации\_исключения*;

Исключение создается как статический объект, тип которого определяется типом значения *выражения\_генерации\_исключения*.

Местоположение оператора throw называют **точкой выброса исключения**.

Среди операторов контролируемого блока может быть любое количество операторов throw.

После формирования исключения исполняемый оператор **throw** передает управление непосредственно за пределы контролируемого блока, где обязательно должны находятся один или несколько обработчиков исключений:

**catch** (*тип\_исключения имя\_исключения*)

{

*//Операторы блока обработки исключений*

}

Если используется несколько обработчиков исключений, то они должны отличаться друг от друга типами исключений

Формы обработчика исключений:

**1. catch** (тип\_параметра имя\_параметра)

Содержит тип и имя параметра. Наличие имени позволяет передавать информацию из обрабатываемого исключения.

**2. catch** (тип параметра)

Содержит только тип параметра. Используются в случаях, когда важен только тип исключения и сам факт его получения.

**3. catch** (...)

Обработчик реагирует на любое исключение независимо от его типа. Как правило помещается в конце списка обработчиков для «перехвата» оставшихся необработанными исключений.

*Блоки catch обрабатываются последовательно. После нахождения первого соответствующего catch последующие обработчики не проверяются. Нельзя, например, помещать обработчик для базового класса перед обработчиком для производного класса.*

Если тип объекта исключения не сооствествует указанному типу то проверяются типы, в которые можно преобразовать объект при помощи стандартных правил преобразования.

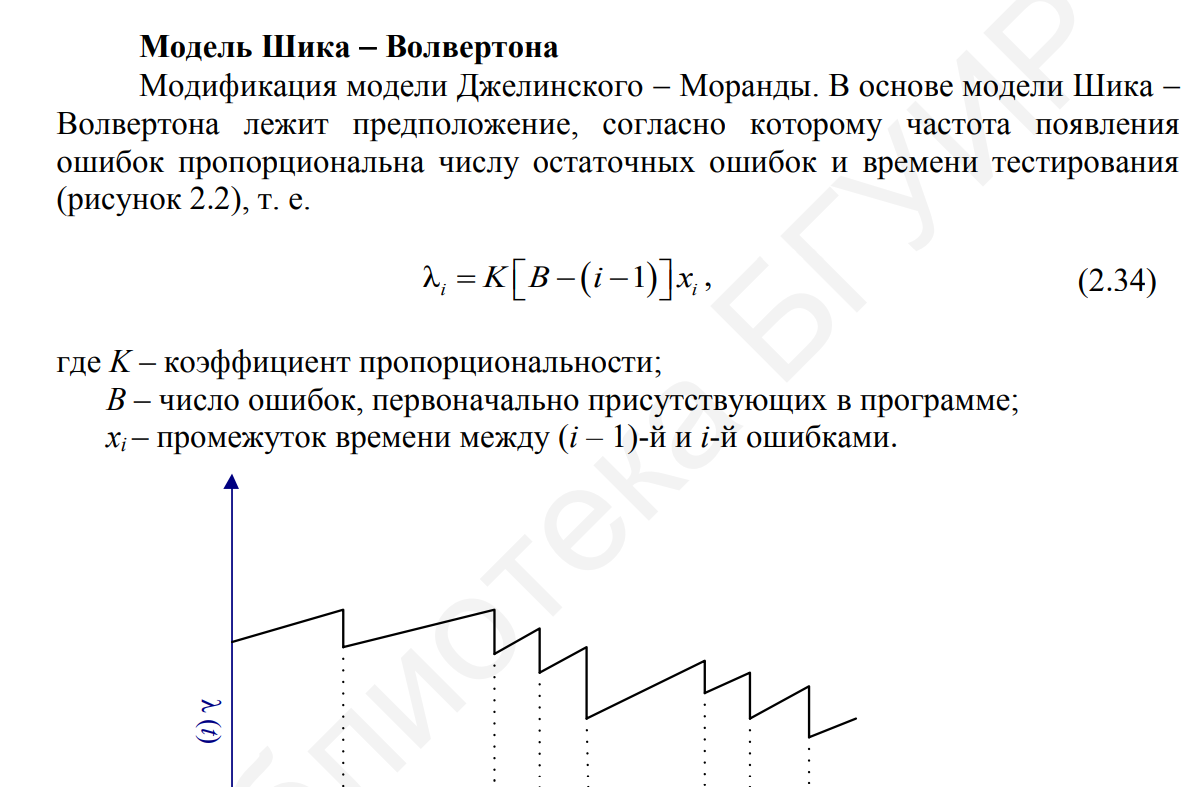
Если в контролируемом блоке **исключение не порождено**, то обработчики исключений обходятся

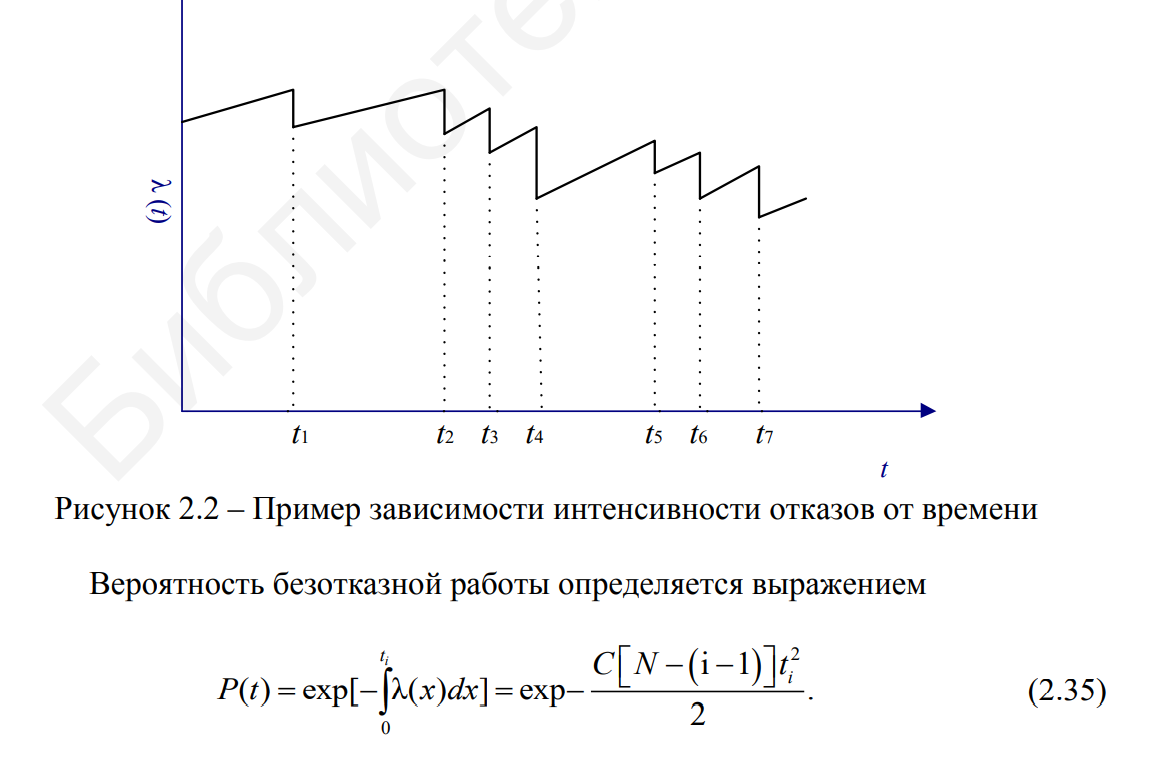
Если поиск подходящего блока обработки исключения в контролируемом блоке окончился неудачей, то поиск подходящего блока осуществляется в блоках верхнего уровня вложенности.

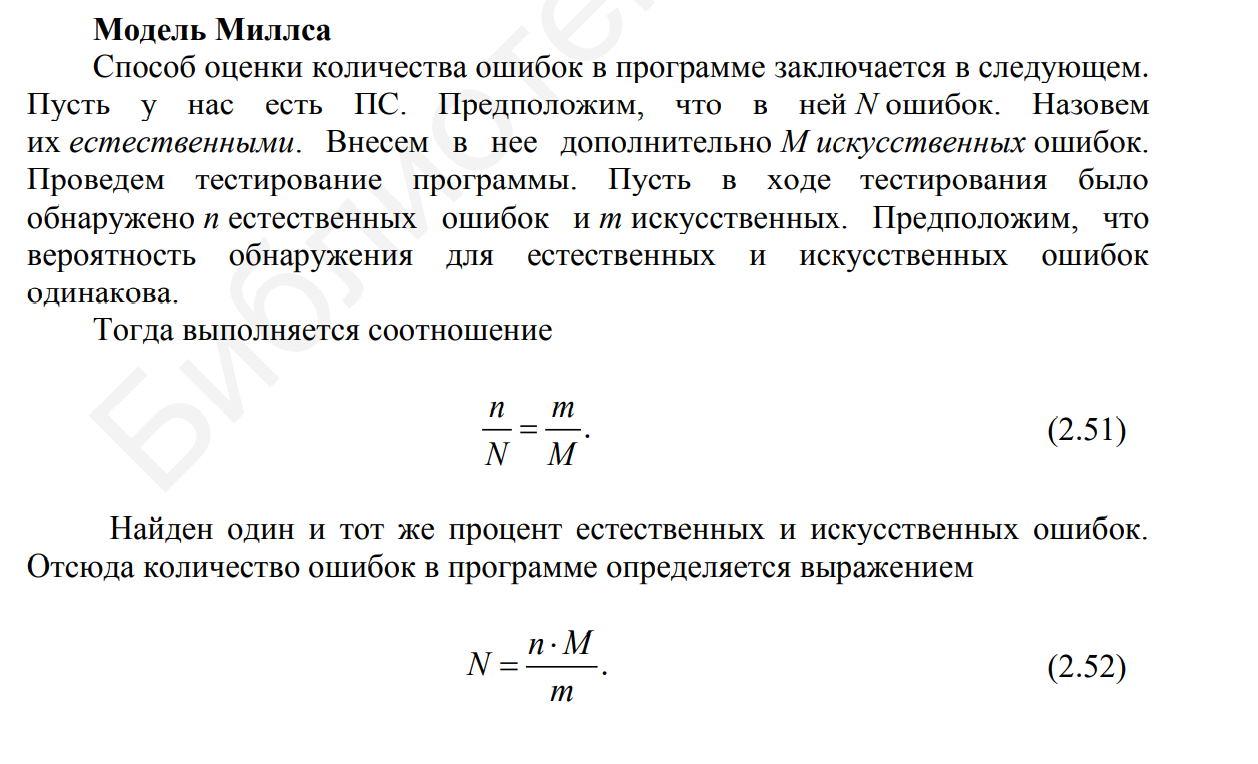
Для перехода на предшествующий уровень вложенности используется оператор **throw** без параметров.

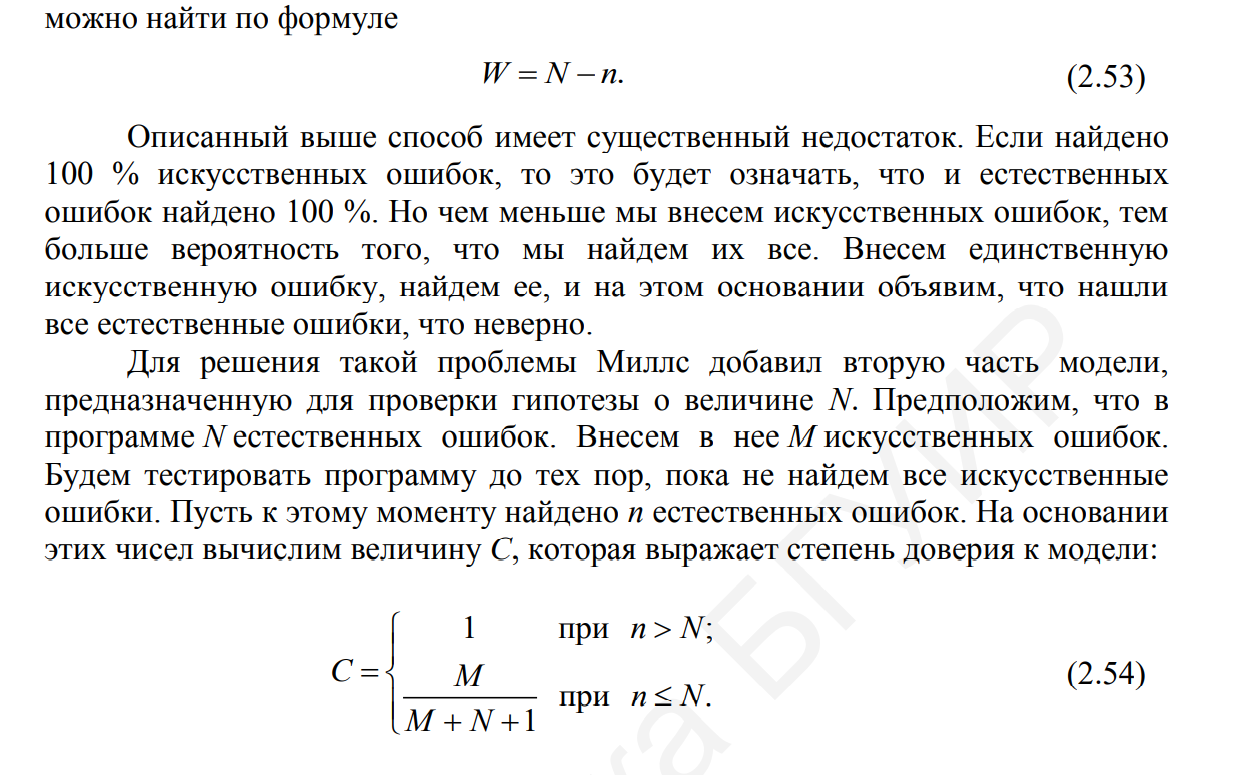
# Билет 16

1. **Метрики Надежности**









**2.Паттерн Итератор**

Шаблон Итератор обеспечивает последовательный доступ ко всем элементам коллекции, не раскрывая при этом её внутренней реализации. Задача итератора – упростить обход и сделать его однообразным для коллекций различных типов. Причём реализация шаблона может находиться как в объекте, представляющим коллекции, так и отдельно от него. В шаблоне Итератор задействованы следующие участники. Интерфейс итератора определяет методы для доступа к элементам коллекций. Итератор – класс, реализующий интерфейс итератора для конкретной коллекции. Интерфейс составного объекта (коллекции) задаёт способ получения итератора клиентом. Наконец, составной объект – это реализация интерфейса коллекции

#include <iostream>

#include <string>

#include <vector>

using namespace std;

template <typename T, typename U>

class Iterator {

public:

typedef typename vector<T>::iterator iter\_type;

Iterator(U\* p\_data, bool reverse = false) : m\_p\_data\_(p\_data) {

m\_it\_ = m\_p\_data\_->m\_data\_.begin();

}

//Устанавливает первый элемент в качестве текущего

void First() {

m\_it\_ = m\_p\_data\_->m\_data\_.begin();

}

//переход на следующий элемент

void Next() {

m\_it\_++;

}

//проверяет есть ли элементы в коллекции

bool IsDone() {

return (m\_it\_ == m\_p\_data\_->m\_data\_.end());

}

//возвращает текущий элемент

iter\_type Current() {

return m\_it\_;

}

private:

U\* m\_p\_data\_;

iter\_type m\_it\_;

};

/\*\*

\* Конкретные Коллекции предоставляют один или несколько методов для получения

\* новых экземпляров итератора, совместимых с классом коллекции.

\*/

template <class T>

class Container {

friend class Iterator<T, Container>;

public:

void Add(T a) {

m\_data\_.push\_back(a);

}

Iterator<T, Container>\* CreateIterator() {

return new Iterator<T, Container>(this);

}

private:

vector<T> m\_data\_;

};

class Data {

public:

Data(int a = 0) : m\_data\_(a) {}

void set\_data(int a) {

m\_data\_ = a;

}

int data() {

return m\_data\_;

}

private:

int m\_data\_;

};

/\*\*

\* The client code may or may not know about the Concrete Iterator or Collection

\* classes, for this implementation the container is generic so you can used

\* with an int or with a custom class.

\*/

void ClientCode() {

cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Squares from 0 to 100\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << endl;

Container<int> cont;

for (int i = 0; i < 101; i++) {

int c = i \* i;

cont.Add(c);

}

Iterator<int, Container<int>>\* it = cont.CreateIterator();

for (it->First(); !it->IsDone(); it->Next()) {

cout << \*it->Current() << endl;

}

Container<Data> cont2;

Data a(14), b(8700), c(2022);

cont2.Add(a);

cont2.Add(b);

cont2.Add(c);

cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Iterator with custom Class\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << endl;

Iterator<Data, Container<Data>>\* it2 = cont2.CreateIterator();

for (it2->First(); !it2->IsDone(); it2->Next()) {

cout << it2->Current()->data() << endl;

}

delete it;

delete it2;

}

int main() {

ClientCode();

return 0;

}

**3.Понятие рефлексии**

Рефлексия позволяет проверять известные типы данных во время выполнения. Рефлексия позволяет перечислять типы данных в заданной сборке, и могут быть обнаружены члены заданного класса или типа значения. Это верно независимо от того, был ли тип известен или на него ссылались во время компиляции. Это делает Рефлексию полезной функцией инструментов разработки и управления кодом.

Наиболее распространенный способ доступа к функциям Рефлексии — через метод GetType. Этот метод предоставляется System.Object, от которого наследуются все классы со сборкой мусора.

#include<iostream>

#include<stdio.h>

#include "pch.h"

using namespace System;

int main() {

Double value = 123456789012.34567;

Console::WriteLine("type of value = '{0}'", value.GetType());

return 0;

}

# Билет 17

**1.Построение систем на основе модельного подхода**

**Сущность структурного подхода к моделированию систем**

Система разбивается на функциональные подсистемы, которые, в свою очередь, делятся на подфункции, подфункции – на задачи и т.д. до конкретных процедур

**Базовые принципы структурного подхода**

принцип *«Разделяй и властвуй»*

принцип *иерархического упорядочивания*

принцип *абстрагирования*

принцип *непротиворечивости*

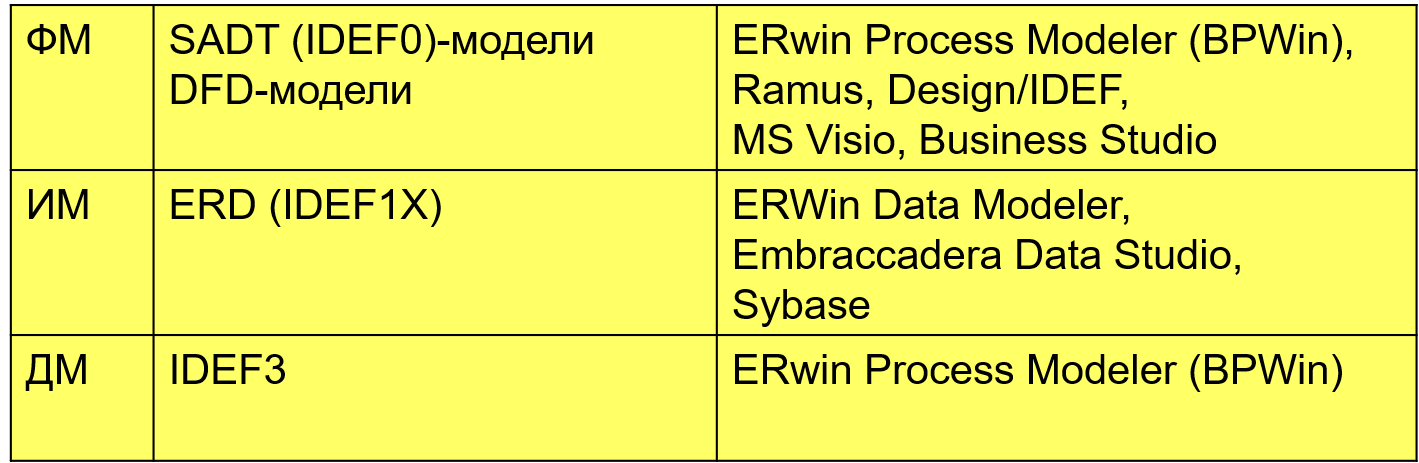
принцип *структурирования данных*

**3 типа моделей, используемых в структурном подходе:**

**1) функциональные модели (ФМ)**

**2) информационные модели (ИМ)**

**3) динамические модели (ДМ)**



Для любой системы определяющим является ее функциональное содержание, так как оно определяет ее основные свойства. Поэтому в основе функционального моделирования лежит функциональное содержание системы, в качестве отношений между функциями рассматривается информация об объектах, связывающих эти функции

**2.Паттерн Состояние**

При использовании шаблона Состояние поведение объекта меняется в зависимости от текущего контекста

#include <stdio.h>

#include <string>

#include<vector>

#include <iostream>

using namespace std;

/\*\*

\* Базовый класс Состояния объявляет методы, которые должны реализовать все

\* Конкретные Состояния, а также предоставляет обратную ссылку на объект

\* Контекст, связанный с Состоянием. Эта обратная ссылка может использоваться

\* Состояниями для передачи Контекста другому Состоянию.

\*/

class Context;

class State {

protected:

Context\* context\_;

public:

virtual ~State() {

}

void set\_context(Context\* context) {

this->context\_ = context;

}

virtual void Handle1() = 0;

virtual void Handle2() = 0;

};

/\*\*

\* Контекст определяет интерфейс, представляющий интерес для клиентов. Он также

\* хранит ссылку на экземпляр подкласса Состояния, который отображает текущее

\* состояние Контекста.

\*/

class Context {

private:

State\* state\_;

public:

Context(State\* state) : state\_(nullptr) {

this->TransitionTo(state);

}

~Context() {

delete state\_;

}

/\*\*

\* Контекст позволяет изменять объект Состояния во время выполнения.

\*/

void TransitionTo(State\* state) {

cout << "\n Sending letter to " << typeid(\*state).name() << ".\n";

if (this->state\_ != nullptr)

delete this->state\_;

this->state\_ = state;

this->state\_->set\_context(this);

}

/\*\*

\* Контекст делегирует часть своего поведения текущему объекту Состояния.

\*/

void Request1() {

this->state\_->Handle1();

}

void Request2() {

this->state\_->Handle2();

}

};

/\*\*

\* Конкретные Состояния реализуют различные модели поведения, связанные с

\* состоянием Контекста.

\*/

class Minsk : public State {

public:

void Handle1() override;

void Handle2() override {

cout << "Minsk handles request2.\n";

}

};

class Bobruisk : public State {

public:

void Handle1() override {

cout << "Bobruisk handles request1.\n";

}

void Handle2() override {

cout << "Bobruisk handles request2.\n";

cout << "Bobruisk wants to send the letter back\n";

this->context\_->TransitionTo(new Minsk);

}

};

void Minsk::Handle1() {

{

cout << "Minsk handles request1.\n";

cout << "Minsk wants to wants to send the letter \n";

this->context\_->TransitionTo(new Bobruisk);

}

}

/\*\*

\* Клиентский код.

\*/

void ClientCode() {

Context\* context = new Context(new Minsk);

context->Request1();

context->Request2();

delete context;

}

int main() {

ClientCode();

return 0;

}

**3.Верификация и тестирование**

**Верификация** - это процесс для определения, выполняют ли программные средства и их компоненты требования, наложенные на них в последовательных этапах ЖЦ ПС. Анализ, просмотры (обзоры) и тестирование от требований являются важнейшей частью верификации и установления корректности программ. Основная цель верификации ПС состоит в том, чтобы обнаружить, зарегистрировать и устранить дефекты и ошибки, которые внесены во время последовательной разработки или модификации программ. Для эффективности затрат ресурсов при ее реализации, верификация должна быть интегрирована как можно раньше с процессами проектирования, разработки и сопровождения. Обычно она проводится сверху вниз, начиная от общих требований, заданных в техническом задании и/или спецификации на всю информационную систему до детальных требований на программные модули и их взаимодействие

**Цели верификации ПC достигаются посредством** последовательного выполнения комбинации из просмотров, анализов, разработки тестовых сценариев и процедур и последующего выполнения этих процедур

Тестирование ПС от требованийимеет две взаимодополняющие цели. Первая цель - показать, что ПС удовлетворяет заданным требованиям к нему. Вторая цель - показать с высокой степенью доверия, что устранены дефекты и ошибки, которые могли бы привести к возникновению недопустимых отказовых ситуаций, влияющих на корректность и надежность системы. Тестирование интеграции программных компонентов, основанное на требованиях, должно акцентироваться на взаимосвязях между требованиями к ПС и на реализации требований к его архитектуре. Цель такого тестирования - гарантировать, что программные компоненты взаимодействуют друг с другом корректно и удовлетворяют требованиям к ПС и к его архитектуре.

**Только совместное и систематическое применение различных методов тестирования позволяет достигать высокое качество функционирования сложных комплексов программ**

**Анализ покрытия тестами требований к ПС** должен определять, какие требования не были протестированы и какие структуры программного средства не были исполнены при тестировании. Анализ тестового покрытия состоит из **двух шагов**, включающих анализ покрытия, основанного на требованиях, и анализ структурного покрытия. **Первый шаг** анализирует тестовые наборы относительно требований ПС, чтобы подтвердить, что выбранные наборы тестов удовлетворяют установленным критериям. Этот анализ покрытия, основанного на требованиях, должен определить, насколько полно тестирование проверило реализацию всех требований в спецификации к ПС, и показать потребность в дополнительных тестовых наборах. Тестовые варианты, основанные на требованиях, могут не полностью покрыть структуру программы. Поэтому дополнительно выполняется анализ структурного покрытия и проводится его верификация. **Второй шаг** должен подтвердить, что процедуры тестирования, основанные на требованиях, покрыли всю структуру программы. Анализ структурного покрытия должен определять, не пропущены ли элементы структуры программы, которые не проверены тестовыми процедурами, основанными на требованиях.

# Билет 18

**1.Принципы ООП**

Основополагающей идеей объектно-ориентированного подхода является «**абстракция**», т.е. такое объединение свойств и поведения объекта, при котором с объектом можно работать как с единым целым, не вдаваясь в детали его внутреннего построения и функционирования.

**Инкапсуляция** **(encapsulation**) – механизм, связывающий вместе данные и методы и защищающий их от произвольного внешнего доступа.

Описание икапсулированного объекта называется **классом**.

Класс является определением, характеризующим данные и методы, включаемые в объект.

При объявлении класса не создаются объекты этого класса (не происходит выделения памяти).

Все объекты типа класс, для которых выделяется память, называются **экземплярами (объектами) классов**.

Использование свойств ранее определенных классов при образовании иерархии производных классов называется **наследованием** (**inheritance**).

Возможность ассоциации некоторого имени с множеством уникальных для каждого уровня иерархии производных объектов называется **полиморфизмом** (polymorphizm).

Свойство ***полиморфизма*** позволяет использовать одинаковое название метода для решения отличающихся на каждом уровне иерархии задач.

В результате в объекте-родителе и объекте-потомке будут действовать одноименные методы с различными алгоритмами.

Три способа реализации полиморфизма:

* **статический полиморфизм** − реализуется на этапе компиляции путем перегрузки функций и операций;
* **динамический полиморфизм** − реализуется во время выполнения программы с использованием виртуальных функций;

**параметрический** **полиморфизм** − реализуется на этапе компиляции с использованием механизма шаблонов

**2.Паттерн Команда**

Шаблон Команда обеспечивает обработку команды в виде объекта, что позволяет сохранять её, передавать в качестве аргумента методам, а также возвращать её в виде результата, как и любой другой объект

#include <stdio.h>

#include <string>

#include<vector>

#include <iostream>

using namespace std;

class Game

{

public:

void create() {

cout << "Create game " << endl;

}

void open(string file) {

cout << "Open game from " << file << endl;

}

void save(string file) {

cout << "Save game in " << file << endl;

}

void make\_move(string move) {

cout << "Make move " << move << endl;

}

};

string getPlayerInput(string prompt) {

string input;

cout << prompt;

cin >> input;

return input;

}

// Базовый класс – интерфейс команды

class Command

{

public:

virtual ~Command() {}

virtual void execute() = 0;

protected:

Command(Game\* p) : pgame(p) {}

Game\* pgame;

};

class CreateGameCommand : public Command

{

public:

CreateGameCommand(Game\* p) : Command(p) {}

void execute() {

pgame->create();

}

};

class OpenGameCommand : public Command

{

public:

OpenGameCommand(Game\* p) : Command(p) {}

void execute() {

string file\_name;

file\_name = getPlayerInput("Enter file name:");

pgame->open(file\_name);

}

};

class SaveGameCommand : public Command

{

public:

SaveGameCommand(Game\* p) : Command(p) {}

void execute() {

string file\_name;

file\_name = getPlayerInput("Enter file name:");

pgame->save(file\_name);

}

};

class MakeMoveCommand : public Command

{

public:

MakeMoveCommand(Game\* p) : Command(p) {}

void execute() {

// Сохраним игру для возможного последующего отката

pgame->save("TEMP\_FILE");

string move;

move = getPlayerInput("Enter your move:");

pgame->make\_move(move);

}

};

class UndoCommand : public Command

{

public:

UndoCommand(Game\* p) : Command(p) {}

void execute() {

// Восстановим игру из временного файла

pgame->open("TEMP\_FILE");

}

};

int main()

{

Game game;

// Имитация действий игрока

vector<Command\*> v;

// Создаем новую игру

v.push\_back(new CreateGameCommand(&game));

// Делаем несколько ходов

v.push\_back(new MakeMoveCommand(&game));

v.push\_back(new MakeMoveCommand(&game));

// Последний ход отменяем

v.push\_back(new UndoCommand(&game));

// Сохраняем игру

v.push\_back(new SaveGameCommand(&game));

for (size\_t i = 0; i < v.size(); ++i)

v[i]->execute();

for (size\_t i = 0; i < v.size(); ++i)

delete v[i];

return 0;

}

**3.Метрики в Visual Studio**

**Индекс удобства обслуживания** — вычисляет значение индекса от 0 до 100, которое представляет относительную простоту обслуживания кода. Высокое значение означает лучшую поддержку. Цветовые оценки можно использовать для быстрого выявления проблемных мест в коде. Зеленая оценка составляет от 20 до 100 и указывает, что код имеет хорошую поддержку. Желтая оценка составляет от 10 до 19 лет и указывает, что код умеренно поддерживается. Красная оценка — это оценка от 0 до 9 и указывает на низкую поддержку.

**Цикломатическая сложность** — измеряет структурную сложность кода. Он создается путем вычисления количества различных путей кода в потоке программы. Программа, которая имеет сложный поток управления, требует больше тестов для достижения хорошего объема протестированного кода и менее поддерживается. Дополнительные сведения см. в [записи Википедии по цикломатической сложности](https://wikipedia.org/wiki/Cyclomatic_complexity).

**Глубина наследования** — указывает количество различных классов, наследуемых друг от друга, вплоть до базового класса. Глубина наследования аналогична взаимозависимости классов, что изменение базового класса может повлиять на любой из унаследованных классов. Чем выше это число, тем глубже наследование и чем выше потенциал для изменений базового класса, что приведет к критическому изменению. Для глубины наследования низкое значение хорошо, и высокое значение плохо.

**Объединение классов** — измеряет связь с уникальными классами с помощью параметров, локальных переменных, возвращаемых типов, вызовов методов, универсальных или шаблонных экземпляров, базовых классов, реализаций интерфейсов, полей, определенных во внешних типах и оформлении атрибутов. Хорошая конструкция программного обеспечения определяет, что типы и методы должны иметь высокую сплоченность и низкую взаимозависимость. Высокая взаимозависимость указывает на структуру, которую трудно использовать и поддерживать из-за многих взаимозависимостей на других типах.

**Строки исходного кода** — указывает точное количество строк исходного кода, присутствующих в исходном файле, включая пустые строки

**Строки исполняемого кода** — указывает приблизительное количество строк или операций исполняемого кода. Это число операций в исполняемом коде.

# Билет 19

**1. Методы рефакторинга кода**

В программировании термин рефакторинг означает изменение исходного кода программы без изменения его внешнего поведения.

Рефакторинг изначально не предназначен для исправления ошибок и добавления новой функциональности, он вообще не меняет поведение программного обеспечения и это помогает избежать ошибок и облегчить добавление функциональности. Он выполняется для улучшения понятности кода или изменения его структуры, для удаления «мёртвого кода» — всё это для того, чтобы в будущем код было легче поддерживать и развивать.

**Методы рефакторинга**

Наиболее употребимые методы рефакторинга:

**Изменение сигнатуры метода (change method signature)**

Суть изменения сигнатуры метода заключается в добавлении, изменении или удалении параметра метода. Изменив сигнатуру метода, необходимо скорректировать обращения к нему в коде всех клиентов. Это изменение может затронуть внешний интерфейс программы, кроме того, не всегда разработчику, изменяющему интерфейс, доступны все клиенты этого интерфейса, поэтому может потребоваться та или иная форма регистрации изменений интерфейса для последующей передачи их вместе с новой версией программы.

**Инкапсуляция поля (encapsulate field)**

В случае, если у класса имеется открытое поле, необходимо сделать его закрытым и обеспечить методы доступа. После «Инкапсуляции поля» часто применяется

«Перемещение метода».

**Выделение метода (extract method)**

Выделение метода заключается в выделении из длинного и/или требующего комментариев кода отдельных фрагментов и преобразовании их в отдельные методы, с подстановкой подходящих вызовов в местах использования. В этом случае действует правило: если фрагмент кода требует комментария о том, что он делает, то он должен быть выделен в отдельный метод.

Также правило: один метод не должен занимать более чем один экран (25-50 строк, в зависимости от условий редактирования), в противном случае некоторые его фрагменты имеют самостоятельную ценность и подлежат выделению. Из анализа связей выделяемого фрагмента с окружающим контекстом делается вывод о перечне параметров нового метода и его локальных переменных.

**Перемещение метода (move method)**

Перемещение метода применяется по отношению к методу, который чаще обращается к другому классу, чем к тому, в котором сам располагается.

**Замена условного оператора полиморфизмом (replace conditional with polymorphism)**

Условный оператор с несколькими ветвями заменяется вызовом полиморфного метода некоторого базового класса, имеющего подклассы для каждой ветви исходного оператора. Выбор ветви осуществляется неявно, в зависимости от того, экземпляру какого из подклассов оказался адресован вызов.

**Основные принципы:**

1.вначале следует создать базовый класс и нужное число подклассов;

в некоторых случаях следует провести оптимизацию условного оператора путём «Выделения метода»;

2.возможно использование «Перемещения метода», чтобы поместить условный оператор в вершину иерархии наследования;

3.выбрав один из подклассов, нужно конкретизировать в нём полиморфный метод базового класса и переместить в него тело соответствующей ветви условного оператора;

4.повторить предыдущее действие для каждой ветви условного оператора;

5.заменить весь условный оператор вызовом полиморфного метода базового класса.

1. **Интеграционное тестирование с использованием Mock ????**

Интеграционное тестирование – это тип тестирования, при котором программные модули объединяются логически и тестируются как группа. Как правило, программный продукт состоит из нескольких программных модулей, написанных разными программистами. Целью нашего тестирования является выявление багов при взаимодействии между этими программными модулями и в первую очередь направлен на проверку обмена данными между этими самими модулями. Именно поэтому оно также называется «I & T» (интеграция и тестирование), «тестирование строк» и иногда «тестирование потоков».

Интеграционное тестирование отличается от других видов тестирования тем, что он сосредоточен в основном на интерфейсах и потоке данных (между модулями). Здесь приоритет проверки присваивается интегрирующим ссылкам, а не функциям блока, которые уже проверены   
**Подход Большого взрыва**   
Здесь все компоненты собираются вместе, а затем тестируются.   
**Преимущества**:

Удобно для небольших систем.

**Недостатки**:

Сложно локализовать баги.

Учитывая огромное количество интерфейсов, некоторые из них при тестировании можно запросто пропустить.

Недостаток времени для группы тестирования, т.к тестирование интеграции может начаться только после того, как все модули спроектированы.

Поскольку все модули тестируются одновременно, критические модули высокого риска не изолируются и тестируются в приоритетном порядке. Периферийные модули, которые имеют дело с пользовательскими интерфейсами, также не изолированы и не проверены на приоритет.  
**Инкрементальный подход**   
В данном подходе тестирование выполняется путем объединения двух или более логически связанных модулей. Затем добавляются другие связанные модули и проверяются на правильность функционирования. Процесс продолжается до тех пор, пока все модули не будут соединены и успешно протестированы.   
Поэтапный подход, в свою очередь, осуществляется **двумя разными методами**:

1.Снизу вверх

2.Сверху вниз  
**Заглушка и драйвер**Инкрементальный подход осуществляется с помощью фиктивных программ, называемых заглушками и драйверами. Заглушки и драйверы не реализуют всю логику программного модуля, а только моделируют обмен данными с вызывающим модулем.   
**Заглушка**: вызывается тестируемым модулем.   
**Драйвер**: вызывает модуль для тестирования.   
**Интеграция «снизу вверх»**   
В восходящей стратегии каждый модуль на более низких уровнях тестируется с модулями более высоких уровней, пока не будут протестированы все модули. Требуется помощь драйверов для тестирования   
**Преимущества**:

Проще локализовать ошибки.

Не тратится время на ожидание разработки всех модулей, в отличие от подхода Большого взрыва.

**Недостатки**:

Критические модули (на верхнем уровне архитектуры программного обеспечения), которые контролируют поток приложения, тестируются последними и могут быть подвержены дефектам.

Невозможно реализовать ранний прототип  
**Интеграция «сверху вниз»**   
При подходе «сверху вниз» тестирование, что логично, выполняется сверху вниз, следуя потоку управления программной системы. Используются заглушки для тестирования.   
**Преимущества:**

Проще локализовать баги.

Возможность получить ранний прототип.

Критические Модули тестируются на приоритет; основные недостатки дизайна могут быть найдены и исправлены в первую очередь.

**Недостатки:**

Нужно много пней.

Модули на более низком уровне тестируются неадекватно  
**Сэндвич (гибридная интеграция)**   
Эта стратегия представляет собой комбинацию подходов «сверху вниз» и «снизу вверх». Здесь верхнеуровневые модули тестируются с нижнеуровневыми, а нижнеуровневые модули интегрируются с верхнеуровневыми, соответственно, и тестируются. Эта стратегия использует и заглушки, и драйверы

**Атрибуты Интеграционного тестирования**  
Включает в себя следующие **атрибуты**:

1.Методы / Подходы к тестированию (об этом говорили выше).

2.Области применения и Тестирование интеграции.

3.Роли и обязанности.

4.Предварительные условия для Интеграционного тестирования.

5.Тестовая среда.

6.Планы по снижению рисков и автоматизации.   
**Критерии старта и окончания интеграционного тестирования**   
**Критерии старта**:

1.Модули и модульные компоненты

2.Все ошибки с высоким приоритетом исправлены и закрыты

3.Все модули должны быть заполнены и успешно интегрированы.

4.Наличие плана Интеграционного тестирования, тестовый набор, сценарии, которые должны быть задокументированы.

5.Наличие необходимой тестовой среды  
**Критерии окончания:**

1.Успешное тестирование интегрированного приложения.

2.Выполненные тестовые случаи задокументированы

3.Все ошибки с высоким приоритетом исправлены и закрыты

4.Технические документы должны быть представлены после выпуска Примечания.

**3.Паттерн строитель**

Шаблон Строитель позволяет отделить процесс создания сложного объекта от его реализации. При этом результатом одних и тех же операций могут быть различные объекты. Данный шаблон используется в случае, если процесс создания объекта можно разделить на стадии (шаги). При этом конструирование должно обеспечивать возможность создавать разные объекты. Шаблон Строитель включает двух участников процесса (рис. 10). Строитель (Builder) предоставляет методы для сборки частей объекта, при необходимости преобразовывает исходные данные в нужный вид, создаёт и выдаёт объект. Распорядитель (Director) определяет стратегию сборки: собирает данные и определяет порядок вызовов методов строителя. Задача распорядителя – сокрытие стратегии сборки. Это позволит, при необходимости, модифицировать или даже полностью менять её, не затрагивая остальной код

#include <iostream>

#include <vector>

using namespace std;

class House1 {

public:

vector<string> parts\_;

void ListParts()const {

cout << "House parts: ";

for (size\_t i = 0; i < parts\_.size(); i++) {

if (parts\_[i] == parts\_.back()) {

cout << parts\_[i];

}

else {

cout << parts\_[i] << ", ";

}

}

cout << "\n\n";

}

};

/\*

Интерфейс Строителя объявляет создающие методы для различных частей объектов

Продуктов.

\*/

class Builder {

public:

virtual ~Builder() {}

virtual void Walls() const = 0;

virtual void Roof() const = 0;

virtual void Windows() const = 0;

};

/\*

Классы Конкретного Строителя следуют интерфейсу Строителя и предоставляют

конкретные реализации шагов построения.

\*/

class ConcreteBuilder1 : public Builder {

private:

House1\* house;

/\*

Новый экземпляр строителя должен содержать пустой объект продукта,

который используется в дальнейшей сборке.

\*/

public:

ConcreteBuilder1() {

this->Reset();

}

~ConcreteBuilder1() {

delete house;

}

void Reset() {

this->house= new House1();

}

/\*\*

\* Все этапы производства работают с одним и тем же экземпляром продукта.

\*/

void Walls()const override {

this->house->parts\_.push\_back("Walls");

}

void Roof()const override {

this->house->parts\_.push\_back("Roof");

}

void Windows()const override {

this->house->parts\_.push\_back("Windows");

}

/\*

Конкретные Строители должны предоставить свои собственные методы

получения результатов. Это связано с тем, что различные типы строителей

могут создавать совершенно разные продукты с разными интерфейсами.

Поэтому такие методы не могут быть объявлены в базовом интерфейсе

Строителя (по крайней мере, в статически типизированном языке

программирования). Обратите внимание, что PHP является динамически

типизированным языком, и этот метод может быть в базовом интерфейсе.

Однако мы не будем объявлять его здесь для ясности.

Как правило, после возвращения конечного результата клиенту, экземпляр

строителя должен быть готов к началу производства следующего продукта.

Поэтому обычной практикой является вызов метода сброса в конце тела

метода getHouse. Однако такое поведение не является обязательным, вы

можете заставить своих строителей ждать явного запроса на сброс из кода

клиента, прежде чем избавиться от предыдущего результата.

\*/

House1\* GetHouse() {

House1\* result = this->house;

this->Reset();

return result;

}

};

/\*

Директор отвечает только за выполнение шагов построения в определённой

последовательности. Это полезно при производстве продуктов в определённом

порядке или особой конфигурации.

\*/

class Director {

private:

Builder\* builder;

/\*

Директор работает с любым экземпляром строителя, который передаётся ему

клиентским кодом. Таким образом, клиентский код может изменить конечный

тип вновь собираемого продукта.

\*/

public:

void set\_builder(Builder\* builder) {

this->builder = builder;

}

/\*

Директор может строить несколько вариаций продукта, используя одинаковые

шаги построения.

\*/

void BuildMinimalHouse() {

this->builder-> Walls();

}

void FullHouse() {

this->builder-> Walls();

this->builder-> Roof();

this->builder-> Windows();

}

};

/\*

Клиентский код создаёт объект-строитель, передаёт его директору, а затем

инициирует процесс построения. Конечный результат извлекается из объекта-

строителя.

\*/

void ClientCode(Director& director)

{

ConcreteBuilder1\* builder = new ConcreteBuilder1();

director.set\_builder(builder);

cout << "Minimal house:\n";

director.BuildMinimalHouse();

House1\* p = builder->GetHouse();

p->ListParts();

delete p;

cout << "Full house:\n";

director.FullHouse();

p = builder->GetHouse();

p->ListParts();

delete p;

// Помните, что паттерн Строитель можно использовать без класса Директор.

cout << "Custom house:\n";

builder-> Walls();

builder-> Roof();

p = builder->GetHouse();

p->ListParts();

delete p;

delete builder;

}

int main() {

Director\* director = new Director();

ClientCode(\*director);

delete director;

return 0;

}

# Билет 20

**1. Различные метрики программ (Холстед, МакКейб, Чепин и др.) Примеры**

Метрики оценки сложности программ разбивают на 3 основные группы

1. метрики размера программ

2. метрики сложности потока управления программ

3. метрики сложности потока данных программ

Основная цель метрик сложности – выявить наиболее критичные участки программного проекта, которые являются потенциальными источниками ошибок и повышенных рисков на всех стадиях его жизненного цикла.

**Метрики первой группы (количественная оценка)** базируются на определении количественных характеристик, связанных с размером программы, и ориентированы на анализ исходного текста программ. Могут использоваться для оценки сложности промежуточных продуктов разработки.

Самой элементарной метрикой является количество строк кода (SLOC). Данная метрика была изначально разработана для оценки трудозатрат по проекту. Однако из-за того, что одна и та же функциональность может быть разбита на несколько строк или записана в одну строку, метрика стала практически неприменимой с появлением языков, в которых в одну строку может быть записано больше одной команды. Поэтому различают логические и физические строки кода. Логические строки кода — это количество команд программы. Данный вариант описания так же имеет свои недостатки, так как сильно зависит от используемого языка программирования и стиля программирования.  
Кроме SLOC к количественным характеристикам относят также:

* количество пустых строк,
* количество комментариев,
* процент комментариев (отношение числа строк, содержащих комментарии к общему количеству строк, выраженное в процентах),
* среднее число строк для функций (классов, файлов),
* среднее число строк, содержащих исходный код для функций (классов, файлов),
* среднее число строк для модулей.

К группе оценок размера программ можно отнести также **метрику Холстеда**. За базу в метрике принят подсчет количества операторов и операндов, используемых в программе, т.е. определение размера программы по ее составляющим.

Основу метрики составляют 4 измеряемых характеристики программы:

n1 – число уникальных операторов программы, включая символы-разделители, имена процедур и знаки операций (словарь операторов);

n2 – число уникальных операндов программы (словарь операндов);

N1– общее число операторов в программе;

N2– общее число операндов в программе.

Опираясь на эти характеристики, получаемые непосредственно при статическом анализе исходных текстов программ, рассчитываются 3 базовые оценки размера программы:

* словарь программы

n=n1+n2(ед.), (1)

* длину программы

N=N1+N2(ед.), (2)

* объем программы (число битов, т.е. логических единиц информации, необходимых для записи программы)

V=N⋅log2(n) (бит). (3)

Через эти базовые оценки в свою очередь различными формулами определяются: уровень качества программирования, сложность понимания программы, трудоемкость кодирования программы, информационное содержание программы и другие оценки программного кода.

Сложность программы по метрике Холстеда рассчитывается по формуле:

Hdiff=(n1/2)⋅(N2/n2)

**Метрики второй группы (оценка сложности потока управления программ)** базируются на анализе управляющего графа программы, поэтому эти метрики могут применяться для оценки сложности промежуточных продуктов разработки. Эти метрики используются главным образом для апостериорного анализа, однако могут применяться и на ранних стадиях работы при осуществлении проектирования.

Одна из самых распространенных метрик сложности потока управления программ – цикломатическая сложность, предложена Томасом МакКейбом (Thomas McCabe). Данная метрика предназначена для оценивания сложности потока управления программы (control flow graph).

Формула вычисления цикломатической сложности выглядит следующим образом:

Z(G)=l–v+2⋅p (1)

Где l – число ребер (дуг), v – число узлов (вершин), p – число компонентов связности графа.

**Число компонентов связности** графа можно рассматривать как количество дуг, которые необходимо добавить для преобразования графа в сильносвязный. **Сильносвязным** называется граф, любые две вершины которого взаимно достижимы.

Следующий класс метрик — метрики сложности потока управления данных.  
**Метрика Чепина**

Суть метода состоит в анализе характера использования переменных в коде программы. Существует несколько ее модификаций.

Все множество переменных программы разбивается на 4 функциональные группы:

* P –параметрические (вводимые переменные для расчетов и для обеспечения вывода).
* M – модифицируемые, или создаваемые внутри программы переменные.
* C – контролирующие (переменные, участвующие в управлении работой программного модуля).
* T – временные (не используемые в программе переменные), «паразитные».

Поскольку каждая переменная может выполнять одновременно несколько функций, необходимо учитывать ее в каждой соответствующей функциональной группе.

Расчет метрики Чепина выполняется по формуле:

Q=a1⋅P+a2⋅M+a3⋅C+a4⋅T

где a1, a2, a3, a4 – весовые коэффициенты.

Весовые коэффициенты использованы для отражения различного влияния на сложность программы каждой функциональной группы.

По мнению автора метрики, наибольший вес, равный трем, имеет функциональная группа С, так как она влияет на поток управления программы. Весовые коэффициенты остальных групп распределяются следующим образом: a1=1;a2=2;a4=0.5. Весовой коэффициент группы T не равен нулю, поскольку «паразитные» переменные не увеличивают сложности потока данных программы, но иногда затрудняют ее понимание.

С учетом весовых коэффициентов выражение примет вид:

Q=1⋅P+2⋅M+3⋅C+0.5⋅T

**2. Понятие вычислительной сложности. Классы P и NP ?????**

**Вычисли́тельная сло́жность** —функция зависимости объёма работы, которая выполняется некоторым алгоритмом, от размера входных данных.

**Класс P, или класс P-сложных задач**. Задачи, относящиеся к этому классу, называются «хорошими» и могут быть решены за полиноминальное время с помощью детерминированной машины Тьюринга. Алгоритмы их решения называют эффективными. Все остальные задачи считаются плохими. Класс P является одним из самых узких классов сложности. Примерами алгоритмов класса P являются стандартные алгоритмы целочисленного сложения, умножения, деления, перемножения матриц, сортировки, поиска и т.п.

**Класс NP, или класс NP -сложных задач**. Это обобщенный абстрактный класс, к которому относятся те задачи, решение которых детерминированным способом найти практически очень сложно. Обычно такие задачи формулируются в виде проблемы разрешимости, когда окончательный ответ «Да» или «Нет» связан с оценкой или проверкой большого количества промежуточных вариантов решения

**3. Образование функций на основе Лямбда-выражений. Примеры**

Лямбда-выражение – способ определения анонимной функции непосредственно в месте вызова.

Лямбда-выражения имеют несколько вариантов синтаксиса, например:

[список\_захвата] (список\_параметров) -> возвращаемый\_тип {тело\_выражения}

Список захвата определяет, к каким переменным из внешнего окружения получает доступ лямбда-выражение.

[] – доступ к переменным из внешней области видимости запрещен;

[=] – доступ ко всем внешним переменным осуществляется по значению;

[&] – доступ ко всем внешним переменным осуществляется по ссылке.

Доступ к переменным с префиксом с амперсандом (&) осуществляется по ссылке, а к переменным без префикса — по значению.

Элементы списка отделяются друг от друга запятыми.

Пример списков захвата:

[x, &y] – x захвачено по значению, y – по ссылке;

[&, x] – x захвачено по значению, остальные по ссылке;

[=, \*this] – захват переменных и объекта по значению.

#include <iostream>

using namespace std;

int main(){

double x = 7, y = 3;

double s = 0;

[=, &s]() {return s = x / y; }();

cout << s << endl; // Выводит: 2.33333

[&](double x, double y) {return s = x / y; }(15, 3);

cout << s << endl; // Выводит: 5

auto w = [=, &s](double x) {return x / y; };

cout << w(9) << endl; // Выводит: 3

cout << w(14) << endl; // Выводит: 4.66667

}