**Лекция 1. Структура case-средства. Структура среды разработки. Основные возможности.**

Термин "CASE" (Computer Aided Software/System Engineering) используется в настоящее время в весьма широком смысле. Первоначальное значение термина "CASE", ограниченное вопросами автоматизации разработки только лишь программного обеспечения (ПО), в настоящее время приобрело новый смысл, охватывающий процесс разработки сложных ИС в целом.

Теперь под термином "CASE-средства" понимаются программные средства, поддерживающие процессы создания и сопровождения ИС, включая анализ и формулировку требований, проектирование прикладного программного обеспечения (приложений) и баз данных, генерацию кода, тестирование, документирование, обеспечение качества, конфигурационное управление и управление проектом, а также другие процессы.

Появлению CASE-технологии и CASE-средств предшествовали исследования в области методологии программирования. Программирование обрело черты системного подхода с разработкой и внедрением языков высокого уровня, методов структурного и модульного программирования, средств визуального моделирования и проектирования на базе языка UML (Unified Modeling Language), средств их поддержки, формальных и неформальных языков описаний системных требований и спецификаций и т. д. Кроме того, появлению CASE-технологии способствовали и такие факторы, как:

* подготовка аналитиков и программистов, восприимчивых к концепциям модульного и структурного программирования;
* широкое внедрение и постоянный рост производительности компьютеров, позволившие использовать эффективные графические средства и автоматизировать большинство этапов проектирования;
* внедрение сетевой технологии, которая предоставила возможность объединения усилий отдельных исполнителей в единый процесс проектирования путем использования разделяемой базы данных, содержащей необходимую информацию о проекте.

CASE-средства позволяют создавать не только продукт, практически готовый к применению, но и обеспечить "правильный" процесс его разработки. Основная цель технологии - отделить проектирование программного обеспечения от его кодирования, сборки, тестирования и максимально "скрыть" от будущих пользователей все детали разработки и функционирования ПО. При этом значительно повышается эффективность работы проектировщика: сокращается время разработки, уменьшается число программных ошибок, программные модули можно использовать при следующих разработках.

**Методология** задает руководящие указания для оценки и выбора проекта разработки ПО, этапы и последовательность работ, правила применения тех или иных методов.

**Метод** - систематическая процедура или технология генерации описаний компонент ПО (например, описание потоков и структур данных).

**Нотации** предназначены для описания системы в целом, ее элементов, таких как графы, диаграммы, таблица, блок-схемы, алгоритмы, формальные языки и языки программирования.

**Средства** - технологические и программные инструменты для поддержки и усиления методов.

CASE-технологии обладают следующими основными достоинствами, которые позволяют широко использовать их при разработке информационных систем:

* ускоряют процесс коллективного проектирования и разработки;
* позволяют за короткий срок создать прототип заказанной системы с заданными свойствами;
* освобождают разработчика от рутинной работы, оставляя время для творчества;
* обеспечивают эффективность и качество разрабатываемого ПО за счет автоматизации контроля всего процесса разработки;
* поддерживают сопровождение и развитие системы на высоком уровне.

Проведение функционального и информационного обследования системы управления (административно-управленческой деятельности) предприятием ( рис. 1.1):

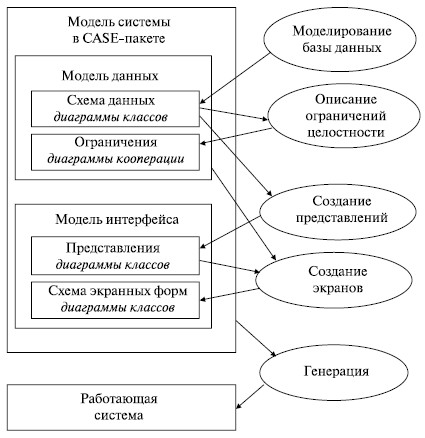


Рисунок 1.1 Модель системы в технологическом CASE-решении

Она позволяет осуществлять эффективное обучение новых работников конкретным направлениям деятельности предприятия, так как соответствующие технологии содержатся в модели.

С ее помощью можно осуществлять предварительное моделирование перспективных направлений деятельности предприятия с целью выявления новых потоков данных, взаимодействующих процессов и структурных элементов.

Современные CASE-пакеты имеют широкие возможности инструментального расширения за счет использования стандартных программных средств, что делает их чрезвычайно удобными при разработке программных и информационных систем (рис. 1.2).

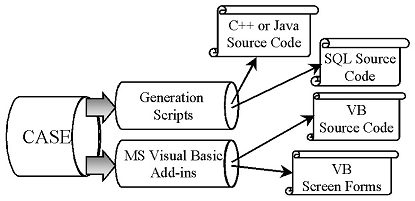


Рисунок 1.2

*Культура*. Готовность к внедрению новых процессов и взаимоотношений между разработчиками и пользователями, ИТ/ИС-управленцами и пользователями.

*Управление*. Четкое руководство и организованность по отношению к наиболее важным этапам и процессам внедрения.

*Технология*. Понимание ограниченности существующих возможностей и способность принять новую технологию.

В качестве примеров популярных CASE-средств укажем программные средства компании Computer Associates, IBM-Rational Software и Oracle:

* BPwin - моделирование бизнес-процессов;
* ERwin - моделирование баз данных и хранилищ данных;
* ERwin Examiner - проверка структуры СУБД и моделей, созданных в Erwin;
* ModelMart - среда для командной работы проектировщиков;
* Paradigm Plus - моделирование приложений и генерация объектного кода;
* Rational Rose - моделирование бизнес-процессов и компонентов приложений;
* Rational Suite AnalystStudio - пакет для аналитиков данных.

Самым мощным из указанных программных пакетов является пакет Rational Rose (RR) компании IBM-Rational.

Пакет RR способен решать практически любые задачи в проектировании информационных систем: от анализа бизнес-процессов до кодогенерации на определенном языке программирования, позволяет разрабатывать как высокоуровневые, так и низкоуровневые модели, осуществляя тем самым абстрактное либо логическое проектирование.

Таким образом, современные CASE-средства вместе с системным программным обеспечением и техническими средствами поддержки образуют полную среду разработки информационных систем.

**Лекция 2. Основные инструменты среды для создания, исполнения и управления информационной системой. Выбор средств обработки информации.**

Обработка информации

Понятие обработки информации является весьма широким. Ведя речь об обработке информации, следует дать понятие инварианта обработки. Обычно им является смысл сообщения (смысл информации, заключенной в сообщении). При автоматизированной обработке информации объектом обработки служит сообщение, и здесь важно провести обработку таким образом, чтобы инварианты преобразований сообщения соответствовали инвариантам преобразования информации.

Цель обработки информации в целом определяется целью функционирования некоторой системы, с которой связан рассматриваемый информационный процесс. Однако для достижения цели всегда приходится решать ряд взаимосвязанных задач.

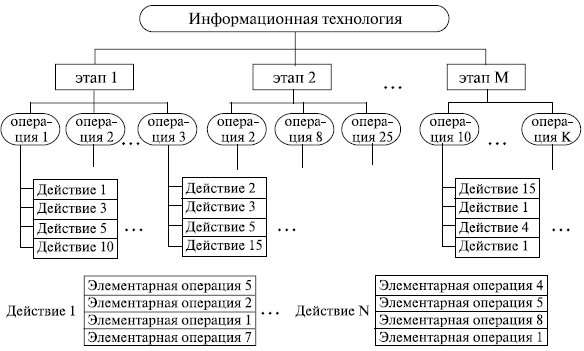


Рисунок 2.1Технологический процесс переработки информации в виде иерархической структуры по уровням

Используемые в производственной сфере такие технологические понятия, как "технологический процесс", "технологическая операция", "метрика", "норматив" и т. п. могут применяться и в ИТ. Для этого нужно начинать с определения цели. Затем следует попытаться провести структурирование всех предполагаемых действий, приводящих к намеченной цели, и выбрать необходимый программный инструментарий ( рис. 2.1).  
 1-й уровень - этапы, где реализуются базовые технологические процессы, состоящие из операций и действий последующих уровней.

2-й уровень - операции, в результате выполнения которых будет создан конкретный объект в выбранной на 1-м уровне программной среде.

3-й уровень - действия, совокупность стандартных для каждой программной среды приемов работы, приводящих к выполнению поставленной в соответствующей операции цели.

4-й уровень - элементарные операции по управлению элементарными действиями объектов.

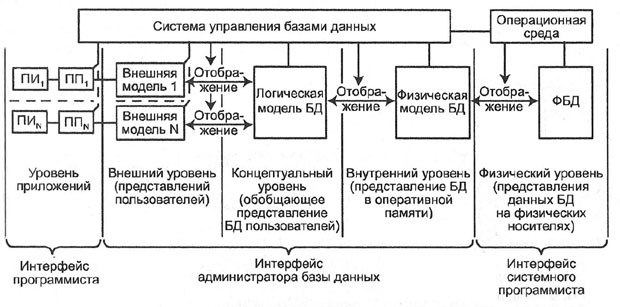
Единственный язык, напрямую выполняемый процессором, - это машинный язык (также называемый машинным кодом). Изначально все программисты прорабатывали весь алгоритм в машинном коде, но сейчас эта трудная работа уже не делается. Вместо этого программисты пишут исходный код на языке высокого уровня (например, С, С++, С#, Java), а компьютер, используя компилятор или интерпретатор и уточняя все детали, транслирует его за один или несколько этапов в машинный код, готовый к исполнению на целевом процессоре. Если требуется полный низкоуровневый контроль над системой, программисты пишут программу на языке ассемблера, мнемонические инструкции которого преобразуются один к одному в соответствующие инструкции машинного языка целевого процессора.

В некоторых языках вместо машинного кода генерируется интерпретируемый двоичный код "виртуальной машины", также называемый байт-кодом (byte-code). Такой подход применяется в языке Forth, некоторых реализациях языков Lisp, Java, Perl, Python, а также в языках платформы Microsoft .NET.

Типичный процесс разработки программ состоит, в общем, из семи этапов:

* постановка задачи;
* формализация и специфицирование;
* выбор или составление алгоритма;
* программирование;
* компиляция (трансляция);
* отладка и тестирование;
* запуск в эксплуатацию.

Эксплуатируемая программа имеет дело с данными различных типов, предназначенных для решения конкретных задач.

  
Рисунок 2.2 Общая схема базы данных

База данных, БД (Data Base) - структурированный организованный набор данных, объединенных в соответствии с некоторой выбранной моделью и описывающих характеристики какой-либо физической или виртуальной системы ( рис. 2.2).

Понятие "динамически обновляемая БД" означает, что соответствие базы данных текущему состоянию предметной области обеспечивается не периодически, а в режиме реального времени. При этом одни и те же данные могут быть по-разному представлены в соответствии с потребностями различных групп пользователей.

Система управления базами данных, СУБД (Data Base Management System) - специализированная программа или комплекс программ, предназначенные для манипулирования базой данных. Для создания информационной системы и управления ею СУБД необходима в той же степени, как для разработки программы на алгоритмическом языке необходим транслятор.

Организация структуры БД формируется исходя из следующих соображений:

* адекватность описываемому объекту/системе - на уровне концептуальной и логической моделей;
* удобство использования для ведения учета и анализа данных - на уровне так называемой физической модели.

Виды концептуальных и логических моделей БД:

* картотеки;
* сетевые;
* иерархические;
* реляционные;
* дедуктивные;
* объектно-ориентированные;
* многомерные.

Виды концептуальных и логических моделей БД:

* картотеки;
* сетевые;
* иерархические;
* реляционные;
* дедуктивные;
* объектно-ориентированные;
* многомерные.

SQL основывается на реляционной алгебре. Язык SQL делится на три части:

* операторы определения данных;
* операторы манипуляции данными (Insert, Select, Update, Delete);
* операторы определения доступа к данным.

Основные функции системы управления базами данных:

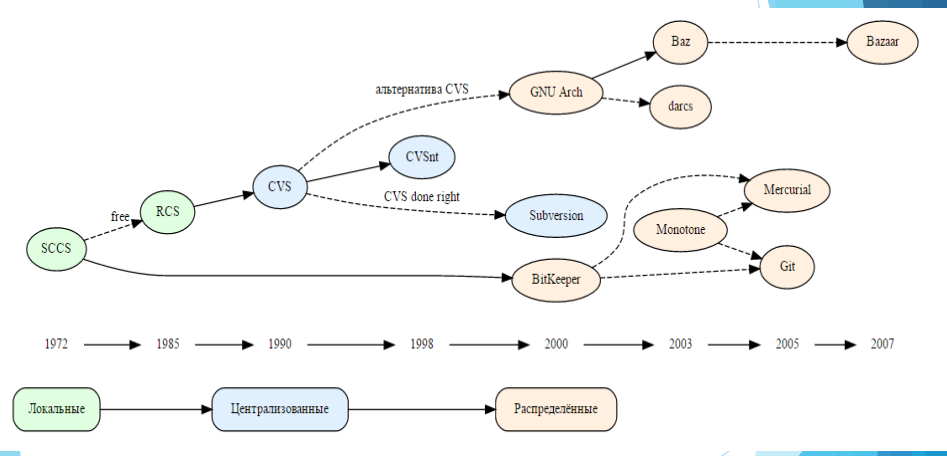
* управление данными во внешней памяти (на различных носителях);
* управление данными в оперативной памяти;
* журналирование изменений и восстановление базы данных после сбоев;
* поддержка языков БД (язык определения данных, язык манипулирования данными, язык определения доступа к данным).

**Лекция 3. Организация работы в команде разработчиков. Система контроля версий: совместимость, установка, настройка.**Система контроля версий (Version Control System, VCS) — программное обеспечение для облегчения работы с изменяющейся информацией.

VCS позволяет хранить несколько версий одного и того же документа, при необходимости возвращаться к более ранним версиям, определять, кто и когда сделал то или иное изменение, и многое другое.

Для чего нужны VCS:

* Хранение полной истории изменений
* Описание причин всех производимых изменений
* Откат изменений, если что-то пошло не так
* Поиск причины и ответственного за появления ошибок в программе
* Совместная работа группы над одним проектом
* Возможность изменять код, не мешая работе других пользователей

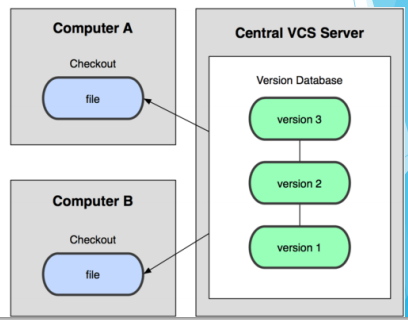


Репозиторий - хранилище версий - в нем хранятся все документы вместе с историей их изменения и другой служебной информацией.

Рабочая копия - копия проекта, связанная с репозиторием.

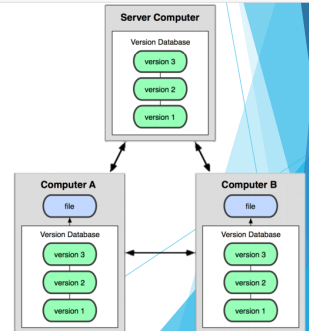
Централизованные VCS

Одно основное хранилище всего проекта.Каждый пользователь копирует себе необходимые ему файлы из этого репозитория, изменяет и, затем, добавляет свои изменения обратно



Децентрализованные VCS

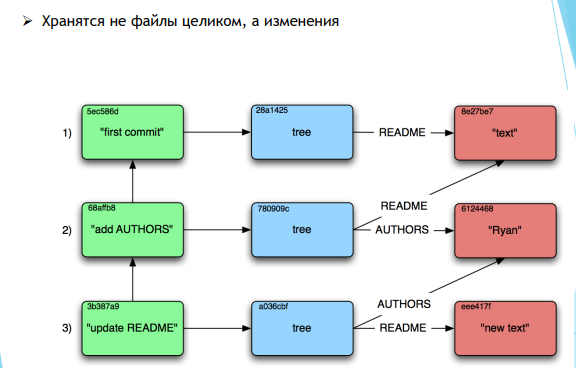
У каждого пользователя свой вариант (возможно не один) репозитория. Присутствует возможность добавлять и забирать изменения из любого репозитория.



Терминология VCS2



**Изменения**



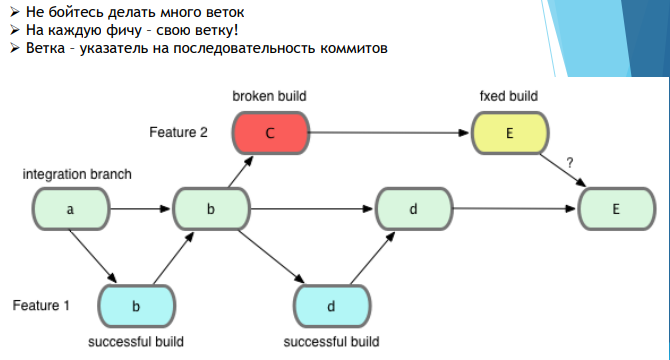
**Редактирование**



**Тегирование**



**Ветвление**



**Основы Git**

Git - распределенная система контроля версий. Первая версия появилась в 2005 г. Разработана Линусом Торвальдсом для использования в управлении разработкой ядра Linux.  
Для минимальной настройки Git на компьютере необходимо задать глобальные параметры, которые будут применяться к вносимым изменениям и подписывать их. Они будут указывать на Вас в истории коммитов в удаленных репозиториях.

Такими глобальными настройками являются имя пользователя и его email. Их можно установить следующими командами в консоли Git:

$ git config --global user.name “Your name"

$ git config --global user.email [email@example.com](mailto:email@example.com)

Все параметры будут помещены в файл с настройками Git .gitconfig, расположенным в домашнем каталоге пользователя.

**Лекция 4. Обеспечение кроссплатформенности информационной системы.**

Кроссплатформенность (межплатформенность) — способность программного обеспечения работать с несколькими аппаратными платформами или операционными системами. Обеспечивается благодаря использованию высокоуровневых языков программирования, сред разработки и выполнения, поддерживающих условную компиляцию, компоновку и выполнение кода для различных платформ. Типичным примером является программное обеспечение, предназначенное для работы в операционных системах Linux и Windows одновременно.

Кроссплатформенными можно назвать большинство современных высокоуровневых языков программирования. Например, Си, С++, Free Pascal, FreeBASIC, PureBasic — кроссплатформенные языки на уровне компиляции, то есть для этих языков есть компиляторы под различные платформы. Это позволяет — при надлежащем качестве кода — не переписывать основной движок программы, меняются только особые системозависимые части.

Не менее важны для кроссплатформенности стандартизованные библиотеки среды выполнения. В частности, стандартом стала библиотека языка Си (POSIX). Из крупных кроссплатформенных библиотек — Qt, GTK+, FLTK, STL, Boost, OpenGL, SDL, OpenAL, OpenCL.

Существуют кросскомпиляторы — компиляторы, генерирующие исполняемый код для платформы, отличной от той, на которой запущен сам компилятор.

Кроссплатформенные среды исполнения

PHP, Perl, Python, Tcl и Ruby — кроссплатформенные интерпретируемые языки, их интерпретаторы существуют для многих платформ.

Среды исполнения ActionScript Virtual Machine, Java Virtual Machine и .NET также кроссплатформенны, однако на их вход подаётся не исходный текст, а промежуточный код. Поэтому программы, написанные на ActionScript, Java и C#, можно запускать под разными операционными системами без предварительной перекомпиляции.

Кроссплатформенный пользовательский интерфейс

На разных операционных системах — независимо от того, как технически достигнута работа в них — стандартные элементы интерфейса имеют разные размеры. Поэтому простое жёсткое позиционирование элементов интерфейса невозможно — под другой операционной системой они могут «налезать» друг на друга. Существует несколько подходов:

1. Единый стиль, общий для всех операционных систем, программы выглядят одинаково под всеми системами. Так работают интерфейсные библиотеки Java наподобие Swing.
   * Плюс: можно жёстко расставлять элементы управления на манер Delphi; оригинальный стиль.
   * Минус: системе приходится иметь свои экранные шрифты; стиль отличается от стиля ОС.
2. Самоадаптирующийся интерфейс, подстраивающий сетку под реальные размеры элементов управления. Типичные примеры — Qt, wxWidgets, XUL.
   * Плюс: стандартный стиль операционной системы, очень быстрый и «скинующийся» под Windows XP, Vista и 7; некоторая автоматизация локализации.
   * Минус: чтобы собрать самоадаптирующуюся сетку, требуется квалифицированный программист; затруднена плотная компоновка.
3. Гибридный подход реализован в GTK+.
   * Плюс: шрифты можно брать из системы, а не «тащить» свои; некоторая автоматизация локализации.
   * Минус: берёт все недостатки от первых двух подходов. Стиль отличается от стиля операционной системы; затруднена плотная компоновка.

В любом случае, под другими операционными системами требуется хотя бы минимальное тестирование, так как возможны ошибки компоновки.

Если вкратце, то кроссплатформенность — это способность программного обеспечения работать с несколькими аппаратными платформами или операционными системами. Нативные программы в свою очередь пишутся для работы на определенной аппаратной или программной платформе. Из этого можно сделать вывод: если ваша программа будет работать с железом, то скорее всего даже в кроссплатформенной программе вам придется прописывать нативный функционал для конкретной платформы.  
 **Electron JS** — это JavaScript фреймворк, позволяющий вам написать полноценную программу используя Web Технологии — HTML & CSS, JS. Electron JS представляет собой движок Chromium в котором и исполняется весь ваш код. У Electron'a есть один, но достаточно серьезный недостаток — большое потребление памяти, как физической, так и оперативной.

Так же как и Electron, **NW.JS** позволяет вам создавать кроссплатформенные приложения с использованием Web технологий. Сегодня NW.JS Спонсируется компанией Intel, и разрабатывается сообществом. NW.JS не может похвастаться таким же богатым списком проектов как Electron, но это все равно очень хороший фреймворк для создания кроссплатформенных приложений. Так же как и Electron, NW.JS тащит за собой движок Chromium и все вытекающие из него проблемы. NW.JS очень похож по своему принципу на Electron, но все-же различия есть. Могу порекомендовать вам эту статью для ознакомления.

**Qt** — очень мощный набор инструментов для создания кроссплатформенных приложений на языке С++ и Python (а также Ruby, PHP, C# и др. но они в отличии поддерживаются сообществом). Qt уже достаточно старый фреймворк, но он продолжает активно развиваться и на нем написаны такие программы как: 2ГИС для Android, Kaspersky Internet Security, Virtual Box, Skype, VLC Media Player, Opera и другие. Известное многим линукс-юзерам окружение рабочего стола KDE тоже написано с использованием Qt. Qt имеет среду разработки — Qt Creator, которая включает в себя Qt Designer с помощью которого можно создавать графический интерфейс.

**wxWidgets** (ранее известная как wxWindows) — это кросс-платформенная библиотека инструментов с открытым исходным кодом для разработки кроссплатформенных на уровне исходного кода приложений, в частности для построения графического интерфейса пользователя (GUI). Она разработана не только для того, чтобы создавать GUI. Она также имеет набор классов для работы с графическими изображениями, HTML, XML документами, архивами, файловыми системами, процессами, подсистемами печати, мультимедиа, сетями, классы для организации многопоточности, отладки, отправки дампов и множество других инструментов.

**Лекция 5. Особенности объектно-ориентированных и структурных языков программирования.**

Объектно-ориентированное программирование.

Технология программирования - это совокупность методов и средств разработки (написания) программ и порядок применения этих методов и средств.

Основные идеи объектно-ориентированного подхода опираются на следующие положения:

* Программа представляет собой модель некоторого реального процесса, части реального мира.
* Модель реального мира или его части может быть описана как совокупность взаимодействующих между собой объектов.
* Объект описывается набором параметров, значения которых определяют состояние объекта, и набором операций (действий), которые может выполнять объект.
* Взаимодействие между объектами осуществляется посылкой специальных сообщений от одного объекта к другому. Сообщение, полученное объектом, может потребовать выполнения определенных действий, например, изменения состояния объекта.
* Объекты, описанные одним и тем же набором параметров и способные выполнять один и тот же набор действий представляют собой класс однотипных объектов.

В основу ООП положены следующие принципы:

**Абстрагирование** – процесс выделения абстракций в предметной области задачи. Абстракция – совокупность существенных характеристик некоторого объекта, которые отличают его от всех других видов объектов и, таким образом, четко определяют особенности данного объекта с точки зрения дальнейшего рассмотрения и анализа.

Применяемая абстракция реального предмета существенно зависит от решаемой задачи: в одном случае нас будет интересовать форма предмета, в другом вес, в третьем – материалы, из которых он сделан, в четвертом – закон движения предмета и т.д.

Современный уровень абстракции предполагает объединение всех свойств абстракции (как касающихся состояния анализируемого объекта, так и определяющих его поведение) в единую программную единицу некий абстрактный тип (класс).

**Ограничение доступа** – сокрытие отдельных элементов реализации абстракции, не затрагивающих существенных характеристик ее как целого.

Необходимость ограничения доступа предполагает разграничение двух частей в описании абстракции:

* интерфейс – совокупность доступных извне элементов реализации абстракции (основные характеристики состояния и поведения);
* реализация – совокупность недоступных извне элементов реализации абстракции (внутренняя организация абстракции и механизмы реализации ее поведения).

Ограничение доступа в ООП позволяет разработчику:

* выполнять конструирование системы поэтапно, не отвлекаясь на особенности реализации используемых абстракций;
* легко модифицировать реализацию отдельных объектов, что в правильно организованной системе не потребует изменения других объектов.

Сочетание объединения всех свойств предмета (составляющих его состояния и поведения) в единую абстракцию и ограничения доступа к реализации этих свойств получило название *инкапсуляции*.

**Модульность** – принцип разработки программной системы, предполагающий реализацию ее в виде отдельных частей (модулей). При выполнении декомпозиции системы на модули желательно объединять логически связанные части, по возможности обеспечивая сокращение количества внешних связей между модулями.

**Иерархия** – ранжированная или упорядоченная система абстракций. Принцип иерархичности предполагает использование иерархий при разработке программных систем.

В ООП используются два вида иерархии:

* иерархия «целое/часть» – показывает, что некоторые абстракции включены в рассматриваемую абстракцию как ее части, например, лампа состоит из цоколя, нити накаливания и колбы. Этот вариант иерархии используется в процессе разбиения системы на разных этапах проектирования;
* иерархия «общее/частное» – показывает, что некоторая абстракция является частным случаем другой абстракции, например, «обеденный стол – конкретный вид стола», а «столы – конкретный вид мебели». Используется при разработке структуры классов, когда сложные классы строятся на базе более простых путем добавления к ним новых характеристик и, возможно, уточнения имеющихся.

Один из важнейших механизмов ООП – наследование свойств в иерархии общее/частное. Наследование – такое соотношение между абстракциями, когда одна из них использует структурную или функциональную часть другой или нескольких других абстракций (соответственно простое и множественное наследование).

**Типизация** – ограничение, накладываемое на свойства объектов и препятствующее взаимозаменяемости абстракций различных типов (или сильно сужающее возможность такой замены).

Использование принципа типизации обеспечивает:

* раннее обнаружение ошибок, связанных с недопустимыми операциями над программными объектами (ошибки обнаруживаются на этапе компиляции программы при проверке допустимости выполнения данной операции над программным объектом);
* упрощение документирования;
* возможность генерации более эффективного кода.

Тип может связываться с программным объектом статически (тип объекта определен на этапе компиляции – раннее связывание) и динамически (тип объекта определяется только во время выполнения программы – позднее связывание). Реализация позднего связывания в языке программирования позволяет создавать переменные-указатели на объекты, принадлежащие различным классам (полиморфные объекты), что существенно расширяет возможности языка.

**Параллелизм** – свойство нескольких абстракций одновременно находиться в активном состоянии, т.е. выполнять некоторые операции.

Существует целый ряд задач, решение которых требует одновременного выполнения некоторых последовательностей действий. К таким задачам, например, относятся задачи автоматического управления несколькими процессами.

Реальный параллелизм достигается только при реализации задач такого типа на многопроцессорных системах, когда имеется возможность выполнения каждого процесса отдельным процессором. Системы с одним процессором имитируют параллелизм за счет разделения времени процессора между задачами управления различными процессами.

**Устойчивость** – свойство абстракции существовать во времени независимо от процесса, породившего данный программный объект, и/или в пространстве, перемещаясь из адресного пространства, в котором он был создан.

Язык считается объектно-ориентированным, если в нем реализованы первые четыре из рассмотренных семи принципов.

***C++ (си-плас-плас)***

[Язык программирования C++](https://bourabai.ru/C-Builder/cpp/index.htm) был разработан Бьерном Страустрапом, сотрудником AT&T Bell Laboratories. Непосредственным предшественником C++ является С with Classes, созданный тем же автором в 1980 году. C++ - это в значительной степени надстройка над С. В определенном смысле можно назвать C++ улучшенным С, тем С, который обеспечивает контроль типов, перегрузку функций и ряд других удобств. Но главное в том, что C++ добавляет к С объектную ориентированность.

Известны [несколько версий C++](https://bourabai.ru/C-Builder/cpp/index.htm). В версии 1.0 реализованы основные механизмы объектно-ориентированного программирования, такие как одиночное наследование и полиморфизм, проверка типов и перегрузка функций.   
***C# (си-шарп)***

C# — объектно-ориентированный язык программирования. Разработан в 1998—2001 годах группой инженеров под руководством Андерса Хейлсберга в компании Microsoft как язык разработки приложений для платформы Microsoft .NET Framework.

C# относится к семье языков с C-подобным синтаксисом, из них его синтаксис наиболее близок к C++ и [Java](https://bourabai.ru/alg/java.htm). Язык имеет статическую типизацию, поддерживает полиморфизм, перегрузку операторов (в том числе операторов явного и неявного приведения типа), делегаты, атрибуты, события, свойства, обобщённые типы и методы, итераторы, анонимные функции с поддержкой замыканий, LINQ, исключения, комментарии в формате XML.

Переняв многое от своих предшественников — языков C++, Pascal, Модула, Smalltalk и в особенности [Java](https://bourabai.ru/alg/java.htm) — С#, опираясь на практику их использования, исключает некоторые модели, зарекомендовавшие себя как проблематичные при разработке программных систем, например, C# в отличие от C++ не поддерживает множественное наследование классов (между тем допускается множественное наследование интерфейсов).

***Java***

Объектно-ориентированный язык программирования [Java](https://bourabai.ru/alg/java.htm), ориентированный на сети компьютеров и, прежде всего, на Internet. [Java](https://bourabai.ru/alg/java.htm) - интерпретируемый на конечной стадии язык: для него определены внутреннее представление (прекомпиляция в байткод - bytecode) и постинтерпретатор этого представления на целевой машине, которые уже сейчас реализованы на большинстве платформ. Постинтерпретатор (виртуальная Java-машина) упрощает отладку программ, написанных на языке [Java](https://bourabai.ru/alg/java.htm), обеспечивает их переносимость на новые платформы и адаптируемость к новым окружениям. Он позволяет исключить влияние программ, написанных на языке [Java](https://bourabai.ru/alg/java.htm), на другие программы и файлы, имеющиеся на новой платформе, и тем самым обеспечить безопасность выполнения этих программ для операционной среды (не нарушать ее работоспособность, что совершенно не определяет безопасности на прагматическом, человеческом уровне, которая как раз страдает от возможности внедрения чужеродных Java-кодов в ОС: троянов, червей и пр.). Положительные свойства [технологии Java](https://bourabai.ru/alg/java.htm) позволяют использовать ее для программ, распространяемых по сетям (в частности, по сети Internet).

**Лекция 6. Разработка сценариев с помощью специализированных языков.**

Множество Мандельброта определено на плоскости комплексных чисел (представленной, как система координат (x,y)). Для каждого комплексного числа c должна существовать ограниченная последовательность:

zk+1 = zk2 + c

Если для числа c последовательность является ограниченной, это число находится в множестве Мандельброта. Если же последовательность является неограниченной, число c не находится в множестве Мандельброта.

При расчете мы будем осуществлять 50-кратную итерацию в рамках последовательности и в том случае, если значение zk + 1 будет покидать окружность радиусом 2 вокруг значения c, мы будем делать вывод о том, что число c не находится в рассматриваемом множестве.

Наша система координат будет иметь ось x с координатами от -2 до 1 и ось y с координатами от -1 до 1. Мы будем отображать ее с помощью ASCII-символов начиная с верхнего левого угла (с координатами x=-2, y=1), причем вывод результатов будет заключаться в последовательном выводе строк по оси y. Каждый ASCII-символ (пробел или \*) будет соответствовать сдвигу на 0.05 по оси координат. Вы сможете самостоятельно заменить жестко заданные значения для того, чтобы оценить полученные в результате изменения или добавить поддержку цветов для того, чтобы визуально усложнить вывод.  
 Язык Perl все еще активно используется и, без сомнения, будет актуален в течение долгого времени. Однако, в последнее время он утратил свою былую популярность, особенно в рамках новых проектов. "Всегда существует более одного способа выполнения конкретной задачи" - девиз разработчиков языка Perl, доказывающий его беспрецедентную гибкость, которая является как его главным достоинством, так и его главным недостатком.

Функции в JavaScript обозначаются с помощью ключевого слова function, причем вы можете передавать в функции множество параметров одновременно.

Как и в Python, в JavaScript переменные не должны иметь четко указанных типов, но при этом они должны объявляться с помощью ключевого слова var перед использованием (хотя это и не обязательно).

Циклы for работают практически так же, как и в двух описанных выше языках сценариев. Так как в JavaScript нет библиотеки для работы с комплексными числами, нам придется обрабатывать их действительные и мнимые части по отдельности. Инструкция document.write не позволяет выводить символы пробелов, поэтому вместо них в данном сценарии осуществляется вывод символов обратных кавычек.

Если вам хочется использовать все возможности HTML для вывода более привлекательной версии фрактала (не на основе ASCII-символов), вы также можете поискать соотвествующий код на веб-сайте Rosetta Code (rosettacode.org). И, как в случае с языками Perl и Python, существует огромное количество сетевых ресурсов с информацией, необходимой для изучения дополнительных возможностей JavaScript.

Функции обозначаются с помощью ключевого слова def. В рамках циклов for используется ключевое слово range. В рамках диапазонов, объявляемых с помощью ключевого слова ranage, должны использоваться целочисленные значения, но, как и в данном случае, при необходимости использования цикла с нецелочисленными значениями, вы можете умножить целочисленное значение из диапазона на нецелочисленное и использовать результат в рамках цикла.

Так как интерпретатор Python автоматически добавляет символ новой строки при использовании функции print, для получения вывода без символов новой строки нам придется использовать метод объекта строки join. Обратите внимание на циклы for в рамках инструкции print/join, а также методику использования условной инструкции if not ... else. В данном случае для сохранения результатов работы функции mandelbrot и объединения их в рамках списка используется абстракция списков.

Как и в случае языка программирования Perl, мы импортируем библиотеку (в данном случае это библиотека math) для работы с комплексными числами. В Python x + yi является аналогом x + y \* 1j.

**Лекция 7. Интегрированные среды разработки для создания независимых программ.**

Интегрированная среда разработки (ИСР) – это система программных средств, используемая программистам для разработки программного обеспечения. В английском языке такая среда называется Integrated development environment или сокращённо IDE.

ИСР обычно включает в себя текстовый редактор, компилятор, интерпретатор, средства автоматизации разработки и сборки программного обеспечения и отладчик. Иногда также содержит средства для интеграции с системами управления версиями и разнообразные инструменты для упрощения конструирования графического интерфейса пользователя. Многие современные среды разработки также включают окно просмотра программных классов, инспектор объектов и диаграмму иерархии классов – для использования при объектно-ориентированной разработке ПО. Большинство современных ИСР предназначенны для разработки программ на нескольких языках программирования одновременно.

Один из частных случаев ИСР – среды визуальной разработки, которые включают в себя возможность визуального редактирования интерфейса программы.

Основным окном, является текстовый редактор, который используется для ввода исходного кода в ИСР и ориентирован на работу с последовательностью символов в текстовых файлах. Такие редакторы обеспечивают расширенную функциональность – подсветку синтаксиса, сортировку строк, шаблоны, конвертацию кодировок, показ кодов символов и т. п. Иногда их называют редакторами кода, так как основное их предназначение – написание исходных кодов компьютерных программ.

Подсветка синтаксиса – выделение синтаксических конструкций текста с использованием различных цветов, шрифтов и начертаний. Обычно применяется в текстовых редакторах для облегчения чтения исходного текста, улучшения визуального восприятия. Часто применяется при публикации исходных кодов в Интернет.

Понятие трансляции, компилятора и интерпретатора было дано в предыдущих лекциях.

Одна из наиболее важных частей ИСР – отладчик, который представляет собой модуль среды разработки или отдельное приложение, предназначенное для поиска ошибок в программе. Отладчик позволяет выполнять пошаговую трассировку, отслеживать, устанавливать или изменять значения переменных в процессе выполнения программы, устанавливать и удалять контрольные точки или условия остановки и т. д.

Наиболее распространёнными отладчиками являются:

* GNU Debugger – отладчик программ от проекта GNU;
* IDA – дизассемблер и низкоуровневый отладчик для операционных систем семейства Windows и GNU/Linux;
* Microsoft Visual Studio – среда разработки программного обеспечения, включающая средства отладки от корпорации Microsoft;
* OllyDbg – бесплатный низкоуровневый отладчик для операционных систем семейства Windows;
* SoftICE – низкоуровневый отладчик для операционных систем семейства Windows;
* Dr. Watson – стандартный отладчик Windows, позволяет создавать дампы памяти;
* WinDbg – бесплатный отладчик от корпорации Microsoft.

Основным процессом отладки является трассировка. Трассировка – это процесс пошагового выполнения программы. В режиме трассировки программист видит последовательность выполнения команд и значения переменных на данном шаге выполнения программы, что позволяет легче обнаруживать ошибки. Трассировка может быть начата и окончена в любом месте программы, выполнение программы может останавливаться на каждой команде или на точках останова, трассировка может выполнятся с заходом в процедуры/функции и без заходов.

branch (ветвь) – направление разработки, независимое от других. Ветвь представляет собой копию части (как правило, одного каталога) хранилища, в которую можно вносить свои изменения, не влияющие на другие ветви. Документы в разных ветвях имеют одинаковую историю до точки ветвления и разные – после неё.

check-in, commit, submit – создание новой версии, публикация изменений. Распространение изменений, сделанных в рабочей копии, на хранилище документов. При этом в хранилище создаётся новая версия изменённых документов.

Check-out, clone – извлечение документа из хранилища и создание рабочей копии.

Conflict – конфликтная ситуация, когда несколько пользователей сделали изменения одного и того же участка документа. Конфликт обнаруживается в случае, когда один пользователь уже опубликовал свои изменения, а второй только пытается их опубликовать и система сама не может корректно слить конфликтующие изменения. Поскольку программа может быть недостаточно разумна для того, чтобы определить, какое изменение является «корректным», второму пользователю нужно самому разрешить конфликт (resolve).

Merge, integration (слияние) - объединение независимых изменений в единую версию документа. Осуществляется, когда два человека изменили один и тот же файл или при переносе изменений из одной ветки в другую.

Repository (хранилище документов) - место, где система управления версиями хранит все документы вместе с историей их изменения и другой служебной информацией.

Revision (версия документа). Системы управления версиями различают версии по номерам, которые назначаются автоматически.

Tag, label (метка) – которую можно присвоить определённой версии документа. Метка представляет собой символическое имя для группы документов, причём описывает она не только набор имён файлов, но и ревизию каждого файла. Ревизии включённых в метку документов могут принадлежать разным моментам времени.

Trunk, mainline (ствол) – основная ветвь разработки проекта. Политика работы со стволом может отличаться от проекта к проекту, но в целом она такова: большинство изменений вносится в ствол; если требуется серьёзное изменение, способное привести к нестабильности, создаётся ветвь, которая сливается со стволом, когда нововведение будет в достаточной мере испытано; перед выпуском очередной версии создаётся «релизная» ветвь, в которую вносятся только исправления.

Update, sync (обновление, синхронизация) – синхронизация рабочей копии до некоторого заданного состояния хранилища. Чаще всего это действие означает обновление рабочей копии до самого свежего состояния хранилища. Однако при необходимости можно синхронизировать рабочую копию и к более старому состоянию, чем текущее.

Working copy (рабочая копия) – рабочая (локальная) копия документов.

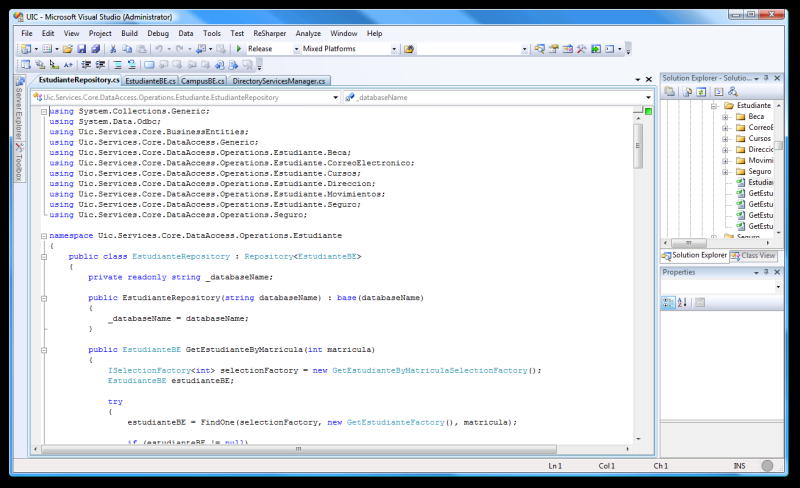
Рассмотрим возможности ИСР на примере наиболее доступных и популярных версий.

Eclipse (от англ. затмение) – свободная интегрированная среда разработки модульных кроссплатформенных приложений (рисунок 69). Развивается и поддерживается некоммерческой организацией Eclipse Foundation

Microsoft Visual Studio включает один или несколько компонентов из следующих: Visual Basic.NET, Visual C++, Visual C#, Visual F#, Microsoft SQL Server, Visual InterDev, Visual J++, Visual J#, Visual FoxPro, Visual Source Safe.

Одним из главных преимуществ Майкрософт Визуал Студия является высокое качество документирования процесса разработки и описания возможных проблем в MSDN Library. Однако наиболее интересная для профессионала часть, посвящённая тонкостям разработки, существует только на английском языке.

Также компания «Майкрософт» предлагает бесплатный аналог продукта Visual Studio Express



**Лекция 8. Построение архитектуры проекта.**

**Эффективность системы**. В первую очередь программа, конечно же, должна решать поставленные задачи и хорошо выполнять свои функции, причем в различных условиях. Сюда можно отнести такие характеристики, как надежность, безопасность, производительность, способность справляться с увеличением нагрузки (масштабируемость) и т.п.

**Гибкость системы**. Любое приложение приходится менять со временем — изменяются требования, добавляются новые. Чем быстрее и удобнее можно внести изменения в существующий функционал, чем меньше проблем и ошибок это вызовет — тем гибче и конкурентоспособнее система. Поэтому в процессе разработки старайтесь оценивать то, что получается, на предмет того, как вам это потом, возможно, придется менять.

**Расширяемость системы**. Возможность добавлять в систему новые сущности и функции, не нарушая ее основной структуры. На начальном этапе в систему имеет смысл закладывать лишь основной и самый необходимый функционал (принцип YAGNI — you ain’t gonna need it, «Вам это не понадобится») Но при этом архитектура должна позволять легко наращивать дополнительный функционал по мере необходимости. Причем так, чтобы внесение наиболее вероятных изменений требовало наименьших усилии.

**Масштабируемость процесса разработки**. Возможность сократить срок разработки за счёт добавления к проекту новых людей. Архитектура должна позволять распараллелить процесс разработки, так чтобы множество людей могли работать над программой одновременно.

**Тестируемость**. Код, который легче тестировать, будет содержать меньше ошибок и надежнее работать. Но тесты не только улучшают качество кода. Многие разработчики приходят к выводу, что требование «хорошей тестируемости» является также направляющей силой, автоматически ведущей к хорошему дизайну, и одновременно одним из важнейших критериев, позволяющих оценить его качество.

Существует целая методология разработки программ на основе тестов, которая так и называется — Разработка через тестирование (Test-Driven Development, TDD).

**Возможность повторного использования**. Систему желательно проектировать так, чтобы ее фрагменты можно было повторно использовать в других системах.

**Хорошо структурированный, читаемый и понятный код**. Сопровождаемость. Над программой, как правило, работает множество людей — одни уходят, приходят новые. После написания сопровождать программу тоже, как правило, приходится людям, не участвовавшем в ее разработке. Поэтому хорошая архитектура должна давать возможность относительно легко и быстро разобраться в системе новым людям. Проект должен быть хорошо структурирован, не содержать дублирования, иметь хорошо оформленный код и желательно документацию. И по возможности в системе лучше применять стандартные, общепринятые решения привычные для программистов.  
 Фасад — это объект-интерфейс, аккумулирующий в себе высокоуровневый набор операций для работы с некоторой подсистемой, скрывающий за собой ее внутреннюю структуру и истинную сложность. Обеспечивает защиту от изменений в реализации подсистемы. Служит единой точкой входа — "вы пинаете фасад, а он знает, кого там надо пнуть в этой подсистеме, чтобы получить нужное".

Таким образом, мы получаем первый, самый важный паттерн, позволяющий использовать концепцию интерфейсов при проектировании модулей и тем самым ослаблять их связанность — «Фасад». Помимо этого «Фасад» вообще дает возможность работать с модулями точно также как с обычными объектами и применять при проектировании модулей все те полезные принципы и техники, которые используются при проектирования классов.

Ядро-посредник может как знать о модулях-клиентах и управлять ими (пример — архитектура apache ), так и может быть полностью, или почти полностью, независимым и ничего о клиентах не знать. В сущности именно этот подход реализован в «шаблоне» Модель-Вид-Контроллер (MVC), где с одной Моделью (являющейся ядром приложение и общим хранилищем данных) могут взаимодействовать множество Пользовательских Интерфейсов, которые работают синхронно и при этом не знают друг о друге, а Модель не знает о них. Ничто не мешает подключить к общей модели и синхронизировать таким образом не только интерфейсы, но и другие вспомогательные модули.

**Лекция 9. Определение конфигурации информационной системы. Выбор технических средств.**

Конфигурирование - процесс настройки ИС с целью их адаптации к специфике области внедрения.

В процессе конфигурирования ИС выполняются следующие действия:

* изменение объектной модели;
* определение авторизаций пользователей;
* настройка интерфейса;
* создание типовых объектов данных: справочников, шаблонов, отчетов, процессов и т. д.;
* настройка вариантов развертывания и взаимодействия программных компонентов, определение параметров системных служб, сервисов ит. д.

Таким образом, на каждом этапе конфигурирования выполняется работа по созданию или изменению определенной части информационного и программного обеспечения ИС.

**Этапы и объекты конфигурирования ИС**

Соответствие этапов конфигурирования ИС и обрабатываемых при этом объектов показано в табл.

|  |  |
| --- | --- |
| Этап конфигурирования | Объект конфигурирования |
| Изменение объектной модели | Объектная модель (модель данных) ИС |
| Определение авторизаций пользователей | Пользователи |
| Настройка интерфейса | Графический интерфейс: формы ввода, пункты меню, карточки объектов |
| Создание типовых объектов данных | Справочники, отчеты, шаблоны, диаграммы процессов |
| Расширение функциональных возможностей | Программная реализация функциональных модулей и дополнительных возможностей, в т. ч. интеграции с другими ИС (ПО) |
| Настройка вариантов развертывания и взаимодействия программных | Параметры системных служб и сервисов |

Модель данных ИС (объектная модель) - главный объект конфигурирования, смысловое ядро системы, которое реализует ее бизнес- логику и определяет поведение остальных объектов конфигурации.

Объектная модель ИС обладает следующим характеристиками:

* отражает важные для разрабатываемой системы понятия и объекты реального мира, прагматику прикладной области (терминология и смысл, назначение объектов);
* описывает структуру объектов, составляющих систему, их атрибуты, операции и взаимовлияние;
* является абстрактным понятием;
* отображается в виде графических схем формальных нотаций (UML, IDEF и т. д.);
* на практике реализуется в виде программного обеспечения ИС.

В настоящее время объектную модель ИС принято представлять в виде UML-диаграммы классов. При практической настройке модели, особенно в корпоративных ИС, понятие класс заменяется термином «справочник».

Справочник - хранилище сущностей в едином контексте, совокупность объектов с однотипным набором характеристик и методов их обработки, а также шаблонами визуализации.

Контекст (смысловое наполнение) определяет структуру представления данных в виде справочника. Таким образом, справочник представляет собой класс, а элементы (записи) справочника- объекты класса. Заданная в справочнике структура представления данных определяет хранящиеся в нем объекты. В наглядном виде хранение данных реализуется с помощью графических форм, характерных для справочника и его отдельного элемента. Посредством таких графических форм реализуется доступ к данным, определенный авторизациями пользователей.

**Лекция 10. Формирование репозитория проекта, определение уровня доступа в системе контроля версий. Распределение ролей.**

Существует несколько моделей хранения данных:

*Примитивная модель хранения версий*

В примитивной модели актуальные копии проекта перезаписываются в отдельную директорию через определённый промежуток времени.

Достоинства:

* возможность восстановления данных одной из записанных версий.

Недостатки:

* сложности с поиском необходимой версии в обширной и плохо структурированной базе данных;
* возможность потери данных вследствие возникновения физических поломок оборудования;
* отсутствие возможности совместной разработки.

*Локальные системы контроля версий*

Локальные СКВ обычно хранят на компьютере список изменений, внесенных в файлы. Основываясь на этих данных, система контроля версий воссоздает нужную версию файла (актуальную на определенный момент времени).

Достоинства:

* возможность восстановления данных из определенной версии (точно определяется по времени записи);
* высокая скорость выполнения восстановления (база данных четко структурирована, поэтому сложностей при поиске не возникает, сетевая задержка отсутствует, поскольку данные хранятся непосредственно на рабочем компьютере).

Недостатки:

* возможность потери данных вследствие возникновения физических поломок оборудования;
* отсутствие возможности совместной разработки.

*Централизованные системы контроля версий*

Централизованные системы контроля версий предполагают сохранение версий проектов на общий сервер, с которого потом получают нужные версии клиенты.

Достоинства:

* возможность восстановления данных из определенной версии (точно определяется по времени записи);
* возможность ведения командной разработки проекта;

Недостатки:

* отсутствие доступа к данным при сбое работы сервера;
* довольно низкая скорость работы (из-за возникновения сетевых задержек).

*Децентрализованные системы контроля версий*

В децентрализованных системах контроля версий при каждом копировании удалённого репозитория (расположенного на сервере) происходит полное копирование данных в локальный репозиторий (установленный на рабочем компьютере). Каждая копия содержит все данные, хранящиеся в удалённом репозитории. В случае, возникновения технической неисправности на стороне сервера, удаленный репозиторий можно перезаписать с любой сохраненной копии.

Достоинства:

* возможность восстановления данных из определенной версии (точно определяется по времени записи);
* возможность ведения командной разработки проекта;
* при сбое работы сервера система сохраняет данные в локальном репозитории, что позволяет эффективно вести процесс разработки, а после восстановления работы сервера, передать все изменения в удаленный репозиторий;
* при физической поломке сервера данные можно легко перенести в новый удалённый репозиторий с любого локального репозитория;
* высокая скорость работы (в ходе работы данные записываются и получаются из локального репозитория, поэтому сетевые задержки отсутствуют).

**Современные системы контроля версий**

Существует много систем контроля версий (Git, Darcs, Mercurial, Bazaar, Monotone и т.д), сходных по принципу работы и конечным задачам. Отличаются они друг от друга архитектурой, использованными решениями и удобством работы.

Самая популярная на сегодняшний день система контроля версий – Git.

Умение работать с git’ом — обязательный навык для программиста любого профиля. Можно долго обсуждать преимущества и недостатки разных систем контроля версий, но большинство компаний используют git, поэтому уметь работать с git’ом нужно всем.

Git – распределённая система контроля версий. Что даёт ей все преимущества децентрализованной СКВ:

* высокую скорость проведения всех операций (за счет отсутствия сетевой задержки);
* идеальные условия для командной разработки;
* страховку от потери информации при возникновении проблем с центральным сервером.

Для контроля версий в git используются 2 репозитория: локальный и удаленный. Локальный репозиторий (полноценный репозиторий, а не ссылки или копии отдельных ветвей) находится на компьютере разработчика, а удаленный на удалённом сервере. Доступ к удаленному репозиторию обеспечивается благодаря гит-хостингу Github, Google Code, GitLab и т.д.

*Как работает git*

Взаимодействие с удаленным репозиторием происходит при наличии интернета и, по сути, представляет собой синхронизацию двух репозиториев.

Команда push копирует новые данные, содержащиеся в локальном репозитории, в удалённый репозиторий, а команда pull передает данные из удаленного репозитория в локальный.

Каждая версия документа, внесенные обновления и т.д записываются в локальный репозиторий.

*Дерево проекта*

В репозитории содержится «дерево» проекта, то есть все сохраненные версии файлов.

Дерево может быть прямым (в этом случае каждое последующее сохранение файлов производилось после предыдущего без возвращения к более ранним версиям) и разветвленным.

К появлению «веток» приводит работа с более ранними версиями и сохранение внесённых изменений.

На различных ветках дерева содержатся сохранения, основой которых был один исходный файл. В ходе работы в файлы на разных ветках были внесены разные изменения. В системе управления версиями можно работать со всеми ветками дерева проекта, пошагово, изменяя и дополняя содержащиеся в них данные. После проведения ряда изменений 2 ветки могут «срастись», в новой версии файла будут учтены все внесенные изменения.

**Лекция 11. Настройка среды разработки.**

При первом запуске Visual Studio вы можете оптимизировать среду разработки для наиболее привычного типа разработки, выбрав коллекцию параметров. Каждая коллекция оптимизирует элементы, такие как сочетания клавиш, макеты окон, шаблоны проектов и элементов, а также видимость команд.

Доступны следующие коллекции параметров.

* Общие сведения
* JavaScript
* Visual Basic
* Visual C#
* Visual C++
* Веб-разработка
* Веб-разработка (только код)

Визуальное представление

Цветовая тема

Для интегрированной среды разработки можно выбрать синюю, светлую, темную или синюю (дополнительный контраст) цветовую тему.

Вы можете устанавливать дополнительные предопределенные темы и создавать пользовательские темы, скачав и установив редактор цветовых тем Visual Studio из Visual Studio Marketplace. После установки этого средства в списке Цветовая тема появятся дополнительные темы.

Применить регистр заголовка к строке меню

Меню используют регистр заголовка по умолчанию. Снимите этот флажок, чтобы использовать стиль верхнего регистра.

Оптимизировать отображение для экранов с другой плотностью пикселей (требуется перезапуск)

Этот параметр включает или отключает отрисовку, учитывающую параметры монитора (PMA) в точках на дюйм (DPI). Когда функция PMA включена, пользовательский интерфейс Visual Studio отображается четко при любом разрешении монитора или масштабе отображения, в том числе на нескольких мониторах. Чтобы включить PMA, вам нужно обновление Windows 10 за апрель 2018 г. или более поздней версии и NET Framework 4.8 или более поздней версии. (Этот параметр отображается серым цветом, если эти два условия не выполняются.)

Автонастройка представления в зависимости от быстродействия клиента

Определяет, задает ли Visual Studio корректировку визуального представления автоматически или это делает пользователь. Эта корректировка может изменить способ отображения цветов с градиентов на простые цвета или может ограничить применение анимации в меню или всплывающих окнах.

Включить расширенное визуальное представление клиента

Включает все визуальные возможности Visual Studio, в том числе градиенты и анимации. Снимите этот флажок при использовании подключений к удаленному рабочему столу или графических адаптеров старых моделей, так как эти возможности могут снизить производительность в таких случаях. Этот параметр доступен, только если флажок Автонастройка представления в зависимости от быстродействия клиента снят.

Элементы для отображения в меню "Окно"

Задает количество окон, отображаемых в списке окон меню Окно. Введите число от 1 до 24. Значение по умолчанию — 10.

Пунктов в списках последних использованных элементов

Задает число недавно использованных проектов и файлов, отображаемых в меню Файл. Введите число от 1 до 24. Значение по умолчанию — 10. Это удобный способ доступа к недавно использованным проектам и файлам.

Показать строку состояния

Отображает строку состояния. Строка состояния расположена в нижней части окна интегрированной среды разработки и содержит сведения о ходе выполнения текущих операций.

Кнопка закрытия действует только на активное окно

Указывает, что при нажатии кнопки Закрыть закрывается только активное окно с фокусом, а не все закрепленные окна инструментов. Этот параметр выбран по умолчанию.

Кнопка автоскрытия действует только на активное окно

Указывает, что при нажатии кнопки Автоматическое скрытие автоматически закрывается только окно инструментов с фокусом, а не все закрепленные окна инструментов. По умолчанию этот флажок не установлен.

Инструменты, доступные в меню "Сервис"

В меню Инструменты содержится несколько встроенных команд, в том числе следующие:

• Диспетчер фрагментов кода для организации фрагментов кода.

• Настройка для настройки меню и панелей инструментов.

• Параметры для задания различных параметров интегрированной среды разработки Visual Studio и других инструментов.

**Лекция 12. Мониторинг разработки проекта. Сохранение версий проекта.**

Мониторинг проекта - это регулярный процесс сбора, измерения и распространения информации об исполнении проекта, ее регистрация, оценка изменений и подготовка для принятия решения. Целью мониторинга является предоставление руководству проекта или компании оперативной интегрированной информации о реализации проекта, достаточной для принятия ключевых решений по проекту. Интегральные показатели состояния проекта должны удовлетворять потребностям руководства как с точки зрения полноты, так и оперативности ее предоставления.

Инспекция является наиболее распространенным инструментом мониторинга. Для этого привлекаются как сотрудники компании, не участвующие в проекте, но компетентные в предметной или управленческой областях проекта, так и внешние опытные инспекторы. Инспекция может осуществляться руководителем или куратором проекта. Нужно просто прийти туда, где осуществляются работы по проекту, и понаблюдать, что там происходит, задавая вопросы и слушая объяснения. Основной особенностью проведения инспекции является то, что она проводится без объявления и неожиданно. Однако она также должна быть открытой и прямой.

Инспекция является эффективным способом выяснить, выполняются ли спецификации проекта, а также существуют ли необязательные потери рабочего времени и небезопасные работы. Инспекция может финансироваться за счет накладных проекта, но она не фигурирует в плане, обеспечивая внезапность ее проведения. Финансирование инспекционных затрат возможно из бюджета компании или заказчика.

В процессе обсуждения состояния проекта с сотрудниками руководитель проекта может играть следующие роли.

Слушатель

Основные правила здесь простые - внимательно слушайте тех, кто отчитывается перед вами по состоянию, по отклонениям от плана, неожиданным проблемам, и предлагайте возможные решения, слушайте не только то, что говорят, но также то, как говорят. Постарайтесь понять, возбужден ли человек, расстроен, обескуражен? Проясните то, о чем говорится, путем постановки правильных вопросов и проверяйте понятое, повторяя полученную информацию. Тот, кто проводит такие встречи, должен обладать хорошими навыками проведения интервью.

Помощник

Если в проекте возникают проблемы, руководитель проекта может способствовать их решению путем направления другого члена команды в сторону возможных вариантов действий. Для помощи проекту можно использовать, по мере необходимости, все знания и опыт руководителя проекта.

Интегратор

Важной ролью руководителя проекта является соединение отдельных частей проекта в единое целое. Ничего ли не упущено? Существует ли дублирование усилий? Как лучше всего использовать способности членов команды?

Лидер

Как уже отмечалось, вероятно, наиболее важной ролью руководителя проекта является роль лидера. Путем использования разнообразных методов он направляет усилия команды на достижение общей цели выполнения проекта в соответствии со спецификациями, в необходимое время и в рамках бюджета. Он должен поддерживать и признавать хорошую работу, корректировать плохую работу, поддерживать на высоком уровне интерес и энтузиазм.

Экспертиза

Экспертиза проекта - это детальный экспертный анализ определенных областей как проектной, так и предметной деятельности в проекте. Она также включает составление обшей картины хода и состояния проекта в целях повышения качества выполнения как данного проекта, так и остальных проектов компании в целом.

Экспертизу осуществляют привлеченные квалифицированные и опытные специалисты, называемые экспертами. Для проведения экспертизы используются как формализованные данные, полученные в результате процедур аудита и мониторинга проекта, так и сведения, получаемые в ходе консультаций и собеседований и относящиеся к неформализованным (или слабо формализованным) областям управления проектом, таким как компетентность персонала, межличностные отношения и т. д. Экспертизу проекта проводят и внешние организации в любое заранее согласованное время.

Аудит

Аудит проекта - это проверка соответствия формализованной организационной деятельности по управлению данным проектом принятым в отрасли или компании регламентам или стандартам управления проектами

Тестирование является еще одним способом проверить качество проекта, в том числе качество создаваемого продукта и качество управления проектом.

**Лекция 13. Требования к интерфейсу пользователя. Принципы создание графического пользовательского интерфейса(GUI).**

Пользовательский интерфейс (ПИ, графический интерфейс пользователя)– это комплекс средств для взаимодействия пользователя с технической системой (в т. ч. с программным приложением, мультимедийным изданием).

В понятие пользовательского интерфейса компьютерной системы входят следующие основные составляющие:

* графическая среда - картинка на экране (трехмерная, анимированная, просто выполненная в модном дизайне);
* набор управляющих элементов пользовательского интерфейса и их расположение на экране.
* технологии взаимодействия пользователя с системой.

Требования к интерфейсу:

• функциональность (соответствие задачам пользователя);

• соответствие технологии;

• понятность и логичность;

• обеспечение высокой скорости работы пользователя;

• обеспечение защиты от человеческих ошибок;

• быстрое обучение пользователя;

• субъективное удовлетворение пользователя

Разработка графического пользовательского интерфейса охватывает весь жизненный цикл производства программного продукта - она начинается на этапе определения требований и продолжается до реализации.

Проектирование пользовательского интерфейса по технологии программной инженерии может осуществляться двумя методами:

* визуальное формирование пользовательского интерфейса, удовлетворяющего различным требованиям практичности;
* собственно проектирование пользовательского интерфейса в терминах классов проектирования, с поддержкой объектов и т. п..
* с последующей конечной реализацией пользовательского интерфейса.

Основной элемент проектирования интерфейса заключается в том, что контроль находится на стороне пользователя (при условии, что система, а не пользователь, контролирует системную целостность, защиту и безопасность). Современные объектно- ориентированные программы управляются событиями. Объекты реагируют на события (сообщения). Внутренние взаимодействия между объектами запускаются внешними событиями, инициируемыми пользователем.

Объектно-ориентированное проектирование GUI

В обозримом будущем объектно-ориентированные технологии, несомненно, станут доминирующей парадигмой разработки компьютерных обучающих систем. Поскольку одним из важнейших компонентов успешной реализации программной разработки является хороший графический пользовательский интерфейс, применение объектно-ориентированных технологий для его разработки должно быть направлено на достижение высокого качества GUI за счет использование всех преимуществ этой технологии.

Визуальное прототипирование GUI

Для окончательного формирования и уточнения требований к разрабатываемой системе в рамках технологического процесса определения требований целесообразно создание прототипа GUI методом визуального формирования с использованием графических систем, например, Visual Basic и Java Beans.

Данные принципы проектирования интерфейса направлены на решение таких вопросов, как локализация управления на стороне пользователя, соответствие, индивидуализация, настройка, терпимость к ошибкам пользователя, обратная связь, эстетичность и удобство использования.

Наибольшее значение для прототипирования пользовательского интерфейса имеет модель прецедентов и дополнительные спецификации, разрабатываемые совместно с пользователями (или их представителями) и другими заинтересованными сторонами

**Лекция 14. Понятие спецификации языка программирования. Синтаксис языка программирования. Стиль программирования.**

Спецификация (стандарт, определение) языка программирования — это предмет документации, который определяет язык программирования, чтобы пользователи и разработчики языка могли согласовывать, что означают программы на данном языке. Спецификации обычно являются подробными и формальными и в основном используются разработчиками языка, в то время как пользователи обращаются к ним в случае двусмысленности: например, спецификация языка C++ часто цитируется пользователями из-за сложности. Сопутствующая документация включает справочник по языку программирования, который специально предназначен для пользователей, и логическое обоснование языка программирования, которое объясняет, почему спецификация написана именно так; последние обычно более неформальны, чем спецификации.

**Синтаксис C#.**

**Идентификаторы**

Идентификатор — это имя сущности в коде. Существует стандарт именования идентификаторов, который следует использовать в коде. Идентификатор может:

Начинаться с символа «\_».

Содержать заглавные и строчные буквы в формате Unicode. Регистр имеет значение.

Идентификатор не может:

* Начинаться с цифры.
* Начинаться с символа, если это ключевое слово.
* Содержать более 511 символов.

**Ключевые слова**

Ключевые слова — это предварительно определенные зарезервированные идентификаторы, имеющие особое синтаксическое значение. Язык имеет два типа ключевых слов — зарезервированные в любой части кода и контекстные. Слова первого типа, например false или byte, могут быть использованы только как ключевые слова. Слова второго типа, например слово where, имеют особое значение только в определённом контексте и могут быть использованы в качестве идентификаторов вне него. Если всё же необходимо использовать ключевое слово в качестве идентификатора, то необходимо объявить его с помощью префикса @ — например, @byte.

Константа — это идентификатор, содержащий значение, которое запрещено менять во время выполнения программы. Константы неявно статические.

**const**

Локальная переменная или свойство с заданным значением могут быть объявлены с ключевым словом const, после чего их значение не может быть изменено.

**Блоки кода**

Операторы {...} используются для обозначения начала и конца блоков кода, а также обозначения области видимости. Идентификатор, объявленный внутри фигурных скобок, видим только внутри них и во вложенных областях видимости. Следующий пример демонстрирует работу блока кода и областей видимости:

**void** doSomething()

{

int a;

{

int b;

a = 0;

}

a = 1;

b = 1; *//Здесь компилятор выдаст ошибку, так как переменная b объявлена в области видимости, недоступной для данного кода.*

}

Любое приложение, консольное оно или графическое, обязано иметь точку входа. В C#-приложении такой точкой является метод Main, который может быть только в одном экземпляре и обязательно должен быть статическим. Этот метод может не возвращать значения

**static** **void** Main(string[] args)

{

}

**Лекция 15. Основные конструкции выбранного языка программирования.**

**Циклы в Си-шарп. Операторы break и continue**

Циклы служат для многократного повторения некоторого фрагмента кода.

В Си-шарп есть четыре оператора циклов: for, while, do-while, foreach.

**Цикл for**

Этот цикл используется тогда, когда наперед известно, сколько повторений нужно сделать. Он имеет следующую структуру:

for (инициализация счетчика; условие продолжения; итерация)

{

//блок кода, который будет повторяться

}

**Цикл while**

Слово while переводится, как «пока», что хорошо его характеризует. Он продолжает выполнятся до тех пор, пока «истинно» некоторое условие. Он имеет такую структуру:

while (условие продолжения)

{

//блок кода, который будет повторяться

}

**Цикл do-while**

Этот тот же цикл while, только здесь сначала выполняется блок кода, а уже потом идет проверка условия. Это гарантирует хотя бы один проход цикла.

do

{

//блок кода, который будет повторяться

}

while (условие продолжения);

**Оператор break**

Из любого цикла можно досрочно выйти, используя оператор break. Использовать данный оператор есть смысл после удовлетворения некоторого условия, иначе цикл завершится на первой итерации.

**Оператор continue**

Данный оператор позволяет перейти к следующей итерации, не завершив до конца текущую.

**Оператор «if-else»**

Данный оператор имеет следующую структуру:

if ([условное выражение])

{

Блок кода, который нужно выполнить при удовлетворении условия, [условное выражение] = true (истина)

}

else

{

Блок кода, который нужно выполнить при неудовлетворении условия, [условное выражение] = false (ложь)

}

**Оператор switch**

В некоторых случаях удобно использовать условный оператор «switch» вместо «if-else». Он имеет следующую структуру:

switch (выражение)

{

case значение1:

блок1;

break;

case значение2:

блок2;

break;

...

case значениеN:

блокN;

break;

default:

блокN+1;

break;

}

***Ключевое слово using***

При использовании класса не требуется уточнять его пространство имён, если оно указано в .cs файле с помощью директивы using. Обычно это делается вверху файла с исходным кодом, однако может быть размещено и в других местах, например внутри класса.

**using** **System**;

**using** **System.Collections**;

**Метод MAIN**

или возвращать целое число, которое является кодом завершения программы: в случае, если программа завершилась без ошибки, этот код — 0.

**static** int Main(string[] args)

{

**return** 0;

}

**Лекция 16. Описание переменных.**

Для хранения данных в программе применяются переменные. Переменная представляет именнованную область памяти, в которой хранится значение определенного типа. Переменная имеет тип, имя и значение. Тип определяет, какого рода информацию может хранить переменная.

Перед использованием любую переменную надо определить. Синтаксис определения переменной выглядит следующим образом:

тип имя\_переменной;

Например, определим простейшую переменную:

string name;

В данном случае определена переменная name, которая имеет тип string. то есть переменная представляет строку. Поскольку определение переменной представляет собой инструкцию, то после него ставится точка с запятой.

При этом следует учитывать, что C# является регистрозависимым языком, поэтому следующие два определения переменных будут представлять две разные переменные:

string name;

string Name;

После определения переменной можно присвоить некоторое значение:

string name;

name = "Tom";

Так как переменная name представляет тип string, то есть строку, то мы можем присвоить ей строку в двойных кавычках. Причем переменной можно присвоить только то значение, которое соответствует ее типу.

В дальнейшем с помощью имени переменной мы сможем обращаться к той области памяти, в которой хранится ее значение.

Также мы можем сразу при определении присвоить переменной значение. Данный прием называется инициализацией:

string name = "Tom";

Отличительной чертой переменных является то, что в программе можно многократно менять их значение. Например, создадим небольшую программу, в которой определим переменную, поменяем ее значение и выведем его на консоль:

using System;

namespace HelloApp

{

    class Program

    {

        static void Main(string[] args)

        {

            string name = "Tom";  // определяем переменную и инициализируем ее

            Console.WriteLine(name);    // Tom

            name = "Bob";       // меняем значение переменной

            Console.WriteLine(name);    // Bob

            Console.Read();

        }

    }

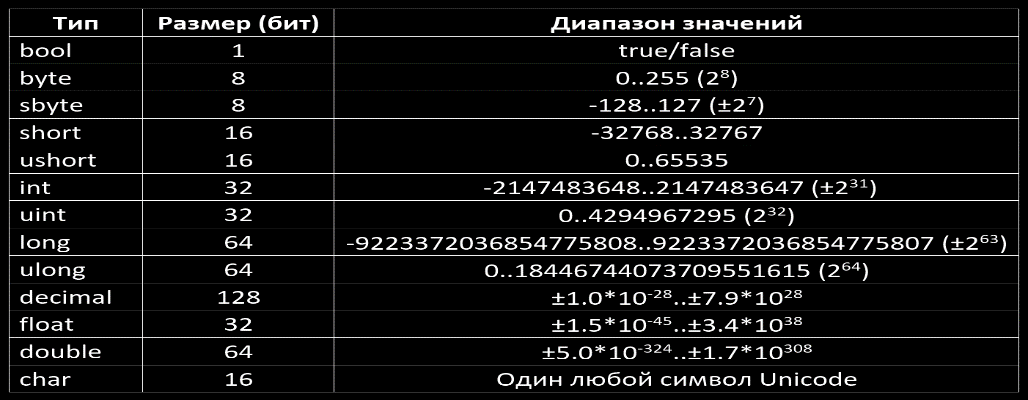
}

Консольный вывод программы:

Tom

Bob

Стандартные типы данных



**Лекция 17. Организация ввода-вывода.**

Для вывода информации на консоль мы уже использовали встроенный метод Console.WriteLine. То есть, если мы хотим вывести некоторую информацию на консоль, то нам надо передать ее в метод Console.WriteLine:

using System;

namespace HelloApp

{

    class Program

    {

        static void Main(string[] args)

        {

            string hello = "Привет мир";

            Console.WriteLine(hello);

            Console.WriteLine("Добро пожаловать в C#!");

            Console.WriteLine("Пока мир...");

            Console.WriteLine(24.5);

            Console.ReadKey();

        }

    }

}

Консольный вывод:

Привет мир!

Добро пожаловать в C#!

Пока мир...

24,5

Нередко возникает необходимость вывести на консоль в одной строке значения сразу нескольких переменных. В этом случае мы можем использовать прием, который называется интерполяцией:

using System;

namespace HelloApp

{

    class Program

    {

        static void Main(string[] args)

        {

            Console.Write("Введите имя: ");

            string name = Console.ReadLine();

            Console.Write("Введите возраст: ");

            int age = Convert.ToInt32(Console.ReadLine());

            Console.Write("Введите рост: ");

            double height = Convert.ToDouble(Console.ReadLine());

            Console.Write("Введите размер зарплаты: ");

            decimal salary = Convert.ToDecimal(Console.ReadLine());

            Console.WriteLine($"Имя: {name}  Возраст: {age}  Рост: {height}м  Зарплата: {salary}$");

            Console.ReadKey();

        }

    }

}

Введите имя: Том

Введите возраст: 25

Введите рост: 1,75

Введите размер зарплаты: 300,67

Имя: Том Возраст: 25 Рост: 1,75м Зарплата: 300,67$

**Лекция 18. Обоснование и осуществление выбора модели построения или модификации информационной системы.**

Многие пользователи компьютерной техники и программного обеспечения неоднократно сталкивались с ситуацией, когда программное обеспечение, хорошо работающее на одном компьютере, не работает на другом таком же устройстве. Или системные блоки одного вычислительного устройства не стыкуются с аппаратной частью другого. Или информационная система другой компании упорно не желает обрабатывать данные, которые вы подготовили в информационной системе у себя на рабочем месте. И так далее... Эта проблема называется проблемой совместимости вычислительных, телекоммуникационных и информационных устройств

**Модель создания информационной системы**

Методологически важно наряду с рассмотренными моделями среды ИС предложить модель создания ИС, которая имела бы те же аспекты функциональных групп компонентов (пользователи, функции, данные, коммуникации). Такой подход обеспечит сквозной процесс проектирования и сопровождения на всех стадиях эксплуатации ИС, а также возможность обоснованного выбора стандартов на разработку систем и документирование проектов.

**Реинжиниринг бизнес-процессов**

Внедрение информационных технологий и реализованных на их основе информационных систем в повседневную деятельность предприятия дает ему тактические и долгосрочные преимущества в бизнесе. Стремление руководства к использованию ИТ может остаться лишь благими намерениями, если оно не будет следовать сложившимся требованиям и правилам разработки, проектирования и внедрения ИТ. Выше говорилось о базовых требованиях к стандартизации объектов и функциональных задач, без которых реализуемая система не будет являться открытой системой, что приведет впоследствии к многочисленным проблемам при ее внедрении и эксплуатации.

Бизнес-процесс представляет собой систему последовательных, целенаправленных и регламентированных видов деятельности, в которой посредством управляющего воздействия и с помощью определенных ресурсов за определенное время входы процесса преобразуются в выходы - в результаты, представляющие ценность для потребителя и приносящие прибыль изготовителю.

Существует несколько базовых правил, которых следует придерживаться в процессе проведения реинжиниринга:

* разработка последовательных пошаговых процедур для перепроектирования процессов;
* использование в проектировании стандартных языков и нотаций;
* наличие эвристических и прагматических показателей, позволяющих оценить или измерить степень соответствия перепроектированного процесса или функциональности заданным целям;
* подход к решению частных задач и к их совокупности должен быть системным;
* даже небольшое улучшение должно давать быстрый положительный эффект.

**Лекция 19. Обоснование и осуществление выбора средства построения информационной системы и программных средств.**

Всякая профессиональная деятельность, так или иначе, связана с информацией, с организацией ее сбора, хранения, выборки. Можно сказать, что неотъемлемой частью повседневной жизни стали базы данных, для поддержки которых требуется некоторый организационный метод, или механизм. Такой механизм называется системой управления базами данных (СУБД).

База данных (БД) - совместно используемый набор логически связанных данных (и их описание), предназначенный для удовлетворения информационных потребностей организации.

СУБД (система управления базами данных) - программное обеспечение, с помощью которого пользователи могут определять, создавать и поддерживать базу данных, а также получать к ней контролируемый доступ.

Управление основными потоками информации осуществляется с помощью, так называемых, систем управления реляционными базами данных, которые берут свое начало в традиционных системах управления базами данных. Именно объединение реляционных баз данных и клиент-серверных технологий позволяет современному предприятию успешно управлять собственными данными, оставаясь конкурентоспособным на рынке товаров и услуг.

Реляционные БД имеют мощный теоретический фундамент, основанный на математической теории отношений. Появление теории реляционных баз данных дало толчок к разработке ряда языков запросов, которые можно отнести к двум классам:

* - алгебраические языки, позволяющие выражать запросы средствами специализированных операторов, применяемых к отношениям;
* - языки исчисления предикатов, представляющие собой набор правил для записи выражения, определяющего новое отношение из заданной совокупности существующих отношений. Следовательно, исчисление предикатов есть метод определения того отношения, которое желательно получить как ответ на запрос из отношений, уже имеющихся в базе данных.

В реляционной модели объекты реального мира и взаимосвязи между ними представляются с помощью совокупности связанных между собой таблиц (отношений).

Даже в том случае, когда функции СУБД используются для выбора информации из одной или нескольких таблиц (т.е. выполняется запрос), результат также представляется в табличном виде. Более того, можно выполнить запрос с применением результатов другого запроса.

Каждая таблица БД представляется как совокупность строк и столбцов, где строки (записи) соответствуют экземпляру объекта, конкретному событию или явлению, а столбцы (поля) - атрибутам (признакам, характеристикам, параметрам) объекта, события, явления.

В каждой таблице БД необходимо наличие первичного ключа - так именуют поле или набор полей, однозначно идентифицирующий каждый экземпляр объекта или запись. Значение первичного ключа в таблице БД должно быть уникальным, т.е. в таблице не допускается наличие двух и более записей с одинаковыми значениями первичного ключа. Он должен быть минимально достаточным, а значит, не содержать полей, удаление которых не отразится на его уникальности.

Связи между объектами реального мира могут находить свое отражение в структуре данных, а могут и подразумеваться, т.е. присутствовать на неформальном уровне.

Между двумя или более таблицами базы данных могут существовать отношения подчиненности, которые определяют, что для каждой записи главной таблицы (называемой еще родительской) возможно наличие одной или нескольких записей в подчиненной таблице (называемой еще дочерней).

Выделяют три разновидности связи между таблицами базы данных:

* - "один-ко-многим";
* - "один-к-одному";
* - "многие-ко-многим".

Отношение "один-ко-многим" имеет место, когда одной записи родительской таблицы может соответствовать несколько записей дочерней. Связь "один-ко-многим" иногда называют связью "многие-к-одному". И в том, и в другом случае сущность связи между таблицами остается неизменной. Связь "один-ко-многим" является самой распространенной для реляционных баз данных. Она позволяет моделировать также иерархические структуры данных.

Отношение "один-к-одному" имеет место, когда одной записи в родительской таблице соответствует одна запись в дочерней. Это отношение встречается намного реже, чем отношение "один-ко-многим". Его используют, если не хотят, чтобы таблица БД "распухала" от второстепенной информации, однако для чтения связанной информации в нескольких таблицах приходится производить ряд операций чтения вместо одной, когда данные хранятся в одной таблице.

Отношение "многие-ко-многим" применяется в следующих случаях. Одной записи в родительской таблице соответствует более одной записи в дочерней. Или же одной записи в дочерней таблице соответствует более одной записи в родительской. Всякую связь "многие-ко-многим" в реляционной базе данных необходимо заменить на связь "один-ко-многим" (одну или более) с помощью введения дополнительных таблиц. [6]

Современные СУБД в основном являются приложениями Windows, так как данная среда позволяет более полно использовать возможности персональной ЭВМ, нежели среда DOS. Снижение стоимости высокопроизводительных ПК обусловил не только широкий переход к среде Windows, где разработчик программного обеспечения может в меньшей степени заботиться о распределении ресурсов, но также сделал программное обеспечение ПК в целом и СУБД в частности менее критичными к аппаратным ресурсам ЭВМ.

Среди наиболее ярких представителей систем управления базами данных можно отметить: Lotus Approach, Microsoft Access, Borland dBase, Borland Paradox, Microsoft Visual FoxPro, Microsoft Visual Basic, а также баз данных Microsoft SQL Server и Oracle.

Любая СУБД позволяет выполнять четыре простейшие операции с данными:

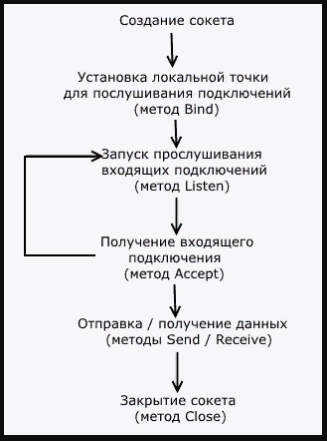
* - добавлять в таблицу одну или несколько записей,
* - удалять из таблицы одну или несколько записей,
* - обновлять значения некоторых полей в одной или нескольких записях,
* - находить одну или несколько записей, удовлетворяющих заданному условию.

Для выполнения этих операций используется механизм запросов. Результатом выполнения запросов является либо отобранное по определенным критериям множество записей из таблицы, либо изменения в таблицах. Запросы к базе формируются на специально созданном для этого языке, который так и называется - язык структурированных запросов (Structured Query Language, SQL).

Важной функцией СУБД является управление данными. Под этим обычно понимают защиту данных от несанкционированного доступа, поддержку многопользовательского режима работы с данными, а также обеспечение целостности и согласованности данных. Защита от несанкционированного доступа позволяет каждому пользователю видеть и изменять только те данные, которые ему разрешено видеть или менять. Средства, обеспечивающие многопользовательскую работу, не позволяют нескольким пользователям одновременно изменять одни и те же данные. Средства обеспечения целостности и согласованности данных не дают выполнять такие изменения, после которых данные могут оказаться несогласованными.

**Лекция 20. Создание сетевого сервера и сетевого клиента.**

Рассмотрим, как создать сервер, работающий по протоколу TCP, с помощью сокетов. Общая схема работы серверного сокета TCP будет следующей:



Код программы сервера будет таким:

using System;

using System.Text;

using System.Net;

using System.Net.Sockets;

namespace SocketTcpServer

{

class Program

{

static int port = 8005; // порт для приема входящих запросов

static void Main(string[] args)

{

// получаем адреса для запуска сокета

IPEndPoint ipPoint = new IPEndPoint(IPAddress.Parse(«127.0.0.1»), port);

// создаем сокет

Socket listenSocket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);

try

{

// связываем сокет с локальной точкой, по которой будем принимать данные

listenSocket.Bind(ipPoint);

// начинаем прослушивание

listenSocket.Listen(10);

Console.WriteLine("Сервер запущен. Ожидание подключений...");

while (true)

{

Socket handler = listenSocket.Accept();

// получаем сообщение

StringBuilder builder = new StringBuilder();

int bytes = 0; // количество полученных байтов

byte[] data = new byte[256]; // буфер для получаемых данных

do

{

bytes = handler.Receive(data);

builder.Append(Encoding.Unicode.GetString(data, 0, bytes));

}

while (handler.Available>0);

Console.WriteLine(DateTime.Now.ToShortTimeString() + ": " + builder.ToString());

// отправляем ответ

string message = "ваше сообщение доставлено";

data = Encoding.Unicode.GetBytes(message);

handler.Send(data);

// закрываем сокет

handler.Shutdown(SocketShutdown.Both);

handler.Close();

}

}

catch(Exception ex)

{

Console.WriteLine(ex.Message);

}

}

}

}

Вначале после создания сокета связываем его с локальной точкой методом Bind:

listenSocket.Bind(ipPoint);

Сокет будет прослушивать подключения по 8005 порту на локальном адресе 127.0.0.1. То есть клиент должен будет подключаться к локальному адресу и порту 8005.

Далее начинаем прослушивание:

listenSocket.Listen(10);

Метод Listen вызывается только после метода Bind. В качестве параметра он принимает количество входящих подключений, которые могут быть поставлены в очередь сокета.

После вызова метода Listen начинается прослушивание входящих подключений, и если подключения приходят на сокет, то их можно получить с помощью метода Accept:

Socket handler = listenSocket.Accept();

Метод Accept извлекает из очереди ожидающих запрос первый запрос и создает для его обработки объект Socket. Если очередь запросов пуста, то метод Accept блокирует вызывающий поток до появления нового подключения.

Для обработки запроса сначала в цикле do..while получаем данные с помощью метода Receive:

do{bytes = handler.Receive(data);builder.Append(Encoding.Unicode.GetString(data, 0, bytes));}while (handler.Available > 0);

Метод Receive в качестве параметра принимает массив байтов, в который считываются полученные данные, и возвращает количество полученных байтов.

Если отсутствуют данные, доступные для чтения, метод Receive блокирует вызывающий поток до тех пор, пока не станут доступны данные, если не было установлено значение тайм-аута путем использования объекта Socket.ReceiveTimeout. Если значение тайм-аута было превышено, объект Receive выдаст исключение SocketException. Чтобы отследить наличие данных в потоке, используется свойство Available.

После получения данных клиенту посылается ответное сообщение с помощью метода Send, который в качестве параметра принимает массив байтов:

handler.Send(data);

В конце обработки запроса надо закрыть связанный с ним сокет:

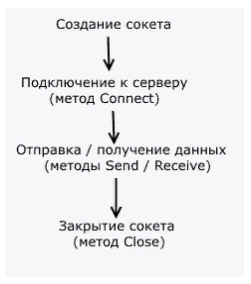
handler.Shutdown(SocketShutdown.Both);handler.Close();

Вызов метода Shutdown() перед закрытием сокета гарантирует, что не останется никаких необработанных данных. Этот метод в качестве параметра принимает значение из перечисления SocketShutdown:

* Both: остановка как отправки, так и получения данных сокетом
* Receive: остановка получения данных
* Send: остановка отправки данных

**Клиент**

Теперь добавим проект для клиента. Общая схема работы клиента на сокетах будет немного отличаться:



Полный код клиента:

using System;

using System.Text;

using System.Net;

using System.Net.Sockets;

namespace SocketTcpClient

{

class Program

{

// адрес и порт сервера, к которому будем подключаться

static int port = 8005; // порт сервера

static string address = «127.0.0.1»; // адрес сервера

static void Main(string[] args)

{

try

{

IPEndPoint ipPoint = new IPEndPoint(IPAddress.Parse(address), port);

Socket socket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);

// подключаемся к удаленному хосту

socket.Connect(ipPoint);

Console.Write("Введите сообщение:");

string message = Console.ReadLine();

byte[] data = Encoding.Unicode.GetBytes(message);

socket.Send(data);

// получаем ответ

data = new byte[256]; // буфер для ответа

StringBuilder builder = new StringBuilder();

int bytes = 0; // количество полученных байт

do

{

bytes = socket.Receive(data, data.Length, 0);

builder.Append(Encoding.Unicode.GetString(data, 0, bytes));

}

while (socket.Available > 0);

Console.WriteLine("ответ сервера: " + builder.ToString());

// закрываем сокет

socket.Shutdown(SocketShutdown.Both);

socket.Close();

}

catch(Exception ex)

{

Console.WriteLine(ex.Message);

}

Console.Read();

}

}

}

Для клиента характерно все то же самое, только теперь после создания сокета вызывается метод Connect(), в который передается адрес сервера:

IPEndPoint ipPoint = new IPEndPoint(IPAddress.Parse(address), port);socket.Connect(ipPoint);

Теперь запустим программы сервера и клиента. Консоль клиента:

Введите сообщение: привет мир

ответ сервера: ваше сообщение доставлено

Консоль сервера:

Сервер запущен. Ожидание подключений...

22:34: привет мир

**Лекция 21. Разработка графического интерфейса пользователя.**

Создали сам пользовательский *интерфейс* класса InputPassword диалогового окна. Теперь нужно настроить *класс* InputPassword, добавив туда метод создания диалогового окна и вызова его на экран.

* Модифицируйте класс InputPassword следующим образом

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Drawing;

using System.Windows.Forms;

namespace WinFormsApp

{

public partial class InputPassword : Form

{

public InputPassword()

{

InitializeComponent();

}

public InputPassword(string windowTitle, string message)

{

InitializeComponent();

this.Text = windowTitle;

lblMessage.Text = message;

}

public static string Show(string windowTitle, string message)

{

using(InputPassword inputDlg =

new InputPassword(windowTitle, message))

{

inputDlg.ShowDialog();

return inputDlg.txtInput.Text;

}

}

Создали параметризованный *конструктор* для ввода заголовков диалогового окна в момент его создания. Мы добавили статический метод создания диалогового окна, который возвращает *значение* текстового поля после щелчка пользователем на кнопке, и такой метод нужно вызывать по имени класса. Упаковка создания экземпляра формы в блок using позволяет вызвать сборщик мусора сразу после выхода из этого блока.

Теперь нужно поместить вызов окна запроса пароля в обработчик события Load запуска формы.

* Откройте файл **Start.cs** в режиме View Designer, выделите форму через ее заголовок, перейдите в панель Properties, установите ее в режим *Events*, найдите событие Load и создайте для него обработчик Start\_Load(). Заполните обработчик события так

private void Start\_Load(object sender, System.EventArgs e)

{

string password = InputPassword.Show(

"Окно аутентификации пользователя",

"Просим ввести Ваш пароль:");

if (password != "root")// Плохой способ хранения пароля

{

MessageBox.Show(

"Извините, но Вам не разрешено\n" // Сообщение

+ "пользоваться этой суперпрограммой!!!",

"Неверный пароль", // Заголовок окна

MessageBoxButtons.OK, // Кнопка OK

MessageBoxIcon.Stop);// Критическая иконка

this.Close();

}

}

На стартовой форме мы создадим *список* с наименованиями упражнений, который будем пополнять по мере их выполнения. В окончательном варианте работы щелчок по пункту списка должен запускать дочернюю форму, соответствующую выполненному упражнению. Это удобно при демонстрации выполненной лабораторной работы.

Вкладка **All Windows Forms** панели **Toolbox** содержит все компоненты, которые необходимы для быстрой разработки интерфейса пользователя. Она становится доступной, когда форма открыта в режиме Design и текущее *приложение* не запущено. Режим Design и компоненты панели *Toolbox* - это лишь инструменты для частичной *автоматизации* труда программиста. С их помощью быстро и синтаксически правильно *оболочка* генерирует код C# приложения в файле \*.cs. Все компоненты поддерживаются соответствующими классами и доступны для управления ими программным путем.

В панели *Toolbox* имеется возможность добавлять свои вкладки и копировать в них нужный набор инструментов из других вкладок. Добавить вкладку можно, щелкнув правой кнопкой мыши в любом месте панели *Toolbox* и выполнив команду контекстного меню Add Tab. Удалить вкладку можно, щелкнув правой кнопкой мыши на заголовке соответствующей вкладки и выполнив команду Delete Tab. Копировать нужные компоненты из одной вкладки в другую можно выполнением команд контекстного меню Copy и Paste.

|  |  |
| --- | --- |
| **Некоторые компоненты панели Toolbox** | |
| **Компонент** | **Назначение** |
| https://intuit.ru/EDI/05_07_17_1/1499206907-9549/tutorial/627/objects/25/files/25_72.png  *Pointer* | Установка курсора в режим выделения объектов. |
| https://intuit.ru/EDI/05_07_17_1/1499206907-9549/tutorial/627/objects/25/files/25_25.png  Button | Инструмент для генерирования события щелчка, по которому вызывается функция - обработчик. В эту функцию можно разместить любой код, который требуется выполнить при щелчке на этом инструменте. |
| https://intuit.ru/EDI/05_07_17_1/1499206907-9549/tutorial/627/objects/25/files/25_26.png  *CheckBox* | Дает возможность пользователю установить нужный режим из двух возможных. При обединении этих элементов в тематическую *группу пользователь* может производить множественные настройки нашего приложения. |
| https://intuit.ru/EDI/05_07_17_1/1499206907-9549/tutorial/627/objects/25/files/25_27.png  CheckedListBox | Множество элементов *CheckBox*, объединенных в скролирующую группу. |
| https://intuit.ru/EDI/05_07_17_1/1499206907-9549/tutorial/627/objects/25/files/25_28.png  ColorDialog | Стандартное диалоговое окно для предоставления пользователю возможности выбирать цвета |
| https://intuit.ru/EDI/05_07_17_1/1499206907-9549/tutorial/627/objects/25/files/25_29.png  ComboBox | Раскрывающийся список. Состоит из верхней части с редактируемым текстовым полем, и нижней части со списком для выбора готовых элементов. |
| https://intuit.ru/EDI/05_07_17_1/1499206907-9549/tutorial/627/objects/25/files/25_30.png  ContextMenuStrip | Контекстные меню обычно вызываются щелчком правой кнопкой мыши на нужном элементе управления формы. Туда обычно включают часто используемые команды из основного меню приложения. Это обеспечивает быстрый доступ пользователя к таким командам. |
| https://intuit.ru/EDI/05_07_17_1/1499206907-9549/tutorial/627/objects/25/files/25_31.png  DataGridView | Этот элемент отображает данные в виде таблицы, состоящей из строк и столбцов. В простейшем случае этот элемент просто отображает все данные связанной с ним одной таблицы базы данных без всяких условий и позволяет оперировать с ними как в обычной электронной таблице. В более сложном случае элемент отображает *выборку данных* из нескольких связанных таблиц по некоторым условиям. |
| https://intuit.ru/EDI/05_07_17_1/1499206907-9549/tutorial/627/objects/25/files/25_32.png  DateTimePicker | Этот элемент дает возможность пользователю быстро выбирать нужные дату из предоставляемого удобного интерфейса в виде календаря. |
| https://intuit.ru/EDI/05_07_17_1/1499206907-9549/tutorial/627/objects/25/files/25_33.png  DomainUpDown | Представляет собой текстовое поле со стрелками для изменения значений списка вверх или вниз. |
| https://intuit.ru/EDI/05_07_17_1/1499206907-9549/tutorial/627/objects/25/files/25_34.png  ErrorProvider | Этот элемент управления (невизуальный компонент) применяется для проверки правильности пользовательского ввода в допустимых пределах данных. Он предоставляет разработчику удобный интерфейс для оповещения пользователя о допущенных нарушениях и является лучшей заменой простому окну сообщений. |
| https://intuit.ru/EDI/05_07_17_1/1499206907-9549/tutorial/627/objects/25/files/25_35.png  FolderBrowserDialog | Представляет собой диалоговое окно для навигации по папкам файловой системы компьютера. |
| https://intuit.ru/EDI/05_07_17_1/1499206907-9549/tutorial/627/objects/25/files/25_36.png  FontDialog | Является стандартным диалоговоым окном, с помощью которого можно выбирать шрифты, установленные в настоящее время на текущем компьютере. |
| https://intuit.ru/EDI/05_07_17_1/1499206907-9549/tutorial/627/objects/25/files/25_37.png  *GroupBox* | Применяется для *связывания* элементов управления в единую группу как дочерних. Используется для наглядности компактного размещения логически связанных элементов и обеспечения их согласованной работы. Перемещение элемента *GroupBox* в режиме дизайна в пределах формы приводит к одновременному перемещению всех включенных в группу элементов. |
| https://intuit.ru/EDI/05_07_17_1/1499206907-9549/tutorial/627/objects/25/files/25_38.png  HelpProvider | Используется для *связывания* файлов справки (\*.chm, \*.htm) с приложением. |
| https://intuit.ru/EDI/05_07_17_1/1499206907-9549/tutorial/627/objects/25/files/25_39.png  HScrollBar | Самостоятельный элемент управления, который присоединяется к другим элементам интерфейса для продвижения по большим массивам информации в горизонтальном направлении. |
| https://intuit.ru/EDI/05_07_17_1/1499206907-9549/tutorial/627/objects/25/files/25_40.png  ImageList | Используется для хранения последовательности рисунков, подлежащих отображению приложением. Рисунки можно выбирать с помощью индексов списка и отображать на экране другими элементами управления. |
| https://intuit.ru/EDI/05_07_17_1/1499206907-9549/tutorial/627/objects/25/files/25_41.png  Label | Простая текстовая метка для отображения нередактируемого текста на экране компьютера. |
| https://intuit.ru/EDI/05_07_17_1/1499206907-9549/tutorial/627/objects/25/files/25_42.png  LinkLabel | Позволяет использовать текстовую метку в стиле гиперссылки. С помощью этого элемента часть текста можно связать с файлом, каталогом или Web-страницей. |
| https://intuit.ru/EDI/05_07_17_1/1499206907-9549/tutorial/627/objects/25/files/25_43.png  ListBox | Отображает список элементов, из которых пользователь может выбрать один или несколько элементов. Если количество элементов в вписке превышает то, что может быть отображено, то автоматически добавляется вертикальная *полоса прокрутки*. При установке свойства MultiColumn в значение true добавляется горизонтальная *полоса прокрутки*. Если установлено в true свойство ScrollAlwaysVisible, то *полоса прокрутки* появляется всегда, независимо от количества элементов. Свойство SelectionMode устанавливает, сколько элементов списка может быть выделено одновременно. Свойство SelectedIndex возвращает индекс первого выделенного элемента в списке, начиная с нуля. Если нет выделенных элементов, свойство возвращает значение -1. |
| https://intuit.ru/EDI/05_07_17_1/1499206907-9549/tutorial/627/objects/25/files/25_44.png  ListView | Отображает список элементов с иконками, наподобие правой области Windows Explorer. Элемент имеет четыре режима отображения: LargeIcon, SmallIcon, List, и *Details*. |
| https://intuit.ru/EDI/05_07_17_1/1499206907-9549/tutorial/627/objects/25/files/25_45.png  MenuStrip | Создает главное меню формы, за пунктами которого можно закреплять различные команды управления приложением. |
| https://intuit.ru/EDI/05_07_17_1/1499206907-9549/tutorial/627/objects/25/files/25_46.png  MonthCalendar | Обеспечивает привычный интерфейс работы с датой в виде календаря. |
| https://intuit.ru/EDI/05_07_17_1/1499206907-9549/tutorial/627/objects/25/files/25_47.png  NotifyIcon | Используется для отображения в виде значков процессов, которые выполняются в фоновом режиме и долгое время не видны пользователю. Примером может служить программа *антивирусной защиты*, которую при необходимости пользователь может вызвать щелчком на уведомляющем значке в панели задач. |
| https://intuit.ru/EDI/05_07_17_1/1499206907-9549/tutorial/627/objects/25/files/25_48.png  NumericUpDown | Представляет собой комбинацию текстового поля и пары стрелок, которые пользователь может применять для наглядного управления целым числовым значением. |
| https://intuit.ru/EDI/05_07_17_1/1499206907-9549/tutorial/627/objects/25/files/25_49.png  OpenFileDialog | Представляет стандартное диалоговое окно *открытия файла*. |
| https://intuit.ru/EDI/05_07_17_1/1499206907-9549/tutorial/627/objects/25/files/25_50.png  PageSetupDialog | Предоставляет стандартное диалоговое окно настройки страницы документа при печати. |
| https://intuit.ru/EDI/05_07_17_1/1499206907-9549/tutorial/627/objects/25/files/25_51.png  Panel | Элемент Panel подобен элементу *GroupBox*. Применяется для *связывания* элементов управления в единую группу как дочерних. Используется для наглядности компактного размещения логически связанных элементов и обеспечения их согласованной работы. Имеет собственные *полосы прокрутки*, но не отображает заголовок. |
| https://intuit.ru/EDI/05_07_17_1/1499206907-9549/tutorial/627/objects/25/files/25_52.png  PictureBox | Элемент служит для отображения рисунков в формате *bitmap*, GIF, *JPEG*, *metafile*, или *icon*. Отображаемый рисунок определяется свойством Image. Свойство SizeMode устанавливает режим согласования размеров элемента и рисунка. Рисунок отображается как в режиме разработки, так и во время выполнения приложения. |
| https://intuit.ru/EDI/05_07_17_1/1499206907-9549/tutorial/627/objects/25/files/25_53.png  PrintDialog | Это стандартное диалоговое окно используется для настроек принтера перед печатью. |
| https://intuit.ru/EDI/05_07_17_1/1499206907-9549/tutorial/627/objects/25/files/25_54.png  PrintDocument | Устанавливает, что печатать и как печатать. Может использоваться совместно с элементом PrintDialog для полного описания всех аспектов печати. |
| https://intuit.ru/EDI/05_07_17_1/1499206907-9549/tutorial/627/objects/25/files/25_55.png  PrintPreviewControl | Позволяет предварительно просмотреть документ перед печатью, как он будет выглядеть после печати. |
| https://intuit.ru/EDI/05_07_17_1/1499206907-9549/tutorial/627/objects/25/files/25_56.png  PrintPreviewDialog | Позволяет предварительно просмотреть документ перед печатью, как он будет выглядеть после печати. Отображает стандартное диалоговое окно с необходимыми элементами управления. |
| https://intuit.ru/EDI/05_07_17_1/1499206907-9549/tutorial/627/objects/25/files/25_57.png  ProgressBar | Индикаторная линейка - отображает продвижение процесса в горизонтальном направлении. |
| https://intuit.ru/EDI/05_07_17_1/1499206907-9549/tutorial/627/objects/25/files/25_58.png  RadioButton | Радиокнопка (переключатель) |
| https://intuit.ru/EDI/05_07_17_1/1499206907-9549/tutorial/627/objects/25/files/25_59.png  RichTextBox | Представляет собой полноценный текстовый редактор. |
| https://intuit.ru/EDI/05_07_17_1/1499206907-9549/tutorial/627/objects/25/files/25_60.png  SaveFileDialog | Обеспечивает стандартный настраиваемый диалог сохранения файлов. |
| https://intuit.ru/EDI/05_07_17_1/1499206907-9549/tutorial/627/objects/25/files/25_61.png  *Splitter* | Используется для изменения *относительных размеров* склеенных элементов управления во время выполнения приложения. |
| https://intuit.ru/EDI/05_07_17_1/1499206907-9549/tutorial/627/objects/25/files/25_62.png  StatusBar | Элемент создает строку состояния. |
| https://intuit.ru/EDI/05_07_17_1/1499206907-9549/tutorial/627/objects/25/files/25_63.png  TabControl | Элемент для создания вкладок. Создает одну или несколько вкладок, на каждой из которых могут быть размещены другие объекты. |
| https://intuit.ru/EDI/05_07_17_1/1499206907-9549/tutorial/627/objects/25/files/25_64.png  TextBox | Редактируемое текстовое поле. |
| https://intuit.ru/EDI/05_07_17_1/1499206907-9549/tutorial/627/objects/25/files/25_65.png  *Timer* | *Таймер* |
| https://intuit.ru/EDI/05_07_17_1/1499206907-9549/tutorial/627/objects/25/files/25_66.png  ToolStrip | Панель инструментов - контейнер для кнопок быстрого доступа. |
| https://intuit.ru/EDI/05_07_17_1/1499206907-9549/tutorial/627/objects/25/files/25_67.png  *ToolTip* | Для создания всплывающих коротких подсказок, которые появляются при наведении курсора на элемент управления. |
| https://intuit.ru/EDI/05_07_17_1/1499206907-9549/tutorial/627/objects/25/files/25_68.png  TrackBar | Регулятор для визуального изменения числовой информации (иногда называют *Slider* - бегунок). |
| https://intuit.ru/EDI/05_07_17_1/1499206907-9549/tutorial/627/objects/25/files/25_69.png  *TreeView* | Отображает информацию в древовидном формате. |

**Лекция 22-1. Виды, цели и уровни интеграции программных модулей.**

Интеграция программных модулей — процесс объединения различных компонентов программного обеспечения в единый функциональный комплекс.

**Роль программных модулей в разработке ПО**

Программные модули играют важную роль в разработке программного обеспечения (ПО). Они представляют собой отдельные компоненты, выполняющие определенные функции и обладающие четко определенным интерфейсом.

При разработке ПО модули позволяют разделить задачу на более мелкие и самостоятельные части. Это позволяет упростить процесс разработки и управление проектом в целом. Каждый модуль может разрабатываться независимо от других, что повышает эффективность работы команды разработчиков и позволяет параллельно выполнять несколько задач.

1. **Упрощение сопровождения и модификации**

Путем разделения ПО на модули, его сопровождение и модификация становятся более удобными и простыми. В случае необходимости изменения функциональности, можно модифицировать только соответствующий модуль, не затрагивая остальные компоненты. Это позволяет сэкономить время и ресурсы, а также избежать возможных ошибок при внесении изменений.

1. **Повторное использование кода**

Разработка ПО на основе модульной архитектуры позволяет повторно использовать уже написанные и протестированные модули. Это ускоряет процесс разработки, снижает вероятность появления ошибок и упрощает поддержку программного продукта.

1. **Улучшение понимания кода**

Модульная структура программы позволяет лучше понимать ее логику и устройство. Каждый модуль выполняет конкретную задачу, что делает их код читабельным и понятным для других разработчиков. Это помогает упростить командную разработку и обеспечить более высокое качество кода.

Программные модули предоставляют множество преимуществ в разработке ПО. Они позволяют упростить сопровождение и модификацию, повторно использовать код и улучшить понимание структуры программы. Использование модульной архитектуры становится все более популярным в современной разработке программного обеспечения, поскольку оно значительно облегчает разработку и поддержку ПО.

**Значение программных модулей для разработчиков**

Программные модули являются важной частью процесса разработки программного обеспечения. Они позволяют разделить функциональность программы на независимые и повторно используемые компоненты. Это дает разработчикам ряд преимуществ и упрощает процесс создания программного продукта.

1. **Модульность:** Программные модули позволяют организовать код программы по отдельным модулям, каждый из которых отвечает за определенную функциональность программы. Это позволяет разработчикам легко управлять и вносить изменения в код, так как они могут работать с отдельными модулями независимо от остальных частей программы.
2. **Повторное использование**: Благодаря модульности код можно повторно использовать в различных проектах. Разработчики могут создавать модули с общими функциями и компонентами, которые могут быть использованы в других проектах. Это сокращает время разработки и уменьшает количество ошибок, так как проверенный код может быть использован снова и снова.
3. **Улучшенная отладка**: Разделение кода на модули делает процесс отладки более эффективным. Если возникнет ошибка, разработчик может ограничиться отдельным модулем для проверки и исправления проблемы. Это упрощает процесс исправления ошибок и увеличивает общую стабильность программы.
4. **Совместная работа**: Программные модули позволяют нескольким разработчикам работать над одним проектом одновременно. Каждый разработчик может взять на себя ответственность за отдельные модули и работать над ними параллельно. Это улучшает эффективность командной работы и ускоряет процесс разработки.

В целом, использование программных модулей помогает разработчикам создавать более эффективное и надежное программное обеспечение. Модульность позволяет легко масштабировать и изменять программу, а повторное использование кода упрощает процесс разработки и сокращает время создания новых программных продуктов.

**Основные этапы интеграции модулей**

**Интеграция программных модулей** — это процесс объединения отдельных компонентов в единую систему. Разработка и установка каждого модуля выполняются независимо друг от друга, а затем происходит их объединение. Основные этапы интеграции модулей включают:

1. **Анализ и планирование** — в этом этапе разработчики определяют общую архитектуру системы и определяют роли и функции каждого модуля. Также они определяют интерфейсное взаимодействие между модулями.
2. **Разработка модулей** — каждый модуль разрабатывается отдельно. Разработчики должны следовать определенным спецификациям и стандартам, чтобы обеспечить совместимость и возможность интеграции.
3. **Тестирование модулей** — каждый модуль должен пройти тщательное тестирование, чтобы убедиться, что он работает правильно и выполняет свои функции. Тестирование может включать модульные тесты, интеграционные тесты и системные тесты.
4. **Интеграция модулей** — на этом этапе разработчики объединяют отдельные модули в единую систему. Это может быть сложным процессом, так как модули могут иметь разные структуры и способы взаимодействия.
5. **Тестирование системы** — после интеграции модулей всю систему необходимо протестировать еще раз, чтобы убедиться, что интеграция прошла успешно и система работает корректно.

**Отладка и исправление ошибок** — если в процессе тестирования обнаружены ошибки или проблемы, разработчики должны исправить их и убедиться, что система функционирует без ошибок.  
 Разработка программного обеспечения (ПО) является сложным и трудоемким процессом. Однако, современные подходы к разработке позволяют значительно увеличить эффективность работы команды разработчиков и сократить время, затрачиваемое на создание качественного ПО.

Основные понятия, связанные с интеграцией программных модулей, включают в себя:

1. **Интерфейсы:** способы взаимодействия между модулями программы, которые обеспечивают передачу данных и управление процессами;
2. **API (Application Programming Interface**): набор соглашений и инструментов, которые позволяют взаимодействовать с программными модулями;
3. **Модульная архитектура**: организация программы на отдельные функциональные блоки (модули), которые могут быть разработаны и тестированы отдельно, а затем интегрированы вместе;

**Лекция 22-2. Виды, цели и уровни интеграции программных модулей.**

Интеграция программных модулей — процесс объединения различных компонентов программного обеспечения в единый функциональный комплекс.

В зависимости от способа взаимодействия между модулями можно выделить несколько основных видов интеграции:

1. **Интеграция на основе API**. API (Application Programming Interface) — это набор готовых программных интерфейсов, методов и функций, предоставляемых разработчиком для взаимодействия с его программными модулями. При интеграции на основе API один модуль может использовать функциональность другого модуля, вызывая соответствующие методы или функции через API.
2. **Интеграция на основе сообщений**. При интеграции на основе сообщений модули взаимодействуют между собой, обмениваясь информацией через передачу сообщений. Каждый модуль может отправлять сообщения другим модулям, и получать сообщения от них. Этот вид интеграции позволяет обеспечить асинхронное взаимодействие между модулями.
3. **Интеграция на основе баз данных**. При интеграции на основе баз данных различные программные модули используют общую базу данных для хранения и обмена информацией. Каждый модуль имеет доступ к базе данных и может считывать из нее данные, а также записывать или изменять данные в базе. Это позволяет разным модулям работать с одними и теми же данными.
4. **Интеграция на основе файлового взаимодействия**. При интеграции на основе файлового взаимодействия модули обмениваются информацией через общие файлы. Каждый модуль может записывать данные в файл, а другие модули могут считывать эти данные из файла. Этот вид интеграции часто используется, когда требуется обмен данными между различными приложениями или системами. Какой вид интеграции использовать зависит от конкретной задачи и требований к программному комплексу. Интеграция программных модулей позволяет создавать более гибкие и функциональные системы, объединяя различные компоненты программного обеспечения в единое целое.

По степени автоматизации выделяются три уровня интеграции:

1. Ручная;
2. Автоматизированная;
3. Автоматическая.

По методу создания связи между интегрируемыми элементами системы рекомендуется выделять уровни:

1. Брокеров;
2. Данных;
3. Сервисов;
4. интерпретирования метаинформации.

Каждый уровень классификации имеет свои цели интеграции и инструменты и может быть рекомендован для решения соответствующего класса задач.

В зависимости от цели внедрения интеграцию, в первую очередь, подразделяют на внутреннюю и внешнюю. Внутренняя подразумевает добавление конкретных программных модулей без привлечения внешних ресурсов. Внешняя интеграция позволяет внутренние процессы синхронизировать с более глобальными, например, подключение к Google Maps для логистических компаний.

Доступны 3 вида интеграции:

* облачная;
* локальная;
* гибридная.

Кроме способов взаимодействия особое внимание следует уделить непосредственной организации маршрутов. Различают 2 подхода:

1. прямое взаимодействие – точка-точка;
2. звездообразная архитектура – хаб-спицы.

От выбранного маршрута напрямую зависит скорость и легкость обмена данными внутри системы.

Точка-точка. Этот подход характеризуется прямым взаимодействием. «Точка-точка» считается самым простым подходом и чаще всего используется для передачи большого объема данных. Важно следить за тем, чтобы приложения использовали одинаковые методы сообщения друг с другом.

Хаб-спицы. Ключевым моментом взаимодействия приложений при подходе «хаб-спицы» является наличие единого центрального узла. Он сам организует взаимодействие между интегрированными компонентами при помощи различных протоколов и методов, контролирует базу данных системы. Приложения связаны с хабом и не взаимодействуют друг с другом.

**Лекция 23.1. Выбор источников и приемников данных. Сопоставление объектов данных.**

Автоматизированная система электронной обработки данных или просто системы обработки данных (СОД) в классификации информационных систем по сложности, представляет собой простейший вид ИС, предназначенный для решения хорошо структурированных задач, по которым имеются входные данные, известны алгоритмы, ведущие к решению задач.

В основе СОД лежат две основные технологии работы с данными:

* данные собираются и обрабатываются специально для решения каждой отдельной задачи;
* для решения различных задач наряду со специфической для каждой задачи информацией используются общие условно-постоянные данные. В этом случае система называется интегрированной.

В настоящее время термин СОД довольно часто используется на уровне оперативного управления фирмой для решения задач автоматизации труда по отдельным управленческим функциям бухгалтерского учета, статистической отчетности, учета валютных операций в банке и т. п.

Основными функциями СОД являются:

* сбор данных и перенос их на машинные носители;
* передача в места хранения и обработки;
* хранение и обработка информации по стандартным алгоритмам;
* вывод и представление информации пользователю в виде регламентных форм.

Анализ внешней информационной среды предполагает:

* выявление источников необходимой информации и связей этих источников с информационной системой потребителя;
* оценку надежности и достоверности источников информации;
* определение объемов и формы представления информации;
* выяснение условий и особенностей предоставления информации источником.

Информационное обеспечение ИС является средством для решения следующих задач:

• однозначного и экономичного представления информации в системе (на основе кодирования объектов);

• организации процедур анализа и обработки информации с учетом характера связей между объектами (на основе классификации объектов);

• организации взаимодействия пользователей с системой (на основе экранных форм ввода-вывода данных);

• обеспечения эффективного использования информации в контуре управления деятельностью объекта автоматизации (на основе унифицированной системы документации).

Информационное обеспечение ИС включает два комплекса: внемашинное ИО (классификаторы технико-экономической информации, документы, методические инструктивные материалы) и внутримашинное ИО (макеты/экранные формы для ввода первичных данных в ЭВМ или вывода результатной информации, структуры информационной базы: входных, выходных файлов, базы данных).

**Классификация** – это разделение множества объектов на подмножества по их сходству или различию в соответствии с принятыми методами. Классификация фиксирует закономерные связи между классами объектов. Система классификации позволяет сгруппировать объекты и выделить определенные классы, которые будут характеризоваться рядом общих свойств. Таким образом, совокупность правил распределения объектов множества на подмножества называется системой классификации.

**Классификатор** – это документ, с помощью которого осуществляется формализованное описание информации в ИС, содержащей наименования объектов, наименования классификационных группировок и их кодовые обозначения.

Для того чтобы обеспечить точность и однозначность поиска с помощью дескрипторного языка, необходимо предварительно определить все постоянные отношения между терминами: родовидовые, отношения синонимии, омонимии и полисемии, а также ассоциативные отношения.

Все выделенные отношения явно описываются в систематическом словаре понятий – тезаурусе, который разрабатывается с целью проведения индексирования документов, показателей и информационных запросов.

Для поиска показателей и документов по набору содержательных признаков используется информационный язык дескрипторного типа, которой характеризуется совокупностью терминов (дескрипторов) и набором отношений между терминами.

**Лекция 23-2. Сопоставление объектов данных**

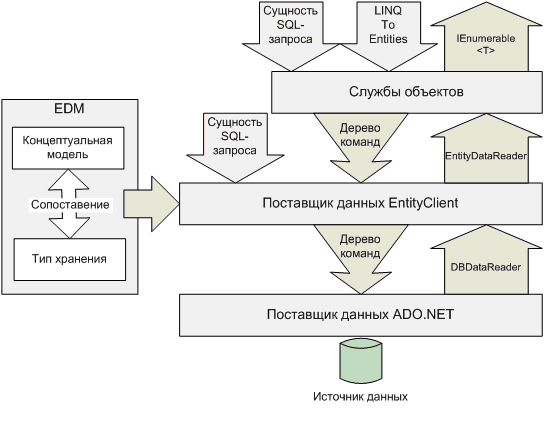
При использовании объектно-ориентированного программирования для взаимодействия с системами хранения данных возникают сложности. Безусловно, организация классов часто напоминает организацию таблиц реляционной базы данных, но такое соответствие не идеально. Несколько нормализованных таблиц часто соответствуют единственному классу, а связи между классами представлены иначе, чем связи между таблицами.

В существующих решениях была предпринята попытка устранить этот разрыв, часто называемый "несоответствием типов данных" (impedance mismatch), путем сопоставления с реляционными таблицами и столбцами только объектно-ориентированных классов и свойств. Вместо данного традиционного подхода в Entity Framework реляционные таблицы, столбцы и ограничения внешнего ключа логических моделей преобразуются в сущности и связи концептуальных моделей. Это позволяет достичь большей гибкости при определении объектов и оптимизации логической модели. С помощью инструментов Entity Data Model формируются расширяемые классы данных, основанные на концептуальной модели.

**Компоненты платформы Entity Framework**

Следующие функции и компоненты платформы Entity Framework работают совместно для обеспечения сквозной среды программирования.

* Модель Entity Data Model (EDM) служит центром приложения Entity Framework. Она задает схему проекта, которая используется для построения программируемых классов, используемых кодом приложения.
* Компонент Object Services позволяет программистам работать с классами CLR, созданными из концептуальной модели. Он также обеспечивает инфраструктурную поддержку для приложения Entity Framework, предоставляя такие службы, как управление состоянием, отслеживание изменений, разрешение идентификаторов, загрузка и переход по связям, распространение изменений объектов в модификации базы данных, а также поддержку запросов для Entity SQL.
* Компонент LINQ to Entities обеспечивает поддержку LINQ при запросах к сущностям. Компонент LINQ to Entities позволяет разработчикам писать запросы к базе данных на одном из поддерживаемых языков программирования .NET Framework, например Visual Basic или Visual C#.
* Язык Entity SQL подобен языку SQL и не зависит от типа хранилища. Он предназначен для создания запросов к сложным графам объектов, основанных на модели EDM, а также для управления ими.
* Поставщик EntityClient расширяет модель поставщика ADO.NET путем доступа к данным в терминах сущностей и связей концептуальной модели. Он выполняет запросы, которые используют язык Entity SQL. Entity SQL предоставляет базовый язык запросов, который позволяет поставщику EntityClient связываться с базой данных.
* Компонент метаданных ADO.NET управляет метаданными для всей платформы Entity Framework как во время разработки, так и во время выполнения. Все метаданные, связанные с моделями и сопоставлениями, доступны через интерфейсы метаданных, которые не зависят от механизма, используемого для хранения метаданных. Текущий механизм хранения использует файлы, которые основаны на трех диалектах XML: языке CSDL, языке SSDL и языке MSL.
* Приложение Entity Framework включает изменяющийся набор средств, которые создают сопоставления и разделяемые классы, представляющие сущности концептуальной модели.
* Приложение Entity Framework включает обновленный поставщик данных SqlClient, который поддерживает канонические деревья команд.



**Средства работы с моделью EDM**

Наряду со средой выполнения Entity Framework, .NET Framework 3.5 с пакетом обновления 1 включает генератор моделей EDM (EdmGen.exe). Программа командной строки соединяется с источником данных и формирует модель EDM на основе сопоставления типа "один к одному" между сущностями и таблицами. В этой программе используется также файл концептуальной модели (с расширением CSDL) для формирования файла уровня объектов, содержащего классы, которые представляют типы сущностей и контекст ObjectContext.

Visual Studio включает в себя обширный набор поддерживаемых инструментов для создания и обслуживания модели EDM в приложении Visual Studio. Конструктор Entity Data Model поддерживает создание усовершенствованных сценариев сопоставления (таких как наследование типа "одна таблица на тип" и "одна таблица на иерархию"), а также разделение сущностей, которые сопоставлены с несколькими таблицами.

**Определение наследования**

EDM также поддерживает модели, не соответствующие базе данных взаимно-однозначно. Например, используя базу данных Northwind, можно создать класс, например, называющийся DiscontinuedProducts, который наследует все свойства класса Products, но содержит только продукты, значение поля Discontinued для которых равно 1. Это упрощенная схема наследования, но она показывает, как применить наследование в EDM.

Первым шагом по созданию класса DiscontinuedProducts в концептуальной модели является открытие файла CSDL, создание нового типа EntityType, называющегося DiscontinuedProducts, и установка его атрибута BaseType в значение NorthwindModel.Products (схема и название базового EntityType ). Порожденный тип EntityType наследует свойства EntityType Products, включая ключи.

Дополнительный код CSDL для всего описанного выглядит так:

<EntityType Name="DiscontinuedProducts" BaseType="NorthwindModel.Products"/>

**Лекция 24. Организация файлового ввода/вывода.**

**Поток** - это некая абстракция производства или потребления информации.

С физическим устройством поток связывает

система ввода-вывода.

Все потоки действуют одинаково -- даже если

они связаны с разными физическими

устройствами (дисковый файл, сетевой канал, место в

памяти или любой другой объект, поддерживающий чтение и

запись в линейном режиме).

Существуют два типа потоков:

* Выходные
* Входные

**Байтовые и символьные потоки**

В среде .NET Framework определены классы как для байтовых, так и для символьных потоков. Но на самом деле классы символьных потоков служат лишь оболочками для превращения, заключенного в них байтового

потока в символьный, автоматически выполняя любые требующиеся преобразования типов данных. Следовательно, символьные потоки основываются на байтовых, хотя они и разделены логически. Основные классы потоков определены в пространстве имен System.IO.

**Порядок работы с файлом**

1. Подключить пространство имен, в

котором описываются стандартные

классы для работы с файлами.

2. Объявить файловую переменную и

связать ее с файлом на диске.

3. Выполнить операции ввода-вывода.

4. Закрыть файл.

**Символьный ввод-вывод в файл**

Несмотря на то что файлы часто обрабатываются побайтово, для этой цели

можно воспользоваться также символьными потоками. Преимущество символьных потоков заключается в том, что они оперируют символами непосредственно в юникоде. На вершине иерархии классов символьных потоков находятся абстрактные классы TextReader (организует ввод) TextWriter (организует вывод).

**Символьный ввод-вывод в файл**

Классы TextReader и TextWriter реализуютсянесколькими классами символьных потоков: StreamReader Предназначен для ввода символов

из байтового потока. Этот класс является оболочкой для байтового потока ввода StreamWriter Предназначен для вывода символов в байтовый поток. Этот класс является оболочкой для байтового потока вывода StringReader Предназначен для ввода символов из символьной строки StringWriter Предназначен для вывода символов в символьную строку.

**Чтение и запись двоичных данных**

В С# имеется также возможность читать и записывать другие типы данных. Например, можно создать файл, содержащий данные типа int, double или short. Для чтения и записи двоичных значений встроенных типов данных

служат классы потоков BinaryReader и BinaryWriter. Используя эти потоки, следует иметь в виду, что данные считываются и записываются во

внутреннем двоичном формате, а не в удобочитаемой текстовой форме.

**Класс BinaryWriter**

Класс BinaryWriter служит оболочкой, в которую заключается байтовый поток, управляющий выводом двоичных данных. Наиболее часто употребляемый конструктор этого класса: BinaryWriter(Stream output);

где output обозначает поток, в который выводятся записываемые данные. Для записи в выходной файл в качестве параметра output может быть

указан объект, создаваемый средствами класса FileStream.

Наиболее часто используемые методы, определенные в классе BinaryWriter:

Write(value), Seek(), Close() и Flush().

**Класс BinaryReader**

Класс BinaryReader служит оболочкой, в которую заключается байтовый поток, управляющий вводом двоичных данных.

Наиболее часто употребляемый конструктор этого класса: BinaryReader(Stream input); где input обозначает поток, из которого вводятся

считываемые данные. Для чтения из входного

файла в качестве параметра input может быть указан объект, создаваемый средствами класса FileStream.

**Буферизованный поток**

Буферизованный поток - это объект, позволяющий операционной системе создавать собственный внутренний буфер для обмена данными с внешней памятью и самостоятельно определять число байтов, считываемых или

записываемых за один раз. Программа по-прежнему будет заполнять буфер

указанными порциями, но данные в этот буфер будут поступать из внутреннего системного буфера, а не из внешней памяти. В результате

ввод/вывод будет происходить эффективнее и, следовательно, быстрее.