## *Тема 2.2.Системы и средства разработки программного обеспечения*

[**2.2.1. О средствах разработки**](#_2.2.1._О_средствах)

[**2.2.2. Основные средства разработки программ**](#_2.2.2._Основные_средства)

[**2.2.3. Понятие интегрированной среды разработки**](#_2.2.3._Понятие_интегрированной)

[**2.2.4. Интегрированная среда разработки VisualStudio .NET**](#_2.2.4._Интегрированная_среда)

[**2.2.5. Архитектура Framework .NET**](#_2.2.5.__Архитектура)

[**2.2.6. Framework .NET – единый каркас среды разработки**](#_2.2.6._Framework_.Net)

[**2.2.7. Среда исполнения программ**](#_2.2.7._Среда_исполнения)

2.2.7.1. Введение

2.2.7.2. Процессор

2.2.7.3. Оперативная память

2.2.7.4. Долговременное хранение информации

2.2.7.5. Классификация программных средств

2.2.7.6. Операционная система

#### **2.2.1. О средствах разработки**

Разработка программного обеспечения к настоящему моменту превратилась в обычный технологический процесс, на разных стадиях которого действуют подготовленные специалисты, применяющие в своей повседневной производственной практике различные технологии. Среди таких специализаций выделяются аналитики, маркетологи, менеджеры, кодировщики, тестировщики, специалисты по созданию документации и многие другие.

Отрасль, связанная с разработкой программного обеспечения, очень молода. Действительно, ей всего лишь около 50 лет, но накопленный в ней объем знаний, технологических решений, методик и просто практических рекомендаций к действию огромен. Кратко рассмотрим процесс производства программного обеспечения, т.е. рассмотрим, как пишется программный код, на каких принципах основан этот процесс, какие технологии применяются, в чем их содержательный смысл.

Представим себе ситуацию, что необходимо решить задачу нахождения среднего арифметического двух чисел. При этом действия могут быть следующие: находим алгоритм решения задачи, осознаем всю глубину заложенного в алгоритме математического аппарата, садимся за компьютер и начинаем писать программу. А на каком языке? Оказывается, прежде необходимо решить, на каком языке программирования, будет писаться программа. Решение это в реальной ситуации зависит от многих факторов. В рамках данного пособия будет изучаться язык программирования **VB 2008**.

А где писать текст (код) программы? А как превратить программу в нечто, что можно выполнить? Здесь ответ уже далеко не тривиален.

Перечень вопросов можно продолжить. Однако вопросов достаточно, чтобы подвести к важной мысли: для поддержки деятельности программиста необходимы специальные средства.

В любой отрасли существуют свои средства, упрощающие труд специалиста. Есть такие средства и в разработке программного обеспечения. Средства эти от года к году совершенствуются, причем, процесс их развития, как и всего остального в программной индустрии, необычайно стремителен.

Далее рассмотрим, какие именно средства помогают программистам в решении их профессиональных задач.

#### **2.2.2. Основные средства разработки программ**

Прежде всего, поговорим о тех средствах создания программ, без которых этот процесс вообще невозможен.

Что значит написать программу? Написать программу – это означает реализовать алгоритм, или иначе, представить его в виде понятных компьютеру указаний того, что необходимо делать. К сожалению, компьютеры не умеют понимать человека с полуслова, то есть словесное описание алгоритма необходимо превратить в абсолютно точный набор инструкций, однозначно интерпретируемых машиной.

Представьте себе, что Ваш друг никогда не слышал про метод половинного деления. Представьте теперь, что Вам жизненно необходимо довести этот метод до его сведения. Какие средства для этого имеются в Вашем распоряжении?

Конечно, прежде всего, это русский язык. Вы просто объясняете другу суть метода, пользуясь некой математической терминологией и обиходными словами русского языка. К сожалению, большая часть информации, необходимой программистам, присутствует именно на английском языке. Если Вы более или менее владеете языком, то для Вас не составит особого труда рассказать по-английски, как работает метод.

Итак, основное средство передачи информации от одного человека к другому – некоторый понятный обоим язык общения.

А как объяснить что-либо машине? Для того чтобы что-то объяснить компьютеру, люди создали специальные языки. Необходимо отметить, что процесс написания программ за последние 50 лет прошел путь от программирования в инструкциях процессора (программирование в машинных кодах) через программирование на низкоуровневых языках (ассемблер) до программирования на языках высокого уровня. Вот на них и остановимся чуть подробнее.

Что такое язык программирования высокого уровня? Чем он отличается от естественного языка?

С формальной точки зрения – это:

**Язык программирования = Синтаксис + Семантика.**

Для естественного языка расшифровать понятия синтаксиса и семантики можно следующим образом:

* **cинтаксис** – это раздел грамматики, изучающий внутреннюю структуру и общие свойства предложения;
* **cемантика** – это раздел языкознания, изучающий значения единиц языка.

Для языков программирования справедливы следующие определения:

* ***cинтаксис*** – это набор правил построения фраз алгоритмического языка, позволяющий определить осмысленные предложения в этом языке;
* ***cемантика*** – это система правил истолкования отдельных языковых конструкций и определяет смысловое значение предложений алгоритмического языка.

Заметим, что определения достаточно похожи по своему смыслу. Действительно, язык программирования – это искусственный (формальный) язык, предназначенный для записи алгоритмов. Язык программирования задается своим описанием и реализуется в виде специальной программы: компилятора или интерпретатора.

Таким образом, если обычные (естественные) языки предназначены для общения людей между собой, то языки программирования – для общения программиста с компьютером.

Что же означает словосочетание язык ***высокого уровня***, и чем языки программирования высокого уровня отличаются от языков ***низкого уровня***?

Говоря об уровнях, прежде всего, имеются в виду степени приближенности языка к машине. Уровень в данном случае – это уровень машинного восприятия. Так, языки низкого уровня (ассемблер) по возможности приближены к машине, что делает соответствующие программы особенно эффективными с точки зрения их быстродействия. Однако существенная проблема использования таких языков заключается в том, что программист - прежде всего человек, и его способы восприятия информации весьма далеки от машинных, что чрезвычайно затрудняет написание программ на ассемблере. В настоящий момент на ассемблере реализуются сравнительно небольшие участки программного кода, связанные преимущественно с программированием аппаратуры. Подавляющее большинство программ пишется на том или ином языке программирования высокого уровня. Такие программы существенно ближе к восприятию человека, наделенного некоторыми профессиональными навыками – программиста.

К настоящему времени создано большое число разных языков программирования высокого уровня, однако реально используются лишь некоторые из них. К числу активно применяемых языков относятся **C** и **C++, C#, Pascal, Java, VisualBasic**.

Под ***транслятором (translator)*** обычно понимают специальную программу, которая переводит код программы в последовательность машинных команд. Напомним еще раз: код программы понятен человеку, набор команд понятен компьютеру.

Заметим, что трансляторы языков высокого уровня, таких как **Pascal, C**, и других, обычно называют ***компиляторами (compiler)***. Этим подчёркивается общепринятый для промышленных языков режим трансляции, при котором вначале осуществляется перевод программы в двоичное представление, а лишь затем программа передаётся на исполнение. Другой способ трансляции, называемый ***интерпретация***, состоит в совмещении перевода и исполнения программы (в этом случае объектный модуль не сохраняется и его, соответственно, нельзя повторно использовать).

К числу основных достоинств компилируемых языков по сравнению с интерпретируемыми языками относятся:

* в компилируемых языках процесс построения (создания) исполняемого модуля выполняется один раз, а не при каждом запуске, что экономит время.
* в компилируемых языках обнаружение синтаксических ошибок происходит до запуска программы на выполнение, а не в его процессе.

Несмотря на очевидные недостатки интерпретируемых языков, они применяются в разных специфических задачах, а также в тех случаях, где простота программы важнее ее производительности (а программы на интерпретируемых языках почти всегда проще своих аналогов на языках компилируемых).

Уже в самом начале развития методов программирования стал применяться простой и эффективный приём выделения часто используемых алгоритмов в самостоятельные программы, получившие название библиотекой стандартных подпрограмм. Примером могут служить подпрограммы вычисления элементарных функций (синус, косинус и др.), а также процедуры обмена с внешними устройствами компьютера. Однажды составленные и откомпилированные, они в дальнейшем могут применяться программистами в своих задачах путём подсоединения их к разработанному коду основного алгоритма. В более широком плане эта идея нашла своё выражение в технологии модульного программирования, которую рассматривается в пособии подробно. В данный момент для нас важно обратить внимание на тот факт, что для обеспечения комплектации оттранслированной программы вспомогательными подпрограммами требуются специальные средства.

В систему программирования входит программа, называемая ***редактор связей***, которая обеспечивает поиск вспомогательных подпрограмм в специальных библиотеках программ и их присоединение к основной программе пользователя. Результатом работы редактора связей является полностью готовый к исполнению двоичный код программы, называемый ***загрузочным модулем***.

Загрузочный модуль может быть немедленно инициирован на выполнение, а может быть записан на диск и в дальнейшем многократно вызываться на выполнение с помощью специальной программы – загрузчика.

В систему программирования входит также программы, облегчающие отладку, а точнее, поиск ошибок – ***отладчики***. При всём многообразии реализаций ***отладчиков*** их основные возможности заключаются в так называемой трассировке работы программы.

***Трассировка*** – это отслеживание (ведение протокола) работы программы. В процессе трассировки программист может проследить порядок исполнения операторов, а также динамику изменения значений переменных программы.

В современных условиях отладка программ является не менее, а зачастую и более важным этапом разработки, чем собственно программирование (написание кода). Реальные задачи, пришедшие из разных областей человеческой деятельности, как правило, являются очень сложными. Объем программ, реализующих их решение, весьма велик. Такой код обычно создается большим коллективом разработчиков, в связи с чем, возникает много дополнительных проблем (в частности, необходимость поддерживать передачу информации между разными участниками проекта и согласовывать их деятельность). В конечном счете, сложность задач приводит к росту числа ошибок в программе. Известна старая шутка о том, что “любая программа содержит хотя бы одну ошибку”.

Но как все-таки определить, есть в программе ошибка или нет? Кажется, что самый простой способ можно сформулировать так: «**сейчас запустим и проверим**!». К сожалению, этот принцип применим лишь для очень ограниченного спектра задач. В реальных программистских компаниях существует целый штат сотрудников, которые занимаются этими проблемами (контролеры качества). Более того, теория в данной области не стоит на месте. Разработано несколько разных подходов к решению рассмотренных проблем.

Мы начали рассмотрение системы программирования с её ключевых частей – ***транслятора*** и ***редактора связей***, которые существовали с самого начала развития систем автоматизации программирования. А вот процесс составления программ долгое время оставался ручным. Программист записывал программу на специальном бланке, относил в отдел перфорации, где операторы с помощью специального оборудования наносили программу на перфокарты или перфоленты. С них программа загружалась в ЭВМ, запускалась, в ней обнаруживались ошибки, программист их исправлял, снова “набивал” перфокарты и так далее.

В настоящее время – время ***персональных компьютеров*** – этот рутинный процесс ушёл в прошлое. Современный программист, как правило, вводит код программы в компьютер, пользуясь так называемыми ***редакторами программного кода (редакторами текстов) или текстовыми процессорами.***

***Редактор кода*** – это программная система, обеспечивающая первоначальную подготовку исходного текста программы и его исправление в процессе разработки. В отличие от универсальных ***текстовых процессоров*** (самым известным из них является **MSWord**), редакторы кода специализированы для работы именно с исходными текстами программ, поэтому они не имеют массы функций обычных редакторов (вроде работы с таблицами и рисунками), зато предоставляют другие специальные функции, не менее полезные. Существует довольно большое количество различных редакторов кода, начиная от простого набора текста, комбинирования отдельных фрагментов, поиска по образцу, выделения цветом различных элементов программы и заканчивая автоматическим форматированием в соответствии с устоявшимися правилами оформления кода для того или иного языка программирования (эти правила часто называют ***стилем***).

Подготовленная с помощью редактора текстов программа запоминается в виде одного или нескольких файлов и в дальнейшем служит входной информацией для транслятора.

К счастью, в разработке программ существует возможность частично избавить себя от рутины с помощью, так называемых ***средств автоматизированной генерации кода***, в некоторых случаях они умеют самостоятельно создавать программный код, выполняющий определенные стандартные действия. Примером таких средств может служить, например, **Microsoft VisualStudio .NET**, которая автоматически создает и заполняет полями класс «**Форма**» при создании нами нового окна и наполнении его различными компонентами.

Представьте себе, что Вы в сотрудничестве с другими программистами создали некую программную систему. После установки ее на реальном объекте (завод, магазин, склад) выяснилось, что при увеличении объемов обрабатываемых данных Ваша программа работает слишком медленно. Вопрос: как узнать, почему это происходит. Где в программном коде те «узкие места», на которые приходится основное время выполнения? Как повысить быстродействие?

Для того чтобы получить ответ на этот вопрос, существуют специальные программы, называемые ***профилировщиками***. При помощи этой и других подобных программ Вы можете серьезно повысить производительность программы, внеся в нее необходимые изменения.

Необходимость в создании документации неизбежно возникает в любом технологическом процессе. Процесс разработки программного обеспечения не является исключением. На всех этапах этого процесса создается масса документов различной направленности. Это могут быть документы: управленческие (инструкции, приказы); юридические (лицензии); постановочные (техническое задание, требования к системе); проектные (проект программного комплекса); описательные, предназначенные для конечного пользователя (руководство пользователя); и, наконец, внутренняя документация, описывающая, как создавалась система, какова ее архитектура, из каких модулей она состоит, что в них находится.

Хорошая инструкция – это не то, что может понять только автор инструкции. Это не книга в 2000 страниц, глядя на которую будущему пользователю хочется быстрее забыть про программный продукт. Хорошая инструкция – это то, при помощи чего можно быстро и самостоятельно научиться пользоваться приобретенной продукцией (в том числе и программным обеспечением). Как обстоит дело с написанием этих инструкций? Как правило, программисты любят писать программы, но не любят писать инструкции. Представьте себе, что один из программистов уволился, а на его место был вынужденно принят на работу новый сотрудник. Как теперь он сможет разобраться, как устроено то, что ему надлежит доделывать и, возможно, в чем-то переделывать? Без наличия документации попытка разобраться в чужой программе во многом равносильна ее переписыванию с нуля.

Таким образом, создание документации – задача не менее важная, чем создание программного кода, и в ее решении нам помогают различные программные средства.

#### **2.2.3. Понятие интегрированной среды разработки**

122 ***Интегрированная среда разработки*** (**IDE**–Integrated Development Environment) – это специальная программа, предоставляющая возможность удобной совместной работы с различными компонентами системы программирования.

Ранее мы рассмотрели компоненты, входящие в систему программирования. Это и ***редакторы кода***, и ***компиляторы***, и ***сборщик***и, и ***отладчики***, и многие другие. При первом же знакомстве со всеми этими программами становится понятно, что каждая из них может работать с разными начальными установками. Так, например, можно настроить множество параметров для редактора кода: цвет фона, цвет шрифта, шрифт, размер символа табуляции и еще сотню разных характеристик. Для компилятора можно указать, как оптимизировать код: по скорости, по размеру, никак не оптимизировать, а также есть возможность управления многими другими параметрами. Аналогично обстоит дело практически со всеми составляющими системы программирования.

Теперь представьте себе, как можно по очереди запускать все эти программы с огромным количеством разных параметров. То есть, сначала запускается редактор кода и пишется в нем программа. После этого подготовленная программа сохраняется, а затем закрывается редактор. Далее запускается компилятор, указав ему в качестве параметра файл с текстом программы и все необходимые настройки. Например, компилятор отработал и нашел 4 ошибки. Снова запускается текстовый редактор и в результате титанических усилий находятся эти строки и ошибки в них. После этого снова сохраняется программа, закрывается редактор и опять запускается компилятор. В результате компилятор создает ***объектный код***, на этот раз без синтаксических ошибок. Теперь запускается сборщик, указывая ему кучу параметров и тот самый объектный файл. Если ошибок нет, то, наконец, получится ***исполняемый файл***, запускаете его и программа запустилась и «повисла». Или не повисла, но сказала, что уравнение не имеет корней, хотя точно известно, что решение есть. Это значит, что с семантикой что-то не то, иначе говоря, программа работает неправильно и ее необходимо отлаживать, искать ошибки. Для этого у нас есть ***отладчик***. И т.д. и т.п.

Для устранения неудобств и повышения эффективности процесса разработки создатели систем программирования стали строить их в виде так называемых ***интегрированных сред разработки***. Термин «***интегрированная***» в названии среды означает, что она включает в себя в качестве элементов все необходимые инструменты для выполнения полного цикла работ над программой: написания, компиляции, построения исполняемого модуля, запуска, отладки. Кроме того, интегрированные среды позволяют выполнять следующие операции:

* визуально (в диалоге) производить быструю настройку параметров каждого из компонентов системы программирования;
* сохранять разные системы настроек и загружать их по мере необходимости;
* нажатием нескольких клавиш или выбором соответствующих пунктов меню осуществлять запуск одного или сразу нескольких компонентов системы программирования, автоматизируя процесс передачи им необходимых параметров.

Так, в любой интегрированной среде исполняемый модуль из исходного текста программы можно получить нажатием пары кнопок на клавиатуре. Единственный минус таких сред является прямым следствием их главного плюса – собрав «под одной крышей» большой набор инструментов, интегрированная среда сама становится весьма сложной программой. Однако время, потраченное на ее изучение, окупается в дальнейшем. И, наконец, еще один положительный момент – устройство большинства сред одинаково в концептуальном плане, различия наблюдаются лишь в комбинациях клавиш для того или иного действия да в названиях пунктов меню.

Одно из последних достижений в области разработки программного обеспечения – визуальные среды программирования (самые известные – **Borland® Delphi™** с базовым языком **ObjectPascal** и многоязыковая среда **Microsoft® Visual Studio.NET**). Их появление связано с двумя важными факторами. Во-первых, это стремление человека максимально автоматизировать собственный труд. Второй фактор связан с тем, что современный пользователь в большинстве своем не станет работать с программой, которая не удовлетворяет его «чувство прекрасного». Говоря серьезно, сейчас при создании программ их внешнему виду уделяется не меньшее значение, чем внутреннему содержанию. Визуальные среды и тут приходят на помощь.

#### **2.2.4. Интегрированная среда разработки Visual Studio .NET**

Microsoft предлагает **Visual Studio .NET 2008** в пяти изданиях: **Express, Standard, Professional**, **TeamSystem** и для **Microsoft Office**, которые покрывают потребности всех категорий разработчиков – от начинающих программистов до профессионалов высшего класса.

**Express** представляет собой набор идентичных по составу продуктов, но с поддержкой только одного из языков – **VB, VC#**или **VC++**. Это упрощенные варианты средства для создания только **Windows**-приложений. Кроме того, имеется инструмент для создания динамических **Web-**сайтов и **Web**-сервисов – **Visual Web Developer 2008 Express Edition**(с возможностью программирования на **VB** и **VC#**). В состав всех этих продуктов входит **SQL Server 2005 Express Edition**. В целом они и предназначены в первую очередь для начинающих программистов, любителей или для использования в учебных целях.

Разница между изданиями **Standard** и **Professional** для многих целей разработки представляется минимальной, серьезные различия видны, только если речь идет о создании достаточно сложных решений. Оба варианта позволяют разрабатывать **Windows**- и **Web**-приложения и программное обеспечение (**ПО)** для мобильных устройств. В **Professional Edition** имеются все возможности **Standard** плюс средства удаленной отладки, генератор отчетов **Crystal Reports**, **SQL Server Developer Edition**, **SQL Server Integration** и расширенные инструменты для развертывания **ПО**.

**Visual Studio 2008 Team System** – это высокопроизводительный, интегрированный, расширяемый набор средств поддержки полного жизненного цикла командной разработки и приложений корпоративного уровня.

Кроме того, в линейку **Visual Studio** входит инструмент **Visual Studio 2008ToolsfortheMicrosoftOfficeSystem** – эффективное средство профессиональной разработки решений на базе приложений **Office System 2008 (Excel, Word, Outlook** и **InfoPath**). Программирование выполняется на **VB** и **VC#.**

В состав **VS 2008** входят компиляторы трех языков – **VB, VC#** и **VC++**. И все же именно первые два фактически олицетворяют собой этот**.** Ведь **VC++,** несмотря на появление в нем расширений для создания управляемого кода, все же ориентирован на разработку программ в классической архитектуре **Win API**, и его роль в создании прикладных решений заметно снижается.

Поддержка ***рефакторинга*** – одно из наиболее существенных новшеств **VS IDE**. Это позволяет автоматизировать процедуры переименования, перемещения и изменения типизированных элементов кода, исправления порядка описания параметров, имплементации программного интерфейса и ряд других операций. Все это выполняется с помощью нового инструмента **ClassDesigner**, который позволяет визуально представить структуру классов приложения или создать приложение путем визуального определения классов. Если щелкнуть правой кнопкой мыши диаграмму классов, то далее в меню можно выбрать вид операции ***рефакторинга***. Далее выполняются необходимые изменения в диаграмме классов, которые также автоматически корректируют исходный код программы.

**ClassDesigner** может рассматриваться как некий базовый инструмент создания приложений, который сводит к минимуму процесс кодирования, используя вместо этого методы визуального проектирования. Так, переместив новый класс с панели инструментов **Toolbox** на форму конструктора, можно затем визуально добавлять методы, определять взаимосвязи между классами, формировать интерфейсы, создавать структуры и т. п.

#### **2.2.5. Архитектура Framework.NET**

Платформа **.NET Framework** состоит из четырех групп программных продуктов:

* ***набор базовых языков***, куда входят **С#, С++** и **Visual Basic .NET**; набор инструментальных средств разработки, в том числе **Visual Studio .NET**; обширная библиотека классов для построения **Web**-служб и приложений, работающих в **Windows** и в **Интернете**; а также среда выполнения программ **CLR** (**Common Language Runtime, общеязыковая среда выполнения**), в которой выполняются объекты, построенные на этой платформе;
* ***набор серверов .NET Enterprise Servers***, которые предоставляют специализированные функциональные возможности для обращения к реляционным базам данных, использования электронной почты, оказания коммерческих услуг и т. д.;
* ***богатый выбор Web-служб, называемых .Net MyServices***. За умеренную плату разработчики могут пользоваться этими службами при построении приложений, требующих идентификации личности пользователя и других данных;
* ***новые некомпьютерные устройства***, поддерживающие средства **.NET**, от сотовых телефонов до игровых приставок.

**Microsoft .NET** поддерживает не только языковую независимость, но и языковую интеграцию. Это означает, что разработчик может наследовать от классов, обрабатывать исключения и использовать преимущества полиморфизма при одновременной работе с несколькими языками. Платформа **.NET Framework** предоставляет такую возможность с помощью спецификации **Common Type System(CTS, общая система типов)**, которая полностью описывает все типы данных, поддерживаемые средой выполнения, определяет, как одни типы данных могут взаимодействовать с другими и как они будут представлены в формате **.NET**. Например, в **.NET** любая сущность является объектом какого-нибудь класса, производного от корневого класса **System.Object**. Спецификация **CTS** поддерживает такие общие понятия, как классы, делегаты, ссылочные и размерные типы.

Важно понимать, что не во всех языках программирования **.NET** обязательно должны поддерживаться все типы данных, которые определены в **CTS**. Спецификация Common Language Specification **(CLS,** общая языковая спецификация**)** – устанавливает основные правила, определяющие законы, которым должны следовать все языки, такие как ключевые слова, типы, примитивные типы, перегрузки методов и т.п. Спецификация **CLS** определяет минимальные требования, предъявляемые к языку платформы **.NET**. Компиляторы, удовлетворяющие этой спецификации, создают объекты, способные взаимодействовать друг с другом. Любой язык, соответствующий требованиям **CLS**, может использовать все возможности библиотеки **FCL** (Framework ClassLibrary, библиотека классов платформы). **CLS** позволяет и разработчикам, и поставщикам, и производителям ПО не выходить за пределы общего набора правил для языков, компиляторов и типов данных.

Платформа **.NET Framework** является надстройкой над операционной системой, в качестве которой может быть **Windows**. Платформа **.NET Framework** включает в себя:

* три официальных языка: **С#**, **VB.NET**, **VC++** ;
* объектно-ориентированную среду **CLR** (Common Language Runtime), совместно используемую этими языками для создания приложений под **Windows** и для **Internet**;
* ряд связанных между собой библиотек классов под общим именем **FCL** (**Framework Class Library**).

Архитектурные компоненты платформы **.NET Framework** представлены на   
рис.2.2.1-1.



Рис. 2.2.5-1. Архитектура .NETFramework.

Самым важным компонентом платформы **.NET Framework** является **CLR(Common Language Runtime)**, предоставляющим среду, в которой выполняются программы. Главная ее роль заключается в том, чтобы обнаруживать и загружать типы .NET и производить управление ими в соответствии с полученными командами. На верхнем уровне среда активизирует объекты, производит проверку безопасности, размещает объекты в памяти, выполняет их, а также запускает сборщик мусора.

Под сборкой мусора понимается освобождение памяти, занятой объектами, которые стали бесполезными и не используются в дальнейшей работе приложения. В ряде языков программирования (например, **C++**) память освобождает сам программист, в явной форме отдавая команды, как на создание, так и на удаление объекта. Однако в **CLR** задача сборки мусора (и другие вопросы, связанные с использованием памяти) решается в нужное время и в нужном месте исполнительной средой, ответственной за выполнение вычислений.

На рис.2.2.5.-1 над уровнем **CLR** находится набор базовых классов платформы, а над ним расположены слой классов данных и **XML**, а также слой классов для создания **Web**-служб (**Web Services**), **Web**- и **Windows**-приложений (**Web Forms** и **Windows Forms**). Собранные воедино, эти классы известны под общим именем **FCL (Framework Class Library).** Это одна из самых больших библиотек классов в истории программирования. Она открывает доступ к системным функциям, также к прикладным функциям для **Web**-разработки (**ASP.NET**), для доступа к данным (**ADO.NET**), обеспечения безопасности и удаленного управления. Имея в своем составе более 4000 классов, библиотека **FCL** способствует быстрой разработке настольных, клиент-серверных и других приложений и   
**Web**-служб.

Набор базовых классов платформы – нижний уровень **FCL** – не только прячет обычные низкоуровневые операции, такие как файловый ввод-вывод, обработка графики и взаимодействие с оборудованием компьютера, но и обеспечивает поддержку большого количества служб, используемых в современных приложениях (управление безопасностью, поддержка сетевой связи, управление вычислительными потоками, работа с коллекциями и т.д.).

Над этим уровнем находится уровень классов, которые расширяют базовые классы с целью обеспечения управления данными и **XML**. Классы данных позволяют реализовать управление информацией, хранящейся в серверных базах данных. В число этих классов входят классы **SQL** (**Structured Query Language**, язык структурированных запросов), дающие возможность обращаться к долговременным хранилищам данных через стандартный интерфейс запросов **SQL**. Кроме того, набор классов, называемый **ADO.NET**, позволяет оперировать постоянными данными. Платформа **.NET Framework** поддерживает также целый ряд классов, позволяющих манипулировать **XML**-данными и выполнять поиск и преобразования **XML**.

Базовые классы, классы данных и **XML** расширяются классами, предназначенными для построения приложений на основе трех различных технологий: **WebServices** (**Web**-**службы**), **WebForms** (**Web**-**формы**) и **WindowsForms** (**Windows-формы**). Поскольку **Web**-**службы** применяют в качестве базовых протоколов связи стандартные протоколы **HTTP** и **SOAP**, эти компоненты поддерживают в киберпространстве подход «**Plug-and-Play**». Напомним, что **PlugandPlay**, дословно переводится как «включил и играй (работай)»  – технология, предназначенная для быстрого определения и конфигурирования устройств в [компьютере](http://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80).

Инструментальные средства **WebForms** и **WindowsForms** позволяют применять технику **Rapid Application Development** (**RAD**, быстрая разработка приложений) для построения **Web** - и **Windows**-**приложений**. Эта техника сводится к перетаскиванию элементов управления с панели инструментов на форму, двойному щелчку по элементу и написанию кода, обрабатывающего события, связанные с этим элементом.

**.NET**-приложения исполняются иначе, чем традиционные **Windows**-приложения. Такие программы компилируются фактически в два этапа. На первом этапе исходный код компилируется во время построения проекта и вместо исполняемого файла с машинными кодами получается **сборка (assembly)**, содержащая команды промежуточного языка **Microsoft Intermediate Language**– (**MS IL**, промежуточный язык **Microsoft**). Код **IL** сохраняется в файле на диске. При этом, файлы **IL**, генерируемые компилятором, например, **VB**, идентичны **IL**-файлам, генерируемым компиляторами с других языков **.NET**. В этом смысле платформа остается в неведении относительно языка. Самой важной характеристикой среды **CLR** является то, что она ***общая****;* одна среда выполняет как программы, написанные на **С#**, так и программы на языке **VB.NET**.

Напомним, что **сборка (assembly)** – это коллекция файлов, которая предстает перед программистом в виде единой библиотеки динамической компоновки (**DLL**) или исполняемого файла (**EXE**); в технологии **.NET** сборка является базовой единицей для повторного использования, контроля версий, защиты и развертывания. Среда CLR представляет программисту ряд классов, позволяющих манипулировать сборками.

Второй этап компиляции наступает непосредственно перед фактическим выполнением страницы. На этом этапе **CLR** транслирует промежуточный код **IL** в низкоуровневый собственный машинный код, выполняемый процессором. Процесс происходит следующим образом: при выполнении .**NET**-программы системы **CLR**активизирует **JIT**-компилятор, который затем превращает **MSIL** во внутренний код процессора. Этот этап известен как оперативная компиляция «точно к нужному моменту» (**Just-In-Time**) или **JIT**-компиляция (**JIT'ing**) и он проходит одинаково для всех приложений **.NET** (включая, например, приложения Windows). При исполнении программы **CLR** берет на себя управление памятью, контроль типов и решает за приложение ряд других задач. На  
 рис.2.2.5-2 показан этот двухэтапный процесс компиляции.



Рис. 2.2.5-2. Схема компиляции .NET-приложения.

Стандартный **JIT**-**компилятор** работает ***по запросу****.* Когда вызывается тот или иной метод, **JIT**-компилятор анализирует **IL**-код и производит высокоэффективный машинный код, выполняемый очень быстро. **JIT**-**компилятор** достаточно интеллектуален, чтобы распознать, был ли код уже скомпилирован, поэтому во время выполнения программы компиляция происходит лишь при необходимости. По ходу своего выполнения **.NET**-программа работает все быстрее и быстрее, так как повторно используется уже скомпилированный код.

Спецификация **CLS** подразумевает, что все языки платформы **.NET** генерируют очень похожий **IL**-код. Кроме того, при компилировании программы в дополнение к **MSIL**формируется еще один компонент – метаданные, которые описывают данные, используемые программой, что позволяет коду взаимодействовать с другим кодом. В результате объекты, созданные на одном языке, доступны и могут наследоваться на другом. То есть, можно создать базовый класс на языке **VB.NET**, а производный от него класс – на языке **С#.**

В целом, при написании приложения создается так называемый управляемый код (**managed code**), который выполняется под контролем не зависящей от языка среды исполнения **CLR** приложения. Поскольку приложение запускается под контролем **CLR**, управляемый код должен соответствовать определенным требованиям. То есть, компилятор должен создать **MS IL**-**файл**, предназначенный для **CLR**, а также использовать библиотеки **.Net Framework**, при выполнении которых он получает множество преимуществ, включая современное управление памятью, способность совмещать языки, высокий уровень безопасности передачи данных, поддержку контроля версии и понятный способ взаимодействия компонентов программного обеспечения. Библиотека классов **.NET** открывает доступ ко всем возможностям **CLR**. Классы, составляющие эту библиотеку, организованы при помощи пространств имен. Причем каждое пространство имен заключает в себе классы, выполняющие близкие функции.

Таким образом, компиляция **.NET** делится на два этапа с целью предоставления разработчикам удобных условий и мобильности. Перед созданием низкоуровневого машинного кода компилятору необходимо знать, в какой операционной системе и на каком базовом оборудовании будет функционировать приложение. Благодаря двум этапам компиляции можно создать скомпилированную сборку с кодом **.NET** и распределить ее более чем на одну платформу.

Конечно, компиляция не будет столь полезна, если ее выполнение будет необходимо каждый раз при запросе пользователем **Web**-страницы. К счастью, приложения **ASP.NET** не нужно компилировать всякий раз при запросе **Web**-страницы или **Web**-службы. Вместо этого код **IL** создается один раз и повторно генерируется только при изменении исходного кода. Подобным образом файлы собственного машинного кода кэшируются в системном каталоге.

#### **2.2.6. Framework .Net – единый каркас среды разработки**

***Открытость***. Среда разработки программных проектов является открытой языковой средой. Это означает, что наряду с языками программирования, включенными в среду фирмой **Microsoft – Visual C++ .Net** (с управляемыми расширениями), **Visual C# .Net , Visual Basic .Net**, – в среду могут добавляться любые языки программирования, компиляторы которых создаются другими фирмами.

Таких расширений среды **Visual Studio** сделано уже достаточно много, практически они существуют для всех известных языков программирования.

Открытость среды не означает полной свободы. Все разработчики компиляторов при включении нового языка в среду разработки должны следовать определенным ограничениям. Главное ограничение, которое можно считать и главным достоинством, состоит в том, что все языки, включаемые в среду разработки **Visual Studio .Ne**t должны использовать единый каркас – **Framework .Net**. Благодаря этому достигаются многие желательные свойства: легкость использования компонентов, разработанных на различных языках; возможность разработки нескольких частей одного приложения на разных языках; возможность бесшовной отладки такого приложения; возможность написать класс на одном языке, а его потомков – на других языках. Единый каркас приводит к сближению языков программирования, позволяя вместе с тем сохранять их индивидуальность и имеющиеся у них достоинства. Преодоление языкового барьера – одна из важнейших задач современного мира**. Visual Studio .Net**, благодаря единому каркасу, в определенной мере решает эту задачу в мире программистов.

В каркасе **Framework .Net** можно выделить два основных компонента:

* статический – **FCL** – библиотеку классов каркаса;
* динамический – **CLR** – общеязыковую среду исполнения.

***Библиотека классов FCL – статический компонент каркаса***. Понятие каркаса приложений – ***Framework Applications*** появилось достаточно давно, оно широко использовалось еще в четвертой версии **Visual Studio**.

За прошедшие годы роль каркаса в построении приложений существенно возросла, прежде всего, за счет появления его динамического компонента. Что же касается статического компонента – библиотеки классов, то здесь появился ряд важных нововведений.

***Единство каркаса***. Каркас стал единым для всех языков среды разработки. Поэтому, на каком бы языке программирования не велась разработка, она использует классы одной и той же библиотеки. Многие классы библиотеки, составляющие общее ядро, используются всеми языками. Отсюда единство интерфейса приложения, на каком бы языке оно не разрабатывалось, единство работы с коллекциями и другими контейнерами данных, единство связывания с различными хранилищами данных и прочая универсальность.

***Встроенные примитивные типы***. Важной частью библиотеки **FCL** стали классы, задающие примитивные типы, те типы, которые считаются встроенными в язык программирования. Типы каркаса покрывают основное множество встроенных типов, встречающихся в языках программирования. Типы языка программирования проецируются на соответствующие типы каркаса. Тип, называемый в языке **Visual Basic – Integer**, а в языках **С++** и **C#** - **int**, проецируется на один и тот же тип каркаса **System.Int32**. В языке программирования, наряду с "родными" для языка названиями типов, разрешается пользоваться именами типов, принятыми в каркасе. Поэтому, по сути, все языки среды разработки могут пользоваться единой системой встроенных типов, что, конечно, способствует облегчению взаимодействия компонентов, написанных на разных языках.

***Структурные типы***. Частью библиотеки стали не только простые встроенные типы, но и структурные типы, задающие организацию данных – строки, массивы, перечисления, структуры (записи). Это также способствует унификации и реальному сближению языков программирования.

***Архитектура приложений***. Существенно расширился набор возможных архитектурных типов построения приложений. Помимо традиционных **Windows**- и консольных приложений, появилась возможность построения **Web**-приложений. Большое внимание уделяется возможности создания повторно используемых компонентов – разрешается строить библиотеки классов, библиотеки элементов управления и библиотеки **Web**-элементов управления.

***Модульность***. Число классов библиотеки **FCL** велико (несколько тысяч). Поэтому понадобился способ их структуризации. Логически классы с близкой функциональностью объединяются в группы, называемые ***Пространством имен*** (**Namespace**). Основным пространством имен библиотеки **FCL** является пространство **System**, содержащее как классы, так и другие вложенные пространства имен. Так, уже упоминавшийся примитивный тип **Int32** непосредственно вложен в пространство имен **Sys**t**em** и его полное имя, включающее имя пространства – **System.Int32**.

В пространство **System** вложен целый ряд других пространств имен. Например, в пространстве **System.Collections** находятся классы и интерфейсы, поддерживающие работу с коллекциями объектов – списками, очередями, словарями. В пространство **System.Collections,** в свою очередь, вложено пространство имен **Specialized,** содержащие классы со специализацией, например, коллекции, элементами которых являются только строки. Пространство **System.Windows.Forms** содержит классы, используемые при создании Windows-приложений. Класс **Form** из этого пространства задает форму – окно, заполняемое элементами управления, графикой, обеспечивающее интерактивное взаимодействие с пользователем.

***Общеязыковая исполнительная среда CLR*** – динамический компонент каркаса. Важным шагом в развитии каркаса **Framework .Net** стало введение динамического компонента каркаса - исполнительной среды CLR. С появлением **CLR** процесс выполнения приложений стал принципиально другим.

***Двухэтапная компиляция. Управляемый модуль и управляемый код***. Компиляторы языков программирования, включенные в **Visual Studio .Net**, создают код на промежуточном языке **IL** (Intermediate Language) – ассемблерном языке. В результате компиляции проекта, содержащего несколько файлов, создается так называемый управляемый модуль – переносимый исполняемый файл (Portable Executable или **PE**-файл). Этот файл содержит код на **IL** и метаданные – всю информацию, необходимую для **CLR**, чтобы под ее управлением **PE**-файл мог быть исполнен. Метаданные доступны и конечным пользователям. Классы, входящие в пространство имен **Reflection**, позволяют извлекать метаинформацию о классах, используемых в проекте. Этот процесс называется отражением. Об атрибутах классов, отображаемых в метаданные **PE**-файла, еще будем говорить неоднократно. В зависимости от выбранного типа проекта, **PE**-файл может иметь разные уточнения - **exe, dll, mod** или **mdl**.

Необходимо отметить, что **PE**-файл, имеющий уточнение **exe**, хотя и является **exe**-файлом, но это не обычный исполняемый **Windows** файл. При его запуске он распознается как **PE**-файл и передается **CLR** для обработки. Исполнительная среда начинает работать с кодом, в котором специфика исходного языка программирования исчезла. Код на **IL** начинает выполняться под управлением **CLR** (по этой причине **код** называется **управляемым**). Исполнительную среду следует рассматривать как виртуальную **IL**-машину. Эта машина транслирует «на лету» требуемые для исполнения участки кода в команды реального процессора, который в действительности и выполняет код.

***Виртуальная машина***. Отделение каркаса от студии явилось естественным шагом. Каркас **Framework .Net** перестал быть частью студии, а стал надстройкой над операционной системой. Теперь компиляция и создание **PE** модулей на **IL** отделено от выполнения, и эти процессы могут быть реализованы на разных платформах.

* В состав **CLR** входят трансляторы **JIT** (JustInTimeCompiler), которые и выполняют трансляцию **IL** в командный код той машины, где установлена и функционирует исполнительная среда **CLR**. **Microsoft** реализовала **CLR** и **FCL** для различных версий **Windows**, включая Windows 98/Me/NT 4/2000, 32 и 64-разрядные версии **WindowsXP** , WindowsVista, Windows 7 и семейство **.Net Server**.
* **Framework .Net** развивается параллельно с развитием языков программирования, среды разработки программных проектов и операционных языков. Версии языков **C# 2.0** и **VB 2005** использовали версию **Framework .Net 2.0**. Операционная система **Windows Vista** включила в качестве надстройки **Framework .Net 3.0**. Языки **VB 2008**, **C# 3.0** и **Visual Studio 2008** включают версию **Framework .Net 3.5**.

**Framework .Net** является свободно распространяемой виртуальной машиной. Это существенно расширяет сферу его применения. Производители различных компиляторов и сред разработки программных продуктов предпочитают теперь также транслировать свой код в **IL**, создавая модули в соответствии со спецификациями **CLR**. Это обеспечивает возможность выполнения их кода на разных платформах.

Компилятор **JIT**, входящий в состав **CLR**, компилирует **IL** код с учетом особенностей текущей платформы. Благодаря этому создаются высокопроизводительные приложения. Следует отметить, что **CLR**, работая с **IL** кодом, выполняет достаточно эффективную оптимизацию и, что не менее важно, защиту кода. Зачастую нецелесообразно выполнять оптимизацию на уровне создания **IL** кода, она иногда может не улучшить, а ухудшить ситуацию, не давая **CLR** провести оптимизацию на нижнем уровне, где можно учесть особенности процессора.

***Метаданные.*** Переносимый исполняемый **PE**-файл является самодокументируемым файлом и, как уже говорилось, содержит программный код и метаданные, описывающие код. Файл начинается с манифеста и включает в себя описание всех классов, хранимых в **PE**-файле, их свойств, методов, всех аргументов этих методов – всю необходимую **CLR** информацию. Поэтому помимо **PE**-файла не требуется никаких дополнительных файлов, записей в реестр, вся нужная информация извлекается из самого файла. Среди классов библиотеки **FCL** имеется класс **Reflection**, методы которого позволяют извлекать необходимую информацию. Введение метаданных не только важная техническая часть **CLR**, но это также часть новой идеологии разработки программных продуктов. Мы увидим, что и на уровне языков **VB** и **C#** самодокументированию уделяется большое внимание.

***Сборщик мусора*** (Garbage Collector) и ***управление памятью***. Еще одной важной особенностью построения **CLR** является то, что исполнительная среда берет на себя часть функций, традиционно входящих в ведение разработчиков трансляторов, и облегчает тем самым их работу. Один из таких наиболее значимых компонентов **CLR** – сборщик мусора (Garbage Collector). Под сборкой мусора понимается освобождение памяти, занятой объектами, которые стали бесполезными и не используются в дальнейшей работе приложения. В ряде языков программирования (классическим примером является язык **C/C++**) память освобождает сам программист, в явной форме отдавая команды, как на создание, так и на удаление объекта. В этом есть своя логика – **я тебя породил, я тебя и убью**. Однако можно и нужно освободить человека от этой работы. Неизбежные ошибки программиста при работе с памятью тяжелы по последствиям, и их крайне тяжело обнаружить. Как правило, объект удаляется в одном модуле, а необходимость в нем обнаруживается в другом далеком модуле. Обоснование того, что программист не должен заниматься удалением объектов, а сборка мусора должна стать частью исполнительной среды, дано достаточно давно.

В **CLR** эта идея реализована в полной мере. Задача сборки мусора снята не только с программистов, но и с разработчиков трансляторов; она решается в нужное время и в нужном месте – исполнительной средой, ответственной за выполнение вычислений. Здесь же решаются и многие другие вопросы, связанные с использованием памяти, в частности, проверяется и не допускается использование "чужой" памяти, не допускаются и другие нарушения. Данные, удовлетворяющие требованиям **CLR** и допускающие сборку мусора, называются **управляемыми данными**.

Но, как быть с языком C++ и другими языками, где есть нетипизированные указатели, адресная арифметика, возможности удаления объектов программистом? Ответ следующий – **CLR** позволяет работать как с управляемыми, так и с **неуправляемыми данными**. Однако использование неуправляемых данных регламентируется и не поощряется. Так, в C# модуль, использующий неуправляемые данные (указатели, адресную арифметику), должен быть помечен как небезопасный (**unsafe**), и эти данные должны быть четко зафиксированы. Исполнительная среда, не ограничивая возможности языка и программистов, вводит определенную дисциплину в применении потенциально опасных средств языков программирования.

***Исключительные ситуации***. Что происходит, когда при вызове некоторой функции (процедуры) обнаруживается, что она не может нормальным образом выполнить свою работу? Возможны разные варианты обработки такой ситуации. Функция может возвращать код ошибки или специальное значение, может выбрасывать исключение, тип которого характеризует возникшую ошибку. В **CLR** принято во всех таких ситуациях выбрасывать исключение. Косвенно это влияет и на язык программирования. Выбрасывание исключений наилучшим образом согласуется с исполнительной средой. В языках **VB** и **C#** выбрасывание исключений, их дальнейший перехват и обработка – основной рекомендуемый способ обработки исключительных ситуаций.

***События***. У **CLR** есть свое видение того, что представляет собой тип. Есть формальное описание общей системы типов **CTS**. В соответствии с этим описанием, каждый тип, помимо полей, методов и свойств, может содержать и события. При возникновении событий в процессе работы с тем или иным объектом данного типа посылаются сообщения, которые могут получать другие объекты. Механизм обмена сообщениями основан на делегатах – функциональном типе. Надо ли говорить, что в язык **C#** встроен механизм событий, полностью согласованный с возможностями **CLR.**

Исполнительная среда **CLR** обладает мощными динамическими механизмами – сборки мусора, динамического связывания, обработки исключительных ситуаций и событий. Все эти механизмы и их реализация в **CLR** созданы на основании практики существующих языков программирования. Но уже созданная исполнительная среда в свою очередь влияет на языки, ориентированные на использование **CLR**. Поскольку языки **VB** и **C#** создавался одновременно с созданием **CLR,** то, естественно, они стали языками, наиболее согласованным с исполнительной средой, и средства этих языков напрямую отображаются в средства исполнительной среды.

***Общие спецификации и совместимые модули***. Уже говорилось, что каркас **Framework .Net** облегчает межъязыковое взаимодействие. Для того чтобы классы, разработанные на разных языках, мирно уживались в рамках одного приложения, для их бесшовной отладки и возможности построения разноязычных потомков, они должны удовлетворять некоторым ограничениям. Эти ограничения задаются набором общеязыковых спецификаций – **CLS**. Класс, удовлетворяющий спецификациям **CLS**, называется **CLS**-совместимым. Он доступен для использования в других языках, классы которых могут быть клиентами или наследниками совместимого класса.

#### **2.2.7. Среда исполнения программ**

##### **2.2.7.1. Введение**

Ранее мы попытались разобраться с тем, зачем нам нужна вычислительная техника, что такое алгоритм, программа и чем они отличаются друг от друга, какими инструментами мы в настоящий момент располагаем для того, чтобы эти самые программы создавать. Также мы должны были осознать, почему техника без программ представляет собой лишь мертвую «груду железа», а программы без своего воплощения – более или менее строгую абстракцию.

Вообще говоря, в контексте обсуждения методов программирования словосочетание «***вычислительная техника***» требует расшифровки. Вроде бы очевидно, что к вычислительной технике относятся компьютеры. Можно ли считать «вычислительной» стиральную машину с программным управлением? А сотовый телефон? Ведь их назначение вовсе не в том, чтобы складывать и умножать числа. Однако это точка зрения потребителя. А для разработчика программного обеспечения важно лишь то, способна ли та или иная техника выполнять программы, поскольку, если способна, то кто-то должен для нее эти самые программы создавать. К счастью, при всем многообразии видов и моделей современной техники написание программ для нее основано на тех же базовых принципах, которые используются при работе с классическим «вычислителем», более знакомым всем под именем «компьютер». Итак, с точки зрения программиста ***к вычислительной технике относится все, что имеет возможность выполнять программы.***

Что нужно для того, чтобы программа, которая есть выраженный на языке программирования алгоритм, могла быть выполнена? Вроде бы ответ очевиден – нужен тот, кто способен шаг за шагом (инструкцию за инструкцией) выполнять сформулированные в алгоритме действия. Поскольку действий много, нам потребуется место для их хранения и последующего считывания. Кроме того, любая программа оперирует данными (входными и результирующими) – их тоже необходимо хранить. Наконец, входные данные для программы обычно поставляет человек, он же «забирает» результаты, а, значит, требуются средства ввода/вывода (обмена информацией).

Здесь, стоит упомянуть о глобальном противоречии, которое до сих пор определяет развитие всей программной индустрии – удобные способы представления информации у человека и у компьютера различны. Человек свободно оперирует образами: это тигр, а это кот, хотя те же «усы, лапы и хвост»; вот эта конструкция о четырех ногах, вон та на колесиках и даже та, что с одной вычурно изукрашенной подставкой – все это стол. Для компьютера же информация, а еще точнее данные, есть всего лишь последовательность (короткая или длинная) нулей и единиц. В самом начале компьютерной эры мощности ЭВМ едва хватало на то, ради чего их создавали – помочь человеку в выполнении численных расчетов, к которым, так или иначе, сводится большинство реальных задач. Естественно ЭВМ освобождали от всех побочных дел, вроде перевода информации из вида, удобного человеку, в вид, понятный машине – на их долю оставались чистые вычисления. Однако, подобно тому, как на подрастающих детей родители начинают перекладывать обязанности по уходу сначала за собой, а потом и за семьей в целом, так и на долю компьютеров с ростом их мощности падало все больше и больше задач, не связанных напрямую с выполнением расчетов. И если когда-то программирование велось в машинном коде, потом на ассемблере, затем на языках высокого уровня, то сейчас компьютер пытаются научить “понимать” обычную человеческую речь. Вполне возможно, что в будущем основным занятием программиста будет не «стучать по клавишам» а с не меньшей скоростью «молоть языком».

Повышение уровня «дружественности» компьютера к человеку ведется, конечно же, не только в области средств разработки программ, а точнее даже не столько, сколько в области прикладного использования компьютера как еще одного инструмента в руках человека. И в числе прочих эффектов это повышение дружественности привело к появлению целого класса специальных обслуживающих программ, реализующих промежуточный слой между «голым железом» и «полезными» программами, теми, что выполняют конкретные задачи пользователя.

Таким образом, необходимо разобраться, из чего в настоящий момент складывается среда, в которой выполняется любая прикладная программа. С чем ей приходится взаимодействовать, и что, в конечном счете, должен иметь в виду программист, желающий, чтобы написанная им программа не просто выполняла то, что от нее требуется, но и делала это по возможности эффективно.

##### **2.2.7.2. Процессор**

Еще раз напомним: изначальное предназначение вычислительной техники, в точном соответствии с названием, *– выполнение расчетов*, следовательно, главный функциональный элемент любого вычислительного устройства должен***«считать»***. Вторая, не менее важная задача – ***управление*** (здесь вполне уместна аналогия с мозгом человека – он тоже одновременно является и центром принятия решений, и центром управления). Управлять приходится самим собой, устройствами хранения инструкций и данных, устройствами ввода/вывода и т.д.

***Процессор***, он же ***центральный процессор***, он же ***ЦП,*** он же ***CentralProcessingUnit***, он же ***CPU***– основной элемент любой вычислительной системы. В силу внутреннего устройства он понимает только двоичную логику.

Человечеству известны ***натуральные, целые***, ***рациональные, иррациональные***, ***вещественные*** (они же действительные), ***комплексные числа***. Из всего перечисленного в компьютере с его двоичной системой счисления достаточно просто могут быть представлены лишь натуральные числа. Чуть сложнее с целыми отрицательными. А с вещественными числами совсем плохо (про комплексные числа и вовсе умолчим: работа с ними – это уже высшая математика, ни в одном из известных процессоров работа с комплексными числами «в железе» не реализована). Из математического анализа известно, что количество вещественных чисел бесконечно не только «в длину» (вправо и влево по координатной оси), но и «в глубину», то есть на любом сколь угодно малом отрезке [a;b] вещественных чисел также бесконечно много. Таким образом, если целые числа «растут» только в одну сторону, слева от десятичной точки, то вещественные еще и вправо от нее. В результате для представления вещественных чисел используется специальный формат, получивший название «***число с плавающей точкой»***.

Из предыдущего нетрудно заключить, что в процессоре вычислительных блоков должно быть как минимум два: один для целых чисел и один для вещественных. На самом деле и тех и других (блоков) больше, чем по одному. Итак, «***Целые и вещественные числа обрабатываются в процессоре по-разному и даже в разных его частях***». И еще: «***Кроме целых и вещественных чисел процессор ничего другого обрабатывать не умеет***».

Теперь вторая задача процессора – ***управление***. Каждая программа содержит набор команд (инструкций), указывающих процессору, что он должен сделать, откуда взять исходные данные, куда поместить результат. Понятно, что каждая такая команда должна быть процессору известна, то есть содержаться в его системе команд. Общее их количество относительно невелико (в последних процессорах компании **Intel** – пара сотен). Условно все команды можно разделить на два типа: ***вычислительные и управляющие (системные)***. С вычислительными командами вроде все ясно. Представим, что имеется горячее желание заставить процессор сложить два числа. Находим в списке команд ту, которая отвечает за сложение (не забывая, что для целых чисел команда будет одна, а для вещественных совсем другая), и… Возникает законный вопрос: как же объяснить процессору, какие именно числа складывать? Следующий вопрос: куда их положить, чтобы процессор сумел их «достать»? Для этого в самом процессоре существует блок регистров – часть процессора, специально предназначенная для хранения данных. Регистров этих немного, да и размер их невелик. Итак, прежде чем мы сможем выполнить команду сложения двух чисел, их необходимо загрузить в регистры, вот для этого, в частности, нам и понадобятся системные инструкции. Естественно, есть множество других ситуаций, в которых процессор должен выполнять не вычисления, а функции управляющего, «говоря» остальным устройствам компьютера: «пойди туда», «сделай то», «подай это» и т.д.

Следующий важный момент – поступление данных в процессор. Взаимодействие с внешним миром процессор производит через шину данных – набор соединений, по которым данные в двоичной системе передаются в виде электрических сигналов. Чем больше соединений, тем большими порциями может передаваться информация. Размер «порции» называется разрядностью шины. Например, в процессоре **Intel® Pentium® 4** шина данных 64-разрядна, то есть за единицу времени этот процессор может воспринять (или передать) блок из шестидесяти четырех нулей и единиц.

Процессор – устройство с дискретным «восприятием» времени. Моменты времени для него следуют друг за другом в виде тактов, в каждый такой такт «помещается» одно элементарное действие. Чем мельче такт, тем быстрее «живет» процессор, а значит, тем выше его быстродействие. Такт современных процессоров составляет менее одной миллиардной секунды. Такие числа не слишком удобны для восприятия, поэтому быстродействие процессора принято характеризовать его ***тактовой частотой*** – ***числом тактов в секунду (герц)***. Таким образом, конкретная модель процессора может быть описана, например, так: процессор **Intel® Pentium® 4** с тактовой частотой 3,4 Ггц (гигагерц или GHz).

Напомним, что вычислительным системам, основанным на архитектуре, предложенной Джоном фон Нейманом в 40-х годах ХХ века присущи следующие характеристики:

1. ***Принцип программного управления***. Из него следует, что программа состоит из набора команд, которые выполняются процессором автоматически друг за другом в определенной последовательности. Последовательность выполнения команд определяется *алгоритмом* работы и связанной с ним программой на языке программирования.
2. ***Принцип однородности памяти***. Программы и данные хранятся в одной и той же памяти. Поэтому компьютер не различает, что хранится в данной ячейке памяти — число, текст или команда
3. ***Принцип адресности***. Структурно основная память состоит из перенумерованных ячеек; процессору в произвольный момент времени доступна любая ячейка. Отсюда следует возможность давать имена областям памяти, так, чтобы к запомненным в них значениям можно было впоследствии обращаться или менять их в процессе выполнения программ с использованием присвоенных имен.

##### **2.2.7.3. Оперативная память**

В каждый момент времени (такт) процессор работает ровно с одной командой и средствами хранения команд не обладает.

Большинство команд представляют собой операции над некоторой порцией данных. Как мы выяснили выше, небольшое количество данных процессор все-таки в состоянии разместить «внутри себя», в регистрах. Однако, кроме самых простейших случаев, регистров для хранения всех данных, которые должны быть обработаны в конкретной программе, недостаточно. Таким образом, и сами данные и команды для их обработки должны быть куда-то записаны. Оттуда процессор будет их извлекать, выполнять указанные действия и в это же «куда-то» сохранять полученные результаты. ***Оперативная память***, она же ***ОП***, она же ***Random Access Memory***, она же ***RAM***,– второе по важности действующее лицо вычислительной системы, при выключении питания «забывает» все, что содержала. В отличие от процессора представляет собой совокупность физических и программных элементов. Должностные обязанности: «хранитель оперативной информации».

Дать точное определение оперативной памяти весьма непросто. Если процессор – это конкретное физическое устройство, которое можно подержать в руках, то на вопрос, что такое **ОП**, разные специалисты ответят Вам совершенно по-разному. Сборщик компьютеров скажет, что оперативная память – это одна, две или три небольших по размеру платы, устанавливаемые в соответствующие слоты на материнской плате. Программист, создающий прикладные программы, сообщит Вам, что для него оперативная память есть место хранения данных, при этом порции данных имеют имена, а размещение данных и связывание с именами осуществляется системой программирования. Программист, разрабатывающий обслуживающие (системные) программы, с уверенностью скажет, что оперативная память – это адресное пространство, то есть непрерывная последовательность пронумерованных ячеек одного и того же размера, в которых размещается код программы и обрабатываемые данные.

Каждое из приведенных определений отражает некоторые аспекты понятия **ОП**.

##### **2.2.7.4. Долговременное хранение информации**

Как уже было сказано, оперативная память хранит информацию, только пока на нее подается питание. Очевидно, в компьютере должно присутствовать и устройство, способное «не забыть» информацию, если вдруг кто-то выдернет «шнур из розетки».

***Жесткий диск***, он же ***винчестер***, он же ***Hard-DiskDrive***, он же *HDD*, он же «винт»– самое ценное действующее лицо вычислительной системы.

В отличие от двух предыдущих элементов вычислительной системы устройства жесткого диска касаться не будем. С точки зрения программиста значение имеет лишь то, как осуществляется долговременное хранение информации. Ключевым понятием в этом процессе является ***файл***. Дать четкое определение этому понятию также не просто, как и оперативной памяти.

Итак, приближение первое: ***файл*** – ***это поименованная область на диске***. Любой файл имеет имя. Чаще всего существуют ограничения на длину имени и допустимые символы, из которых оно может быть составлено. Любой файл некоторым образом располагается на диске, имеет начало и длину в байтах.

Приближение второе: нередко ***файл*** записывается на диск частями и в разные моменты времени, как следствие физически он может состоять из отдельных фрагментов дискового пространства. Таким образом, более точно можно сказать, что ***файл – последовательность областей диска, логически связанных и имеющих общее имя.***

Подавляющее большинство долговременно хранящейся информации представляется в виде файлов, в том числе и сами программы.

На этом мы закончим наш краткий экскурс в мир компьютерного «железа» и перейдем к «софту» – программному обеспечению.

##### **2.2.7.5. Классификация программных средств**

Существенная часть потребностей современного человека связана с восприятием и обработкой информации во всех ее видах– ***от текстового до мультимедийного, и от формул до музыки.***

***Программное обеспечение***, оно же **ПО**, оно же **Computer Software**, оно же «**Софт**» – совокупность всех программных средств, имеющихся в данной вычислительной системе. Если «железо» вычислительной системы – это мозг, то программы, хранящиеся на диске, – умения и навыки, а программы и команды которых выполняются процессорами – мысли.

Вернемся к задаче о сложении двух чисел. Есть команда процессора, есть оперативная память, в которую можно разместить аргументы, есть клавиатура, на которой можно их набрать, есть жесткий диск, где можно сохранить результат. Как связать все это в единое целое? Программа для выполнения этой простейшей операции должна быть способна обработать электрические импульсы от клавиатуры, «перекодировать» их в числа, разместить эти числа в оперативной памяти, передать адреса в команду сложения, получить адрес результата, считать его, найти место на жестком диске, куда записать файл с результатом, осуществить запись. Это если вкратце. Какой процент от размера такой программы составит собственно команда сложения?

Очевидно, что взаимодействие с аппаратурой вычислительной техники, – задача, которую необходимо решать в любой программе, – должно быть запрограммировано отдельно, то есть между программами, выполняющими задачи пользователя системы, и аппаратурой должен располагаться промежуточный слой из специальных обслуживающих программ. Этот промежуточный слой называется ***системное программное обеспечение***. Все остальные относятся к ***прикладным программам***. Классы программ пересекаются, то есть одна и та же программа может быть в зависимости от точки зрения отнесена либо к системным, либо к прикладным.

В качестве примера: в системном программном обеспечении существуют так называемые драйверы – программы, осуществляющие эффективное управление функциональными блоками вычислительной системы, например, жестким диском. Изготовление драйверов обычно берет на себя фирма-производитель.

##### **2.2.7.6. Операционная система**

Задача обеспечения удобного взаимодействия человека и вычислительной системы уже довольно давно ставится во главу угла при разработке программного обеспечения. При этом первые реальные пользователи вычислительной системы – это программисты. А среди программистов первыми «вступают в контакт» с аппаратурой программисты системные – создатели программ системного уровня. Однако при всем нашем уважении к системным программистам прикладных программистов все же существенно больше. Прикладные программисты, пишущие программы на языках высокого уровня, обычно способны путем некоторых манипуляций с набором исходных текстов получить исполняемый модуль, готовый к запуску, и ожидают, что все дальнейшее возьмет на себя кто-то еще.

***Операционная система***, она же ***ОС***, она же ***Operating System***, она же ***OS***, она же «***операционка***», она же – ***основное управляющее лицо любой современной вычислительной системы***. Она представляет собой совокупность программных средств системного уровня. Должностные обязанности: обеспечение взаимодействия пользователя и вычислительной системы, обеспечение эффективного использования ресурсов последней, организация надежного функционирования программного обеспечения.

Устройство современных операционных систем более сложно, чем устройство самих вычислительных систем. Здесь отметим лишь два существенных момента. Пользователю вычислительной системы ОС предоставляет средства управления, программисту – набор функций системного уровня (так называемый ***API – Application Programming Interface***).

Подавляющее большинство персональных компьютеров в мире работают под управлением различных версий **ОС Microsoft Windows.** Не вдаваясь в детали, отметим основные моменты, которые необходимо иметь в виду программисту при работе с ***Windows***.

Прежде всего, ***Windows*** – операционная система с ***графическим интерфейсом пользователя.*** Это означает, что существенную часть кода любой программы будет составлять обеспечение соответствующего внешнего вида. Правда, для любителей, а также для ситуаций, в которых пользовательский интерфейс не имеет существенного значения, существует возможность создания ***приложений консольных***, функционирующих в текстовом режиме.

Во-вторых, ***Windows*** – система многозадачная, а значит, при написании программы Вы не можете рассчитывать на «единоличное» использование ресурсов вычислительной системы: процессорного времени, оперативной памяти, экрана монитора и т.д. Правда, существенную часть необходимой работы для обеспечения разделения ресурсов берет на себя сама операционная система.

В-третьих, ***Windows*** любой программе при запуске предоставляет «отдельное» линейное адресное пространство, размером в 4 Гб. Поскольку большинство компьютеров обладают меньшим объемом оперативной памяти, в реальности это пространство обеспечивается механизмами поддержки виртуальной памяти (с использованием жесткого диска).