## Введение в алгоритмизацию и программирования

### B.1. Основные понятия и определения

***Программное обеспечение* (ПО)** является логическим продолжением технических средств. Сфера применения конкретного компьютера определяется созданным для него ПО.

Сам по себе компьютер не обладает знаниями ни в одной области применения. Все эти знания сосредоточены в выполняемых на компьютерах программах. К ПО можно отнести также всю **область деятельности по проектированию и разработке ПО:** технологию проектирования (например, нисходящее проектирование, структурное и объектно-ориентированное проектирование); методы [тестирования](http://www.tomsk.ru/Books/informatica/theory/chapter8/1_8_7.html); методы доказательства правильности работы программ; документирование программ; разработку и использование программных средств, облегчающих процесс проектирования ПО, и многое другое.

Все программы, работающие на компьютере, можно условно разделить на три категории: [***прикладные программы***](http://www.tomsk.ru/Books/informatica/theory/chapter6/1_6_3.html), непосредственно обеспечивающие выполнение необходимых пользователям задач; [***системные программы***](http://www.tomsk.ru/Books/informatica/theory/chapter6/1_6_4.html), выполняющие различные вспомогательные функции; [***инструментальные программные системы***](http://www.tomsk.ru/Books/informatica/theory/chapter6/1_6_12.html), облегчающие процесс создания новых программ.

***Прикладная программа*** – это любая программа, способствующая решению какой-либо задачи в пределах данной предметной области. Прикладные программы могут носить и общий характер, например, обеспечивать составление и печатание документов и т.п. Причем прикладные программы могут использоваться либо автономно, то есть решать поставленную задачу без помощи других программ, либо в составе программных комплексов или пакетов программ.

***Системные программы*** выполняются вместе с прикладными и служат для управления ресурсами компьютера – центральным процессором, памятью, вводом-выводом. Это программы общего пользования, которые предназначены для всех пользователей компьютера. Системное программное обеспечение разрабатывается так, чтобы компьютер мог эффективно выполнять прикладные программы.

Среди большого количества системных программ особое место занимают ***операционные системы (ОС)***, а также программы вспомогательного назначения – *утилиты*. Утилиты либо расширяют и дополняют соответствующие возможности ОС, либо решают самостоятельные важные задачи. Причем часть утилит входит в состав операционной системы, а другая часть функционирует независимо от нее, т.е. автономно.

***Операционная система*** – это комплекс взаимосвязанных системных программ, назначение которых – организовать взаимодействие пользователя с компьютерным оборудованием и выполнение всех других программ. Операционная система – связующее звено между компьютером, с одной стороны, и выполняемыми программами, а также пользователем, с другой стороны.

Операционную систему можно назвать программным продолжением устройства управления компьютера.

В различных моделях компьютеров используют ОС с разной архитектурой и возможностями. Для их работы требуются разные ресурсы. Они предоставляют разную степень сервиса для программирования и работы с готовыми программами.

Кроме того, программное обеспечение условно можно разбить на два класса: «**малое**» и «**большое**».

***«Малое»*** программное обеспечение имеет следующие характеристики: решает одну несложную, четко поставленную задачу; размер исходного программного кода не превышает нескольких сотен строк; скорость работы программного обеспечения и необходимые ему ресурсы не играют большой роли; ущерб от неправильной работы не имеет большого значения; модернизация программного обеспечения, дополнение его возможностей требуется редко; как правило, разрабатывается одним разработчиком или небольшой группой; подробная документация не требуется, ее может заменить исходным кодом с комментариями.

Примером «малого» ПО могут служить программы базовых алгоритмов, изучаемые в данном пособии и примеры проектов из лабораторных работ.

***«Большое» программное обеспечение*** имеет две три или более характеристик из следующего перечня: решает совокупность взаимосвязанных задач; использование приносит значимую выгоду; удобство его использования играет важную роль; обязательно наличие полной и понятной документации; низкая скорость работы приводит к потерям; сбои, неправильная работа, наносит ощутимый ущерб; программы в составе ПО во время работы взаимодействует с другими программами и программно-аппаратными комплексами; работает на разных платформах; требуется развитие, исправление ошибок, добавление новых возможностей; группа разработчиков состоит из более чем 5 человек [ 9 ].

Необходимо отметить***,*** что ***сложное*** или ***«большое» ПО,*** которое называют также ***программными системами***, ***программными комплексами***, ***программными продуктами***, отличается от «небольшого» не столько по размерам, сколько по наличию дополнительных факторов, связанных с его востребованностью и готовностью пользователей «платить» как за приобретение самого продукта, так и за его сопровождение, и за с обучение работе с ним.

Проектирование ПО подразумевает выработку свойств системы на основе анализа постановки задачи, а именно: ***модели предметной области***, требований к ПО, а также опыта разработчика с учетом ***структуры программных систем.*** Кроме того, модель предметной области накладывает ограничения на логику предметной области и структур данных.

Что же определяет ***структуру и стиль программных систем***? Прежде всего, это ***парадигма программирования***, которой необходимо следовать, работая в любой среде системы программирования.

***Парадигма программирования*** представляет и определяет то, как программист видит процесс создания и выполнения программного проекта – способ мышления, как представляются задачи, подлежащие решению на компьютере с помощью конкретной среды и языка программирования, как все должно быть структурировано и организовано.

В данном учебном пособии при изучениибазовых средств алгоритмического языка высокого уровня Visual Basic используется среда программирования Visual Studio .NET. Поскольку в основе этой среды лежат различные технологии, в том числе технологии ***структурного, процедурного, визуального и объектно-ориентированного программирования,*** то ***в первую очередь*** необходимо следовать, правилам этих технологий. Все они – всего лишь различные инструменты, которые можно использовать при разработке ПО. Каждый из этих инструментов по-своему хорош. То есть, на самом деле, различные методики «программирования» дают разный выигрыш для решения задач разных классов.

Итак, ***парадигма программирования*** – это совокупность идей и понятий, определяющая стиль написания программ. Парадигма, в первую очередь, определяется базовой программной единицей и самим принципом достижения модульности программы.

Кроме того, в современном программировании очень часто парадигма программирования определяется набором инструментов программиста, а именно, [***языком программировани****я*](http://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F) и используемыми средствами разработки ПО.

Парадигма программирования определяет и то, в каких терминах описывается логика программы. Так, в [***императивном программировании***](http://ru.wikipedia.org/wiki/%D0%98%D0%BC%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) программа описывается как последовательность действий, а в [***объектно-ориентированном программировании***](http://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) (ООП) программу принято рассматривать как набор взаимодействующих объектов.

ООП есть, по сути, императивное программирование, дополненное принципом инкапсуляции данных и методов в объекте (принцип модульности) и наследованием (принципом повторного использования разработанного функционала). Сами же методы (процедуры) разрабатываются на принципах [***императивного программирования***](http://ru.wikipedia.org/wiki/%D0%98%D0%BC%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5).

В тоже время, важно отметить, что парадигма программирования не определяется однозначно языком программирования – многие современные языки программирования допускают использование различных парадигм.

***Процедурное программирование* –** это [парадигма программирования](http://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%B4%D0%B8%D0%B3%D0%BC%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F), основанная на концепции описания и вызова процедур. Процедуры также известны как [подпрограммы](http://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B4%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0), методы или функции.

***Процедурное программирование*** – это, прежде всего выделение кода в отдельные, желательно небольшие функциональные блоки (называемые процедурами) с целью упростить код (сделать его проще для понимания текст алгоритма), для повторного использования кода (чтобы избежать его многократного повторения в тексте программы), ну и само-собой для разделения кода на понятные специализированные блоки, намного более легкие в понимании и обслуживании, чем очень длинный монолитный код.

Процедуры просто содержат последовательность шагов для выполнения конкретной функциональной задачи или ее фрагмента. При этом в ходе выполнения программы любая процедура может быть вызвана из любой точки произвольное число раз.

***Структурное программирование*** – методология разработки [ПО](http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5), в основе которой лежит представление программы в виде иерархической структуры базовых ***алгоритмических*** [***блоков***](http://ru.wikipedia.org/wiki/%D0%91%D0%BB%D0%BE%D0%BA_%28%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%29). Она предложена в 70-х годах XX века [Э. Дейкстрой](http://ru.wikipedia.org/wiki/%D0%94%D0%B5%D0%B9%D0%BA%D1%81%D1%82%D1%80%D0%B0,_%D0%AD%D0%B4%D1%81%D0%B3%D0%B5%D1%80_%D0%92%D0%B0%D0%B9%D0%B1), разработана и дополнена [Н. Виртом](http://ru.wikipedia.org/wiki/%D0%92%D0%B8%D1%80%D1%82,_%D0%9D%D0%B8%D0%BA%D0%BB%D0%B0%D1%83%D1%81). В соответствии с этой методологией:

1. Любая программа представляет собой структуру, построенную из трёх типов базовых конструкций: *последовательность* – однократное выполнение операций в том порядке, в котором они записаны в тексте программы; *разветвление* – однократное выполнение одной из двух или более операций, в зависимости от выполнения некоторого заданного условия; *цикл* – многократное исполнение одной и той же операции до тех пор, пока выполняется некоторое заданное условие (условие продолжения цикла); в программе базовые конструкции могут быть вложены друг в друга произвольным образом, но никаких других средств управления последовательностью выполнения операций не предусматривается.
2. Повторяющиеся фрагменты программы (либо не повторяющиеся, но представляющие собой логически целостные вычислительные блоки) могут оформляться в виде т. н. [**процедур**](http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D0%B4%D1%83%D1%80%D0%B0) ([**подпрограмм**](http://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B4%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0) или [**функций**](http://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F_%28%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%29)). В этом случае в тексте основной программы, вместо помещённого в подпрограмму фрагмента, вставляется инструкция вызова подпрограммы. При выполнении такой инструкции выполняется вызванная подпрограмма, после чего исполнение программы продолжается с инструкции, следующей за командой вызова подпрограммы.
3. Разработка программы ведётся пошагово, методом «сверху вниз».

Суть метода разработки программы «сверху вниз» заключается в следующем. Сначала пишется текст основной программы, в котором, вместо каждого связного логического фрагмента текста, вставляется вызов подпрограммы, которая будет выполнять этот фрагмент. Вместо настоящих, работающих подпрограмм, в программу вставляются «заглушки», которые ничего не делают. Полученная программа проверяется и отлаживается. После того, как [разработчик](http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%81%D1%82) уверен, что процедуры вызываются в правильной последовательности (то есть общая структура программы верна), то процедуры-заглушки последовательно заменяются реально работающими. Причём разработка каждой процедуры ведётся тем же методом, что и основной программы. Разработка заканчивается тогда, когда не останется ни одной «затычки», которая не была бы удалена. Такая последовательность гарантирует, что на каждом этапе разработки программист одновременно имеет дело с обозримым и понятным ему множеством фрагментов, и может быть уверен, что общая структура всех более высоких уровней программы верна. При сопровождении и внесении изменений в программу выясняется, в какие именно процедуры нужно внести изменения, и они вносятся, не затрагивая части программы, непосредственно не связанные с ними. Это позволяет гарантировать, что при внесении изменений и исправлении ошибок не выйдет из строя какая-то часть программы, находящаяся в данный момент вне зоны внимания программиста.

***Технология*** ***нисходящего проектирования*** с ***пошаговой детализацией*** является неотъемлемой частью создания хорошо структурированных программ. При написании программы с использованием этой технологии вся задача рассматривается как единственное предложение (вершина), выражающее общее назначение программы. Так как вершина редко отображает достаточное количество деталей, на основании которых можно написать программу, то поэтому надо начинать процесс детализации – ***функциональной декомпозиции***. Вершина разделяется на ряд более мелких задач (функций) в том порядке, в котором эти задачи должны выполнятся. В результате получим первую детализацию. Далее каждая из подзадач разбивается на подзадачи, принадлежащие второму уровню детализации. Программист завершает процесс нисходящей разработки с пошаговой детализацией, когда алгоритм настолько детализирован, чтобы его можно было бы преобразовать в программу.

Важно, что при декомпозиции используются только указанные выше три управляющих конструкции, что позволяет говорить о ***структурной декомпозиции*** или ***структурном проектировании*** программ.

Таким образом, пошаговая реализация – это тактика разработки программы, а нисходящее проектирование – это стратегия программирования.

***Объектно-ориентированное программирование (ООП)* –** [парадигма программирования](http://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%B4%D0%B8%D0%B3%D0%BC%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F), в которой основными концепциями являются понятия[***объектов***](http://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82_%28%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%29)и[***классов***](http://ru.wikipedia.org/wiki/%D0%9A%D0%BB%D0%B0%D1%81%D1%81_%28%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%29)**.**

*Класс* – это тип, описывающий структуру объектов. Понятие «класс» подразумевает некоторое поведение и способ представления. Понятие «объект» подразумевает нечто, что обладает определённым поведением и способом представления. Говорят, что объект – это экземпляр класса. Класс можно сравнить с чертежом, согласно которому создаются объекты. Обычно классы разрабатывают таким образом, чтобы их объекты соответствовали объектам предметной области. Класс является описываемой на языке исходного кода моделью ещё не существующей сущности – объекта.

**ООП возникло в результате развития идеологии** [процедурного программирования](http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D0%B4%D1%83%D1%80%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)**, где данные и процедуры (подпрограммы и функции) их обработки формально не связаны. Кроме того, в современном объектно-ориентированном программировании часто большое значение имеют понятия события (так называемое** [событийно-ориентированное программирование](http://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D0%B1%D1%8B%D1%82%D0%B8%D0%B9%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)**).**

### В.2. Этапы решения задач

Почти все языки высокого уровня современных интегрированных систем разработки ПО объектно-ориентированные. Для изучения базовых средств VB мы будем использовать «простейшие» программы с использованием процедур. Данный подход используется для разработки **методов** классов.

 Тогда общая схема решение «простейших задач», **ориентированная на разработку процедур (методов)** включает в себя следующие основные этапы, часть из которых осуществляются до использования компьютера:

1. **Постановка задачи**. Этап включает в себя: сбор информации о задаче; определение конечных целей решения задачи; определение формы выдачи результатов; описание данных.
2. **Анализ и исследование задачи**. На этом этапе анализируются существующие аналогичные задачи; разрабатывается математическая модель задачи; осуществляется формализация; определяются структуры данных.
3. **Разработка алгоритма**. Этап заключается в выборе формы записи алгоритма и в последующем процессе разработки алгоритма и его описания.
4. **Программирование (Разработка программного кода).** На этом этапе вначале осуществляется выбор алгоритмического языка и уточнение способов организации данных, а затем разрабатывается программный код, описывающий разработанный алгоритм.
5. **Тестирование и отладка**. При тестировании и отладке выявляют синтаксические, семантические (смысловые) и логические ошибки, допущенные при разработке алгоритма и программировании. Анализ результатов тестирования позволяет устранить все выявленные семантические и логические ошибки.
6. **Анализ результатов решения задачи**. На этом этапе осуществляется прогон программы при реальных исходных данных. В результате анализа результатов расчета возможно уточнение математической модели и повторение этапов 2-5.

Одними из самых трудоемких и ответственных этапов, из вышеперечисленных, являются этапы ***алгоритмизации и программирования***.

***Процесс алгоритмизации*** заключается в опи­сании необходимой последовательности действий, с помощью которой можно однозначно реализовать выбранный метод решения задачи. На практике только очень простые задачи представляются в виде известной последовательности арифметических или логических действий. Для боль­шинства задач перед написанием программы требуется разработать соответствующую последовательность действий, приводящую к решению задачи, то есть алгоритм ее решения. Причем, при разработке алгоритма сложной задачи це­лесообразно провести декомпозицию вычислительного процесса, составить укрупненную схему алгоритма с це­лью выявления типовых участков алгоритма и использования для их реализации стандартных или ранее разработанных алгоритмов (**процедурное программирование**). Заметим, что время, потраченное на разработку вначале укрупненного, а затем детального алгоритма, полно­стью окупается при программировании и отладке программы.

***Алгоритм*** формально можно определить, как точное предписание (действие, группу действий), определяющее ***процесс преобразования исходных данных в результат*.** Из определения алгоритма вытекают и его основ­ные свойства:

* детерминированность – однозначность получения результата при одних и тех же исходных данных;
* результативность – обязательность получения искомого ре­зультата за конечное число шагов;
* массовость – возможность получения результата при различ­ных исходных данных рассматриваемого класса задач;
* дискретность – возможность разбиения алгоритма на отдель­ные элементарные действия, позволяющие рассматривать ал­горитм с различным уровнем детализации.

Существуют различные способы описания алгоритмов. На прак­тике наиболее распространены следующие формы представления алгоритмов:

* [словесная](http://www.tomsk.ru/Books/informatica/theory/chapter7/1_7_5.html)– **последовательность действий, описанная на естественном языке;**
* [графическая](http://www.tomsk.ru/Books/informatica/theory/chapter7/1_7_6.html)– **изображения в виде схем алгоритмов, содержащей функциональные общепринятые** *графические блоки***алгоритма;**
* [программная](http://www.tomsk.ru/Books/informatica/theory/chapter7/1_7_12.html) **– текст программы на языке программирования.**

Рассмотрим некоторые формы представления алгоритмов подробнее.

***Словесный способ записи алгоритмов***, представляет собой описание последовательных этапов обработки данных и имеет произвольную форму изложения на естественном языке.

Словесный способ не имеет широкого распространения, поскольку такие описания: строго не формализуются; страдают многословностью записей; допускают неоднозначность толкования отдельных предписаний.

***Графический способ представления алгоритмов*** является более компактным и наглядным по сравнению со словесным представлением.

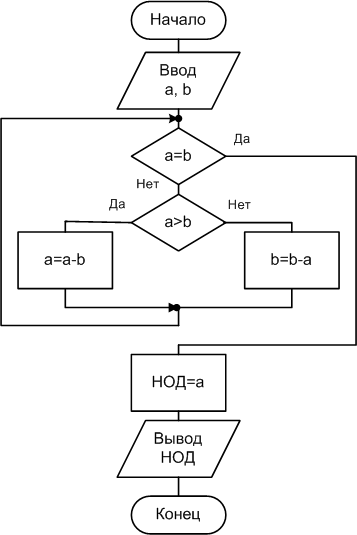
При графическом представлении алгоритм изображается в виде последовательности связанных между собой **функциональных блоков**, каждый из которых соответствует выполнению одного или нескольких действий. Такое графическое представление называется **схемой алгоритма**. В схеме алгоритма каждому типу действий (вводу исходных данных, вычислению значений выражений, проверке условий, управлению повторением действий, окончанию обработки и т.п.) соответствует геометрическая фигура, представленная в виде **блочного символа**. В соответствии с логикой решения конкретной задачи, блоки связы­вают друг с другом линиями, которые называются **линиями потока**. Использование этого способа описания алгоритма особенно полезно на начальном этапе изучения языка программирования. На рис. В.2-1 приведен перечень основных обозначений, принятых в схемах алгоритмов.

|  |  |  |
| --- | --- | --- |
| Наименование | Обозначение | Назначение |
| Пуск/останов |  | Начало или конец обработки данных. |
| Ввод/вывод |  | Ввод исходных данных или вывод результатов. |
| Процесс |  | Выполнение операции или группы операций, в результате которых изменяется значение, форма представления или расположения данных. |
| Решение |  | Выбор направления выполнения алгоритма в зависимости от значения логического условия |
| Модификация |  | Начало регулярной циклической структуры. |
| Предопределенный процесс |  | Использование ранее созданных и отдельно описанных алгоритмов. |
| Линии потока |  | Указание последовательности связей между блоками. |
| Соединитель |  | Указатель связей между прерван­ными линиями потока. |
| Комментарий |  | Пояснения элементов схемы. |
| Межстраничный соединитель |  | Указатель связей между страницами, если схема алгоритма занимает несколько листов. |

*Рис. В.2-1. Основные обозначения, принятые в схемах алгоритмов*

**Пример В.2-1.** Рассмотрим запись алгоритма нахождения наибольшего общего делителя (НОД) двух натуральных чисел (алгоритм Эвклида) с помощью схемы алгоритма.

Из выше перечисленных способов описания алгоритмов самым распространенным и наиболее наглядным является графический способ, поэтому в дальнейшем все алгоритмы будут изображаться графически, в виде схем алгоритмов, реализованные средствами графической программы Visio [10] и см. Приложение 2 настоящего пособия.



*Рис В.2-2. Пример графического описания алгоритмов*

### В.3. Базовые алгоритмические структуры

***Базовыми алгоритмическими структурами*** принято называть подмножество алгоритмических структур, которые позволяют составить алгоритм решения сколь угодно сложной задачи (**структурное программирование)**. Эти структуры могут быть вы­браны разработчиком программы в зависимости от специфики решаемой задачи. К простейшим базовым алгоритмическим структурам относятся: последовательные структуры; разветв­ляющиеся структуры; циклические структуры регулярного типа; циклические структуры итеративного типа.

В свою очередь из базовых алгоритмических струк­тур могут быть составлены алгоритмы решения простейших типовых за­дач, часто встречающихся в качестве составляющих при решении мно­гих инженерных задач. Такие типовые алгоритмы будем в дальнейшем называть ***базовыми алгоритмами***.

***Последовательными*** называются такие алгоритмические струк­туры, в которых функциональные элементарные блоки вы­полняются в том порядке, в котором они записаны или изо­бражены на схеме алгоритма. Такая структура может быть составлена из совокупности блоков «**Пуск/останов**», «**Ввод/вывод**», а также блоков «**Процесс**», используемых для проведения вычислений (рис.В.3-1).

|  |  |
| --- | --- |
| Рис-3-1-3 | Рис-3-1-4 |

*Рис. В.3-1. Алгоритмическая Рис.В.3-2. Алоритмическая*

*структура структура*

*– прследовательность – стандартное разветвление*

***Разветвляющимися*** называются такие алгоритмические струк­туры, в которых порядок выполнения функциональных блоков определяется значе­ниями логических выражений, использующихся для организации разветвления. Разветвляющийся алгоритм может состоять из нескольких ветвей, каждая из которых может содержать любую, сколь угодно сложную, алгоритми­ческую структуру.

|  |  |
| --- | --- |
| Рис-3-1-5 | Рис-3-1-6 |

*Рис. В.3-3. Алгоритмическая Рис.В.3-4. Алгоритмическая*

*структура – усеченное разветвление структура – усеченное разветвление*

В процессе работы алгоритма в первую очередь вычисляются ***логиче­ские выражения***. Если логическое выражение принимает значение «Истина», то выполняется часть алгоритма, расположенная по ветви «Да», если принимает значение «**Л**ожь», то – часть алгоритма по ветви «Нет».

Анализ разветвляющихся алгоритмов, применяемых в практиче­ских задачах, показывает, что наиболее часто используются три типа разветвлений:

***Стандартное разветвление*** содержит функциональные блоки как в ветви «Да», так и в ветви «Нет» (рис.В.3-2).

**Усеченное разветвление** содержит функциональные блоки только в ветви «**Да**» или только в ветви «**Нет**» (рис.В.3-3, В.3-4).

**Вложенное разветвление** содержит одно или несколько дополнительных разветвлений (рис.В.3-5).

|  |  |
| --- | --- |
| Рис-3-1-7 | Рис-3-1-8 |

*Рис. В.3-5. Алгоритмическая Рис.В.3-6. Алгоритмическая*

*структура – вложенное разветвление структура – регулярный цикл*

В соответствии с основным принципом **структурного программиро­вания**, все разветвляющиеся структуры, как и все другие алгоритмические структуры, должны иметь **один вход** и **один выход**.

**Циклическими** называются структуры, в которых предусмотрена возможность многократного повторения выполнения фрагмента алгоритма. Этот участок называется **телом цикла**. Различают циклические структуры двух видов: с заранее известным и с неизвестным числом повторений цикла.

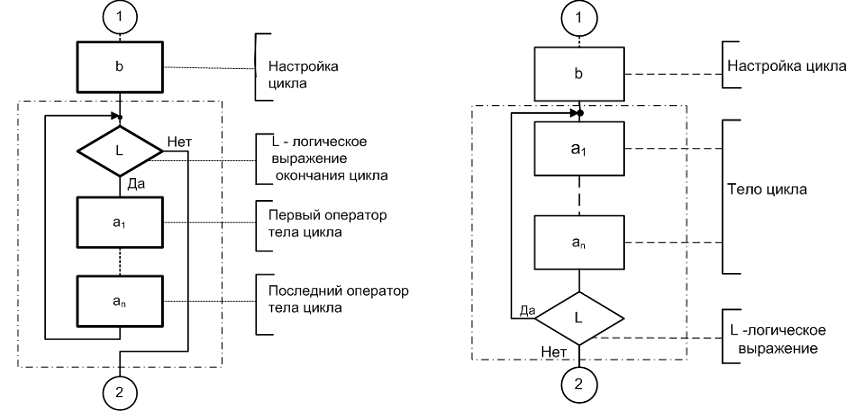
Циклические структуры, в которых число повторений цикла заранее известно или может быть определено до начала цикла, называются **регулярными циклическими структурами** (рис.В.3-6).

В блоке организации цикла используется специальная пере­менная, которая предназначена для определения условия останова цикла (i). Эта переменная называется **параметром цикла**. Блоки, следующие за заголовком цикла, составляют **тело цикла**. Тело цикла выполняется для всех значений параметра цикла i, начинаю­щегося со значения m1 и изменяющегося с шагом m3 до значения m2.

 Если из условия задачи следует, что число повторений цикла за­ранее не определено, а вычисляется в процессе выполнения алгоритма, то условие выхода из цикла должно быть определено в процессе его вы­полнения. При этом важно, чтобы в условие выхода из цикла входила перемен­ная, значение которой изменялось бы в теле цикла, иначе выполнение цикла будет бесконечным.

 Циклическая структура, в которой число повторений цикла заранее неизвестно, а определяется только в процессе выполнения алгоритма, называется **итеративной циклической структурой**.

В зависимости от места расположения условия продолжения цикла (или выхода из цикла) **итеративные циклические структуры** подразделяются на два вида: **с предусловием** и **с постусловием**.



*Рис. В.3-7. Алгоритмическая Рис.В.3-8. Алгоритмическая*

*структура – итеративный цикл структура – итеративный цикл*

При организации цикла с предусловием (рис.В.3-7) блоки тела цикла, следующие за блоком, в котором проверяется условие выхода из цикла, выполняются всякий раз, когда условие L принимает значение «Истина». При первом невыполнении этого условия происходит выход из цикла. Таким образом, возможен случай, когда тело цикла не будет выполнено ни разу. Поэтому циклические структуры с известным числом повторений (регулярные циклы) относятся к числу циклических алгоритмов с предусловием.

При организации циклов с постусловием, для которых условие выхода из цикла (или повторения тела цикла) проверяется после выполнения цикла   
(рис.В.3-8), цикл всегда выпол­няется хотя бы один раз, независимо от значения L, и только после его выполнения принимается решение – продолжать выполнение цикла или выйти из него.

### В.4. Основные средства разработки программ, и платформа Framework

Разработка программного обеспечения в настоящее время превратилась в обычный технологический процесс, на разных стадиях которого действуют подготовленные специалисты, применяющие в своей повседневной производственной практике различные технологии.

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.

***Программирование*** – процесс ***создания компьютерных программ с помощью языков программирования.*** Программирование сочетает в себе элементы искусства, науки, математики и инженерии.

В узком смысле, программирование рассматривается как кодирование – реализация одного или нескольких взаимосвязанных алгоритмов на некотором языке программирования.

В широком смысле*, программирование – процесс создания программ, то есть разработка программного обеспечения*. Причем одна из основных частей работы разработчика программ связана с написанием исходного кода проекта на одном из языков программирования.

Рассмотрим средства разработки программного обеспечения, которые помогают программистам в решении их профессиональных задач.

Прежде всего, это система программирования.

***Система программирования*** – это программная система для разработки ПО средствами конкретного языка программирования. Современные системы программирования обычно предоставляют пользователям мощные и удобные средства разработки программ**.** В них входят: средства создания («написания») и редактирования текстов программ (программного кода); трансляторы; [библиотеки стандартных](http://www.tomsk.ru/Books/informatica/theory/chapter6/1_6_18.html) процедур; [отладочные средства](http://www.tomsk.ru/Books/informatica/theory/chapter8/1_8_6.html), помогающие находить и устранять ошибки в программе.

«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 словесное описание алгоритма, поэтому необходимо превратить его в абсолютно точный набор инструкций, которые однозначно будут интерпретировать ПК. Для этого были созданы языки программирования. Причем процесс «написания» программ за последние 50 лет прошел путь от программирования в инструкциях процессора (машинных командах), через программирование на низкоуровневых языках (языках ассемблера) до программирования на языках высокого уровня.

Уровень в данном случае – это уровень машинного восприятия. Так, языки низкого уровня (ассемблер) по возможности приближены к ПК, что делает соответствующие программы особенно эффективными с точки зрения их быстродействия. Однако существенная проблема использования таких языков заключается в том, что разработчик программы – прежде всего человек, и его способы восприятия информации весьма далеки от машинных, что чрезвычайно затрудняет написание программ на ассемблере. Подавляющее большинство программ пишется на том или ином языке программирования высокого уровня.

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

С языками программирования связанны следующие понятия:

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

Действительно, язык программирования – это искусственный (формальный) язык, предназначенный ***для записи алгоритмов***. Язык программирования задается своим описанием и реализуется в виде специальных программ: ***компилятора*** или ***интерпретатора***.

Под ***транслятором*** обычно понимают специальную программу, которая переводит код программы в последовательность машинных команд. Напомним еще раз: код программы понятен человеку, набор команд понятен компьютеру (процессору). Заметим, что трансляторы языков программирования высокого уровня, таких как **Pascal, C**, VB и других, обычно называют ***компиляторами***. Этим подчёркивается общепринятый для подобных языков режим трансляции, при котором вначале осуществляется перевод программы в двоичное представление, а лишь затем программа передаётся на исполнение. Другой способ трансляции, называемый ***интерпретацией***, 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.

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 процедуры 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, которая заложена во всех современных системах программирования. В данный момент для нас важно обратить внимание на тот факт, что для обеспечения комплектации оттранслированной программы вспомогательными подпрограммами требуются специальные средства.

Для этого в систему программирования входит программа, называемая ***редактор связей***, и, которая обеспечивает поиск вспомогательных подпрограмм в библиотеках 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 входят 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 программный код обычно создается большим коллективом разработчиков, в связи с чем, возникает много дополнительных проблем.

Современный разработчик программ, как правило, вводит код программы в компьютер, пользуясь ***редакторами программного кода или текстовыми процессорами.***

***Редактор кода*** – это программная подсистема, обеспечивающая первоначальную подготовку исходного текста программы и его исправление в процессе разработки. В отличие от универсальных ***текстовых процессоров***, редакторы кода специализированы для работы именно с исходными текстами программ, поэтому они не имеют массы функций обычных редакторов, зато предоставляют другие специальные функции, не менее полезные. Существует довольно большое количество различных редакторов программного 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 языка программирования)***.

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

В настоящее время при разработке программ существует возможность частично избавить себя от «рутины» с помощью, так называемых ***средств автоматизированной генерации кода***, в некоторых случаях эти средства умеют самостоятельно создавать программный код, выполняющий определенные стандартные действия. Примером таких средств может служить MS Visual Studio .NET, которая автоматически создает и заполняет полями класс «**Форма**» при создании новой формы и наполнении его различными компонентами. 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. Таким образом, создание документации – задача не менее важная, чем создание программного кода, и в ее решении нам помогают различные программные средства.

Мы рассмотрели компоненты, входящие в систему программирования. Это и ***редакторы кода***, и ***трансляторы***, и ***сборщик***и, и ***отладчики***. 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.

Для устранения неудобств и повышения эффективности процесса разработки разработчики 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.

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

Кроме того, интегрированные среды разработки ПО позволяют выполнять и другие операции: визуально (в диалоге) производить быструю настройку параметров каждого из компонентов системы программирования; сохранять разные системы настроек и загружать их по мере необходимости; осуществлять запуск одного или сразу нескольких компонентов системы программирования нажатием нескольких клавиш или выбором соответствующих пунктов меню и многое другое.

Единственный минус таких сред является прямым следствием их главного плюса – собрав «под одной крышей» большой набор инструментов, интегрированная среда сама становится весьма сложной программой системой. Однако время, потраченное на ее изучение, окупается в дальнейшем. И, наконец, еще один положительный момент – устройство большинства сред одинаково в концептуальном плане.

***Платформа .NET*** – это совокупность всех средств разработки, распространения и выполнения приложений .NET, включая операционные системы, серверы, сервисы, стандарты, спецификации и документацию.

В состав платформы входят следующие программные средства. В первую очередь это средства разработки приложений – Visual Studio .NET. Созданные приложения выполняются при помощи надстройкой над операционной средой .NET Framework – и это второй элемент платформы. Специально для нужд .NET разработано семейство серверных операционных систем Windows. И наконец, расширяемый набор служб .NET объединяет службы со стандартными функциями, используемыми приложениями.

***Архитектура .NET*** – это устройство платформы .NET со всеми ее составными частями и схемами взаимодействия между ними. Говоря об архитектуре, имеют в виду общие принципы и правила создания и использования приложений .NET, представленные в виде описаний, спецификаций и стандартов.

***Операционная среда .NET Framework*** – это важнейшая составная часть платформы .NET, обеспечивающая разработку, распространение и выполнение приложений .NET. Ее роль настолько велика, что довольно часто происходит подмена понятий и под заголовком «Архитектура .NET» находишь хорошее, подробное описание .NET Framework. Поэтому всегда нужно помнить, что .NET Framework – это составная часть платформы .NET. Именно она представляет наибольший интерес для разработчиков. Хочешь писать приложения для .NET, – знай устройство .NET Framework.

***Приложения .NET***– тип приложений, которые могут выполняться только под правлением операционной среды, т. к. они откомпилированы не в двоичный код операционной системы, а в коды промежуточного языка MSIL. Такие приложения не могут напрямую вызвать, к примеру, функцию Win API, а всегда обращаются к операционной среде как промежуточному слою, изолирующему приложение от деталей реализации операционной системы.

***Службы******.NET***– Web-службы, разработанные и функционирующие под управлением операционной среды .NET Framework. Хотя некоторые источники определяют .NET как архитектуру и платформу для создания нового поколения именно Web-служб, это только часть (безусловно, важная и перспективная) ее предназначения.

*Visual Studio .NET*– это высокопроизводительный, интегрированный, расширяемый набор средств поддержки полного жизненного цикла, в том числе для командной разработки, приложений. Во все последние версии в состав VS входят компиляторы трех языков – VB, VC#, VC++, F# и Java. И все же именно первые два фактически олицетворяют собой эту системы. 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.

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

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

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

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 VB **– 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.

***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 другим.

***Двухэтапная компиляция. Управляемый модуль и управляемый код***. Компиляторы языков программирования, включенные в VS, 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. В зависимости от выбранного типа проекта, PE-файл может иметь разные уточнения - **exe, dll, mod** или **mdl**.

Исполнительную среду следует рассматривать как виртуальную IL-машину. Эта машина транслирует «на лету» требуемые для исполнения участки кода в команды реального процессора, который в и выполняет код.

***Виртуальная машина***. Отделение каркаса от студии явилось естественным шагом. Каркас Framework .Netперестал 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.

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.

***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 VB 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.

В настоящее время компания Microsoft выпустила версию Visual Studio 2013, которая покрывают потребности всех категорий разработчиков – от начинающих программистов до профессионалов высшего класса.

На сайте компании Microsoft (<http://www.visualstudio.com>) можно получить любую информацию по интегрированной среде Visual Studio.

Мы рекомендуем остановить свой выбор на версии **Visual Studio Professional 2013**. Для того, чтобы установить на свой компьютер VS, необходимо, прежде всего, провести инсталляцию этой системы.

***Инсталляция*** – процесс установки программного продукта на компьютер с целью его дальнейшего использования. Стабильность будущей работы приобретаемого оборудования и программного обеспечения основывается на его правильной инсталляции и корректной настройке.

Для получения бесплатных версий VS, можно воспользоваться акцией Microsoft – **DreamSpark** (https://www.dreamspark.com/Default.aspx), которая создана для поддержки технического образования путем предоставления доступа к программному обеспечению компании учебных, преподавательских и исследовательских целей.

### В.5. Общие рекомендации по разработке учебных программ

Рассмотрим некоторые проблемы, возникающие при разработке простейших учебных программ.

Во-первых, предполагается, что при разработке программ используется технологии ***императивного процедурного программирования c использованием технологии структурного программирования.*** Здесь важно понимать,что после того, как программа написана, мы должны убедиться в ее правильности. Следует заметить, что наличие ошибок в только что разработанной программе – это вполне нормальное и закономерное явление. Практически невозможно составить реальную, достаточно сложную программу без ошибок. Кроме того, нельзя делать вывод, что программа правильна, лишь на том основании, что она выполняется, и выдает результаты, поскольку эти результаты еще не обязательно правильные. В программе может оставаться большое количество логических ошибок. Ответственные участки программы рекомендуется проверять отдельно при помощи тестов, ориентированных на проверку конкретного участка программы.

Проконтролировать программу можно еще до ввода в компьютер, то есть за столом, с помощью просмотра, проверки и прокрутки.

**Просмотр** текста программы предусматривает **обнаружение описок и расхождений с алгоритмом**. Просматривается **организация всех циклов** с тем, чтобы убедиться в правильности операторов, задающих кратности циклов. Полезно посмотреть еще раз **условия в условных операторах, аргументы в обращениях к подпрограммам** и т.п.

При **проверке программы** программист по тексту программы мысленно воспроизводит тот вычислительный процесс, который определяет программа, после чего сверяет его с требуемым процессом. На время проверки нужно «забыть», что должна делать программа, и «узнавать» об этом по ходу её проверки. Только после окончания проверки программы можно «вспомнить» о том, что она должна делать и сравнить реальные действия программы с требуемыми.

Основой **прокрутки** является **имитация выполнения программы.**  Для выполнения прокрутки используют простейшие исходные данные и над ними производят все необходимые вычисления, следуя тексту программы. **Прокрутка –** это трудоемкий процесс, поэтому ее следует применять лишь для контроля логически сложных участков программ.

Следующим этапом контроля правильности программы является отладка и **тестирование** на компьютере.

**Отладка программы –** это процесс поиска и устранения ошибок в программе, производимый по результатам её прогона на компьютере, а **тестирование** –это испытание, проверка правильности работы программы в целом, либо её составных частей. Отладка и тестирование – это два четко различимых и непохожих друг на друга этапа, поскольку при **отладке** происходит локализация и устранение синтаксических ошибок и явных ошибок кодирования, а в процессе тестирования проверяется работоспособность программы, не содержащей явных ошибок. Таким образом, тестирование устанавливает факт наличия ошибок, а отладка выясняет ее причину.

В современных программных системах отладка осуществляется часто с использованием специальных программных средств, называемых отладчиками. **Программа-отладчик** обычно обеспечивает следующие возможности:

* **пошаговое исполнение программы** с остановкой на каждой команде (операторе);
* **просмотр текущего значения любой переменной или нахождение значения любого выражения**, в том числе, с использованием стандартных функций; при необходимости можно установить новое значение переменной;
* **установку в программе «контрольных точек»**, т.е. точек, в которых программа временно прекращает свое выполнение, так что можно оценить промежуточные результаты, и др.

Но даже если не используются средства программы-отладчика, при отладке программ важно помнить, что:

* в начале процесса отладки следует использовать **простые тестовые данные**;
* возникающие затруднения необходимо четко **разделять** и **устранять строго поочередно**;
* **не нужно считать причиной ошибок компьютер**, поскольку современные компьютеры и трансляторы обладают чрезвычайно высокой надежностью.

Как бы ни была тщательно отлажена программа, решающим этапом, устанавливающим ее пригодность для работы, является контроль программы по результатам ее выполнения на системе тестов.

Под **тестом** понимается некоторая совокупность исходных данных для программы и точное описание результатов, которые должна выработать программа при этих данных, в том виде, котором эти результаты должны быть выданы программой. Если для выбранной системы тестовых исходных данных программа дает правильные результаты, то программу условно можно считать правильной, поскольку тестирование может показать лишь наличие ошибок, но не их отсутствие. Нередки случаи, когда новые входные данные вызывают «ошибку» или получение неверных результатов работы программы**,** которая уже считалась полностью отлаженной. Для реализации метода тестов должны быть изготовлены или заранее известны эталонные результаты.

Тестовые данные должны обеспечить проверку всех возможных условий возникновения ошибок. При проведении тестирования руководствуются следующим:

* первый тест должен быть **максимально прост**, чтобы проверить, работает ли программа вообще;
* должна быть испытана **каждая ветвь** алгоритма;
* очередной тестовый прогон должен контролировать нечто такое, что еще **не было проверено** на предыдущих прогонах;
* арифметические операции в тестах должны **предельно упрощаться** для уменьшения объема вычислений;
* в тестовых примерах количество элементов последовательностей, точность для итерационных вычислений, количество проходов цикла должны задаваться из соображений **сокращения объема вычислений;**
* тестирование должно быть **целенаправленным и систематизиро- ванным**, поскольку случайный выбор исходных данных приводит к трудностям в проверке ожидаемых результатов, а также при случайном выборе тестовых данных многие ситуации могут оказаться непроверенными;
* усложнение тестовых данных должно происходить постепенно.

**Пример В.4-2.** **Разработать систему тестов для задачи нахождения корней квадратного уравнения **

При нахождении корней возможны следующие случаи:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Номер**  **теста** | **Проверяемый**  **случай** | **Коэффициенты** | | | **Результаты** |
| **a** | **b** | **с** |
| **1** | **d >0** | **1** | **1** | **-2** | **x1 = 1, x2 =  - 2.** |
| **2** | **d=0** | **1** | **2** | **1** | Корни равны: **x1 = - 1, x2 =  - 1.** |
| **3** | **d < 0** | **2** | **1** | **2** | Действительных корней нет |
| **4** | **a=0, b=0, c=0** | **0** | **0** | **0** | Все коэффициенты равны нулю.  **х** — любое число. |
| **5** | **a=0, b=0, c<>0** | **0** | **0** | **2** | Неправильное уравнение. |
| **6** | **a=0,   b<>0** | **0** | **2** | **1** | Линейное уравнение.  Один корень: **x=- 0,5.** |
| **7** | **a<>0,b<>0,с=0** | **2** | **1** | **0** | **x1= 0, x2= - 0,5.** |

Процесс тестирования можно разделить на три этапа:

1. Проверка **в нормальных условиях.** Предполагает тестирование на основе данных, которые характерны для реальных условий функционирования программы.
2. Проверка **в экстремальных условиях.** Тестовые данные включают **граничные значения** области изменения входных переменных, которые должны восприниматься программой как правильные данные. Типичными примерами таких значений являются очень маленькие или очень большие числа и отсутствие данных. Еще один тип экстремальных условий **–** это **граничные объемы** данных, когда массивы состоят из слишком малого или слишком большого числа элементов.
3. **Проверка в исключительных ситуациях.** Проводится с использованием данных, значения которых лежат **за пределами допустимой области изменений.** Известно, что все программы разрабатываются в расчете на обработку ограниченного набора данных.

При тестировании важно получить ответ на следующие вопросы:

* что произойдет, если программе, не рассчитанной на обработку отрицательных и нулевых значений переменных, в результате какой-либо ошибки придется иметь дело как раз с такими данными?
* как будет вести себя программа, работающая с массивами, если количество их элементов повысит величину, указанную в объявлении массива?
* что произойдет, если числа будут слишком малыми или слишком большими?

Наихудшая ситуация складывается тогда, когда программа воспринимает неверные данные как правильные и выдает неверный, но правдоподобный результат. Программа должна сама отвергать любые данные, которые она не в состоянии обрабатывать правильно. При этом ошибки могут быть допущены на всех этапах решения задачи **–** от ее постановки до оформления.

Полный перечень ошибок, как правило, приводится в литературе, посвященной полному описанию языка программирования.

### В.6. Общие рекомендации и требования к выполнению лабораторных работ

1. Изучение каждой темы следует начинать с теоретического материала, изложенного в основной части учебного пособия*.*
2. Приступая к выполнению лабораторной работе каждой темы, необходимо обратить внимание на список вопросов, подлежащих изучению, если какие-либо из вопросов остались неясными, следует вернуться к изучению теоретической части.
3. Общее задание представляет собой перечень всех пунктов, которые необходимо выполнить в данной работе, а конкретный вариант индивидуального задания выбирается из указанных в таблице вариантов заданий.
4. Программный код проекта студент должен написать и отладить самостоятельно. Причем для проверки того, что программа работает правильно, необходимо провести контрольное тестирование на примерах, для которых известны результаты. Таким образом, если при подстановке в программу известных исходных данных получен известный результат, то считается, что программа прошла тестирование.
5. Все пункты выполнения задания фиксируются в отчете. Поэтому перед оформлением отчета рекомендуется изучить раздел «Содержание отчета». Отчет должен быть оформлен аккуратно, а схемы алгоритмов и программ должны быть выполнены с использованием компьютерной программы Microsoft Visio.
6. При подготовке к защите лабораторной работы желательно решить несколько задач по текущей теме.
7. На занятии студент представляет преподавателю отчет по текущей лабораторной работе и после этого получает допуск к работе на компьютере. Преподаватель обязан проверить отчет и, указав на ошибки, потребовать их исправления. Допуск к работе преподаватель фиксирует в журнале.
8. Лабораторная работа считается сданной, если студент представил преподавателю отчет, продемонстрировал выполнение проекта на компьютере, а также защитил ее.
9. Защита лабораторных работ предполагает выполнение студентом дополнительного задания или ответа на ряд поставленных вопросов по данной теме. Результаты защиты лабораторной работы преподаватель также фиксирует в журнале.

### В.7. Общие рекомендации и требования к выполнению контрольных заданий

Контрольные задания предназначены для осуществления контроля знаний студентов по каждой из тем учебного пособия и используются при принятии зачета по каждой лабораторной работе. В зависимости от этапа контроля и изучаемой темы преподаватель имеет право задать вид контроля или выбрать несколько его видов.

Предполагается, что в примерах пособия и при проведении лабораторного практикума используется среда программирования Microsoft Visual Studio.

Если о типе исходных или результирующих числовых данных в задании ничего не сказано, то предполагается использование вещественных типов данных. При обработке наборов вещественных чисел следует предполагать, что все элементы набора являются различными (таким образом, любой набор вещественных чисел содержит единственный минимальный и единственный максимальный элемент). В наборах целых чисел могут присутствовать одинаковые элементы, в частности, наборы целых чисел, могут содержать несколько минимальных и максимальных элементов. Аналогичные предположения спра­ведливы для числовых массивов, а также для файлов, содержащих числовые данные.

Максимальный размер исходных файлов не указывается, поэтому при ре­шении заданий на файлы не следует использовать вспомогательные массивы, содержащие все элементы исходных файлов, однако допускается использова­ние вспомогательных файлов. Все исходные файлы считаются существующи­ми, за исключением специально оговоренных случаев, в которых существова­ние исходных файлов требуется проверять в ходе выполнения задания.

При выполнении учебных заданий ввод-вывод данных должен осуществляться с помощью перегруженных процедур-подпрограмм или процедур-функций.