## Введение в разработку

## прикладных программ

B.1. Основные подходы к созданию прикладных программ

В.2. Объектно-ориентированная декомпозиция и моделирование

В.3. Основные средства разработки программ на платформе .NET

В.4. Общие рекомендации по разработке учебных программ   
 средствами ООП

В.5. Контрольные вопросы

В.6. Краткие итоги

### B.1. Основные подходы к созданию

### прикладных программ

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

К ПО можно отнести 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 программного кода, документирование программ, разработку и использование программных средств, облегчающих процесс проектирования ПО, и многое другое.

Поэтому все ПО, работающее на компьютере, можно условно разделить на три категории: [прикладное ПО](http://www.tomsk.ru/Books/informatica/theory/chapter6/1_6_3.html), непосредственно обеспечивающее выполнение необходимых пользователям задач; [системное ПО](http://www.tomsk.ru/Books/informatica/theory/chapter6/1_6_4.html), выполняющее различные вспомогательные функции; [инструментальные программные системы](http://www.tomsk.ru/Books/informatica/theory/chapter6/1_6_12.html), облегчающие процесс создания новых программ.

***Системное ПО*** служит 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 огромного 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, – 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 аппаратурой компьютера, с одной стороны, и выполняемыми программами, а также пользователем, с другой стороны. Операционную систему можно назвать программным продолжением устройств управления компьютером.

***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.

В данном пособии будем говорить о разработке прикладных программ – приложений. В связи с этим необходимо отметить, что разрабатываемое прикладное ПО условно можно разбить на два класса: «**малое**» и «**большое**» [3,4].

***«Малое» прикладное ПО,*** как правило***,*** решает одну несложную, четко поставленную задачу; размер исходного программного кода не превышает нескольких сотен строк; скорость работы этого 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. Примерами ***«малых»*** ***прикладных программ*** могут служить программы, которые реализованы по базовым алгоритмам, и примеры проектов лабораторных работ [1,2].

***«Большое» ПО*** имеет две-три или более следующих характеристик: решает совокупность взаимосвязанных задач; его использование приносит значимую выгоду, а удобство использования играет важную роль; обязательно наличие полной и понятной документации; низкая скорость работы приводит к потерям; сбои, неправильная работа наносят ощутимый ущерб; программы в составе ПО во время работы взаимодействуют 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.

Необходимо отметить***,*** что ***«сложные»*** или ***«большие» прикладные программы,*** которые часто называют также ***программными системами***, ***программными комплексами или*** ***программными продуктами***, отличаются от «небольших» не столько по размерам, сколько по наличию дополнительных факторов, связанных с их востребованностью и готовностью пользователей «платить» как за приобретение самого продукта, так и за его сопровождение, и за обучение работе с ним.

Что же определяет ***структуру и стиль разработки ПО***? 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 учебном пособии [1,2] для изучениябазовых средств алгоритмического языка Visual Basic используется среда программирования Visual Studio .NET. А 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. На самом деле, различные методики программирования дают разный выигрыш для решения задач разных типов. Обычно этот выигрыш измеряют по двум параметрам: эффективностью ПО на современных ЭВМ; общими затратами на разработку и дальнейшее сопровождение ПО.

Различные языки программирования поддерживают различные стили программирования (парадигмы программирования) и требуют от программиста различного уровня внимания к деталям при реализации поставленных задач.

Так, в традиционном [**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) логика программы 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) программу принято рассматривать как набор взаимодействующих объектов предметной области.

В то же время важно отметить, что парадигма программирования не определяется однозначно языком программирования – многие современные языки программирования допускают использование различных парадигм. О просто последовательном, неструктурированном программировании даже не будем говорить, а рассмотрим две современные парадигмы: структурного (процедурно-ориентированного) и объектно-ориентированного программирования

Структурное (процедурное) 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 истинности или ложности 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 [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. То есть, процедурное программирование – это, прежде всего выделение программного кода в отдельные, желательно небольшие функциональные блоки, называемые процедурами, с целью упростить код (сделать проще для понимания текст алгоритма), для повторного использования кода (чтобы избежать его многократного повторения в тексте программы), для разделения кода на понятные специализированные блоки, намного более легкие в понимании и сопровождении, чем очень длинный монолитный программный код.
3. Разработка программы ведётся пошагово, как правило, методом «сверху вниз».

*Обратите внимание, что в* *окончательном виде любая программа представляет собой последовательность инструкций процессора*.

Исторически парадигмы процедурного, а затем объектно-ориентированного программирования появились следующим образом.

Работа первых компьютеров сводилась к вычислениям по формулам различной сложности, число переменных и массивов было невелико, и программист легко помнил взаимосвязи между ними и детали алгоритма. С ростом объема и сложности программы становится невозможно удерживать в памяти все детали, требуется структурировать информацию, выделять главное и отбрасывать несущественное. Этот процесс называется повышением *абстракции* программы. И первым шагом к повышению абстракции является *использование процедур,* позволяющее после написания и отладки процедуры отвлечься от деталей ее реализации, поскольку для вызова процедуры требуется только знать список ее формальных параметров. Так в программировании появилась и использовалась структурированная процедурно-ориентированная модель. Сначала программист разбирался, что должна делать программа, а затем выбирал одно из двух:

1. Задача разбивалась на подзадачи, те в свою очередь, на подзадачи следующего уровня и т.д. Этот процесс, называемый декомпозицией, длился до тех пор, пока упрощение подзадач не сводилось к элементарным процедурам, которые могли быть формализованы. Этот подход называ­ется «сверху вниз».
2. Программист писал процедуры для решения простых задач и последовательно объединял их в более сложные процедуры, пока не добивался решения поставленной задачи. Такой подход называется «снизу вверх».

Конечно, многие опытные программисты не следовали рекомендациям теоретиков, выступавших за первый способ, и предпочитали решать практические задачи комбинацией этих двух стратегий.

Таким образом, программа, написанная на основе процедурного подхода, разделена на процедуры, каждая из которых в идеальном случае выполняет некоторую законченную последовательность действий и имеет явно выраженные связи с другими процедурами программы. При этом задачи «реального мира» приходится переформулировать, представляя все данные в виде переменных, массивов, списков и других структур данных. При моделировании больших систем объем этих данных увеличивается, они становятся плохо управляемыми, и это приводит к большому числу ошибок. Другая большая проблема возникает из-за того, что любая процедура может обратиться к глобальным (общедоступным) данным, что повышает риск недопустимого (несанкционированного) изменения каких-то значений.

Итак, процедурно-ориентированный подход имеет два главных недостатка: первый заключается в неограниченности доступа процедур к глобальным данным; второй – в том, что разделение данных и процедур, являющееся основой процедурно-ориентированного программирования плохо отображает картину ***реального мира***. Рассмотрим подробнее эти недостатки.

В процедурной программе существуют два типа данных. ***Локальные данные*** находятся внутри процедуры и могут использоваться исключительно этой процедурой, и недоступны никому, кроме этой процедуры. При необходимости совместного использования одних и тех же данных несколькими процедурами, такие данные надо объявлять, как *глобальные*. Концепция локальных и глобальных данных показана на рис. В.1-1

Большие программы обычно содержат множество процедур и глобальных переменных. Между процедурами и глобальными данными возможно большое число связей, как показано на рис.В.1-2, что также порождает несколько проблем.

|  |
| --- |
| Процедура А  Процедура В |

*Рис. В.1-1. Глобальные и локальные переменные*

|  |
| --- |
| Процедура  Процедура  Процедура  Процедура |

*Рис. В.1-2. Процедурный подход   
к проектированию прикладных программ*

Во-первых, усложняется структура программы. Во-вторых, в программу становится трудно вносить изменения. Например, если разработчик программы складского учета решит сделать код продукта не 5-значным, а 12-значным, то будет необходимо изменить соответствующий тип данных с **Short** на **Long**. Это означает, что во все процедуры, оперирующие кодом продукта, должны быть внесены изменения. А когда изменения вносятся в глобальные данные больших программ, то непросто быстро определить, какие процедуры необходимо скорректировать, и кроме того, из-за большого числа связей между процедурами и данными исправленные процедуры начинают некорректно работать с другими глобальными данными. Таким образом, любое изменение влечет за собой далеко идущие последствия.

Вторая, более важная проблема процедурного подхода заключается в том, что отделение данных от процедур оказывается малопригодным для отображения картины реального мира. В реальном мире нам приходится иметь дело с физическими объектами, такими, например, как люди, машины или процессы. Эти объекты нельзя отнести ни к данным, ни к процедурам, поскольку реальные «вещи» представляют собой совокупность **атрибутов** и **поведения**. Примерами атрибутов для людей могут являться цвет глаз или место работы, для машин – мощность двигателя и количество дверей. То есть ***атрибуты объектов******равносильны данным*** в программах: они имеют определенное значение, например, «зеленый» для цвета глаз или 4 для количества дверей автомобиля.

***Поведение*** – ***это некоторая реакция объекта в ответ на какие-либо внешние или внутренние воздействия***. Например, мама на просьбу ребенка купить игрушку может ответить «да» или «нет». Если нажать на тормоз автомобиля, это повлечет за собой его остановку. Ответ и остановка являются примерами поведения. ***Поведение сходно с процедурой***: вызывается процедура, чтобы совершить какое-то действие (например, вывести на экран массив целых чисел), и процедура совершает это действие.

Таким образом, ни отдельно взятые данные, ни отдельно взятые процедуры не способны ***адекватно отражать (моделировать)*** объекты реального мира.

|  |
| --- |
|  |

*Рис. В.1-3. Объектно-ориентированный подход*

Основополагающей идеей ***объектно-ориентированного подхода*** является ***объединение данных*****и *их поведения****, т.е.* ***действий*, *производимых над этими данными*** ***в единое целое***, которое называется ***объектом***. При этом процедуры объекта, реализующие любые действия объекта, предназначены для доступа к данным объекта (атрибутам). В ООП доступ к данным (атрибутам) объектов может быть скрыт от внешнего воздействия, что позволяет защитить их от случайного изменения, то есть данные и процедуры могут быть ***инкапсулированы***. ***Объектно-ориентированный подход*** предлагает все, что входит в состав программного приложения, считать объектами, которые взаимодействуют друг с другом и с пользователем, посылая друг другу сообщения и откликаясь на них (рис. В.1-3).

Так что же такое объект?

***Объект*** – это некая отдельная сущность – элемент рассматриваемой предметной области, которая при программировании представлена своими ***данными, поведением и взаимодействием с другими объектами прикладной программы. Данные и поведение*** являются признаками программных объектов, отличающими их друг от друга. Понятно, что в реальном мире каждый предмет или процесс обладает набором как статических, так и динамических характеристик, т.е. атрибутами и поведением. Поведение объекта зависит от его состояния и внешних воздействий. Например, объект «автомобиль» никуда не поедет, если в баке нет бензина, а если повернуть руль, изменится положение колес. Объекты взаимодействуют, реагируют на поведение других объектов. Например, действие «повернуть руль» можно интерпретировать как посылка сообщения «автомобиль, поверни руль!».

Таким образом, каждый объект характеризуется следующими основными понятиями:

* ***Атрибутами*** – данными, которые представляют некоторое отдельное качество (параметр) объекта. Например, атрибутами могут быть размеры объекта, заголовок, его наименование и т.д. Совокупность атрибутов объекта с их значениями определяют состояние объекта. Как правило, атрибуты – это набор данных, в которых хранятся значения, определяющие параметры объекта;
* ***Состоянием*** – каждый объект всегда находится в определенном состоянии, которое характеризуется набором данных (значениями атрибутов) объекта. Под воздействием событий объект переходит в другие состояния. При этом объект может сам генерировать события при переходе в другое состояние;
* ***Поведением*** – это действия (операции), которые объект может выполнить;
* ***Событиями*** – это средство взаимодействия объектов друг с другом. Объекты генерируют заданные события и выполняют действия в ответ на заданные события. События – это аналог сообщений, которые получают и отправляют объекты.

Все объекты можно отличить один от другого, например, два апельсина, имеющие одинаковый цвет, форму и вкус – это все равно два апельсина (а не один), в чем легко убедиться, съев один из них (другой останется).

Между объектами можно установить отношение тождества: объекты, удовлетворяющие этому отношению, одинаковы (тождественны) в некотором смысле (например, страной-производителем), как вышеупомянутые апельсины. В этом случае иногда говорят о двух экземплярах объекта апельсин. Для нас объект и экземпляр некоторой обобщенной сущности – это одно и то же.

Необходимо обратить внимание, что наряду с физическими могут существовать также и абстрактные объекты, типичными представителями которых являются, например, комплексные числа и другие пользовательские типы данных или участники компьютерных игр. Таким образом, **объект – это любая физическая или абстрактная четко идентифицируемая сущность предметной области**.

***Объектно-ориентированное программирование (ООП)*** – это технология, которая фиксирует поведение реального мира таким способом, при котором некоторые детали его реализации скрыты. Применение такой технологии позволяет представить структуру программы в виде ***объектной модели*** – ***множества взаимодействующих друг с другом объектов.***

В объектно-ориентированном программировании, как и в реальном мире, пользователи программных приложений изолированы от деталей, необходимых для выполнения задач. Например, для печати страницы в текстовом редакторе пользователь вызывает определенную функцию нажатием кнопки на панели инструментов, но не видит происходящих при этом внутренних процессов. При печати страницы во время работы программы происходит сложное взаимодействие объектов, ко­торые, в свою очередь, взаимодействуют с принтером.

При создании объектно-ориентированного приложения предметная область представляется в виде совокупности объектов, которые объединены в **обобщенные сущности - классы**. При описании этого обобщения (сущности) тождественных объектов, принадлежащих предметной области с помощью программного класса, необходимо выделить в реальных объектах существенные особенности (характеристики) и проигнорировать многие другие, ограничиваясь лишь теми, которые нужны для решения практической задачи. Такой подход называется ***абстракцией****.*

***Абстракция*** – это выделение существенных характеристик объектов, отличающих их от других объектов. Причем список существенных атрибутов (свойств) для разных задач может быть совершенно различным. Например, объект «утка» с точки зрения биолога, изучающего миграции, ветеринара, или повара будет иметь совершенно разные характеристики.

При создании программных систем обычно рассматривают объекта предметной области двух видов:

* абстрактные объекты;
* реальные объекты.

***Абстрактный объект*** – это описание реального объекта предметной области без подробно­стей. Например, абстрактный человек – это описание человека, которое содержит атрибуты и ва­рианты поведения. Вот четыре атрибута, которые могут быть найдены у абстрактного человека: *имя, фамилия, рост, вес*. Обратите внимание, что эти атрибуты только **идентифицируют** тип атрибута, например, имя или вес, но не определяют самоимя или значение веса. Абстрактный объект используется как модель для реального объекта конкретной предметной области.

***Реальный объект*** имеет все атрибуты и варианты поведения, определенные в абстрактном объекте, а также те подробности, которые были упущены в абстрактном объекте. Например, абстрактный человек – это модель реального че­ловека. Абстрактный человек определяет, что реальный человек **должен** иметь имя, фамилию, рост и вес. Реальный человек определяет **значения**, связанные с этими атрибутами, например, такие: Иван, Петров, 180 см, 80 кг.

**В ООП создается** абстрактный объект, а затем его используют для создания ***реального объекта***. Реальный объект называется ***экземпляром*** абстрактного объекта. Можно ска­зать, что реальный человек – это **экземпляр** абстрактного человека.

Абстрактные объекты можно трактовать, например, как **формочки (шаблоны)** для печенья. Формочка для печенья выглядит как печенье, но печеньем не является.

В то же время объекты позволяют уде­лять внимание важным для нас подробностям и игнорировать те подробности, которые нам неинтересны. Например, преподаватель – это человек, и **он имеет множество атрибутов и вари­антов поведения,** которые вам интересны. Тем не менее, можно игнорировать множест­во атрибутов и вариантов поведения преподавателя и сфокусироваться только на тех, которые имеют отношение к вашему образованию. Например, для студента важно знать ФИО, возможно, ученую степень преподавателя и читаемый им курс.

Аналогично, преподаватель фокусируется на ваших атрибутах и вариантах поведения, которые показывают, например, как хорошо вы изучили учебный материал. Другие атрибуты, такие как ваша ра­бота на других занятиях, или ваш рост и вес, никак не учитываются преподавателем.

Таким образом, и студент, и преподаватель упрощают отношения, решая, какие атрибуты и варианты поведе­ния являются важными для конкретных целей, и используя в своих отношениях только их.

***Класс объектов –*** это множество объектов, имеющих общие характеристики, структуру и поведение. Таким образом, ***класс*** можно определить, как описание совокупности сходных между собой конкретных объектов, как описание того, какими они должны быть и что они должны делать. Говорят, что ***объект*** является ***экземпляром класса***. Если объекты реально существуют в предметной области, то класс – это абстракция, объединяющая объекты в одну группу согласно их особенностям, атрибутам, характеристикам и поведению. Например, кнопка **Button1** на форме со всеми своими конкретными атрибутами-свойствами и действием является объектом класса **Button**.

В основе **технологии объектно-ориентированного программирования** лежат «три фундамента»: ***инкапсуляция, наследование*** и ***полиморфизм***.

***Инкапсуляция*** – способность объединять внутри одной структуры (класса или объекта) состояние и поведение, и скрытие внутреннего устройства объекта и деталей реализации (от слова «капсула»). Важная характеристика любого объекта - его обособленность. Объект обособлен, если детали реализации объекта (внутренние структуры данных и алгоритмы их обработки) скрыты от пользователя объекта и недоступны для непреднамеренных изменений. При этом элементы объекта доступны только через интерфейс доступа к классу – совокупность правил доступа.

С инкапсуляцией мы постоянно сталкиваемся в жизни. Например, для того чтобы переключить телевизионную программу, нам достаточно на пульте дистанционного управления набрать ее номер, что запустит сложный механизм, который в итоге и приведет к желаемому результату. Нам совершенно необязательно знать, что происходит в пульте дистанционного управления и телевизоре, нам лишь достаточно знать, что телевизор обладает такой возможностью (функцией) и как ее можно активировать. Инкапсуляция, или сокрытие реализации, является основополагающим принципом ООП.

Другим примером инкапсуляции может быть отдел кадров предприятия. В нем собирается (инкапсулируется) вся информация о служащих. Здесь хранятся их личные дела, обработ­ку которых могут выполнять только сотрудники этого отдела. Еще одним примером инкапсуляции может служить политика безопасности в вычислительной сети. Каждый запрос на информацию о системе безопасности или на изменение политики безопасности должен выполняться через сетево­го администратора. От пользователей сети информация такого рода скрыта.

Инкапсуляция позволяет обеспечить высокий уровень безопасности данных в информацион­ной системе. Разработчик знает, как получить доступ к данным и какие операции над ними можно производить. При этом упрощаются процессы сопровождения и отладки программ, можно легко изменять процедуры обработки данных.

Она позволяет создавать пользовательские объекты, обладающие требуемыми функциями, и далее оперировать ими, не вдаваясь, в то, как устроены эти объекты. Таким образом, инкапсуляция – механизм, который объединяет данные и методы обработки этих данных и защищает их от внешнего вмешательства или неправильного использования. Инкапсуляция программного кода внутри класса обеспечивает невозможность «испортить» этот программный код при любом изменении деталей реализации отдельных классов. Поэтому можно использовать объект в другом окружении, и быть уверенным, что он не испортит не принадлежащие ему области памяти. Если же все-таки надо что-то изменить или дополнить в классе, то используются механизмы *наследования* и *полиморфизма*.

***Наследование*** – возможность создания ***иерархии классов***, когда потомки включают в себя все свойства и поведение классов-предков, а также могут их изменять и добавлять к ним свое собственное поведение и свойства. Прежде всего, это связано с тем, что каждый год в мире пишется множество программ и важно использовать уже написанный код. Преимущество объектно-ориентированного программирования состоит в том, что по мере необходимости для объекта можно определить наследников, корректирующих или дополняющих его поведение. При этом нет необходимости не только повторять исходный код родительского объекта, но даже иметь к нему доступ. Это упрощает модификацию программного кода и создание новых программ на основе существующей. Только благодаря наследованию можно использовать объекты, исходный программный код которых недоступен, но в которые требуется внести изменения. Таким образом, при наследовании можно не просто добавлять новые методы, но и изменить существующие. И во многом это обеспечивается благодаря полиморфизму*.*

***Полиморфизм*** («много форм») – возможность использовать в различных классах иерархии одно имя для обозначения сходных по смыслу действий, т.е. возможность во время выполнения программного кода с помощью одного и того же имени выполнять разные действия или обращаться к объектам разного типа. Полиморфизм реализуется через переопределение функций в классах-наследниках (процедура имеет **одно** имя и **одинаковые** параметры, но работает по-разному) – это механизм **виртуальных методов через динамическое *связывание***. Также полиморфизм реализуется как «перегрузка» методов (метод имеет **одно** имя и **разные** параметры) - это, например, использование знака «+» для обозначения сложения в классе целых чисел и классе строк: одна и та же по виду операция «+» дает совершенно разные результаты.

Методы структурного программирования помогают упростить процесс разработки сложных систем за счет использования алгоритмов как готовых строительных блоков. Аналогично, методы объектно-ориентированного проектирования созданы, чтобы помочь разработчикам применять мощные выразительные средства объектно-ориентированного программирования, использующего в качестве блоков модули, реализованные как модули пользователей и модули классов объектов (подробно модули будут описаны в **Теме 1**).

***Модуль*** – функционально законченный фрагмент [программы](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0), оформленный в виде отдельного [файла](https://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB) с [программным кодом](https://ru.wikipedia.org/wiki/%D0%98%D1%81%D1%85%D0%BE%D0%B4%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D0%B4) или поименованной непрерывной её части, предназначенный для использования в других программах. Модули позволяют разбивать сложные задачи на более мелкие в соответствии с [***принципом модульности***](https://ru.wikipedia.org/wiki/%D0%9C%D0%BE%D0%B4%D1%83%D0%BB%D1%8C%D0%BD%D0%BE%D1%81%D1%82%D1%8C_(%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))***.*** Обычно прикладные программы проектируются таким образом, чтобы предоставлять [программистам](https://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) удобную для многократного использования функциональность ([интерфейс](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D1%8B%D0%B9_%D0%B8%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81)) в виде набора [функций](https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F_(%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)) и [классов](https://ru.wikipedia.org/wiki/%D0%9A%D0%BB%D0%B0%D1%81%D1%81_(%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)). Удобство использования модульной архитектуры заключается в возможности обновления (замены) модуля, без необходимости изменения остальной системы.

***Модульное программирование*** – это организация программы как совокупности небольших независимых блоков, называемых модулями, структура и поведение которых подчиняются определенным правилам. Использование модульного программирования позволяет упростить тестирование программы и обнаружение ошибок.

Несмотря на то, что ООП действительно помогает в проектировании сложных и больших программ, ООП не панацея. ООП является другим пониманием того, что называется вычислениями. ООП – это не просто набор некоторых свойств, добавленных в уже существующие языки. Это новый шаг в осмыслении задач и разработки ПО. ООП – это взгляд на программирование, сосредоточенный на данных, в котором данные и поведе­ние жестко связаны. Программные системы в этих случаях – это совокупность взаимодействующих объектов. При этом каждый объ­ект отвечает за конкретную задачу, а функционирование все программной системы осуществляется посредством взаимодействия объектов.

Таким образом, на первом этапе разработки объектно-ориентированной прикладной программы необходимо определить множество образующих ее модулей и классов объектов, т.е. осуществить декомпозицию.

### В.2. Объектно-ориентированная декомпозиция и моделирование

В предыдущем параграфе мы рассмотрели основные подходы к созданию ПО. В этом параграфе рассмотрим подходы к построению объектных моделей программных систем.

Способ проектирования сложных систем был известен еще в древности – «разделяй и властвуй». При проектировании сложной программной системы необходимо разделять ее на все меньшие и меньшие подсистемы, каждую из которых можно совершенствовать независимо. В этом случае для понимания любого уровня системы нам необходимо одновременно держать в уме информацию лишь о немногих ее частях и этот способ называется ***декомпозицией***.

***Алгоритмическая декомпозиция***. Большинство специалистов, которые при создании программных систем используют технологию структурного процедурного программирования, и, в частности, проектирование «сверху вниз» воспринимают декомпозицию как обычное разделение алгоритмов, где каждый модуль системы выполняет одну из подзадач общего процесса.

***Объектно-ориентированная декомпозиция***, в отличие от алгоритмической, предписывает разделение модели предметной области на элементы, выбирая в качестве критерия декомпозиции принадлежность этих элементов к различным абстракциям данной предметной области. Прежде, чем разделять задачу на шаги, необходимо определить объекты предметной области.

Хотя оба варианта декомпозиции решают одну и ту же задачу, но они делают это разными способами. Во втором случае предметная область представлена совокупностью автономных элементов предметной области, которые взаимодействуют друг с другом, чтобы обеспечить поведение системы, соответствующее более высокому уровню. Каждый элемент обладает своим собственным поведением, и каждый из них моделирует некоторый объект реального мира. С этой точки зрения объект является вполне осязаемой вещью, которая демонстрирует вполне определенное поведение конкретной части предметной области. Объекты что-то делают, и им можно послать сообщение, попросив их выполнить то-то и то-то. Так как данная декомпозиция основана на объектах, а не на алгоритмах, она называется объектно-ориентированной декомпозицией.

  Для сложной программной системы важны оба способа декомпозиции: и по алгоритмам, и по объектам. Разделение по алгоритмам концентрирует внимание на порядке ***происходящих событий и вычислительном процессе***, а разделение по объектам придает особое значение элементам, которые являются ***объектами***. Однако нельзя сконструировать сложную систему одновременно двумя способами. Разделение системы возможно либо по алгоритмам, либо по объектам, а затем, используя полученную структуру, надо попытаться рассмотреть проблему с другой точки зрения.

Опыт показывает, что для сложных программных систем ***полезнее начинать с объектной декомпозиции***, которая имеет несколько важных преимуществ перед алгоритмической декомпозицией. Объектная декомпозиция помогает уменьшить размер программных систем за счет повторного использования общих механизмов. Объектно-ориентированные системы более гибки и проще эволюционируют со временем. Действительно, объектная декомпозиция существенно снижает риск при создании сложной программной системы, так как она развивается из меньших систем, в которых мы уже уверены. Более того, объектная декомпозиция помогает на ранних стадиях проектирования разобраться в сложной программной системе. А уже после создания объектной модели и описания структуры каждого класса для разработки операций класса можно использовать ***алгоритмическую декомпозицию для проектирования процедур.***

В результате ***объектно-ориентированной декомпозиции*** разработчик создает ***объектно-ориентированные модели***, т.е. такие модели, которые фокусируют внимание на объектах заданной предметной области и решаемой задачи.

Моделирование широко распространено во всех инженерных дисциплинах, в значительной степени из-за того, что оно реализует принципы декомпозиции, абстракции, и иерархии. Каждая модель описывает определенную часть рассматриваемой системы, далее строятся новые модели на базе старых моделей и т.д. Модели позволяют нам контролировать наши неудачи. Мы оцениваем поведение каждой модели в обычных и необычных ситуациях, а затем проводим соответствующие доработки, если нас что-то не удовлетворяет.

Чтобы понять во всех тонкостях поведение сложной системы, приходится использовать не одну модель. Например, проектируя компьютер на одной плате, инженер-электронщик должен рассматривать систему как на уровне отдельных элементов схемы (микросхем), так и на уровне схемы. Схема помогает инженеру разобраться в совместном поведении микросхем. Схема представляет собой план физической реализации системы микросхем, в которой учтены размер платы, потребляемая мощность и типы имеющихся интегральных микросхем. С этой точки зрения инженер может независимо оценивать такие параметры системы, как температурное распределение и технологичность изготовления. Проектировщик платы может также рассматривать динамические и статические особенности системы.

Определение программной системы в терминах объектов – это наиболее понятный способ разработки реального программного обеспечения. Объекты заставляют вас воспринимать все с той точки зрения, **что объ­ект делает, т.е. мысленно моделировать его поведение**. Благодаря этому вы можете отвлечь­ся от рассмотрения объекта с точки зрения того, как он будет реализован в процессе исполнения программы. Таким образом, в процессе написания программы можно использовать естественные термины реального мира. Вместо того чтобы строить программу в форме отдельных процедур и данных, т.е. в терминах мира компьютеров, программа строится из объектов. Причем объекты позволяют в программе моделировать реальный мир с помощью элементов, взятых из ***предметной области***.

***Предметная область*** – это **абстрактное пространство**, в котором формулируется определенная задача, т.е. набор понятий, представляющих важные аспекты решаемой задачи

Рассуждая в терминах решаемой задачи, можно избежать опасности увязнуть в деталях реализа­ции. Конечно, некоторые высокоуровневые объекты должны взаимодействовать с компьютером, пользуясь низкоуровневыми, машинно-ориентированными методами. Однако объект изолирует это взаимодействие от остальной части системы.

Объектно-ориентированная программа, как и реальный мир, состоит из объектов. В объектно-ориентированных языках программирования, например, в **VB** и **С#** все является объектами, начиная от самых пер­вичных, базовых типов, целых, логических и до наиболее сложных экземпляров типов пользователя.

Итак, разработчики программного обеспечения нашли механизм борьбы со сложностью – это аб­страгирование, т.е. способность отделить логический смысл фрагмента программы от проблемы его реализации. В результате объектно-ориентированной декомпозиции разработчик создает объектно-ориентированные модели, т.е. такие модели, которые фокусируют внимание на объектах заданной предметной области и решаемой задачи.

К числу основных преимуществ использования ***объектно-ориентированных моделей*** относятся:

* наглядность процесса перехода от алгоритмических процессов к программным моделям;
* наличие средств быстрой разработки приложений;
* эффективная организация процесса создания приложений за счет коллективной работы не­скольких специалистов над отдельными частями системы;
* возможность повторного использования кода в различных приложениях и применение компо­нентов, написанных другими разработчиками, что позволяет без больших усилий расширять функциональность программ;
* улучшенные средства взаимодействия с распределенными приложениями и операционными системами;
* функции создания интуитивных графических интерфейсов пользователя.

Итак, основным элементом деятельности, ведущей к созданию первоклассного программного обеспечения, является моделирование. Модели позволяют нам наглядно продемонстрировать желаемую структуру и поведение системы. Они также необходимы для визуализации и управления ее архитектурой. Модели помогают добиться лучшего понимания создаваемой нами системы, что зачастую приводит к ее упрощению и возможности повторного использования. Наконец, модели нужны для минимизации риска.

Неудачные проекты заканчиваются крахом в силу самых разных причин, а вот успешные, как правило, имеют много общего. Хотя успех программного проекта обеспечивается множеством разных слагаемых, одним из общих является применение моделирования.

Моделирование – это устоявшаяся и повсеместно принятая инженерная методика. Мы строим архитектурные модели зданий, чтобы помочь их будущим обитателям во всех подробностях представить себе готовый продукт. Иногда прибегают даже к математическому моделированию зданий, чтобы учесть влияние сильного ветра или землетрясения.

Итак, что же такое модель? Попросту говоря, она является упрощенным представлением реальности. Модель – это визуальное отображение системы (чертеж системы): в нее может входить как детальный план, так и более абстрактное представление системы. Хорошая модель всегда включает элементы, существенно влияющие на результат, и не включает те, которые малозначимы на данном уровне абстракции. Каждая система может быть описана с разных точек зрения, для чего используются различные модели, каждая из которых, следовательно, является семантически замкнутой абстракцией системы. Модель может быть структурной, подчеркивающей организацию системы, или поведенческой, то есть отражающей ее динамику.

Зачем мы моделируем? На это есть фундаментальная причина. Мы строим модели для того, чтобы лучше понимать разрабатываемую прикладную программу.

Моделирование позволяет решить четыре различных задачи:

* визуализировать программную систему в ее текущем или желательном для нас состоянии;
* определить структуру или поведение системы;
* получить шаблон, позволяющий затем сконструировать систему;
* документировать принимаемые решения, используя полученные модели.

Моделирование предназначено не только для создания больших систем. Чем больше и сложнее система, тем большее значение приобретает моделирование при ее разработке. Дело в том, что моделировать сложную систему необходимо, поскольку иначе мы не можем воспринять ее как единое целое.

Восприятие человеком сложных сущностей ограничено. Моделируя, мы сужаем проблему, заостряя внимание в данный момент только на одном аспекте. Кроме того, моделирование усиливает возможности человеческого интеллекта, поскольку правильно выбранная модель дает возможность создавать проекты на более высоких уровнях абстракции.

На практике даже при реализации простейшего проекта разработчики в той или иной мере применяют моделирование, хотя бы неформально. Для визуализации части системы ее проектировщик может нарисовать что-то на доске или на клочке бумаги. Ничего плохого в таких моделях нет. Если они работают, их существование вполне оправдано. Но эти неформальные модели часто создаются для однократного применения и не обеспечивают общего языка, который был бы понятен другим участникам проекта.

От моделирования может выиграть любой проект. Даже при создании одноразовых программ, когда зачастую бывает полезнее выбросить неподходящий код из-за преимущества в скорости разработки, которые дают языки визуального программирования, моделирование поможет коллективу разработчиков лучше представить план системы, а значит, выполнить проект быстрее и создать именно то, что подразумевал изначальный замысел. Чем сложнее проект, тем более вероятно, что из-за отсутствия моделирования он потерпит неудачу или будет создано не то, что нужно. Все полезные и интересные системы с течением времени обычно усложняются. Пренебрегая моделированием в самом начале создания системы, вы, возможно, горько пожалеете об этом, когда будет уже слишком поздно.

Моделирование имеет богатую историю во всех инженерных дисциплинах. Длительный опыт его использования позволил сформулировать четыре основных принципа создания моделей [5].

***Во-первых, выбор модели оказывает определяющее влияние на подход к решению проблемы и на то, как будет выглядеть это решение***. Иначе говоря, подходите к выбору модели вдумчиво. Правильно выбранная модель высветит самые коварные проблемы разработки и позволит проникнуть в самую суть задачи, что при ином подходе было бы попросту невозможно. Неправильная модель заведет вас в тупик, поскольку внимание будет заостряться на несущественных вопросах.

Можно с уверенностью сказать, что взгляд на мир существенно зависит от выбираемой модели. Если смотреть на систему глазами разработчика баз данных, то основное внимание будете уделять моделям "сущность-связь", где поведение инкапсулировано в хранимых процедурах. Структурный аналитик, скорее всего, создал бы модель, в центре которой находятся алгоритмы и передача данных от одного процесса к другому. Результатом труда разработчика, пользующегося объектно-ориентированным методом, будет система, архитектура которой основана на множестве классов и образцах взаимодействия, определяющих, как эти классы действуют совместно. Любой из этих вариантов может оказаться подходящим для данного приложения и методики разработки, хотя опыт подсказывает, что объектно-ориентированная точка зрения более эффективна при создании гибких архитектур, даже если система должна будет работать с большими базами данных или производить сложные математические расчеты. При этом надо учитывать, что различные точки зрения на мир приводят к созданию различных систем, со своими преимуществами и недостатками.

Второй принцип формулируется так: каждая модель может быть воплощена с разной степенью абстракции.

При строительстве небоскреба может возникнуть необходимость показать его с высоты птичьего полета, например, чтобы с проектом могли ознакомиться инвесторы. В других случаях, наоборот, требуется самое детальное описание – допустим, чтобы показать какой-нибудь сложный изгиб трубы или необычный элемент конструкции.

То же происходит и при моделировании программного обеспечения. Иногда простая и быстро созданная модель пользовательского интерфейса – самый подходящий вариант. В других случаях приходится работать на уровне битов, например, когда вы специфицируете межсистемные интерфейсы или боретесь с узкими местами в сети. В любом случае лучшей моделью будет та, которая позволяет выбрать уровень детализации в зависимости от того, кто и с какой целью на нее смотрит. Для аналитика или конечного пользователя наибольший интерес представляет вопрос «что», а для разработчика – вопрос «как». В обоих случаях необходима возможность рассматривать систему на разных уровнях детализации в разное время.

Третий принцип: лучшие модели – те, что ближе к реальности.

Физическая модель здания, которая ведет себя не так, как изготовленная из реальных материалов, имеет лишь ограниченную ценность. Математическая модель самолета, для которой предполагаются идеальные условия работы и безупречная сборка, может и не обладать некоторыми характеристиками, присущими настоящему изделию, что в ряде случаев приводит к фатальным последствиям. Лучше всего, если ваши модели будут во всем соотноситься с реальностью, а там, где связь ослабевает, должно быть понятно, в чем заключается различие и что из этого следует. Поскольку модель всегда упрощает реальность, задача в том, чтобы это упрощение не повлекло за собой какие-либо существенные потери.

Возвращаясь к программному обеспечению, можно сказать, что "ахиллесова пята" структурного анализа – несоответствие принятой в нем модели и модели системного проекта. Если этот разрыв не будет устранен, то поведение созданной системы с течением времени начнет все больше отличаться от задуманного. При объектно-ориентированном подходе можно объединить все почти независимые представления системы в единое семантическое целое.

***Четвертый принцип заключается в том, что нельзя ограничиваться созданием только одной модели***. Наилучший подход при разработке любой нетривиальной системы – использовать совокупность нескольких моделей, почти независимых друг от друга.

Если вы конструируете здание, то никакой отдельный комплект чертежей не поможет вам прояснить до конца все детали. Понадобятся, как минимум, поэтажные планы, виды в разрезе, схемы электропроводки, центрального отопления и водопровода.

При разработке программного обеспечения тоже существует несколько подходов к моделированию. Важнейшие из них – алгоритмический и объектно-ориентированный подход, о котором мы говорили в п.**В.1**.

**Алгоритмический метод** представляет традиционный подход к созданию программного обеспечения. Основными строительными блоками являются процедуры, а внимание уделяется, прежде всего, вопросам передачи управления и декомпозиции больших алгоритмов на меньшие. Ничего плохого в этом нет, если не считать того, что системы не слишком легко адаптируются. При изменении требований или увеличении размера приложения (что происходит нередко) сопровождать их становится сложнее.

**Объектно-ориентированный метод** в качестве основных строительных блоков использует **объект** или **класс**. В самом общем смысле объект – это сущность, обычно извлекаемая из словаря предметной области или решения, а класс является описанием множества однотипных объектов. Каждый объект обладает идентичностью (его можно поименовать или как-то по-другому отличить от прочих объектов), состоянием (обычно с объектом бывают связаны некоторые данные) и поведением (с ним можно что-то делать или он сам может что-то делать с другими объектами).

Объектно-ориентированный подход к разработке программного обеспечения является сейчас преобладающим просто потому, что он продемонстрировал свою полезность при построении систем в самых разных областях любого размера и сложности. Кроме того, большинство современных языков программирования, инструментальных средств и операционных систем являются в той или иной мере объектно-ориентированными, а это дает веские основания судить о мире в терминах объектов.

***Сущности*** - это абстракции, являющиеся основными элементами модели. Сущности являются основными объектно-ориентированными блоками языка. С их помощью можно создавать корректные модели.

Как мы уже говорили, при объектно-ориентированном подходе в объектной модели предметная область может быть представлена в виде совокупности объектов, которые объединены в классы. Классы могут существовать на нескольких уровнях представления: концептуальном – уровне анализа, уровне проектирования и уровне реализации. При моделировании объектов, выражающих концепции реального мира, очень важно отразить их действительное состояние, поведение и взаимо­отношения. Причем концепции, относящиеся к реализации (сокрытие информации, производительность, видимость и методы), не являются концепциями реального мира, поэтому многие потенциальные свойства класса являются на этом уровне ненужными.

*На* ***концептуальном уровне*** классы описываются в терминах реальных или предполагаемых сущностей из предметной области.

*На* ***логическом******уровне (анализе)*** классы представляют собой логическую концепцию предметной области. Аналитическая модель вообще содержит только самые общие представления о моделируемой системе. На этом уровне важно уловить основную логику системы, в том числе отношений между классами, а не вни­кать в детали ее работы и построения.

*На* ***начальном уровне проектирования*** для классов становятся актуальны такие концепции, как сопоставление состояния с классами, эффективность переходов между объектами, отделение внеш­него представления от внутренней реализации и определение точных сигнатур операций.

В дальнейшем класс объединяет в единый пакет информацию о состоянии и операциях, отражает основные проектные решения по локализации информации и распределению функциональности. На этом уровне класс обладает содержанием, относящимся как к реальному миру, так и непосредственно к моделируемой системе.

И*,* наконец*,* ***при реализации в программном коде*** класс принимает форму, максимально подходящую для отображения в выбранном языке программирования. При этом могут пропасть те об­щие свойства класса, которые не находят в этом языке прямого применения. На уровне реализации класс уже в точности отражает программный код.

*Таким образом,* ***модель ПО*** – это формализованное описание системы ПО на определенном уровне абстракции. Каждая модель описывает конкретный аспект системы, использует набор диаграмм или формальных описаний и документов заданного формата, а также отражает точку зрения и является объектом деятельности различных людей с конкретными интересами, ролями или задачами. Модели служат полезным инструментом анализа проблем, обмена информацией между всеми заинтересованными сторонами, проектирования ПО. Моделирование способствует более полному усвоению требований, улучшению качества системы и повышению степени ее управляемости.

### В.3. Основные средства разработки приложений

### на платформе Visual Studio.NET

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.

***Программирование*** – процесс ***создания программного обеспечения с помощью языков программирования***

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 программного 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; обширные [библиотеки стандартных подпрограмм](http://www.tomsk.ru/Books/informatica/theory/chapter6/1_6_18.html) и функций; [отладочные средства](http://www.tomsk.ru/Books/informatica/theory/chapter8/1_8_6.html), помогающие находить и устранять ошибки в программе др.

«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, необходимо превратить его в абсолютно точный набор инструкций, которые однозначно будут интерпретировать ПК. Для этого были созданы алгоритмические 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.

Говоря об уровнях, прежде всего, имеются в виду степени приближенности языка к процессору компьютера. Уровень в данном случае – это уровень машинного восприятия. Так, языки низкого уровня (ассемблер) приближены к компьютеру, что делает соответствующие программы особенно эффективными с точки зрения их быстродействия. Однако существенная проблема использования таких языков заключается в том, что программист – это 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.

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

Под ***транслятором*** обычно понимают специальную системную 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 VB, C, C# и других, обычно называют ***компиляторами***. Этим подчёркивается общепринятый для подобных языков режим трансляции, при котором вначале осуществляется перевод программы в двоичное представление (объектный модуль), 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 компиляторов по сравнению с интерпретаторами относятся:

* в компиляторах процесс построения (создания) исполняемого модуля выполняется один раз, а не при каждом запуске, что экономит время.
* в компиляторах 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 библиотек 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***, которая обеспечивает поиск вспомогательных подпрограмм в 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 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. В конечном счете, сложность задач приводит к росту числа ошибок в программе, с которыми помогает справиться отладка.

Мы начали рассмотрение системных программирования с её ключевых частей – ***транслятора*** и ***редактора связей***, которые существовали с самого начала развития средств автоматизации программирования. А вот процесс «написания» программ долгое время оставался «ручным». В настоящее время – время ***персональных компьютеров*** (ПК) – этот рутинный процесс ушёл в прошлое. Современный программист, как правило, вводит код программы в компьютер, пользуясь ***редакторами программного кода или текстовыми процессорами.***

***Редактор программного кода*** – это программная подсистема, обеспечивающая первоначальную подготовку исходного текста программы, его форматирование и 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 программных проектов существует возможность частично избавить себя от рутинного написания стандартных фрагментов программного кода с помощью, так называемых ***средств автоматизированной генерации кода***, а в некоторых случаях эти средства умеют самостоятельно создавать программный код, выполняющий определенные стандартные действия. Примером таких средств может служить Microsoft Visual Studio .NET (VS), 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.

Система программирования содержит много компонент. Это и ***редакторы кода***, и ***трансляторы***, и ***сборщик***, и ***отладчик***, 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.

Для 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 и др.

***Интегрированная среда разработки*** (**IDE** – Integrated Development Environment*)* – это специальная программная система, предоставляющая возможность удобной совместной работы с различными компонентами системы программирования – набор инструментов для разработки и отладки программ, имеющий общую интерактивную графическую оболочку, поддерживающую выполнение всех основных функций жизненного цикла разработки программы – набор и редактирование исходного текста (кода), компиляцию (сборку), исполнение, отладку, профилирование и др.

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 программной системой. Однако время, потраченное на ее изучение, окупается в дальнейшем. И, наконец, еще один положительный момент – устройство большинства сред одинаково в концептуальном плане.

Одно из последних достижений в области разработки программного обеспечения – визуальные среды программирования. Самая известная – многоязыковая среда Microsoft® Visual Studio .NET.

Корпорация Microsoft внесла особо выдающийся вклад в развитие интегрированных сред, благодаря созданию и развитию среды Visual Studio, которая является одним из лучших образцов современной интегрированной среды. Ее новую версию, Visual Studio 2013, мы и рассмотрим в данном пособии.

***Рассмотрим основные понятия этой среды.***

***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 (прикладных программ) на нескольких языках программирования – Visual Studio .NET. Созданные приложения выполняются при помощи надстройки над операционной системой – 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 ввиду общие принципы и правила создания и использования приложений .NET, представленные в виде описаний, спецификаций и стандартов.

***Операционная среда .NET Framework*** – это важнейшая (основная) 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 управлением 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 четырех языков – VB, C#, C++ и F#. И все же именно первые два фактически олицетворяют собой эту систему. Ведь C++, 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,*** будучи частью платформы .NET является надстройкой над операционной системой, в качестве которой может быть Windows. Платформа .NET Framework включает в себя:

* четыре официальных языков программирования: VB, C#, C++ и F#;
* объектно-ориентированную среду исполнения CLR (Common Language Runtime – общеязыковая среда исполнения), под управлением которой создаются и выполняются различные приложения;
* ряд связанных между собой библиотек классов под общим именем FCL (Framework Class Library – библиотека классов каркаса), которые можно использовать при программировании на любом языке .NET.

Таким образом, в .NET Framework можно выделить два основных компонента платформы (каркаса):

* 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 **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.

1. ***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.
2. ***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. Например, 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.
3. ***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.
4. ***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***.***
5. ***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 **System** 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. 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 платформы .NET Framework 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, который не содержит команд, зависимых от языка, операционной системы и типа компилятора. 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 расширения - **exe, dll, mod** или **mdl**.

Необходимо отметить, что PE-файл, имеющий расширение **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 и выполняет код.

***Виртуальная машина***. Отделение каркаса от среды разработки приложения VS явилось естественным шагом. Каркас Framework .Netперестал быть частью среды разработки приложения VS, 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.

Таким образом, среда CLR может быть реализована для любой операционной системы. При выполнении программы CLR вызывает так называемый JIT-компилятор, переводящий код с языка IL в машинные команды конкретного процессора, которые немедленно выполняются. Причем компилируются только те части программы, которые требуется выполнить в данный момент. Каждая часть программы компилируется только один раз в нужный момент и сохраняется оперативной памяти для дальнейшего использования.

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.

***Метаданные.*** Переносимый исполняемый PE-файл является само документируемым 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.

***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****.*

***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 языке VB выбрасывание исключений, их дальнейший перехват и обработка – основной рекомендуемый способ обработки исключительных ситуаций – рассмотрен в **Теме 4**.

***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. События и делегаты также рассмотрены в **Теме 4**.

Исполнительная среда CLR обладает мощными динамическими механизмами – сборки мусора, динамического связывания, обработки исключительных ситуаций и событий. Все эти механизмы и их реализация в CLRсозданы 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# создавались одновременно с созданием CLR**,** то, естественно, они стали языками, наиболее согласованными 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.

В настоящее время компания Microsoft выпустила версию Visual Professional, Ultimate 2012 – 2013, которая покрывает 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.

Можно воспользоваться программой от Microsoft – **DreamSpark** (https://www.dreamspark.com/Default.aspx), которая создана для поддержки технического образования путем предоставления доступа к программному обеспечению компании для учебных, преподавательских и исследовательских целей.

Также вам следует узнать, подписано ли ваше учебное заведение на эту программу, в этом случае вы получите больший выбор программного обеспечения. Если вы преподаватель, то вы тоже получите доступ к программе, если ваше учебное заведение оформит подписку. Можно предложить администрации своего учебного заведения подписаться на программу DreamSpark.

### В.4. Общие рекомендации по разработке учебных программ средствами ООП

ВСТАВИЛА АБЗАЦ В САМЫЙ КОНЕЦ

Объектно-ориентированное программирование никак не связано с процессом выполнения программы, а является лишь способом её организации. Основное внимание при его изучении необходимо уделять организации программы, а не вопросам написания кода. С использованием объектно-ориентированного подхода надо построить объектную модель задачи, проведя объектно-ориентированную декомпозицию. То есть вместо проблемы разбиения задачи на процедуры надо решать проблему разбиения её на классы объектов. Главным компонентом объектно-ориентированной программы является класс объектов, содержащий данные (поля) и методы (процедуры) для их обработки. Класс является формой или образцом для множества сходных между собой объектов.

Что должно в программе представляться в виде класса объектов? Каждый программист решает эту задачу по-своему, когда, анализируя предметную область, вычленяет из неё отдельные объекты. Это могут быть: физические объекты (автомобили при моделировании уличного движения, страны при создании экономической модели); элементы интерфейса (окна, меню, мышь); структуры данных (массивы); группы людей (студенты, покупатели, продавцы); хранилища данных (списки сотрудников, словари, описи); пользовательские типы данных (комплексные числа, точки на плоскости) и т.п. Соответствие между реальными и программными объектами является следствием объединения данных и процедур (методов).

При разработке программных приложений необходимо тщательно изучить потенциальных пользователей и их требования и составить диаграмму вариантов использования (модель), которая демонстрирует действующие субъекты и инициируемые ими операции (варианты использования). Любое существительное из описаний вариантов использования может в будущем стать именем класса или атрибута. Глаголы превращаются в методы.

При создании класса, то есть нового типа данных следует хорошо продумать его интерфейс пользователя – средства работы с классом, доступные потенциальным пользователям приложения. Как правило, он не должен включать поля данных, которые предпочтительнее делать закрытыми (**Private**). Это дает возможность впоследствии изменить реализацию класса без изменений в его интерфейсе, а также регулировать доступ к полям класса с помощью свойств и методов этого класса. Важно помнить, что поля класса вводятся только для того, чтобы реализовать характеристики класса, предоставляемые в его интерфейсе пользователя с помощью свойств и методов. В идеале интерфейс пользователя должен быть полным, т.е. предоставлять возможность выполнять любые разумные действия с классом, и одновременно минимально необходимым – без дублирования и пересечения возможностей методов.

Методы определяют поведение класса. Каждый метод класса должен решать только одну задачу, не стоит объединять два даже коротких, но независимых фрагмента кода в один метод. А если метод реализует сложные действия, следует разбить его на несколько последовательных шагов и каждый шаг оформить в виде отдельной процедуры (закрытого метода с доступом **Private**). Для реализации принципа инкапсуляции следует максимально сокращать область действия каждой переменной, т.е. величины, используемые только в процедуре, надо описывать внутри неё как локальные переменные. Такой подход упрощает отладку программы, так как область поиска ошибки ограничена.

Поля и методы, характеризующие класс в целом, то есть имеющие одно и то же значение для всех экземпляров, следует описывать как разделяемые (статические – **Shared**).

Механизм наследования позволяет создавать новые классы на основе существующих. Класс-наследник наследует все данные и методы родительского класса, но также имеет свои собственные, что предоставляет программисту богатейшие возможности организации кода и его повторного использования.

На уровне базового класса можно написать универсальный код, с помощью которого работать также с объектами производного класса, что реализуется с помощью виртуальных методов. Как виртуальные следует описывать методы, которые во всех классах иерархии выполняют одну и ту же функцию, но, возможно, разными способами.

Абстрактные классы используются для представления общих понятий, которые предполагается конкретизировать в производных классах, то есть в абстрактном классе задается набор методов (интерфейс), который потомки будут реализовывать по-своему. ВСТАВИЛА АБЗАЦ НИЖЕ

Рекомендуется предусматривать реакцию на неверные входные данные генерацией исключения или печатью сообщения. Следует заключать потенциально опасные фрагменты программы в блок обработки исключений **Try** и обрабатывать хотя бы исключение **Exception**, а лучше по отдельности все исключения, которые могут в нем возникнуть.

### В.5. Контрольные вопросы

1. Сформулируйте определение интегрированной среды разработки программ.
2. Каковы основные компоненты интегрированной среды?
3. Что такое парадигма программирования?
4. Сформулируйте основные принципы структурного (процедурного) программирования.
5. Сформулируйте основные принципы объектно-ориентированного программирования.
6. Что такое процедурно-ориентированный подход к программированию? Каковы его недостатки?
7. В чем преимущества объектно-ориентированного программирования?
8. Что такое сборка программ?
9. Что такое отладчик и каковы его типовые команды?
10. Что такое моно языковые и многоязыковые интегрированные среды?
11. Что такое .NET?
12. Каковы основные принципы .NET?
13. Что такое общая система типов .NET?
14. Что такое общая система поддержки выполнения .NET?
15. Что такое единый промежуточный язык .NET?
16. Что такое объект?
17. Почему необходимо моделировать?