Государственный университет – Высшая школа экономики

Факультет Бизнес-Информатики

Кафедра Основ информатики
и прикладного программного обеспечения

Объектно-ориентированный анализ и программирование на языке **C# (C_Sharp)**

Материалы к 5-й лекции

Проф. Забудский Е.И.

Москва 2007

Лекция 5

Тема 3. Интерфейс (interface) – аналог множественного наследования

Основное назначение интерфейсов *– обеспечить возможность классу иметь* ***несколько родителей*** *-* один *полноценный класс (*базовый*),* ***а остальные*** *в виде* ***интерфейсов.***

см. также Материалы к Практ. зан. 7, раздел 9

Наследование – механизм, дающий возможность создавать новый класс на основе уже существующего класса и использовать в новом классе его свойства и методы

Интерфейс (interface**) – частный случай класса;** то есть interface – это полностью абстрактный класс

NB	**Задание на дом по итогам 2-го модуля:** **Написать C#-программу – Банковский счет — применение простого наследования.** **Программу реализовать в среде MS VS .NET 2005. Получить результат в консольном варианте.** **См. с. 40…51 в Материалах к 3-й лекции**
	Срок представления кода и результата – 1-е занятие в 3-м модуле, 10.01. 2007 г.

Уважаемые студенты!

Основная цель, которую необходимо достигнуть в результате изучения дисциплины Объектно-ориентированный анализ и программирование – научиться разрабатывать компьютерные модели реальных и концептуальных систем соответствующих направлению Бизнес-информатика.

Необходимым условием усвоения дисциплины является ВАША самостоятельная работа

Советую Вам все материалы, подготовленные мной к лекциям и практическим занятиям, распечатать и прорабатывать их! Приведенные C#-программы реализовать в среде MS VS .NET 2005 и разобраться в них.

// **КОММЕНТАРИЙ**. На сайте http://www.firststeps.ru/ “Первые шаги” представлено много интересных обучающих материалов по различным интегрированным средам и языкам программирования, в том числе представлены **C#** и платформа **.NET** (step by step).

Данное пособие распространяется свободно. Изложенный материал, предназначенный для публикации в “твердом” варианте, дополнен и откорректирован.

© Забудский Е.И., 2007

Содержание

Тема следующей (их) лекции (й):
«Программная модель Windows Forms — основа для разработки приложений .NET Framework с графическим интерфейсом пользователя».

1. Наследование: единственный способ программировать

Наследование (**inheritance**) в объектно-ориентированном программировании очень похоже на то, как мы наследуем характеристики наших родителей. Характеристики в терминах объектно-ориентированного программирования — это **атрибуты и варианты поведения класса, то есть данные и методы класса.** Биологическое наследование создает иерархическую классификацию, что позволяет вам проследить наследование по поколениям родственников, которые существовали до вас. То же самое справедливо и для объектно-ориентированного программирования. ***Вы можете проследить эти отношения, чтобы определить происхождение класса***.

Наследование является краеугольным камнем объектно-ориентированного программирования, так как оно позволяет объектам наследовать атрибуты и поведение других объектов, таким образом, уменьшая объем нового кода, который надо спроектировать, написать и протестировать каждый раз, когда вы разрабатываете новую программу.

Наследование обеспечивает распределение контроля над разработкой и поддержкой объектов. Например, программист может быть ответственен за создание и поддержку объекта ***студент***. Другой программист может разрабатывать и поддерживать объект ***студент-выпускник***. Всякий раз, когда происходит изменение, касающееся всех студентов, эти изменения осуществляются в объекте ***студент*** и наследуются объектом ***студент-выпускник***. Эти изменения надо вносить только программисту, ответственному за объект ***студент***, так как объект ***студент-выпускник*** наследует все изменения, сделанные в объекте ***студент***. Наследование также обеспечивает ограничение доступа к атрибутам и поведению. С помощью спецификаторов доступа **public**, **private** и **protected** определяются части программы, из которых можно осуществлять доступ к атрибутам и поведению.

1.1. Иерархия классов

Иерархические отношения между классами иногда называются отношениями «**родитель-потомок**» (**parent-child relationship**). В отношении «**родитель-потомок**» потомок наследует все атрибуты и варианты поведения родителя, а родительские спецификаторы доступа используются для контроля того, каким образом эти унаследованные элементы будут доступны другим классам или методам. В **C#** (как и в C++) **родитель называется базовым классом**, а **потомок — производным классом**. В Java родитель называется *суперклассом,* потомок — *подклассом.* Независимо от терминологии, отношения и функциональные возможности родительского класса и класса-потомка в этих языках одинаковые.

Определение отношения «**родитель-потомок**» во многих ситуациях является интуитивным. Например, легко увидеть, что ***студент*** является родителем для ***студента-выпускника***, так как ***студент-выпускник*** имеет те же самые атрибуты и варианты поведения, что и ***студент***, и еще некоторые свои. Однако иногда эти отношения иллюзорны, так как они не ясны — может быть, их вовсе нет.

Для того чтобы определить, существует ли отношение между классами, программисты используют тест «**is a**» («**является**»). Тест «**is a**» определяет, «**является**» ли **потомок** родителем. Например, ***студент-выпускник*** «**является**» ***студентом***. Если отношение «**is a**» имеет смысл, это означает, что существует отношение «**родитель-потомок**» и что **потомок** может быть унаследован от **родителя**. Если отношение «**is a**» не имеет смысла, то отношения «**родитель-потомок**» не существует и **потомок** не может наследоваться от **родителя**, как, например, в случае автомобиля и самолета. Автомобиль «является» самолетом? Это ерунда, так что вы не должны пытаться создать такое отношение.

1.2. Типы наследования

Существует три способа реализации наследования в программе:

A. Простое, **см. разд. 1.2.1**
B. Многоуровневое, **см. разд. 1.2.2, с. 5,сл.**
C. и **Множественное** и наследование **см. разд. 1.2.3, с.11,сл**.

Каждый из этих типов позволяет классу обращаться к атрибутам и поведению другого класса, используя немного разные методы.

1.2.1. Простое наследование

Простое наследование **используется**, **если** существует одно отношение «**родитель-потомок**». То есть один потомок наследуется от одного родителя. Простое наследование показано на рис. 1. На этой диаграмме приведены два класса: **Student** и **GradStudent**. Класс **Student** в этом отношении является родительским и наследуется классом **GradStudent**, являющимся потомком.

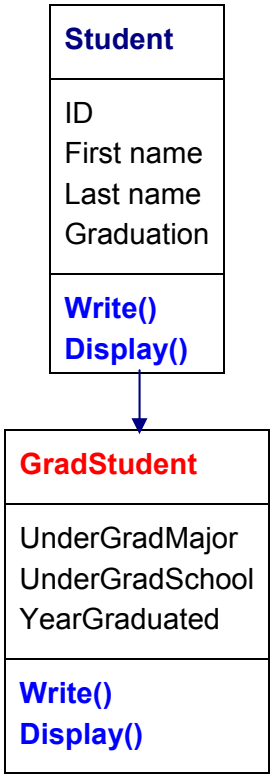


Рис. **1**. **Простое наследование** состоит из одного отношения «**родитель-потомок**». Класс **Student** здесь является (базовым) родительским, а класс **GradStudent** — (производным) потомком

Наследование происходит от родителя к потомку. **Родительский** класс не может получить доступ к атрибутам и поведению класса-**потомка**. На рис. 1 класс **Student** не может вызвать члены **Write()** **и** **Display()** класса **GradStudent**. Однако класс **GradStudent** может вызывать версии этих членов класса **Student** **(Пример Простого наследования - см. листинг 6, лекция 3, с.30,сл.**).

1.2.2. Многоуровневое наследование

Многоуровневое наследование появляется, когда **потомок** наследуется от **родителя**, а затем сам становится **родителем**. Это может показаться немного запутанным, но все станет ясно, если посмотреть на рис. **2**, на котором классы **Person**, **Student** и **GradStudent** образуют многоуровневое наследование.

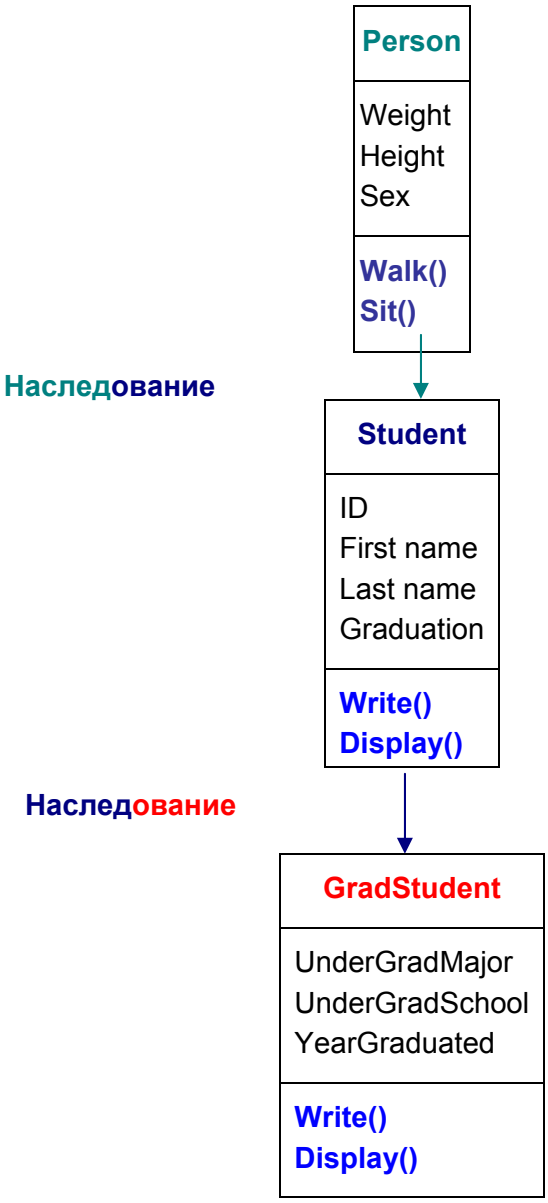


Рис. **2**. **Многоуровневое наследование**

Класс **Person** — это родительский (**базовый**) класс, который наследуется классом **Student**. Класс **Student** — это класс-потомок (**производный**) в отношении классов **Person**-**Student**. Однако появляется другое отношение «**родитель**-**потомок**», когда класс **GradStudent** наследуется от класса **Student**. В этом отношении класс **Student** является родителем, а класс **GradStudent** — потомком. Это означает, что класс **Student** играет двойную роль в наследовании. Он является и потомком, и родительским классом. Каждое отношение «родитель-потомок» считается уровнем. То есть отношение классов **Person**-**Student** — это первый уровень, а отношение классов **Student**-**GradStudent** — второй. В вашей программе вы можете иметь столько уровней, сколько вам необходимо; однако **многие программисты** останавливаются на третьем уровне, так как при большем количестве уровней иерархией становится трудно управлять.

В многоуровневом наследовании последний класс-потомок, которым в предыдущем примере является класс **GradStudent**, наследует атрибуты и поведение всех классов в цепочке наследования. Вот как это работает: класс **Student** наследует атрибуты и поведение класса **Person**. После того унаследования эти атрибуты и поведение считаются членами класса **Student**, как если бы они были определены в классе **Student**. Когда класс **GradStudent** наследуется от класса **Student**,

класс **GradStudent** имеет доступ к атрибутам и поведению класса **Student**, который теперь включает атрибуты и поведение, унаследованные от класса **Person**. Как вы помните, наследуются только те атрибуты и поведение, которые обозначены как **public** или **protected**.

Хотя последний класс-потомок (то есть класс **GradStudent**) не наследуется напрямую от класса **Person**, класс **Person** все равно должен проходить тест «**is a**». То есть ***студент-выпускник*** «является» («**is a**») *человеком.*

1.2.2.1. Многоуровневое наследование в C#

Многоуровневое наследование в C# реализуется с помощью объявления как минимум трех классов. Первые два класса имеют отношение «**родитель-потомок**», а второй и третий классы также должны иметь отношение «**родитель-потомок**». Каждый потомок, чтобы наследоваться от родительского класса, должен проходить тест «**is a**».

В **листинге 1** показана программа на **C#**. В этом примере определены три класса: **Person (строка 6…33)**, **Student (строка 35…56)** и **GradStudent (строка 58…87)**. Класс **Person** является базовым, и он является родительским для класса **Student**.

Класс **Student** определяет два метода в секции **public**. Это методы **Write() (строка 51)** и **Display() (срока 47)**. Метод **Write()** присваивает информацию о человеке, переданную ей в качестве аргументов атрибутам класса. Метод **Display()** отображает значения этих атрибутов на экране.

```
1  using System; //  МНОГОУРОВНЕВОЕ наследование                    Листинг 1
2  /* м-ду Write()  передается информация о Человеке, Студенте и Аспиранте;
     м-д Displey() отображает информацию о Человеке, Студенте и Аспиранте */
3
4  namespace Cons_Appl_OOP_Keo_c85_86
5  {
6     class Person                                   // базовый класс Person
7     {
8        protected int nID;
9        protected String sFirst;
10       protected String sLast;
11
12       public Person()                        // конструктор
13       {
14          nID = 0;
15          sFirst = "";
16          sLast = "";
17       }
18
19       public void Display()
20       {
21          Console.WriteLine();
22          Console.WriteLine("Идентиф. #: " + nID);
23          Console.WriteLine("Имя:        " + sFirst);
24          Console.WriteLine("Фамилия:    " + sLast);
25       }
26
27       public void Write(int ID, String First, String Last)
```

```
28      {                                   // информация о человеке - ID, First, Last
29          nID = ID;
30          sFirst = First;
31          sLast = Last;
32      }
33  }  ////////////end class Person/////////////////
34
35  class Student : Person
36  {                                   // произв-й класс Student : базовый класс Person
37      protected int nGraduation;
38
39      public Student()                        // конструктор
40      {
41          nGraduation = 0;
42
43      }
44
45      public new void  Display()
46      {
47          base.Display();                     // строка 19
48          Console.WriteLine("Год окончания: " + nGraduation);
49      }
50
51      public void Write(int ID, String First, String Last, int Graduation)
52      {                                   // информация о студенте : Graduation - окончание
53          base.Write(ID, First, Last);            // строка 27
54          nGraduation = Graduation;
55      }
56  }  ////////////end class Student////////////
57
58  class GradStudent : Student
59  {                   // произв-й класс GradStudent (аспирант): базовый класс Student
60      protected String sMajor;
61      protected String sUndergradSchool;
62      protected int nUndergradGraduation;
63
64      public GradStudent()                    // конструктор
65      {
66          sMajor = "";
67          sUndergradSchool = "";
68          nUndergradGraduation = 0;
69      }
70
71      public new void Display()
72      {
73          base.Display();                                 // строка 19
74          Console.WriteLine();
```

```
75          Console.WriteLine("Специальность: " + sMajor);
76          Console.WriteLine("Год окончания учебы в вузе: " + nUndergradGraduation);
77          Console.WriteLine("Название вуза: " + sUndergradSchool);
78      }
79
80  public void Write(int ID, String First, String Last, int Graduation, String Major, String
                        UndergradSchool, int UndergradGraduation)
81      {       // информация об аспиранте - Major, UndergradSchool, UndergradGraduation
82          base.Write(ID, First, Last, Graduation);            // строка 27
83          sUndergradSchool = UndergradSchool;
84          sMajor = Major;
85          nUndergradGraduation = UndergradGraduation;
86      }
87  }  ////////////////end class GradStudent/////////
88
89  public class StudentTest
90  {// начальный класс
91      public static void Main(String[] args)
92      {
93          Student s = new Student();                  // строка 39
94          GradStudent g = new GradStudent();          // строка 64
95          s.Write(100, "Иван", "Петров", 2008);       // строки 51, 53 и 27
96          g.Write(101, "Мария", "Алексеева", 2008, "Computer Science", "ГУ-ВШЭ", 2002);
                                                        // строки 80, 82 и 27
97          s.Display();            // строки 45, 47 и 19
98          g.Display();            // строки 71, 73 и 19
99          Console.ReadLine();
100     }
101 }  //////////end class StudentTest//////////
102 }
```

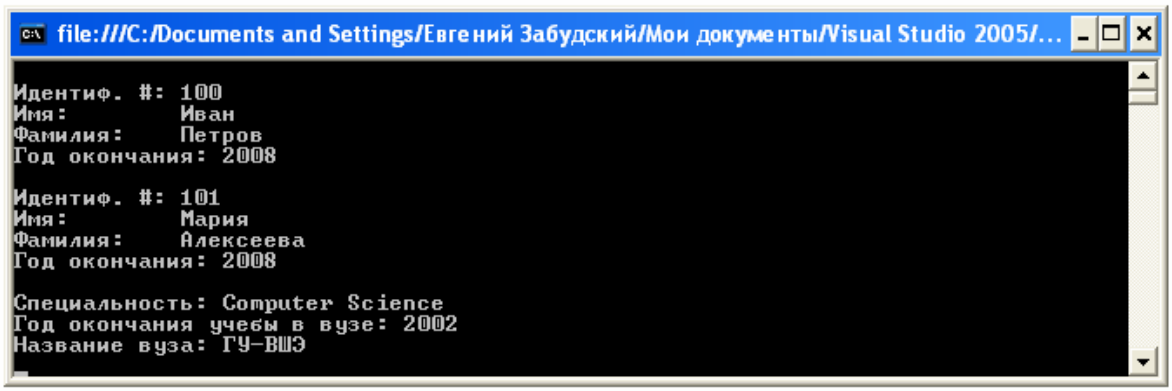


Рис. 3 Вывод программы **листинга 1**

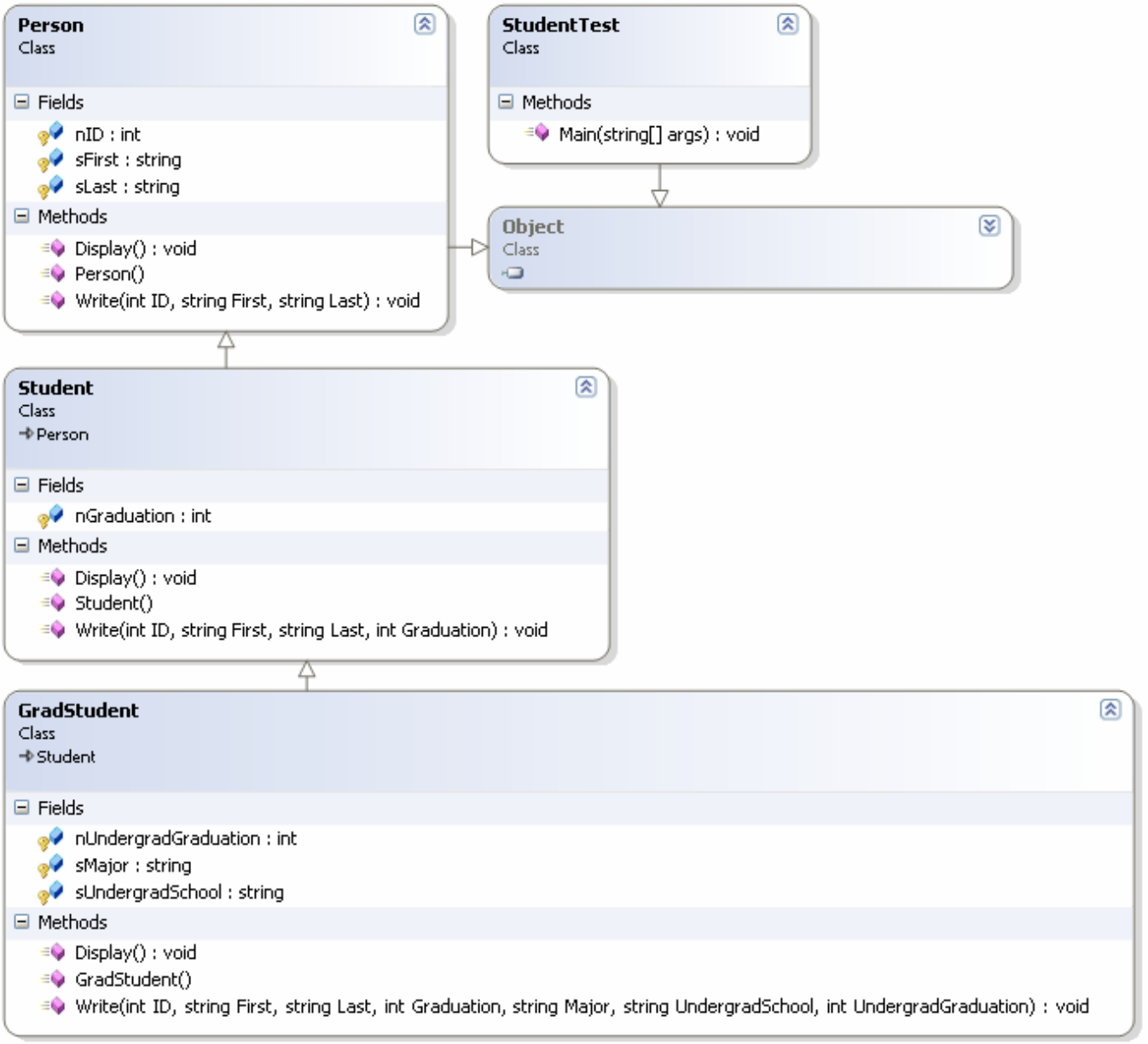


Рис. 4. Диаграмма классов программы **листинга 1**

Заметьте, что класс **Student** наследуется от класса **Person**, а класс **GradStudent** — от класса **Student**. Класс **Student** является и производным, и базовым классом. Он является производным в отношении «**родитель-потомок**» с классом **Person**, а базовым — в отношении «**родитель-потомок**» с классом **GradStudent**.

Посмотрите внимательно на определения методов **Write(), Display()** **(строки 80… и 71…)** класса **GradStudent**, и вы заметите, что обе эти функции обращаются к методам классов **Person (строки 27… и 19…)** и **Student (строки 51… и 47…)**. Это осуществляется с помощью многоуровневого наследования.

Класс **Student** наследует члены класса **Person**, которые определены в секциях **public** и **protected (строки 8…10)**. Это наследование повторяется, когда класс **GradStudent** наследуется от класса **Student**. Любой член класса **Person**, который доступен классу **Student**, также доступен классу **GradStudent**.

В методе **Write()** передается информация: ***1)*** и о человеке, ***2)*** и о студенте, ***3)*** и о студенте-выпускнике. Вывод программы листинга **1** приведен на **рис. 3**, диаграмма классов – на **рис. 4**.

1.2.3. Множественное наследование и «проблема бриллианта»

Множественное наследование используется, когда отношение включает **нескольких родителей** и **одного потомка**. Другими словами, потомок наследуется более чем от одного родителя. Это показано на **рис. 5.** В этом примере класс **GradStudent** наследуется и от класса **Person**, и от класса **Student**. Классы **Person** и **Student** оба являются родителями для класса **GradStudent**, который в этом отношении является потомком.

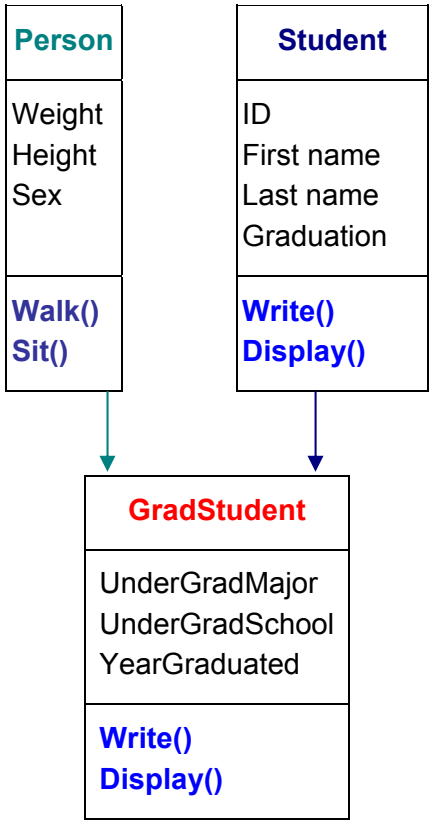


Рис. 5. В этом примере **множественного наследования** один класс **GradStudent** является потомком двух классов **Person** и **Student**

Класс **GradStudent** наследует характеристики человека из класса **Person**. Это атрибуты — вес, рост и пол, и методы **Walk()** **и** **Sit()**. Вы могли бы задаться вопросом, как ***студент-выпускник*** идет или сидит в программе. Сложно представить, как это делается. Хотя мы используем эти варианты поведения в целях иллюстрации, их можно запрограммировать в приложении виртуальной реальности, которое бы показывало ***студента-выпускника***, идущего через университетский городок на занятия.

При реализации множественного наследования вы должны помнить о нескольких факторах.

1. Каждый наследуемый класс должен проходить тест «**is a**». На **рис. 5** ***студент-выпускник*** должен быть и ***человеком***, и ***студентом***, чтобы наследоваться от обоих родителей. Если ***студент-выпускник*** не проходит этот тест, он не может наследоваться от соответствующего родителя.
2. Родительские классы ***независимы*** друг от друга. То есть класс **Student** не знает ничего о классе **Person**, и наоборот. Класс **Student** не может обращаться к атрибутам и поведению класса **Person**, так как только класс **GradStudent** наследуется от класса **Person**. Аналогично, класс **Person** не наследуется от класса **Student**.
3. Наследование осуществляется в одном направлении **от родителя к потомку**, что является

идентичным простому наследованию.

4. Любое количество родительских классов может наследоваться классом-потомком, если они проходят тест «**is a**».

NB	Множественное наследование может привести к интересной и потенциально запутывающей проблеме, известной как «**проблема бриллианта**» (**the diamond problem**). Представьте, что у вас есть класс **IdObject**, который содержит атрибут — идентификационный номер, содержащий уникальное значение. Унаследуем от этого класса классы **Student** и **Instructor**. Пока все хорошо: у нас есть классы **Student** и **Instructor**, оба они имеют унаследованный атрибут с уникальным значением. Теперь представьте, что вы создали класс **TeacherAssistant**, который унаследован от классов **Student** и **Instructor**. Этот новый класс имеет два уникальных идентификатора. Это и есть проблема «**бриллианта**», так как если мы нарисуем граф наследования, он будет выглядеть как бриллиант. Такая схема множественного наследования будет работать до тех пор, пока мы не попытаемся получить идентификатор класса **TeacherAssistant**: в этом случае мы не будем знать точно, который из них нам нужен (**рис. 6**). C++ поддерживает множественное наследование, тогда как **Java и C# его не поддерживают**. **Java и C# предоставляют нечто, называемое «интерфейсами»**, о чем более подробно будет рассказано далее в разделах 1.4 и 2. Вы увидите, что **механизм интерфейсов отличается от наследования**.

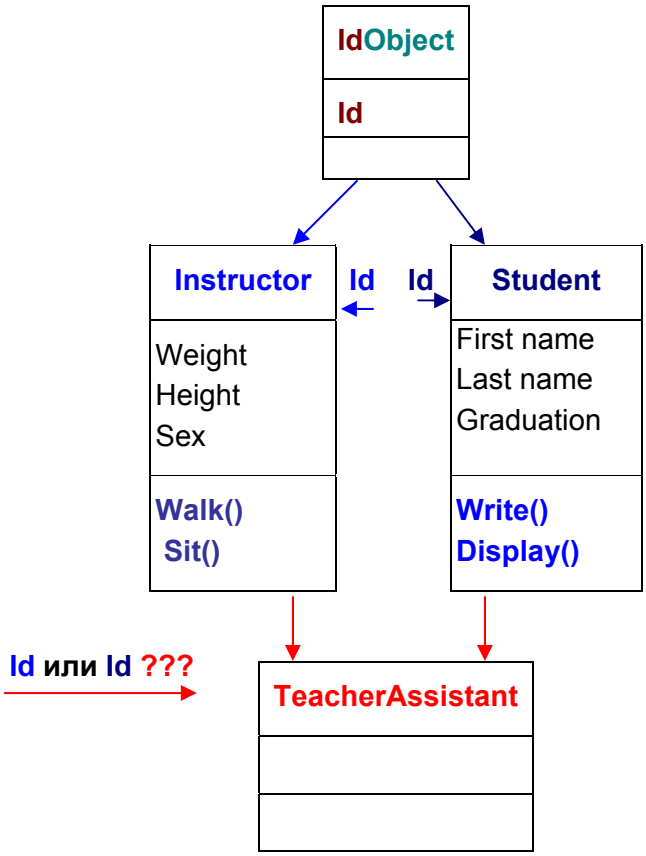


Рис. 6. К иллюстрации «**проблемы бриллианта**», возникающей при множественном наследовании

1.3. Выбор подходящего типа наследования

Если у вас есть класс-потомок, который наследуется от единственного класса-родителя, то простое наследование — ваш единственный выбор.

Приходится ломать голову, когда потомок напрямую или косвенно наследуется более чем от одного родителя. Это тот случай, когда класс **GradStudent** наследуется и от **класса Person**, и от **класса Student**. В этом случае у вас есть два варианта: ***1)*** использовать множественное наследование ***2)*** или многоуровневое наследование.

Некоторые программисты в этом случае принимают решение, основываясь на том, существует ли отношение «**is a**» между двумя родительскими классами. Например, является ли **студент человеком**? Если да, то лучше выбрать многоуровневое наследование, так как за счет этого мы получим естественное отношение между родительскими классами. То есть другие типы **студентов** помимо **GradStudent**, вероятно, унаследуют класс **Student** и класс **Person**. Как только класс **Student** унаследует класс **Person**, все классы **студентов** будут наследовать класс **Person** при наследовании от класса **Student**. Это показано на **рис. 7**, где классы **UndergradStudent**, **GradStudent** и **ContinuingEdStudent** наследуются от класса **Student** и косвенно наследуются от класса **Person**.

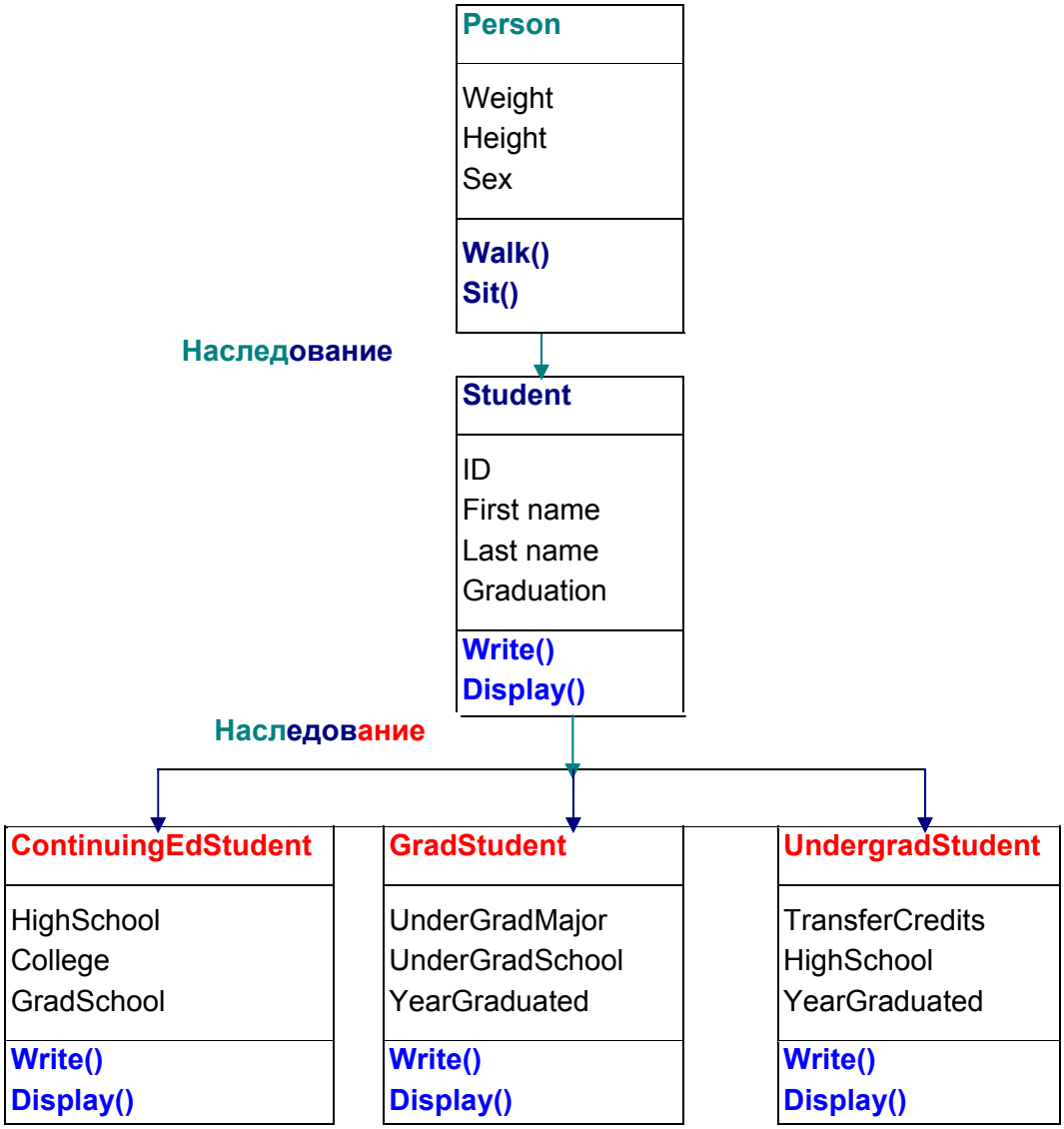


Рис.7. Если отношение наследования («**is-a**»)существует между двумя и более родительскими классами, следует использовать **многоуровневое** наследование.

В данном случае отношение между классами **Student** и **Person** существует благодаря тому, что студент является человеком

Напротив, **множественное наследование применяется, когда нет окончательного отношения между двумя родителями**. Отношение является окончательным, когда отношение всегда проходит тест «**is a**». Если отношение иногда проходит этот тест, а иногда — нет, то это **не окончательное** отношение.

Пусть, например, обучающийся студент является атлетом и писателем. Это означает, что класс **UndergradStudent** наследует атрибуты и поведение классов **Athlete** и **Writer**. Однако окончательного отношения между этими двумя родительскими классами не существует. То есть атлет может быть, а может не быть писателем, а писатель может быть, а может и не быть атлетом. Это тот самый случай, когда в программе лучше всего применять множественное наследование, как показано на **рис. 8**.

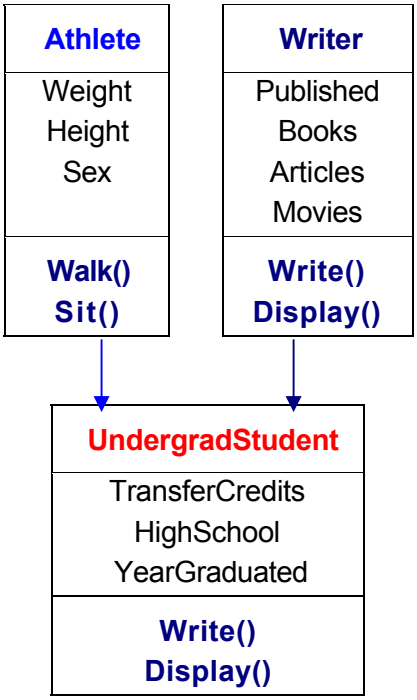


Рис.8. Если между двумя или более родительскими классами нет четко выраженного отношения «**is a**», то **используется множественное наследование**.
В этом примере такого отношения нет между классами **Athlete** и **Writer**

Существует важное отличие между **множественным** и **многоуровневым** наследованием. Во **множественном** наследовании родительские классы **независимы** друг от друга. В **многоуровневом** наследовании родительский класс, который также является классом-потомком (например, класс **Student**), может обращаться к другим родительским классам в цепочке наследования.

C++ поддерживает множественное наследование.

C# (и Java) **его не поддерживают**.

Взамен C# (и Java) **предоставляет нечто, называемое «интерфейсами»,**

Механизм интерфейсов отличается от наследования.

1.4. ~~Множественное наследование в C#~~ ? – *не* поддерживается

Множественное наследование **не поддерживается в C#** (и в Java). Поэтому, если необходимо наследовать класс от двух или более других классов, вам необходимо использовать многоуровневое наследование. При этом каждый класс должен проходить тест «**is a**». Классы, не проходящие этот тест («**is a**»), не должны использоваться в многоуровневом наследовании.

Вместо множественного наследования **C#** предоставляет **интерфейсы** (interface), которые в некоторой степени могут работать наподобие множественного наследования. Однако лучше считать интерфейсы чисто абстрактными классами. То есть интерфейс определяет ***только* названия: *1)*** методов, ***2)*** свойств, ***3)*** индексаторов ***4)*** и событий, но фактически не предоставляет никакого повторно используемого кода, который можно было бы использовать в наследовании. Если вы применяете интерфейс в **C#**-классе, вы должны создать весь код, необходимый для работы этих функций. По этой причине интерфейсы на самом деле **не относятся к наследованию (так как интерфейсы не наследуют реализации)**.

В действительности интерфейсы надо рассматривать как набор обязательств. Если класс *реализует* интерфейс (и поэтому класс содержит весь код, необходимый для его реализации), то другие объекты могут взаимодействовать или работать с этим классом через этот интерфейс.

2. Интерфейсы в C# – аналог множественного наследования

Слово "**интерфейс**" многозначное и в разных контекстах оно имеет различный смысл. В данной лекции речь идет о понятии интерфейса, соответствующем ключевому словом **interface**. В таком понимании интерфейс - это частный случай класса. Интерфейс представляет собой полностью абстрактный класс, все методы которого абстрактны. От абстрактного класса интерфейс отличается некоторыми деталями: ***1)*** в синтаксисе ***2)*** и в поведении. ***Синтаксическое отличие* состоит в том, что методы интерфейса объявляются без указания модификатора доступа**. ***Отличие в поведении* заключается в более жестких требованиях к потомкам.** Класс, наследующий интерфейс, обязан ***полностью* реализовать все методы интерфейса**. В этом - отличие от класса, наследующего абстрактный класс, где потомок может реализовать лишь некоторые методы родительского абстрактного класса, оставаясь абстрактным классом. Но, конечно, не ради этих отличий были введены интерфейсы в язык C#. У них значительно более важная роль.

Интерфейсы позволяют частично справиться с таким существенным недостатком языка, как отсутствие множественного наследования классов. Хотя реализация множественного наследования встречается с рядом проблем (раздел 1.2.3. и рис. 6), его отсутствие существенно снижает выразительную мощь языка. В языке C# полного множественного наследования классов нет. Чтобы частично сгладить этот пробел, допускается множественное наследование интерфейсов.

Обеспечить возможность классу иметь несколько родителей - один полноценный класс, а остальные в виде интерфейсов, - ***в этом и состоит основное назначение интерфейсов***.

Отметим одно важное назначение интерфейсов. Интерфейс позволяет описывать некоторые желательные свойства, которыми могут обладать объекты разных классов. В библиотеке **FCL** (**.NET Framework Class Library – библиотека базовых классов**) имеется большое число подобных стандартных интерфейсов (**см. раздел 2.4 на с. 29,сл.**). Например, все классы, допускающие сравнение своих объектов, обычно наследуют интерфейс **IComparable**, реализация которого позволяет сравнивать объекты не только на “равенство”, но и на "больше", "меньше".

Интерфейсом называется ссылочный тип .NET, который определяет открытые (**public**) элементы типа. Интерфейсы могут наследоваться; любой тип (**класс**), который наследует интерфейсу, **обязательно** должен реализовывать все элементы, определенные в этом интерфейсе. Интерфейс можно понимать также как контракт, согласно которому любой тип (**класс**), реализующий интерфейс, *обязуется* реализовать его элементы (иначе компилятор сообщит об ошибке).

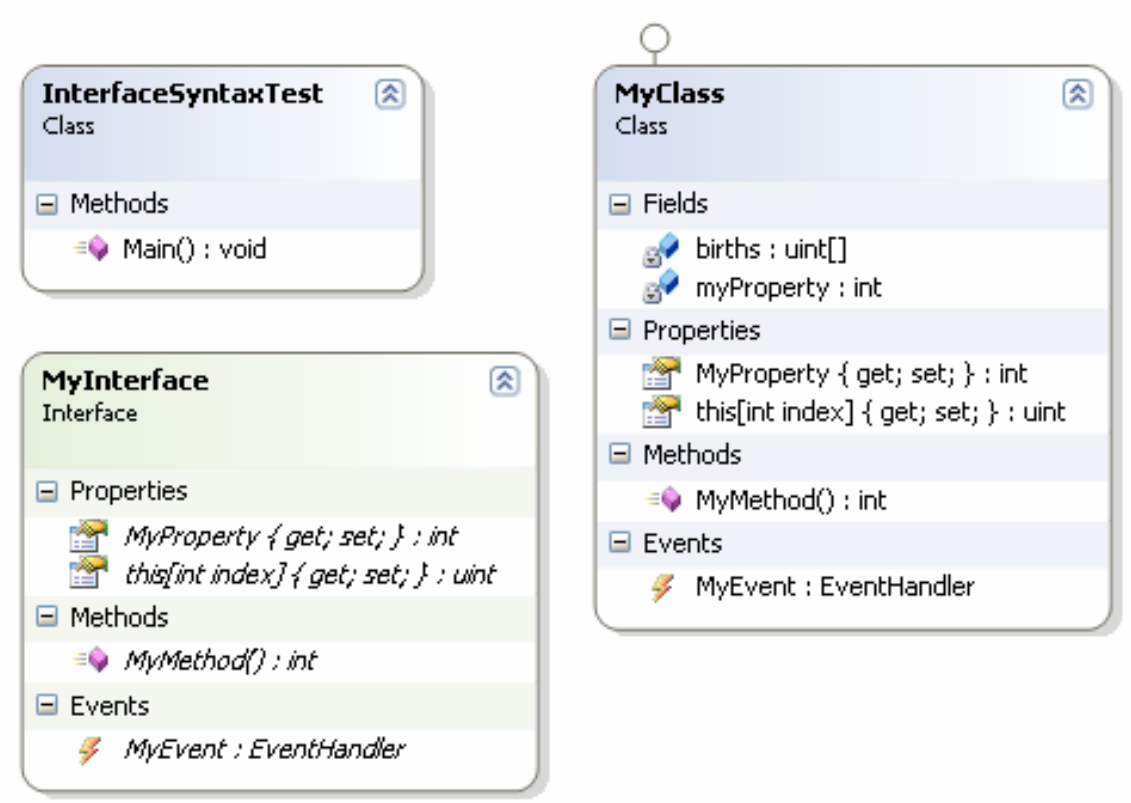
В **листинге 2** демонстрируется синтаксис реализации интерфейса.

Интерфейс **MyInterface (строки 6…23)** демонстрирует способ объявления каждого элемента типа. Обратите внимание на отсутствие полей — в интерфейсах они не приводятся, но по умолчанию в интерфейсах всё **public**. Класс **MyClass** реализует интерфейс **MyInterface**, как указано в объявлении наследования **(строка 25)**. В классах и структурах можно реализовывать несколько интерфейсов, разделяя их имена запятыми в объявлении наследования. Поскольку класс **MyClass** наследует интерфейс **MyInterface**, компилятор C# **заставляет реализовывать в нем каждый элемент** интерфейса **MyInterface (см. пункты 1), 2), 3) и 4) в** интерфейсе **MyInterface** и классе **MyClass)**.

```
1  using System;                                    // Синтаксис интерфейса
2                                                                          //Листинг 2
3  namespace ConsAppl_Mayo_c96_Interface
4  {
5
6      public interface MyInterface     // interface может содержать:
7      {
8          event EventHandler MyEvent;  // 1) событие см. строку 32
9
10         int MyMethod();                    // 2) метод; см. реализацию в строке 32
11
12         uint this[int index]               // 3) индексатор; см. реализацию в строке 37
13         {
14             get;
15             set;
16         }
17
18         int MyProperty                     // 4) свойство; см. реализацию в строке 49
19         {
20             get;
21             set;
22         }
23     }////////// конец interface MyInterface //////////
24
25     public class MyClass : MyInterface                // класс, РЕАЛИЗУЮЩИЙ интерфейс
26     {
27         private int myProperty;
28         private uint[ ] births = new uint[4];
29
30         public event EventHandler MyEvent; // 1) реализация события
31
32         public int MyMethod()                  // 2) реализация метода
```

```
33          {
34              return MyProperty * MyProperty;
35          }
36
37          public uint this[int index]                 // 3) реализация индексатора
38          {
39              get
40              {
41                  return births[index];
42              }
43              set
44              {
45                  births[index] = value;
46              }
47          }
48
49          public int MyProperty                       // 4) реализация свойства
50          {
51              get
52              {
53                  return myProperty;
54              }
55              set
56              {
57                  if (value >= 0) myProperty = value;
58              }
59          }
60      }                               /////////// конец class MyClass ///////////
61
62      class InterfaceSyntaxTest               // начальный класс
63      {
64          public static void Main()
65          {
66              uint sum = 0;
67
68              MyClass ob = new MyClass();             // работает конструктор по умолчанию
69              MyClass regRussia = new MyClass();      // работает конструктор по умолчанию
70
71              regRussia[0] = 10200; // блок set индексатора this[int index]  , см. строку 43
72              regRussia[1] = 18300;                   // -- « --
73              regRussia[2] = 21530;                   // -- « --
74              regRussia[3] = 973;                     // -- « --
75
76              for (int i = 0; i < 4; i++)
77              {
78                  sum += regRussia[i];        // блок get индексатора this[int index] , см. строку 39
79              }
```

```
80  Console.WriteLine("\nОбщее кол-во рождений по регионам России (блок get инд-ра) - " + sum);
81
82          ob.MyProperty = 10;                  // блок set свойства MyProperty, см. строку 55
83  Console.WriteLine("\n3начение  св-ва ob.MyProperty (блок get, 51) - " + ob.MyProperty);
84   Console.WriteLine("\nМетод - возведение во 2-ю степень - " + ob.MyMethod());   // стр. 32
85      Console.ReadLine();
86      }
87   }////////////// конец class InterfaceSyntaxTest ////////////
88  }
```



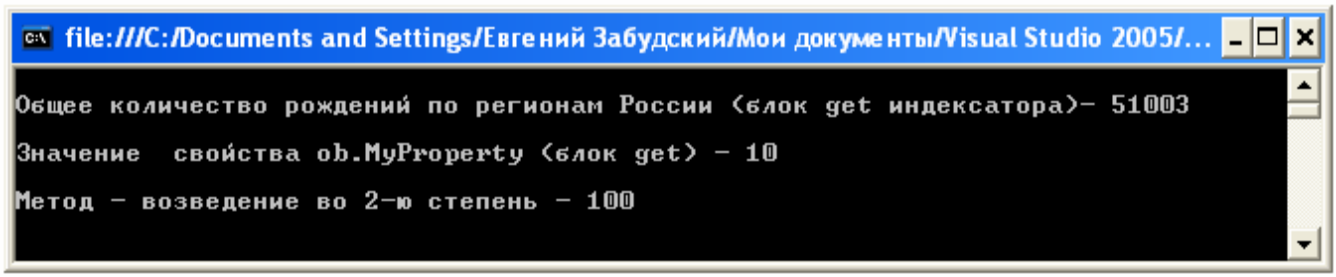


Рис. 9. Диаграмма классов и результаты работы программы на **листинге 2**

2.1. Необходимость интерфейса

Интерфейс является наиболее трудным понятием для начинающих программировать на языке C#. Чаще всего возникает вопрос: «Если я уверен, что все необходимые методы введены в класс, то зачем нужны интерфейсы?». Ответ заключается в том, что они обеспечивают простые средства безопасности типов при разработке программ, принимающих объекты, спецификацию типов которых вы не знаете. Единственное, что вы знаете об этих объектах, которые передаются в вашу программу, это что они имеют определенные элементы, которые необходимы вашей программе для работы с объектами.

Лучшим примером необходимости использования интерфейса может служить работа нескольких программистов над одним проектом. Интерфейсы помогают определить взаимодействие различных компонентов. Используя интерфейс, вы уменьшаете вероятность того, что программисты будут ошибочно трактовать элементы типов и неправильно вызывать другой тип, который определяет интерфейс. Без использования интерфейса в разрабатываемую программу вползают ошибки, которые проявляются только во время исполнения, когда их уже трудно диагностировать. При использовании интерфейсов ошибки в определении типов обнаруживаются сразу же во время компиляции, поэтому устранить их значительно проще.

Интерфейсы определяют открытые (**public**) элементы, которые экспонирует тип (**класс**). Они, по сути, являются контрактом, который гарантирует вызывающей программе, что любой тип (**класс**) наследующий интерфейсу, реализовал все элементы, перечисленные в интерфейсе. Интерфейсы широко используются в **.NET Framework BCL** (Библиотеке Базовых Классов), и позволяют программисту делать то, что без них было бы трудно или даже невозможно сделать. Например, интерфейсы обеспечивают итерации по коллекции оператором **foreach.** Интерфейс **IDisposable** обеспечивает конструкцию **Dispose,** которая является основным средством реализации детерминистского освобождения неуправляемых ресурсов в C# (**см. раздел 2.4 на с. 29,сл.**). Интерфейсы способствуют разработке хорошо спроектированных и надежных программ.

2.2. Преобразование к классу интерфейса

Создать объект класса интерфейса обычным путем с использованием конструктора и операции ***new*** нельзя. Тем не менее, можно объявить объект интерфейсного класса и связать его с реальным объектом путем приведения объекта наследника к классу интерфейса (см. далее **строки 39 и 40**). Это преобразование задается явно. Имея объект, можно вызывать методы интерфейса - даже если они закрыты в классе, для интерфейсных объектов они являются открытыми (**строки 43 и 46**).

При реализации двух интерфейсов с одинаковыми членами (например, с одинаковыми методами) полезно использовать **явную** реализацию. Если член интерфейса реализован явно, доступ к нему может быть получен только через значение типа интерфейса (см. **Листинг 3**).

Это дает классу двойную выгоду. *Во-первых*, класс может реализовать несколько интерфейсов, не беспокоясь о конфликтах имен (**см. строку 36, 37**). *Во-вторых*, класс может «спрятать» метод, если он не должен быть доступен всем пользователям класса. Чтобы получить доступ к методам, при реализации которых было явно указано имя интерфейса (**см. строки 19** и **24**), необходимо использовать явное приведение типов (**см. строки 39** и **40**):

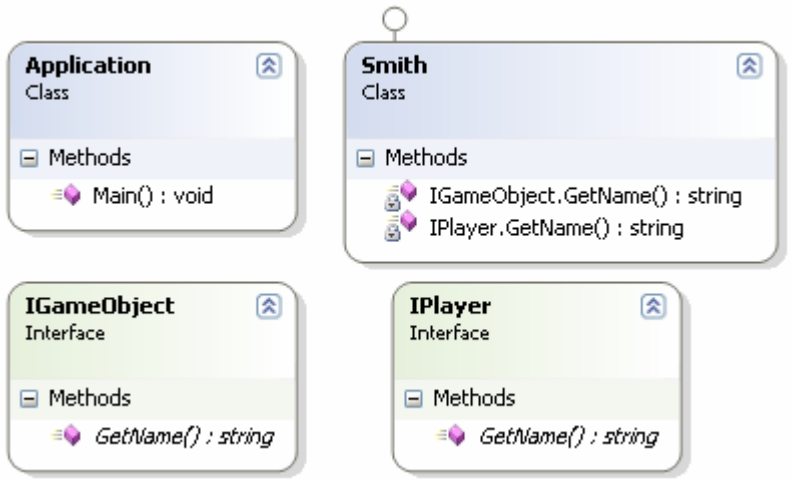
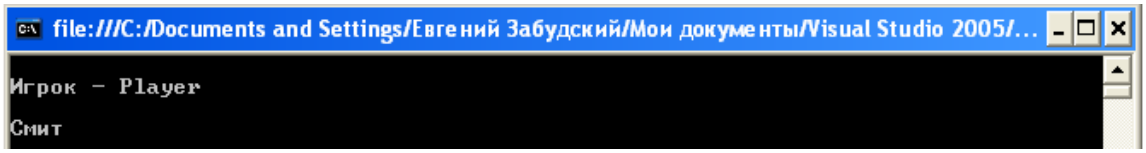


Рис. 10. Диаграмма классов программы на Листинге 3

```
1  using System;            // Интерфейсы                    Листинг 3
2  /* При реализации двух интерфейсов с одинаковыми членами полезно использовать
      явную реализацию.  Если член интерфейса реализован явно,
      доступ к нему может быть получен только через значение типа интерфейса*/
3
4  namespace ConsAppl_Inteface_dotSite
5  {
6     interface IGameObject
7     {
8        string GetName();
9     }                                  /////////////// end interface IGameObject ///////////////
10
11    interface IPlayer
12    {
13       string GetName();
14    }                                   ///////////////end interface Iplayer ///////////////
15
16    class Smith : IGameObject, IPlayer
17    {
18 string IGameObject.GetName()   // метод  GetName() интерфейса  IGameObject  реализован явно
19       {
20          return "Игрок - Player";
21       }
22
23       string IPlayer.GetName()    // метод  GetName() интерфейса  IPlayer  реализован явно
24       {
25          return "Смит";
26       }
27    }                                          ///////////////// end класс Smith ///////////////
28
29    class Application
30    {
31       public static void Main()
32       {
33          Smith smith = new Smith();              // объект класса  Smith
34
35    // Ошибка ; сделать эксперимент: раскомментировать и запустить на выполнение
36    // smith.GetName();
37                                                  //  в строках 38 и 40– явное приведение типов
38  IGameObject gosmith = (IGameObject)smith; // приведение к типу  IgameObject, строка 6
39  IPlayer psmith = (IPlayer)smith;              // приведение к типу   IPlayer ,        строка 11
40
41          Console.WriteLine("\n" + gosmith.GetName());        // Player – игрок, строка 18
42
43          Console.Write("\n" + psmith.GetName());             // Smith – Смит, строка 23
44          Console.ReadLine();
45       }
46    }////////////////////////////////
47 }
```



2.3. Отличия интерфейсов от наследования

Чем отличается **интерфейс** (**interface**) от **базового класса**?

В языке C# **интерфейс** (**interface**) служит для определения контракта.

Интерфейс (**interface**) в C# может содержать (**см. листинг 2 на с. 16…18 и рис.9**):

- методы,
- свойства,
- индексаторы,
- события.

Интерфейс (**interface**) является контрактом между базовым и производным классами и дает возможность установить некоторые *стандарты*.

Интерфейс (**interface**) не имеет переменных экземпляра, и его методы не содержат операторов действия (тело метода), а только определения методов (**сигнатуры**), точно так же как абстрактные методы в абстрактных классах.

Чем интерфейс отличается от абстрактного класса?

В абстрактном классе, хотя это и не обязательно, могут быть методы, содержащие операторы действия (тело метода), тогда как в интерфейсе это недопустимо. В **интерфейсе** (**interface**) не должно быть вообще *никакой реализации*.

В чем преимущество создания **интерфейса** (**interface**) вместо абстрактного класса?

Главное достоинство интерфейса, заключается в том, что **один класс может** ***реализовать более одного интерфейса***. Это позволяет осуществлять множество возможностей и в объектно-ориентированных программах называется *множественным наследованием*.

Пример – Можно создать **интерфейс** (**interface**) с именем **Employee**, описывающий служащего, и другой **интерфейс** (**interface**) с именем **Pilot**, описывающий людей, умеющих управлять самолетом. Если необходимо создать класс для служащих компании, умеющих управлять самолетом, то можно ***реализовать*** оба интерфейса в одном классе. Однако не следует забывать, что интерфейс **не содержит тела** методов, а **содержит только сигнатуры** методов.

Нельзя ли сделать то же самое и с классами, то есть создать класс, производный от двух базовых классов?

Нет. В ОО языке программирования C# любой класс может наследовать **только от одного** базового класса.

Каково назначение интерфейса? Разве это не просто форма наследования?

Интерфейсы ***не позволяют наследовать*** программный код. В связи с тем, что в методы интерфейса не вводится вообще никакого кода реализации (**и, следовательно, нечего наследовать**), то *методы интерфейса нельзя вызывать ни в каком классе*. Для программиста интерфейсы служат только указанием, какие методы **необходимо** ***реализовать*** в классе, использующем интерфейс (если это не сделать, то **компилятор выдаст сообщение об ошибке**). Но эти методы необходимо программировать полностью с нуля.

Зачем определять интерфейс, если это не дает никакой экономии времени и усилий?

Причина заключается в стандартизации. **Например** – в больших корпоративных средах программисту-новичку могут дать указание, чтобы он при доступе к корпоративной базе данных на **Oracle** всегда использовал **интерфейс ISecurity** (безопасность), а значит, всегда использовал

три метода: **determineUserID()** (определить пользователя), **obtainPassword()** (получить пароль) и **writeAuditRecord()** (сделать запись в журнал). Использование **интерфейса** заставит программиста реализовать код для этих методов.

Почему нельзя ввести эти обязательные методы в базовый класс и обязать всех программистов наследовать свои классы от него**?**

Тут возможны варианты. Одной из причин, по которой не стоит так поступать, может быть секретность доступа к конкретным базам и таблицам базы данных **Oracle**, в этом случае только программист, разрабатывающий программу доступа, может знать детали. Требование к разрабатываемым классам, осуществляющим доступ к базе данных **Oracle**, выражающееся в необходимости всегда определять эти три метода, гарантирует, что программист не забудет написать программный код для реализации этих функций. Таким образом, **интерфейс** (**interface**) устанавливает **стандарты**.

Создадим интерфейс **ISecurity**, о котором шла речь выше. Напомню, что в интерфейс исполняемый код не вводится, там могут быть **только сигнатуры** метода.

```
   interface ISecurity                    // объявление интерфейса ISecurity
{
   void determineUserID();                // объявление метода
   void obtainPassword();                 // объявление метода
   void writeAuditRecord();               // объявление метода
}
```

Можно сохранить интерфейс **ISecurity** в файле под именем **ISecurity.cs** и откомпилировать его, как любой другой класс. Если не считать выдачи сообщения об отсутствии точки входа (то есть метода **Main()**):

«ConsoleApplication8.exe does not contain a static 'Main' method suitable for an entry point ConsoleApplication8»

«ConsoleApplication8.exe не содержит статический метод Main обеспечивающий точку входа ConsoleApplication8» ,

то компиляция проходит нормально.

Как видно объявление интерфейса похоже на объявление класса, но отсутствует слово **class**. Вместо него объявление начинается со слова **interface** строчными буквами, за которым указывается имя интерфейса. Принято (но не обязательно) начинать имена интерфейсов с прописной буквы **I (ISecurity)**. После чего идут объявления свойств и методов. Обратите внимание, методы не содержат фигурных скобок, просто объявление метода и точка с запятой “ **;** ”, также методы не содержат никакого кода реализации (то есть не содержат тела метода).

Объявляемые в интерфейсе методы считаются абстрактными по умолчанию, следовательно, нет причин явно указывать ключевое слово **abstract**. Более того, *если это сделать, то компилятор выдаст сообщение об ошибке*.

Проверим работу интерфейса **ISecurity** внутри класса **HourlyEmployee**. Это учебный пример (он не является образцом построения интерфейса) (см. далее **Листинг 4**).

Если мы применим интерфейс **ISecurity** в классе **HourlyEmployee**, что он (интерфейс) потребует от нас**?**

Он **потребует** от нас осуществить методы **determineUserID(), obtainPassword()** и **writeAuditRecord()**, объявленные в интерфейсе **ISecurity,** во всех тех классах, которые реализуют этот интерфейс. Однако какие именно действия мы заложим в эти методы (то есть, какое составим тело метода), зависит полностью от нас.

Если определен программный код методов базового класса, его **можно** использовать в производном классе, но для интерфейса это не так. Когда мы реализуем в классе интерфейс, мы **вынуждены** осуществить его методы, но как мы это будем делать, зависит исключительно от нас. Примере, в качестве теста мы определим в каждом из этих трех методов просто вывод на консоль имени вызванного метода (сигнатуры этих методов отражены в интерфейсе **ISecurity – см. ниже строки 7…12**).

Как описать интерфейс в классе**?**

Указать, что класс ***реализует*** интерфейс, можно аналогично указанию ***наследования*** от базового класса. После имени класса ставится двоеточие, после него указываем имя интерфейса. **Очень важно не забывать**, что если один класс наследует от другого, при обращении к объектам класса всегда **сначала** указывается имя базового класса, **потом** имя интерфейса или другое имя (**см. ниже строку 58**).

Пример, представленный на **листинге 4**, демонстрирует, как можно в производном классе наследовать от базового класса и при этом реализовать один или несколько интерфейсов. Просто необходимо указать все нужные интерфейсы после указания базового класса **(строка 58**):

```
58  class HourlyEmployee : EmployeeBase, ISecurity
59    {
60    ……
```

В одном классе можно реализовать несколько интерфейсов. Объявляя несколько интерфейсов в классе, необходимо отделять каждый интерфейс запятой. Но наследовать можно только один класс. Для производного класса нельзя указывать более одного базового,

```
1   using System;                                          // Демонстрация interface в C#
2   using System.Windows.Forms;                                                  // Листинг 4
3
4   namespace ConsAppl_Smil_347_interface
5   {
6       ////////////////////// начало интерфейса Isecurity ////////////////////////
7     interface ISecurity                  // объявление интерфейса ISecurity
8       {                                  // реализация  интерфейса – см. строки 113…124
9           void determineUserID();                  // объявление метода интерфейса
10          void obtainPassword();                   // -- " -- " --
11          void writeAuditRecord();                 // -- " -- " --
12      }////////////////////// конец интерфейса Isecurity ////////////////////////
13
14    abstract class EmployeeBase         // абстрактный базовый класс EmployeeBase
15    {
16        protected string empID;     // защищенные перем-ые, доступ из произв-х кл-сов
17        protected string name;
18        protected float grossPay;
19
20        public abstract void Display();      // абстрактный метод Display
21
22          public abstract float GrossPay   // абстрактное свойство GrossPay
23        {
24          get;
```

```
25          }
26
27          public EmployeeBase()               // метод конструктора
28          {
29              Console.WriteLine("\nКонструктор класса EmployeeBase' s");
30          }
31
32          public string EmpID                 // свойство EmpID
33          {
34              get
35              {
36                  return empID;
37              }
38              set
39              {
40                  empID = value;
41              }
42          }
43
44          public string Name                  // свойство Name
45          {
46              get
47              {
48                  return name;
49              }
50              set
51              {
52                  name = value;
53              }
54          }
55      }///////////////// конец класса EmployeeBase ////////////////////////////////
56
57      /* Производный класс HourlyEmployee наследует базовый класс EmployeeBase и
                                    реализует интерфейс ISecurity */
58  class HourlyEmployee : EmployeeBase, Isecurity
59      {
60      private float hourlyRate;   // закрытые перем-ные, доступ извне посредством свойств
61          private int hoursWorked;
62
63          public HourlyEmployee()                 // метод конструктора
64          {
65              Console.WriteLine("Конструктор класса HourlyEmployee's\n");
66          }
67
68          public float HourlyRate                 // свойство HourlyRate
69          {
70              get
```

```
71          {
72              return hourlyRate;
73          }
74          set
75          {
76              hourlyRate = value;
77          }
78      }
79
80      public int HoursWorked              // свойство HoursWorked
81      {
82          get
83          {
84              return hoursWorked;
85          }
86          set
87          {
88              hoursWorked = value;
89          }
90      }
91
92      override public float GrossPay  // переопределяющее свойство GrossPay, строка 22
93      {
94          get
95          {
96              grossPay = hourlyRate * hoursWorked;  // строки 22, 68, 80
97              return grossPay;
98          }
99      }
100
101     override public void Display()          // переопределяющий метод Display, стр. 20
102     {
103 MessageBox.Show("*** ЗАПИСЬ ДЛЯ СЛУЖАЩЕГО С ПОЧАСОВОЙ ОПЛАТОЙ ***\n\n" +
104         "Идентификатор служащего:  " + empID + "\n" +              // строка 36
105         "Имя: " + name + "\n" +                                     // строка 44
106         "Ставка часа: $ " + hourlyRate + "\n" +                     // строка 68
107         "Рабочие часы: " + hoursWorked + "\n" +                     // строка 80
108         "Итоговая оплата: $ " + GrossPay, "Информация Служащего",  // строка 92
109          MessageBoxButtons.OK,
110         MessageBoxIcon.Information);
111     }
112
113     public void determineUserID()        //три метода интерфейса ISecurity , строка 7…12
114     {
115         Console.WriteLine("Метод determineUserID  интерфейса ISecurity");
116     }
117     public void obtainPassword()
```

```
118         {
119             Console.WriteLine("Метод obtainPassword   интерфейса ISecurity");
120         }
121         public void writeAuditRecord()
122         {
123             Console.WriteLine("Метод writeAuditRecord интерфейса ISecurity");
124         }
125     } ///////////////// конец класса HourlyEmployee ////////////////////////////////
126
127     class SalariedEmployee : EmployeeBase  // производный класс SalariedEmployee
128     {
129   private float annualSalary;  // закрытая перем-ная, доступ извне посредством свойства
130
131         public SalariedEmployee()                   // метод конструктора
132         {
133             Console.WriteLine("Конструктор класса SalariedEmployee's");
134         }
135
136         public float AnnualSalary         // свойство AnnualSalary
137         {
138             get
139             {
140                 return annualSalary;
141             }
142             set
143             {
144                 annualSalary = value;
145             }
146         }
147
148       override public float GrossPay  // переопределяющее свойство GrossPay, строка 22
149         {
150             get
151             {
152                 grossPay = annualSalary / 26;                    //  строка 136
153                 return grossPay;
154             }
155         }
156
157         override public void Display()      // переопределяющий метод Display, строка 20
158         {
159             MessageBox.Show("*** ЗАПИСЬ ДЛЯ ШТАТНОГО СЛУЖАЩЕГО ***\n\n" +
160             "Идентификатор служащего:  " + empID + "\n" +                    // строка 36
161             "Имя: " + name + "\n" +                                          // строка 44
162             "Годовое Жалованье: $ " + annualSalary + "\n" +                  // строка 136
163             "Итоговая оплата: $ " + GrossPay, "Информация Служащего",    // строка 92
164             MessageBoxButtons.OK,
```

```
165             MessageBoxIcon.Information);
166         }
167     }                   ////////////////// конец класса SalariedEmployee ////////////////////////
168
169     public class InterfaceTest              // начальный класс InterfaceTest
170     {
171         public static void Main(string[] args)
172         {
173             HourlyEmployee hourlyEmployee = new HourlyEmployee();     // см. строки 63 и 29
174             hourlyEmployee.EmpID = "086";           // обращение к свойству, строка 32
175             hourlyEmployee.Name = "Иван Петров";     //-- " -- " --, строка 44
176             hourlyEmployee.HourlyRate = 12.50F;          //-- " -- " --, строка 68
177             hourlyEmployee.HoursWorked = 40;             //-- " -- " --, строка 80
178             hourlyEmployee.Display();     // см. строки 101 и 20
179         hourlyEmployee.determineUserID() //обращение к м-ду интерфейса ISecurity, строка 113
180             hourlyEmployee.obtainPassword();             //-- " -- " --, строка 117
181             hourlyEmployee.writeAuditRecord();           //-- " -- " --, строка 121
182             SalariedEmployee salariedEmployee = new SalariedEmployee();// см. строки 131и 29
183             salariedEmployee.EmpID = "337";          // обращение к свойству, строка 32
184             salariedEmployee.Name = "Мария Лескова";     //-- " -- " --, строка 44
185             salariedEmployee.AnnualSalary = 52000;       //-- " -- " ---, строка 136
186             salariedEmployee.Display(); // см. строки 157 и 20
187
188             Console.ReadLine();
189         }
190     }                   ////////////////// конец класса InterfaceTest ////////////////////////
191 }
```

В выводе программы, мы наблюдаем два окна сообщений **“Информация для служащего”**, в которых выводятся значения начислений для получающих почасовую оплату и для штатных служащих. Дополнительно на консоль выводятся следующие сообщения трассировки **(рис. 12 и рис. 13).**

Как видите когда был создан экземпляр класса **HourlyEmployee** (**строка 173**), то были вызваны оба конструктора (**рис. 11**) — **EmployeeBase** (**строка 27**) и **HourlyEmployee** (**строка 63**). Так и должно быть потому, что класс **HourlyEmployee** наследует от класса **EmployeeBase**:

```
173             HourlyEmployee hourlyEmployee = new HourlyEmployee();
```

В классе **HourlyEmployee** реализован интерфейс **ISecurity** (**строка 58**), поэтому в методе **Main** можно вызвать методы **determineUserID()**, **obtainPassword()** и **writeAuditRecord()** (**строки 179…181**) класса **HourlyEmployee** (**строки 113…124**):

```
179             hourlyEmployee.determineUserID()     //обращение к методу интерфейса ISecurity
180             hourlyEmployee.obtainPassword();             //-- " -- " --
181             hourlyEmployee.writeAuditRecord();           //-- " -- " --
```

Сообщения на консоль выводились в результате вызова этих методов. Если бы программа была не учебной, то реализация интерфейса **ISecurity** потребовала бы написания намного большего объема кода. Рассмотренная программа предназначена только для демонстрационных целей.

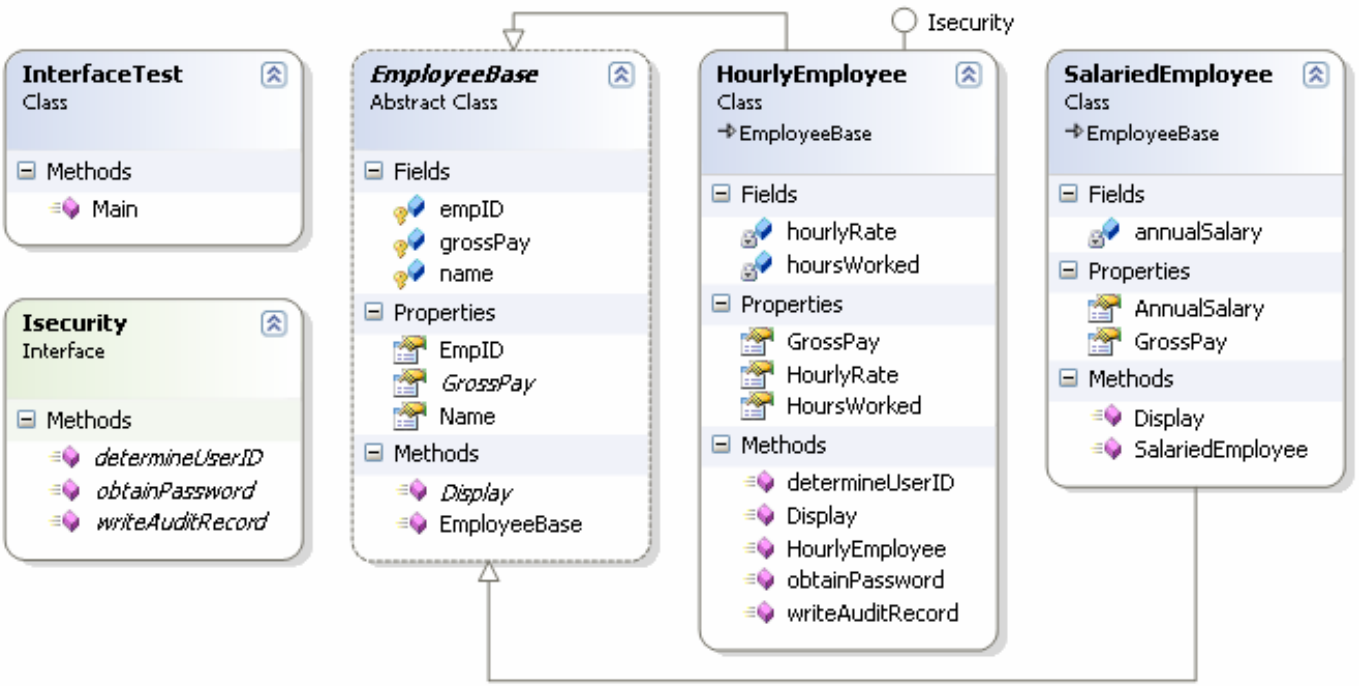
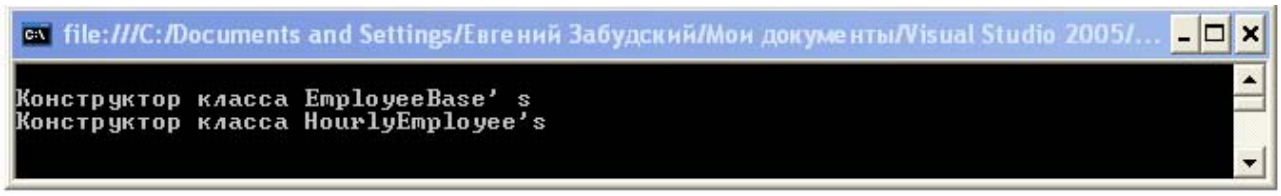


Рис. 11. Диаграмма классов программы на **листинге 4**



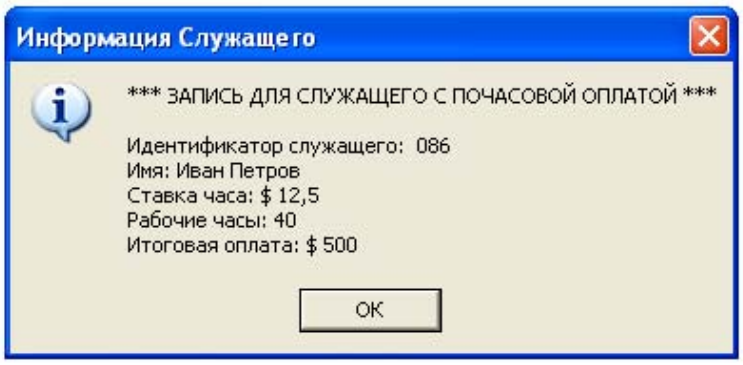


Рис. 12. Первый вывод программы на **листинге 4**

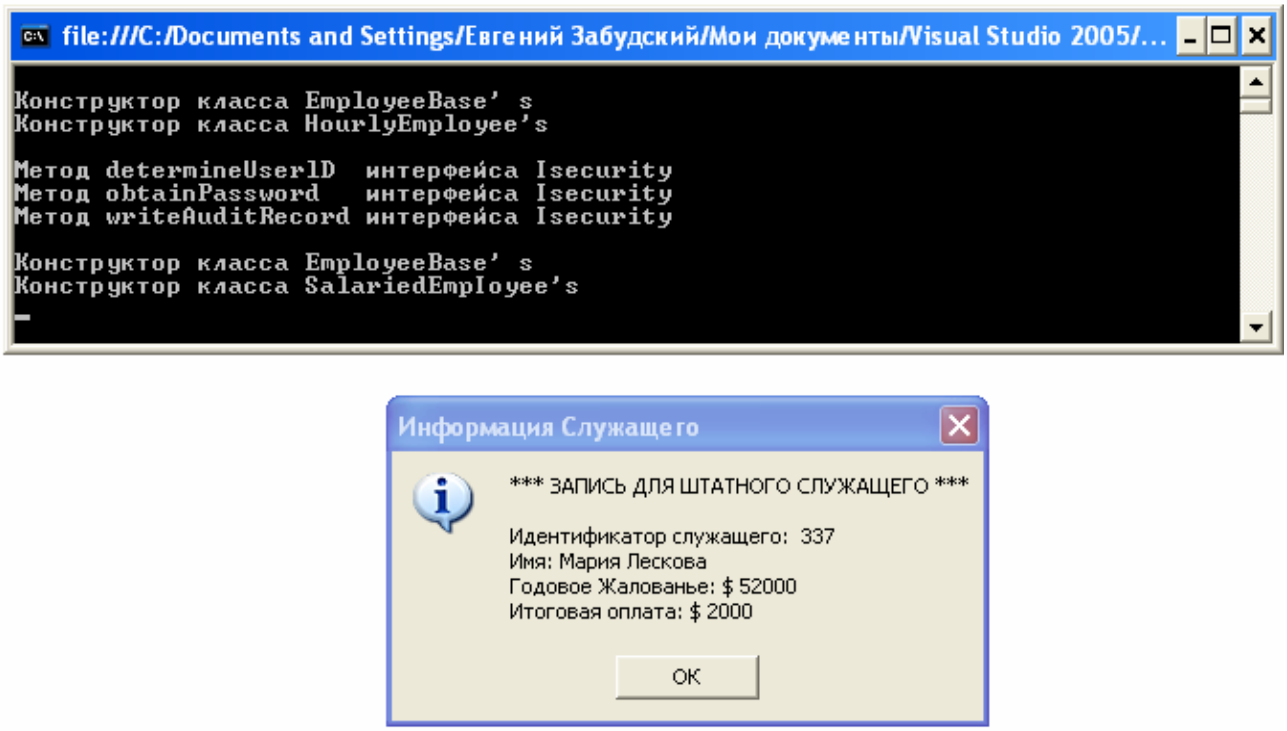


Рис. 13. Второй вывод программы на **листинге 4**

2.4. Стандартные интерфейсы C#

Чтобы разобраться в интерфейсах, полезно взглянуть на то, что создают другие разработчики. В этом разделе перечислены некоторые основные интерфейсы **C#, включенные в .NET FCL.** Этот список интерфейсов не является полным, но он покажет образцы того, как архитекторы языка **C#** разрабатывали библиотеки классов и интерфейсов для их систем.

Далее перечислены наиболее часто используемые интерфейсы **C#**:

ICloneable. Реализуется классами, которые поддерживают клонирование.

IComponent. Если класс является компонентом, он должен реализовывать этот интерфейс. Обратите внимание, что класс **Component** реализует этот интерфейс.

IContainer. Этот интерфейс реализуют классы, которые содержат компоненты. Например, класс **Form** наследуется от класса, который реализует этот интерфейс.

INumerator. Классы, обеспечивающие итерацию и перечисление, реализуют этот интерфейс.

ISerializable. Классы, которые могут сохранять свои данные в поток и загружать их из потока, реализуют этот интерфейс

. **Примечание:**

По соглашению идентификатор интерфейса должен начинаться с прописной буквы **I** (от **Interface**), а поскольку интерфейс разрешает (**enable**) классу определенные действия, заставляя его реализовать свои абстрактные методы, идентификатор интерфейса часто заканчивается суффиксом **-able** (он обозначает способность к действию): **IComparable, ICloneable, IStorable** и т. д. Ни один из элементов интерфейса (абстрактные методы, свойства, индексаторы и события) не может иметь спецификаторов доступности. Все они неявно объявлены как **public**, поскольку должны быть доступны извне класса. Свойства и индексаторы, определенные в интерфейсе, могут иметь **get**- или **set**-аксессор. Аксессоры объявлены **abstract** неявно.

2.4.1. Иерархия интерфейсов в пространстве имен System.Collections

В пространстве имен **System.Collections** определено множество интерфейсов (**рис. 14**). Но мы рассмотрим интерфейсы коллекций **ICollection**, поскольку они определяют функции, общие для всех классов коллекций.

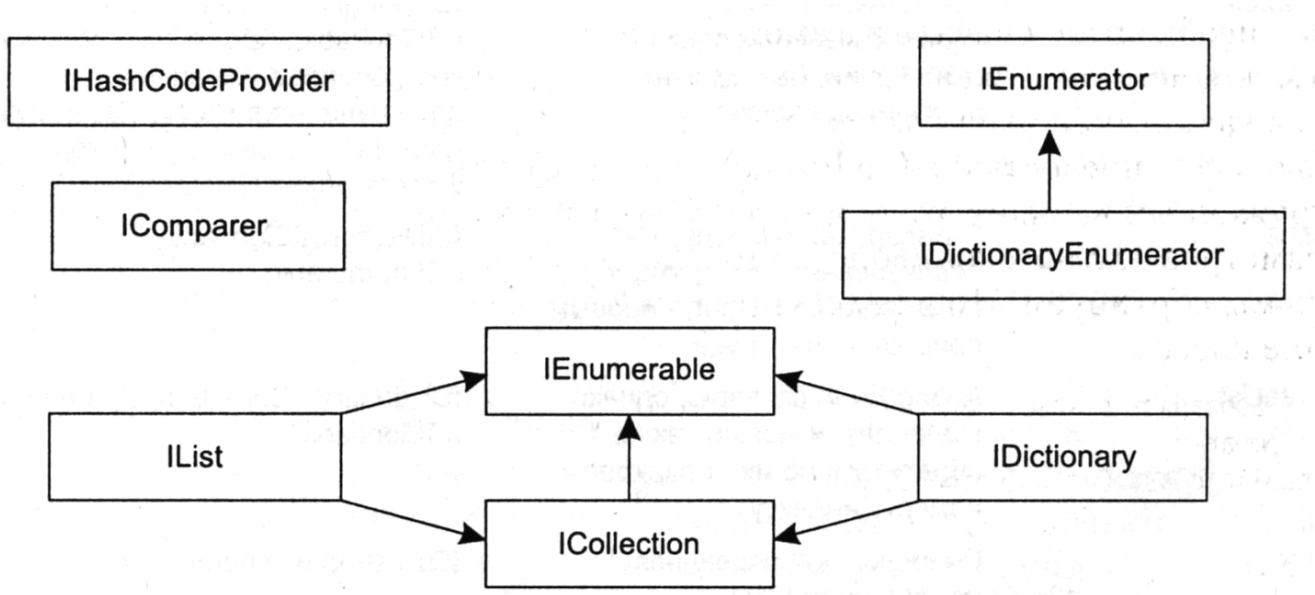


Рис. 14. Иерархия интерфейсов **System.Collections**

Классы, используемые для работы с коллекциями данных, реализуют **ICollection**. Многие классы уже содержат реализацию этого интерфейса (см. ниже **Листинг 5**).

Интерфейсы представляют собой универсальное поведение, например «реализация коллекции». Интерфейсы составляют ключевую часть работы компонентов **C#**, они имеют **предопределенное** поведение, которое новые классы **должны реализовать**, чтобы их можно было считать компонентами.

2.4.2. Анализ кода листинга 5

A. Пространство имен System.Collections

В **C#** под коллекцией понимается группа объектов (например, массив). Пространство имен **System.Collections** (**см. 2-ю строку кода**)

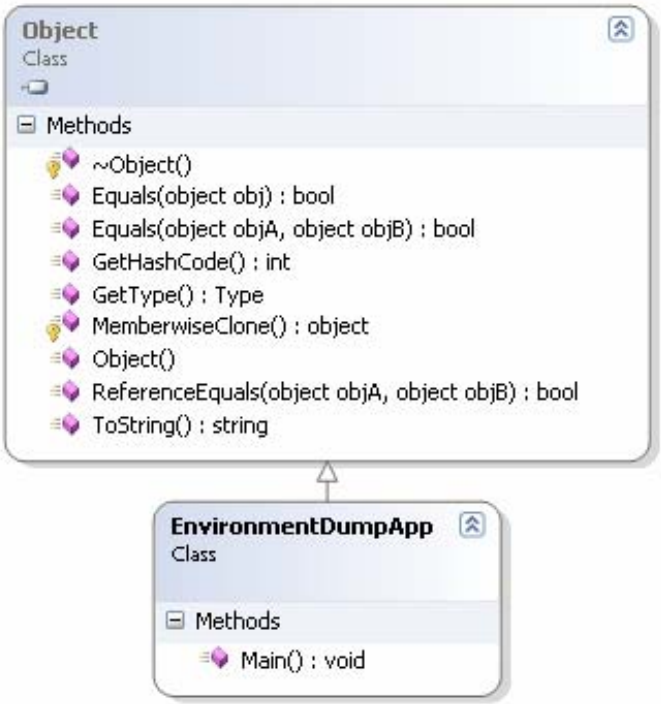
```
2  using System.Collections;
```

содержит множество интерфейсов и классов, которые определяют и реализуют коллекции различных типов. Коллекции упрощают программирование, предлагая уже готовые решения для построения структур данных, разработка которых "с нуля" отличается большой трудоемкостью. Речь идет о встроенных коллекциях, которые поддерживают, например, функционирование стеков, очередей и хеш-таблиц. Коллекции пользуются большой популярностью у всех **C#**-программистов.

```
1  using System;                          // Стандартный интерфейс IDictionary
2  using System.Collections;
3                                                                  // Листинг 5
4  class EnvironmentDumpApp
5  {
```

```
6     public static void Main()
7     {
8         IDictionary envvars = Environment.GetEnvironmentVariables();
9                 /*Получение полезной информации об операционной системе,
                          в которой будет работать ваше приложение*/
10        Console.WriteLine("\nОпределено {0} переменных окружения.", envvars.Keys.Count);
11
12        foreach (String strKey in envvars.Keys)
13        {
14            Console.WriteLine("\n{0} = {1}", strKey, envvars[strKey].ToString());
15        }
16                                  // А какая версия платформы .NET у нас используется?
17    Console.WriteLine("\nУстановленная версия платформы .NET {0}", Environment.Version);
18        Console.ReadLine();
19    }
20 }
```

Основное достоинство коллекций состоит в том, что они стандартизируют способ обработки групп объектов в прикладных программах (z.B., массивов). Все коллекции разработаны на основе набора четко определенных интерфейсов.



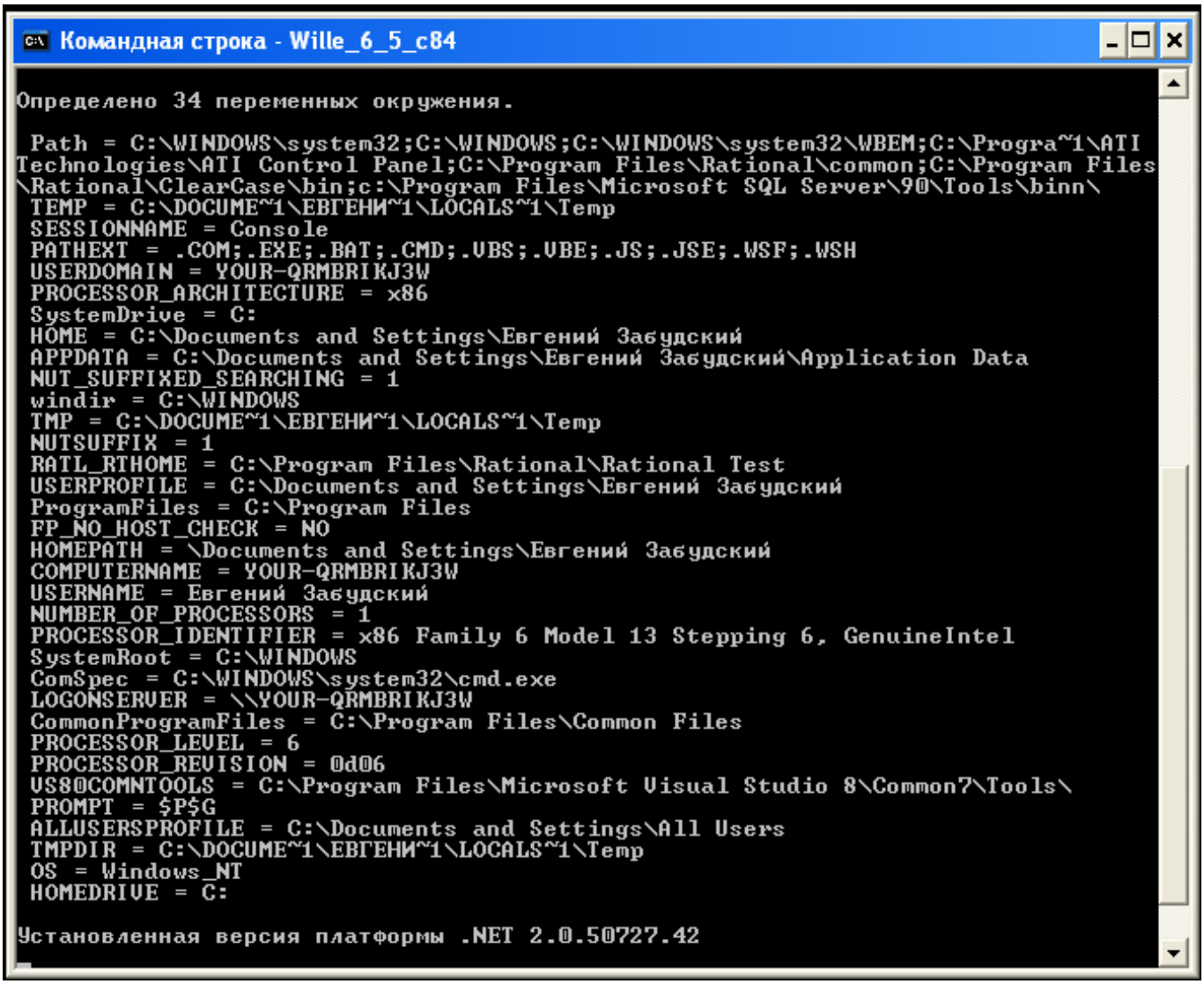


Рис. 15. Диаграмма классов и вывод программы на **Листинге 5**

B. Интерфейс ICollection

Интерфейс **ICollection** (**рис. 14**) можно назвать фундаментом, на котором построены все коллекции. В нем объявлены основные методы и свойства, без которых не может обойтись ни одна коллекция. Он наследует интерфейс **IEnumerable**. Не зная сути интерфейса **ICollection**, невозможно понять механизм действия коллекции.

В интерфейсе **ICollection** определен четыре свойства, в том числе самое востребованное свойство **Count** **(см. выше строку 10)**

10	Console.WriteLine("\nОпределено {0} переменных окружения.", envvars.Keys.Count);

```
int Count
 {
 get;
  }   ,
```

так как оно содержит количество элементов, хранимых в коллекции в данный момент. Если свойство **Count** равно нулю, значит, коллекция пуста.

C. Интерфейс IDictionary

```
8   IDictionary envvars = Environment.GetEnvironmentVariables();
```

Интерфейс **IDictionary** (рис. **14**) определяет поведение коллекции, которая устанавливает соответствие между уникальными ключами и значениями. **Ключ** — это объект, который используется для получения соответствующего ему значения. Следовательно, коллекция, которая реализует интерфейс **IDictionary**, служит для хранения пар ключ/значение. Сохраненную однажды пару можно затем извлечь по заданному ключу. Интерфейс **IDictionary** наследует интерфейс **ICollection**. В интерфейсе **IDictionary имеется шесть** методов и четыре свойства, в том числе свойство **Keys (см. выше строку 10)**

```
ICollection Keys
{
get;
}  .
```

С помощью свойств **Keys** ключи, хранимые в словарной коллекции, можно получить в виде отдельного списка.

D. Класс Environment

Environment – это еще один из множества классов (**строка 8**), определенных в пространстве имен **System**. В этом классе определено несколько статических типов, которые можно использовать для получения полезной информации об операционной системе, в которой будет работать ваше приложение, а также о самой среде выполнения **.NET (строка 17)**.

Вызов метода **GetEnvironmentVariables** (**строка 8**) класса **Environment** возвращает интерфейс типа **IDictionary**, реализованный во многих классах структуры **.NET**. При помощи интерфейса **IDictionary** доступны две структуры: **Keys** и **Values**. В этом примере оператор **foreach** использует коллекцию ключей **Keys**, а затем выполняется поиск значения на основе текущего значения ключа (**строка 12**).

E. Цикл foreach

```
12      foreach (String strKey in envvars.Keys)
```

Цикл **foreach** предназначен для опроса элементов коллекций. **Коллекция** — это группа объектов. В **C#** определено несколько типов коллекций, среди которых можно выделить массив.

В заголовке оператора цикла (**строка 12**) элементы **String strKey** задают тип **String** и имя **strKey** итерационной переменной, которая при функционировании цикла **foreach** будет получать значения элементов из коллекции. Элемент **envvars.Keys** служит для указания опрашиваемой коллекции (в данном случае в качестве коллекции используется массив ключей **Keys**). Элемент тип (**String)** должен совпадать (или быть совместимым) с базовым типом массива. Итерационную переменную **strKey** применительно к массиву можно использовать только для чтения. Следовательно, невозможно изменить содержимое массива, присвоив итерационной переменной **strKey** новое значение.

Цикл **foreach** последовательно опрашивает элементы массива в направлении от наименьшего индекса к наибольшему.

Несмотря на то что цикл **foreach** работает до тех пор, пока не будут опрошены все элементы массива, существует возможность досрочного его останова с помощью инструкции **break**.

F. Класс Object

В C# определен специальный класс с именем **Object**, который **является неявным базовым классом всех других классов** и типов (включая типы значений). Другими словами, все остальные типы выводятся из класса **Object**. Это означает, что ссылочная переменная типа **object** может указывать на объ-

ект любого типа. Кроме того, поскольку C#-массивы реализованы как классы, переменная типа **object** также может ссылаться на любой массив. Строго говоря, C#-имя **object** — еще одно имя для класса **System.Object**, который является частью библиотеки классов **.NET Framework.**

Класс **Object** определяет девять методов (см. диаграмму классов на рис. **15**). Эти методы доступны для каждого объекта. Один из них метод **ToString()** (**см. строку 14**)

public virtual string ToString() – возвращает строку, которая описывает объект

```
14    Console.WriteLine("\n{0} = {1}", strKey, envvars[strKey].ToString());
```

Метод **ToString()** возвращает строку, содержащую описание объекта, для которого вызывается этот метод. Кроме того, метод **ToString()** автоматически вызывается при выводе объекта с помощью метода **WriteLine()**. Метод **ToString()** переопределяется во многих классах, что позволяет подобрать описание специально для типов объектов, которые они создают. В примере этот метод представляет информацию о внутреннем состоянии **операционной системы,** в которой будет работать ваше приложение (в формате имя-значение).

Еще раз определение: Интерфейс (interface) – это конструкция **C#**, определяющая подобно классам методы, но **не представляющая никакой реализации** этих методов. Класс может реализовать **интерфейс** (**interface**) путем реализации каждого метода в своем (класса) интерфейсе.

П.1. Абстрактные классы и методы

Каждый класс, содержащий абстрактный метод, должен быть объявлен абстрактным. **Следствием объявления класса абстрактным является то, что от такого класса нельзя напрямую создавать объекты.**

Класс, содержащий абстрактный метод, автоматически становится абстрактным классом, а это означает, что он является просто *шаблоном* для создания производных классов и ничего более.

Существуют **классы, содержащие *только* абстрактные методы**. Такие классы образуют так называемый **интерфейс** (**interface**).

Предположим, нужно определить несколько подобных классов-фигур: **Rectangle, Square, Ellipse, Triangle** и т. д. У этих классов может быть два базовых метода: **area()** и **circumference()**. Теперь для упрощения работы с массивами фигур полезно предусмотреть, чтобы все фигуры имели общего родителя - класс **Shape**. Если выстроить иерархию классов подобным образом, то любая фигура, вне зависимости от того, какую конкретно форму она представляет, может быть присвоена простой переменной, или элементу массива типа **Shape**. Желательно, чтобы класс **Shape** инкапсулировал особенности, присущие всем нашим фигурам, например методы **area()** и **circumference()**. Но общий класс **Shape** в действительности не представляет ни одной реальной фигуры, поэтому он не может определять полезных реальных реализаций методов. C# обрабатывает подобные ситуации с помощью абстрактных методов.

C# позволяет определить метод без реализации, объявив его с модификатором **abstract**. У абстрактного метода нет тела; у него есть только заголовок, заканчивающийся точкой с запятой.

Вот правила для абстрактных методов и абстрактных классов, которые их содержат:

A. Любой класс с абстрактным методом автоматически становится абстрактным и должен быть объявлен как **abstract**.

B. Нельзя создавать ни одного экземпляра абстрактного класса.

C. Экземпляры *под*классов абстрактного класса, можно создавать только в том случае, если все методы, объявленные как **abstract**, замещены и реализованы (то есть имеют тело). Такие классы часто называют реальными, чтобы подчеркнуть тот факт, что они не абстрактные.

D. Если *под*класс абстрактного класса, не реализует всех методов, объявленных как **abstract**, то этот подкласс сам является абстрактным.

E. Методы, объявленные как **static, private** и **sealed**, не могут быть объявлены как **abstract**, поскольку такие методы не могут быть замещены *под*классами. Точно так же класс, объявленный как **sealed**, не может содержать методов, объявленных как **abstract**.

F. Класс может быть объявлен как **abstract**, даже если он не содержит ни одного абстрактного метода. Такое объявление означает, что реализация класса все еще не закончена и класс будет служить родителем для одного или нескольких подклассов, которые завершат реализацию. Экземпляр такого класса не может быть создан.

Есть одна важная особенность этих правил. Если мы определим класс **Shape** с методами **area() и circumference()**, объявленными как **abstract**, то любой подкласс должен будет реализовать данные методы с тем, чтобы можно было создавать экземпляры этого класса. Другими словами, у каждого объекта **Shape** обязательно есть реализации этих методов. В **листинге 6** показано, как это работает. В нем объявлен абстрактный класс **Shape** с двумя реальными подклассами.

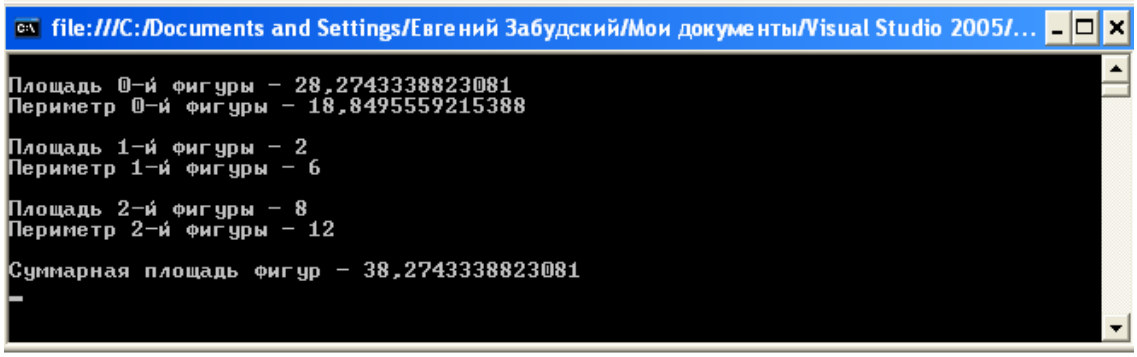
В конце каждого абстрактного метода класса **Shape** после скобок стоит точка с запятой (см. строки **7** и **8**). У таких методов (**абстрактных**) нет фигурных скобок и не определено тело метода.

Здесь есть два важных замечания:

- Подклассы **Shape** могут быть присвоены элементам массива типа **Shape** **(см. строки 54 и 56…58)**. Приведение типа не требуется.
- Вы можете вызывать методы **area()** и **circumference()** для любых объектов **Shape**, несмотря на то, что сам класс **Shape** не определяет тела этих методов. В этом случае вызываемый метод находится динамически **(строки 63…65)**, так что площадь окружности вычисляется методом, определенным в классе **Circle** **(строки 22…)**, а площадь прямоугольника - методом, определенным в классе **Rectangle (строки 40…)**.

```
1  using System;                    // Абстрактный базовый класс и производные классы
2                                                                    // Листинг 6
3  namespace ConsAppl_Java_Спр_132_abstract
4  {
5      public abstract class Shape          // Абстрактный базовый класс
6      {
5          public abstract double area();              // см. реализацию – строки 22 и 40
8          public abstract double circumference();  // длина замкнутой кривой – строки 26 и 44
9      } /////////////////////////// конец класса Shape ////////////////////
10
11     class Circle : Shape                 // Производный класс Circle
12     {
13         public static double PI = 3.14159265358979323846;
14
15         protected double r;          // Данные экземпляра
16
17         public Circle(double r)        // Конструктор
18         {
19             this.r = r;
20         }
21
22         public override double area()   // Реализации абстрактных методов:
23         {
24             return PI * r * r;                  // 1) площадь: строка 5
25         }
26         public override double circumference()
27         {       return 2 * PI * r;             // 2) длина окружности: строка 6
28         }
29     } /////////////////////////// Конец класса Circle ////////////////////
30
31     class Rectangle : Shape              // Производный класс Rectangle
32     {
33         protected double w, h;           // Данные экземпляра
34
35         public Rectangle(double w, double h)  // Конструктор
```

```
36          {
37              this.w = w; this.h = h;
38          }
39
40          public override double area() // Реализации абстрактных методов:
41          {
42              return w * h;                       // 1) площадь: строка 5
43          }
44          public override double circumference()
45          {
46              return 2 * (w + h);                 // 2) периметр: строка 6
47          }
48      }       ////////////////////////// Конец класса Rectangle ////////////////////
49
50      public class TestShape
51      {
52          public static void Main(String[ ] args)
53          {           // shapes - полиморфная переменная базового класса
54              Shape[ ] shapes = new Shape[3];                 // строки 5…9
55
56              shapes[0] = new Circle(3.0);                    // строки 17…20
57              shapes[1] = new Rectangle(1.0, 2.0);            // строки 35…37
58              shapes[2] = new Rectangle(4.0, 2.0);            // строки 35…37
59
60              double total_area = 0;
61              for (int i = 0; i < shapes.Length; i++)
62              {                                                       // строки 22, 40 и 5
63                  Console.WriteLine("\nПлощадь " + i + "-й фигуры - " + shapes[i].area());
64                                                                      // строки 26, 44 и 6
65                  Console.WriteLine("Периметр " + i + "-й фигуры - " + shapes[i].circumference());
66                  total_area += shapes[i].area();
67              }
68              Console.WriteLine("\nСуммарная площадь фигур - " + total_area);
69          Console.ReadLine();
70          }
71      }
72  }
```



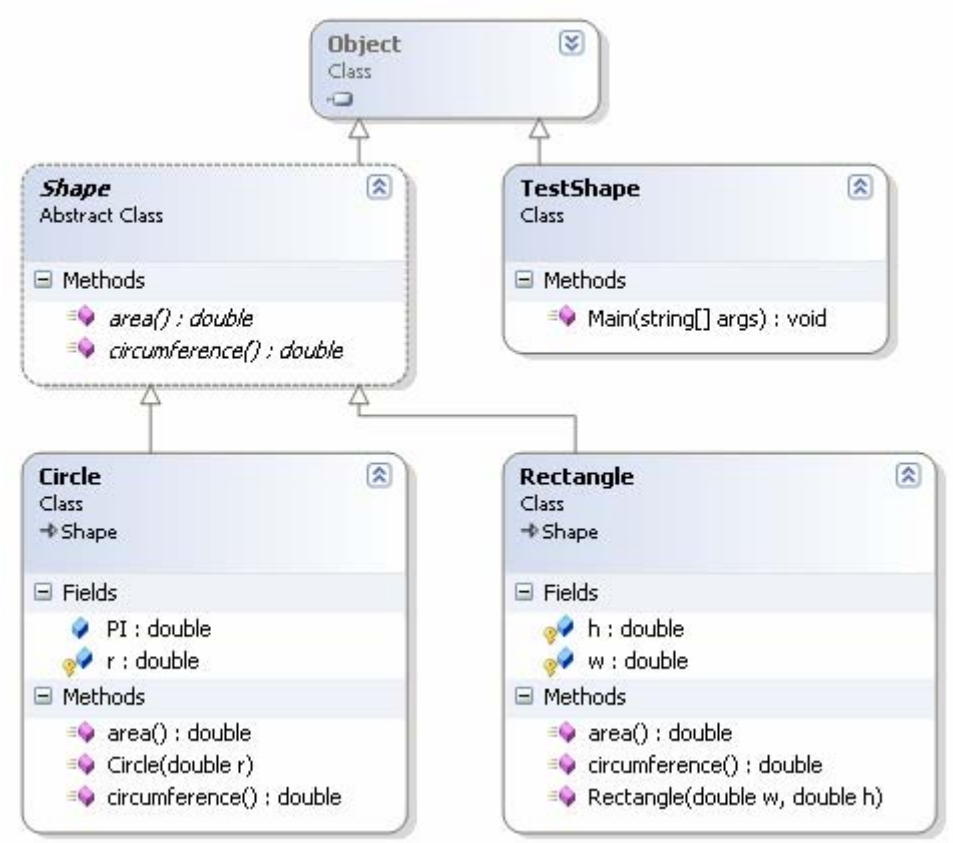


Рис. 16. Диаграмма классов и результат работы программы **Листинга 6**

П.2. Интерфейсы – сопоставление с абстрактным классом (сравнить с Листингом 6)

Расширим программу **Листинга 6**. Предположим, что теперь нужно реализовать несколько фигур, которые знают: ***1)*** не только свой размер, но и ***2)*** позиции своих центров в декартовой системе координат. Один из способов создать такие фигуры - определить **абстрактный** класс **CenteredShape**, а затем реализовать нужные *под*классы, например **CenteredCircle, CenteredRectangle** и т. д.

Но также желательно, чтобы позиционируемые классы поддерживали ранее определенные методы **area()** и **circumference()** без повторной реализации этих методов. Например, мы хотим определить класс **CenteredCircle** как подкласс **Circle (класс Circle уже наследует класс Shape, см. строку 11 в Листинге 6)**, чтобы он унаследовал методы **area()** и **circumference()**. Но у класса в C# может быть только один непосредственный родительский (базовый) класс. Если класс **CenteredCircle** расширяет (наследует) класс **Circle**, то он не может также расширять (наследовать) абстрактный класс **CenteredShape**![1]

В C# (и в Java) решение этой проблемы сводится к созданию интерфейсов. Хотя C#-класс может расширять (**наследовать**) только один родительский класс, он способен ***реализовать*** (implement) несколько интерфейсов.

П.2.1. Определение интерфейса

Интерфейс - это ссылочный тип, очень похожий на класс. Определение интерфейса во многом похоже на определение абстрактного класса, за исключением того, что слова **abstract** и **class** заменяются на слово **interface**. Когда вы определяете интерфейс, вы создаете новый ссылочный тип, как при создании класса. В соответствии со своим названием **интерфейс** предоставляет API (**A**pplication **P**rogramming **I**nterface – программный интерфейс приложения) к определенной функциональности. Интерфейс **не** определяет реализации этого API. Есть несколько ограничений, накладываемых на члены интерфейса:

A. Интерфейс не содержит реализации чего бы то ни было. Все методы интерфейса являются абстрактными, даже если модификатор **abstract** опущен. **У методов интерфейса нет реализации; вместо тела метода стоит точка с запятой.** Так как интерфейс содержит только абстрактные методы, а методы класса не могут быть абстрактными, то *методы интерфейса всегда являются* **методами экземпляра**.

B. Интерфейс определяет открытый **API**. Все методы интерфейса неявно объявлены как **public**. Определение protected- или private-методов в интерфейсе недопустимо.

C. Хотя класс определяет данные и методы их обработки, **интерфейс не может определить поля экземпляра**. Поля являются деталями реализации, а интерфейс не определяет никакой реализации. В определении интерфейса могут появляться только **константы**, объявленные с модификаторами **static** и **sealed.**

D. Нельзя создать ни одного экземпляра интерфейса, поэтому у него нет конструктора.

В **листинге 7** представлено определение интерфейса с именем **Centered** (**см. строки 6…11**). Этот интерфейс определяет методы, которые подкласс класса **Shape**, должен реализовать, зная координаты **x** и **y** своего центра.

---

[1] C++ позволяет классам иметь несколько родителей с помощью техники, известной как множественное наследование. Множественное наследование сильно усложняет язык; C# поддерживает более элегантное решение – **интерфейс** (**interface**).

П.2.2. Реализация интерфейса

Подобно тому, как класс использует двоеточие « **:** » для указания базового класса, он может применить двоеточие « **:** » для указания интерфейсов, которые он реализует. За двоеточием « **:** » должно следовать **имя** или имена интерфейсов, **реализуемых** классом. Имена разделяются запятыми (см. строки **62** и **88**).

Когда класс объявляет интерфейс, он сообщает о предоставлении реализации (то есть тела) для каждого метода этого интерфейса. Если класс реализует интерфейс, но не предоставляет реализации для всех его методов, то он наследует *не*реализованные методы как абстрактные и сам должен быть объявлен как **abstract**. Если класс реализует более одного интерфейса, то он должен предоставить реализацию каждого метода каждого интерфейса, либо он должен быть объявлен как **abstract**.

В **листинге 7** определен класс **CenteredRectangle** (см. строки **62…85**), который расширяет (наследует) класс **Rectangle** и реализует интерфейс **Centered**.

Как отмечено ранее, в определении интерфейса могут появляться константы. Любой класс, реализующий интерфейс, наследует эти константы и может использовать их так, словно они были определены непосредственно в классе. Нет необходимости ставить перед константами имя интерфейса или как-то их реализовывать.

```
1  using System;    // Производный класс НАСЛЕДУЕТ базовый и РЕАЛИЗУЕТ интерфейс
2                                                    // Листинг 7 (сравнить с листингом 6)
3  namespace ConsAppl_Java_Cпр_134_interfac
4  {
5                                                ////////////// Интерфейс ////////////////////
6      public interface Centered
7      {
8          void setCenter(double x, double y);
9          double getCenterX();
10         double getCenterY();
11     }
12
13                                               ///////// Абстрактный базовый класс Shape/////////
14     public abstract class Shape
15     {
16         public abstract double area();
17         public abstract double circumference();
18     }
19
20                                               ///////////Производный класс Circle/////////////
21     public class Circle : Shape
22     {
23         public static double PI = 3.14159265358979323846;
24         protected double r;           // Данные экземпляра
25
26         public Circle(double r)       // Конструктор
27         {
28             this.r = r;
```

```
29      }
30
31      public override double area()           // Реализация абстрактного метода, строка  16
32      {
33          return PI * r * r;
34      }
35      public override double circumference()// Реализация абстрактного метода, строка  17
36      {
37          return 2 * PI * r;
38      }
39  }
40
41  ////////////Производный класс Rectangle////////
42  public class Rectangle : Shape
43  {
44      protected double w, h;                          // Данные экземпляра
45
46      public Rectangle(double w, double h)        // Конструктор
47      {
48          this.w = w; this.h = h;
49      }
50
51      public override double area()           // Реализация абстрактного метода, строка  16
52      {
53          return w * h;
54      }
55      public override double circumference()// Реализация абстрактного метода, строка  17
56      {
57          return 2 * (w + h);
58      }
59  }
60
61  //////////Производный класс CenteredRectangle//////////
62  public class CenteredRectangle : Rectangle, Centered
63  {
64      private double cx, cy;    // Новые поля  экземпляра
65
66      public CenteredRectangle(double cx, double cy, double w, double h) : base(w, h)
                                    // вызов конструктора баз-го кл-са Rectangle, строка 46
67      {
68          this.cx = cx;
69          this.cy = cy;
70      }
71
72                          /* НАСЛЕДУЕМ ВСЕ методы класса Rectangle,
        но должны обеспечить РЕАЛИЗАЦИЮ ВСЕХ методов интерфейса Centered */
73      public void setCenter(double x, double y)    // строка 8
```

```
74          {
75              cx = x; cy = y;
76          }
77          public double getCenterX()                  // строка 9
78          {
79              return cx;
80          }
81          public double getCenterY()                  // строка 10
82          {
83              return cy;
84          }
85      }//////////end class CenteredRectangle//////////
86
87      //////////Производный класс CenteredCircle//////////
88      public class CenteredCircle : Circle, Centered
89      {
90
91          private double cx, cy;          // Новые поля  экземпляра
92
93
94          public CenteredCircle(double cx, double cy, double r)
95              : base(r)                       // вызов конструктора баз-го кл-са Circle, строка 26
96          {
97              this.cx = cx;
98              this.cy = cy;
99          }
100
101                             /* НАСЛЕДУЕМ ВСЕ методы класса Circle,
            но должны обеспечить РЕАЛИЗАЦИЮ ВСЕХ методов интерфейса Centered */
102         public void setCenter(double x, double y)           // строка 8
103         {
104             cx = x; cy = y;
105         }
106         public double getCenterX()                          // строка 9
107         {
108             return cx;
109         }
110         public double getCenterY()                          // строка 10
111         {
112             return cy;
113         }
114     }   //////////////end class CenteredCircle//////////////
115
116     public class TestShapeInterface
117     {// начальный класс
118         public static void Main(String[ ] args)
119         {                   // shapes - полиморфная переменная. Она может иметь два типа:
```

```
120         Shape[ ] shapes = new Shape[2];                    // 1-й тип  – Shape, строка 13,
121
122         shapes[0] = new CenteredCircle(1.0, 1.0, 1.0);          // строки 94, 95 и 26
123         shapes[1] = new CenteredRectangle(2.3, 4.5, 3.0, 4.0);  // строки 66 и 46
124
125         double total_area = 0;
126         double total_distace = 0;
127
128         for (int i = 0; i < shapes.Length; i++)
129         {                                                       // строки 31, 46 и 16
130            Console.WriteLine("\nПлощадь " + i + "-й фигуры - " + shapes[i].area());
131                                                                 // строки 35, 55 и 17
132            Console.WriteLine("Периметр " + i + "-й фигуры - " + shapes[i].circumference());
133            total_area += shapes[i].area();
134
135             if (shapes[i] is Centered)                         // 2-й тип  – Centered, строка 5
136             {  Centered c = (Centered)shapes[i];
137                double cx = c.getCenterX();             // строки 106, 77 и 9
138                double cy = c.getCenterY();             // строки 110, 81 и 10
139                total_distace += Math.Sqrt(cx * cx + cy * cy);
140 Console.WriteLine("\nРасстояние до центра " + i + "-й фигуры - " + total_distace + "\n");
141             }
142         }
143         Console.WriteLine("\nСредняя площадь фигур - " + total_area / shapes.Length);
144         Console.WriteLine("\nСреднее расстояние - " + total_distace / shapes.Length);
145         Console.ReadLine();
146      }
147   }
148 }
```

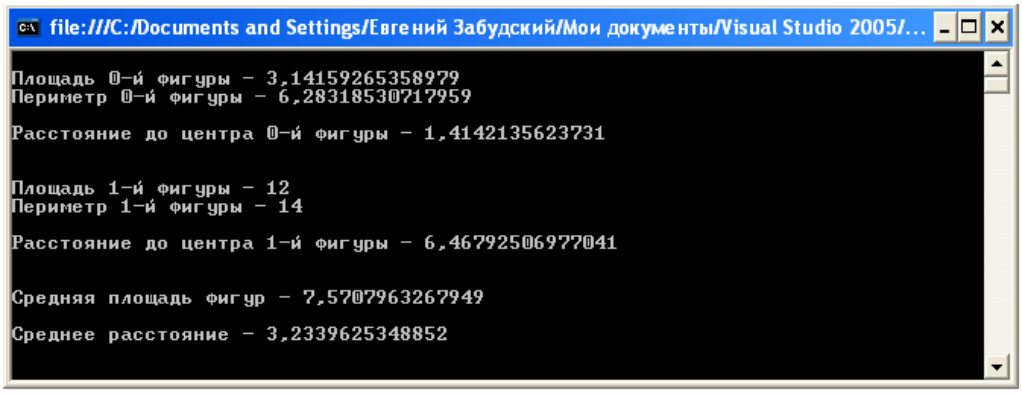


Рис. 17. Результат работы программы **листинга 7**

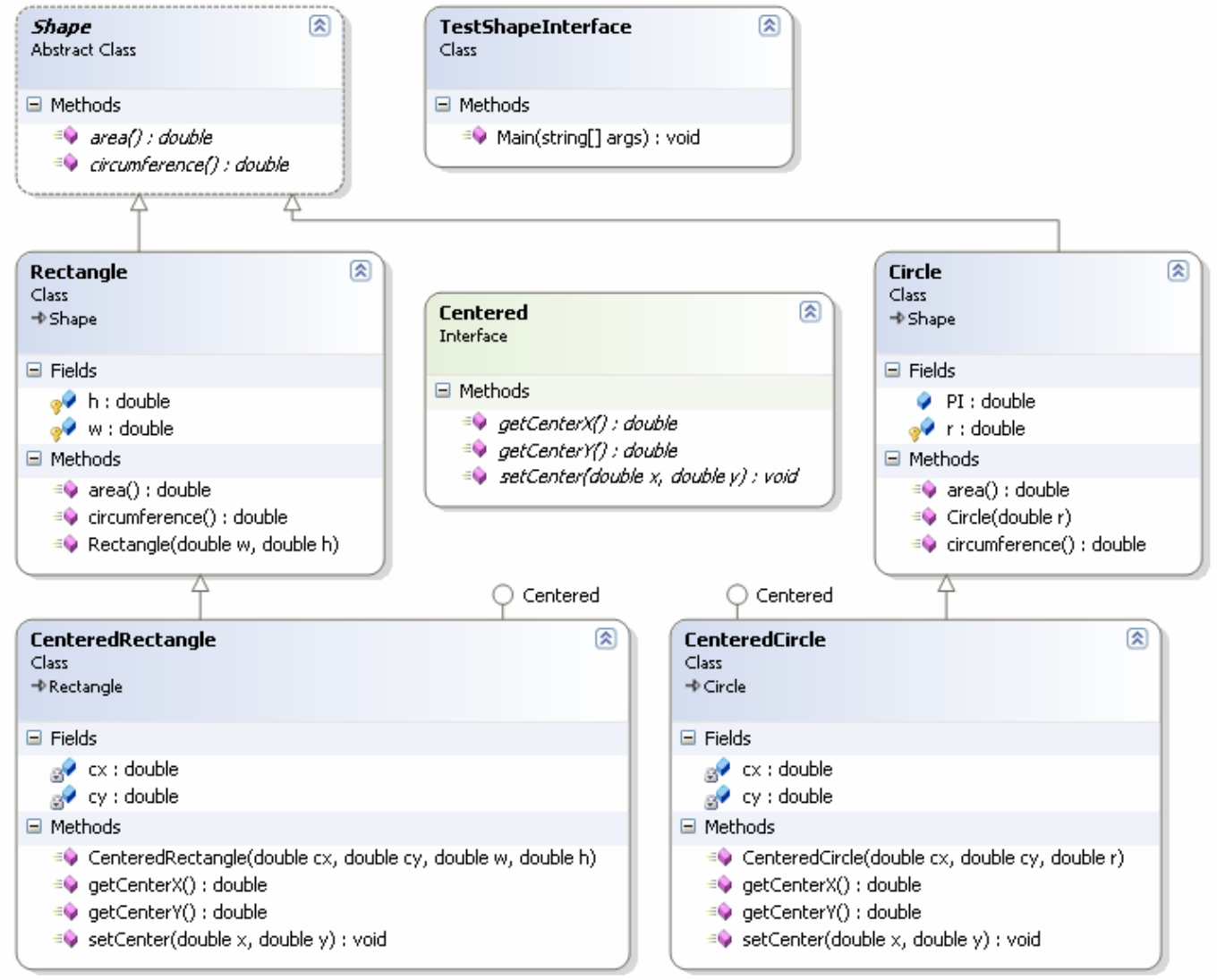


Рис. 18. Диаграмма классов программы **листинга 7**

П.2.3. Когда нужно применять интерфейсы

При определении абстрактного типа (например, **Shape, см. выше строку 14,сл.**), у которого будет несколько подтипов (например, **Circle /строка 21,сл./**, **Rectangle /строка 42,сл./**, **Square**), вы можете оказаться перед выбором между интерфейсами и абстрактными классами. Так как у них похожие возможности, то не всегда ясно, что лучше использовать.

Интерфейс полезен тем, что его может реализовать любой класс, даже если этот класс расширяет другой родительский класс (см. строки **62 или 88**). Но интерфейс - **не** содержит реализаций. Если у интерфейса есть много методов, то вам может наскучить снова и снова реализовывать эти методы, особенно если большая часть реализации дублируется в каждом классе.

С другой стороны, класс, расширяющий (наследующий) абстрактный класс, не может расширять (наследовать) другой класс. В некоторых ситуациях это вызывает трудности при проектировании. Впрочем, если класс не является полностью абстрактным, он может содержать частичную реализацию, которую могут применять подклассы. В некоторых случаях множество подклассов может рассчитывать на реализации методов по умолчанию, предоставленные абстрактным классом.

Другое важное отличие между интерфейсами и абстрактными классами состоит в обеспечении совместимости. Если вы объявили интерфейс как часть открытого (**public**) **API**, а затем добавили в интерфейс новый метод, то вы нарушили целостность всех классов, которые реализуют предыдущую версию интерфейса. Если вы используете абстрактный класс, вы можете благополучно

добавить в этот класс неабстрактный метод без необходимости модификации существующих классов, расширяющих абстрактный класс.

В некоторых ситуациях интерфейс или абстрактный класс являются правильным выбором. В других случаях типовой шаблон подразумевает использование обоих вариантов. Во-первых, определите тип как полностью абстрактный интерфейс. Затем создайте абстрактный класс, который реализует интерфейс и предоставляет полезные предопределенные реализации подклассам.

П.3. Интерфейс HTMLSource

Создадим свой собственный интерфейс (пользовательский), который назовем **HTMLSource (см. далее строки 5…8 в Листинге 8)**. Функциональность этого интерфейса будет состоять в форматировании данных по правилам **HTML (см. таблицу и далее строки 54…58)**. **Таблица**

Открывающий тег	Закрывающий тег	Описание
<B>	</B>	Текст между этими тегами выводится с использованием **полужирного** шрифта. Пример: <B>Полужирный</B> текст.

Интерфейс **HTMLSource** будет иметь единственный метод **GetHTML() (строка 7)**, который возвращает данные объекта в виде **HTML**-форматированной строки **(строка 57)**.

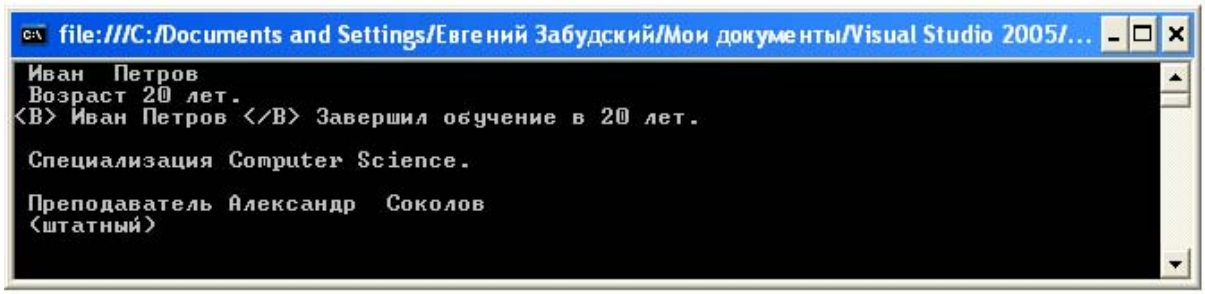
Помните: интерфейсы реализуют поведение, которое подразумевает, что какой-то другой код будет вызывать методы интерфейса. В нашем примере классы **Student (строка 27)** реализуют интерфейс **HTMLSource**, но использует этот интерфейс только метод **ShowHTML (см. строки 36…39 и 54…58)**.

```
1  using System;                    // Интерфейсы
2                                                          // Листинг 8
3  namespace ConsAppl_OOP_Keo_c168_171
4  {        /* функциональность интерфейса IHTMLSource будет состоять в
                форматировании данных по правилам HTML, см. строку 57*/
5     public interface IHTMLSource
6     {
7        string GetHTML();
8     }                                  //////////////end interface IHTMLSource//////////////
9
10    public class Person
11    {
12       protected string FirstStr, LastStr;
13
14
15       public void Modify(String First, String Last)
16       {
17          FirstStr = First;
18          LastStr = Last;
19       }
20
21       public void Display()
22       {
23          Console.WriteLine(FirstStr + " " + LastStr);
```

```
24        }
25    }                                          //////////end class Person//////////
26
27    public class Student : Person, IHTMLSource
28    {
29        protected int GraduationYear;
30
31        public Student()                       // Конструктор
32        {
33            GraduationYear = 0;
34        }
35  // Этот метод ShowHTML использует интерфейс IHTMLSource
36        public void ShowHTML(IHTMLSource someObject)
37        {
38            Console.WriteLine(someObject.GetHTML());        // см. строку 54
39        }
40
41        public void Modify(String First, String Last, int Graduation)
42        {
43            base.Modify(First, Last);          // см. строку 15
44            GraduationYear = Graduation;
45        }
46
47                                               // "new" свидетельствует о том, что м-д Display() новый
48        new public void Display()     // о new - см. Мик-н, глава 17, с. 676 - 680, с. 693, с.697
49        {
50            base.Display();                    // см. строку 21
51            Console.WriteLine(" Возраст " + GraduationYear + " лет.");
52        }
53
54        public string GetHTML()                // см. строку 36, 38 и  5
55        {
56  // в строке 57 реализация метода GetHTML() интерфейса IHTMLSource, см. строку 5
57  return "<B>" + FirstStr + LastStr + "</B>" + " Завершил обучение в " + GraduationYear + " лет." ;
58        }
59    }                                          //////////////end class Student//////////////
60
61    public class Instructor : Person
62    {
63        protected bool TenuredBool;
64
65        public Instructor()
66        {
67            TenuredBool = false;
68        }
69
70        public void Modify(String First, String Last, bool Tenured)
```

```
71          {
72              base.Modify(First, Last);                   // см. строку 15
73              TenuredBool = Tenured;
74          }
75                                                  // "new" свидетельствует о том, что м-д Display() новый
76          new public void Display() // о new - см. Мик-н, глава 17, с. 676 - 680, с. 693,  с.697
77          {
78              base.Display();                             // см. строку 21
79              if (TenuredBool)
80                  Console.WriteLine(" (штатный)");
81              else
82                  Console.WriteLine(" (Не штатный)");
83          }
84      }                                                   ////////////end class Instructor//////////
85
86      public class Course
87      {
88          private string NameStr;
89
90          public void Modify(String Name)
91          {
92              NameStr = Name;
93          }
94
95          public void Display()
96          {
97              Console.WriteLine(NameStr);
98          }
99      }                               ////////////end class Course/////////////////
100
101     class Tester
102     {
103         public static void Main()
104         {
105             Student someObject = new Student();   // см. строку 31
106
107             someObject.Modify(" Иван ", " Петров ", 20);        // см. строку 41, 43 и 15
108             someObject.Display();                               // см. строку 48, 50 и  21
109             someObject.ShowHTML(someObject);                    // см. строку 36, 38 и 54
110
111              Course course = new Course();                                  // см. строку 86
112             course.Modify("\n Специализация Computer Science.");  // см. строку 90
113              course.Display();                                              // см. строку 97
114
115             Instructor instructor = new Instructor();      // см. строку 65
116 instructor.Modify("\n Преподаватель Александр ", "Соколов ", true);   // см. строку 70, 72 и 15
117              instructor.Display();         // см. строку 76, 78 и  21
```

```
118
119
120             Console.ReadLine();
121         }
122     }                    ////////////end class Tester/////////////////
123 }
```



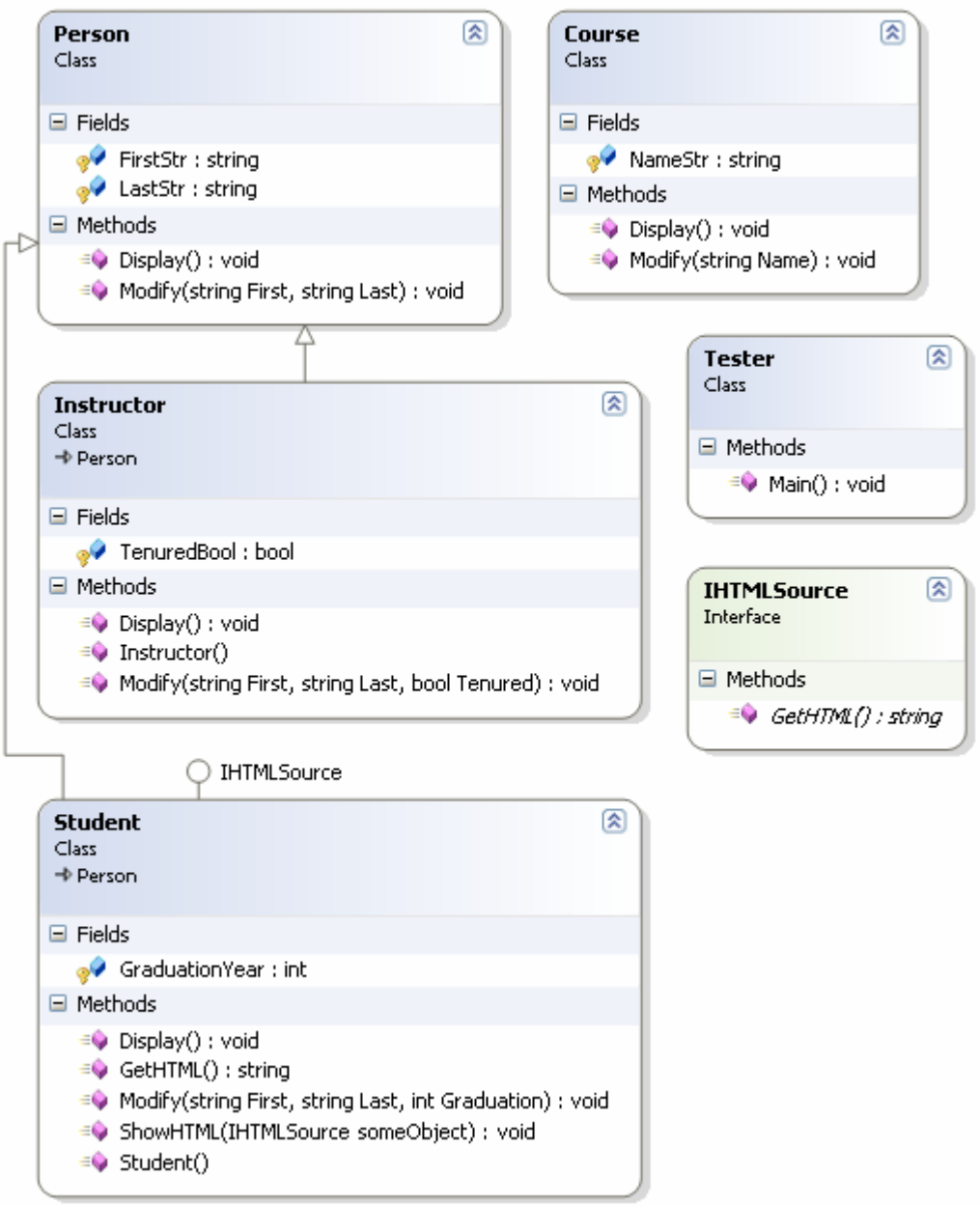


Рис. 19. Вывод и Диаграмма классов программы листинга 8

Литература

Базовый учебник

1. Мейер Б. Объектно-ориентированное конструирование программных систем. М.: Русская Редакция, 2005.

Основная

2. Буч Г., Якобсон А., Рамбо Дж. **UML**. С.-Петербург: Питер, 2006.

3. Гамма Э. Приемы объектно-ориентированного проектирования. Паттерны проектирования. С.-Петербург: Питер, 2006.

4. Забудский Е.И. Учебно-методические материалы по дисциплине «Объектно-ориентированный анализ и программирование». М.: Кафедра ОИиППО ГУ-ВШЭ, 2005,
Internet-ресурс – http://new.hse.ru/C7/C17/zabudskiy-e-i/default.aspx или http://zei.narod.ru/ .

5. Кватрани Т. Визуальное моделирование с помощью Rational Rose 2002 и UML. М.: Вильямс, 2003.

6. Лафоре Р. Объектно-ориентированное программирование в C++. С.-Петербург: Питер, 2005.

7. Троелсен Э. C# и платформа **.**NET. С.-Петербург: Питер, 2006.

8. Синтес А. Освой самостоятельно объектно-ориентированное программирование за 21 день. Москва; С.-Петербург; Киев: Вильямс, 2002.

Дополнительная – Internet-ресурсы

9. Новые книги раздела **C#**. – http://books.dore.ru/bs/f6sid16.html .

10. **C#** и ***.NET*** по шагам. – http://www.firststeps.ru .

11. ***UML*** – язык графического моделирования. – http://www.uml.org/ .

12. ***JUnit*** – каркас тестирования для испытания *Java*-классов. – http://www.junit.org .

13. Пакет объектного моделирования ***Rational Rose***. – http://www-306.ibm.com/software/rational/ .

Дополнительная – книги

14. Мэтт Вайсфельд. Объектно-ориентированный подход: Java, **.**NET, C++. М.: КУДИЦ-ОБРАЗ, 2005.

15. Дж. Кьоу, М. Джеанини. Объектно-ориентированное программирование. С.-Петербург: Питер, 2005.