студентов по всем предметам. При этом время выполнения лабораторных работ в компьютерных кабинетах сократилось в среднем на 7%, в результате чего появилась возможность, на более качественном уровне, переработать лекции, лабораторные и самостоятельные работы.

Литература

1. Профессиональные стандарты в области информационных технологий – М.: АП КИТ, 2008.-616с.

2. Федеральный государственный стандарт общего образования. Среднее (полное) общее образование. Проект. – М.: ИСИО Российской академии образования, 2011. – 41 с. http://mon.gov.ru/files/materials/7956/11.04.19-proekt.10-11.pdf

Интернет ресурсы:

3. Информационно-коммуникационные технологии в образовании.// Система федеральных образовательных порталов: [сайт]. - ©Государственный научно-исследовательский институт информационных технологий и телекоммуникаций, [2003-2011]. - Режим доступа: http://www.ict.edu.ru

О НЕПРЕДНАМЕРЕННОМ НАРУШЕНИИ ПРИНЦИПА ИНКАПСУЛЯЦИИ ПРИ РАЗРАБОТКЕ ОБЪЕКТНО-ОРИЕНТИРОВАННЫХ ПРОГРАММ

Кириченко А.А.

Москва, Национальный исследовательский университет «Высшая школа экономики»

При обучении программированию компьютерное тестирование студентов показывает удовлетворительное знание ими принципов объектно-ориентированного программирования (ООП). Но в процессе выполнения сколько-нибудь сложных программ (домашних заданий, курсовых работ) большинство из них нарушает принцип инкапсуляции. В статье рассматриваются причины непреднамеренного нарушения студентами принципа инкапсуляции и возможные способы их устранения.

About inadvertent violation of the principle of encapsulation when developing object-oriented programs. Kirichenko A.A. Moscow, National research university "The higher school of economy"

When training in programming computer testing of students shows satisfactory knowledge of the principles of object-oriented programming. But in the course of performance some difficult programs (homeworks, term papers) the majority of them breaks the principle of encapsulation. In article the reasons of inadvertent violation by students of the principle of encapsulation and possible ways of their elimination are considered.

Создание ООП ведётся на языке C# при активном использовании библиотеки классов MSDN и пакета Visual Studio (VS), и приводит к манипулированию такими понятиями C#, как классы (двух типов: типа «модуль» и типа «абстрактный тип данных») и объекты. Практика показывает, что создание ООП - технически сложная проблема, предусматривающая:

а) объектную декомпозицию решаемой задачи, в результате которой выясняется состав взаимодействующих объектов;

б) разработка классов типа «абстрактный тип данных» для создания чертежей (схем) требуемого состава объектов;

в) разработка класса – модуля, содержащего метод «main», в котором создаются и взаимодействуют объекты системы.

При таком подходе в конечном итоге объектно-ориентированная программа должна содержать программную структуру, в которой имеется по меньшей мере один класс типа «модуль», несколько классов типа «абстрактный тип данных», и необходимое количество взаимодействующих объектов.

Инкапсуляция является одним из ключевых понятий ООП. Формально это понятие определяется следующим образом: *Инкапсуляция - это механизм, который* ***объединяет*** *данные и методы, манипулирующие этими данными, и* ***защищает*** *их от внешнего вмешательства или неправильного использования.*

Согласно принципам ООП смешивать данные и методы их обработки для разных типов объектов в одном классе недопустимо.

Г.Шилдт в [1] утверждает: « Несмотря на отсутствие специального синтаксического правила определения класса (его качественного и количественного состава), все же считается, что класс должен определять только одну логическую сущность. Например, класс, в котором хранятся имена лиц и их телефонные номера, не должен (по общепринятым меркам) также содержать информацию о среднем количестве осадков, циклах возникновения пятен на Солнце и прочую не связанную с конкретными лицами информацию.

Другими словами, правильно определенный класс должен содержать логически связанные данные».

Разработка на C# программ с асинхронным интерфейсом предусматривает создание такой конструкции, как формы, и нанесение на неё компонент и элементов управления. При комплектовании формы приходится определять размеры элементов управления, их локализацию на форме, цвет, поясняющие надписи, обработчики событий, возникающих при активизации элементов управления. При этом должны учитываться принципы объектно-ориентированного программирования.

При практическом же использовании средств фирмы Микрософт приходится использовать программную систему Visual Studio, которая в принципе позволяет реализовать сколь угодно сложные программы для любой модели программирования. Но по умолчанию она предназначена для реализации модели визуального программирования, иногда называемого событийным программированием, а не ООП.

Если программирование ведётся на Visual Studio, то наносимые на форму элементы управления размещают свой код в классе, характеризующем форму. При визуальном программировании для программ, управляемых событиями, включение в класс, характеризующий форму, обработчиков событий, возникающих при активизации элементов управления, допустимо и не нарушает требований этого стиля программирования.

Стиль объектно-ориентированного программирования отличается от стиля событийного программирования прежде всего тем, что классы в нём могут являться не только модулями программы, но и абстрактными типами данных.

Класс, как модуль программы – это коробочка (ёмкость), в которую можно поместить какие-то элементы программы. Для событийного программирования вполне допустимо всю программу разместить в одном – единственном классе. Это не противоречит его принципам. Visual Studio так и делает. При комплектовании формы все её характеристики, включая и коды обработчиков событий, возникающих при активизации находящихся на форме компонент и элементов управления, размещаются в одном и том же классе.

Рассмотрим пример, реализованный как Windows-приложение и демонстрирующий форму, позволяющую проводить исследование структуры диска:

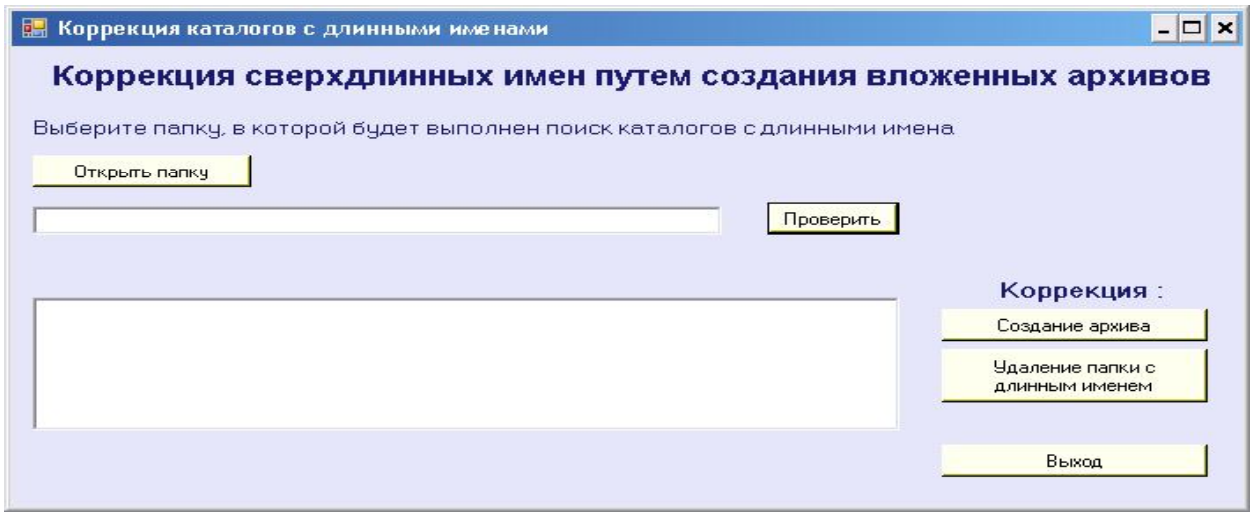

Рис. 1. Интерфейс программы для исследования структуры диска

При построении интерфейса использовались такие элементы управления, как метки (текстовые окна для вывода текста), окна редактирования, позволяющие выводить и вводить в них текст и редактировать его, командные кнопки.

Код программы, реализующей форму создаётся VS в виде:

```
using System;
using System.IO;
using System.Diagnostics;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace _2term_work
{
   public partial class Form1 : Form
   {

//Определение переменных
      string[] a;
      string str;

//Конструктор
      public Form1()
      {
         InitializeComponent();
      }

//Обработчики событий
//------------------------------------------------------------
private void button1_Click(object sender, EventArgs e)
      {
```

```
if (textBox1.Text.Equals("")) { MessageBox.Show("Выберите каталог для проверки"); }
        else
        {
           a = new string[100];
           try
           {
string[]          subdir          =          Directory.GetDirectories(textBox1.Text,          "*",
System.IO.SearchOption.AllDirectories);
              int k = 0;
              int j = 0;
              for (int i = 0; i < subdir.Length; i++)
              {
                 if (subdir[i].Length > 200)
                 {
  label1.Text = "В папке обнаружены имена длиной более 200 символов";
                    k += 1;
                    a[j] = subdir[i]; j++;
                 }
              }
              listBox1.DataSource = a;
              if (k == 0)
              {
   label1.Text = "Папок с длиной имени более 200 символов не обнаружено";
              }
           }
 catch { MessageBox.Show("Выбранная папка содержит защищенные каталоги"); }
        }
     }
//---------------------------------------------------------------------------
     private void button2_Click(object sender, EventArgs e)
     {
        FolderBrowserDialog d = new FolderBrowserDialog();
        d.Description = "Выберите папку для проверки";
        d.SelectedPath = @"c:\";
        if (d.ShowDialog() == DialogResult.OK)
        {
           textBox1.Text = d.SelectedPath;
        }
     }
//---------------------------------------------------------------------------
     private void button3_Click(object sender, EventArgs e)
     {
        string dir = "", dir1 = "";
if (listBox1.SelectedIndex == -1) { MessageBox.Show("Выберите в списке слева папку
для реконструкции"); }
        else
        {
           int k = listBox1.SelectedIndex;
           string[] d = a[k].Split('\\');
           string[] g = textBox1.Text.Split('\\');
```

```
            dir = textBox1.Text + "\\" + d[g.Length];
            dir1 = dir + ".rar";
            Process pro = new Process();
            pro.StartInfo.CreateNoWindow = true;
            pro.StartInfo.WindowStyle = ProcessWindowStyle.Hidden;
        pro.StartInfo.Arguments = "a -y -inul -ibck " + dir1 + " " + dir;
            pro.StartInfo.FileName = "WinRAR";
            pro.Start();
            str = dir;
          }
        }
//----------------------------------------------------------------------
        private void button4_Click(object sender, EventArgs e)
        {
          Dispose();
        }
//----------------------------------------------------------------------
        private void button5_Click(object sender, EventArgs e)
        {
          int k = 0;
          Directory.Delete(str, true);
          str = "";
          a = new string[100];
          listBox1.DataSource = a;
string[]          subdir          =          Directory.GetDirectories(textBox1.Text,          "*",
System.IO.SearchOption.AllDirectories);
          int p = 0;
          int j = 0;
          for (int i = 0; i < subdir.Length; i++)
          {
            if (subdir[i].Length > 200)
            {
    label1.Text = "В папке обнаружены имена длиной более 200 символов!";
              p += 1;
              a[j] = subdir[i]; j++;
            }
          }
          listBox1.DataSource = a;
          if (k == 0)
          {
   label1.Text = "Папок с длиной имени более 200 символов не обнаружено";
          }
        }
//Метод Main()
//----------------------------------------------------------------------
  public static void Main()
        {
          Application.EnableVisualStyles();
          Application.SetCompatibleTextRenderingDefault(false);
          Application.Run(new Form1());
```

```
        }
    }     //Конец класса Form1
}   //Конец пространства имён _2term_work
```

Этот код содержит один класс, в котором спроектированы:

1/ выводимая на экран форма,

2/ код для выбора проверяемого каталога (в виде обработчика события),

3/ код выбора папки для реконструкции (в виде обработчика события),

4. код проверки длины имён файлов и папок (в виде обработчика события) и т.д.

Практически весь код этого программного проекта сосредоточен в одном классе Form1. Модель программирования, реализованная в VS, к размещению кода строгих требований не предъявляет.

При создании компонент и элементов управления определяются некоторые их атрибуты, в число которых входит и функциональность элемента – т.е. определение того, что должно происходить при активизации этого элемента. Для этого создаются обработчики событий, которые системой VS размещаются так же в классе Form1.

Но форма создаётся только для визуального отображения элементов управления, и код, определяющий размеры, локализацию, цвет, надписи для формы необходим, не является для неё посторонним. А обработчики возникающих при активизации размещённых на форме элементов управления событий содержат код, не имеющий к визуальному отображению элементов управления никакого отношения. Включение такого кода в состав класса, характеризующего форму, нарушает один из основных принципов объектно – ориентированного программирования – инкапсуляцию, и является недопустимым для программирования в стиле ООП.

В отличии от модели событийного программирования при объектно-ориентированном программировании необходимо учитывать дополнительные условия, предъявляющие к структуре программы специфические требования.

Характерные особенности объектно-ориентированных программ:

1. Объектная ориентация связана с использованием особой конструкции программы – объектов.

2. Класс (class) – это шаблон, чертёж, схема объектов. Он определяет лишь типовые черты объектов.

3. Объект (object)- это конкретная реализация, экземпляр класса.

4. Для классов характерно, что методы и свойства взаимосвязаны, класс должен определять только одну логическую сущность. Обычно класс при объектно – ориентированном программировании содержит абстрактный тип данных - группу тесно связанных между собой данных и методов (функций), которые могут осуществлять операции над этими данными. Смешивать данные и методы их обработки для разных типов объектов в одном классе такого типа недопустимо.

5. В конечном итоге программа может содержать большое количество классов, как универсальных, так и специализированных, как правило, слабо связанных между собой, или даже совсем не связанных.

6. Кроме этого в объектно-ориентированной программе должна существовать какая-то объединяющая конструкция, в которой из этих классов создаются объекты и в которой эти объекты живут, взаимодействуют, развиваются.

7. В языке C# такая конструкция создаётся в виде отдельного класса, отличительной чертой которого является находящийся в нём метод Main().

8. С метода Main() начинается исполнение любой самой сложной программы. Имя этого метода является зарезервированным. Главное его назначение – он является

ведущим методом программного проекта, именно в нём должны создаваться, жить, развиваться и взаимодействовать все объекты.

9. Класс, в котором находится метод Main(), не имеет какого-либо специального имени, оно может быть любым, выбирает его пользователь. Кроме метода Main() в этом классе могут содержаться и другие методы и свойства, необходимые для работы программы и не имеющие отношения к другим классам проекта. Этот класс не является абстрактным типом данных. Это класс – модуль.

10. Обычно Visual Studio размещает метод Main() в файле Program.cs, создавая для этого класс «static class Program».

Удачным примером объектно-ориентированной программы, в которой видны взаимоотношения между классами, является общая схема реализации паттерна «Издатель –Подписчик».

```
using System;
//-------------------------------------------------------
//объявление делегата
 delegate void MyEventHandler();
//-------------------------------------------------------
// объвление класса – издателя события
 class MyEvent {
 public event MyEventHandler SomeEvent;   //объявление события SomeEvent
 public void FireSomeEvent() {
 if(SomeEvent != null)         //проверка, можно ли зажигать событие
   SomeEvent();                //метод для зажигания события SomeEvent
}
}
//конец класса – издателя
//-------------------------------------------------------
// объявление класса – ресивера
 class Resiver {
 public static void handler()   {
    Console.WriteLine("Событие произошло");
                    }
}
// конец класса - ресивера
//-------------------------------------------------------
// создание класса – демонстратора
 class EventDemo
  {
 public static void Main()   {
 // создание экземпляра класса – издателя
 MyEvent evt = new MyEvent();
// добавление обработчика события в общий список
 evt.SomeEvent += new MyEventHandler(Resiver.handler);
// зажигание события
evt.FireSomeEvent();
  }    // конец определения метода Main

}   // конец класса – демонстратора
```

Данный программный проект содержит один класс (Демонстратор) типа «модуль» и три класса типа «Абстрактный тип данных» (класс «ресивер», класс «издатель» и специализированный класс «делегат»).

Программа объектно-ориентированная. В этой программе объектами являются ресивер и событие. Но класс ресивер имеет статический обработчик события – для его использования не нужно создавать объект этого класса. А вот событие является объектом класса-издателя. Причём, объектом не статическим, поэтому для обеспечения доступа к нему создаётся объект evt.

Все действия с объектами в этой программе производятся в методе main.

Простейшим способом ухода от описанного нарушения принципа инкапсуляции является преобразование программы: если при работе в Visual Studio обработчики событий не создавать в классе Form1, а вынести их, например, в класс – модуль, то нарушения инкапсуляции не происходит:

Form1.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace WindowsFormsApplication8
{
   public partial class Form1 : Form
   {
      public Form1()
      {
         InitializeComponent();
      }
   }
}
```

Program.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Windows.Forms;

namespace WindowsFormsApplication8
{
   static class Program
   {
      static Form1 f1;
      static Form2 f2;

      public static Stack st;

      delegate void MyEventHandler();
//=================================================================
```

```
        [STAThread]
        static void Main()
        {
            f1 = new Form1();
            f1.button1.Click += new EventHandler(button1_Click);
            f1.button2.Click += new EventHandler(button2_Click);
            Application.EnableVisualStyles();
            Application.Run(f1);
        }
//--------------------------------------------------------------
       //Коллекция обработчиков событий:
        static void button2_Click(object sender, EventArgs e)
        {
            f2 = new Form2();
            f2.Show();
            f2.label1.Visible = false;
            f2.label2.Visible = false;
            f2.button1.Click += new EventHandler(button21_Click);
        }

        static void button1_Click(object sender, EventArgs e)
        {
            f2 = new Form2();
            f2.label2.Visible = false;
            f2.label3.Visible = false;
            f2.Show();
            f2.button1.Click += new EventHandler(button11_Click);
        }
    }
  }
```

Но при таком преобразовании программы возникают другие проблемы, относиться к которым нужно критически. Часть из них может быть выражена следующим образом:

1. Ряд проблем сформулирован в книге О.Герман, Ю.Герман «Программирование на Java и C# для студента», С-П., «БХВ – Петербург», 2005г. ISBN 5-94157-710-9:

> Вообще, из объектно-ориентированного программирования только наследование должно пониматься по-настоящему глубоко. И вряд ли студентом будут активно использоваться инкапсуляция и полиморфизм, ибо он не является профессиональным программистом. Инкапсуляция связана с ограничением доступа к членам классов (что, вообще говоря, никак не влияет на работоспособность создаваемых программ), а полиморфизм — с использованием одноименных методов, но с различными типами аргументов и, вообще говоря, различным их числом. Однако всегда можно назвать методы разными именами, так что такое ухищрение, как полиморфизм, — лишь более удобный способ записи функциональности программы.

Другие проблемы высказываются студентами:

2. Например: при объектно-ориентированном программировании все классы являются чисто конструктивными элементами (модулями). Классы типа «Абстрактный

тип данных» создавать необязательно, тем более, что когда программа состоит из большого количества типов объектов, она становится труднореализуемой.

3. В некоторых случаях трудно определить состав объектов, провести объектную декомпозицию. И создаётся впечатление, что объектно-ориентированная программа в принципе нереализуема.

При обучении объектно-ориентированному программированию оказалось, что нечёткое освоение некоторых особенностей языка приводит к появлению специфических ошибок. Для их устранения необходимо иметь в виду:

1. если класс предназначен для визуализации элементов управления, то в нём не должны содержаться обрабатываемые этими элементами данные, не должны содержаться методы их получения и хранения в файловой системе, методы их преобразования – для этого должны создаваться другие классы. Согласно принципам ООП смешивать данные и методы их обработки для разных типов объектов в одном классе недопустимо.

2. Использование интерактивного интерфейса с размещением на экране различных элементов управления и компонент приводит к созданию асинхронной программы. Каждый компонент или элемент управления желательно создавать в виде объекта соответствующего класса. Для визуального отображения элементов управления создаётся специальный класс Windows.Forms.

3. Использование Visual Studio для создания асинхронных программ предусматривает автоматическое описание объектов для используемых компонент и элементов управления, размещаемых на форме в специальной зоне класса Form1. В их число могут входить и обработчики событий, возникающих при активизации находящихся на форме элементов управления. Получается, что в классе Form1 размещаются несвойственные для него элементы, которые не являются необходимыми и достаточными для класса, основное назначение которого – визуальное отображение компонент и элементов управления.

4. Создание объектов и их взаимодействие должно быть в основном сосредоточено в методе Main(), который обычно VS размещает в файле Program.cs, создавая для этого класс «static class Program».

Литература.

1. Г.Шилдт «Полный справочник по C#.pdf».
2. Библиотека MSDN на русском языке: http://msdn.microsoft.com/ru-ru/library

ОДИН ИЗ ПОДХОДОВ ПРОВЕРКИ ТРАЕКТОРИЙ ВЫПОЛНЕНИЯ УЧЕБНЫХ ЗАДАНИЙ

Козлов О. А., Сердюков В. И., Садков Е. В.
ФГНУ Институт информатизации образования Российской академии образования (ФГНУ ИИО РАО), г. Москва

В статье рассматривается один из подходов проверки траекторий выполнения учебных заданий, для решения которых обучаемый выполняет последовательно несколько шагов (действий). Для оценки результатов решения задачипомимо конечного ответа, анализируются полученные промежуточные результатыи траектория процесса решения.