Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Кафедра электронных вычислительных машин

А. В. Отвагин, Н. А. Павлёнок

ПЛАТФОРМА MICROSOFT.NET

Лабораторный практикум
по дисциплине «Объектно-ориентированное
проектирование и программирование»
для студентов специальности 1-40 02 01
«Вычислительные машины, системы и сети»
всех форм обучения

Минск БГУИР 2011

УДК 004.42(076.5)
ББК 32.973.26-018.2я73
О-80

Р е ц е н з е н т:
ведущий научный сотрудник лаборатории №222
Объединенного института проблем информатики НАН Беларуси,
кандидат технических наук А. А. Дудкин

Отвагин, А. В.

О-80 Платформа Microsoft.NET : лаб. практикум по дисц. «Объектно-ориентированное проектирование и программирование» для студ. спец. 1-40 02 01 «Вычислительные машины, системы и сети» всех форм обуч. / А. В. Отвагин, Н. А. Павлёнок. – Минск : БГУИР, 2011. – 46 с. : ил.

ISBN 978-985-488-694-7.

Содержит методические указания и задания к лабораторным работам, предусмотренным учебной программой дисциплины «Объектно-ориентированное проектирование и программирование». Рассмотрены основные средства и библиотеки разработки программ, используемые в платформе Microsoft.NET, приведены примеры использования возможностей платформы Microsoft.NET. Практикум рассчитан на студентов всех форм обучения.

УДК 004.42(076.5)
ББК 32.973.26-018.2я73

ISBN 978-985-488-694-7

СОДЕРЖАНИЕ

Лабораторная работа №1
Знакомство с системой программирования Microsoft Visual Studio .NET

Цель работы: изучить возможности и пользовательский интерфейс системы программирования Microsoft Visual Studio .NET, а также возможности программной платформы .NET Framework по разработке приложений различных предметных областей. Создать минимальное консольное приложение, изучить структуру и основные элементы его кода.

1.1. Основные сведения о системе Microsoft Visual Studio .NET

Microsoft Visual Studio .NET – это интегрированная среда разработки (Integrated Development Environment (IDE)) для создания, документирования, запуска и отладки программ, написанных на языках платформы .NET Framework.

Как и любая другая хорошая среда разработки, Visual Studio .NET включает средства управления проектами, редактор исходного текста, конструкторы пользовательского интерфейса, мастера создания приложений, компиляторы, компоновщики, инструменты, утилиты, документацию и отладчики. Она позволяет создавать приложения для 32- и 64-разрядных Windows-платформ, а также новой платформы .NET Framework. Одно из важнейших усовершенствований – возможность работы с разными языками в единой среде разработки.

Microsoft также предоставляет новый набор инструментов – .NET Framework SDK. Он распространяется бесплатно и включает компилятор для всех языков, множество утилит и документацию. С помощью этого SDK можно создавать приложения для .NET Framework без Visual Studio NET.

В семействе продуктов Visual Studio используется единая интегрированная среда разработки (IDE), состоящая из нескольких элементов: строки меню, панели инструментов «Стандартная», различных закрепленных или автоматически скрываемых окон инструментов в левой, нижней или правой областях, а также области редакторов. Набор доступных окон инструментов, меню и панелей инструментов зависит от типа проекта или файла, в котором выполняется разработка. Расположение окон инструментов и других элементов интегрированной среды разработки может изменяться в зависимости от примененных параметров и настроек, выполняемых пользователем в процессе работы.

Выбор используемых редакторов и конструкторов зависит от типа создаваемого файла или документа. Редактор текста – это основной текстовый процессор интегрированной среды разработки, а редактор кода – основной редактор исходного кода.

Другие редакторы, такие, как редактор таблиц CSS, конструктор HTML и конструктор веб-страниц, совместно используют целый ряд возможностей редактора кода, но обладают и дополнительными средствами, связанными с поддерживаемыми ими типами кода или разметки. В редакторах и конструкторах, как правило, используется два представления: графическое представление кон-

структора и представление связанного кода или исходного кода. Представление конструктора позволяет определить расположение элементов управления и других объектов пользовательского интерфейса или веб-страницы. Элементы управления можно легко перемещать из панели элементов и располагать в рабочей области конструирования.

В среде Visual Studio предусмотрен мощный набор средств построения и отладки. Благодаря конфигурациям построения можно выбирать компоненты для построения, исключать компоненты, которые не требуется включать в построение, а также определять, как будут построены выбранные проекты и для какой платформы. Конфигурации построений доступны как для решений, так и для проектов.

1.2. Основные сведения о .NET Framework

Платформа .NET Framework – это интегрированный компонент Windows, который поддерживает создание и выполнение нового поколения приложений и веб-служб XML.

Двумя основными компонентами платформы .NET Framework являются общеязыковая среда выполнения (CLR) и библиотека классов .NET Framework. Основой платформы .NET Framework является среда CLR. Среду выполнения можно считать агентом, который управляет кодом во время выполнения и предоставляет основные службы, такие, как управление памятью, управление потоками и удаленное взаимодействие. При этом накладываются условия строгой типизации и другие виды проверки точности кода, обеспечивающие безопасность и надежность. Фактически основной задачей среды выполнения является управление кодом. Код, который обращается к среде выполнения, называют управляемым кодом, а код, который не обращается к среде выполнения, называют неуправляемым кодом. Другой основной компонент платформы .NET Framework, библиотека классов, представляет полную объектно-ориентированную коллекцию типов, которые применяются для разработки приложений, начиная от обычных, запускаемых из командной строки или с графическим интерфейсом пользователя, и заканчивая приложениями, использующими последние технологические возможности ASP.NET, такие, как Web Forms и веб-службы XML.

Библиотека классов платформы .NET Framework представляет собой коллекцию типов, которые тесно интегрируются со средой CLR. Библиотека классов является объектно-ориентированной и предоставляет типы, из которых управляемый код пользователя может наследовать функции. Это не только упрощает работу с типами .NET Framework, но также уменьшает время, затрачиваемое на изучение новых средств платформы .NET Framework. Кроме того, компоненты независимых производителей можно легко объединять с классами.

Как и ожидается от объектно-ориентированной библиотеки классов, типы .NET Framework позволяют решать типовые задачи программирования, включая работу со строками, сбор данных, подключения к базам данных и доступ к файлам. В дополнение к этим обычным задачам библиотека классов содержит

типы, поддерживающие многие специализированные сценарии разработки. Например, можно использовать платформу .NET Framework для разработки следующих типов приложений и служб:

– консольные приложения;

– приложения с графическим интерфейсом пользователя Windows (Windows Forms);

– приложения ASP.NET;

– веб-службы.

Архитектура платформы .NET Framework приведена на рис. 1.1.

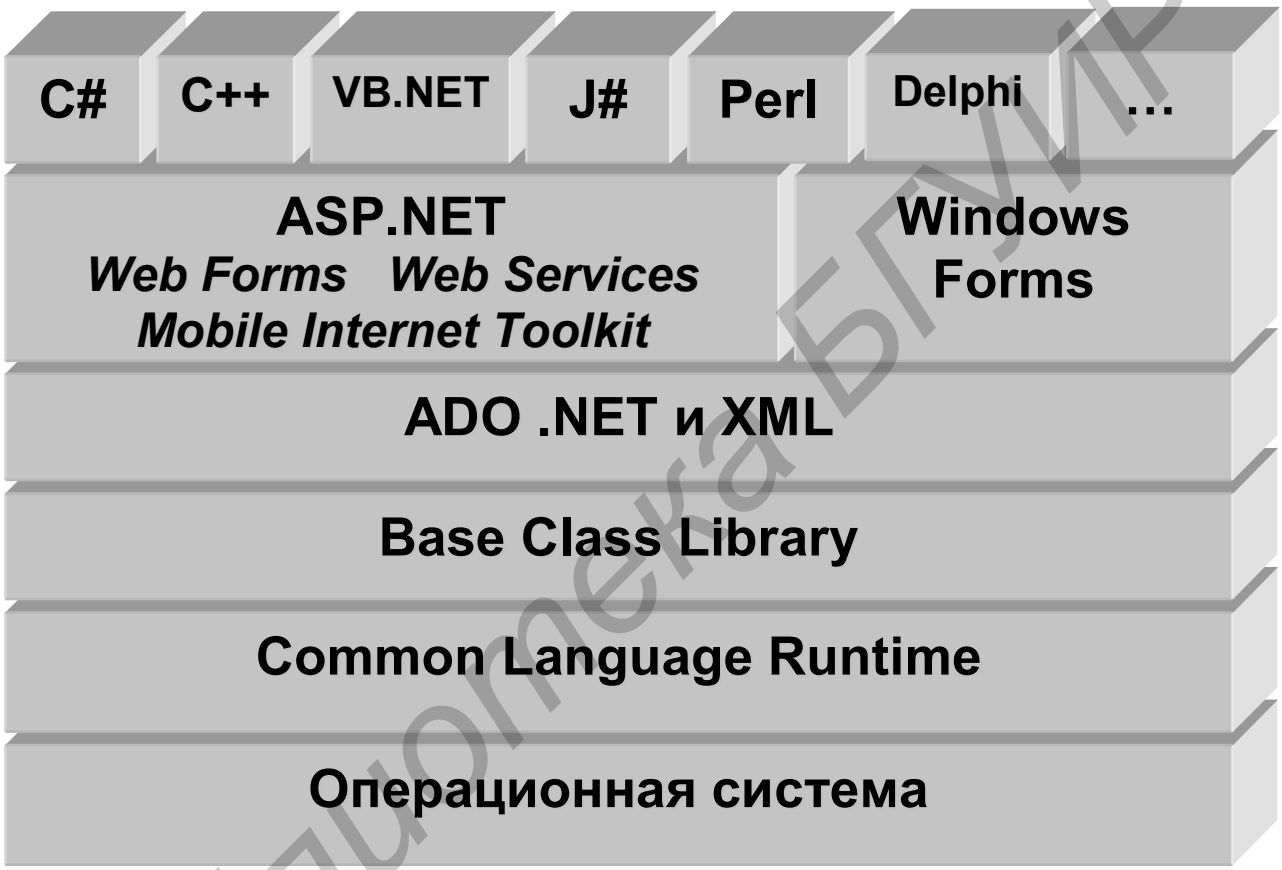

Рис. 1.1. Архитектура платформы .NET Framework

В состав платформы входят следующие компоненты (снизу вверх):

– Common Language Runtime (CLR) – среда управляемого исполнения кода .NET, реализуемая виртуальной машиной;

– Base Class Library – библиотека классов, используемая при создании пользовательских приложений. В ее состав входят средства ввода-вывода, сетевого взаимодействия, работа со строками, коллекциями объектов, средствами интернационализации и т. д.;

– ADO .NET и XML – средства обеспечения доступа к системам управления базами данных и обработки структурированной информации, представленной на языке XML;

– ASP.NET – средства разработки приложений, размещаемых в сети Интернет и реализованных в виде распределенных сервис-ориентированных архитектур. Сюда же входят средства организации веб-интерфейса пользователя;

– Windows Forms – средства разработки графического интерфейса пользователя в приложениях. Здесь расположены элементы управления, размещаемые в пользовательских формах, обеспечивающих пользователю удобство работы с прикладными программами;

– языки программирования, поддерживающие спецификации .NET Framework и гарантирующие межъязыковую совместимость кода приложения. Различные языки введены для удобства пользователя.

1.3. Разработка простого консольного приложения

Большинство типов приложений в Microsoft Visual Studio .NET может быть создано с использованием так называемого мастера – сценария, исполняемого средой разработки для создания структуры и заготовок кода приложений. Существует множество мастеров, применяемых для создания практически любых типов приложений.

В меню «Файл» выберите пункт «Новый проект». Появится окно, в котором будут перечислены все известные мастера, которые среда Microsoft Visual Studio .NET предоставляет для создания приложений различных типов. Для создания консольного приложения следует выбрать категорию мастеров Visual C# и в данной категории указать «Консольное приложение». Затем следует указать имя проекта и нажать OK.

После завершения работы мастер создает заготовку кода, приведенную на рис. 1.2.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Рис. 1.2. Заготовка кода консольного приложения

Как видно из примера, приведенный код не выполняет никаких действий, он лишь содержит объявление используемых системных библиотек, а также за-

готовку класса Program со статическим методом Main, являющимся точкой входа в приложение.

Расширим этот пример стандартным сообщением «Hello, World!», выводимым в консоль. Для этого код дополняется следующими строками:

```
Console.WriteLine("Hello World!");
Console.ReadLine();
```

Для того чтобы собрать приложение, следует выбрать пункт меню «Сборка» (Build), а в нем команду «Собрать решение» (Build Solution). После построения программы компилятор сообщит о завершении или укажет ошибки в тексте программы.

Теперь изменим пример так, чтобы он мог обрабатывать аргументы командной строки. Для этого метод Main изменим следующим образом:

```
static void Main(string[] args)
{
    Console.WriteLine("Аргументы командной строки");
    for (int i = 0; i < args.Length; i++)
        Console.WriteLine("Аргумент: {0} ", args[i]);
}
```

Поскольку все аргументы командной строки передаются в виде строк, иногда требуется разбирать значения строк и приводить их к определенному числовому типу. Все примитивные типы C# имеют соответствующие реализации методов приведения строки в значение типа, в частности метод Parse(). Пример получения значения типа с плавающей запятой двойной точности из строки приведен ниже:

```
string s = "17,345,342.38";
double value = Double.Parse(s, NumberStyles.Number,
    new CultureInfo("en-US"));
```

Созданный код приложения для платформы .NET можно проанализировать с помощью утилиты ildasm. Для этого нужно запустить ildasm и открыть полученный исполняемый файл приложения. Ildasm показывает структуру кода в виде дерева функций, а при выборе отдельной функции можно увидеть код на языке CIL (Common Intermediate Language). Например, конструктор по умолчанию для приведенного приложения выглядит так:

```
.method public hidebysig specialname rtspecialname
       instance void  .ctor() cil managed
{
  // Code size       7 (0x7)
  .maxstack  8
  IL_0000:  ldarg.0
  IL_0001:  call       instance void [mscor-
lib]System.Object::.ctor()
  IL_0006:  ret
} // end of method Program::.ctor
```

1.4. Задание к лабораторной работе

В соответствии с вариантом необходимо написать простейшую программу (аналогичную сгенерированной автоматически), в которой помимо Main будет функция по варианту. Ввод чисел осуществить через командную строку.

Варианты:

1. Вычисление суммы двух вещественных чисел.
2. Вычисление разности двух вещественных чисел.
3. Вычисление произведения двух вещественных чисел.
4. Вычисление частного двух вещественных чисел (* с проверкой на ненулевой делитель).
5. Вычисление суммы квадратов двух вещественных чисел.
6. Вычисление разности квадратов двух вещественных чисел.
7. Вычисление произведения квадратов двух вещественных чисел.
8. Вычисление частного квадратов двух вещественных чисел (* с проверкой на ненулевой делитель).
9. Вычисление квадрата суммы двух вещественных чисел.
10. Вычисление квадрата разности двух вещественных чисел.
11. Вычисление квадрата произведения двух вещественных чисел.
12. Вычисление квадрата частного двух вещественных чисел (* с проверкой на ненулевой делитель).
13. Возведение первого введённого числа в степень второго (* если первое число отрицательное, степень должна быть целой).

1.5. Контрольные вопросы

1. Какие основные компоненты входят в состав платформы Microsoft .NET Framework?
2. Какие основные элементы входят в состав среды разработки Microsoft Visual Studio .NET?
3. Что такое ildasm, для чего применяется этот инструмент?

Лабораторная работа №2
Генерация и обработка исключений

Цель работы: изучить средства генерации и обработки исключений в программах платформы Microsoft .NET, научиться создавать собственные классы исключений для обработки нештатных ситуаций в процессе исполнения программ.

2.1. Понятие исключения

Во время работы программы могут возникать ситуации, выходящие за пределы, предусмотренные спецификацией ее поведения. Такие ситуации называются исключительными. Обработка исключительных ситуаций – процесс, направленный на достижение устойчивости функционирования программы, т. е. сохранения способности программной системы должным образом реагировать на исключительные ситуации и принимать меры к уменьшению их влияния на решение задачи.

Проблема обработки нештатных ситуаций характерна для всех программных систем, при этом средства обработки могут быть различными и применяться на разных стадиях разработки или исполнения программы. Для эффективного использования в процессе отладки необходимо предусмотреть средства локализации и вывода сообщений о возникших ошибках. В платформе .NET Framework для этого предусмотрен механизм исключений и обработчиков исключений.

Кроме наличия механизма, позволяющего определить момент наступления исключения и отреагировать на него должным образом, важным моментом является то, что платформа .NET Framework сама по себе содержит развитый механизм обработки стандартных исключительных ситуаций, чаще встречающихся при работе приложений. В случае отсутствия стандартного обработчика исключений программист может самостоятельно определить класс исключения и использовать его для сохранения устойчивого функционирования программы.

В C# исключение – это объект, созданный (сгенерированный*)* при наступлении определенной исключительной ошибочной ситуации. Этот объект содержит информацию, которая может помочь отследить причину возникновения проблемы. В случае пользовательских классов исключений в данный объект может быть внесена специальная информация, помогающая корректно определить проблему и найти эффективные методы ее разрешения.

2.2. Принципы обработки исключительных ситуаций

Язык C# наследовал схему исключений языка C++, внеся в нее свои коррективы. Рассмотрим схему подробнее и начнем с синтаксиса конструкции, применяемой для оформления блоков перехвата и обработки исключений try-catch-finally:

```
try {...}
catch (T1 e1) {...}
...
catch(Tk ek) {...}
finally {...}
```

В любом месте текста программного модуля, где синтаксически допускается использование блока перехвата и обработки исключения, этот блок можно сделать охраняемым, добавив ключевое слово try. Внутри блока try возможно возникновение одного или нескольких типов исключений. Вслед за try-блоком могут следовать catch-блоки, называемые блоками-обработчиками исключительных ситуаций, их может быть несколько, но они могут и отсутствовать. Завершает эту последовательность finally-блок – блок финализации, который также может отсутствовать. Вся эта конструкция может быть вложенной: в состав try-блока может входить конструкция try-catch-finally.

Рассмотрим назначение блоков:

– try-блок инкапсулирует код, формирующий часть нормальных действий программы, которая, тем не менее, потенциально может привести к появлению серьезных ошибочных ситуаций. Например, в этом блоке могут выполняться операции с файлами или какими-либо системными ресурсами;

– catch-блок инкапсулирует код, который обрабатывает ошибочные ситуации, происходящие в коде блока try. Это также удобное место для протоколирования ошибок. В границах этого блока находится сам обработчик исключительной ситуации или обращение к нему;

– finally-блок инкапсулирует код, очищающий любые ресурсы или выполняющий другие действия, которые обычно нужно выполнить в конце блоков try или catch. Важно понимать, что этот блок выполняется независимо от того, было возбуждено исключение или нет. Поскольку целью блока finally является выполнение кода очистки, который должен быть выполнен в любом случае, если поместить внутрь блока finally оператор return, то компилятор выдаст ошибку. Например, в блоке finally можно закрыть любое соединение, которое было открыто в блоке try. Важно также понимать, что блок finally необязателен. Если нет необходимости в очистке кода (таких как закрытие любых открытых объектов, например, файлов), то в этом блоке тоже нет необходимости.

Рассмотрим код, иллюстрирующий перехват исключения при преобразовании значения типа данных из строки (рис. 2.1). Этот код использует стандартные классы исключения `FormatException и OverflowException`, чтобы обработать ошибки, которые могут возникнуть при вводе информации пользователем. В данном случае программа пытается преобразовать число, введенное пользователем с клавиатуры, в соответствующее ему значение. При вводе пользователь может допустить как ошибки формата числа, например, ввести букву вместо цифры, так и ошибки, связанные с вводом слишком большого числа, не соответствующего целочисленному типу. В обоих случаях сработает блок пе-

рехвата соответствующего исключения и пользователь получит информативное сообщение об ошибке.

```
static void Main(string[] args)
{
    string s = Console.ReadLine();
    try
    {
        Int32.Parse(s);
        Console.WriteLine(
          "Вы ввели правильное значение типа Int32 = {0}.", s);
    }
    catch (FormatException)
    {
        Console.WriteLine("Неправильный формат числа!");
    }
    catch (OverflowException)
    {
        Console.WriteLine(
            "Слишком большое число для Int32!");
    }
}
```

Рис. 2.1. Пример обработки исключения

Все исключения в платформе .NET Framework формируют иерархию. Базовым классом для любых исключений в CLR является класс System.Exception. Класс System.Exception содержит информацию о произошедшей ошибке или нештатной ситуации:

– Message – текстовое описание ошибки;

– StackTrace – текстовая визуализация состояния стека в момент возникновения исключения;

– InnerException – исключение, которое является причиной возникновения текущего исключения.

Все исключения в .NET наследуют базовый класс System.Exception. Системные исключения наследуют класс System.SystemException, например System.NullReferenceException, System.ArgumentException, System.StackOverflowException. Исключения, определенные пользователем, должны наследовать класс System.ApplicationException (рис. 2.2).

При перехвате исключений данного класса перехватываются и исключения всех его наследников. Например, конструкция:

```
try
{
    // Выполнить вычисления с арифметической ошибкой
}
catch (System.ArithmeticException)
{
    // Обработать арифметическую ошибку
}
```

перехватывает само исключение ArithmeticException и исключения-наследники DivideByZeroException и OverflowException. Поэтому необходимо очень внимательно следить за порядком блоков catch и перехватываемыми ими исключениями.

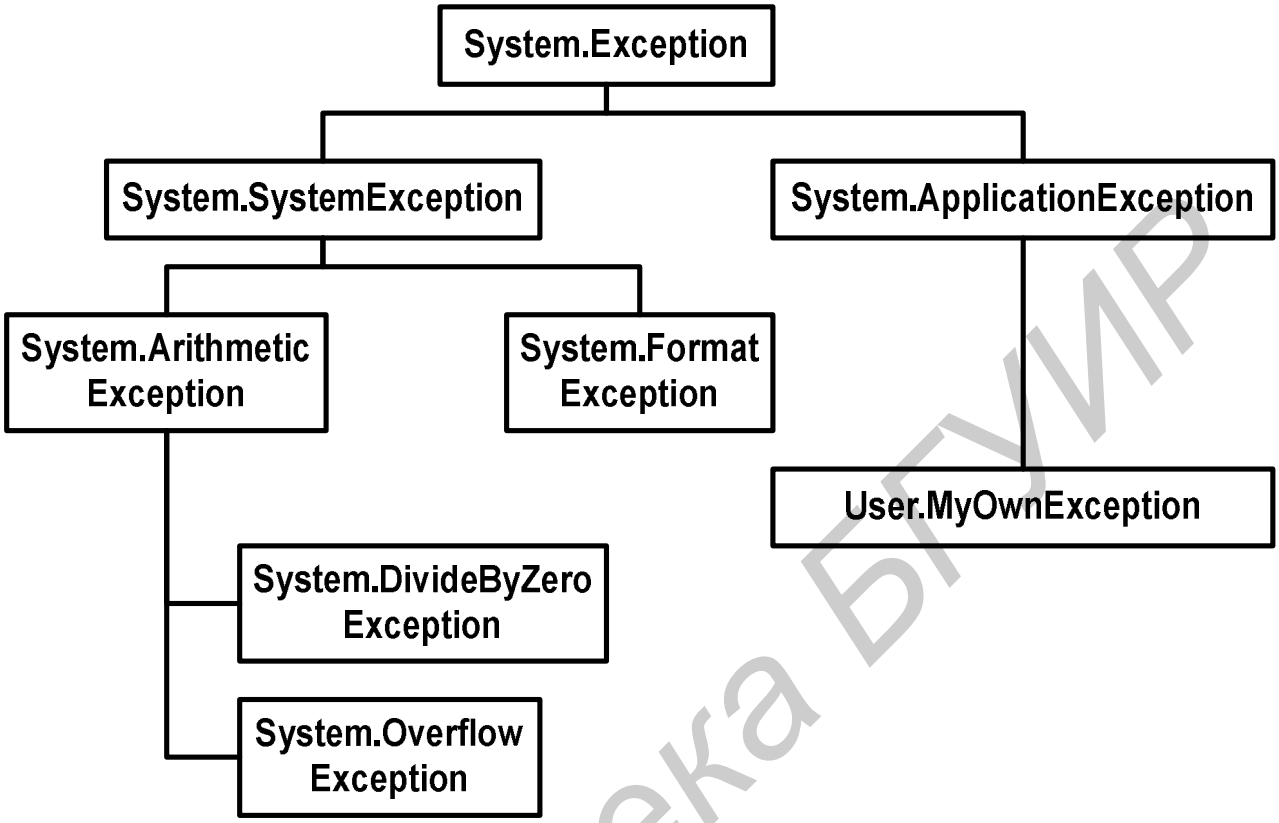

Рис. 2.2. Иерархия исключений

2.3. Создание собственных классов и генерация исключений

При использовании исключений разработчик может создать код для реакции на многие стандартные ошибки. Однако, в сложных программах ошибки порождаются не только кодом, но и логикой самой программы. Поэтому для более эффективной реакции на потенциальные ошибки создаются собственные классы исключений, которые содержат специализированную информацию о произошедшем событии и дают возможность отреагировать на него с максимальной пользой.

Рассмотрим пример генерации исключения стандартного класса. Такие исключения можно использовать в программе, чтобы выполнить стандартную обработку ошибок, которая может быть реализована в других библиотеках платформы. Аналогичным образом генерируются исключения, созданные самим пользователем.

В данном примере производится контроль диапазона значений для выполнения операции извлечения квадратного корня. В случае нарушения границ диапазона порождается исключение, которое обрабатывается программой. Для этого используется служебное слово `throw`. Данное исключение с тем же ус-

пехом может быть передано в другой модуль, вызвавший функцию вычисления квадратного корня, и обработано вызывающим модулем. В этом случае необходимо явно указать вызывающему модулю возможность появления исключения.

```
public static double Sqrt(double aValue)
{
    if (aValue < 0)
        throw new System.ArgumentOutOfRangeException(
            "Функция не выполняется для отрицательных чисел!");
    return Math.Sqrt(aValue);
}

static void Main(string[] args)
{
    try
    {
        Sqrt(-1);
    }
    catch (ArgumentOutOfRangeException ex)
    {
        Console.Error.WriteLine("Error: " + ex.Message);
        throw;
    }
}
```

Здесь порождается стандартное исключение класса `System.ArgumentOutOfRangeException`, в котором задается собственное сообщение пользователя, поясняющее причину ошибки.

Для определения собственного исключения наследуется класс System.ApplicationException, для которого создаются подходящие конструкторы. Ему также можно добавить дополнительные свойства, дающие информацию о возникшей проблеме (информационное сообщение, информацию о причинах исключения в виде значений некоторых параметров).

```
class ParseFileException : ApplicationException
{
    private string mFileName;
    private long mLineNumber;
    public string FileName
    {
        get
        {
            return mFileName;
        }
    }
    public long LineNumber
    {
        get
        {
            return mLineNumber;
        }
    }
    public ParseFileException(string aMessage,
```

```
            string aFileName, long aLineNumber,
            Exception aCauseException) : base(aMessage,
            aCauseException)
        {
            mFileName = aFileName;
            mLineNumber = aLineNumber;
        }
        public ParseFileException(string aMessage, string
            aFileName, Exception aCauseException) : this(
            aMessage, aFileName, 0, aCauseException)
        {
        }
        public ParseFileException(string aMessage,
            string aFileName) : this(aMessage, aFileName, null)
        {
        }
}
```

В приведенном выше примере определен класс исключения, возникающего при разборе файла. Это исключение содержит информацию о причине его возникновения, а также параметры, указывающие, в каком месте файла возникла ошибка разбора. Пример использования исключения приведен ниже:

```
static long CalculateSumOfLines(string aFileName)
{
    StreamReader inF;
    try
    {
        inF = File.OpenText(aFileName);
    }
    catch (IOException ioe)
    {
        throw new ParseFileException(String.Format(
            "Файл {0} не открыт для чтения.",
            aFileName), aFileName, ioe);
    }
    try
    {
        long sum = 0;
        long lineNumber = 0;
        while (true)
        {
            lineNumber++;
            string line;
            try
            {
                line = inF.ReadLine();
            }
            catch (IOException ioe)
            {
                throw new ParseFileException(
                    "Ошибка чтения файла.",
                    aFileName, lineNumber, ioe);
```

```
                }

                if (line == null)
                    break;  // конец файла

                try
                {
                    sum += Int32.Parse(line);
                }
catch (SystemException se)
            {
                throw new ParseFileException(String.Format(
                    "Ошибка разбора строки '{0}'.", line),
                    aFileName, lineNumber, se);
            }
        }
        return sum;
    }
    finally
    {
        inF.Close();
    }
}
static void Main(string[] args)
{
    long sumOfLines = CalculateSumOfLines(@"c:\test.txt");
    Console.WriteLine("Количество строк={0}", sumOfLines);
}
```

Данная программа подсчитывает количество строк в текстовом файле.

2.4. Задание к лабораторной работе

Создайте программу, содержащую обработку системных и собственных типов исключений. Искусственно сгенерируйте исключения, проиллюстрируйте срабатывание обработчиков. Реализуйте несколько обработчиков по иерархии типов исключений, покажите, как влияет тип обработчика на порядок обработки исключений.

Варианты:

1. Программа обработки файлов (поиск, подсчет слов, фраз, символов).
2. Программа арифметических действий (переполнение, контроль диапазона значений, преобразование типов).

2.5. Контрольные вопросы

1. Что такое исключение, для чего используются исключения?
2. Как реализуется обработка исключений?
3. Что такое иерархия исключений, как она влияет на обработку?
4. Как реализуются собственные классы исключений?

Лабораторная работа №3
Асинхронное программирование и потоки

Цель работы: изучить средства организации асинхронной многопоточной обработки информации в платформе Microsoft.NET, возможности работы с потоками и примитивами синхронизации.

3.1. Понятие потоков, возможности многопоточного программирования

Развитие аппаратных средств и режимов их работы привело к появлению новых возможностей по организации эффективной работы вычислительных ресурсов. Новые средства мультипроцессирования позволяют решить сложные задачи организации вычислительного процесса:

1. Приложение должно выполнять длительные операции обработки данных или ожидать освобождения некоторого ресурса.
2. Приложение должно выполнять операции в фоновом режиме, одновременно обслуживая пользовательский интерфейс.
3. Нужен механизм, позволяющий нескольким операциям выполняться одновременно с целью повышения степени использования компьютера и ускорения вычислений.

Поток является единицей диспетчеризации при обработке данных, т. е. его исполнение подвергается планированию со стороны операционной системы. Многозадачность – это одновременное исполнение нескольких потоков. Существует два вида многозадачности – кооперативная (cooperative) и вытесняющая (preemptive). Платформа .NET работает только в системах с вытесняющей многозадачностью.

При вытесняющей многозадачности процессор отвечает за выделение каждому потоку определенного количества времени, в течение которого поток может выполняться, – кванта времени (timeslice). Далее процессор переключается между разными потоками, выдавая каждому потоку его квант времени, а программист не заботится о том, как и когда возвращать управление, в результате чего могут работать и другие потоки. В случае вытесняющей многозадачности на однопроцессорной машине в любой момент времени реально будет исполняться только один поток. Поскольку интервалы между переключениями процессора от процесса к процессу очень малы (в пределах микросекунд), возникает иллюзия многозадачности. Чтобы несколько потоков на самом деле работали одновременно, потребуется многопроцессорная машина.

Преимущества использования потоков состоят в повышении производительности, ускорении вычислительного процесса при решении задачи, обеспечении интерактивного режима работы, а также режимов реального времени. Использование потоков дает возможность эквивалентного разделения вычислительной мощности процессора между приложениями, возможность сокращения простоев и стоимости обработки информации.

3.2. Создание потоков, свойства потоков

Потоки выполнения в платформе Microsoft.NET создаются на основе системного класса System.Thread, экземпляры которого соответствуют реальным потокам исполнения в рамках операционной системы. Рассмотрим пример создания потоков исполнения:

```
using System;
using System.Threading;

class SimpleThreadApp {
   public static void WorkerThreadMethodQ
   {
   Console.WriteLine("Запущен рабочий поток");
   }

   public static void Main() {
   ThreadStart worker = new ThreadStart(WorkerThreadMethod);
   Console.WriteLine("Main – Создаем рабочий поток");
   Thread t = new Thread(worker); t.StartQ;
   Console.WriteLine("Main – Рабочий поток запущен");
   }
}
```

Скомпилировав и запустив это приложение, можно заметить, что сообщение метода Main выводится перед сообщением рабочего потока. Это доказывает, что рабочий поток действительно работает асинхронно. Пространство имен System. Threading содержит классы, необходимые для организации потоков в среде .NET.

```
ThreadStart WorkerThreadMethod =
     new ThreadStart(WorkerThreadMethod);
```

`ThreadStart` — это делегат. Он должен быть задействован при создании нового потока. Он используется, чтобы задать метод, который должен вызываться как метод потока. Здесь создаётся экземпляр объекта `Thread`, при этом конструктор принимает в качестве аргумента только делегат `ThreadStart`:

```
Thread t = new Thread(worker);
```

После этого вызывается метод `Start` объекта `Thread`, в результате чего вызывается метод `WorkerThreadMethod`.

На рис. 3.1. показан жизненный цикл потока и смена его состояний. Поток начинает существование в состоянии `Start`. В процессе работы он может самостоятельно или под влиянием других потоков перейти в состояние приостановки (`Suspend`, `Sleep`, `WaitX`), завершиться (`Abort`) или слиться (`Join`) с другим потоком. Из этих состояний он может быть выведен по требованию программы (`Resume`), по истечении времени (`Expire Time`), по прерыванию (`Interrupt`) или по сигналу (`Notified`). После завершения работы поток прекращается (`All Done`).

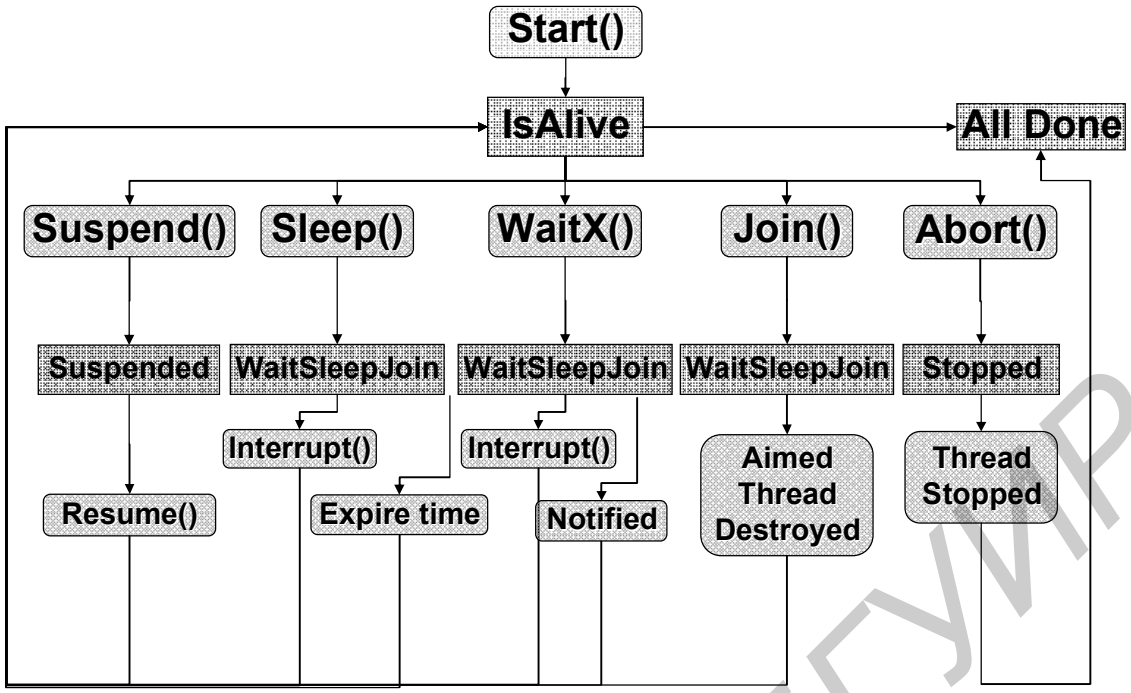

Рис. 3.1. Жизненный цикл потока исполнения

Каждый поток обладает набором свойств, позволяющим определить его состояние или отличить его от других потоков:

`IsAlive` – указывает, что поток существует;

`IsBackground` – указывает, что это фоновый поток;

`IsThreadPoolThread` – указывает, что поток относится к пулу потоков;

`Name` – указывает имя потока;

`Priority` – указывает и устанавливает приоритет потока;

`ThreadState` – указывает состояние потока.

3.3. Планирование потоков, методы синхронизации

При переключении процессора по окончании выделенного потоку кванта времени процесс выбора следующего потока, предназначенного для исполнения, далеко не произволен. У каждого потока есть приоритет, указывающий процессору, как должно планироваться выполнение этого потока по отношению к другим потокам системы. Для потоков, создаваемых в период выполнения, уровень приоритета по умолчанию равен `Normal`. Для просмотра и установления этого значения служит свойство `Thread.Priority`. Установщик свойства `Thread.Priority` принимает аргумент типа `Thread.ThreadPriority`, который представляет собой перечислимый тип, имеющий значения `Highest`, `AboveNormal`, `Normal`, `BelowNormal` и `Lowest`.

Чтобы проиллюстрировать, как приоритет влияет на код, рассмотрим пример, в котором один рабочий поток считает от 1 до 10, а другой – от 11 до 20.

```
using System;
using System.Threading;
class ThreadSchedulelApp
{
    public static void WorkerThreadMethod1(object param)
    {
        int start = (int)param;
        Console.WriteLine("Worker thread started");
        Console.WriteLine
        ("Worker thread - counting slowly from 1 to 10");
        for (int i = start; i < start + 10; i++)
        {
            // Код, который имитирует работу
            Console.Write(".");
            Thread.Sleep(500);
            Console.Write("{0}", i);
        }
        Console.WriteLine("Worker thread finished");
    }
    public static void Main()
    {
        Thread worker1 = new Thread(WorkerThreadMethod1);
        Thread worker2 = new Thread(WorkerThreadMethod1);
        Console.WriteLine("Main - Creating worker threads");
        worker1.Start(1); worker2.Start(11);
    }
}
```

Обратите внимание на то, что метод, реализующий работу потока, принимает параметр param, позволяющий передать в поток параметр, на основе которого поток строит свое выполнение. В данном случае параметр задает начальное число для счетчика. Примерный результат работы программы показан ниже:

```
Main - Creating worker threads
Worker thread started
Worker thread started
.1.11.12.2.13.3.14.4.15.5.16.6.17.7.8.18.19.9.20.10
Worker thread finished
Worker thread finished
```

В зависимости от поведения диспетчера порядок цифр может изменяться. Теперь изменим свойство Priority каждого потока, как это сделано в следующем коде. Результаты будут во многом отличаться от предыдущих:

```
Main - Creating worker threads
Worker thread started
Worker thread started
.1.11.2.12.3.13.4.14.5.15.6.16.7.17.8.18.9.19.10.20
Worker thread finished
Worker thread finished
```

В данном случае первый поток имел наивысший приоритет, поэтому он всегда срабатывал первым.

Рассмотрим механизмы явной синхронизации, позволяющие управлять порядком выполнения потоков. Пусть у нас есть программа имитации доступа к банковскому счету, которая позволяет снимать со счета деньги:

```
using System;
using System.Threading;
class Account
{
    public int mBalance;
    public void Withdraw100()
    {
        int oldBalance = mBalance;
        Console.WriteLine("Withdrawing 100...");
        Thread.Sleep(100);
        int newBalance = oldBalance - 100;
        mBalance = newBalance;
    }
    static void Main(string[] args)
    {
        Account acc = new Account();
        acc.mBalance = 500;
        Console.WriteLine("Account balance = {0}",
            acc.mBalance);
        Thread user1 = new Thread(
            new ThreadStart(acc.Withdraw100));
        Thread user2 = new Thread(
            new ThreadStart(acc.Withdraw100));
        user1.Start();
        user2.Start();
        user1.Join();
        user2.Join();
        Console.WriteLine("Account balance = {0}",
            acc.mBalance);
    }
}
```

Результат работы этой программы без синхронизации показан ниже: со счета сняли 200 единиц за 2 транзакции, но баланс скорректировался только на 100.

```
Account balance = 500
Withdrawing 100...
Withdrawing 100...
Account balance = 400
```

Как избежать подобных непредсказуемых состояний? На самом деле, как это обычно бывает в программировании, существует несколько способов решения этой хорошо известной проблемы. Стандартное средство – синхронизация. Синхронизация позволяет задавать критические секции (critical sections) кода, в которые в каждый отдельный момент может входить только один поток, гарантируя, что любые временные недействительные состояния вашего объекта будут невидимы его клиентам. Рассмотрим средство определения критических секций – класс .NET Monitor. Он позволяет оградить критическую секцию кода,

гарантируя ее атомарное исполнение. Пример его использования в функции `Withdraw100` приведен ниже:

```
public void Withdraw100()
{
    Monitor.Enter(this);
    try
    {
        int oldBalance = mBalance;
        Console.WriteLine("Withdrawing 100...");
        Thread.Sleep(100);
        int newBalance = oldBalance - 100;
        mBalance = newBalance;
    }
    finally
    {
        Monitor.Exit(this);
    }
}
```

Теперь действия в границах монитора будут выполняться последовательно для всех потоков, и результат работы программы будет таким:

```
Account balance = 500
Withdrawing 100...
Withdrawing 100...
Account balance = 300
```

3.4. Задание к лабораторной работе

Создать приложение с элементарным графическим интерфейсом. Действия по варианту выполнять во вторичном потоке.

1. Копирование файла (* с прогрессбаром).
2. Поиск вхождения строки в большой файл (* с прогрессбаром).
3. Подсчет размера каталога (со всеми подкаталогами, файлами) (* со строкой состояния, показывающей текущий обрабатываемый файл или подкаталог).
4. Поиск файла по шаблону (? – символ, * – множество символов). (В папке, все подпапки – с возможно большим ветвлением).

3.5. Контрольные вопросы

1. Что такое потоки, для чего их применяют?
2. Какие состояния имеет поток при своем существовании? Чем они отличаются?
3. Для чего нужен приоритет потока, как регулируется приоритет?
4. Что такое синхронизация потоков, какими средствами она выполняется?

Лабораторная работа №4
Создание Web приложений на базе ASP.NET

Цель работы: изучить принципы и средства создания приложений для Интернет на платформе Microsoft ASP.NET, основные элементы приложений, методы их развертывания и исполнения. Создать Web-приложение по индивидуальному заданию.

4.1. Основы Microsoft ASP.NET

Microsoft ASP.NET – это платформа для создания, развертывания и запуска web-сервисов и приложений. Она предоставляет высокопроизводительную, основанную на стандартах многоязыковую среду, которая позволяет решать задачи развертывания и использования интернет-приложений. ASP.NET – это часть технологии .NET, используемая для написания мощных клиент-серверных интернет-приложений. Она позволяет создавать динамические страницы HTML на основе множества готовых элементов управления, применяя которые, можно быстро создавать интерактивные web-сайты.

Основное преимущество технологии ASP.NET состоит в создании динамических веб-страниц. Статическая страница содержит код на языке гипертекстовой разметки HTML. Когда автор страницы пишет ее, он определяет, как будет выглядеть страница для всех пользователей. Содержание страницы будет всегда одинаковым, независимо от того, кто и когда решит ее просмотреть. Языка HTML вполне достаточно для отображения информации, которая редко изменяется и не зависит от того, кто ее просматривает. Страница HTML – простой ASCII-текст, следовательно, клиент может работать в любой операционной системе. Однако, такая страница в силу статической природы будет нести в себе ту же информацию, что была записана в нее в момент создания, и переданные пользователем данные не смогут быть использованы для модификации содержимого отображаемых ему страниц.

Динамическими принято называть web-страницы, которые перед отправкой клиенту проходят цикл обработки на сервере. В самом простом случае это может быть некоторая программа, которая модифицирует запрашиваемые клиентом статические страницы, используя параметры полученного запроса и некоторое хранилище данных. Таким образом, предоставляемая пользователю информация может отображать наиболее актуальное состояние связанной с приложением информационной модели, выполнять интерактивные действия, т. е. представляет собой полноценное приложение, расположенное на сервере в Интернет.

ASP .NET – это концептуально новая технология Microsoft, созданная в рамках идеологии .NET. В ASP .NET заложено все для того, чтобы сделать весь цикл разработки web-приложения более быстрым, а поддержку – более простой. ASP .NET основана на объектно-ориентированной технологии. В ASP .NET используются компилируемые языки. Во время компиляции проверяется

синтаксическая корректность исходного текста. Скомпилированный в промежуточный язык код выполнится быстрее и будет таким же независимо от языка разработки. Компиляция происходит на сервере в момент первого обращения пользователя к странице. Если программист изменил текст страницы, программа перекомпилируется автоматически. При написании кода можно использовать набор компонентов, поставляемых с .NET.

Платформа .NET Framework предоставляет приложениям среду выполнения, непосредственно взаимодействуя с операционной системой. Выше лежит интерфейс ASP .NET-приложений, на котором, в свою очередь, базируются web-формы (ASP .NET-страницы) и web-сервисы. Интерфейс .NET Framework позволяет стандартизировать обращение к системным вызовам и предоставляет среду для более быстрой и удобной разработки.

4.2. Создание приложения ASP.NET

Для создания веб-приложения с помощью ASP.NET необходимо создать пустой проект через меню File->New->Website->Visual C#->ASP.NET Web Site. По умолчанию проект создается в файловой системе. По желанию его можно создать на HTTP или FTP-сервере. Из файловой системы проект всегда можно скопировать на сервер нажатием одной кнопки в заголовке Solution Explorer.

В проекте будет создана страница Default.aspx. Выберите ее в окне браузера проекта (Solution Explorer), и появится окно редактирования с закладками Design и Source. Не меняя ничего, запустите проект, чтобы просмотреть страницу в браузере. Появится окно, в котором спрашивается, нужно ли добавить в файл web.config возможность отладки. Нажмите «OK». На панели задач должен появиться значок web-сервера. Откроется браузер, показывающий страницу по адресу http://localhost:номер_порта/Website1/default.aspx. «Localhost» обозначает сервер, работающий на вашем компьютере. При этом ваша страница будет скомпилирована.

Вы можете просмотреть внешний вид страницы, используя вкладку Design. Вкладка Source показывает исходный код страницы. В данном случае он выглядит следующим образом:

```
    <%@ Page Title="Home Page" Language="C#" MasterPage-
File="~/Site.master" AutoEventWireup="true"
        CodeFile="Default.aspx.cs" Inherits="_Default" %>

    <asp:Content ID="HeaderContent" runat="server" ContentPlaceHol-
derID="HeadContent">
    </asp:Content>
    <asp:Content ID="BodyContent" runat="server" ContentPlaceHolde-
rID="MainContent">
        <h2>
            Welcome to ASP.NET!
        </h2>
        <p>
            To learn more about ASP.NET visit <a
href="http://www.asp.net" title="ASP.NET Website">www.asp.net</a>.
```

```
        </p>
        <p>
            You can also find <a
href="http://go.microsoft.com/fwlink/?LinkID=152368&amp;clcid=0x40
9"
                title="MSDN ASP.NET Docs">documentation on ASP.NET
at MSDN</a>.
        </p>
    </asp:Content>
```

Первая строка содержит преамбулу документа:

```
    <%@ Page Title="Home Page" Language="C#" MasterPage-
File="~/Site.master" AutoEventWireup="true"
         CodeFile="Default.aspx.cs" Inherits="_Default" %>.
```

Тег <% всегда предназначается для интерпретации ASP-кода. Директива Page всегда присутствует на странице aspx. Атрибут Title – указание имени страницы, отображаемого в заголовке окна браузера. Атрибут Language – это указание, что в скриптах данной страницы будет использоваться C#. CodeFile – имя файла с отделенным кодом (code-behind). Inherits – класс, определенный в том файле, от которого наследуется класс страницы. Одновременно будет создан и файл Default.aspx.cs.

В данном случае используется технология разделения кода. Сама форма находится в файле Default.aspx, а в файле Default.aspx.cs находится класс страницы на языке C#. Таким образом, дизайн страницы может быть изменен, не затрагивая кода страницы, что позволяет разделить ответственность за внешний вид и работу страницы между дизайнером и программистом.

```
<asp:Content ID="HeaderContent" runat="server" …>
```

Этот тег дает компилятору указание обрабатывать элементы управления страницы. Обратите внимание на то, что данный тег имеет свойство runat, для которого установлено значение «server» (других значений не бывает). При использовании этого свойства элемент управления обрабатывается компилятором, а не передается браузеру «как есть».

Далее идет стандартная разметка страницы с помощью директив HTML. В данном случае страница содержит в основном статическую информацию. Рассмотрим возможности изменения поведения страницы.

Вначале нужно добавить странице элементы управления, для чего перед строкой `</asp:Content>`, отмечающей конец страницы, нужно добавить код:

```
    <p>
            <asp:Label ID="Label1" runat="server"
            Text="Мой элемент управления.
            Текущее время "></asp:Label>
    </p>
```

Этот код добавит к форме метку, которая может содержать текстовую информацию. Запустите приложение, и вы увидите изменения в дизайне страницы. Строка `ID="Label1"` указывает название элемента управления, по которому к нему можно обратиться в коде.

4.3. Добавление динамического поведения в Web-приложение

Технология разделения кода позволяет создать программный код, связанный с веб-страницей и выполняющийся при любых действиях с ней. Этот код содержится в отдельном файле и срабатывает всякий раз, когда страница получает от пользователя или браузера определенное событие. В нашем случае код для Default.aspx расположен в файле Default.aspx.cs. Вот его примерное содержимое:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;

public partial class _Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {

    }
}
```

Как видно, он представляет собой обычный программный код. Слово partial в определении класса `_Default` указывает, что код данного класса может храниться в нескольких файлах. Сейчас код содержит только заготовку метода, реагирующего на событие загрузки страницы в окне браузера. Добавьте внутрь метода код:

```
Label1.Text += DateTime.Now.ToString();
```

Теперь запустите приложение и несколько раз обновите страницу в браузере. Вы увидите изменения в строке вашего элемента управления.

Работа среды ASP .NET со страницей начинается с получения и обработки web-сервером IIS-запроса к данной странице и передачи этого запроса среде выполнения ASP .NET. Среда выполнения анализирует, нужно ли компилировать страницу или можно выдать в качестве ответа страницу из кэша. Затем начинается жизненный цикл страницы. Он начинается с этапа PreInit. После получения запроса среда выполнения загружает класс вызываемой страницы, устанавливает свойства класса страницы, выстраивает дерево элементов, заполняет свойства Request, Response и UICulture и вызывает метод IHttpHandler.ProcessRequest. После этого среда выполнения проверяет, каким образом была вызвана эта страница, и если страница вызвана путем передачи данных с другой страницы (о чем будет рассказано далее), то среда выполнения устанавливает свойство PreviousPage.

На этом этапе устанавливается также свойство IsPostback объекта Page, которое позволяет узнать, в первый раз загружается форма или формируется как результат обработки данных, введенных пользователем.

В обработчиках событий страницы можно проверить это свойство:

```
if (!Page.IsPostBack)
{
    // обрабатывать
}
```

Дальше происходит инициализация страницы – событие Init. Во время инициализации создаются дочерние пользовательские элементы управления, для которых устанавливаются свойства id. В это же время к странице применяются темы оформления. Если страница вызвана в результате постбэка, то на этом этапе данные, отправленные на сервер, еще не загружены в свойства элементов управления. Программист может инициализировать их свойства.

Во время события Load на основании информации о состоянии устанавливаются свойства элементов управления , если страница создается в результате отправки данных формы. Если на странице существуют валидаторы (классы проверки данных), то для них вызывается метод Validate(). Затем вызываются обработчики событий (при условии, что страница генерируется в ответ на действия пользователя).

В методе Render генерируется сам HTML-код выводимой страницы. При этом страница вызывает соответствующие методы дочерних элементов, а те в свою очередь – методы своих дочерних элементов. Наконец, страница выгружается из памяти сервера и происходит событие Unload.

Полный список событий страницы, которые можно переопределить в классе страницы:

1. `PreInit`
2. `Init`
3. `InitComplete`
4. `PreLoad`
5. `Load`
6. `LoadComplete`
7. `PreRender`
8. `PreRenderComplete`
9. `Unload`

Для всех событий определены обработчики – виртуальные функции `OnInit`, `OnLoad`. Когда `AutoEventWireup` равно `true`, в классе автоматически объявляются функции-обработчики событий с префиксом `Page` – `Page_Load`, `Page_Init` и т. д.

4.4. Формы ASP.NET

Основным средством взаимодействия Web-приложения с пользователем являются Web-формы, представляющие собой аналог оконных форм Windows. Web-формы состоят из графических элементов управления, позволяющих пользователю взаимодействовать с Web-приложением. При этом сам код приложения расположен на сервере, и для работы с ним пользователю нужен лишь браузер.

Web-формы строятся и работают по технологии ASP.NET, разделяя представление интерфейса и код обработки пользовательских действий. Для быстрой разработки формы существуют визуальные редакторы, в том числе в составе Microsoft Visual Studio.NET.

При создании формы программист вначале рисует форму в визуальном редакторе с помощью набора элементов управления, а затем настраивает обработку событий для формы. Обработчики могут относиться как к уровню страницы, так и к отдельному компоненту формы. Рассмотрим простой пример формы:

```
    <%@ Page Language="C#" AutoEventWireup="true"
CodeFile="Default2.aspx.cs" Inherits="Default2" %>

    <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

    <html xmlns="http://www.w3.org/1999/xhtml">
    <head runat="server">
        <title></title>
    </head>

    <body>
        <form id="form1" runat="server">
        <div>
        <asp:Label ID="Label1" runat="server"
         Text="Введите свое имя"></asp:Label>
         <asp:TextBox ID="TextBox1" runat="server" Height="46px"
            Width="257px"></asp:TextBox>
         <asp:Button ID="Button1" runat="server" Text="Отправить"
            onclick="Button1_Click" />
        </div>
        <p>
         <asp:Label ID="Label2" runat="server" Text=""></asp:Label>
        </p>
        </form>
    </body>

    </html>
```

В данной форме присутствует четыре компонента: метка Label1, содержащая текст «Введите свое имя», поле ввода TextBox1 для ввода имени, кнопка Button1, а также метка Label2. Вначале она пуста, а в процессе работы в нее выводится сообщение. В параметрах кнопки указан обработчик нажатия `Button1_Click`. При нажатии на кнопку введенное в поле ввода имя внедряется в сообщение, которое выводится в метке Label2.

Код обработки данной формы приведен ниже. Как видно, он представляет собой код класса на C#, содержащий обычные функции обработки событий. Код, связанный с формой, может быть достаточно сложным, однако его исполнение на стороне сервера никоим образом не отразится на производительности вашего компьютера.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;

public partial class Default2 : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {

    }
    protected void Button1_Click(object sender, EventArgs e)
    {
        Label2.Text = "Привет, " + TextBox1.Text + "!";
    }
}
```

4.5. Задание к лабораторной работе

Реализовать на платформе ASP.NET графическое приложение для работы с данными, представленными коллекцией элементов. Приложение должно реализовать добавление, удаление, просмотр и поиск сведений. В качестве вариантов задания использовать:

1. Телефонный справочник.
2. Адресную книгу.
3. Словарь.
4. Интернет-магазин.
5. Склад комплектующих.

4.6. Контрольные вопросы

1. Опишите технологию работы приложений ASP.NET.
2. Для чего используется разделение кода?
3. Что такое динамическая страница HTML?
4. Какие события страницы можно обрабатывать в ASP.NET?

Лабораторная работа №5
Элементы управления

Цель работы: изучить компонентную модель и средства разработки пользовательских элементов управления для программирования графического интерфейса пользователя. Создать собственный элемент управления и приложение, иллюстрирующее его работу.

5.1. Основные компоненты интерфейса Windows Forms

Графический интерфейс пользователя предполагает использование изображений различных объектов для представления входных и выходных данных программы. Программа с графическим интерфейсом всегда работает в рамках определенной оконной подсистемы. Программа выводит определенные кнопки, пиктограммы, диалоговые окна в своем окне, а пользователь управляет ею путем перемещения указателя по окну и выбора объектов управления.

В платформе Microsoft.NET существует специализированная библиотека разработки приложений с графическим интерфейсом пользователя, которая называется Windows Forms. Эта библиотека основана на принципах быстрой разработки приложений на базе готовых компонентов, поэтому она компонентно-ориентирована. Основным механизмом взаимодействия компонентов в рамках библиотеки является взаимодействие, основанное на событиях. Библиотека Windows Forms содержит большое множество различных элементов управления, существенно облегчающих процесс проектирования и реализации приложений с графическим интерфейсом пользователя.

Рассмотрим простое приложение на базе библиотеки Windows Forms.

```
using System;
using System.Windows.Forms;
public class SampleForm : System.Windows.Forms.Form
    {
        static void Main()
        {
            SampleForm sampleForm = new SampleForm();
            sampleForm.Text = "Sample Form";
            Button button = new Button();
            button.Text = "Close";
            button.Click +=
                new EventHandler(sampleForm.button_Click);
            sampleForm.Controls.Add(button);
            sampleForm.ShowDialog();
        }

        private void button_Click(object sender, EventArgs e)
        {
            Close();
        }
    }
```

Это приложение само создает форму и ее внутренние компоненты. Как правило, дизайн формы разрабатывается в визуальном редакторе, а код обработки событий компонентов содержится в соответствующих классах приложения. Дизайнер форм содержит панели инструментов, предоставляющие компоненты, которые можно размещать на форме и связывать с переменными в приложении.

5.2. Обработка событий элементов управления

Общая архитектура приложения для многооконного графического интерфейса представляет собой цикл обработки сообщений о событиях, происходящих во внешней информационной среде приложения. Событие – это действие, которое программист может обработать в программном коде, обеспечивая реакцию на него. События могут быть сгенерированы действиями пользователя, такими, как щелчок мыши или нажатие клавиши, кодом программы или действиями системы.

Управляемые событиями приложения выполняют код в процессе реакции на событие. Каждая форма и элемент управления содержат предопределенный набор событий, обработку которых можно программировать. Если происходит одно из этих событий и существует код связанного с ним обработчика событий, этот код будет вызван.

Для создания кода обработки сообщения о событиях, происходящих для конкретного элемента управления, следует выбрать его мышью и нажать правую кнопку мыши. Выполнив эти действия, получим доступ к всплывающему меню, в котором следует выбрать пункт «Свойства» (Properties). В этом окне можно изменять свойства самого элемента управления, а также добавлять код для обработки сообщения о событии. В качестве примера обработчика событий рассмотрим код обработки нажатия кнопки формы.

```
private void button_Click(object sender, EventArgs e)
{
    Close();
}
```

Для установки реакции на событие к списку обработчиков нажатия кнопки добавляется ссылка на данный код обработки.

```
button.Click +=
    new EventHandler(sampleForm.button_Click);
```

Поскольку для обработки используется делегат, который может быть многоадресным, сюда можно добавить еще ряд обработчиков. Все они будут выполняться последовательно по порядку их регистрации.

5.3. Компонентная модель .NET

Компонентная модель .NET Framework определяет правила создания и использования компонентов .NET (программную модель), а также классы и интерфейсы, поддерживающие описание компонентов. Иерархия компонентов Windows Forms приведена на рис. 5.1.

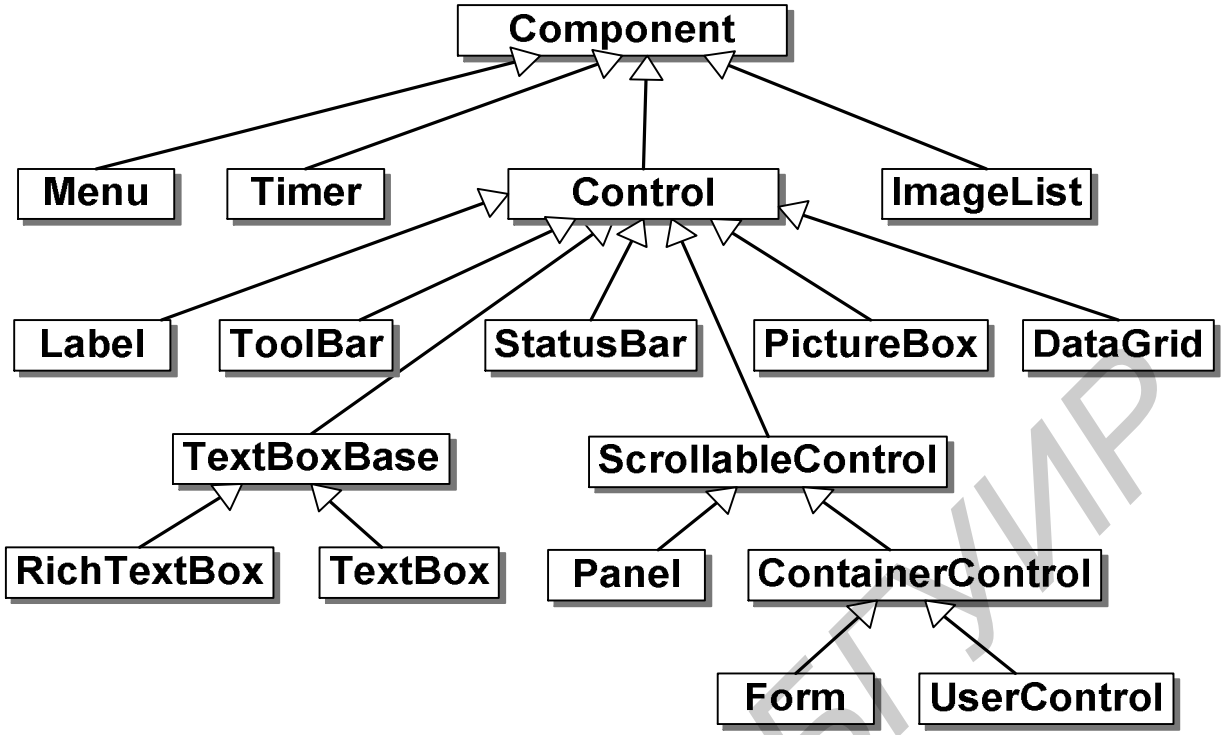

Рис. 5.1. Иерархия графических компонентов Windows Forms

Класс System.Windows.Forms.Control является основой любого графического элемента управления Windows Forms. Его свойства типичны для всех элементов управления Windows Forms. Основные свойства класса Control:

– Anchor, Dock – задает способ, которым элемент управления закрепляется за контейнером;
– Bounds – задает размер и позицию элемента управления в его контейнере;
– BackColor – задает цвет фона;
– ContextMenu – задает контекстное меню (popup menu) элемента управления;
– Controls – содержит коллекцию вложенных элементов управления (если они есть);
– CanFocus – указывает, может ли элемент управления получать фокус;
– Enabled – позволяет запретить элемент управления (он остается видимым, но неактивным);
– Font – задает шрифт (имя, стиль, размер);
– ForeColor – задает цвет элемента управления;
– Location – содержит позицию элемента управления в его контейнере;
– Parent – задает контейнер, содержащий текущий элемент управления;
– Size – содержит размеры элемента управления;
– TabIndex – определяет порядок при навигации с клавишей [TAB];
– TabStop – определяет, может ли элемент управления фокусироваться при навигации с [TAB];

– Text – задает текст, связанный с элементом управления;
– Visible – задает видимость элемента управления.

Рассмотрим основные этапы разработки пользовательского элемента управления Windows Forms. Простейший элемент управления, разработанный в этом пошаговом руководстве, позволяет изменять выравнивание своего свойства Text. Он не инициирует и не обрабатывает события.

Вначале определяется класс, производный от класса System.Windows.Forms.Control.

```
public class FirstControl:Control{}
```

Затем определяются свойства элемента управления. Элемент управления наследует многие свойства от класса Control, но для большинства пользовательских элементов управления обычно определяются дополнительные свойства. В следующем фрагменте кода определяется свойство с именем TextAlignment , которое используется FirstControl для форматирования значения свойства Text, унаследованного от Control.

```
private ContentAlignment alignmentValue =
        ContentAlignment.MiddleLeft;
```

Когда задается свойство, изменяющее отрисовку элемента управления, следует вызвать метод Invalidate, чтобы перерисовать элемент управления. Метод Invalidate определен в базовом классе Control.

```
public ContentAlignment TextAlignment
        {

            get
            {
                return alignmentValue;
            }

            set
            {
                alignmentValue = value;
                Invalidate();
            }
        }
```

Затем переопределяется защищенный метод OnPaint, наследуемый от Control, чтобы обеспечить логику отрисовки элемента управления. Если не переопределить метод OnPaint, элемент управления не сможет себя нарисовать. В следующем фрагменте кода метод OnPaint отображает значение свойства Text, унаследованного от Control, с выравниванием, определяемым значением поля alignmentValue.

```
protected override void OnPaint(PaintEventArgs e)
{
    base.OnPaint(e);
    StringFormat style = new StringFormat();
    style.Alignment = StringAlignment.Near;
    switch (alignmentValue)
    {
        case ContentAlignment.MiddleLeft:
            style.Alignment = StringAlignment.Near;
            break;
        case ContentAlignment.MiddleRight:
            style.Alignment = StringAlignment.Far;
            break;
        case ContentAlignment.MiddleCenter:
            style.Alignment = StringAlignment.Center;
            break;
    }
    e.Graphics.DrawString(
        Text,
        Font,
        new SolidBrush(ForeColor),
        ClientRectangle, style);
}
```

Можно задать атрибуты элемента управления. Атрибуты позволят визуальному конструктору отображать элемент управления и, соответственно, его свойства и события в режиме разработки. Следующий фрагмент кода применяет атрибуты для свойства TextAlignment.

```
[ Category("Alignment"),
Description("Specifies the alignment of text.")]
```

Для развертывания элемента управления следует скомпилировать исходный код в сборку и сохранить ее в каталоге приложения. Для этого выполняется следующая команда из каталога, содержащего исходный файл.

```
csc /t:library
/out:[путь к каталогу приложения]/CustomWinControls.dll
r:System.dll /r:System.Windows.Forms.dll
r:System.Drawing.dll FirstControl.cs
```

Параметр компилятора /t:library сообщает компилятору, что сборка, создаваемая пользователем, является библиотекой (а не исполняемым файлом). Параметр /out задает путь и имя сборки. Параметр /r задает имена сборок, на которые ссылается код. В этом примере создается закрытая сборка, которую может использовать только приложение пользователя.

В следующем примере рассмотрена простая форма, использующая элемент управления FirstControl.

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
namespace CustomWinControls
```

```
{
    public class SimpleForm : System.Windows.Forms.Form
    {
        private FirstControl firstControl1;
        private System.ComponentModel.Container components = null;
        public SimpleForm()
        {
            InitializeComponent();
        }
        protected override void Dispose( bool disposing )
        {
            if( disposing )
            {
                if (components != null)
                {
                    components.Dispose();
                }
            }
            base.Dispose( disposing );
        }
        private void InitializeComponent()
        {
            this.firstControl1 = new FirstControl();
            this.SuspendLayout();

            this.firstControl1.BackColor =
                   System.Drawing.SystemColors.ControlDark;
            this.firstControl1.Location =
                   new System.Drawing.Point(96, 104);
            this.firstControl1.Name = "firstControl1";
            this.firstControl1.Size =
                   new System.Drawing.Size(75, 16);
            this.firstControl1.TabIndex = 0;
            this.firstControl1.Text = "Hello World";
            this.firstControl1.TextAlignment =
                   System.Drawing.ContentAlignment.MiddleCenter;
            this.ClientSize = new System.Drawing.Size(292, 266);
            this.Controls.Add(this.firstControl1);
            this.Name = "SimpleForm";
            this.Text = "SimpleForm";
            this.ResumeLayout(false);
        }
        [STAThread]
        static void Main()
        {
            Application.Run(new SimpleForm());
        }
    }
}
```

5.4. Задание к лабораторной работе

Создайте собственный элемент управления, представляющий собой поле для вывода текста с возможностью простейшей подсветки синтаксиса (разные цвета для чисел, переменных, служебных слов, строк, разделителей), а также программу, иллюстрирующую его применение.

5.5. Контрольные вопросы

1. Что такое библиотека Windows Forms, для чего она используется?
2. Что такое обработчик сообщения и для чего он нужен?
3. Что такое компонентная модель .NET Framework?
4. Перечислите этапы создания собственных элементов управления.

Лабораторная работа №6
Удаленное взаимодействие Microsoft .NET

Цель работы: изучить возможности организации удаленного взаимодействия между компонентами приложения посредством среды .NET Remoting. Создать приложение, иллюстрирующее возможности удаленного взаимодействия.

6.1. Основы использования .NET Remoting

Среда Remoting является универсальным средством доступа к удаленным объектам, которое может быть приспособлено к широкому классу задач взаимодействия компонентов распределенного приложения. Благодаря своей расширяемой архитектуре среда Remoting может быть доработана для использования с практически любыми каналами передачи данных.

С точки зрения среды Remoting, все классы объектов среды CLR делятся на три вида.

1. Классы, маршализируемые по значению. Объекты этих классов могут копироваться между доменами приложений, если для них определены операции сериализации и десериализации. В результате десериализации создается копия объекта, не связанная с его оригиналом.

2. Классы, маршализируемые по ссылке. Такие классы наследуются от класса System.MarshalByRefObject. Объекты этих классов не покидают свой домен приложения, но на стороне клиента создается посредник, позволяющий осуществлять удаленный доступ к объекту. Именно экземпляры таких классов могут использоваться как удаленные объекты при помощи среды Remoting. Remoting поддерживает все три вида удаленных объектов: объекты единственного вызова, единственного экземпляра и объекты, активируемые по запросу клиента.

3. Немаршализируемые классы, а также классы без определенной процедуры сериализации или отмеченные как несериализуемые. Объекты этих классов недоступны вне своего домена приложения. Это, в частности, касается и

классов исключений: порождаемые на сервере Remoting исключения должны маршализироваться по значению для передачи клиенту.

Инфраструктура .NET Remoting состоит из:

– каналов – переносят сообщения от и к удаленным объектам;
– форматтеров – кодируют и декодируют сообщения в определенный формат;
– прокси-классов – передают вызовы методов к удаленным объектам;
– механизмов активации – обеспечивают удаленное создание объектов;
– маршалинга – обеспечивает перенос объектов, их свойств, полей и т. д.

Схема применения указанных элементов приведена на рис. 6.1.

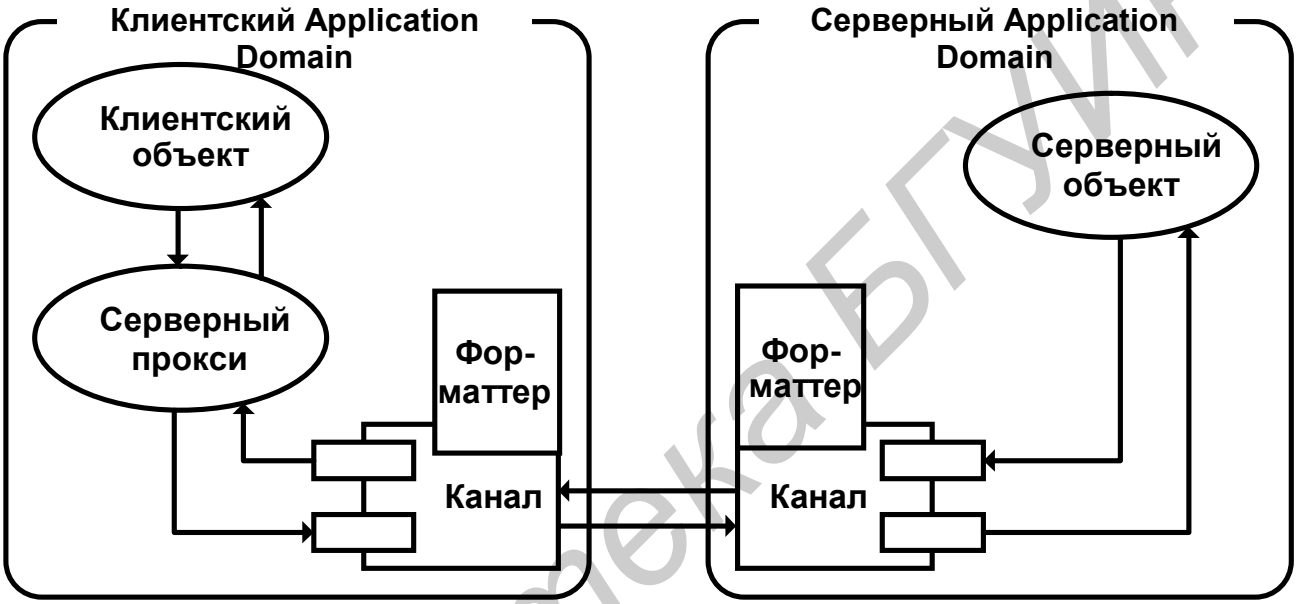

Рис. 6.1. Основные элементы технологии .NET Remoting

Для передачи информации об объекте сервера и создания на клиенте «прозрачного» посредника используется особый класс System.Runtime.Remoting.ObjRef. При сериализации наследников класса System.MarshalByRefObject по каналу Remoting вместо них передается маршализируемый по значению объект класса ObjRef, который после десериализации становится «прозрачным» посредником на клиенте Remoting.

6.2. Создание простого приложения на базе .NET Remoting

Среда Remoting имеет штатную возможность описания своей конфигурации в XML-файле. К конфигурации среды Remoting относятся:

– используемый класс канала;
– параметры канала (например, используемый сервером порт TCP);
– используемый класс форматирования;
– адрес и тип используемого удаленного объекта.

Разработчик может задать конфигурацию среды Remoting непосредственно в программе. Однако, применение файлов конфигураций позволяет отде-

лить используемый в программе удаленный объект от места его физического размещения и допускает настройку одного и того же распределенного приложения на использование различных каналов передачи данных без изменения исходного кода программы.

Файл конфигурации на стороне, где находится маршализируемый по ссылке объект, используется приложением, являющимся сервером среды Remoting. В роли такого приложения может выступать либо IIS, либо пользовательская программа, обычно реализованная как сервис операционной системы. Для учебных целей можно воспользоваться следующим сервером .NET Remoting.

```
// server.cs
using System;
using System.Runtime.Remoting;
using System.Configuration;

public class Server
{
    public static void Main(string[] args)
    {
        string config =
  AppDomain.CurrentDomain.SetupInformation.ConfigurationFile;
        Console.WriteLine(config);
        RemotingConfiguration.Configure(config);
        // Для .NET Framework 2.0 рекомендуется использовать
        // RemotingConfiguration.Configure(config, false);

        Console.WriteLine("Нажмите ENTER для завершения работы
сервера.");
        Console.ReadLine();
    }
}
```

Назначение такого сервера – настройка среды Remoting на основе содержимого стандартного конфигурационного файла программы. Рассмотрим следующий пример конфигурации сервера.

```
<configuration>
    <system.runtime.remoting>
        <application>
            <service>
                <wellknown mode="SingleCall"
                    type="RemoteTest.TestService,
                      RemoteTestService"
                    objectUri="endpoint"/>
            </service>
            <channels>
                <channel ref="tcp" port="2080" />
            </channels>
        </application>
    </system.runtime.remoting>
</configuration>
```

В разделе <application> <service> указываются публикуемые сервером объекты. Для каждого из них указывается модель вызова, имя класса (через запятую идет полное или краткое имя сборки, которая содержит класс), и уникальный URI, связываемый с данным классом. Объекты, активируемые сервером, описываются как

```
<wellknown mode="SingleCall" type=...
```

для режима единственного вызова или

```
<wellknown mode="Singletone" type=...
```

для режима одного экземпляра. Объекты, активируемые по запросу клиента, описываются иначе.

```
<activated type=...
```

Раздел <application><channels> сервера в простейшем случае содержит идентификатор канала и номер порта. Стандартные каналы имеют идентификаторы, соответствующие названиям их протоколов.

Файл конфигурации клиента обычно выглядит примерно следующим образом.

```
<configuration>
  <system.runtime.remoting>
    <application>
      <client>
        <wellknown mode="SingleCall"
         type="RemoteTest.TestService, RemoteTestService"
         objectUri="tcp://127.0.0.1:2080/endpoint" />
      </client >
    </application>
  </system.runtime.remoting>
</configuration>
```

Раздел <application><client> включает описание используемых удаленных объектов. Следует отметить, что один и тот же файл может содержать и разделы <service>, и <client>. Программа при этом будет настроена как сервер и клиент Remoting одновременно.

Клиент, использующий удаленный объект, сначала должен настроить среду Remoting. После чего при создании объекта любого из перечисленных в файле конфигурации в разделе <client> классов будет создан посредник, связанный с соответствующим удаленным объектом.

```
using System;
using System.Runtime.Remoting;

using RemotingTest;

public class Client
{
    public static void Main(string[] args)
    {
        string config =
AppDomain.CurrentDomain.SetupInformation.ConfigurationFile;
        Console.WriteLine(config);
        RemotingConfiguration.Configure(config);
```

```
// Для объекта единственного вызова в следующей строчке будут
// созданы посредники, конструктор самого объекта
.. не вызывается
        TestService service = new RemotingTest.TestService();
// При удаленном вызове на сервере
// будет сконструирован новый объект
        Console.WriteLine(service.Sum(2, 2));
    }
}
```

Ниже приведен код самого маршализируемого по ссылке объекта. Конструктор и деструктор объекта выводят сообщения в консоль, чтобы продемонстрировать моменты создания и удаления объекта на сервере.

```
// TestService.cs
using System;
namespace RemotingTest
{
    public class TestService : MarshalByRefObject
    {
        public TestService()
        {
            Console.WriteLine("[ctor]");
        }

        ~TestService()
        {
            Console.WriteLine("[dtor]");
        }

        public int Sum(int a, int b)
        {
            return a + b;
        }
    }
}
```

Вначале запускается сервер, затем клиент. После этого клиент обращается к серверу и выполняет удаленный вызов процедуры суммирования.

6.3. Задание к лабораторной работе

Создать распределенное приложение, выводящее текущее время на удаленной машине, проиллюстрировать его работу.

6.4. Контрольные вопросы

1. Для чего используется удаленное взаимодействие?
2. Какие основные элементы входят в архитектуру .NET Remoting?
3. Что описывает конфигурационный файл?
4. Что такое маршалинг?

Лабораторная работа №7
Доступ к данным с использованием Microsoft ADO.NET

Цель работы: изучить возможности разработки приложений для работы с базами данных с помощью технологии Microsoft ADO.NET. Создать приложение ведения базы данных и ее визуализации.

7.1. Назначение ADO.NET

База данных – это совокупность сведений (об объектах, процессах, событиях или явлениях), относящихся к определенной теме или задаче. Она организована таким образом, чтобы обеспечить удобное представление как этой совокупности в целом, так и любой ее части.

Для работы с данными используются системы управления базами данных (СУБД). Основные функции СУБД – это определение данных (описание структуры баз данных), обработка данных и управление данными.

Любая СУБД позволяет выполнять следующие операции с данными:

– добавлять в таблицу одну или несколько записей;
– удалять из таблицы одну или несколько записей;
– обновлять значения некоторых полей в одной или нескольких записях;
– находить одну или несколько записей, удовлетворяющих заданному условию.

Для выполнения этих операций применяется механизм запросов. Результатом выполнения запросов является либо отобранное по определенным критериям множество записей, либо изменения в таблицах. Запросы к базе формируются на специально созданном для этого языке, который так и называется «язык структурированных запросов» (SQL – Structured Query Language).

Программная библиотека ADO.NET представляет собой набор классов для работы с данными. Она определяет программную модель работы с данными, обеспечивает возможность работы в несвязной среде (когда данные редактируются локально, а затем отправляются в базу данных). Технология ADO.NET очень тесно связана с XML и является развитием ADO (технологии доступа к базам данных Windows).

Основные компоненты платформы ADO.NET показаны на рис. 7.1. Она поддерживает две модели взаимодействия: несвязную и связную. В несвязной модели данные извлекаются из базы данных (так называемый снимок базы) в объект `DataSet`, корректируются на стороне пользователя, а затем отправляются назад. Объекты `DataAdapter` служат для подключения к различным СУБД. В связной модели работа с базой происходит непрерывно в режиме прямого соединения через объекты `DataReader`. Объекты `Command` используются для выполнения модификации базы средствами СУБД. Поставщики данных (Data Provider) обеспечивают единый интерфейс работы с СУБД различных производителей.

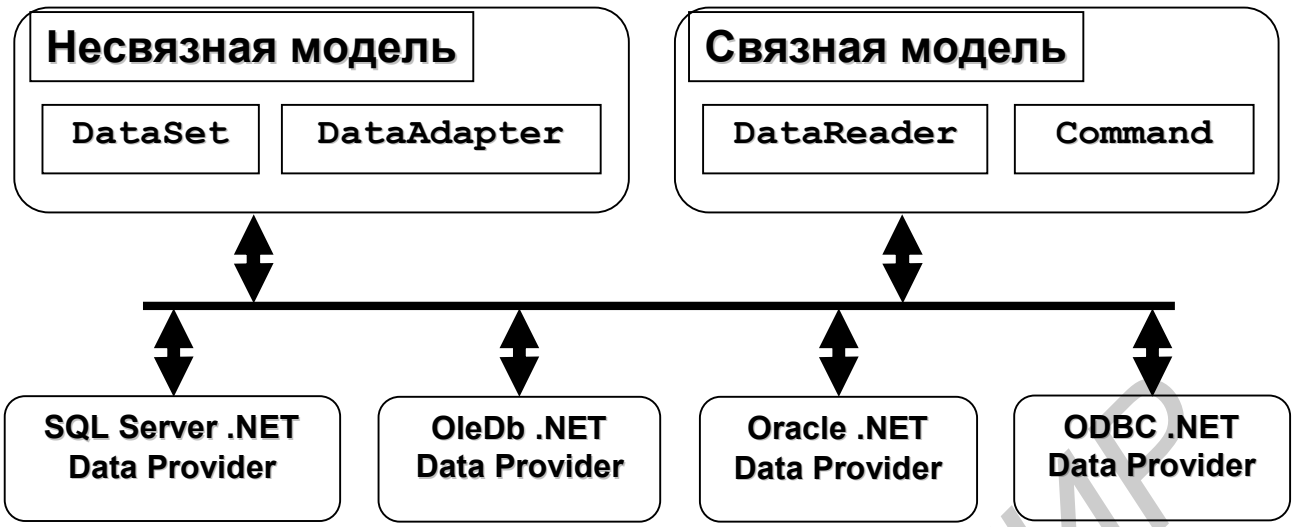

Рис. 7.1. Основные элементы ADO.NET

7.2. Создание приложения для работы с базой данных

Мастер Data Form Wizard из Visual Studio .NET позволяет быстро создать связанную с данными форму. Код, генерируемый мастером, можно будет просматривать и изменять. Выбрав тип мастера Data Form Wizard, необходимо указать название новой формы. В следующем шаге определяется объект DataSet, название которого должно соответствовать содержимому базы данных. В следующем шаге мастера «Choose a data connection» требуется создать новое подключение к базе данных через кнопку «New Connection». В появившемся окне «Свойства связи с данными» необходимо определить параметры создаваемого подключения, т. е. поставщика данных в зависимости от типа базы.

После этого следует указать файл базы данных и определить, какую таблицу базы данных нужно извлекать. В окне мастера перечислены все доступные в схеме базы данных таблицы, представления и хранимые процедуры. Нужные таблицы надо выбрать в списке доступных и переместить их в список выбранных, щелкнув на кнопку с направленной вправо стрелкой.

Теперь необходимо определить отношение между таблицами. Отношения позволяют обеспечивать соблюдение правил ссылочной целостности, каскадно передавая изменения от одной таблицы к другой. Кроме того, они упрощают поиск данных в таблицах. Можно также выбрать поля, которые будут извлекаться из таблицы, если ее структура слишком обширна.

В последнем шаге мастера предстоит определить вид размещения данных на форме – всех записей в виде таблицы (All records in a grid) либо каждой записи в отдельном текстовом поле (Single records in individual control). Здесь же можно определить наличие дополнительных элементов управления – кнопок навигации и изменения записей.

Теперь необходимо изменить код приложения так, чтобы при запуске выводилась созданная форма. Пусть она имеет название MyDBForm, тогда функция Main будет иметь вид:

```
[STAThread]
      static void Main()
      {
         Application.Run(new MyDBForm());
   }
```

В созданном приложении имеется несколько кнопок. Для загрузки данных используется кнопка Load. Для перемещения по записям используются навигационные кнопки. Для добавления новой или удаления текущей записи служат кнопки Add и Delete, для отмены изменений текущей записи – Cancel. Все изменения буферизуются и могут быть отменены нажатием кнопки Cancel All. Для передачи изменений в базу данных служит кнопка Update – тогда записи изменяются в самом файле базы данных

7.3. Выполнение запросов к базе данных

В данном примере кода используется учебная база данных Northwind из Microsoft SQL Server 7.0. Код создает команду SqlCommand для выборки строк из таблицы Products, к которой добавляется параметр SqlParameter, ограничивающий результат строками, для которых значение UnitPrice превышает указанное значение параметра, в данном случае 5. Соединение SqlConnection открывается в блоке using, что гарантирует закрытие и освобождение ресурсов после завершения работы кода. Команда выполняется с помощью объекта SqlDataReader, а результаты выводятся в окно консоли.

```
using System;
using System.Data;
using System.Data.SqlClient;

class Program
{
    static void Main()
    {
        string connectionString =
            "Data Source=(local);Initial Catalog=Northwind;"
            + "Integrated Security=true";

        // Создаем строку запроса.
        string queryString =
            "SELECT ProductID, UnitPrice,
            ProductName from dbo.products "
                + "WHERE UnitPrice > @pricePoint "
                + "ORDER BY UnitPrice DESC;";

        // Указываем значение параметра.
        int paramValue = 5;

        // Создаем и открываем соединение
        using (SqlConnection connection =
            new SqlConnection(connectionString))
        {
```

```
            // Создаем объекты Command и Parameter.
            SqlCommand command =
            new SqlCommand(queryString, connection);
            command.Parameters.AddWithValue(
            "@pricePoint", paramValue);

            // Открываем соединение.
            // Выполняем чтение данных через DataReader
            try
            {
                connection.Open();
                SqlDataReader reader =
                command.ExecuteReader();
                while (reader.Read())
                {
                    Console.WriteLine("\t{0}\t{1}\t{2}",
                        reader[0], reader[1], reader[2]);
                }
                reader.Close();
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
            Console.ReadLine();
        }
    }
```

7.4. Задание к лабораторной работе

Создайте базу данных, например, в Microsoft Access. Для данной базы данных разработайте приложение ADO.NET для ее редактирования и визуализации, а также выполнения запросов с формированием результатов в текстовый файл.

7.5. Контрольные вопросы

1. Для чего применяется технология ADO.NET?
2. Какие режимы работы с базами данных реализованы в ADO.NET?
3. Как реализуется несвязное взаимодействие с базой данных?
4. Как реализуется связное взаимодействие с базой данных?

Литература

1. Гамильтон, Б. ADO.NET. Сборник рецептов / Б. Гамильтон. – СПб. : Питер, 2005. – 576 с.

2. Макдональд, М. ASP.NET / М. Макдональд. – СПб. : «БХВ-Петербург», 2003. – 992 с.

3. Рихтер, Дж. Программирование на платформе Microsoft.NET Framework / Дж. Рихтер. – 2-е изд., испр. – М. : «Русская редакция», 2003. – 512 с.

4. Троелсен, Э. Язык программирования C# 2005 и платформа .NET 2.0 / Э. Троелсен. – М. : Изд. дом «Вильямс», 2006. – 1168 с.

5. Хортон, А. Microsoft Visual C++ 2005: базовый курс / А. Хортон. – М. : Диалектика, 2007. – 1152 с.

Св. план 2011, поз. 56

Учебное издание

Отвагин Алексей Владимирович
Павлёнок Наталья Александровна

ПЛАТФОРМА MICROSOFT.NET

Лабораторный практикум
по дисциплине «Объектно-ориентированное
проектирование и программирование»
для студентов специальности 1-40 02 01
«Вычислительные машины, системы и сети»
всех форм обучения

Редактор И. П. Острикова
Корректор А. В. Тюхай

Подписано в печать 30.08.2011.	Формат 60x84 1/16.	Бумага офсетная.
Гарнитура «Таймс».	Отпечатано на ризографе.	Усл. печ. л. 2,91.
Уч.-изд. л. 2,5.	Тираж 100 экз.	Заказ 49.

Издатель и полиграфическое исполнение: учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники»
ЛИ №02330/0494371 от 16.03.2009. ЛП №02330/0494175 от 03.04.2009.
220013, Минск, П. Бровки, 6