Правительство Российской Федерации

**Федеральное государственное автономное образовательное учреждение
высшего профессионального образования
«Национальный исследовательский университет
«Высшая школа экономики»**

**Московский институт электроники и математики
Национального исследовательского университета
«Высшая школа экономики»**

Кафедра информационных технологий
и автоматизированных систем

МОДЕЛИРОВАНИЕ СИСТЕМ МАССОВОГО ОБСЛУЖИВАНИЯ СРЕДСТВАМИ ЯЗЫКА C#

Учебно-методическое пособие
по проведению лабораторного практикума
по дисциплине “Алгоритмы и структуры данных”

Москва 2012

Составители: аспирант Л.Л. ВОЛКОВА
доц., к.т.н. Э.С. КЛЫШИНСКИЙ

Пособие содержит описание средств языка C#, теоретические положения, необходимые для выполнения лабораторных работ по курсу “Алгоритмы и структуры данных”. Также в пособии приведены задания на лабораторные работы.

УДК 681.3

Моделирование систем массового обслуживания средствами языка C#: Учебно-метод. пособие по проведению лабораторного практикума по дисциплине “Алгоритмы и структуры данных” / Моск. институт электроники и математики Национального исследовательского университета «Высшая школа экономики». Сост.: Л.Л. Волкова, Э.С. Клышинский. М., 2012. 32 с.

Библиогр.: 9 назв.

ISBN 978-5-94506-311-2

Содержание

1. Стек, дек и очередь

1.1. Статические и динамические структуры данных

Структуры данных подразделяются на статические (постоянство размеров во время выполнения, память под них выделена один раз, элементы структуры в памяти смежны), динамические (размер непостоянен, отсутствие физической смежности элементов, следовательно, необходимость хранения адреса). Если количество элементов не постоянно, структуры считаются полустатическими. К последним относятся стек, дек, очередь, список.

Массив – пример статической структуры. Это набор однотипных компонентов (элементов), расположенных в памяти непосредственно друг за другом, доступ к компонентам осуществляется по индексу (индексам).

Стек организован по принципу LIFO (last in, first out), доступ к элементам осуществляется только с вершины стека; при просмотре элемент извлекается из стека. Очередь – FIFO (first in, first out): включение в хвост, исключение с головы. Дек – двунаправленная очередь: включение и исключение возможны с обоих концов; здесь левый и правый конец очереди. Ко всем трем структурам применимы следующие операции: очистка, включение элемента, исключение элемента. Следует отслеживать пустоту и непереполнение каждой структуры.

Списки бывают односвязными и многосвязными. Односвязный список – упорядоченная последовательность элементов данных, каждый из которых характеризуется одним и тем же набором полей. Операции над списком: очистка списка; включение элемента в список на i-ю позицию и исключение с i-й позиции (по умолчанию – в конец и из конца соответственно). Многосвязный список – это набор списков, один из которых является общим; любой элемент может быть включен в любое количество списков (быть связанным с любым элементом, иметь произвольное число таких связей). Связи могут иметь направление и вес (ориентированный граф со взвешенными дугами).

1.2. C#: основы синтаксиса. Типы. Ветвление алгоритмов

Типы

Типы в C# подразделяются на типы по значению (не требуют new при инициализации; переменные этих типов хранятся в стеке) и ссылочные типы (переменная хранит ссылку на объект; если две переменные ссылаются на один объект, изменения одной переменной отразятся и во второй, поскольку они ссылаются на те же данные; память выделяется в т.н. управляемой куче). К ссылочным типам относятся, например, object, Array, List<T>, Button, EventArgs, StreamReader, File, DateTime и др. Любому объекту ссылочного типа можно присвоить значение null (этим значением он также инициализируется по умолчанию), поэтому в документации эти типы отмечаются как nullable.

К типам по значению относятся: булевский boolean, символ char, строка string, числовые чипы. Целочисленные:

sbyte (SByte, 8 бит), short (Int16, 16 бит), int (Int32, 32 бита), long (Int64, 64 бита).

Целочисленные неотрицательные (unsigned):

byte (Byte, 8 бит), ushort (UInt16, 16 бит), uint (UInt32), ulong (UInt64).

С плавающей запятой:

float (Single, 32 бита), double (Double, 64 бита), decimal (Decimal, 128 бит).

Массивы могут задаваться различным образом: одномерные (int[]) и многомерные массивы (int[,,]), массивы массивов (int[][]), а также их базовый тип Array. Максимальное число элементов массива определено размером long, индексация элементов в массивах начинается с 0. При создании массив элементов ссылочного типа по умолчанию инициализируется значениями null, массив элементов типа по значению – соответствующими типу значениями по умолчанию (для числовых элементов – 0, для строки – пустая строка ""). Размер массива хранится в свойстве Count. Массивы, как и скалярные переменные, могут быть проинициализированы явно или неявно (по умолчанию), как показано в примере ниже. Обратите внимание: поскольку массив – это ссылочный тип, то при выполнении присвоения элементу c2 во второй строчке это изменение отразится на обоих массивах, которые ссылаются на одну область памяти, в которую и внесено изменение. В массиве массивов g число элементов во вложенных массивах различно.

```
int[] a = new int[10], b = new int[] { 1, 2, 3 }, c = { 4, 5, 6, 7 }, c2 = c;
a[1] = 5; c2[0] = 9;       // теперь c[0] == c2[0]
int[,] d = new int[2, 3], e = { { 1, 2 }, { 3, 4 } };
int[][] g = new int[][] { new int[] { 5, 6, 7 }, new int[] { 8, 9 } };
```

Если использовать тип Array, можно менять тип массива:

```
Array a = new double[5];
a.SetValue(1.5, 0);
a.SetValue(a.GetValue(0), 1); // Оператор [] не определен для типа Array.
a = new string[(long)Int32.MaxValue + 2];
a.SetValue("Hi!", 1);
```

Обратите внимание, что предпоследняя строчка указанного выше примера при выполнении вызывает ошибку переполнения памяти (как минимум, в 32-разрядной версии ОС). При этом размер массива – величина допустимая.

Список (List<T>) – строго типизированный список объектов, доступных по индексу. Здесь T – тип элементов. Класс List поддерживает методы поиска по списку (IndexOf(T), Contains(T)), сортировки (Sort()) и другие операции со списками (Add(T), AddRange(List<T>), RemoveAt(int), Insert(int, T), InsertRange(List<T>), Clear(), ToArray()), Reverse().

```
List<int> L = new List<int>();
L.Add(5); L.Add(6); L.RemoveAt(0);
```

Перечисление (enum) – это тип, состоящий из набора именованных констант. По умолчанию первой константе в перечислителе присваивается значение 0, но это значение можно изменить, явно указав первую константу; каждая следующая константа будет инкрементироваться. Каждый тип перечисления имеет базовый тип, определяющий ёмкость хранилища каждого перечислителя, по умолчанию это int. Можно изменить базовый тип, как указано ниже; базовым типом могут служить byte, sbyte, short, ushort, int, uint, long, ulong. Переменной перечислимого типа может быть присвоено любое значение, входящее в диапазон базового типа, значения не ограничены именованными константами. Значением enum E по умолчанию является значение, созданное выражением (E)0. Перечисление может быть вложено в класс или структуру, но, как правило, удобнее определять перечисление в пространстве имен, чтобы все классы могли напрямую получить к нему доступ.

```
enum ColorsBW {Black, White}, ColorsRGB: byte {Red = 1, Green, Blue};
```

Строковый тип (string) – массив символов типа char в кодировке Unicode. Длина строки хранится в свойстве Length. У этого типа есть ряд методов для поиска подстрок и их позиций, к примеру, Contains(string), IndexOf(string), IndexOf(string value, int startIndex),

LastIndexOf(char), а также методы, возвращающие значение типа string (но не изменяющие исходную строку): Insert(int startIndex, string value), Remove(int startIndex, int count), Substring(int startIndex, int count), Replace(string oldValue, string newValue), Trim().

Указателей как таковых в C# нет в связи с обеспечением безопасности кода. Как и в C++, C# предоставляет такую возможность, как объявление переменных в любом месте. К примеру, можно следующим образом объявить переменную цикла: (for int i = 0; i < 5; i++) { }. Для освобождения памяти, выделенной по динамическим запросам, в .NET есть сборщик мусора (garbage collector). Динамически выделяемая память распределяется в область кучи, и объекты, на которые уже нет ссылок, удаляются сборщиком мусора. C# чувствителен к регистру: переменные int a и int A различны.

Приведение и преобразование типов

Поскольку в C# переменные типизируются при компиляции, нельзя присвоить переменной одного типа значение другого типа (только в случае, если последний тип преобразуется в тип переменной). Однако иногда возникает необходимость копирования значения в переменную другого типа (к примеру, int в double). Существует неявная конверсия (от целочисленных типов меньшей ёмкости к большей ёмкости или от производных классов к базовым и обратно), явная конверсия, или приведение типов (целочисленный тип к типу с плавающей запятой; через оператор приведения (Type)var, т.е. указанный перед переменной в скобках тип, к которому выполняется приведение), пользовательская конверсия и конверсия через специализированные классы (например, специализированный класс Convert или встроенные методы конверсии в различных типах, например, string DateTime.ToShortDateString(); встроенные методы ToString() у многих типов, включая числовые). Также для выполнения определенного вида преобразований между совместимыми ссылочными типами используется оператор as. Он подобен оператору приведения, однако если преобразование невозможно, возвращает значение null, а не вызывает исключение. Оператор as выполняет только преобразования ссылок и упаковки-преобразования.

В рассмотренном выше примере, демонстрирующем смену типа элементов массива a, использовано приведение размера массива к типу long. Если эту строку заменить строкой a = new string[Int32.MaxValue + 2], компилятор выдаст ошибку, потому что значение в скобках будет интерпретировано как Int32, так как Int32.MaxValue имеет этот тип. Чтобы ошибки не возникло, используется приведение к типу long. В примере ниже, чтобы обратиться к объекту obj как к списку, мы используем для преобразования типов оператор as:

```
object obj = new List<int>();
(obj as List<int>).Add(9);
```

Подобный синтаксис удобен, если в функцию передаётся параметром объект, который может быть, к примеру, одного из двух типов:

```
if (obj.GetType() == typeof(List<int>))
        (obj as List<int>).Add(1);
else
        if (obj.GetType() == typeof(List<double>))
                (obj as List<double>).Add(0.5);
```

Ветвление алгоритмов

Условный оператор с одной ветвью:

```
if ( flag )
```

```
{   // Действия при выполнении условия
}
```

Условный оператор с двумя ветвями:

```
if ( flag )
{   // Действия при выполнении условия
}
else
{   // Действия при невыполнении условия
}
```

Переключатель:

```
switch ( i )
{
   case 1:     // Сквозное выполнение (без break, в верхнем case нет кода)
   case 2:     // Действия для случаев 1 и 2
      break;
   case 10:   // Действия для случая 10 (каждое значение switch case уникально)
      break;
   default:    // Действия по умолчанию, т.е. «иначе» (необязательная ветвь)
      break;
}
```

Цикл с предусловием:

```
while( condition )
{   // Тело цикла
}
```

Цикл с постусловием:

```
do
{   // Тело цикла
}
while(ContinuingCondition)
```

Цикл со счётчиком:

```
for ( i = 0; i < 10; i++ )
{   // Тело цикла
}
```

Цикл по коллекции – цикл, задающий выполнение некоторой операции для объектов из заданного множества без явного указания порядка перечисления этих объектов. В качестве множества могут выступать, в частности, массив и список.

```
foreach ( type item in set )
{   // Тело цикла
}
```

Безусловные переходы и досрочные выходы из цикла методом прерывания (break и continue) являются плохим стилем. Первые, являвшиеся в 80-х (в языке Basic) необходимостью, вели к т.н. спагетти-коду, чего категорически следует избегать. Досрочный выход из цикла правильнее реализовывать через прохождение условия, используя дополнительный флаг в условии.

```
bool flag = false; // флаг досрочного завершения
while(condition && !flag)
{
   // Выполнение действий
   if (SomeError)      flag = true;
}
flag = true;
for (int i = 0; i < 200 & flag; i++)
{
   // Сложные вычисления
```

```
    if (HitCondition)       flag = false;
}
i--;
```

Инструкция try-catch-finally представляет собой метод ветвления алгоритма в сочетании с обработкой возможных ошибок. Она состоит из блока try с защищаемым кодом, за которым следует один или несколько блоков catch, в которых определяются обработчики различных исключений, возможных в блоке try, и необязательный блок finally, который будет выполнен вне зависимости от того, как был осуществлен выход из блока try.

Блок try выполняется до момента возникновения исключения (например, попытка приведения null к типу вызовет исключение NullReferenceException) или до своего успешного завершения. В одной инструкции try-catch можно использовать несколько блоков catch; важен порядок их следования (от частного к общему), поскольку исключения будут проверяться именно в заданном порядке. При возникновении исключения среда выполнения (CLR) ищет оператор catch, который обрабатывает данное исключение. Если выполняющийся в данный момент метод не содержит подходящего блока catch (к примеру, все они обрабатывают другие, узкие исключения), то CLR производит поиск в методе, который вызвал текущий метод, и далее по стеку вызовов. Если блок catch не найден, то среда CLR отображает пользователю сообщение о необработанном исключении и останавливает выполнение программы. Оператор catch можно использовать без аргументов для перехвата всех типов исключения, однако на практике используется аргумент базового типа исключений System.Exception и/или производный от него тип (типы) для перехвата только ожидаемых типов исключений (например, ArgumentNullException). Блок finally позволяет освободить все ресурсы, выделенные в блоке try, а также выполнить код, который должен выполняться даже в случае возникновения исключения. Рассмотрим пример употребления. Контрольный вопрос: что произойдёт, если инициализировать массив StrLst значением null?

```
List<string> StrLst = new List<string>() { "2", "-3,5", "1,25e-02", "pi", "3.5", "" };
List<double> NumLst = null;
string msg = ""; double sum = 0;
try
{
    int i, n = StrLst.Count;
    NumLst = new List<double>();
    for (i = 0; i < n; i++)
    {
        NumLst.Add(Convert.ToDouble(StrLst[i]));
        sum += NumLst[i];
    }
    NumLst.Sort();
    for (i = 0; i < n; i++) msg += NumLst[i] + ' ';
}
catch (FormatException ex)
{msg = "Input error: element ' " + StrLst[i] + " ' is not suitable as double: " + ex.Message;
}
catch (Exception ex)
{msg = "Error: " + ex.Message;
}
finally
{if (NumLst != null) NumLst.Clear();
}
```

Структура программы на C#

В начале файла с кодом указываются используемые пространства имён, следом объявляется текущее пространство имён (namespace) – это область видимости объектов (scope). Оно служит для группировки классов и структур, для создания глобально уникальных имён классов и для ограничения доступа к классам и структурам из внешнего пространства имён (например, классы с модификаторами public и private). Для использования в программе классов из других пространств имен необходимо указать их с директивой using. К примеру, не объявив использование System.Drawing, нельзя использовать краткое собственное имя класса Bitmap: для создания экземпляра класса понадобится употребление при каждом использовании конструкции с полными путями к классу вида System.Drawing.Bitmap bmp = new System.Drawing.Bitmap();. Перечень используемых в проекте библиотек, как системных (входящих в .NET Framework), так и пользовательских либо сторонних, можно просмотреть и редактировать в Solution Explorer в списке References (компоненты, библиотеки, проекты).

Приведём некоторые общеупотребимые пространства имён:

System – фундаментальные и базовые классы;

System.IO – реализация ввода-вывода, включая чтение и запись данных из/в файлы, обработку входящих и исходящих потоков данных;

System.Collections.Generic – интерфейсы и классы, определяющие универсальные коллекции для создания строго типизированных коллекций (к примеру, шаблон класса List<T> – работа со списками объектов класса T);

System.Windows.Forms – классы для создания приложений Windows (пользовательские интерфейсы – формы/окна);

System.ComponentModel – типы, реализующие поведение компонентов и элементов управления во время разработки и выполнения;

System.Threading – работа с потоками;

System.Drawing – возможности графического интерфейса GDI+ (работа с изображениями, рисование);

System.Data – классы для работы с данными из разных источников (основные – DataTable, DataRow, Data Column и DataView, с которыми понадобится работать при использовании DataGridView из System.Windows.Forms);

System.Linq – поддержка запросов с использованием LINQ (Language INtegrated Query – добавленный в языки программирования платформы .NET Framework язык запросов с SQL-подобным синтаксисом; дает возможность функционального программирования непосредственно в коде программы).

Для работы в консоли понадобятся методы из System: string Console.ReadLine() – считывание строки, Console.ReadKey() – считывание любой нажатой клавиши, Console.Write(string), Console.Write(string, params object[] arg), Console.WriteLine(string), Console.WriteLine(string, params object[] arg) – вывод текста в консоль.

Хотя многие особенности взяты из языка Java (к примеру, механизм сборки мусора и отсутствие множественного наследования), синтаксис C# во многом похож на C++. К примеру, те же операторы присваивания =, инкремента += и декремента -=, сравнения ==, <=, >=, !=, бинарные операторы &, |, &&, ||, !, константы true, false и null, блоки кода ограничиваются фигурными скобками { }, и т.п. Точкой входа приложения является метод Main (без параметров или с массивом строковых параметров).

Для создания комментариев используются парные знаки /* и */; для однострочных комментариев используется // . В C# есть описание структур, классов,

полей (разметка вида XML), созданное с учетом возможностей систем типа doxygen: описание начинается с /// перед объявлением и содержит разметку по полям – описание метода или объекта <summary>, описание параметров <param> по имени параметра name и возвращаемое значение <returns>. Такое документирование кода удобно, так как на его основе автоматически формируются всплывающие подсказки о методах, полях, параметрах, которые упрощают разработку.

В пространстве имён могут быть объявлены классы (class), структуры (struct), перечисления (enum), интерфейсы (interface), делегаты (delegate) и вложенные пространства имён. Доступ к их членам осуществляется через точку.

Ниже приведён пример консольного приложения на C# с использованием входных параметров приложения.

```
using System;
namespace ASD_Samples
{
   static class WeatherTalkProgram
   {
/// <summary>Точка входа в программу</summary>
/// <param name="args">Первый аргумент - имя, второй - необходимость диалога</param>
      static void Main(string[] args)
      {
         string s;
         if (args.Length < 2)
            args = new string[] { "Anonymous", "true" };
         Console.WriteLine(Greeting(args[0]));
         if (args[1] == "1" | args[1] == "true")
         {
            Console.WriteLine("What's the weather like today?");
            s = Console.ReadLine();
            switch (s)
            {
               case "snowy": s = "I like snow!";
                  break;
               case "hot": s = "Wish it would rain down!";
                  break;
               case "freezy": s = "Have a cup of tea!";
                  break;
               case "rainy": s = "I hope tomorrow the sun will sunshine!";
                  break;
               default: s = "The weather is so changeable!";
                  break;
            }
            Console.Write(s + ' ');
         }
         Console.WriteLine("It was nice to meet you! Bye!");
         Console.ReadKey();
      }
      /// <summary>Метод приветствия</summary>
      /// <param name="name">Кого приветствовать</param>
      /// <returns>Персонализированное приветствие</returns>
      static string Greeting(string name)
      {
         return "Greetings, " + name + '!';
      }
   }
}
```

Вопросы к примеру:

1) Зачем проводится проверка количества элементов массива args?

2) Как проверить второй (№ 1) элемент массива args с помощью преобразования типов?

1.3.C#: объектная модель. Наследование, права доступа

C# – объектно-ориентированный язык программирования, следовательно, поддерживает трёх китов объектно-ориентированного программирования: инкапсуляцию, наследование и полиморфизм. Язык разработан в 1998-2001 гг. в компании Microsoft как язык разработки приложений для платформы Microsoft .NET Framework. C# поддерживает статическую типизацию, перегрузку операторов (в том числе операторов явного и неявного приведения типа), делегаты, атрибуты, события, свойства, обобщённые типы и методы, итераторы, анонимные функции и типы, лямбда-выражения, LINQ, исключения и методы-расширения.

Класс в C# является ссылочным типом и представляет собой структуру данных, предназначенную для инкапсуляции данных объекта и его поведения. Данные и поведение – это члены класса, ими могут быть методы, свойства, события и т.д. Классы могут являться наследниками от других классов (базовых) и реализовывать интерфейсы. C# не поддерживает множественное наследование: классы не могут являться наследниками нескольких классов, однако могут реализовывать несколько интерфейсов (понятие интерфейса дано ниже). Класс может содержать объявления следующих членов: конструкторов, деструкторов, констант, полей, методов, свойств, индексаторов, операторов, событий, делегатов, классов, интерфейсов, структур. Типы, объявленные в классе без модификатора доступа, по умолчанию являются private. Во вложенных классах разрешены только уровни доступа protected и private.

Свойство отличается от поля тем, что имеет черты метода. Свойство может быть virtual, его методы get и set, будучи определёнными явно, могут иметь модификаторы доступа, в т.ч. разноимённые; свойство может содержать вызов исключения, а также иметь побочные эффекты и длительное время выполнения при обращении. С точки зрения памяти, поле указывает на ячейку памяти, выделенную под переменную заданного типа, а свойство – это ссылка или ссылки (если определена только одна или обе) на методы доступа get и set. Поля могут быть переданы в качестве параметров out или ref, в отличие от свойств. В отличие от поля, значение, получаемое при последовательном чтении свойства, может меняться, несмотря на отсутствие его модификаций в коде. Это связано с тем, что модификатору get может быть приписан некоторый фрагмент кода, который будет динамически вычислять текущее значение свойства, к примеру, DateTime.Now возвращает текущее время и дату при каждом обращении.

Можно объявить универсальные классы, имеющие параметры типа. Они инкапсулируют операции, не зависящие от типа данных (к примеру, добавление элементов в коллекцию или их удаление: List<T>.Add(T), List<T>.RemoveAt(int)). Универсальные классы часто используются с коллекциями: списками, очередями, деревьями и т.п.

Структура (struct) – тип значения, используемый для инкапсуляции небольших групп связанных переменных, например, координат треугольника или характеристик номенклатуры радиоэлектронной аппаратуры. Структуры являются типами значений, следовательно, копируются при присваивании. При присвоении новой переменной значения существующей структуры выполняется копирование всех данных, а любое

изменение новой копии не влияет на данные в исходной копии (в отличие от присвоения и копирования класса стандартными методами). Это важно помнить при работе с коллекциями типов значений, такими как Dictionary<string, myStruct> (словарь). В объявлении структуры поля могут быть инициализированы только при их маркировке как const или static. У структуры не может быть конструктора без параметров (по умолчанию) или деструктора, но могут иметься конструкторы с параметрами. Структуры могут быть инициализированы без new (см. пример), при этом аналогичная инициализация класса невозможна. Структура не может наследовать от структуры или класса или быть унаследованной. Может использоваться как nullable тип. К примеру, DateTime является структурой.

Интерфейсы (interface) описывают группу функциональных возможностей, которыми могут обладать класс или структура (методы, свойства, события, индексаторы, но не константы, поля, операторы, конструкторы, деструкторы, типы, статические члены; всё это автоматически public), однако реализация (implementation) этих возможностей возлагается на класс или структуру, реализующие данный интерфейс. Интерфейс нельзя инстанциировать (создать его экземпляр). Класс или структура могут реализовать более одного интерфейса; в них должна быть написана реализация свойств и методов, поскольку от интерфейса они получают только сигнатуры. Интерфейс может наследовать от других интерфейсов.

В языке C# как наследование, так и реализация интерфейса определяются оператором : . Базовый класс должен занимать крайнее левое положение после двоеточия в объявлении класса. При реализации нескольких интерфейсов либо при наследовании и реализации интерфейсов перечисление ведется через запятую.

Модификаторы служат для изменения объявления типов и их членов. В частности, модификаторы доступа задают уровень доступа к типам и их членам: public (без ограничений доступа), protected (доступ внутри класса и из его потомков), private (доступ только в классе или структуре, где определён данный член), а также internal (доступ только в файлах данной сборки). Рассмотрим основные модификаторы в C#:

abstract – указывает на то, что класс предназначен только для использования в качестве базового класса для других классов;

const – указывает на то, что значение поля или локальной переменной не может быть изменено;

event – объявляет событие;

extern – указывает на то, что объявляется метод с внешней реализацией;

new – скрывает наследуемый член от члена базового класса;

override – указывает на то, что создается новая реализация виртуального члена, унаследованного от базового класса;

partial – определение классов, структур и методов по частям в разных местах (файлах кода) в рамках одной сборки;

readonly – объявление поля, которому можно присваивать значения только на этапе объявления или с помощью конструктора этого же класса;

sealed – указывает на то, что нельзя создавать производные классы от данного;

static – объявление члена, который принадлежит типу, а не конкретному объекту (к примеру, неинстанциируемый класс со служебными методами конверсии, сравнения и т.д.);

virtual – объявление обычного метода или метода доступа, реализацию которых можно переопределить в производном классе.

Третий базовый элемент объектно-ориентированного программирования после наследования и инкапсуляции – полиморфизм. У этого понятия два аспекта. Во-первых, во время выполнения (runtime) программы объекты производного класса могут рассматриваться как объекты базового класса. При этом объявленный тип объекта больше не идентичен его типу времени выполнения. Во-вторых, базовые классы могут определять и реализовывать виртуальные методы, а производные классы могут переопределять их, то есть предоставлять свою собственную реализацию. Если клиентский код вызывает метод во время выполнения, общеязыковая среда выполнения (CLR) ищет для объекта тип времени выполнения и вызывает переопределенный виртуальный метод. Таким образом, в исходном коде можно вызвать метод в базовом классе и вызвать выполнение метода версии производного класса. Виртуальные методы позволяют единым образом работать с группами неодинаковых объектов. В C# каждый тип является полиморфным, т.к. все типы, включая пользовательские, наследуют от Object.

Хорошим тоном является написание классов в разных файлах. Большие классы стоит разбивать на несколько файлов, для этого нужен модификатор partial. Кроме того, данный модификатор используется для добавления в существующие классы новых методов, необходимых в данном проекте.

Ниже приведён пример консольного приложения на C# с интерфейсами, структурами и перечислениями. Опишем интерфейс для сравнения и реализующий его класс: по умолчанию существующий метод Equals возвращает при сравнении ссылочных типов true, если переменные ссылаются на тот же объект (идентичность), а для проверки совпадения по значению (эквивалентности) может пригодиться рукописный метод IsEqual, аналогичный по действию оператору == для типов по значению.

```
using System;
namespace ASD_Samples
{
   /// <summary>Пробная программа с interface, struct, enum</summary>
   static class WeatherTest
   {
      interface IConsiderable { bool IsGood(); }
      public enum WeatherType { warm, hot, cold, freezy }
      public struct WeatherStruct : IConsiderable
      {
         public WeatherType WeatherLike;
         public bool Sun;
         public bool Rain;
         public WeatherStruct(WeatherType weatherType, bool sun, bool rain)
         {
            WeatherLike = weatherType; Sun = sun; Rain = rain;
         }
         /// <summary>Реализация интерфейса IConsiderable&lt;T&gt;</summary>
         public bool IsGood()
         {
            return Sun |
               WeatherLike == WeatherType.hot & Rain |
               WeatherLike == WeatherType.warm & !Rain;
         }
      }
      interface IComparable<T>
      {
```

```
        bool IsEqual(T obj);
    }
    public class WeatherClass : IComparable<WeatherClass>, IConsiderable
    {
        public WeatherType WeatherLike;
        public bool Rain;
        public WeatherClass(WeatherType weatherType, bool rain)
        {
            WeatherLike = weatherType; Rain = rain;
        }
        public bool IsGood()
        {
            return !Rain & WeatherLike != WeatherType.freezy;
        }
        public bool IsEqual(WeatherClass aWeather)
        {
            // this ссылается на объект, для которого вызвана функция.
            return this.WeatherLike == aWeather.WeatherLike &
                this.Rain == aWeather.Rain;
        }
    }
    public static void Main()
    {
        string s;
        WeatherStruct Weather1, Weather2;
        Weather1 = new WeatherStruct(WeatherType.warm, true, false);
        Weather2.WeatherLike = WeatherType.warm;
        Weather2.Sun = true; Weather2.Rain = false;
        s = "The weather1 is ";
        if (Weather1.IsGood()) s += "good."; else s += "not good.";
        Console.WriteLine(s);
        Console.WriteLine("Is weather1 equal to weather2?");
        Console.WriteLine("For structs Equals() returns " +
            Weather1.Equals(Weather2).ToString());
        Weather2.Rain = true;
        Console.WriteLine("Weather2 changed: it rains!");
        Console.WriteLine("Is weather1 equal to weather2 now?");
        Console.WriteLine("For structs Equals() returns {0}",
            Weather1.Equals(Weather2));
        WeatherClass Weather3, Weather4;
        Weather3 = new WeatherClass(WeatherType.freezy, false);
        Weather4 = new WeatherClass(WeatherType.freezy, false);
        s = "The weather3 is ";
        if (!Weather3.IsGood()) s += "not ";
        s += "good.";
        Console.WriteLine(s);
        Console.WriteLine("Is weather3 equal to weather4?");
        Console.WriteLine("For classes Equals() returns {0}; IsEqual() returns {1}",
            Weather3.Equals(Weather4).ToString(),
            Weather3.IsEqual(Weather4).ToString());
    }
  }
}
```

Вопросы к примеру:

1) В чём разница между инициализацией Weather1 и Weather2?
2) Чем отличаются методы сравнения для структуры и класса?

1.4. Лабораторная работа 1

Задание: реализовать с помощью массива и с помощью списка на основе описанных в примере ниже классов:

а) стек и очередь;

б) дек и очередь.

В каждом из указанных типов данных задать максимальный размер пула заявок, реализовать изменение позиций начала и конца (концов) очереди и стека либо дека (на массивах и на списках). Элементы массива не сдвигать, изменять позиции начала и конца (концов) так, чтобы получился циклический список (первая заявка покинула очередь, значит, первой, т.е. в голове очереди, становится следующая позиция в массиве; если вдобавок все, кроме первой, позиции заняты заявками, последняя заявка, добавляемая в хвост, займёт первое место).

Объект в очереди (стеке, деке) имеет тип Request – заявка (в данной лабораторной работе это условное название, заявки будут рассмотрены в следующем параграфе). Заявка имеет уникальный номер. Требуется смоделировать добавление заявок, рассмотреть ситуации пустоты и полноты пула заявок (т.е. хранилища в очереди, стеке, деке).

```
using System;
using System.Collections.Generic;
namespace ADS_Samples
{
   enum TypeOfPool { Queue, Deque, Stack }
   abstract class CommonTitle
   {
      public ulong ID { get; protected set; }
      public string Title { get; protected set; }
      public CommonTitle() { }
   }
   abstract class CommonPool : CommonTitle
   {
      public int NumReq { get; protected set; }
      public int MaxSize { get; protected set; }
      public TypeOfPool PoolType { get; protected set; }
      public object Pool;
      public CommonPool() { Init(); }
      protected virtual void Init() { NumReq = 0; }
      protected void InitPool(ulong _ID, string _Title, TypeOfPool poolType, int maxSize)
      {
         ID = _ID; Title = _Title; PoolType = poolType; MaxSize = maxSize;
      }
      /// <param name="dir">параметр для дека; в иных случаях значение любое</param>
      public virtual void AddReq(Request req, bool dir) { }
      /// <param name="dir">параметр для дека; в иных случаях значение любое</param>
      public virtual void RemoveReq(bool dir) { }
      public bool IsFull { get { return NumReq == MaxSize; } }
      public bool IsEmpty { get { return NumReq == 0; } }
   }
   abstract class Queue : CommonPool
   {
      protected int FirstReq, LastReq;
      protected bool DoneQueueAssigned;
      /// <summary>Глобальный пул обработанных заявок (при его наличии), в который
```

```
        /// отсюда перемещаются завершённые либо отклонённые заявки</summary>
        public Queue DoneQueue;
        public Queue() { Init(); }
        protected override void Init() { FirstReq = -1; LastReq = -1; DoneQueueAssigned = false; }
        public void AttachGlobalQueue(Queue doneQueue)
        {
            DoneQueue = doneQueue; DoneQueueAssigned = true;
        }
    }
    sealed class QueueArr : Queue
    {
        public new Request[] Pool;
        public QueueArr(ulong _ID, string _Title, TypeOfPool poolType, int maxSize)
        {
            InitPool(_ID, _Title, poolType, maxSize); Init();
        }
        protected override void Init() { Pool = new Request[MaxSize]; }
        public override void AddReq(Request Req, bool dir) { /*ToDo*/ }
        public override void RemoveReq(bool dir) { /*ToDo*/ }
    }
    sealed class QueueLst : Queue
    {
        public new List<Request> Pool;
        public QueueLst(ulong _ID, string _Title, TypeOfPool poolType, int maxSize)
        {   InitPool(_ID, _Title, poolType, maxSize); Init(); }
        protected override void Init() { Pool = new List<Request>(); }
        public override void AddReq(Request Req, bool dir)
        {
            if (!IsFull)
            {
                Req.QueueRequest(this);  /* Внутри метода задать время начала, состояние, ссылку
на эту очередь */
                Pool.Add(Req);
                NumReq++; FirstReq = 0; LastReq++;
            }
        }
        public override void RemoveReq(bool dir)
        {
            if (!IsEmpty)
            {
                // Перемещение заявки Pool[FirstReq] в некоторый
                // внешний пул для сбора статистики впоследствии
                if (DoneQueueAssigned)
                    DoneQueue.AddReq(Pool[FirstReq], true);
                Pool.RemoveAt(FirstReq);
                NumReq--;
                if (IsEmpty) { FirstReq = -1; LastReq = -1; }
                else LastReq--;
            }
        }
    }
    ///<summary>Для глобальной нумерации заявок</summary>
    sealed class Numerator
    {
        private ulong curnum;
        /// <summary>Хранит следующий доступный порядковый номер</summary>
        public ulong CurNum { get { return curnum; } }
```

```
        public Numerator() { curnum = 0; }
        public ulong GetNewNum() { return curnum++; }
    }
}
```

Вопросы к примеру:

1) В каком порядке выполняются конструкторы классов?

2) Создайте экземпляр класса QueueLst. Какой метод Init будет выполнен? Сколько раз он выполняется при такой реализации? Что нужно исправить, чтобы не выполнять метод Init многократно и чтобы корректно инициализировать NumReq, FirstReq, LastReq, DoneQueueAssigned и Pool? Для ответа на вопрос установите точки останова на конструкторах классов и методах Init и выполните создание экземпляра класса QueueLst в режиме пошаговой отладки (меню Debug: Step Over и Step Into).

2. Системы массового обслуживания

2.1. Системы массового обслуживания. Обработка заявок

Система массового обслуживания (СМО, англ. Queuing model) – система, которая производит обслуживание поступающих в неё заявок (запросов на обслуживание). Обслуживание производится обслуживающими приборами. СМО подразделяются на системы с потерями (нет накопителя для ожидания начала обслуживания, при занятости всех приборов заявка теряется), системы с ожиданием (с накопителем бесконечной ёмкости для буферизации поступивших заявок, с очередью), системы с накопителем конечной ёмкости (с ожиданием и ограничениями, длина очереди не превышает ёмкости накопителя, при переполнении очередей заявка теряется). В последних двух случаях для заявки можно выделить две основные фазы: ожидание заявкой обслуживания и обслуживание заявки. Прибор обслуживания состоит из накопителя и канала обслуживания; его функционирование можно представить как процесс изменения состояний элементов во времени (переход в новое состояние означает изменение количества заявок в накопителе либо канале). Обслуживание бывает многоканальное (параллельно функционирующие приборы) и многофазное (последовательное в разных каналах). Выбор заявки из очереди на обслуживание производится с помощью так называемой дисциплины обслуживания (очередь, стек, случайный выбор, выбор с приоритетами).

Характерным для СМО является случайное появление заявок на обслуживание и завершение их выполнения в случайный момент времени, то есть стохастический принцип (отсюда название: непрерывно-стохастические модели). Поток событий – последовательность событий, происходящих одно за другим в случайные моменты времени. Выделяют поток заявок (входящий), поток обслуживания (интервалы времени обслуживания заявки), выходной поток (обслуженные каналом и покинувшие канал необслуженными заявки). Для СМО вводятся характеристики входных потоков, длительности обслуживания запросов, их функций распределения, дисциплины обслуживания. Для описания развивающейся формы случайного процесса используют Марковские случайные процессы: для каждого момента времени вероятность любого состояния в будущем зависит только от состояния системы в настоящем и не зависит от того, когда и каким образом система пришла в это состояние.

В ходе работы вычисляются такие критерии оценки работы СМО, как среднее число заявок в очереди или в системе (в очереди и в процессе обслуживания), среднее

время, проведенное заявкой в очереди или в системе, статистическое распределение этих величин, вероятность полноты и пустоты очереди и других состояний системы.

2.2.Событийная модель в C#

При реализации событийной модели ход программы определяется событиями. Существует очередь событий, в которую события добавляются и из которой они последовательно извлекаются для обработки методами-обработчиками, подписанными на данное событие. Таким образом, в основном цикле работы программы осуществляется диспетчеризация: от очереди событий к обработчикам и обратно. При работе в Windows Forms обрабатываются события движения и нажатия кнопок мыши, нажатия кнопок клавиатуры, изменения данных элементов на форме и самой формы (включая размеры, состояние – создание, обновление, закрытие) и т.д. Также можно обрабатывать сообщения других программ и потоков, события операционной системы (например, поступление сетевого пакета), пользовательские события. Как правило, в реальных задачах оказывается недопустимым длительное выполнение обработчика события, поскольку при этом программа не может реагировать на другие события. Для решения этой проблемы используются многопоточность и разделяемые ресурсы, ресурсы диспетчеризации, позволяющие осуществлять безопасный доступ к данным, используемым, к примеру, разными потоками, а также синхронизацию данных и действий.

Делегат (delegate) – ссылочный тип, указывающий на функцию с фиксированными типами параметров и возвращаемым типом. Переменной типа delegate нельзя присвоить ссылку на функцию, возвращаемое значение или параметры которой отличаются от заданных при ее объявлении. Такой переменной можно присвоить ссылку на именованный или анонимный метод. Анонимный метод позволяет сослаться на блок кода, не создавая для него функции.

```
delegate int MarkAction(int percentDone);    // Объявление делегата
class DelegateTest
{
    static int ExamStudent(int input)
    {
        int res = (int)Math.Round((double)input / 100 * 5);
        if (res < 2) res = 2;
        if (res > 5) res = 5;
        return res;
    }
    public static void Main()
    {
        MarkAction ma = ExamStudent;    // Инстанциация делегата именованным методом
        int markAGoodStudent = ma(95);  // Вызов делегата ma
        Console.WriteLine(markAGoodStudent);
        MarkAction ReexamStudent = delegate(int input)
        {                                       // Инстанциация делегата анонимным методом
            int res = (int)Math.Round((double)input / 100 * 5);
            if (res > 5) res = 5;
            if (res < 3) res = 3;
            return --res;
        };
        int markALateStudent = ReexamStudent(75);
        Console.WriteLine(markALateStudent);
        Console.ReadKey();
    }
```

```
}
```

Делегаты являются базой для событий. Событие (event) – это особый тип многоадресных делегатов, его можно вызвать только из класса или структуры, в которой оно объявлено (класс издателя). Делегаты, которые будут выполнены при возникновении события, называются обработчиками события и добавляются в переменную события – такой подход называется подпиской на событие (см. пример).

```
public class ChainingDelegates
{
   public delegate void VoidDelegate();
   public event VoidDelegate evt;
   public ChainingDelegates()
   {
      evt += delegate { Console.WriteLine("Hello"); }; // Подписка на событие.
      evt += delegate { Console.WriteLine("!"); };
      evt();                                          // Вызов события.
      Console.Read();
   }
}
```

Если на событие подписаны другие классы или структуры, их методы обработчиков событий будут вызваны, когда класс издателя вызовет событие. Рассмотрим пример, в котором дочерний класс обрабатывает событие класса-родителя. Объявим класс аргументов события QModelEventArgs. Событийной модели требуется наследование от базового класса аргументов событий EventArgs. Выполним в OnRaisePInterruptEvent дополнительную проверку, чтобы избежать возможного обращения по null. Дочерний класс подписывается на событие прерывания родителя, опубликованное родителем. При прерывании извне родитель инициирует событие, вызывая метод OnRaisePInterruptEvent, а тот, в свою очередь, вызывает обработчик(и) RaisePInterruptEvent. Поскольку на событие подписан только дочерний класс, он выполнит метод-обработчик CHandleInterruptEvent.

```
class QModelEventArgs : EventArgs
{
   public string Text;
   public ReqStateType NewState;
   public readonly DateTime TimeStamp;
   public QModelEventArgs(string s, ReqStateType newState)
   {  Text = s; TimeStamp = DateTime.Now; NewState = newState; }
}
class ParentObject
{
   public string ID;
   public bool IsOn;
   public ChildObject Child;
   public bool ChildOn;
   public ParentObject()
   {  ID = "P"; }
   public void Run()
   {  IsOn = true; ChildOn = false; Child = new ChildObject(this); Child.Run(); }
   public void Stop()
   {  if (!ChildOn) IsOn = false; }
   public void Interrupt()
   {
      OnRaisePInterruptEvent(new QModelEventArgs("Interrupt occured.",
ReqStateType.Interrupted));
   }
```

```
    // Объявление события с помощью EventHandler<T>
    public event EventHandler<QModelEventArgs> RaisePInterruptEvent;

    public void OnRaisePInterruptEvent(QModelEventArgs e)
    {
        // Создание временной копии события во избежание возможной
        // ситуации гонок, когда последний подписчик отписывается сразу
        // после проверки на null, но до вызова события
        EventHandler<QModelEventArgs> handler = RaisePInterruptEvent;
        // Событие будет null, если нет подписчиков
        if (handler != null)
        {
            // Форматирование строки для передачи в параметре CustomEventArgs
            e.Text += String.Format("Interrupted at {0}", DateTime.Now.ToString());
            // Использование оператора () для вызова события.
            handler(this, e);
        }
    }
}
class ChildObject
{
    string ID;
    public bool IsOn;
    ParentObject parent;
    public ChildObject(ParentObject Parent)
    {
        ID = "C"; this.parent = Parent;
        parent.RaisePInterruptEvent += CHandleInterruptEvent;   // Подписка на событие
    }
    public void Run()
    {   IsOn = true; parent.ChildOn = true; }
    public void Interrupt()
    {   IsOn = false; parent.ChildOn = false; parent.Stop(); }
    /// <summary>Действия, выполняемые при событии прерывания</summary>
    void CHandleInterruptEvent(object sender, QModelEventArgs e)
    {   this.Interrupt(); }
}
```

Дополним пример с очередями, чтобы использовать созданные классы для СМО. Для этого понадобятся классы с реализацией заявок.

```
/* NotProcessed - в очереди, но не обработана к концу работы; Rejected -
 получила отказ при обработке; Missed - потеряна до начала обработки */
public enum ReqStateType { Undefined, Queued, Started, Finalized, Interrupted, Continued,
NotProcessed, Rejected, Missed }
abstract class CommonReq : CommonTitle
{
    public string Task;
    public ReqStateType ReqState { get; protected set; }
    public CommonReq() { }
}
class Request : CommonReq
{
    public DateTime TimeQueued, TimeStarted, TimeFinalized;
    public string WorkNote;
    public Queue ParentQueue { get; protected set; }
    public Request(ulong id, string title, string task)
```

```
    {
        ID = id; Title = title; Task = task;
        ReqState = ReqStateType.Undefined;
    }
    public void QueueRequest(Queue curQueue)
    {
        ParentQueue = curQueue;
        ReqState = ReqStateType.Queued;
        TimeQueued = DateTime.Now;
    }
    public void StartProcessing()
    {
        ReqState = ReqStateType.Started;
        TimeStarted = DateTime.Now;
    }
    public void FinalizeProcessing()
    {
        ReqState = ReqStateType.Finalized;
        TimeFinalized = DateTime.Now;
    }
    public void GetStatistics(out string TimeInQueue, out string TimeOfProcessing)
    {
        switch (ReqState)
        {
            case ReqStateType.Finalized:
                TimeInQueue = (TimeFinalized - TimeQueued).ToString();
                TimeOfProcessing = (TimeFinalized - TimeStarted).ToString();
                break;
            case ReqStateType.Started:
                TimeInQueue = (DateTime.Now - TimeQueued).ToString();
                TimeOfProcessing = (DateTime.Now - TimeStarted).ToString();
                break;
            case ReqStateType.Queued:
                TimeInQueue = (TimeFinalized - TimeQueued).ToString();
                TimeOfProcessing = "not yet";
                break;
            default:
                TimeInQueue = "undefined";
                TimeOfProcessing = "undefined";
                break;
        }
    }
}
```

Для события добавления заявки нужен класс аргументов этого события, включающий описание заявки и позицию, на которую она была добавлена.

```
enum ReqQueuePos { First, Second, Last }
class QCUpdReqArgs : EventArgs
{
    public string msg;              // для сообщений об ошибке (если таковые будут)
    public string ReqText;
    public ReqQueuePos ReqPos;
    public QCUpdReqArgs(string reqText, ReqQueuePos reqPos)
    {   ReqText = reqText; ReqPos = reqPos; msg = ""; }
}
```

В классе Queue объявим событие и реализуем метод вызова события:

```
// Объявление события с помощью EventHandler<T>
```

```
    public event EventHandler<QCUpdReqArgs> RaiseQAddReqEvent;
    public void OnRaiseQAddReqEvent(QCUpdReqArgs e)
    {
       // Создание временной копии события во избежание возможной
       // ситуации гонок, когда последний подписчик отписывается сразу
       // после проверки на null, но до вызова события
       EventHandler<QCUpdReqArgs> handler = RaiseQAddReqEvent;
       // Событие будет null, если нет подписчиков
       if (handler != null)
       {
          // Форматирование строки для передачи в параметре QCUpdReqArgs
          e.msg += String.Format("RaiseQAddReqEvent at {0}", DateTime.Now.ToString());
          // Использование оператора () для вызова события.
          handler(this, e);
       }
    }
```

В метод AddReq класса QueueLst после строки LastReq++; добавится вызов события добавления заявки: OnRaiseQAddReqEvent(new QCUpdReqArgs(Req.Task + '_' + Req.Title, ReqQueuePos.Last));

2.3.Визуальная среда разработки на C#. Интерфейс в Windows Forms

При создании оконного приложения (Windows Forms Application) удобно создавать интерфейс в формах Windows, используя такие элементы управления, как Label, TextBox, Button, CheckBox, RadioButton, TrackBar, DomainUpDown, ListBox, GroupBox, TabContainer, Timer и др.

Точка входа в приложение будет выглядеть иначе: произойдёт запуск формы.

```
using System;
using System.Windows.Forms;
namespace ADS_Samples
{
   static class Program
   {
      /// <summary>Главная точка входа в приложение.</summary>
      [STAThread]
      static void Main()
      {
         Application.EnableVisualStyles();
         Application.SetCompatibleTextRenderingDefault(false);
         Application.Run(new QModelForm());
      }
   }
}
```

Создадим форму QModelForm. На форму добавим из панели Toolbox элемент Label, вручную заменим на панели Properties (выбрав надпись в выпадающем списке в верхней части панели либо кликнув по ней на форме) имя надписи на lblFormTitle с помощью свойства Name и сам текст с помощью свойства Text. Зададим размер формы, уменьшив её мышью во вкладке QModelForm.cs[Design]. Эти изменения отразятся в автоматически сгенерированном средой файле QModelForm.Designer.cs, который содержит в себе описание части класса QModelForm, в том числе код, необходимый для создания, инициализации и уничтожения формы и элементов управления, расположенных на ней.

Файл QModelForm.cs имеет лаконичный вид, пока не наполнен кодом. Если дважды кликнуть по форме в дизайнере, в QModelForm.cs появится заготовка

обработчика события загрузки формы FormLoad. В это же время в QModelForm.Designer.cs появится строчка, добавляющая подписчика на событие загрузки формы, т.е. этот метод. Добавим обработчик события двойного нажатия мыши на форму: панель Properties со свойствами формы, события, двойное нажатие на строку DoubleClick. Будем выводить в этом случае информационное сообщение. Параметры вызова диалогового окна следующие: текст сообщения, заглавие, допустимые кнопки, иконка. Также добавим обработчик события FormClosing, вызываемого перед закрытием формы (закрытию соответствует событие FormClosed). В этом случае запросим у пользователя подтверждение выхода, и при отрицательном результате установим флаг отмены закрытия. Для этого проверим значение перечисляемого типа DialogResult, возвращаемое методом Show класса диалогового окна DialogBox: если была нажата кнопка Cancel, то необходимо продолжить работу.

```
private void QModelForm_DoubleClick(object sender, EventArgs e)
{
   MessageBox.Show("Лабораторная работа №2",
      "СМО", MessageBoxButtons.OK, MessageBoxIcon.Information);
}
private void QModelForm_FormClosing(object sender, FormClosingEventArgs e)
{
   e.Cancel = !OnExiting(sender);
}
private bool OnExiting(object sender)
{
   bool res = MessageBox.Show("Вы уверены, что хотите выйти из программы?\n" +
         "Все несохраненные данные будут потеряны.", "Выход из программы",
         MessageBoxButtons.OKCancel, MessageBoxIcon.Warning) == DialogResult.OK;
   return res;
}
```

В QModelForm.Designer.cs добавится следующее:

```
   this.FormClosing += new
System.Windows.Forms.FormClosingEventHandler(this.QModelForm_FormClosing);
   this.DoubleClick += new System.EventHandler(this.QModelForm_DoubleClick);
```

Для автоматического добавления на форму наборов элементов управления, привязанных к экземплярам очереди QueueLst, создадим контейнерный класс. Привязка нужна для обновления выводимой информации о состоянии очереди с помощью обработки событий изменения состояния очереди и заявок, в частности, добавления заявки в очередь.

Необходимы данные о размерах и положении элементов, а также установки изменяемости содержимого текстовых полей (ReadOnly) и возможности переносов в них (Multiline). Данные о размере можно получить, создав вручную такую группу объектов на форме, при этом можно использовать часть кода из дизайнера в пользовательских методах класса, дополнив её инициализацией необходимых полей. Размеры окна в приведённом примере подбираются автоматически при добавлении экземпляров класса QContainer в форму.

```
class QContainer
{
   public readonly ulong ID;
   public string Title;
   public int NumReqs;
   public Queue ParentQueue;
   public QModelForm ParentForm;
   public GroupBox grpQueue;
```

```
public Label lblNumReq;
public TextBox edtNumReq;
public TextBox edtRequests;
public readonly int xpos, ypos;
public const int width = 166, height = 247;
public const int widDelta = 7, heiDelta = 5;
public QContainer(QModelForm parentForm, Queue parentQueue, ulong id, string title,
            int leftShift, int topShift, int rowIdx, int colIdx)
{
   ID = id; Title = title; ParentForm = parentForm;
   xpos = leftShift + colIdx * (width + widDelta);
   ypos = topShift + rowIdx * (height + heiDelta);

   NumReqs = 0; ParentQueue = parentQueue;
   ParentQueue.RaiseQAddReqEvent += QCHandleAddReqEvent;

   grpQueue = new GroupBox();
   lblNumReq = new Label();
   edtNumReq = new TextBox();
   edtRequests = new TextBox();
   grpQueue.SuspendLayout();

   edtRequests.Location = new System.Drawing.Point(9, 45);
   edtRequests.Name = "textBox_" + ID.ToString() + "_2";
   edtRequests.Size = new System.Drawing.Size(148, 195);
   edtRequests.TabIndex = 2;
   edtRequests.Text = "";
   edtRequests.ReadOnly = true;
   edtRequests.Multiline = true;

   edtNumReq.Location = new System.Drawing.Point(107, 19);
   edtNumReq.Name = "textBox_" + ID.ToString() + "_1";
   edtNumReq.Size = new System.Drawing.Size(50, 20);
   edtNumReq.TabIndex = 1;
   edtNumReq.Text = "0";
   edtNumReq.ReadOnly = true;

   lblNumReq.AutoSize = true;
   lblNumReq.Location = new System.Drawing.Point(7, 22);
   lblNumReq.Name = "label_" + ID.ToString() + "_1";
   lblNumReq.Size = new System.Drawing.Size(97, 13);
   lblNumReq.TabIndex = 0;
   lblNumReq.Text = "Заявок в очереди";

   grpQueue.Controls.Add(edtRequests);
   grpQueue.Controls.Add(edtNumReq);
   grpQueue.Controls.Add(lblNumReq);
   grpQueue.Location = new System.Drawing.Point(xpos, ypos);
   grpQueue.Name = "groupBox_" + ID.ToString() + "_1";
   grpQueue.Size = new System.Drawing.Size(width, height);
   grpQueue.TabIndex = 0;
   grpQueue.TabStop = false;
   grpQueue.Text = title;
   grpQueue.ResumeLayout(false);
   grpQueue.PerformLayout();
   AddToForm(colIdx);
}
```

```
    private void AddToForm(int colidx)
    {
        ParentForm.SuspendLayout();
        int pwid = ParentForm.Width;
        int phei = ParentForm.Height;
        if (pwid < xpos + width + widDelta)
            pwid = xpos + width + widDelta - 14;
        if (phei < ypos + height + heiDelta)
            phei = ypos + height + heiDelta;
        ParentForm.ClientSize = new System.Drawing.Size(pwid, phei);
        ParentForm.Controls.Add(grpQueue);
        ParentForm.ResumeLayout(false);
        ParentForm.PerformLayout();
    }
    /// <summary>Действия, выполняемые при событии AddReq.</summary>
    public void QCHandleAddReqEvent(object sender, QCUpdReqArgs e)
    {
        this.HandleAddReq(e);
    }
    private void HandleAddReq(QCUpdReqArgs e)
    {
        bool b = edtRequests.Lines.Length < 1;
        if (!b) { b = edtRequests.Lines.Length==1; if (b) b=edtRequests.Lines[0].Trim().Length==0; }
        if (b) edtRequests.AppendText(e.ReqText);
        else
            switch (e.ReqPos)
            {
                case ReqQueuePos.First:
                    edtRequests.Text.Insert(0, e.ReqText + "\r\n");
                    break;
                case ReqQueuePos.Second:
                    int i = edtRequests.GetFirstCharIndexFromLine(1);
                    edtRequests.Text.Insert(0, e.ReqText + "\r\n");
                    break;
                case ReqQueuePos.Last:
                    edtRequests.AppendText("\r\n" + e.ReqText);
                    break;
            }
        NumReqs++;
        edtNumReq.Text = NumReqs.ToString();
    }
}
```

В форму добавим шесть полей и наполним конструктор. На форме вручную (в дизайнере) создадим кнопку btnFill, которая будет запускать пример: создание четырёх очередей и привязанных к ним экземпляров QContainer.

```
    public Numerator QCNumerator, ReqNumerator;
    private const int xShift = 7, yShift = 26;
    List<QContainer> QCL;
    List<QueueLst> QsList;
    public QModelForm()
    {
        InitializeComponent();
        ReqNumerator = new Numerator();
        QCNumerator = new Numerator();
        QCL = new List<QContainer>();
        QsList = new List<QueueLst>();
```

```
    }
    private void btnFill_Click(object sender, EventArgs e)
    {
        ulong curID = QCNumerator.GetNewNum();
        string curTitle = "Desk_" + curID.ToString();
        QueueLst CurQueue = new QueueLst(curID, curTitle, TypeOfPool.Queue, 2);
        QContainer CurQC = new QContainer(this, CurQueue, curID, CurQueue.Title, xShift,
yShift, 0, 0);
        QsList.Add(CurQueue);
        QCL.Add(CurQC);
        CurQueue.AddReq(new Request(ReqNumerator.GetNewNum(),"CRED","taskA"),false);
        CurQueue.AddReq(new Request(ReqNumerator.GetNewNum(),"CRED","taskB"),false);
        CurQueue.AddReq(new Request(ReqNumerator.GetNewNum(),"CARD","taskC"),false);

        curID = QCNumerator.GetNewNum();
        curTitle = "Desk_" + curID.ToString();
        CurQueue = new QueueLst(curID, curTitle, TypeOfPool.Queue, 3);
        CurQC = new QContainer(this, CurQueue, curID, CurQueue.Title, xShift, yShift, 0, 1);
        QsList.Add(CurQueue);
        QCL.Add(CurQC);
        CurQueue.AddReq(new Request(ReqNumerator.GetNewNum(),"CARD","taskD"),false);
        CurQueue.AddReq(new Request(ReqNumerator.GetNewNum(),"CARD","taskE"),false);
        CurQueue.AddReq(new Request(ReqNumerator.GetNewNum(),"CRED","taskF"),false);

        curID = QCNumerator.GetNewNum();
        curTitle = "Desk_" + curID.ToString();
        CurQueue = new QueueLst(curID, curTitle, TypeOfPool.Queue, 4);
        CurQC = new QContainer(this, CurQueue, curID, CurQueue.Title, xShift, yShift, 0, 2);
        QsList.Add(CurQueue);
        QCL.Add(CurQC);
        CurQueue.AddReq(new Request(ReqNumerator.GetNewNum(),"ACNT","taskG"),false);

        curID = QCNumerator.GetNewNum();
        curTitle = "Desk_" + curID.ToString();
        CurQueue = new QueueLst(curID, curTitle, TypeOfPool.Queue, 3);
        CurQC = new QContainer(this, CurQueue, curID, CurQueue.Title, xShift, yShift, 1, 0);
        QsList.Add(CurQueue);
        QCL.Add(CurQC);
        CurQueue.AddReq(new Request(ReqNumerator.GetNewNum(),"XCHG","taskH"),false);
    }
```

Для генерации повторяющихся отсроченных событий в режиме реального времени можно использовать класс System.Windows.Forms.Timer. Создадим класс QTimer, включающий в себя таймер, идентификатор, ссылку на владельца и описание события срабатывания таймера; в конструкторе инициализируем таймер aTimer, зададим интервал его срабатывания, активируем таймер и создадим подписку данного экземпляра класса QTimer на событие срабатывания таймера aTimer. Также создадим класс пользовательских аргументов события срабатывания таймера.

```
using System;
using System.Collections.Generic;
using System.Windows.Forms;
namespace ADS_Samples
{
   class TimerEventArgs : EventArgs
   {
      public int TargetID;
```

```
        public readonly DateTime TimeStamp;
        public object Target;
        public string Msg;
        public TimerEventArgs(int targetID, DateTime timeStamp, object target, string msg)
        {
            TargetID = targetID; TimeStamp = timeStamp; Target = target; Msg = msg;
        }
    }
    class QTimer
    {
        public object Parent;
        public int TargetID;
        public Timer aTimer;
        public QTimer(object parent, int targetID, int interval)
        {
            Parent = parent;
            TargetID = targetID;
            aTimer = new Timer();
            aTimer.Tick += ATimer_Tick;
            aTimer.Interval = interval;
            aTimer.Enabled = true;
        }
        private void ATimer_Tick(object sender, EventArgs e)
        {
            aTimer.Enabled = false;
            Parent.OnRaiseTimerTickEvent(new TimerEventArgs(TargetID, DateTime.Now,
                Parent /*использовать привязку к нужному объекту*/, "Timer for target: Tick"));
        }
    }
}
```

В зависимости от логики использования в качестве Parent и/или Target может выступать и очередь (очереди), и заявка, и QModelForm. Туда будет помещена ответная часть события:

```
        // Объявление события с помощью EventHandler<T>
        public event EventHandler<TimerEventArgs> RaiseTimerTickEvent;

        public void OnRaiseTimerTickEvent(TimerEventArgs e)
        {
            // Создание временной копии события во избежание возможной
            // ситуации гонок, когда последний подписчик отписывается сразу
            // после проверки на null, но до вызова события
            EventHandler<TimerEventArgs> handler = RaiseTimerTickEvent;
            // Событие будет null, если нет подписчиков
            if (handler != null)
            {
                // Форматирование строки для передачи в параметре TimerEventArgs
                e.Msg += String.Format("RaiseTimerTickEvent at {0}", e.TimeStamp.ToString());
                // Использование оператора () для вызова события.
                handler(this, e);
            }
        }
```

2.4.Лабораторная работа 2

Смоделировать обработку в банковском отделении заявок типа XCHG, CARD, CRED, ACNT. Отделы – обменные и операционные кассы (NX и NO шт.), первые обрабатывают заявки типа XCHG, вторые – все или некоторые из остальных типов заявок. Смоделировать поток заявок, поступающих через случайные промежутки времени за рабочий день. Задаётся количество NT всех заявок, пропорция заявок разного типа, математическое ожидание времён обработки одной заявки каждого типа. Поступление заявок прерывается за полчаса до конца рабочего дня, в течение ещё получаса заявки из очереди принимаются к обработке.

Реализовать механизмы прерывания и возобновления обработки заявки, отказа в обслуживании во время обслуживания (к примеру, бумаги подготовлены, но заявка не может быть выполнена ввиду недостатка средств), изменения состояния заявки, присвоения и/или изменения значений временных полей класса заявки, работы с пулом (пулами) заявок. Выводить статистику: времена постановки в очередь, начала обработки, конца обработки, время, проведённое заявкой в очереди, время обработки заявки (за вычетом времени прерывания, если таковое случилось), также времена прерывания и возобновления обработки заявки; количество обработанных и необработанных заявок по типам. При разном количестве и пропорциях заявок моделировать нормальный режим работы, отказы в обслуживании конца очереди по одному или нескольким типам заявок. В системе есть 2 обменных кассы и 5 операционных. Смоделировать работу СМО, операционные кассы – по вариантам.

а) Кассы 1, 2, 3 обрабатывают заявки типа CARD и CRED, 4 и 5 – ACNT и CRED. Смоделировать СМО разными очередями и выборкой из одной общей очереди.

б) Кассы 1, 2 обрабатывают заявки типа CARD и CRED, 3 и 4 – ACNT и CRED, 5 – CARD, CRED, ACNT. СМО имеет единую очередь. Заявки типа CRED обрабатываются вне очереди, но не более 2 подряд: в этом случае за ними следуют 5 заявок другого типа. В статистику включить количество заявок каждого типа, обработанных каждой кассой.

в) 5 касс принимают 3 типа заявок в порядке общей очереди. В NA % случаев происходит досрочный отказ в обработке заявки, в NF % – досрочное выполнение заявки. Если величина очереди в нужную кассу, включая обменную, превышает пороговую величину SZM (клиент оценивает очередь как «большую»), приходящий клиент уходит (заявка получает статус ReqStateType.Missed из примера). Отказы и пропущенные заявки также выводить в статистику.

г) 5 касс принимают 3 типа заявок в порядке общей очереди. В NR % случаев в середине выполнения заявки клиента отсылают к банкомату, что занимает ещё половину времени выполнения той же заявки, после чего клиент становится в начало очереди к той же кассе, в которой обслуживался, и будет обслужен после клиента, обслуживаемого в ней в настоящий момент. Выполнение его заявки продолжается с момента прерывания. Время прерывания и возобновления обработки заявки выводить в статистику.

д) 5 касс принимают 3 типа заявок в порядке общей очереди с приоритетами обслуживания: после подряд обработанных 3 заявок типа CARD или ACNT следует заявка типа CRED, если таковая есть в очереди. Моделировать разными очередями.

е) 5 касс принимают 3 типа заявок в порядке общей очереди. При выполнении заявки типа CRED или ACNT в NC % случаев тот же клиент встаёт в конец очереди (на текущий момент) с заявкой типа XCHG, и наоборот. Выводить

количество таких клиентов, которые были обслужены с обеими заявками или только с одной из двух, в зависимости от типа заявок.

ж) Кассы 1, 2 обрабатывают заявки типа CARD, 3, 4 и 5 – ACNT и CRED. При выполнении заявки типа CRED или ACNT в NB % случаев тот же клиент встаёт в конец очереди (на текущий момент) с заявкой типа CARD или XCHG. Выводить количество таких клиентов, которые были обслужены со всеми своими заявками или только с частью, в зависимости от типа заявок.

з) Кассы 1, 2, 3 обрабатывают заявки типа CARD и CRED, 4 и 5 – ACNT и CRED. Рассмотреть разное соотношение заявок разных типов. Для каждой последовательности заявок моделировать результаты обработки заявок 5, 4, 3 кассами; сравнить количество обработанных и необработанных заявок в этих случаях. Провести тот же эксперимент для случая обработки 5 кассами 3 типов заявок в порядке общей очереди.

и) 5 касс принимают 3 типа заявок в порядке общей очереди. В NP2 % случаев заявка состоит из 2 заявок, в NP3 % случаев – из 3 (в любых сочетаниях), которые либо обрабатываются подряд в той же кассе, либо, если присутствует обменная операция, клиент последовательно встаёт с ней в конец очереди в нужную кассу. Выводить статистику по заявкам, а также по клиентам.

к) В 5 кассах сидят менеджеры, являющиеся предпочтительными для клиентов. Клиент с любым типом заявки из трёх встаёт в очередь предпочтительно к «своему» менеджеру, однако, если к нужному менеджеру большая очередь, он будет направлен туда, где очередь меньше. Моделировать разные соотношения клиентов по менеджерам.

л) 5 касс принимают 3 типа заявок в порядке общей очереди. В ND % случаев клиент берёт 2 талончика в разные кассы (в обменную кассу и в операционную) и встаёт в две очереди. Если клиент обслуживается в одной кассе, а подходит его очередь в другую кассу, после завершения обслуживания ему придётся брать второй талончик заново. Выводить статистику по заявкам и по клиентам.

м) Кассы 1, 2, 3 обрабатывают заявки типа CARD и CRED, 4 и 5 – ACNT и CRED. В ND % случаев клиент берёт 2 талончика в разные кассы, включая обменные, и встаёт в две очереди. Если клиент обслуживается в одной кассе, а подходит его очередь в другую кассу, ему придётся брать талончик в последнюю после того, как он будет обслужен в первой.

Реализовать СМО на основе примеров из предыдущей лабораторной работы и из этого параграфа. Реализовать событийную модель.

Реализовать пользовательский интерфейс. Для каждой кассы добавлять на форму унифицированный набор объектов (заглавие с идентификатором и типами принимаемых заявок, текстовые поля с предельным и текущим количеством заявок в очереди, текстовое поле с текстовым представлением очереди заявок, кнопки ручного добавления заявки, и т.д., а также элемент GroupBox, в котором эти элементы расположены) и контейнер, ссылающийся на вышеперечисленные объекты, а также содержащий привязку к собственно экземпляру реализованного класса кассы. Реализовать возможность автоматической генерации последовательности заявок, а также ручное их добавление.

Приложение. Среда разработки Visual Studio

Создание проекта:
File – New – Project – Visual C# – Console application | Windows Forms Application

Окна просмотра: Output – вывод компилятора при сборке программы. Подробности причин некомпилируемости выводятся в Error List. Эти окна за счёт настраиваемых свойств Dock и Auto Hide удобно скрываются в нижней, левой или правой части интерфейса среды программирования и всплывают при наведении мыши либо сами при определённых событиях, к примеру, Output – при сборке программы. Перечень таких окон есть в пункте меню View. Наиболее употребимы следующие: 1) Solution Explorer – обзор решения (Solution), его ссылок (References), входящих в него проектов и файлов кода в виде дерева; 2) Toolbox – панель инструментов с компонентами Windows Forms, которые можно перетащить на форму; 3) Properties – окно свойств (файла, объекта Windows Forms, добавленного пользователем в режиме дизайнера на форму, и т.п.); 4) Watch List – значения переменных, необходимых к просмотру разработчиком в процессе отладки программы.

При работе с Windows Forms есть два режима добавления объектов на форму и их изменения: 1) путём перетаскивания элементов с панели инструментов и их дальнейшего редактирования на форме с помощью окна свойств; 2) путём объявления этих объектов и изменения их свойств программно, то есть в файлах с кодом. В первом случае при пользовательских (в визуальном режиме) действиях изменения отображаются в коде дизайнера (Designer) формы. В дереве файлов проекта, если развернуть список, к примеру, Form1.cs (клик левой кнопкой – откроется дизайнер, правой – пунктом View Code будет предложено посмотреть код формы), отобразится Form1.Designer.cs – автоматически сгенерированный средой разработки код, описывающий созданные в визуальном режиме объекты формы Form1.

Настройки решения и проекта позволяют задавать пространство имён по умолчанию, версию платформы .NET Framework, очерёдность сборки проектов и т.д.

Горячие клавиши:

Ctrl+Tab – увеличить отступы всех выделенных строк на одну табуляцию;

Ctrl+Shift+Tab – уменьшить отступы всех выделенных строк на одну табуляцию;

Ctrl+Space – вывести список допустимых конструкций (где это возможно, фокусировка ставится на метод по умолчанию, к примеру, при сравнении переменной с одной из констант коллекции автоматически будет предложено имя коллекции; ввод уточняется по первым введённым символам);

Ctrl+Shift+Space – вывести имена и типы параметров, требуемых для данного метода (сигнатуру метода);

F5 – отладка, F10 – пошаговая отладка без захода в методы, F11 – пошаговая отладка с заходом в методы.

Полный перечень можно найти, набрав Visual Studio Keybinding в поиске на сайте http://www.microsoft.com/

Документация

Базы справочной документации к Visual Studio и, в частности, C# есть в MSDN (Microsoft Developers Network): в составе дистрибутива (документация будет соответствующей версии) Visual Studio либо на сайте http://msdn.microsoft.com/ (все доступные версии документации, предыдущие и последние).

Рекомендуемая литература

1. Вирт Н. Алгоритмы и структуры данных. — М.: Мир, 1989. — 360 с.
2. Ахо А.В., Хопкрофт Дж., Ульман Дж. Д. Структуры данных и алгоритмы. — М.: Издательский дом "Вильямс", 2003. — 384 с.
3. Писсанецки С. Технология разреженных матриц. — М.: Мир, 1988. — 356 с.
4. Шень А. Программирование: теоремы и задачи. — М.: МЦМНО, 2004. — 296 с.
5. Кормен Т.Х., Лейзерсон Ч.И., Ривест Р.Л., Штайн К. Алгоритмы: построение и анализ, 2-е издание. — М.: Издательский дом "Вильямс", 2005. — 1296 с.
6. Петцольд Ч. Программирование в тональности C#. — М.: Русская Редакция, 2004. — 512 с.
7. Рихтер Дж. Программирование на платформе Microsoft .NET Framework. — М.: Русская Редакция, 2005. — 486 с.
8. Бокс Д., Селлз К. Основы платформы .NET, том 1. Общеязыковая исполняющая среда. — М.: Издательский дом "Вильямс", 2003. — 288 с.
9. Нейгел К., Ивьен Б., Глин Дж., Уотсон К. C# 4.0 и платформа .NET для профессионалов. — М.: ООО "И.Д. Вильямс", 2011. — 1440 с.

Учебное издание

МОДЕЛИРОВАНИЕ СИСТЕМ МАССОВОГО ОБСЛУЖИВАНИЯ СРЕДСТВАМИ ЯЗЫКА C#

Составители:
ВОЛКОВА Лилия Леонидовна
КЛЫШИНСКИЙ Эдуард Станиславович

Редактор С.П. Клышинская
Технический редактор О.Г. Завьялова

Подписано в печать 26.12.12. Формат 60x84/16.
Бумага офсетная. Печать – ризография.
Усл. печ. л. 2.0. Уч.-изд. л.1,8. Тираж 50 экз.
Заказ . Бесплатно. Изд. № 70.

Московский институт электроники и математики
Национального исследовательского университета «Высшая школа экономики».
109028 Москва, Б. Трехсвятительский пер., 3.
Редакционно-издательский отдел
Московского института электроники и математики
Национального исследовательского университета «Высшая школа экономики».
113054 Москва, ул. М. Пионерская, 12.