ЛАБОРАТОРНАЯ РАБОТА №2
АССОЦИАЦИИ МЕЖДУ КЛАССАМИ. РАСПРЕДЕЛЕНИЕ ОБЯЗАННОСТЕЙ

Цель работы: формирование практических навыков реализации объектно-ориентированной модели и распределения обязанностей между объектами модели.

Задачи:

1. Изучить интерфейс класса List<T> (конструкторы, свойства, методы)
2. Изучить часто используемые классы библиотеки FCL
3. Получить навыки реализации зависимостей между классами в объектно-ориентированной модели

Результатами работы являются:

- Библиотека классов, моделирующая предметную область согласно варианту задания
- Набор модульных тестов для демонстрации возможностей разработанной библиотеки классов
- Подготовленный отчет

КРАТКАЯ ХАРАКТЕРИСТИКА ОБЪЕКТА ИЗУЧЕНИЯ, ИССЛЕДОВАНИЯ

List<T> и ArrayList

Обобщенный класс List и необобщенный класс ArrayList предоставляют массив объектов с возможностью динамического изменения размера и относятся к числу наиболее часто применяемых классов коллекций. Класс ArrayList реализует интерфейс IList, в то время как класс List<T> - интерфейсы IList и IList<T> (а также IReadOnlyList<T> - новую версию, предназначенную только для чтения). В отличие от массивов, все интерфейсы реализованы открытым образом, а методы вроде Add и Remove открыты и работают так, как можно было ожидать. Классы List<T> и ArrayList работают за счет поддержки внутреннего массива объектов, который заменяется более крупным массивом при достижении предела емкости. Добавление элементов производится эффективно (т.к. в конце обычно имеются свободные позиции), но вставка элементов может быть медленной (потому что все элементы после точки вставки должны быть сдвинуты для освобождения позиции). Как и в случае массивов, поиск будет эффективным, если метод BinarySearch используется со списком, который был отсортирован, но в противном случае он не эффективен, поскольку каждый элемент должен проверяться индивидуально.

Класс List<T> в несколько раз быстрее класса ArrayList, если T является типом значения, т.к. List<T> избегает накладных расходов, связанных с упаковкой и распаковкой элементов.

Классы List<T> и ArrayList предоставляют конструкторы, которые принимают существующую коллекцию элементов: они копируют каждый элемент из нее в новый экземпляр List<T> или ArrayList:

```
public class List<T>: IList<T>, IReadOnlyList<T>
{
 public List ();
 public List (IEnumerable<T> collection);
 public List (int capacity);
```

```
// Добавление и вставка
public void Add (T item);
public void AddRange (IEnumerable<T> collection);
public void Insert (int index, T item);
public void InsertRange (int index,
    IEnurnerable<T> collection);

// Удаление
bool Remove (T item);
void RemoveAt (int index);
public void RemoveRange (int index, int count);
int RemoveAll (Predicate<T> match);

// Индексация
public T this [int index] { get; set; }
public List<T> GetRange (int index, int count);
public Enumerator<T> GetEnurnerator();

// Экспортирование, копирование и преобразование
public т [ J ToArray ();
public void CopyTo (T [] array);
public void CopyTo (T[] array, int arrayIndex);
public void CopyTo (int index, T[] array,
    int arrayIndex, int count);
public ReadOnlyCollection<T> AsReadOnly();
public List<TOutput> ConvertAll<TOutput>
    (Converter <T,TOutput> converter);

// Другие public void Reverse();
public int Capacity { get;set};
public void TrimExcess();
public void Clear();
}
public delegate TOutput Converter <Tinput, TOutput>
    (Tinput input);
```

В дополнение к этим членам класс List<T> предлагает версии экземпляра для всех методов поиска и сортировки из класса Array.

В показанном ниже коде демонстрируется работа свойств и методов List:

```
List<string> words = new List<string>();
words.Add ("melon");
words.Add ("avocado");
words.AddRange (new[] { "banana", "plum"} );
words.Insert (O, "lemon");
words.InsertRange (O, new[] "peach", "nashi" });
words.Remove ("melon");
words.RemoveAt (3);
words.RemoveRange (O, 2);

// Удалить все строки, начинающиеся с n:
words.RemoveAll (s => s.StartsWith ("n"));
Console.WriteLine (words [O]);
Console.WriteLine (words [words.Count -1]);
foreach (string s in words) Console.WriteLine (s);
List<string> subset = words.GetRange (1, 2);
string[] wordsArray = words.ToArray();

// Копировать первые 2 элемента в конец массива:
string() existing = new string (1000];
words.CopyTo (0, existing, 998, 2);

List<string> upperCastWords =
   words.ConvertAll (s => s.ToUpper());
List<int> lengths =
   words.ConvertAll(s => s.Length);
```

Необобщенный класс ArrayList применяется главным образом для обратной совместимости с кодом .NET Framework 1.x и требует довольно неуклюжих приведений, как показано в следующем примере:

```
ArrayList al = new ArrayList(};
al.Add ("hello");
string first = (string) al [O];
string(] strArr = (string[]) al
    .ToArray(typeof (string));

int first = (int) al [O]; // Исключение
```

Класс ArrayList функционально похож на класс List<object>. Оба класса удобны, когда необходим список элементов смешанных типов, которые не разделяют общий базовый тип (кроме object). Выбор ArrayList в этом случае может обеспечить преимущество, если требуется иметь дело со списком, используя рефлексию. Рефлексия реализуется проще с помощью необобщенного класса ArrayList, чем List<object>.

Если импортировать пространство имен System.Linq, то можно будет преобразовывать ArrayList в обобщенный List путем вызова метода Cast и затем ToList:

```
ArrayList al = new ArrayList();
al.AddRange (new [] { 1, 5, 9 } ) ;
List<int> list = al.Cast<int>().ToList();
```

`Cast` и `ToList` - это расширяющие методы в классе `System.Linq.Enumerable`.

Queue<T> и Queue

Queue<T> и Queue - это структуры данных FIFO (first-in first-out - первым зашел, первым обслужен; т.е. очередь), предоставляющие методы Enqueue (добавление элемента в конец очереди) и Dequeue (извлечение и удаление элемента с начала очереди). Также имеется метод Peek, предназначенный для возвращения элемента с начала очереди без его удаления, и свойство Count (удобно для проверки, существуют ли элементы в очереди, перед извлечением из очереди). Хотя очереди являются перечислимыми, они не реализуют интерфейс

IList< T> / IList, потому что доступ к членам напрямую по индексу никогда не производится. Тем не менее, имеется метод ToArray, который предназначен для копирования элементов в массив, где к ним возможен произвольный доступ:

```
  public class Queue<T> IEnumerable<T>, ICollection,
IEnumerable
  {
        public Queue ();
        public Queue (IEnumerable<T> collection);
        //Копирует существующие элементы
        public Queue (int capacity);
        // Сокращает количество изменений размера
        public void Clear();
        public bool Contains (T item);
        public void CopyTo(T[] array,
           int arrayindex);
        public int Count { get; } public T Dequeue();
        public void Enqueue (T item);
        public Enumerator<T> GetEnumerator();
        // Для поддержки оператора foreach
        public T Peek () ;
        public T[] ToArray();
        public void TrimExcess();
  }
```

Вот пример использования класса Queue<int>:

```
  var q = new Queue<int>();
  q.Enqueue (10);
  q.Enqueue (20);
  int[] data = q.ToArray(); // Экспортирует в массив
  Console.WriteLine Jq.Count); // "2"
  Console.WriteLine (q.Peek()); // "10"
  Console.WriteLine (q.Dequeue()); // "10"
```

```
Console.WriteLine (q.Dequeue()); // "20"
Console.WriteLine (q.Dequeue());
// Генерируется исключение (очередь пуста)
```

Внутренне очереди реализованы с применением массива, который при необходимости расширяется, что очень похоже на обобщенный класс List. Очередь поддерживает индексы, которые указывают непосредственно на начальный и хвостовой элементы; таким образом, постановка и извлечение из очереди являются очень быстрыми операциями (кроме случая, когда требуется внутреннее изменение размера).

Stack<T> и Stack

Stack<T> и Stack -это структуры данных LIFO (last-in first-out -последним зашел, первым обслужен; т.е. стек), которые предоставляют методы Push (добавление элемента на верхушку стека) и Pop (извлечение и удаление элемента из верхушки стека). Также определены недеструктивный метод Peek, свойство Count и метод ToArray для экспорта данных в массив с целью произвольного к ним доступа:

```
public class Stack<T>: IEnumerable<T>,
   ICollection, IEnumerable
{
      public Stack ();
      public Stack (IEnumerable<T> collection);
      // Копирует существующие элементы
      public Stack (int capacity);
      // Сокращает количество изменений размера
      public void Clear();
      public bool Contains (T item);
      public void CopyTo (T[] array,
         int arrayindex);
      public int Count { get; }
      public Enumerator<T> GetEnumerator();
```

```
        // Для поддержки оператора foreach
        public T Peek ();
        public т Pop();
        public void Push (T item);
        public T[] ToArray();
        public void TrimExcess();
}
```

В следующем примере демонстрируется использование класса Stack<int>:

```
var s = new Stack<int>();
s.Push (1); // Содержимое стека: 1
s.Push (2); // Содержимое стека: 1,2
s.Push (3); // Содержимое стека: 1,2,3
Console.WriteLine (s.Count); //Выводит 3
Console.WriteLine (s.Peek());
//Выводит 3, содержимое стека: 1,2,3
Console.WriteLine (s. Pop ());
//Выводит 3, содержимое стека: 1,2
Console.WriteLine (S. Pop ());
//Выводит 2, содержимое стека: 1
Console.WriteLine (s. Pop ());
//Выводит 1, содержимое стека: <пуст>
Console.WriteLine (s. Pop());
//Генерируется исключение
```

Внутренне стеки реализованы с помощью массива, который при необходимости расширяется, что очень похоже на Queue<T> и List<T>.

HashSet<T> и SortedSet<T>

HashSet<T> и SortedSet<T> - это обобщенные коллекции, которые появились в .NET Framework 3.5 и .NET Framework 4.0, соответственно. Обе они обладают следующим отличительными особенностями:

- их методы Contains выполняются быстро, применяя поиск на основе хеширования
- они не хранят дублированные элементы и молча игнорируют запросы на добавление дубликатов;
- доступ к элементам по позициям невозможен.

Коллекция SortedSet<T> хранит элементы упорядоченными, тогда как HashSet<T> - нет.

Коллекция HashSet<T> реализована с помощью хеш-таблицы, в которой хранятся только ключи, а SortedSet<T> - посредством красно-черного дерева.

Обе коллекции реализуют интерфейс ICollection<T> и предлагают вполне предсказуемые методы, такие как Contains, Add и Rernove. Вдобавок предусмотрен метод удаления на основе предиката по имени RernoveWhere.

В следующем коде из существующей коллекции конструируется экземпляр HashSet<char>, затем выполняется проверка членства и перечисление коллекции (обратите внимание на отсутствие дубликатов):

```
var letters = new HashSet<char>
    ("the quick brown fox");
Console.WriteLine (letters.Contains ('t'));
// true
Console.WriteLine (letters.Contains ('j'));
// false foreach (char c in letters)
Console.Write (c); // the quickbrownfx
```

(Причина передачи значения string конструктору HashSet<char> объясняется тем, что тип string реализует интерфейс IEnumerable<char>.) Самый большой интерес вызывают операции над множествами. Приведенные ниже методы операций над множествами являются деструктивными, т.к. они модифицируют набор:

```
  public void UnionWith (IEnumerable <T> other);
  public void IntersectWith (IEnumerable <T> other);
  public void ExceptWith (IEnumerable <T> other);
  public void SymmetricExceptWith (IEnumerable <T>
other);
```

тогда как следующие методы просто запрашивают набор и потому недеструктивны:

```
  public bool IsSubsetOf (IEnumerable <T> other);
  public bool IsProperSubsetOf
     (IEnumerable <T> other);
  public bool IsSupersetOf (IEnumerable <T> other);
  public bool IsProperSupersetOf
     (IEnumerable <T> other);
  public bool Overlaps (IEnumerable <T> other);
  public bool SetEquals(IEnumerable <T> other);
```

Метод UnionWith добавляет все элементы из второго набора в исходный набор (исключая дубликаты). Метод IntersectWidth удаляет элементы, которые не находятся сразу в обоих наборах. Вот как можно извлечь все гласные из набора символов:

```
  var letters = new HashSet<char>
     ("the quick brown fox");
  letters.IntersectWith ("aeiou");
  foreach (char c in letters) Console.Write (c);
  // euio
```

Метод ExceptWidth удаляет указанные элементы из исходного набора. Ниже показано, как удалить все гласные из набора:

```
  var letters = new HashSet<char>
     ("the quick brown fox");
  letters.ExceptWith ("aeiou");
```

```
foreach (char c in letters) Console.Write (c);
// th qckbrwnfx
```

Метод SymmetricExceptWith удаляет все элементы, кроме тех, которые являются уникальными в одном или в другом наборе:

```
var letters = new HashSet<char>
   ("the quick brown fox");
letters.SymmetricExceptWith("the lazy brown fox");
foreach (char c in letters) Console.Write (c);
//quicklazy
```

Обратите внимание, что поскольку типы HashSet<T> и SortedSet<T> реализуют интерфейс IEnumerable<T>, в качестве аргумента в любом методе операции над множествами можно использовать другой тип набора (или коллекции).

Тип SortedSet<T> предлагает все члены типа HashSet<T> и вдобавок следующие:

```
 public virtual SortedSet<T> GetViewBetween
   (T lowerValue, T upperValue)
public IEnumerable<T> Reverse()
public T Min { get; }
public T Max { get; 1}
```

Конструктор типа SortedSet<T> также принимает дополнительный параметр типа IComparer<T> (а не компаратор эквивалентности). Ниже приведен пример загрузки тех же букв, что и ранее, в SortedSet<char>:

```
var letters = new SortedSet<char>
   ("the quick brown fox");
foreach (char c in letters) Console.Write (c);
// bcefhiknoqrtuwx
```

Исходя из этого, получить буквы между f и j можно так:

```
foreach (char c in letters.GetViewBetween
    ('f', 'j'))
Console.Write (c); // fhi
```

Класс StringBuilder

Класс StringBuilder (из пространства имен Systern. Text) представляет изменяемую (редактируемую) строку. С помощью StringBuilder можно добавлять (Append), вставлять (Insert), удалять (Rernove) и заменять (Replace) подстроки, не заменяя целиком объект StringBuilder.

Конструктор StringBuilder дополнительно принимает начальное строковое значение, а также стартовый размер для внутреннего объема (по умолчанию составляющий 16 символов). Если этот объем превышается, StringBuilder автоматически изменяет размеры своих внутренних структур (за счет небольшого снижения производительности) вплоть до максимального объема (который по умолчанию равен int.MaxValue).

Популярное применение класса StringBuilder связано с построением длинной строки путем повторяющихся вызовов Append. Такой подход намного более эффективен, чем выполнение множества конкатенаций обычных строковых типов:

```
StringBuilder sb = new StringBuilder();
for (int i = O; i < 50; i++)
    sb.Append (i + ", ");
```

Для получения финального результата необходимо вызвать ToString ():

```
Console.WriteLine (sb.ToString());
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 2 6, 27,
28, 2 9, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40,
41, 42, 43, 44, 45, 46, 47, 48, 49
```

В приведенном выше примере выражение i + " , " означает, что мы по-прежнему многократно выполняем конкатенацию строк. Тем не менее, это требует лишь незначительных затрат производительности, т.к. строки небольшие и они не растут с каждой новой итерацией цикла. Однако для достижения максимальной производительности тело цикла можно было бы изменить следующим образом:

```
{sb.Append (i); sb.Append (", "); }
```

Метод AppendLine выполняет добавление последовательности новой строки ("\r\n" в Windows). Метод AppendForrnat принимает смешанную форматную строку в точности как String. Forrnat.

Помимо методов Insert, Rernove и Replace (метод Replace функционирует подобно методу Replace в типе string) в классе StringBuilder определено свойство Length и записываемый индексатор для получения/установки отдельных символов.

Для очистки содержимого StringBuilder необходимо либо создать новый экземпляр StringBuilder, либо установить его свойство Length в 0.

Установка свойства Length экземпляра StringBuilder в 0 не сокращает его внутренний объем. Таким образом, если ранее экземпляр StringBuilder содержал один миллион символов, то после обнуления его свойства Length он продолжит занимать около 2 Мбайт памяти. Чтобы освободить эту память, потребуется создать новый экземпляр StringBuilder и дать возможность старому покинуть область видимости (и попасть под действие сборщика мусора).

Дата и время

Работу по представлению даты и времени выполняют три неизменяемых структуры из пространства имен System: DateTime, DateTimeOffset и TimeSpan. Специальные ключевые слова, которые бы отображались на эти типы, в языке C# отсутствуют.

Структура TimeSpan

Структура TimeSpan представляет временной интервал или время суток. В последней роли это просто "часы" (не имеющие даты), которые эквивалентны времени, прошедшему с полуночи, при условии отсутствия перехода на летнее время. Разрешающая способность TimeSpan составляет 100 наносекунд, максимальное значение примерно соответствует 10 миллионам дней, и значение может быть положительным или отрицательным. Существуют три способа конструирования TimeSpan:

- с помощью одного из конструкторов;
- путем вызова одного из статических методов From ... ;
- за счет вычитания одного экземпляра DateTime из другого.
- Ниже перечислены доступные конструкторы:

```
public TimeSpan (int hours, int minutes,
    int seconds);
public TimeSpan (int days, int hours, int minutes,
    int seconds);
public TimeSpan (int days, int hours, int minutes,
    int seconds, int milliseconds);
public TimeSpan (long ticks);
// Каждый тик равен 100ns
```

Статические методы From более удобны, когда необходимо указать интервал в каких-то одних единицах, скажем, минутах, часах и т.д.:

```
public static TimeSpan FromDays (double value);
public static TimeSpan FromHours (double value);
public static TimeSpan FromMinutes(double value);
public static TimeSpan FromSeconds (double value);
public static TimeSpan FromMilliSeconds
    (double value);
```

Например:

```
Console.WriteLine (new TimeSpan (2, 30, 0));
// 02:30:00
Console.WriteLine (TimeSpan.FrornHours (2.5) );
// 02:30:00
Console.WriteLine (TimeSpan.FromHours (-2.5));
// -02:30:00
```

В структуре TimeSpan перегружены операции < и >, а также операции+ и-. В результате вычисления приведенного ниже выражения получается значение TimeSpan, соответствующее 2,5 часам:

```
TimeSpan.FromHours(2) + TimeSpan.FromMinutes(30);
```

Следующее выражение дает в результате значение, на 1 секунду меньше 10 дней:

```
TimeSpan.FromDays(lO) - TimeSpan.FromSeconds(l);
// 9.23:59:59
```

С помощью этого выражения можно проиллюстрировать работу целочисленных свойств Days, Hours, Minutes, Seconds и Milliseconds:

```
TimeSpan nearlyTenDays = TimeSpan.FromDays(10) -
    TimeSpan.FromSeconds(1);
Console.WriteLine (nearlyTenDays.Days);
Console.WriteLine (nearlyTenDays.Hours);
Console.WriteLine (nearlyTenDays.Minutes);
Console.WriteLine (nearlyTenDays.Seconds);
Console.WriteLine (nearlyTenDays.Milliseconds);
```

В противоположность этому свойства Total ... возвращают значения типа double, описывающие весь промежуток времени:

```
Console.WriteLine (nearlyTenDays.TotalDays);
// 9.99998842592593
```

```
Console.WriteLine (nearlyTenDays.TotalHours);
// 239.999722222222
Console.WriteLine (nearlyTenDays.TotalMinutes);
// 14399.9833333333
Console.WriteLine (nearlyTenDays.TotalSeconds);
// 863999
Console.WriteLine (nearlyTenDays.TotalMilliseconds);
// 863999000
```

Статический метод Parse является противоположностью ToString, преобразуя строку в значение TimeSpan. Метод TryParse делает то же самое, но в случае неудачного преобразования возвращает false вместо генерации исключения. Класс XmlConvert также предоставляет методы для преобразования между TimeSpan и string, которые следуют стандартным протоколам форматирования XML.

Стандартным значением для TimeSpan является TimeSpan. Zero.

Структуру TimeSpan также можно применять для представления времени суток (времени, прошедшего с полуночи). Для получения текущего времени суток необходимо обратиться к свойству DateTime. Now. TimeOfDay.

Структуры DateTime и DateTimeOffset

Типы DateTime и DateTimeOffset -это неизменяемые структуры для представления даты и дополнительно времени. Их разрешающая способность составляет 100 наносекунд, а поддерживаемый диапазон лет -от 0001 до 9999. Структура DateTimeOffset появилась в .NET Framework 3.5 и функционально похожа на DateTime. Ее отличительной особенностью является сохранение дополнительно смещения UTC; это позволяет получать более осмысленные результаты при сравнении значений из разных часовых поясов.

Выбор между DateTime и DateTimeOffset

Структуры DateTime и DateTimeOffset отличаются способом обработки часовых поясов. Структура DateTime содержит флаг с тремя

состояниями, который указывает, относительно чего отсчитывается значение DateTime:

- местное время на текущем компьютере;
- UTC (современный эквивалент Гринвичского времени);
- не определено.

Структура DateTimeOffset более специфична -она хранит смещение UTC в видеTimeSpan:

```
July 01 2015 03:00:00 -06:00
```

Это влияет на сравнения эквивалентности, которые являются главным фактором при выборе между DateTime и DateTimeOffset. В частности:

- во время сравнения DateTime игнорирует флаг с тремя состояниями и считает,что два значения равны, если они имеют одинаковый год, месяц, день, часы,минуты и т.д.;
- DateTimeOffset считает два значения равными, если они ссылаются на одну и ту же точку во времени

Таким образом, DateTime считает следующие два значения различными, тогда как DateTimeOffset - равными:

```
July 01 2015 09:00:00 +00:00 (GMT) J
uly 01 2015 03:00:00 -06:00
    (местное время, Центральная Америка)
```

В большинстве случаев предпочтительнее оказывается логика эквивалентности DateTimeOffset. Скажем, при выяснении того, какое из двух международных событий произошло позже, структура DateTimeOffset даст полностью правильный ответ. Аналогично хакер, планирующий атаку типа распределенного отказа в обслуживании, определенно выберет DateTimeOffset! Чтобы сделать то же самое с помощью DateTime, потребуется приведение к единому часовому поясу (обычно UTC) повсеместно в приложении. Такой подход проблематичен по двум причинам.

– Для обеспечения дружественности к конечному пользователю UTC-значения DateTime перед форматированием требуют явного преобразования в местное время.
– Довольно легко забыть и случайно воспользоваться местным значением DateTime

Однако структура DateTime лучше при указании значения относительно к локальному компьютеру во время выполнения - например, когда нужно запланировать архивацию в каждом международном офисе на следующее воскресенье в 3 утра местного времени (когда наблюдается минимальная активность). В такой ситуации больше подойдет структура DateTime, т.к. она будет отражать местное время на каждой площадке.

ToString и Parse

Метод ToString является простейшим механизмом форматирования. Он обеспечивает осмысленный вывод для всех простых типов значений (т.е. bool, DateTime, DateTimeOffset, TimeSpan, Guid и все числовые типы). Для обратной операции в каждом из указанных типов определен статический метод Parse. Например:

```
string s = true.ToString(); // s = "True"
bool b = bool.Parse (s); / / b = true
```

Если разбор терпит неудачу, то генерируется исключение FormatException. Во многих типах также определен метод TryParse, который в случае отказа преобразования возвращает false вместо генерации исключения:

```
int i;
bool failure = int.TryParse ("qwerty", out i);
bool success = int.TryParse ("123", out i);
```

Если вы ожидаете ошибку, то вызов TryParse будет более быстрым и элегантным решением, чем вызов Parse в блоке обработки исключения.

Методы Parse и TryParse в DateTime (DateTimeOffset) и числовых типах учитывают местные настройки культуры; это можно изменить, указывая объект Cultureinfo. Часто указание инвариантной культуры является удачной идеей. Например, разбор "1. 234" в double дает 1234 для Германии:

```
Console.WriteLine (double.Parse ("1.234"));
// 1234 (в Германии)
```

Причина в том, что символ точки в Германии используется в качестве разделителя тысяч, а не как десятичная точка. Указание инвариантной культуры исправляет это:

```
double x = double.Parse
    ("1.234", CultureInfo.InvariantCulture);
```

То же самое применимо и в отношении вызова ToString:

```
string x = 1.234.ToString
    (Cultureinfo.InvariantCulture);
```

Поставщики форматов

Временами требуется больший контроль над тем, как происходит форматирование и разбор. К примеру, существуют десятки способов форматирования DateTime (DateTimeOffset). Поставщики форматов позволяют получить обширный контроль над форматированием и разбором и поддерживаются для числовых типов и типов даты/времени. Поставщики форматов также используются элементами управления пользовательского интерфейса для выполнения форматирования и разбора.

Интерфейсом для применения поставщика формата является IFormattable. Этот интерфейс реализуют все числовые типы и тип DateTime (DateTimeOffset):

```
public interface IFormattaЫe
{
    string ToString (string format,
          IFormatProvider formatProvider);
}
```

Методу ToString в первом аргументе передается форматная строка, а во втором - поставщик формата. Форматная строка предоставляет инструкции; поставщик формата определяет то, как эти инструкции транслируются. Например:

```
NumberFormatInfo f = new NumberFormatInfo();
f.CurrencySymbol = "$$";
Console.WriteLine (3.ToString ("C", f)); //$$3.00
```

Здесь "C" представляет собой форматную строку, которая указывает денежное значение, а объект NumberFormatInfo является поставщиком формата, определяющим то, каким образом должно визуализироваться денежное значение (и другие числовые представления). Этот механизм допускает глобализацию.

Если для форматной строки или поставщика указать null, то будет применен стандартный вариант. Стандартный поставщик формата-это Cultureinfo.CurrentCulture, который, если только он не переустановлен, отражает настройки панели управления компьютера во время выполнения. Например:

```
Console.WriteLine (10.3.ToString ("C", null));
// $10.30
```

Для удобства в большинстве типов метод ToString перегружен, так что null для поставщика можно не указывать:

```
Console.WriteLine (10.3.ToString ("C")); // $10.30
Console.WriteLine (10.3.ToString ("F4"));
// 10.3000(четыре десятичных позиции)
```

Вызов метода ToString без аргументов для типа DateTime (DateTimeOffset) или числового типа эквивалентен использованию стандартного поставщика формата с пустой форматной строкой.

Перечисления

Перечисление -это специальный тип значения, который позволяет указывать группу именованных числовых констант. Например:

```
public enum BorderSide {Left, Right, Top, Bottom}
```

Это перечисление можно применять следующим образом:

```
BorderSide topSide = BorderSide.Top;
bool isTop = ( topSide == BorderSide. Top); // true
```

Каждый член перечисления имеет лежащее в его основе целочисленное значение. По умолчанию:

- лежащие в основе значения относятся к типу int;
- членам перечисления присваиваются константы 0, 1, 2 ... (в порядке их объявления).

Можно указать другой целочисленный тип:

```
public enum BorderSide : byte { Left, Right,
    Top, Bottom}
```

Для каждого члена перечисления можно также указывать явные лежащие в основе значения:

```
public enum BorderSide: byte {Left=1, Right=2,
   Top=10, Bottom=11}
```

Компилятор также позволяет явно присваивать значения некоторым членам перечисления. Члены, которым значения не были присвоены, получают значения на основе инкрементирования последнего явно указанного значения. Предыдущий пример эквивалентен следующему коду:

```
public enum BorderSide : byte
{Left=1, Right, Top=10, Bottom}
```

Преобразования перечислений

Экземпляр перечисления может быть преобразован в и из лежащего в основе целочисленного значения с помощью явного приведения:

```
int i = (int) BorderSide.Left;
BorderSide side = (BorderSide) i;
bool leftOrRight = (int) side <= 2;
```

Можно также явно приводить один тип перечисления к другому. Предположим, что определение HorizontalAlignment выглядит следующим образом:

```
public enum HorizontalAlignment
{
      Left = BorderSide.Left,
      Right = BorderSide.Right,
      Center
}
```

При трансляции между типами перечислений используются лежащие в их основе целочисленные значения:

```
HorizontalAlignment h = (HorizontalAlignment)
   BorderSide.Right;
// То же самое, что и:
HorizontalAlignment h = (HorizontalAlignment)
   (int) BorderSide.Right;
```

Числовой литерал 0 в выражении enum трактуется компилятором особым образом и явного приведения не требует:

```
BorderSide b = 0; / / Приведение не требуется
if (b == 0) ...
```

Существуют две причины для специальной трактовки значения 0:

- первый член перечисления часто используется как "стандартное" значение;
- для типов комбинированных перечислений значение 0 означает "отсутствие флагов".

Перечисления флагов

Члены перечислений можно комбинировать. Чтобы предотвратить неоднозначности, члены комбинируемого перечисления требуют явного присваивания значений, обычно являющихся степенью двойки. Например:

```
[Flags]
public enum BorderSides { None=0, Left=l, Right=2,
   Top=4, Bottom=8 }
```

Для работы со значениями комбинированного перечисления применяются побитовые операции, такие как | и &. Они имеют дело с лежащими в основе целыми значениями:

```
BorderSides leftRight = BorderSides
   .LeftBorderSides.Right;
if ( (leftRight & BorderSides.Left) != 0)
   Console.WriteLine ("Includes Left");
   // Includes Left
string formatted = leftRight.ToString();
// "Left, Right"
BorderSides s = BorderSides.Left;
s |BorderSides.Right;
Console.WriteLine (s == leftRight);
s ^= BorderSides.Right;
Console.WriteLine (s);
```

По соглашению к типу перечисления должен всегда применяться атрибут Flags, когда члены перечисления являются комбинируемыми. Если объявить такое перечисление без атрибута Flags, то комбинировать члены по-прежнему можно будет, но вызов ToString на экземпляре перечисления будет выдавать число, а не последовательность имен.

По соглашению типу комбинируемого перечисления назначается имя во множественном, а не единственном числе.

Для удобства члены комбинаций могут быть помещены в само объявление перечисления:

```
[Flags]
public enum BorderSides
{
      None=1, Left=1, Right=2,
      Top=4, Bottom=B,
      LeftRight = Left I Riqht,
      TopBottom = Top I Bottom
      All = LeftRiqht I TopBottom
}
```

Операции над перечислениями

Ниже указаны операции, которые могут работать с перечислениями: += ! = < > <= >= + ++ -- sizeof &

Побитовые, арифметические и операции сравнения возвращают результат обработки лежащих в основе целочисленных значений. Сложение разрешено для перечисления и целочисленного типа, но не для двух перечислений.

Проблемы безопасности типов

Рассмотрим следующее перечисление:

```
public enum BorderSide { Left, Right, Top, Bottom}
```

Поскольку тип перечисления может быть приведен к лежащему в основе целому типу и наоборот, фактическое значение может выходить за пределы допустимых границ для членов перечисления. Например:

```
BorderSide b = (BorderSide) 12345;
Console.WriteLine (b); // 12345
```

Побитовые и арифметические операции могут давать в результате аналогичные недопустимые значения:

```
BorderSide b = BorderSide.Bottom;
b++; // Ошибки не возникают
void Draw (BorderSide side)
{
      if {side == BorderSide.Left) { . . . }
      else if (side == BorderSide.Right) { . . . }
      else if {side == BorderSide.Top) { . . . }
      else
}
```

Одно из решений заключается в добавлении дополнительной конструкции else:

```
…
else if (side == BorderSide.Bottom} ...
else throw new ArgurnentException ( "Invalid BorderSide: " + side, "side");
 // Недопустимое значение BorderSide
```

Еще один обходной прием предусматривает явную проверку значения перечисления на предмет допустимости. Эту работу выполняет статический метод Enum. IsDefined:

```
BorderSide side = (BorderSide) 12345;
Console. Wri teLine (Enum. IsDefined
   { typeof (BorderSide) , side) ) ; // False
```

К сожалению, метод Enum. IsDefined не работает с перечислениями флагов. Тем не менее, следующий вспомогательный метод (трюк, зависящий от поведения Enum. ToString ()) возвращает true, если заданное перечисление флагов является допустимым:

```
static bool IsFlagDefined (Enum e)
{
      decimal d;
      return !decimal.TryParse(e.ToString(), out d);
}

[Flags]
public enum BorderSides { Left=1, Right=2,
   Top=4, Bottom=8}
static void Main()
{
      for (int i = 0; i <= 16; i++)
      {
            BorderSides side = (BorderSides)i;
            Console.WriteLine
               (IsFlagDefined (side) +" "+ side);
      }
}
```

ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РБОТУ

Разработать предметную модель согласно варианту задания, полученному у преподавателя.

ТРЕБОВАНИЯ К РЕАЛИЗАЦИИ

При выполнении лабораторной работы необходимо использовать стандарт кодирования языка C#. Для разработанных классов создать набор модульных тестов, проверяющих корректность кода, а также демонстрирующих полученный API.

ВАРИАНТЫ ЗАДАНИЙ

Вариант № 1. «Служба доставки»

Модель модуля определения стоимости заказов для приложения службы доставки представлена на рис. 2.1.

Клиент компании (сущность Customer) имеет следующие атрибуты: персональный код (Code), Ф.И.О. (FullName), контактный телефон (ContactPhone). Некоторые клиенты являются «привилегированными (Privileged)».

Каждый заказ (класс Order) идентифицируется номером (атрибут Number) и описывается следующими атрибутами: датой поступления (CreationDate), адресом доставки (Address) и пометкой, является ли заказ срочным (ExpressDelivery).

Один заказ подразумевает доставку одного или более товаров. Позиция заказа представлена сущностью OrderLine и имеет следующие атрибуты: заказанный товар/услуга (Item) и количество (Quantity).

Товар (сущность Item) описывается артикулом (Article), наименованием (Name) и ценой единицы товара (UnitPrice).

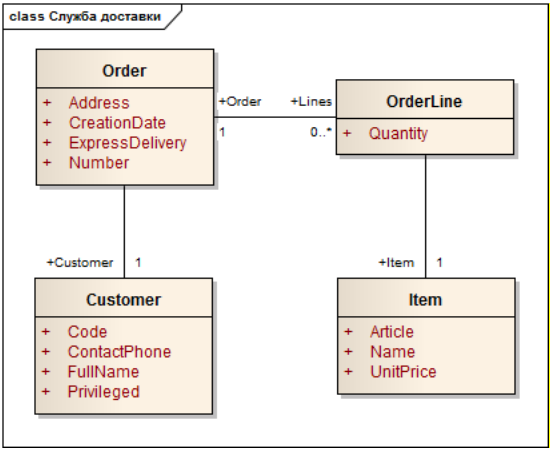

Рис. 2.1. Модель предметной области приложения «Служба доставки»

Стоимость позиции заказа определяется как произведение цены единицы товара/услуги на заказанное количество (Cost = Item.UnitPrice * Quantity).

Общая стоимость заказа (TotalCost) определяется как сумма стоимостей всех его позиций. Срочная доставка увеличивает стоимость заказа на 25%. Если заказчик – «привилегированный» клиент, и общая стоимость превысила 1 500, то предоставляется 15% скидка.

Необходимо реализовать классы Customer, Item, Order, OrderLine. Реализованная модель должна позволять:

- создавать заказ, указав клиента, номер, дату, адрес доставки и «срочность»;
- формировать заказ, т. е. добавлять, редактировать и удалять позиции (при добавлении позиции заказа указывается товар и количество);
- определять общую стоимость заказа по описанным выше правилам.

Вариант № 2 «Управление задачами»

Модель приложения для управления задачами в рамках проектов представлена на рис. 2.2.

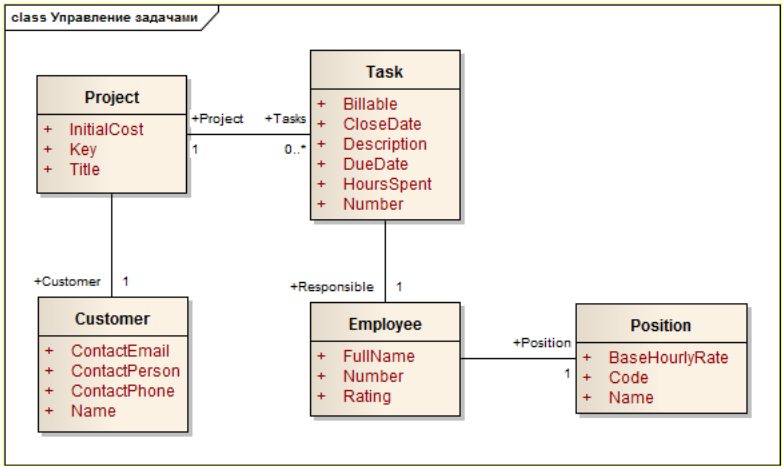

Рис. 2.2. Модель предметной области приложения «Управление задачами»

Проект (класс Project) определяется ключевой строкой (атрибут Key) и имеет следующие атрибуты: бюджет (InitialCost) и название (Title). Клиент, по заказу которого выполняется проект, представлен сущностью Customer со следующими атрибутами: наименование (Name), Ф.И.О. контактного лица (ContactPerson), контактный телефон (ContactPhone) и контактный e-mail (ContactEmail).

В рамках проекта выделяются задачи (класс Task). Задача представляется кратким номером (атрибут Number), подробным описанием (атрибут Description), сроком исполнения (атрибут DueDate), датой завершения работ (атрибут CloseDate), затраченным на выполнение временем (в часах; атрибут HoursSpent), признаком «Отдельно оплачивается заказчиком» (атрибут Billable).

За выполнение каждой задачи отвечает один из сотрудников компании. Сотрудник представлен сущностью Employee со следующими атрибутами: табельный номер (Number), Ф.И.О. (FullName), разряд (Rating, целое число от 1 до 5). Должность

сотрудника моделируется сущностью Position, описываемой атрибутами: шифр (Code), название (Name) и базовая почасовая ставка (BaseHourlyRate). Стоимость часа работ сотрудника (обозначим ее HourlyRate) определяется как сумма базовой почасовой ставки, определяемой должностью, и надбавкой за разряд (5% от базовой ставки для 2-го разряда, 10% – для 3-го и т. д.; по 5% за каждый разряд). Например: Захаров Степан Борисович, занимающий должность «ведущий разработчик» (базовая ставка: 300 руб./час) и имеющий 5-й разряд, должен получать 375 руб./час.

Стоимость задачи, отдельно оплачиваемой заказчиком, определяется следующим образом. Если работы выполнены в срок, то стоимость равна произведению стоимости часа работ ответственного сотрудника на затраченное время (Cost = Responsible.HourlyRate * HoursSpent). Если согласованный срок превышен, то за каждый день просрочки начисляется пеня 1%, но не более 25%.

Стоимость проекта определяется как сумма бюджета и стоимостей отдельно оплачиваемых задач.

Необходимо реализовать классы Project, Customer, Task, Employee. Реализованная модель должна позволять:

- создавать проект, указав клиента, ключевую строку, название и бюджет;
- добавлять, редактировать и удалять задачи (при добавлении задачи указываются ее атрибуты и ответственный сотрудник);
- определять общую стоимость проекта по описанным выше правилам.

Вариант № 3. «Учебная программа»

Фрагмент модели предметной области приложения, управляющего индивидуальными учебными программами студентов, представлен на рис. 2.3.

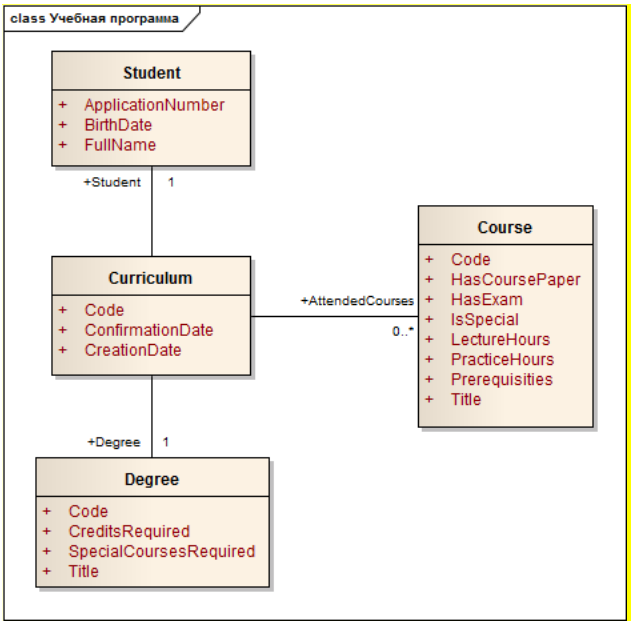

Рис. 2.3. Модель предметной области приложения «Учебная программа»

Университет предлагает студентам для изучения ряд учебных курсов, некоторые из них являются общеобразовательными, а некоторые – специальными. «Трудоемкость» курса определяется т. н. кредитными единицами, количество которых вычисляется следующим образом:

C=[L+1,25P18,00],

где C – количество кредитных единиц курса,

L – количество часов лекций,

P – количество часов практических занятий.

Экзамен увеличивает трудоемкость курса на 1 кредитную единицу, а курсовая работа – на 2 единицы.

Кроме того, некоторые курсы могут быть выбраны только вместе с определенными курсами (например, если студент выбирает курс «Логика и теория алгоритмов», он обязательно должен прослушать и подготовительные курсы – «Дискретная математика» и «Теория вероятностей»).

Для получения квалификационной степени за все время обучения студент должен прослушать определенное количество спецкурсов, а также набрать в сумме некоторое количество кредитных единиц. После зачисления в Университет, студент составляет индивидуальную учебную программу, содержащую список посещаемых курсов. Составленная программа является допустимой, если для каждого выбранного курса были выбраны и все требуемые предварительные курсы, а так же выполняются требования квалификационной степени.

Студент представлен сущностью Student и имеет следующие атрибуты: номер заявления (ApplicationNumber), Ф.И.О. (FullName) и дата рождения (BirthDate).

Индивидуальная учебная программа студента представлена сущностью Curriculum и имеет следующие атрибуты: регистрационный код (Code), дата составления (CreationDate) и дата утверждения (ConfirmationDate).

Выбранный курс (класс Course) описывается следующими атрибутами: регистрационный код (Code), наименование (Title), общеобразовательный или спецкурс (флаг IsSpecial), количество часов лекций (LectureHours) , количество часов практики (PracticeHours), наличие экзамена (флаг HasExam), наличие курсовой работы (HasCoursePaper) и список регистрационных кодов курсов, которые должны быть обязательно выбраны вместе с данным курсом (атрибут Prerequisities).

Квалификационная степень (класс Degree) имеет следующие атрибуты: шифр (Code), наименование (Title), требуемое общее количество кредитных единиц (CreditsRequired) и минимальное количество спецкурсов (SpecialCoursesRequired)

Необходимо реализовать классы Student, Degree, Curriculum, Course. Реализованная модель должна позволять:

- создавать учебную программу, указав студента, получаемую квалификационную степень, регистрационный номер, дату заполнения;
- добавлять и удалять в программу курсы;
- определять допустимость составленной программы по описанным выше правилам.

Вариант № 4. «Начисление зарплаты»

Модель приложения начисления зарплаты сотрудникам представлена на рис. 4.

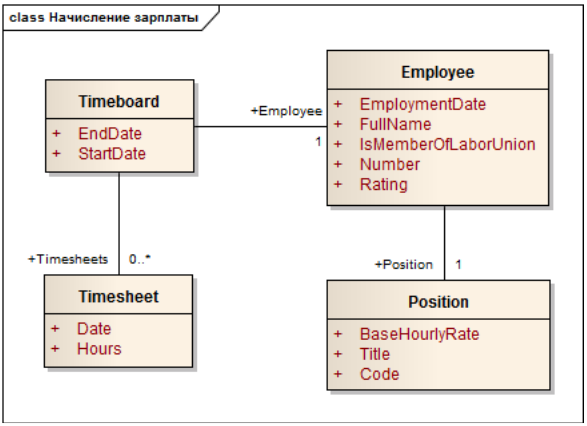

Рис. 2.4. Модель предметной области приложения «Начисление зарплаты»

Время, отработанное сотрудником компании, учитывается в табеле рабочего времени в виде записи: дата – число отработанных часов.

Стоимость часа работ сотрудника (обозначим ее HourlyRate) определяется как сумма базовой почасовой ставки, определяемой должностью, и надбавкой за разряд (10% от базовой ставки для 2-го разряда, 20% – для 3-го и т. д.; по 10% за каждый разряд). Например: Захаров Степан Борисович, занимающий должность «ведущий разработчик» (базовая ставка: 300 руб./час) и имеющий 5-й разряд, должен получать 420 руб./час.

Переработки (т. е. превышение 8-часового рабочего дня), а также работу в выходные дни (субботу и воскресенье) компания оплачивает по двойному тарифу.

Итоговая сумма, выплачиваемая сотруднику на руки, складывается из оплаты отработанного им времени, из которой удерживается 13% подоходного налога, а также 2% профсоюзных отчислений (если сотрудник является членом профсоюза).

Пример. Пусть почасовая ставка Захарова Степана Борисовича составляет 420 руб./час. В табеле рабочего времени имеются следующие записи:

01.02 (пятница)	10 час.
02.02 (суббота)	6 час.
04.02 (понедельник)	8 час.
05.02 (вторник)	8 час.
06.02 (среда)	6 час.

Тогда 2 часа переработок в пятницу 01.02, а также 6 часов работы в субботу 02.02 будут оплачиваться по двойному тарифу 840 руб./час. Остальное рабочее время (по 8 часов 01.02, 04.02, 05.02 и 6 часов 06.02) оплачивается по обычному тарифу 420 руб./час. Оплата рабочего времени составляет:

(2+6)·840+(8+8+8+6)·420=6720+12600=19320 (руб.)

Из этой суммы удерживается подоходный налог 19320·0,13=2511,60 (руб.) и профсоюзные отчисления (т. к. данный сотрудник является членом профсоюза): 19320·0,02=386,40 (руб.). Итого, «на руки» Степан Борисович должен получить 16 422 руб.

Сотрудник представлен сущностью Employee со следующими атрибутами: табельный номер (Number), Ф.И.О. (FullName), разряд (Rating, целое число от 1 до 5), дата приема на работу (EmploymentDate) и членство в профсоюзе (IsMemberOfLaborUnion).

Должность сотрудника моделируется сущностью Position, описываемой атрибутами: шифр (Code), название (Name) и базовая почасовая ставка (BaseHourlyRate).

Табель рабочего времени (сущность Timeboard) имеет атрибуты «Начало заполнения» (StartDate) и «конец заполнения» (EndDate) и содержит записи о рабочем времени – экземпляры сущности Timesheet (атрибуты: Date – дата и Hours – отработанное время).

Необходимо реализовать классы Employee, Position, Timesheet, Timeboard. Реализованная модель должна позволять:

- заполнять табель рабочего времени, указывая даты и отработанные часы;
- определять величину зарплаты за рабочее время, указанное в табеле.

Вариант № 5. «Качество работы»

Модель приложения оценки качества работы предприятия представлена на рис. 2.5.

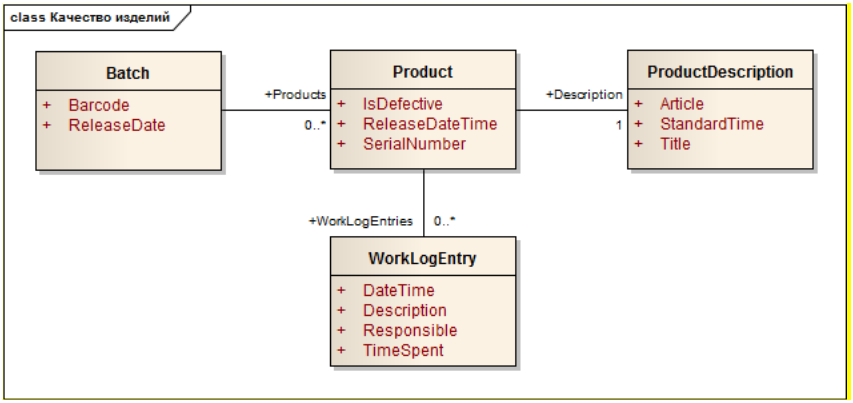

Рис. 2.5. Модель предметной области приложения «Качество работы»

Завод выпускает партии изделий. После выпуска и присвоения серийного номера каждое изделие проходит контроль качества, по результатам которого признается либо бракованным (и в дальнейшем направляется на утилизацию), либо годным. Все работы, выполняемые с изделием, фиксируются в журнале работ.

После выпуска партии присваивается категория, отражающая качество работы бригады:

- 1-я категория: не более 3% брака; 80% изделий изготовлены за установленное нормативом время; для остальных изделий среднее время переработки не превышает 5 минут;
- 2-я категория: не более 5% брака; 75% изделий изготовлены за установленное нормативом время;
- 3-я категория: в остальных случаях.

Партия представляется классом Batch и имеет следующие атрибуты: штрих-код (Barcode) и дата выпуска (ReleaseDate).

Изделие представляется классом Product и имеет следующие атрибуты: серийный номер (Barcode), дата/время выпуска (ReleaseDateTime) и флаг «Брак» (IsDefective).

Сущность ProductDescription моделирует номенклатуру изготавливаемых изделий и содержит следующие атрибуты: артикул (Article), наименование (Title) и норма рабочего времени на изготовление (StandardTime).

Записи в журнале рабочего времени, потраченного на изготовление изделия, представлены сущностью WorkLogEntry. Эта сущность имеет следующие атрибуты: дата и время внесения записи (DateTime), рабочий, выполнявший работу (Responsible), время работ (TimeSpent), описание работ (Description).

Необходимо реализовать классы Batch, Product, ProductDescription, WorkLogEntry. Реализованная модель должна позволять:

- добавлять информацию об изделиях, входящих в партию;
- добавлять записи в журнал рабочего времени изделия;
- определять категорию качества партии.

Вариант № 6. «Лаборатория»

Модель приложения фиксации результатов проводимых лабораторией анализов представлена на рис. 2.6.

Заводская лаборатория проводит исследования производственных образцов (пробы масел, металлов, пластика и проч.). Направляя пробу, цех запрашивает проведение ряда типовых анализов (например, определение кислотности, плотности, воспламеняемости для пробы масла).

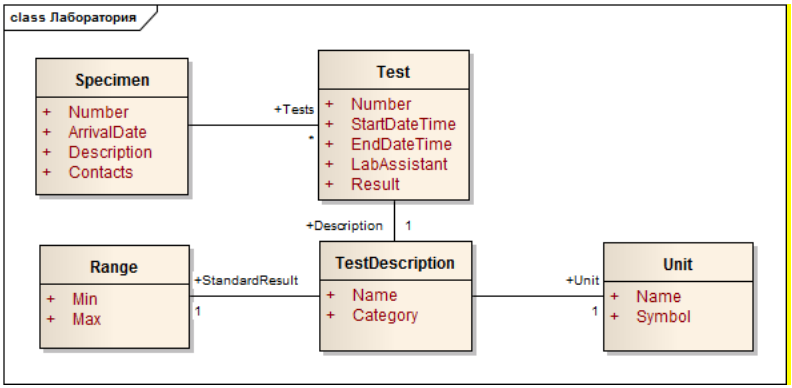

Рис. 2.6. Модель предметной области приложения «Лаборатория»

Каждый анализ проводится по инструкции, которая определяет в том числе и норму результата в виде диапазона значений (пример, значение, полученное в результате анализа «кинематическая вязкость масла при 40°C» должно лежать в диапазоне 28,8–35,2 мм2/с). Анализ считается успешным, если полученное значение лежит в пределах нормы.

Типовые анализы делятся на категории: важные и обычные.

Исследуемый образец считается успешно прошедшим все анализы, если:

- успешно пройдены все важные анализы;
- не пройдено не более 2 обычных анализов.

Образец представляется классом Specimen и имеет следующие атрибуты: входящий номер (Number), дата поступления (ArrivalDate), описание (Description) и контактная информация заказчика (Contacts).

Анализ образца представляется классом Test и имеет следующие атрибуты: номер (Number), дата/время начала анализа (StartDateTime), дата/время окончания (EndDateTime), лаборант, выполнявший анализ (LabAssistant) и значение, полученное в результате анализа (Result).

Сущность TestDescription моделирует типовой анализ (т.е. вид анализа, например, «кинематическая вязкость масла при 40°C») и содержит следующие атрибуты: название (Name), признак важный/обычный (Category), диапазон значений нормы (StandardResult) и единицу измерения результата (Unit).

Диапазон значений описывается сущностью Range с атрибутами: минимальное допустимое значение (Min) и максимальное допустимое значение (Max).

Единица измерения представляется сущностью Unit с атрибутами: наименование (Name; например, «градус Цельсия») и обозначение (Symbol, например, «°C»).

Необходимо реализовать классы Specimen, Test, TestDescription, Range, Unit. Реализованная модель должна позволять:

- добавлять информацию об анализах, проводимых с образцом;
- определять итоговый результат всех анализов образца (успешно пройдены или нет).

Вариант № 7. «Учебные сертификаты»

Модель приложения учета обучения студентов приведена на рис. 2.7.

Профессора Университета осуществляют обучение различным дисциплинам в режиме онлайн для всех желающих. Любой может зарегистрироваться в Университете как студент и записываться в дальнейшем на разные курсы.

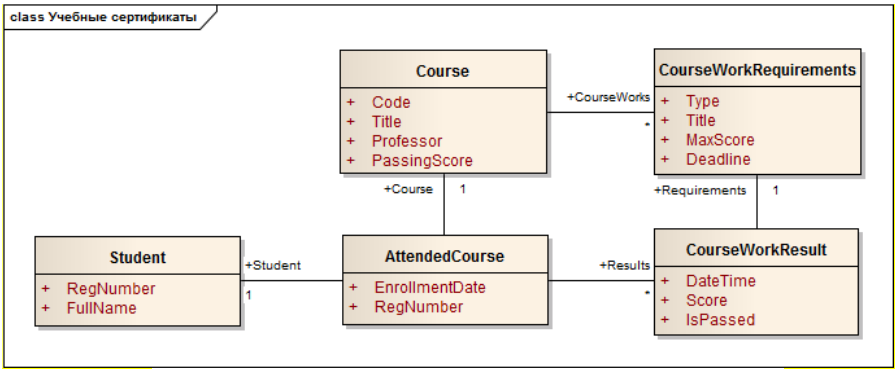

Рис. 2.7. Модель предметной области приложения «Учебные сертификаты»

В ходе обучения студенты выполняют разные виды работ: рубежные тесты, домашние задания, сдают экзамены. За каждую успешно сданную работу студент получает рейтинговые баллы. После завершения курса студент получает сертификат, если сумма рейтинговых баллов по всем выполненным баллам равна или превышает балл, заданный профессором для курса. При подсчете баллов учитываются следующие правила:

- успешно принятые работы не могут пересдаваться (с целью повышения балла);
- если работа сдана не в срок, то при подсчете учитываются только 75% набранных за нее баллов (округление вниз).

Сущность Student представляет студента и содержит атрибуты «ФИО» (FullName) и регистрационный номер в Университете (RegNumber).

Предоставляемый для изучения курс моделируется сущностью Course, имеющей следующие атрибуты: шифр (Code), наименование (Title), ФИО профессора (Professor), балл, который необходимо набрать для получения сертификата о прохождении курса (PassingScore).

Требования к работе, которую необходимо сдать в ходе обучения, описываются сущностью CourseWorkRequirements с атрибутами: тип работы (Type, рубежный тест/домашнее задание/экзамен), название (Title), максимальный балл, который можно получить за работу (MaxScore), срок сдачи (Deadline).

Факт выбора студентом для изучения курса моделируется сущностью AttendedCourse, хранящей дату зачисления студента на курс (EnrollmentDate) и регистрационный номер студента на курсе (RegNumber).

Результат сдачи студентом работы представляется сущностью CourseWorkResult. Она содержит следующие атрибуты: дата/время сдачи (DateTime), признак, успешно ли сдана работа (IsPassed) и набранный рейтинговый балл.

Необходимо реализовать классы Student, Course, CourseWorkRequirements, AttendedCourse, CourseWorkResult. Реализованная модель должна позволять:

- добавлять информацию о требованиях к работам по курсу;
- добавлять информацию о результатах сдачи работ студентами;
- определять результат изучения студентом курса (получает сертификат или нет).

КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАНИЯ

1. Опишите принцип «Command-Query Separation». Каково его назначение?
2. Раскройте значение обязанности класса. Как следует распределять обязанности?
3. Опишите назначение класса List<T>
4. Опишите интерфейс класса List<T>.
5. Как представляются строки в .NET?
6. Изложите концепцию форматирования строки
7. В чем особенности сравнения и упорядочения строк?
8. Опишите назначение и интерфейс класса StringBuilder.
9. Для чего предназначены классы DateTime, TimeSpan?

ФОРМА ОТЧЕТА ПО ЛАБОРАТОРНОЙ РАБОТЕ

На выполнение лабораторной работы отводится 8 часов: 7 часов на выполнение работы, 1 час на подготовку и защиту отчета по лабораторной работе.

Номер варианта студенту выдается преподавателем.

Отчет на защиту предоставляется в печатном виде.

Структура отчета (на отдельном листе(-ах)): титульный лист, формулировка задания (вариант), UML-диаграмма классов модели, этапы выполнения работы (со скриншотами), результаты выполнения работы. выводы.