Методические указания к лабораторным работам по дисциплине «Основы объектно-ориентированного программирования»

Автор: Калентьев Алексей Анатольевич

1 Лабораторная работа №1. – Классы

1. Реализуйте класс *Person*, содержащий: имя, фамилию, возраст, пол (используйте перечисления – *enum*), конструктор класса.
2. Реализуйте класс *PersonList*, описывающий абстракцию списка, содержащего объекты класса Person. Класс должен позволить: добавлять элементы, удалять элементы, удалять элементы из списка по индексу, искать элемент по указанному индексу, вернуть индекс элемента при наличии его в списке, очистить список, получить количество элементов в списке.
3. Продемонстрируйте правильность работы ваших классов. Для этого:
    a. Создайте программно два списка персон, в каждом из которых будет по три человека
    b. Выведите содержимое каждого списка на экран с соответствующими подписями списков.
    c. Добавьте нового человека в первый список.
    d. Скопируйте второго человека из первого списка в конец второго списка. Покажите, что один и тот же человек находится в обоих списках.
    e. Удалите второго человека из первого списка. Покажите, что удаление человека из первого списка не привело к уничтожению этого же человека во втором списке.
    f. Очистите второй список.

    Каждый новый шаг должен выполняться по нажатию любой клавиши клавиатуры.
4. Реализуйте в классе *Person* методы чтения персоны с клавиатуры и вывод персоны на экран.
5. Создайте в классе *Person* статический метод *GetRandomPerson()*. Реализация данного метода заметно упростит тестирование и демонстрацию программы.
6. *Добавьте в функцию чтения проверку правильности данных:
    a. Возраст не может быть отрицательным
    b. При вводе возраста не должно быть возможности ввода символов
    c. Имя и фамилия должны содержать только русские или английские символы
    d. При вводе имени и фамилии с клавиатуры, они автоматически должны преобразовываться в правильные регистры: первая буква заглавная, остальные прописные.
    e. Учтите возможность ввода двойного имени и двойной фамилии.

7. Весь код должен быть оформлен в соответствии со стандартом оформления кода RSDN.
8. А теперь посмотрите на стандартный .NET тип List<T> =)

(*) – задание повышенной сложности

1.1 Вопросы для проверки:

1. Что такое класс? Что такое объект?
2. Что такое поле объекта? Что такое метод объекта?
3. Что такое интерфейс класса? Что такое реализация класса?
4. Что такое конструктор класс? Чем конструктор отличается от обычного метода? Сколько конструкторов может быть у одного класса? Что такое конструктор по умолчанию?
5. Что такое деструктор? Для чего он нужен? Сколько деструкторов может быть в одном классе?
6. Что такое инкапсуляция? Какие преимущества с точки зрения разработки ПО позволяет получить инкапсуляция?
7. Что такое модификатор доступа? Какие есть модификаторы доступа в языке C#? Для чего нужен каждый из них?
8. Что такое статическое поле класса? Что такое статический метод класса? Для чего они нужны?
9. Оформление исходного кода класса согласно RSDN: как именуется класс? Как именуются закрытые поля? Как именуются открытые поля? Как именуются методы класса? Какие поля и методы класса должны быть прокомментированы? В каком порядке должны следовать поля и методы внутри класса? Какие ограничения на размер класса существуют в стандарте RSDN?

1.2 **Рекомендуемая литература:**

- Шилдт Г. C# 4.0 //Полное руководство. Вильямс. – 2011.
- Троелсен, Эндрю. Язык программирования C# 5.0 и платформа .NET 4.5 [Текст] / Pro C# 5.0 and the .NET 4.5 framework / Э. Троелсен ; [пер. с англ. Ю. Н. Артеменко]. - 6-е изд. - М. : ООО "И.Д. Вильямс", 2013. - 1311 с. : рис. - Парал. тит. л.: англ.

2 Лабораторная работа №2. – Наследование и полиморфизм

1. На основе предыдущей лабораторной создайте путём наследования от класса Person два дочерних класса *Adult* и *Child*. В класс *Adult* добавьте информацию о паспортных данных, состоянии брака (со ссылкой на партнёра), название места работы. В класс *Child* добавьте информацию о родителях, а также название детского сада/школы.
2. Добавьте в базовый класс *Person* абстрактный метод получения информации и сделайте уникальную реализацию в базовом и дочерних классах. Информация должна содержать перечисление всех данных о конкретной персоне в виде строки. Обратите внимание, что сам метод ничего не должен выводить на экран, а только формировать строку и возвращать её из метода.
   Для экземпляра *Adult* метод описания зависит от данных. Если человек женат, то в описании должно присутствовать информация о том, на ком женат. Если человек не женат (null), то отразить эту информацию в описании. По аналогии должна добавляться информация о месте работы человека и «Безработный», если место работы не было указано. Для ребёнка стоит учесть различные формулировки при наличии/отсутствии каждого из родителей.
3. *Реализуйте в соответствующих классах статические методы для генерации случайного взрослого или ребёнка. Метод должен возвращать новый созданный объект персоны.
4. Инкапсулируйте поле *Age* для класса *Person* и создайте уникальные механизмы задания возраста для каждого класса. На данный момент, не смотря на различия в концепциях *Adult* и *Child* ничто не мешает присвоить взрослому возраст 2, а ребёнку 59. Для этого сделайте свойство с обработкой корректного возраста для этих классов.
5. Продемонстрируйте работу новых классов, выполнив следующую последовательность действий:
   a. В функции *main*() программно создайте список *PersonList,* в который добавьте семь человек – разное количество взрослых и детей в случайном порядке. (Обратите внимание, что ранее реализованный список, хранящий в себе исключительно класс *Person*, работает также и для дочерних классов)
   b. Выведите на экран описание всех людей списка. Продемонстрируйте, что для различных типов людей описания содержат разную информацию

c. Программно определите тип четвёртого человека в вашем списке. Для демонстрации корректности определения типа выполните какой-нибудь из методов, присущий этому классу.

(*) – задание повышенной сложности

2.1 Вопросы для проверки:

1. Что такое наследование? Как наследование реализуется в C#?
2. В чём отличие наследования от агрегации?
3. Что такое указатель на базовый класс?
4. Что такое виртуальные функции? Что такое чисто виртуальные функции?
5. Что такое полиморфизм? Как полиморфизм реализуется в C#?
6. Какие проблемы могут возникнуть при работе с полиморфными объектами через указатель на базовый класс?
7. Что такое абстрактный класс?
8. Что такое статическое поле, статический метод, статический класс? Для чего они нужны?
9. Что такое свойства? Для чего они нужны?

2.2 **Рекомендуемая литература:**

- Шилдт Г. C# 4.0 //Полное руководство. Вильямс. – 2011.
- Троелсен, Эндрю. Язык программирования C# 5.0 и платформа .NET 4.5 [Текст] / Pro C# 5.0 and the .NET 4.5 framework / Э. Троелсен ; [пер. с англ. Ю. Н. Артеменко]. - 6-е изд. - М. : ООО "И.Д. Вильямс", 2013. - 1311 с. : рис. - Парал. тит. л.: англ

3 Лабораторная работа №3. Бизнес-логика

Целью данной работы является реализация простого проекта в IDE Visual Studio 2013 на языке C#.

3.1 Integrated development environment

Когда программирование только зарождалось и им еще занималась не программисты, а научные работники, не было никаких средств для эффективного написания кода, программы писались в примитивных текстовых редакторах и компилировались из командной строки. Такой подход не вызывает сложностей при написании небольших программ. При разработке крупных программных проектов он становится затруднительным. Поэтому со временем стали появляться специализированные среды разработки, позволяющие по возможности автоматизировать часть задач и упростить написание кода, такие среды называются IDE.

IDE (*Integrated development environment*, интегрированная среда разработки) - система программных средств, используемая программистами для разработки программного обеспечения. В базовом виде IDE включают в себя текстовый редактор, компилятор, средства автоматизации сборки, отладчик. Однако современные IDE включают в себя большое количество дополнительных инструментов, позволяющих значительно облегчить процесс написание кода, например:

1) Подсветка синтаксиса;
2) Статические анализаторы, которые проверяют ошибки в программе прямо во время написания (а не на этапе компиляции);
3) Инструменты для эффективного рефакторинга;
4) Фреймворки для генерации и написания модульных тестов и др.

Существует множество различных IDE, которые поддерживают как несколько языков (Microsoft Visual Studio, IntelliJ IDEA, NetBeans), так и всего один (PyCharm, Delphi). Одной из лучших IDE для разработки приложений для Windows является Microsoft Visual Studio. Кроме того, что Microsoft Visual Studio позволяет эффективно разрабатывать на популярных языках (C++, C#, VB.NET), значительным преимуществом является возможность ее расширения различными плагинами, начиная от продвинутой подсветки синтаксиса и интеграции систем контроля версий, до подключения дополнительных языком программирования. На сайте Microsoft можно

скачать бесплатную версию Visual Studio Express 2013, которая понадобится для выполнения лабораторных работ.

3.2 Знакомство с Microsoft Visual Studio

После установки и запуска Microsoft Visual Studio Express 2013 (далее MSVS) запустится стартовое окно программы. Для того чтобы создать проект необходимо нажать *Create Project* на стартовой странице, либо выбрать *FILE->Create->Project* после чего появится окно создания проекта.

В лабораторных работах будут использоваться 3 вида проектов:

1) *Class library* – библиотека классов – проект такого типа компилируется в файл формата *.dll. Используются для описания бизнес-логики приложения.
2) *Console application* – консольное приложение – компилируется в *.exe. Позволяет создать приложение с которым можно взаимодействовать через консоль.
3) *Windows forms application* – оконное приложение – компилируется в *.exe. Позволяет создавать оконные приложения для Windows, используя технологию WinForms.

После создания проекта MSVS создает ***Solution*** (Решение), в котором находится созданный проект. Решение содержит элементы, необходимые для создания приложения. Решение может включать один или несколько проектов, а также файлы и метаданные, необходимые для определения решения в целом. Решение необходимо для того чтобы хранить все проекты, которые относятся к одному приложению, а также для отслеживания их взаимодействий. MSVS хранит определение решения в двух файлах: *.sln и *.suo. Файл решения (*.sln) содержит метаданные, которые определяют решение, в том числе:

1) Проекты, связанные с решением.
2) Элементы, которые не связаны с определенным проектом (текстовые файлы, картинки и т.д.).
3) Конфигурации сборки, определяющие, какие конфигурации проекта, применяемых в каждом типе сборки.

Для добавления файлов в проект необходимо вызвать контекстное меню решения (нажать правой кнопкой на корневом узле в Обозревателе решения (Solution Explorer)) после чего выбрать пункт Добавить (Add). Таким образом

в решение можно добавить новые проект или различные файлы (Create element…).

Project (Проект) MSVS служит контейнером для файлов с исходным кодом, подключенным библиотекам и файлам. Управление проектом также осуществляется через контекстное меню, которое можно вызвать через Обозреватель решения. Через контекстное меню можно добавлять файлы с исходным кодом в проект (Add->Create element…).

Важно!!! После установки MSVS, файлы с расширением *.cs ассоциируются с ней. То есть такие файлы будут открываться в MSVS. Следует понимать, что открытые таким образом файлы не добавляются в проект или решение и они не могут быть скомпилированы. Чтобы добавить существующий файл в проект необходимо вызвать контекстное меню проекта и в пункте Добавить выбрать существующий проект (Add->Exiting element…).

Часто один проект должен использовать некоторые типы данных, определенные в другом проекте. Для этого необходимо в основной проект добавить ссылку на зависимый проект. Для этого необходимо вызвать контекстное меню элемента References основного проекта и выбрать пункт Добавить ссылку (Add Reference…). После этого появится окно изображенное на рис.3.1. Зависимость можно добавить, как на проекты, находящиеся в том же решении (для этого надо выбрать пункт Решение), так и на существующие сборки входящие в .NET Framework или дополнительно установленные библиотеки (пункт Сборки). После выбора необходимой библиотеки необходимо установить флаг слева от названия в положение используется и нажать ОК. После этого можно использовать типы данных определенные в выбранной библиотеке в вашем проекте.

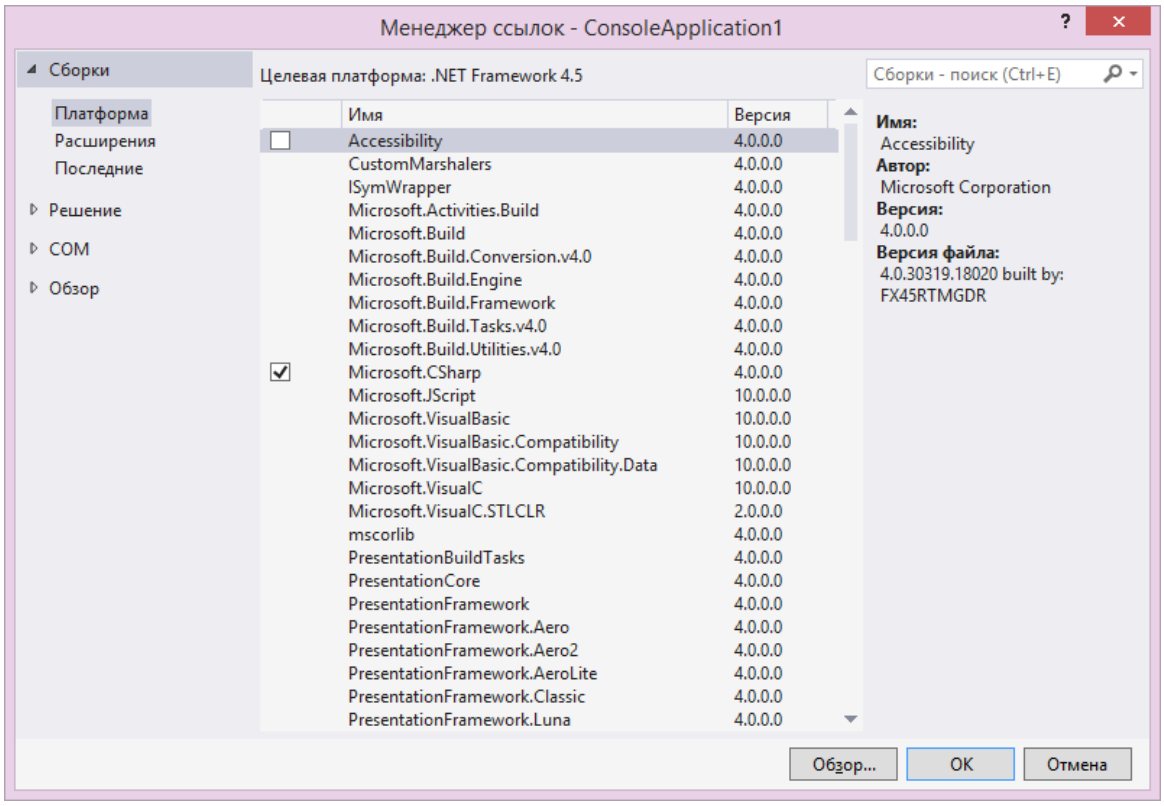

Рисунок 3.1 – Окно добавления зависимостей проекта

3.3 Первое приложение на C#

Во всех учебниках по программированию в качестве первой программы всегда используется «Hello, World». Единственной функцией этой программы выводить в консоль строку «Hello, World!».

В первую очередь необходимо создать новое консольное приложение. Назовем его HelloWorld. После создания MSVS сгенерирует Решение HelloWorld и проект HelloWorld. Будет сгенерирован единственный файл с кодом Program.cs. Ниже следует его содержимое.

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

Рассмотрим листинг подробнее. На первой строке находится объявление пространства имен, название которого по умолчанию совпадает с именем проекта.

Затем идет объявление класса Program. Ввиду того, что C# – полностью объектно-ориентированный язык, то в нем не может быть функций, которые не принадлежат какому-либо классу, в данном случае класс Program необходим для определения метода Main. Класс Program, является обычным классам и с ним можно делать все то же, что и с другими, созданными пользователем, классами. Например, его можно переименовать.

Далее идет объявление функции Main. Любое консольное приложение должно содержать метод Main. Метод Main – точка входа в программу, начало ее выполнения. В случае отсутствия этого метода будет ошибка при компиляции, потому что компилятор не будет знать где начинается ваша программа. Соответственно этот метод нельзя переименовывать и перегружать.

Вообще существует несколько вариантов сигнатуры этого метода:

```
static void Main(string[] args)
static int Main(string[] args)
static void Main()
static int Main()
```

Эти функции будут отличаться типом возвращаемого значения и наличием входных параметров. Рассмотрим подробнее: в первом случае из программы не возвращается никаких значений (void) и программа принимает на вход параметры. Строка *string[] args* содержит в себе список параметров, которые пользователь может передать в программу при вызове из командной строки. Во втором случае программа возвращает в консоль целое число, обычно это делается, чтобы после окончания выполнения программы пользователь мог узнать корректно она завершилась или нет, если корректно, то возвращается 0, в противном случае 1, либо код произошедшей ошибки. В третьем и четвёртом случаях отсутствуют входные аргументы программы. Читатель знающий C++ может провести прямую аналогию с функцией main.

Теперь необходимо добавить в функцию Main, код, который будет выводить необходимую строку. Сделать это можно следующим образом:

```
System.Console.Write("Hello, World!");
```

или

```
System.Console.WriteLine("Hello, World!");
```

Отличия этих двух способов в том, что в первом случае, каретка останется на той же строке, а во втором передвинется на следующую строчку. То есть если у нас будет несколько вызовов System.Console.Write подряд, то

аргументы функции будут выводиться на одной строке, а если будет несколько вызовов System.Console.WriteLine каждый аргумент будет выводиться на новой строчке.

Для того чтобы запустить приложение, необходимо либо на верхней панели нажать кнопку Запуск (Start), либо через главное меню Отладка (Debug)-> Начать отладку (Start debugging), также можно нажать клавишу F5.

При первом запуске скорее всего вы увидите, как мелькнет окно консоли и тут же пропадет. Это не ошибка, просто программа очень быстро выполнилась и закрылась. Чтобы посмотреть результаты выполнения необходимо остановить выполнение программы. Для этого, например, можно заставить программу ждать, пока не будет нажата какая-нибудь клавиша клавиатуры. Для этого необходимо в конец программы добавить строку:

```
System.Console.Read();
```

Если запустить программу теперь, то на экране появится окно консоли с текстом “Hello, World!”. Чтобы окно пропало надо нажать произвольную клавишу.

В итоге ваша программа должна выглядеть следующим образом:

```
namespace HelloWorld
{
class Program
{
      static void Main()
      {
            System.Console.WriteLine("Hello, World!");
            System.Console.Read();
      }
}
}
```

3.4 Ввод/вывод на языке C#

В общем случае, когда говорят о программах, будь то консольное приложение или оконное, подразумевается, что пользователю необходимо с ней взаимодействовать. Программа должна сообщить пользователю о процессе или результатах выполнения, а также работать с данными, который пользователь ей подает. Поэтому в данной части речь пойдет о операторах ввода и вывода на C#.

Для получения данных с клавиатуры можно использовать следующий метод:

```
string str = Console.ReadLine();
```

Как видно, этот метод возвращает строку, и не существует его перегрузок или других методов чтобы считывать целочисленные, символьные и другие типы переменных. Соответственно, для того чтобы ввести с клавиатуры необходимо получить строковое значение, а затем сконвертировать его в необходимый нам тип данных. Для этого можно использовать класс Convert. В нем определено множество статических методов конвертирования строк в иные типы данных. Далее следуют примеры использования класса Convert.

```
int intValue = Convert.ToInt32(Console.ReadLine());
double doubleValue = Convert.ToDouble(Console.ReadLine());
bool boolValue = Convert.ToBoolean(Console.ReadLine());
long longValue = Convert.ToInt64(Console.ReadLine());
```

При этом надо следить чтобы типы вводимых значений совпадали с типами указанных переменных, потому что в противном случае возникнет исключение. Например, если ввести строку 123.456, и попытаться сконвертировать в целочисленное значение, то сгенерируется исключение.

Для более корректной работы программы, если программа ожидает от пользователя какое-то определенное значение, то лучше ему об этом сказать. Необходимо придерживаться правила: Перед тем как считывать что-либо с консоли, необходимо сообщить пользователю, что именно он должен ввести: смысл вводимой информации, тип данных, максимальное и минимальное допустимые значения и т.п. Примером таких запросов могут служить:

«Введите имя пользователя (не больше 20 знаков)»

«Введите возраст, целочисленное значение, от 1 до 100»

«Введите пол, 0 – мужской, 1 - женский»

Для вывода текста на экран можно использовать команды, про которые говорилось в предыдущей главе: Console.Write или Console.WriteLine. Далее идут примеры использования этих команд.

```
Console.WriteLine(s);// переменная
Console.WriteLine(55.3);// константа
Console.WriteLine(y*3+7);// выражение

Console.Write(z);// переменная
Console.Write(-5.3);// константа
Console.Write(i*3+7/j);// выражение
```

Очень часто в процессе работы необходимо выводить осмысленные предложения с результатами выполнения программы. Например «Через насос X, было перекачано Y литров воды, температура насоса Z градусов», где в различные моменты выполнения программы X, Y, Z могут быть различными значениями. Конечно, можно использовать следующий подход:

Console.WriteLine(«Через насос » + X + «, было перекачано » + Y+ « литров воды, температура насоса » + Z + « градусов»);

Однако такой подход слишком громоздкий, и с ним возникает масса трудностей, например, при необходимости добавить дополнительные данные, даже читать такую запись достаточно затруднительно. Поэтому принято использовать подход с использованием форматной строки. Сама строка формата содержит большую часть отображаемого текста, но всякий раз, когда в нее должно быть вставлено значение переменной, в фигурных скобках указывается индекс. В фигурные скобки может быть включена и другая информация, относящаяся к формату данного элемента, например, та, что описана ниже:

- Количество символов, которое займет представление элемента, снабженное префиксом-запятой. Отрицательное число указывает, что элемент должен быть выровнен по левой границе, а положительное — по правой. Если элемент на самом деле занимает больше символов, чем ему отведено форматом, он отображается полностью.
- Спецификатор формата предваряется двоеточием. Это указывает, каким образом необходимо отформатировать элемент. Например, можно указать, должно ли число быть форматировано как денежное значение, либо его следует отобразить в научной нотации, в степенном виде, либо шестнадцатиричном.

Console.WriteLine(«Через насос {0}, было перекачано {1} литров воды, температура насоса {2} градусов», X, Y, Z);

3.5 Задание на лабораторную работу

1) Создайте проект на языке C# в среде Microsoft Visual Studio. Назовите его в соответствии с вашим вариантом задания, в качестве исходного проекта выберите проект динамической библиотеки (*.dll). Назовите его либо согласно вашему варианту, либо просто Model. Данный проект будет содержать в себе бизнес-логику приложения, т.е. ключевые структуры данных и способы их взаимодействия.

2) Создайте сущность-интерфейс согласно вашему варианту. Опишите ключевые свойства и методы интерфейса. Не забудьте о правильном именовании типов данных, согласно RSDN. Подумайте, какие свойства и методы будут являться общими (будут в интерфейсе), а какие должны быть реализованы в конкретных классах.
3) Создайте 2 или более класса, реализующих данный интерфейс. Классы обязательно должны иметь различные реализации методов интерфейса. При этом дочерние классы не должны иметь никаких ссылок друг на друга, также как и интерфейс не должен ничего знать о дочерних классах.
4) Реализуйте проверку правильности передаваемых свойствам данных (валидацию свойств) с помощью механизма обработки исключений – если на вход приходят некорректные данные, выходящие за допустимые пределы, свойство должно сгенерировать исключение соответствующего типа с описанием ошибки. Например, если свойству Возраст, пытаются присвоить отрицательное значение, необходимо сгенерировать экземпляр исключения IncorrectArgumentException. Внимательно продумайте все возможные некорректные варианты входных данных, в том числе ссылки на null. В случае если механизмы валидации у всех свойств одинаковы, измените архитектуру: вместо реализации интерфейса используйте наследование от абстрактного класса, в котором будут реализованы механизмы валидации.
5) Добавьте в решение еще один проект, на этот раз консольное приложение и назовите его «ConsoleLoader». В этом проекте будет проводиться первичное тестирование бизнес-логики приложения.

   Примечание: данный проект является временным, и, впоследствии, будет заменён на проект графического интерфейса Windows (WinForms Application). Однако, если вы уже можете продемонстрировать работу бизнес-логики на оконном пользовательском интерфейсе, можете сразу создать необходимый проект.
6) Продемонстрируйте корректную работу бизнес логики. Создайте переменную-ссылку на интерфейс, и присваивайте в нее экземпляры реализуемых классов. Продемонстрируйте разную реализацию интерфейсных свойств и методов. Для этого необходимо реализовать ввод с клавиатуры значений, которыми будут инициализированы поля классов-наследников.

3.6 Варианты заданий

1) Геометрические фигуры с различными реализациями расчета площади фигуры: круг, прямоугольник, треугольник.
2) Трехмерные фигуры с различными реализациями расчета объема: шар, пирамида, параллелепипед.
3) Работники фирмы с различными способами начисления зарплаты: почасовая оплата, оплата по окладу и ставке.
4) Транспортные средства с различными реализациями расчета затраченного топлива: машина, машина-гибрид, вертолет.
5) Система скидок с различными реализациями расчета скидок: процентная, по сертификату.
6) Система библиотечных карточек для разных изданий: книга, журнал, сборник, диссертация. Каждое издание характеризуется различным набором полей, перегружаемый метод возвращает информацию об издании в виде строки, оформленной по ГОСТу [3].
7) Различные пассивные элементы электрических схем: резистор, конденсатор, индуктивность. Перегружаемый метод – расчет комплексного сопротивления элемента.
8) Расчет координаты для различных видов движения: равномерное, равноускоренное, колебательное.
9) Расчет затраченных калорий в зависимости от вида упражнений: бег (интенсивность, расстояние), плавание (стиль, расстояние), жим штанги (вес, количество повторений).
10) Собственный вариант.

3.7 Рекомендуемая литература

1) Г. Шилдт// C# 4.0 Полное руководство. Изд. «Вильямс». 2011
2) Microsoft Developer Network (MSDN) [Электронный ресурс]. – URL: msdn.microsoft.com/ru-RU/ (дата обращения: 21.12.2014)
3) Библиографическое описание. Государственный УНПК [Электронный ресурс]. – URL: http://www.ostu.ru/libraries/bibl_opisanie.php (дата обращения 18.01.2015)

4 Лабораторная работа №4. Пользовательский интерфейс

Целью данной работы является знакомство с разработкой оконных приложений в среде Microsoft Visual Studio.

4.1 Создание оконного приложения

Для создания оконного приложения на основе уже существующей логики (решения), необходимо добавить в него новый проект WinForms (WinForms Application Project). Это можно сделать через контекстное меню решения в Обозревателе решений, либо через главное меню (File->Add->Create Project…). После нажатия на кнопку OK на форме добавления проекта, новый проект добавится в ваше решение.

Рассмотрим подробнее содержимое нового проекта. По прежнему в нем есть узел References и Properties. К этому добавился файл App.config, Program.cs, Form1.cs.

- App.config – данный файл позволяет создавать конфигурации приложения, что позволить менять некоторые параметры приложения без его перекомпиляции, в лабораторных работах тема конфигураций рассматриваться не будет.
- Program.cs – файл содержащий метод Main. Именно отсюда запускается главная форма приложения с помощью строки

```
Application.Run(new Form1());
```

- Form1.cs – файл в котором хранится код формы. Вообще для описания логики формы используется 2 файла: Form1.cs, где пользователь описывает логику взаимодействия элементов на форме и Form1.Designer.cs который генерирует MSVS, когда пользователь изменяет форму через дизайнер.

Для того, чтобы открыть дизайнер формы, необходимо дважды кликнуть на узел формы в Обозревателе решений, чтобы открыть код с пользовательской логикой к форме необходимо либо выбрать форму в Обозревателе решений и нажать F7, либо в контекстном меню выбрать пункт *Перейти к коду*.

4.2 Дизайнер форм

При запуске дизайнера форм вы увидите окно изображенное на рис. 4.1.

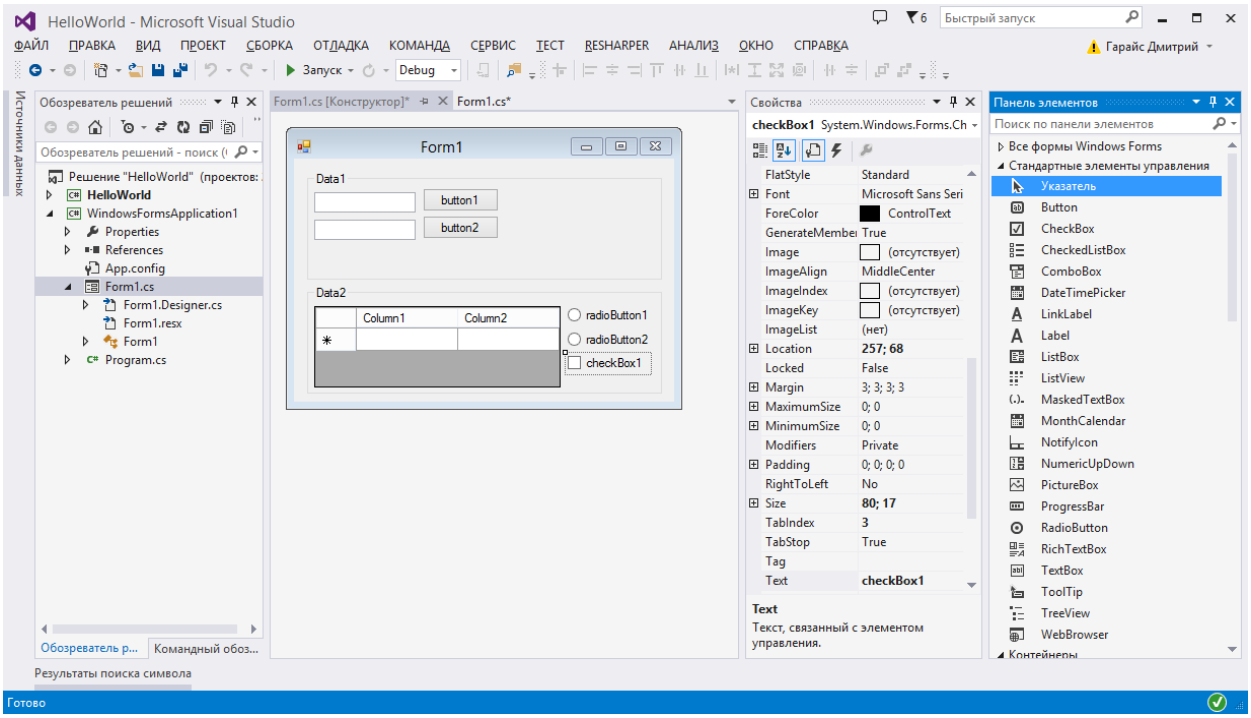

Рисунок 4.1 – Окно дизайнера форм

Для эффективной работы с дизайнером окон, помимо самого дизайнера, необходимы окна *Панель элементов* (Toolbox) и *Свойства* (Properties). В случае. Если этих окон не появилось, можно настроить их отображение, используя Главное меню->Вид->Панель элементов и Главное меню->Вид->Окно свойств.

На *Панели инструментов* находятся все доступные для использования элементы управления. Для добавления того или иного элемента на форму его необходимо перетащить с панели инструментов.

В окне *Свойства* отображаются свойства выбранного элемента управления (чтобы выбрать элемент, необходимо кликнуть на него в дизайнере). Через это окно можно изменять основные свойства элементов, например, имя, по которому можно обратиться к этому элементу в коде, название элемента читаемое для пользователя (например, Главная форма, вместо Form1), размеры элемента, также можно задать его начальное значение. При размещении новых элементов пользовательского интерфейса на форме, Visual Studio генерирует их имена автоматически, например gridControl1, button3 и т.д. Однако такие имена не отражают назначения элемента и усложняют понимание кода. Необходимо переименовывать элементы управления согласно нотации RSDN, это будет оцениваться в лабораторных работах. В верхней части окна *Свойства,* можно перейти в режим *События*,

здесь отображаются все доступные события элемента и их обработчики, также здесь можно назначить новый обработчик для любого события.

4.3 Валидация данных

При разработке интерфейса программы важнейшей задачей для программиста является обеспечение правильности вводимых пользователем данных. В идеале, программист должен так спроектировать и запрограммировать интерфейс, чтобы пользователь НЕ МОГ совершить ошибку, даже если бы захотел.

Для этого, обязательно, после ввода данных проводится проверка их корректности. Реализовать проверку можно несколькими способами:

- при нажатии на определенную кнопку (например, Добавить объект), происходит проверка всех данных на форме, и, в случае обнаружения некорректных, пользователю выводится сообщение с ошибкой.
- при изменении конкретного поля. В данном случае проверки реализуются за счет событий. У любого элемента управления, который позволяет вводить произвольные данные существует событие позволяющее проводить проверку корректности, например это событие Validating у элемента TextBox или событие CellValidating у элемента DataGridView. Данные события срабатывают когда текущий элемент управления теряет фокус, то есть, когда пользователь переключается на любой другой элемент, например другое поле ввода или кнопку. Преимущество использования этого подхода в том, что в случае ввода некорректных данных, пользователь СРАЗУ видит, в каком поле ошибка и может оперативно её поправить, в то время как при использовании первого подхода, ошибка проявляется после того как все данные введены, при этом пользователю придется разбираться в чем ошибка и искать необходимое поле.
- третий подход заключается в том, что пользователь не может ввести то, чего быть не должно. Для этого используются более сложные пользовательские элементы, например поля ввода с проверкой на основе регулярных выражений [1,2,3]. Такой подход самый лучший, потому что, пользователь просто не может написать не правильно. Например, ясно, что в имени не может быть цифр, тогда полю, которое отвечает за ввод имени присваивается соответствующее регулярное выражение и при нажатии цифр, элемент управления сверяется со своим регулярным

выражением и просто игнорирует некорректные данные введенные пользователем. Простым случаем такого элемента управления может служить MaskedTextBox. MaskedTextBox – аналог обычного TextBox, однако позволяет использовать специальные маски ввода, запрещающие вводить в поле цифры или, наоборот, символы. Именно его рекомендуется использовать при реализации заданий лабораторной работы.

При этом стоит отметить, что нельзя полагаться только на один подход, в любом случае необходимо комбинировать эти подходы для обеспечения большей безопасности ввода данных.

Также следует отметить, что все предупреждения об ошибках должны нести явный смысл, они должны сообщать где произошла ошибка и в чем конкретно ошибка. То есть, сообщение типа «Age error» абсолютно некорректно, потому что не понятно в чем именно ошибка возраста. Лучше использовать сообщения типа «Age value must be greater than zero.», в этом сообщении ясно видно в чем именно заключается ошибка.

В случае появления ошибки необходимо сообщить о ней пользователю, для этого можно использовать класс MessageBox. Пример использования приведен ниже.

```
MessageBox.Show(errorText, errorWindowCaption,
MessageBoxButtons.OK,
MessageBoxIcon.Error);
```

В данном случае errorText – текст ошибка, errorWindowCaption – название окна ошибки. MessageBox очень удобный класс для обеспечения взаимодействия с пользователем, потому что легко настраивается за счет параметров, например, изменением третьего параметра можно добавить окошку кнопок, а четвертым – изменить картинку на форме, таким образом, что окошко будет не сигнализировать об ошибке, а предлагать дополнительную информацию, или спрашивать пользователя о некоторых неочевидных действиях (например, следует ли сохранить несохраненный проект при закрытии программы).

4.4 Условная компиляция

Иногда у разработчика возникает необходимость откомпилировать код так, чтобы для разных конфигураций одни участки кода компилировались, а другие – нет. Для этого используются директивы препроцессора и механизм под названием условная компиляция.

В C# существуют следующие директивы: **#define**, **#if**, **#else**, **#endif**. Их смысл в общем-то ясен. Рассмотрим пример условной компиляции.

```
    #define PARAM1
    #if                                                    PARAM1
Console.Write("Defined                                    PARAM1");
#else
Console.Write("Not                  defined               PARAM1");
#endif
```

В таком виде скомилированный код при вызове программы выведет на экран строку: «Defined PARAM1», если же закомментировать первую строку в листинге и заново скомпилировать приложение и запустить, то выведется строка «("Not defined PARAM1"». То есть на основе одного и того же кода, путем задания констант с помощью директивы #define, можно получать программы с различной функциональностью. Чаще всего это используется, когда, необходимо использовать в универсальном приложении платформозависимые вещи, например запись на диск в мобильном приложении, для платформ Android и iOS, в этом случае код может выглядеть следующим образом:

```
    #define ANDROID
    #if                                                    ANDROID
//используем              API              OC              Android
#else
//используем                API              OC              iOS
#endif
```

Очень часто при написании приложения, версия Debug, используемая разработчиками для отладки, отличается от версии Release, которая поставляется конечному пользователю. Переключиться между ними можно на панели инструментов, находящейся вверху в центре экрана. Эти версии отличаются обычно тем, что в Debug больше разнообразных проверок на корректность данных, и, возможно, есть элементы управления, которые нужны исключительно для отладки, но ненужные конечному пользователю. И чтобы не удалять их всякий раз, когда компилируешь программу в Release, используется условная компиляция. Для этого в MSVS существует константа DEBUG, автоматически определяемая в соответствующем режиме. Используя этот механизм, можно, например, спрятать кнопку, которая генерирует отладочную информацию, следующим образом:

```
    public class AddObjectForm : Form
    {
        …
        public AddObjectForm()
```

```
        {
            InitializeComponent();
            …
            #if !DEBUG
            CreateRandomDataButton.Visible = false;
            #endif
        }
        …
}
```

В этом коде мы используем директивы препроцессора #if, указывая тот код, который будет скомпилирован только для сборки Release. В нашем случае, мы устанавливаем для созданной кнопки CreateRandomDataButton поле Visible в состояние false.

4.5 Сериализация

Сериализация представляет собой процесс преобразования объекта в поток байтов для хранения объекта или передачи его в память, базу данных или файл. Ее основное назначение — сохранить состояние объекта для того, чтобы иметь возможность воссоздать его при необходимости. Обратный процесс называется десериализацией. С помощью сериализации разработчик может выполнять такие действия, как отправка объекта удаленному приложению посредством веб-службы, передача объекта из одного домена в другой, передача объекта через брандмауэр в виде XML-строки и хранение информации о безопасности или конкретном пользователе, используемой несколькими приложениями.

Сериализация бывает нескольких видов:

- *Двоичная сериализация*. При двоичной сериализации используется двоичная кодировка, обеспечивающая компактную сериализацию объекта для хранения или передачи в сетевых потоках на основе сокетов.
- *XML-сериализация*. При XML-сериализации открытые поля и свойства объекта или параметры и возвращаемые значения методов сериализуются в XML-поток. XML-сериализация приводит к образованию строго типизированных классов с открытыми свойствами и полями, которые преобразуются в формат XML. Для управления процессом сериализации или десериализации можно применять атрибуты к классам и членам класса.

- *SOAP-сериализация.* XML-сериализация может также использоваться для сериализации объектов в потоки XML, которые соответствуют спецификации SOAP. SOAP — это протокол, основанный на XML и созданный специально для передачи вызовов процедур с использованием XML. Как и в обычной XML-сериализации, атрибуты можно использовать для управления формой SOAP-сообщений в литеральном стиле, генерируемых веб-службой XML.

Рассмотрим пример сериализации объекта класса Person на примере XML-сериализация. Класс Person описан ниже.

```
public class Person
{
      public string Name { get; set; }
      public string Surname { get; set; }

      public int Age { get; set; }
}
```

Для того, чтобы сериализовать данный объект можно воспользоваться следующим кодом:

```
var writer = new
System.Xml.Serialization.XmlSerializer(typeof(Person));
using (var file = System.IO.File.Create(_filePath)
{
writer.Serialize(file, _person);
file.Close();
}
```

В данном примере данные, которые хранятся в объекте _person, сохраняются в файл с именем _filePath. Адрес файл должен служить параметром метода для сериализации, потому что пользователь должен иметь возможность выбирать куда именно и под каким именем данные должны быть сохранены.

Для десериализации можно воспользоваться следующим кодом:

```
var reader = new
System.Xml.Serialization.XmlSerializer(typeof(Person));
var file = new System.IO.StreamReader(_filePath);
_person = (Person)reader.Deserialize(file);
```

В данном случае, данные загружаются из файла с именем _filePath и присваиваются объекту _person.

При этом в процессе сериализации в той директории, которую укажет пользователь появится файл с именем, которое хранилось в переменной

_filePath. Этот файл можно открыть с помощью блокнота и он может выглядеть следующим образом:

```
<?xml version="1.0"?>
<Person>
  <Name>Jonh</Name>
  <Surname>Connor</Surname>
  <Age>16</Age>
</Person>
```

Хорошо видно, что xml-сериализация точно повторяет структуру сохраняемого объекта, и представляет его в удобном для чтения виде.

4.6 Задание на лабораторную работу

1. Создайте в решении новый проект WinForms (WinForms Application Project) и задайте ему соответствующее имя. Если проект бизнес-логики назван как Model, для проекта пользовательского интерфейса логично дать название View. Данный подход в проектировании архитектуры приложения называется Model-View: когда бизнес-логика и пользовательский интерфейс разделены на разные сборки. В дальнейшем такой подход облегчает ориентирование в рамках проекта. Обратите внимание, что теперь данный проект должен быть стартовым, для этого установите его запускаемым проектом по умолчанию.

Примечание: ранее созданный проект ConsoleLoader теперь можно удалить. Удаление проекта из решения не приводит к его физическому удалению с носителя, в отличие от классов проекта. Помните об этом при удалении каких-либо компонентов проекта.

2. Добавьте в проект View новую форму. Название формы должно отражать назначение формы и оканчиваться словом Form. Как и имена других классов, имя формы оформляется в стиле Pascal.

3. Добавьте на форму элемент GridControl из панели инструментов. Для повышения удобства пользовательского интерфейса, лучше сначала разместить на форме элемент GroupBox, в который поместить GridControl. Это позволит поместить в заголовок GroupBox фразу, поясняющую назначение GridControl. Под GridControl разместите две кнопки Button. Назовите кнопки Add Object и Remove Object, где вместо Object подставьте название того объекта, который реализован в вашей бизнес-логике.

4. Создайте внутри формы поле, хранящее список (List) сущностей, соответствующих вашему варианту. Список должен иметь возможность

хранения в себе все дочерние классы вашей сущности (все виды геометрических фигур, все типы работников, все виды скидок и т.д.).

5. Необходимо реализовать следующую логику формы: GridControl должен отображать (без возможности редактирования) все объекты созданного списка. Кнопка Add Object должна добавлять новый объект в GridControl и в список объектов. Кнопка Remove Object должна удалять выбранный в GridControl объект и удалять его из списка объектов.

6. Для добавления новых объектов в программу нужно разработать специальную форму, которая вызывалась бы по нажатию клавиши Add Object. В форме должна присутствовать возможность заполнения полей, общих для всех дочерних классов, выбор в виде ComboBox или RadioButton типа объекта, и, в зависимости от типа объекта, должна появляться возможность заполнения полей данного типа объекта. Например, если создается новый работник, то в форме обязательно есть поля ФИО и даты принятия на работу, но в зависимости от RadioButton с типом оплаты должны появляться поля либо почасовой оплаты, либо оплаты по ставке.

7. На форме создания нового объекта должны присутствовать кнопки Ok и Cancel. Если пользователь нажмет кнопку Ok – в главной форме должен быть добавлен созданный объект. Если пользователь нажмет Cancel – должна быть выполнена отмена добавления.

8. Форма создания нового объекта должна учитывать ограничения на значения полей объекта (например, неотрицательный размер стороны геометрической фигуры). Фактически, здесь должна производиться обработка исключений при попытке ввода неправильных значений.

9. Особое внимание обратите на визуальную аккуратность создаваемых вами пользовательских интерфейсов. Старайтесь выравнивать элементы по левому краю относительно друг друга, делать одинаковые отступы между элементами, правильно подписывать элементы, кнопки и заголовки. Грамотно рассчитывайте размеры элементов – если в TextBox должно вводиться целое число со значением до 100, не имеет смысла делать его длиннее 50 пикселей. Также, поля для фамилии должны быть подходящего размера, чтобы корректно отображать обычную фамилию – не слишком длинным, но и не слишком коротким. Аккуратность и удобство пользовательского интерфейса может стать решающим фактором в выборе именно вашей программы конечным пользователем.

10. При тестировании и отладке программы не очень удобно вручную добавлять новые объекты – каждый раз вводить данные для 10 объектов может сильно пошатнуть психическое состояние разработчика (или вашего преподавателя). Чтобы облегчить тестирование программы, а, значит, и собственную разработку, добавьте на форму создания нового объекта кнопку Create Random Data. По нажатию данной кнопки все поля будут заполняться случайными правильными данными для объекта. Пользователю останется только нажать кнопку Ok для добавления нового объекта на главную форму.

11. Кнопка Create Random Data является отладочной, и в версии, которая будет поставляться конечному пользователю, этой кнопки быть не должно – не будет же бухгалтерия создавать «случайных» работников со случайными зарплатами! Удалять же и заново создавать эту кнопку при необходимости нового установщика опять же не очень удобно – вы можете просто забыть это сделать. Используйте механизм условной компиляции.

12. Добавьте форму, на которой можно будет провести поиск объекта по каждому из полей общих для всех дочерних классов. Помните, что результатом поиска может быть не один объект. Добавьте на главную форму кнопу, для вызова формы поиска.

13. Добавьте возможность сохранения и загрузки введенных пользователем данных, используя любой механизм сериализации, на ваше усмотрение. Сохранять данные необходимо в файл с расширением, которое будет характерно только для вашей программы (не надо использовать известные форматы, например *.doc, *.txt или *.xml).

4.7 Список используемых источников

1. Регулярные выражения. Википедия, свободная энциклопедия. [Электронный ресурс]. – URL: https://ru.wikipedia.org/wiki/ Регулярные_выражения (дата обращения 29.12.2014)
2. Элементы языка регулярных выражений - краткий справочник. Microsoft Software Developer Network [Электронный ресурс]. – URL: http://msdn.microsoft.com/ru-ru/library/az24scfc(v=vs.110).aspx (дата обращения 29.12.2014)
3. Регулярные выражения в .NET Framework. Microsoft Software Developer Network [Электронный ресурс]. – URL: http://msdn.microsoft.com/ru-ru/library/hs600312(v=vs.110).aspx (дата обращения 29.12.2014)

5 Лабораторная работа №5. Проектная документация

Лабораторная работа направлена на обучение разработки проектной документации на созданный программный продукт. Имеющиеся типы проектной документации, которые необходимо включить в конечный отчёт более подробно описаны в [1]. Оформление отчёта согласно стандарту, является обязательным этапом подготовки технического специалиста, т.к. очень часто рабочая документация на предприятии или фирме регламентируется либо внутренними документами, либо ГОСТами. Оформление отчёта можно выполнять в доступном текстовом процессоре (Microsoft Word, OpenOffice, LaTeX, Words и пр.).

5.1 Задание на лабораторную работу

Ниже представлены основные пункты отчёта, которые необходимо включить в конечный документ:

1. Титульный лист;
2. Содержание. При составлении содержания ознакомьтесь с инструментами автоматической генерации содержания по имеющемуся документу (с автоматической вставкой названий глав и номеров страниц). Изучение этой возможности текстового процессора поможет в дальнейшем сэкономить большое количество времени при добавлении/удалении глав из отчёта.
3. Введение, в котором необходимо описать назначение программной документации, разрабатываемой в лабораторной работе.
4. Основная часть, в которой приводится описание программной системы:
    a. Составьте UML диаграмму вариантов использования для разработанной программы. Подробнее о том, что такое диаграммы вариантов использования и как их составлять, можно прочитать в главе 7 учебного пособия [1];
    b. Составьте UML диаграмму классов. Подробнее о том, что такое диаграммы классов и как их составлять, можно прочитать в главе 7 учебного пособия [1];
    c. Для классов, образующих связь типа «общее-частное» (наследование, реализация) приведите описание. В описание включите имеющиеся поля, свойства и методы класса, их типы и входные параметры в случае методов класса. Пример оформления описания класса Person и Student из главы 7 учебного пособия [1] приведён в Таблица 5.1 и Таблица 5.2.

Таблица 5.1 – Описание класса Person

Название	Тип	Описание
Описание класса		
Класс *Person* – сущность для описания абстрактного человека в программе.		
Свойства		
+ DateOfBirth	Data	Дата рождения человека
+ Name	string	Имя человека
+ Surname	string	Фамилия человека
Методы		
+ DoSomeWork()	void	Виртуальный метод, описывающий некоторую работу, совершаемую человеком. Перегружается в производных классах.
+ GetAge()	int	Метод для расчёта возраста человека. Возвращает возраст человека.
# Person (Person, Data)		Конструктор для создания нового человека с помощью фамилии и имени другого человека и даты рождения. *Person* – человек, на основе имени и фамилии которого, планируется создать новый экземпляр класса. *Data* – дата рождения человека.
# Person (string, string, Date)		Конструктор для создания нового человека с помощью фамилии, имени и даты рождения. *String* – имя человека. *String* – фамилия человека. *Data* – дата рождения человека.
+ TalkAboutOthers (Person)	string	Метод, позволяющий одному человеку рассказать информацию о другом человеке

Таблица 5.2 – Описание класса Student

Название	Тип	Описание
Описание класса		
Класс *Student* – сущность для описания студента университета в программе.		
Методы		

+ DoSomeWork()	void	Перегруженный метод базового класса, в котором описывается процесс обучения студентом.
# Student (Person, Data)		Конструктор для создания нового студента с помощью фамилии и имени другого человека и даты рождения. Вызывает конструктор базового класса.
# Student (string, string, Date)		Конструктор для создания нового студента с помощью фамилии, имени и даты рождения. Вызывает конструктор базового класса.

При описании свойств и методов класса допускается для краткости использовать принятые в UML обозначения модификаторов доступа. Помимо этого, подобное описание в MSDN-подобном [3] формате можно сгенерировать автоматически при наличии качественных XML-комментариев в коде. Использование подобных инструментов и подробный разбор XML-комментариев в пособии по лабораторному практикуму не предусмотрен, однако для общего развития и повышения профессиональных компетенций подробное изучение этого материала вполне оправдано.

5.2 Список использованных источников

1. А.А. Калентьев, Д.В. Гарайс, А.Е. Горяинов Новые технологии в программировании, Учебное пособие, Томск «Эль Контент» 2014, – 176 с.
2. Microsoft Software Developer Network [Электронный ресурс]. – URL: http://msdn.microsoft.com/ru-ru/default.aspx (дата обращения 18.12.2014)