Практическая работа 27
Использование стандартных и пользовательских атрибутов

Цель занятия: Получить практические навыки использования стандартных и пользовательских атрибутов

Перечень оборудования и программного обеспечения

Персональный компьютер
Microsoft Office (Word, Visio)
Microsoft Visual Studio 2010

Краткие теоретические сведения

1 Использование предопределенного атрибута Conditional

Условное выполнение является полезным методом, если вы хотите включить тестирование и отладку кода в проект, но не хотите редактировать проект и удалять код отладки после того, как проект является законченным и функционирует правильно. Атрибут Conditional, который применяется для условного выполнения, может быть полезен при отладке программ, когда Вы имеете ряд процедур, выдающих отладочную информацию. После отладки программы при компиляции release-версии вызовы ваших отладочных процедур не создаются, хотя сами процедуры компилируются.

Атрибут Conditional будем использовать для выполнения метода в зависимости от значения символа под названием DEBUG_ACCOUNT.

Чтобы применить атрибут Conditional

3.1.1. Создайте класс Student как указано в примере:

```
public class Student
   {
public string fam;
public string name;
public void NameToScreen()
      {
Console.WriteLine(" Fam is {0}. Name is {1}. ",
this.fam, this.name);
Console.ReadKey();
}
   }
```

Этот метод должен отображать содержимое записи о студенте: фамилию и имя. Метод NameToScreen отображает содержимое полей класса.

3.1.2. Использование метода в зависимости от символа DEBUG_ACCOUNT.

Добавьте следующий атрибут Conditional перед методом следующим образом:

[Conditional("DEBUG_ACCOUNT")]

Добавьте using для пространства имен System.Diagnostics.

Компилируйте код.

3.1.3. Для тестирования атрибута Conditional

Добавьте следующий код в Main для вызова метода NameToScreen:

```
Student Petrov = new Student();
Petrov.fam = "Petrov";
  Petrov.name = "Ivan";
Petrov.NameToScreen();
```

Запустите тестовую программу.

Обратите внимание, что ничего не происходит.

В верхней части файла Program.cs, перед первым использованием директивы, добавьте следующий код:

```
# define DEBUG_ACCOUNT
```

Это определяет символ DEBUG_ACCOUNT.

Запустите тестовую программу.

Обратите внимание, что метод NameToScreen отображает информацию из Student.

2 Использование пользовательского атрибута

В этом примере показывается, как создать простой атрибут, документирующий некоторый фрагмент кода. Атрибут из этого примера содержит информацию об имени и уровне программиста, а также о времени последнего пересмотра кода. Он содержит три закрытых переменных, в которых хранятся данные. Каждая переменная связана с открытым свойством для чтения и записи значений. Также имеется конструктор с двумя обязательными параметрами.

Пример: В классе создайте следующий код

```
[Attribute Usage(AttributeTargets.All)]
public class DeveloperAttribute : System.Attribute
    {
      // Закрытые поля.
private string name;
private string level;
private bool reviewed;
```

```
// Конструктор принимает два обязательных параметра: имя и уровень.
Public DeveloperAttribute (string name, string level)
        {
           this.name = name;
           this.level = level;
           this.reviewed = false;
        }

        // Свойство Name.
        // Толькодлячтения.
public virtual string Name
        {
get { return name; }
        }

// Свойство Level.
        // Только для чтения.
public virtual string Level
        {
get { return level; }
        }

        // Свойство Reviewed.
        // Чтение / Запись.
public virtual bool Reviewed
        {
get { return reviewed; }
set { reviewed = value; }
}
      }
```

В головном модуле сначала объявляется переменная с типом атрибута, который нужно получить, затем она инициализируется с помощью вызова метода Attribute.GetCustomAttribute. Теперь можно использовать любые доступные свойства атрибута.

В следующем примере атрибут DeveloperAttribute применяется к классу Program в целом. Метод GetAttribute использует Attribyte.GetCustomAttribute для получения состояния атрибута DeveloperAttribute перед тем, как вывести информацию на консоль.

Для тестирования атрибута добавьте следующий код в Main:

```
[Developer("ИванСеменов", "Тестировщик", Reviewed = true)]
class Program
   {
static void Main(string[] args)
```

```
        {
GetAttribute(typeof(Program));
        }

public static void GetAttribute(Type t)
        {
            // Получитьатрибут.
DeveloperAttributeMyAttribute =
              (DeveloperAttribute)Attribute.GetCustomAttribute(t,
typeof(DeveloperAttribute));

if (MyAttribute == null)
            {
Console.WriteLine("Атрибут не найден.");
Console.ReadKey();
            }
else
            {
               // Получить поле Имя.
Console.WriteLine("Имя: {0}.", MyAttribute.Name);
               // ПолучитьполеУровень.
Console.WriteLine("Уровень: {0}.", MyAttribute.Level);
               // ПолучитьполеПроверено.
Console.WriteLine("Проверено: {0}.", MyAttribute.Reviewed);
Console.ReadKey();
            }
        }
    }
```

Задания

1 Изучить теоретические сведения и задание к работе.

2 Создать методы с использованием атрибута Conditional. Использовать DEBUG_ACCOUNT для демонстрации атрибута.

3 В соответствии с вариантом задания составить отлаженную программу с простым пользовательским атрибутом.

Порядок выполнения работы

Пример условного метода

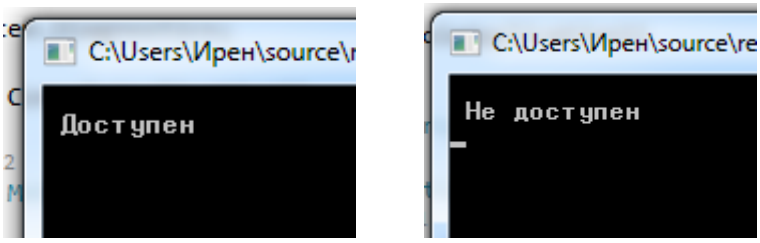
Атрибут Conditional позволяет создавать условные методы, которые вызываются только в том случае, если с помощью директивы #define

определен конкретный идентификатор, а иначе метод пропускается. Следовательно, условный метод служит альтернативой условной компиляции по директиве #if.

Создадим класс Motorcycle, а в нем два метода, один для использования мотоцикла, в нем сообщение Доступен, другой с сообщением Не доступен. К каждому методу добавить атрибут Conditional с соответствующим параметром.

```
eApp11                                          ConsoleApplication11.
#define NoUse
using System;
using System.Diagnostics;

namespace ConsoleApplication11
{
    Ссылок: 2
    class Motorcycle
    {
        [Conditional("Use")]
        ссылка: 1
        public void UseMotorcycle()
        {
            Console.WriteLine("Доступен");
        }

        [Conditional("NoUse")]
        ссылка: 1
        public void NoUseMotorcycle()
        {
            Console.WriteLine("Не доступен");
        }
    }

    Ссылок: 0
    class Program
    {
        Ссылок: 0
        static void Main()
        {
            Motorcycle moto = new Motorcycle();
            moto.UseMotorcycle();
            moto.NoUseMotorcycle();
            Console.ReadLine();
        }
    }
}
```

Если менять код #define Use / NoUse, на консоль будет выдано одно из сообщений метода:



Пример создания атрибутного класса

Рассмотрим пример создания собственного атрибутного класса. Атрибуты этого класса позволят запоминать историю создания и модификации программных сущностей проекта. В атрибуте будет сохраняться *информация* об авторе, который создает или вносит изменения в код, дате внесения изменений и сопутствующий комментарий.

```
[AttributeUsage (AttributeTargets.All,
      AllowMultiple = true, Inherited = true)]
   class HistoryAttribute: Attribute
   {
      //позиционные параметры
      string author;
      string date;

     // именованный параметр
      public string comment;
```

Атрибут Usage заданный для нашего атрибутного класса History, указывает, что *атрибут* может быть задан для всех программных сущностей и может появляться в нескольких экземплярах. Поля этого класса содержат информацию, которую предполагается задавать в момент определения атрибута. *Автор* и дата относятся к обязательным параметрам. Именованный *параметр* comment может опускаться.

Дополним *класс* заданием конструктора и методами свойствами, осуществляющими *доступ* к закрытым полям класса:

```
public HistoryAttribute(string author, string date)
      {
         this.author = author; this.date = date;
      }
      public string Author
      {
         get { return author; }
      }
      public string Date
      {
```

```
        get { return date; }
    }
}
```

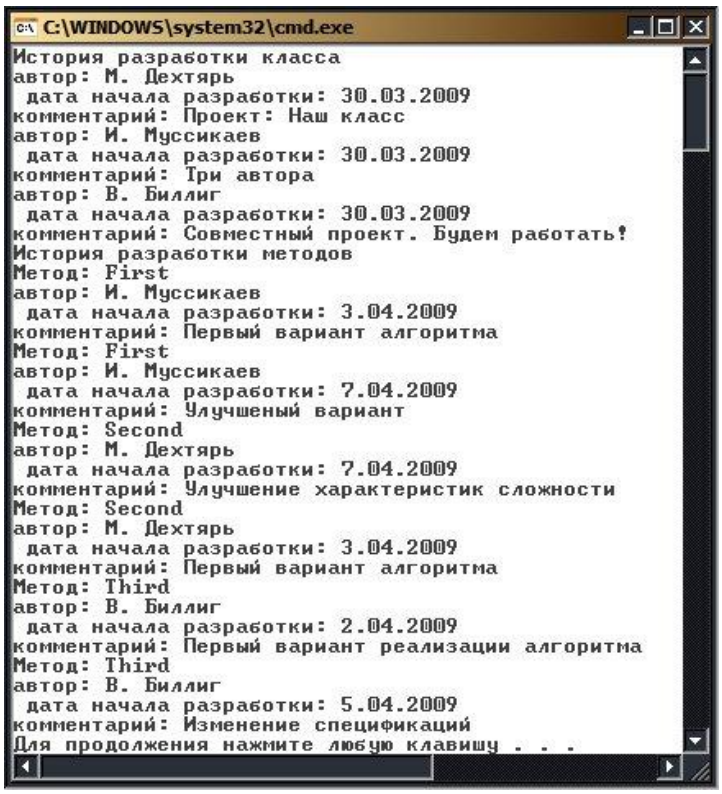
Теперь атрибутный *класс* полностью определен, и его можно использовать в процессе разработки программного проекта. В качестве модельного примера приведу *класс*, сохраняющий историю разработки:

```
[History("В. Биллиг", "30.03.2009",
    comment = "Совместный проект. Будем работать!")]
[History("М. Дехтярь", "30.03.2009",
    comment = "Проект: Наш класс")]
[History("И. Муссикаев", "30.03.2009",
    comment = "Три автора")]
class OurClass
{
    [History("И. Муссикаев", "3.04.2009",
     comment = "Первый вариант алгоритма")]
    [History("И. Муссикаев", "7.04.2009",
    comment = "Улучшеный вариант")]
    public void First()
    {
    }

    [History("М. Дехтярь", "3.04.2009",
    comment = "Первый вариант алгоритма")]
    [History("М. Дехтярь", "7.04.2009",
    comment = "Улучшение характеристик сложности")]
    public void Second()
    {
    }

    [History("В. Биллиг", "2.04.2009",
    comment = "Первый вариант реализации алгоритма")]
    [History("В. Биллиг", "5.04.2009",
          comment = "Изменение спецификаций")]
    public void Third()
    {
    }
}
```

Как видите, *атрибут* позволяет хранить вместе с кодом и историю внесения изменений. Процесс отражения позволит при необходимости извлечь историю разработки. Объяснение того, как это можно делать, уже дано. Поэтому просто приведу код, выполняющий эту работу:

```
public void TestOurClass()
   {
      //История разработки класса
      Console.WriteLine("История разработки класса");
      Type clstype = typeof(OurClass);
      Attribute[] attrs;
      HistoryAttribute history;
      attrs = Attribute.GetCustomAttributes(clstype);
      foreach(Attribute attr in attrs)
      {
         history = attr as HistoryAttribute;
         if (history != null)
            Console.WriteLine("автор: {0}", history.Author);
            Console.WriteLine(" дата начала разработки: {0} ",
                  history.Date);
            Console.WriteLine("комментарий: {0} ",
               history.comment);
      }
      Console.WriteLine("История разработки методов");
      MethodInfo[] methods = clstype.GetMethods();
      foreach (MethodInfo method in methods)
      {
         attrs = Attribute.GetCustomAttributes(method);
         foreach(Attribute attr in attrs)
         {
            history = attr as HistoryAttribute;
            if (history != null)
            Console.WriteLine("Метод: {0} ", method.Name);
            Console.WriteLine("автор: {0}", history.Author);
            Console.WriteLine(" дата начала разработки: {0}",
                  history.Date);
            Console.WriteLine("комментарий: {0} ",
                  history.comment);
         }
      }
   }
```

Вот результаты работы этого метода.



Содержание отчета

1 Название работы
2 Цель работы
3 Технические средства обучения
4 Задания (условия задач)
5 Порядок выполнения работы
6 Ответы на контрольные вопросы
7 Вывод

Варианты заданий

Создать в классе метод с использованием атрибута Conditional в соответствии с вариантами практической работы 23.

Контрольные вопросы

1. Что такое атрибуты?
2. Назначение атрибутов.
3. Где хранится информация о атрибутах?
4. Описание атрибута.
5. Назначение условных атрибутов.

Используемая литература

1 Гниденко, И. Г. Технология разработки программного обеспечения: учеб. пособие для СПО / И. Г. Гниденко, Ф. Ф. Павлов, Д. Ю. Федоров. — М.: Издательство Юрайт, 2017.

2 Шарп Джон Ш26 Microsoft Visual C#. Подробное руководство. 8-е изд. — СПб.: Питер, 2017.

3 Васильев А.Н. Программирование на C# для начинающих. Основные сведения. – Москва: Эксмо, 2018.

4 Васильев А.Н. Программирование на C# для начинающих. Особенности языка. – Москва: Эксмо, 2019.

5 http://msdn.microsoft.com/ru-ru/library/67ef8sbd.aspx.