**Федеральное государственное бюджетное образовательное учреждение**

**высшего образования**

**«Московский государственный технический университет**

**имени Н.Э. Баумана**

**(национальный исследовательский университет)»**

**(МГТУ им. Н.Э. Баумана)**

Факультет «Информатика и вычислительная техника»

Кафедра ИУ5 «Системы обработки информации и управления»

**Отчет по лабораторной работе № 6**

Задание: 1. Разработать программу, использующую делегаты.

2. Разработать программу, реализующую работу с рефлексией

Студент Левин Михаил Александрович

*фамилия, имя, отчество*

Группа ИУ5-32Б

Москва, 2020 г.

# Задание

**Часть 1. Разработать программу, использующую делегаты.**

(В качестве примера можно использовать проект «Delegates»).

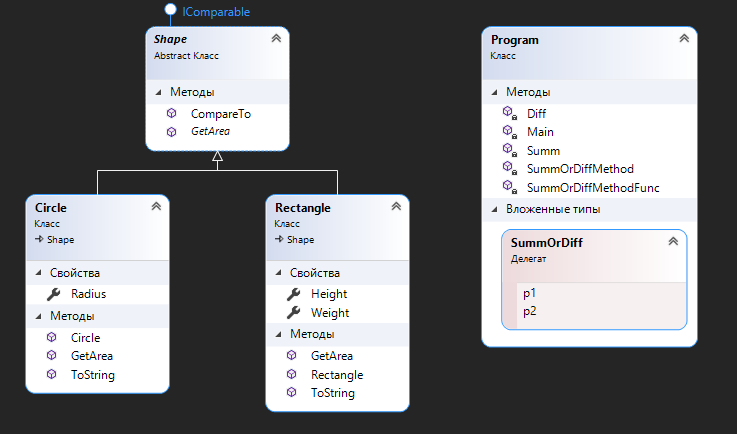
1. Программа должна быть разработана в виде консольного приложения на языке C#.
2. Определите делегат, принимающий несколько параметров различных типов и возвращающий значение произвольного типа.
3. Напишите метод, соответствующий данному делегату.
4. Напишите метод, принимающий разработанный Вами делегат, в качестве одного из входным параметров. Осуществите вызов метода, передавая в качестве параметра-делегата:
   * метод, разработанный в пункте 3;
   * лямбда-выражение.
5. Повторите пункт 4, используя вместо разработанного Вами делегата, обобщенный делегат Func< > или Action< >, соответствующий сигнатуре разработанного Вами делегата.

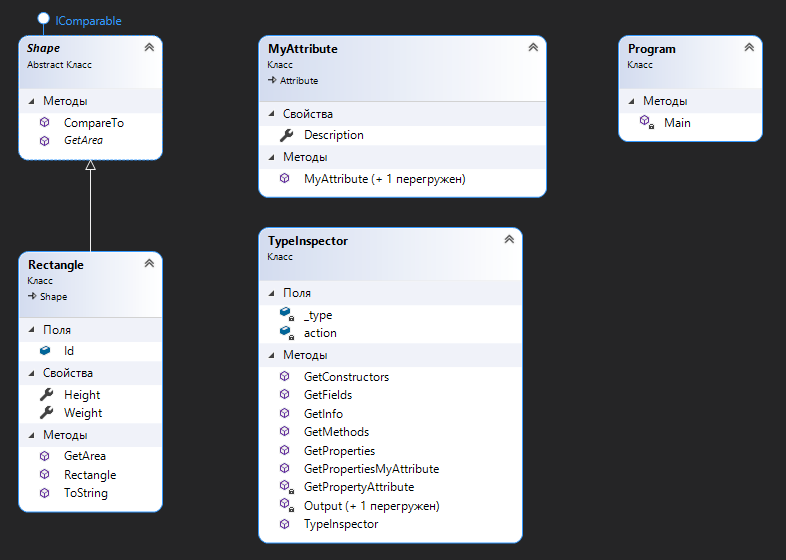
**Часть 2. Разработать программу, реализующую работу с рефлексией.**

(В качестве примера можно использовать проект «Reflection»).

1. Программа должна быть разработана в виде консольного приложения на языке C#.
2. Создайте класс, содержащий конструкторы, свойства, методы.
3. С использованием рефлексии выведите информацию о конструкторах, свойствах, методах.
4. Создайте класс атрибута (унаследован от класса System.Attribute).
5. Назначьте атрибут некоторым свойствам классам. Выведите только те свойства, которым назначен атрибут.
6. Вызовите один из методов класса с использованием рефлексии.

# Диаграмма классов





# Текст программы

**Проект DelegateSample**

**Circle.cs**

using System;

namespace Lab6

{

class Circle:Shape

{

public double Radius { get; set; }

public Circle(double r)

{

Radius = r;

}

public override double GetArea()

{

return Math.PI \* Radius \* Radius;

}

public override string ToString()

{

return $"Круг. Радиус: {Radius} Площадь: {GetArea()}";

}

}

}

**Rectangle.cs**

namespace Lab6

{

/// <summary>

/// Класс прямоугольник

/// </summary>

class Rectangle:Shape

{

public double Height { get; set; }

public double Weight { get; set; }

public override double GetArea()

{

return Height \* Weight;

}

public Rectangle(double w, double h)

{

Weight = w;

Height = h;

}

public override string ToString()

{

return $"Прямоугольник. Ширина: {Weight}. Высота: {Height}. Площадь: {GetArea()}";

}

}

}

**Shape.cs**

using System;

namespace Lab6

{

/// <summary>

/// Абстрактный класс геометрическая фигура

/// </summary>

abstract class Shape:IComparable

{

/// <summary>

/// Метод вычисления площади

/// </summary>

/// <returns></returns>

public abstract double GetArea();

public int CompareTo(object obj)

{

double areaCurrent = GetArea();

double areaObj = (obj as Shape).GetArea();

if (areaCurrent > areaObj) return 1;

if (areaCurrent < areaObj) return -1;

return 0;

}

}

}

**Programs.cs**

using System;

namespace Lab6

{

class Program

{

//делагат, описывающий метод принимающий два вещественных параметра и возвращающий вещественное значение

delegate double SummOrDiff(double p1, double p2);

//Методы, реализующие делегат (методы "типа" делегата)

static double Summ(double a1, double a2) { return a1 + a2; } //сумма

static double Diff(double a1, double a2) { return Math.Abs(a1 - a2); } //разница (по модулю)

/// <summary>

/// Использование обощенного делегата Func<>

/// </summary>

static void SummOrDiffMethodFunc(string str, double i1, double i2, Func<double, double, double> SummOrDiffParam)

{

double result = SummOrDiffParam(i1, i2);

Console.WriteLine(str + result);

}

/// <summary>

/// Использование делегата

/// </summary>

static void SummOrDiffMethod(string str, double i1, double i2, SummOrDiff SummOfDiffParam)

{

double result = SummOfDiffParam(i1, i2);

Console.WriteLine(str + result);

}

static void Main(string[] args)

{

var rectangle = new Rectangle(5,6);

var circle = new Circle(7);

double rectAr = rectangle.GetArea();

double circAr = circle.GetArea();

Console.WriteLine(rectangle);

Console.WriteLine($"{circle}\n");

SummOrDiffMethod("Суммы площадей: ",rectAr, circAr,Summ);

SummOrDiffMethod("Разность площадей: ", rectAr, circAr, Diff);

//Создание экземпляра делегата на основе метода

SummOrDiff pm1 = new SummOrDiff(Summ);

SummOrDiffMethod("Создание экземпляра делегата на основе метода: ", rectAr, circAr, pm1);

//Создание экземпляра делегата на основе 'предположения' делегата

//Компилятор 'пердполагает' что метод Plus типа делегата

SummOrDiff pm2 = Summ;

SummOrDiffMethod("Создание экземпляра делегата на основе 'предположения' делегата: ", rectAr, circAr, pm2);

//Создание анонимного метода

SummOrDiff pm3 = delegate (double param1, double param2)

{

return param1 + param2;

};

SummOrDiffMethod("Создание экземпляра делегата на основе анонимного метода: ", rectAr, circAr, pm3);

SummOrDiffMethod("Создание экземпляра делегата на основе лямбда-выражения 1: ", rectAr, circAr, (double x, double y) =>

{

double z = x + y;

return z;

}

);

SummOrDiffMethod("Создание экземпляра делегата на основе лямбда-выражения 2: ", rectAr, circAr, (x, y) =>

{

return x + y;

}

);

SummOrDiffMethod("Создание экземпляра делегата на основе лямбда-выражения 3: ", rectAr, circAr, (x, y) => x + y);

Console.WriteLine("\n\nИспользование обощенного делегата Func<>");

SummOrDiffMethodFunc("Создание экземпляра делегата на основе метода:", rectAr, circAr, Summ);

string OuterString = "ВНЕШНЯЯ ПЕРЕМЕННАЯ";

SummOrDiffMethodFunc("Создание экземпляра делегата на основе лямбда-выражения 1: ", rectAr, circAr,

(double x, double y) =>

{

Console.WriteLine("Эта переменная объявлена вне лямбда-выражения: " + OuterString);

double z = x + y;

return z;

}

);

SummOrDiffMethodFunc("Создание экземпляра делегата на основе лямбда-выражения 2: ", rectAr, circAr, (x, y) =>

{

return x + y;

}

);

SummOrDiffMethodFunc("Создание экземпляра делегата на основе лямбда-выражения 3: ", rectAr, circAr, (x, y) => x + y);

//////////////////////////////////////////////////////////////

//Групповой делегат всегда возвращает значение типа void

Console.WriteLine("Пример группового делегата");

Action<double, double> a1 = (x, y) =>

{

Console.WriteLine("{0} + {1} = {2}", x, y, x + y);

};

Action<double, double> a2 = (x, y) =>

{

Console.WriteLine("{0} - {1} = {2}", x, y, Math.Abs(x - y));

};

Action<double, double> group = a1 + a2;

group(rectAr, circAr);

Action<double, double> group2 = a1;

Console.WriteLine("Добавление вызова метода к групповому делегату");

group2 += a2;

group2(rectAr, circAr);

Console.WriteLine("Удаление вызова метода из группового делегата");

group2 -= a1;

group2(rectAr, circAr);

}

}

}

**Проект ReflectionSimple**

**MyAttribute.cs**

using System;

namespace ReflectionSample

{

/// <summary>

/// Класс атрибута

/// </summary>

[AttributeUsage(AttributeTargets.Property, AllowMultiple = false, Inherited = false)]

public class MyAttribute : Attribute

{

public MyAttribute() { }

public MyAttribute(string descriptionParam)

{

Description = descriptionParam;

}

public string Description { get; set; }

}

}

**Shape.cs**

using System;

namespace ReflectionSample

{

/// <summary>

/// Абстрактный класс геометрическая фигура

/// </summary>

abstract class Shape:IComparable

{

/// <summary>

/// Метод вычисления площади

/// </summary>

/// <returns></returns>

public abstract double GetArea();

public int CompareTo(object obj)

{

double areaCurrent = GetArea();

double areaObj = (obj as Shape).GetArea();

if (areaCurrent > areaObj) return 1;

if (areaCurrent < areaObj) return -1;

return 0;

}

}

}

**Rectangle.cs**

namespace ReflectionSample

{

/// <summary>

/// Класс прямоугольник

/// </summary>

class Rectangle:Shape

{

[MyAttribute("Свойство - высота прямоугольника")]

public double Height { get; set; }

[MyAttribute(Description = "Свойство - ширина прямоугольника")]

public double Weight { get; set; }

public int Id; //идентификатор прямоугольника

public override double GetArea()

{

return Height \* Weight;

}

public Rectangle(double w, double h)

{

Weight = w;

Height = h;

}

public override string ToString()

{

return $"Прямоугольник. Ширина: {Weight}. Высота: {Height}. Площадь: {GetArea()}";

}

}

}

**TypeInspector.cs**

using System;

using System.CodeDom;

using System.Collections.Generic;

using System.Linq;

using System.Reflection;

using System.Text;

using System.Threading.Tasks;

namespace ReflectionSample

{

/// <summary>

/// Класс для обследования типо с помощь рефлекции

/// </summary>

public class TypeInspector

{

private readonly Type \_type; //обследуемый тип

private Action<string> action; //делегат метода вывода строки

public TypeInspector(Type t, Action<string> a )

{

\_type = t;

action = a;

}

private bool GetPropertyAttribute(PropertyInfo checkType, Type attributeType, out object attribute)

{

bool Result = false;

attribute = null;

//Поиск атрибутов с заданным типом

var isAttribute = checkType.GetCustomAttributes(attributeType, false);

if (isAttribute.Length > 0)

{

Result = true;

attribute = isAttribute[0];

}

return Result;

}

private void Output(string str)

{

action(str);

}

private void Output(object[] args)

{

foreach (var x in args)

{

Output(x.ToString());

}

}

/// <summary>

/// Выводит общую информацию о типе

/// </summary>

public void GetInfo()

{

Output("Тип " + \_type.FullName + " унаследован от " + \_type.BaseType.FullName);

Output("Пространство имен " + \_type.Namespace);

Output("Находится в сборке " + \_type.AssemblyQualifiedName);

}

public void GetConstructors()

{

Output(\_type.GetConstructors());

}

public void GetMethods()

{

Output(\_type.GetMethods());

}

public void GetProperties()

{

Output(\_type.GetProperties());

}

public void GetFields()

{

Output(\_type.GetFields());

}

public void GetPropertiesMyAttribute()

{

foreach (var x in \_type.GetProperties())

{

if (GetPropertyAttribute(x, typeof(MyAttribute), out object attrObj))

{

MyAttribute attr = attrObj as MyAttribute;

Output(x.Name + " - " + attr.Description);

}

}

}

}

}

**Program.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Reflection;

using System.Text;

using System.Threading.Tasks;

namespace ReflectionSample

{

class Program

{

static void Main(string[] args)

{

Type t = typeof(Rectangle); //получаем тип обследуемого класса

var typeInspector = new TypeInspector(t, Console.WriteLine);

//выводим на консоль основные свойста типа (имя, сборку, базовый тип)

typeInspector.GetInfo();

Console.WriteLine("\nКонструкторы:");

typeInspector.GetConstructors();

Console.WriteLine("\nМетоды:");

typeInspector.GetMethods();

Console.WriteLine("\nСвойства:");

typeInspector.GetProperties();

Console.WriteLine("\nПоля данных (public):");

typeInspector.GetFields();

Console.WriteLine($"\nСвойства, помеченные атрибутом MyAttribute: ");

typeInspector.GetPropertiesMyAttribute();

Console.WriteLine("\nВызов метода:");

//Создание экземпляра прямоугольника через рефлексию (с передачей параметров в конструктор)

Rectangle rectangle = (Rectangle)t.InvokeMember(null, BindingFlags.CreateInstance, null, null, new object[] {3,2});

//Параметры вызова метода

object[] parameters = { } ;

//Вызов метода ToString

object result = t.InvokeMember("ToString", BindingFlags.InvokeMethod, null, rectangle, parameters);

Console.WriteLine($"ToString: {result}");

}

}

}

# Экранные формы

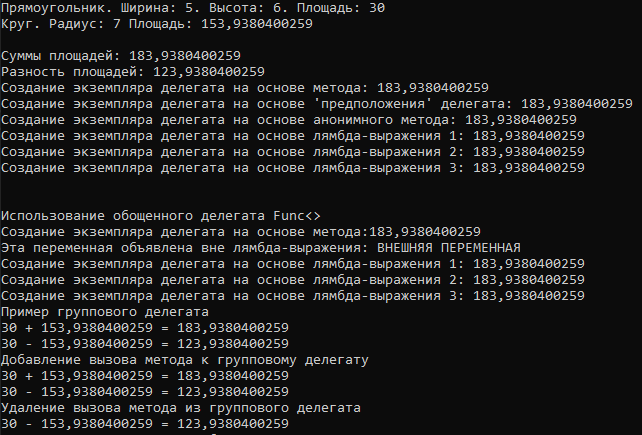


Рисунок 1 – результат работы приложения проекта DelegateSample

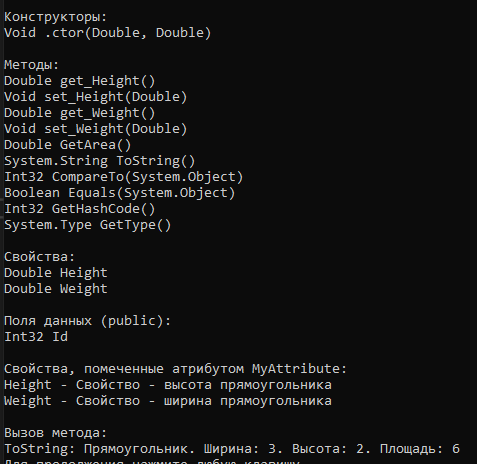


Рисунок 2 – результат работы приложения ReflectionSample

# Контрольные вопросы:

1. Что такое делегат?

Делегат — это тип, который представляет ссылки на методы с определенным списком параметров и типом возвращаемого значения. При создании экземпляра делегата этот экземпляр можно связать с любым методом с совместимой сигнатурой и типом возвращаемого значения. Метод можно вызвать (активировать) с помощью экземпляра делегата.

1. Что такое неявная типизация и как она используется для делегатов?

Неявная типизация - это вывод типа объекта компилятором во время его создания без указания типа программистом на основе предположения о том, что тип делегата. Неявная типизация используется для сокращения записи при создании экземпляров делегата.

1. Как используется обобщенный делегат Func?

Обобщенный делегат используется следующим образом: все аргументы, кроме последнего являются входными, а последний параметр является возвращаемым.

1. Как используется обобщенный делегат Action?

Обобщенный делегат Action, используется для работы с методами, у которых возвращаемое значение типа void.

1. Что такое лямбда-выражения и как они используются?

Лямбда-выражения представляют упрощенную запись анонимных методов. Лямбда-выражения позволяют создать емкие лаконичные методы, которые могут возвращать некоторое значение и которые можно передать в качестве параметров в другие методы. Лямбда-выражения применяются для описания действий, не заключая их в отдельные методы.

1. Что такое рефлексия и для чего она используется?

Механизм отражения позволяет получать объекты (типа Type), которые описывают сборки, модули и типы. Отражение можно использовать для динамического создания экземпляра типа, привязки типа к существующему объекту, а также получения типа из существующего объекта и вызова его методов или доступа к его полям и свойствам. Если в коде используются атрибуты, отражение обеспечивает доступ к ним.

1. Как реализуется работа с атрибутами?

Атрибуты в .NET представляют специальные инструменты, которые позволяют встраивать в сборку дополнительные метаданные. Атрибуты могут применяться как ко всему типу (классу, интерфейсу и т.д.), так и к отдельным его частям (методу, свойству и т.д.). Основу атрибутов составляет класс System.Attribute, от которого образованы все остальные классы атрибутов. Например, можно создать пользовательский атрибут, и пометить им необходимые классы, методы и т.д., далее через рефлексию проверять у объектов наличие данных атрибутов

1. Как реализуются динамические действия с объектами классов?

С помощью отражения можно создавать объект, без явного использования его конструктора, так же можно вызывать методы объекта используя имя метода в качестве параметра метода Type.InvokeMember.

1. Как реализуется работа со сборками?

Для работы со сборками предназначен объект Assembly, который является атрибутов Type.Assembly. У объекта Assembly есть свойства FullName, Location и другие, позволяющие получить сведения о сборке.

1. Как получить информацию о типе на основе класса и на основе инициализированного объекта класса?

Для получения информации о типа, не основе класса, используется оператор typeof, которому в качестве параметра передаётся класс, в результате использования оператора будет получен объект Type переданного класса.

У инициализированного объекта класса получить информацию о типе можно двумя способами: 1) с использование оператора typeof, по аналогии с классом; 2) использование метода GetType() у объекта. Принадлежность объекта к классу можно проверить с использование ключевого слова «is».