|  |  |
| --- | --- |
|  |  |
| МИНОБРНАУКИ РОССИИ | | |
| Федеральное государственное бюджетное образовательное учреждение  высшего профессионального образования  **«Московский технологический университет»**  **МИРЭА** | | |

Институт Информационных Технологий

Кафедра Корпоративных Информационных Систем

**ОТЧЕТ**

по Лабораторной Работе №3

на тему

«RTTI»

по дисциплине

«Технологии программирования»

Выполнил студент группы ИСБО-11-16 Шайхуллин С.В.

Принял Cтарший преподаватель Мирзоян Д.И.

Выполнено «\_\_\_» \_\_\_\_\_\_\_\_\_2017 г.

Зачтено «\_\_\_» \_\_\_\_\_\_\_\_\_ 2017г.

Москва, 2017

**Теоретическая часть**

Динамическая идентификация типа данных (англ. run-time type information, run-time type identification, RTTI) — механизм в некоторых языках программирования, который позволяет определить тип данных переменной или объекта во время выполнения программы.

Существует множество реализаций такого механизма, но наиболее распространёнными являются:

таблица указателей на объекты;

хранение информации об объекте в памяти вместе с ним.

Таким образом, операция определения типа сводится либо к поиску в таблице, либо к просмотру нескольких байт до адреса, на который указывает указатель на объект. У каждого способа есть свои преимущества и недостатки

В C# для определения типа объекта во время исполнения используется метод GetType, а также ключевые слова is и as, которые являются аналогами для typeid и dynamic\_cast в C++ соответственно.

**Задание**

Создать приложение, выводящее информацию RTTI для каждого узла дерева объектов.

Вывести:

1. Название класса
2. Название класса-предка
3. Область видимости класса
4. Название сборки, в которой класс определен
5. Публичный интерфейс класса:

А) список полей с типами

Б) список свойств с типами и указанием на наличие методов считывания и установки значения

В) список методов с типами возвращемых значений, названиями и типами аргументов

1. Определяемый классом интерфейс:

А) список полей с типами

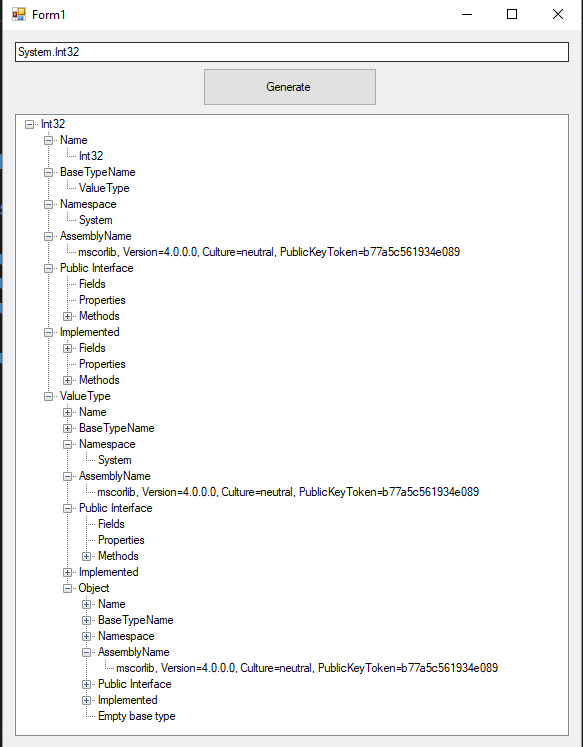
Б) список свойств с типами и указанием на наличие методов считывания и установки значения

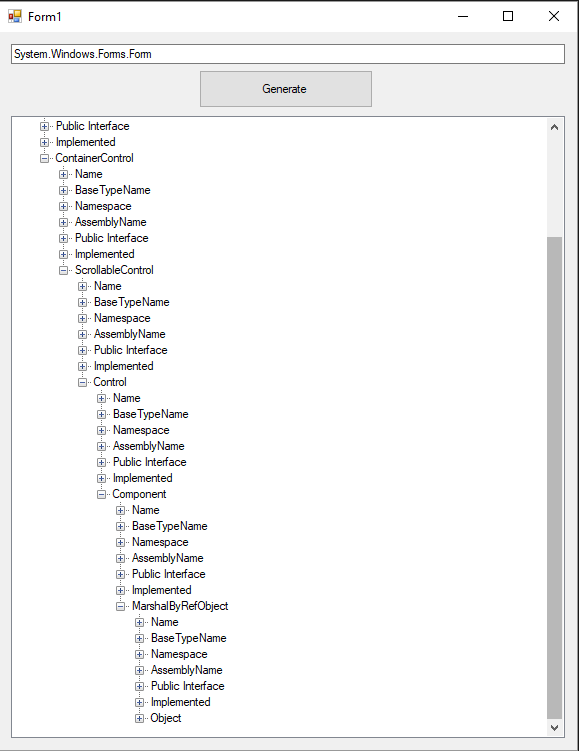
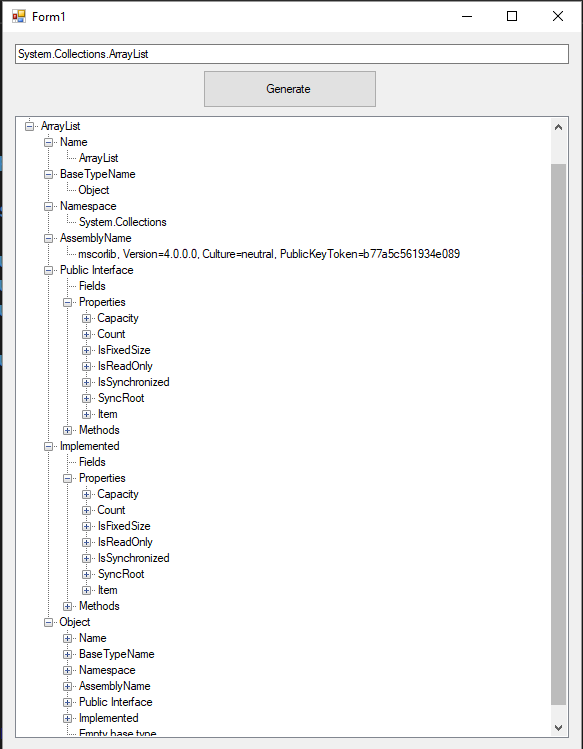
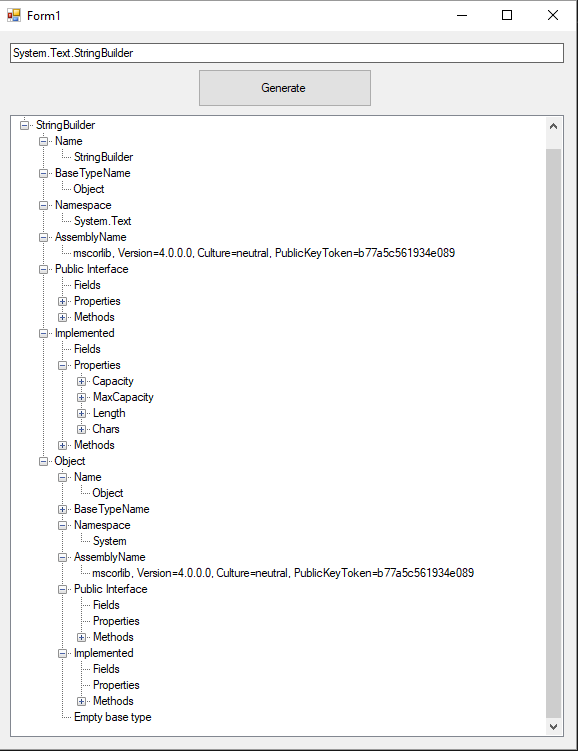
В) список методов с типами возвращемых значений, названиями и типами аргументов

Для каждого элемента следует также указать спецификатор области видимости. Если метод виртуальный, необходимо это указать, а также указать класс, в котором находится реализация этого метода и класс, в котором он был впервые объявлен.

Дополните данную схему классов информацией о членах (полях, методах и свойствах) каждого класса. Если виртуальный элемент впервые объявлен в данном классе, выделите его зеленым, если виртуальный элемент переопределен — желтым. Невиртуальные элементы должны иметь черный цвет, переопределенные —красный.

Дополните данную схему классов информацией о членах (полях, методах и свойствах) каждого класса. Если виртуальный элемент впервые объявлен в данном классе, выделите его зеленым, если виртуальный элемент переопределен — желтым. Невиртуальные элементы должны иметь черный цвет, переопределенные —красный.

**Пример работы**



**Заключение**

В данный лабораторной работе мы научились использовать ефлексию для получения сведений о тпе, его предках, методах, перегрузках и так далее

**Исходный код**

using System.Reflection;

using System.Windows.Forms;

namespace Lab3

{

class ArgumentDto

{

public string Name { get; set; }

public string Type { get; set; }

public ArgumentDto(ParameterInfo x)

{

Name = x.Name;

Type = x.ParameterType.Name;

}

public TreeNode ToTree()

{

return new TreeNode(Name, new[]

{

new TreeNode(Name),

new TreeNode(Type)

});

}

}

}

using System.Reflection;

using System.Windows.Forms;

namespace Lab3

{

class FieldDto

{

public string AccessModifier { get; set; }

public string Name { get; set; }

public string Type { get; set; }

public FieldDto(FieldInfo x)

{

if (x.IsPublic)

{

AccessModifier = "Public";

}

else if (x.IsPrivate)

{

AccessModifier = "Private";

}

else if (x.IsFamily)

{

AccessModifier = "Protected";

}

else if(x.IsFamilyAndAssembly)

{

AccessModifier = "Internal";

}

else if(x.IsFamilyOrAssembly)

{

AccessModifier = "Protected internal";

}

Name = x.Name;

Type = x.DeclaringType.Name;

}

public TreeNode ToTree()

{

return new TreeNode(Name, new[]

{

new TreeNode("AccessModifier", new[]{ new TreeNode(AccessModifier)}),

new TreeNode("Name", new[]{ new TreeNode(Name) }),

new TreeNode("Type", new[]{ new TreeNode(Type) }),

});

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Windows.Forms;

namespace Lab3

{

class ImplementedInterfaceDto

{

public List<FieldDto> Fields { get; set; }

public List<PropertyDto> Properties { get; set; }

public List<MethodDto> Methods { get; set; }

public ImplementedInterfaceDto(Type type)

{

Fields = type.GetFields()

.Select(fieldInfo => new FieldDto(fieldInfo))

.ToList();

Properties = type.GetProperties()

.Select(propertyInfo => new PropertyDto(propertyInfo))

.ToList();

Methods = type.GetMethods()

.Select(methodInfo => new MethodDto(methodInfo))

.ToList();

}

public TreeNode ToTree()

{

return new TreeNode("Implemented", new[]

{

new TreeNode("Fields", Fields

.Select(x => x.ToTree())

.ToArray()),

new TreeNode("Properties",

Properties

.Select(x => x.ToTree())

.ToArray()),

new TreeNode("Methods", Methods

.Select(x => x.ToTree())

.ToArray())

});

}

}

}

using System.Linq;

using System.Reflection;

using System.Windows.Forms;

using System.Collections.Generic;

namespace Lab3

{

class MethodDto

{

public string AccessModifier { get; set; }

public string Name { get; set; }

public string Type { get; set; }

public VirtualDto IsVirtual { get; set; }

public List<ArgumentDto> Arguments { get; set; }

public MethodDto(MethodInfo info)

{

AccessModifier = Access.GetAccess(info);

Name = info.Name;

Type = info.ReturnType.Name;

IsVirtual = new VirtualDto(info);

Arguments = info.GetParameters()

.Select(x => new ArgumentDto(x))

.ToList();

}

public TreeNode ToTree()

{

return new TreeNode(Name, new[]

{

new TreeNode(AccessModifier),

new TreeNode(Name),

new TreeNode(Type),

IsVirtual.ToTree(),

new TreeNode("Arguments", Arguments

.Select(x => x.ToTree())

.ToArray())

});

}

}

}

using System.Reflection;

using System.Windows.Forms;

namespace Lab3

{

class PropertyDto

{

public string AccessModifier { get; set; }

public string Name { get; set; }

public string Type { get; set; }

public bool HasGet { get; set; }

public bool HasSet { get; set; }

public PropertyDto(PropertyInfo x)

{

var get = x.GetGetMethod();

var set = x.GetSetMethod();

AccessModifier = Access.GetAccess(get) ?? Access.GetAccess(set);

Name = x.Name;

Type = x.PropertyType.Name;

HasGet = x.GetGetMethod() != null;

HasSet = x.GetSetMethod() != null;

}

public TreeNode ToTree()

{

return new TreeNode(Name, new[]

{

new TreeNode("AccessModifier", new[]{ new TreeNode(AccessModifier)}),

new TreeNode("Name", new[]{ new TreeNode(Name) }),

new TreeNode("Type", new[]{ new TreeNode(Type) }),

new TreeNode("HasGet", new[]{ new TreeNode(HasGet.ToString()) }),

new TreeNode("HasSet", new[]{ new TreeNode(HasSet.ToString()) })

});

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Reflection;

using System.Windows.Forms;

namespace Lab3

{

class PublicInterfaceDto

{

public List<FieldDto> Fields { get; set; }

public List<PropertyDto> Properties { get; set; }

public List<MethodDto> Methods { get; set; }

public PublicInterfaceDto(Type type)

{

Fields = GetFieldInfo(type)

.Select(x => new FieldDto(x))

.ToList();

Properties = GetPropertyInfo(type)

.Select(x => new PropertyDto(x))

.ToList();

Methods = GetMethodInfo(type)

.Select(x => new MethodDto(x))

.ToList();

}

private IEnumerable<FieldInfo> GetFieldInfo(Type type)

{

while (type != null)

{

foreach (var info in type.GetFields(BindingFlags.Public | BindingFlags.Instance))

yield return info;

type = type.BaseType;

}

}

private IEnumerable<PropertyInfo> GetPropertyInfo(Type type)

{

while (type != null)

{

foreach (var info in type.GetProperties(BindingFlags.Public | BindingFlags.Instance))

yield return info;

type = type.BaseType;

}

}

private IEnumerable<MethodInfo> GetMethodInfo(Type type)

{

while (type != null)

{

foreach (var info in type.GetMethods(BindingFlags.Public | BindingFlags.Instance))

yield return info;

type = type.BaseType;

}

}

public TreeNode ToTree()

{

return new TreeNode("Public Interface", new[]

{

new TreeNode("Fields", Fields

.Select(x => x.ToTree())

.ToArray()),

new TreeNode("Properties",

Properties

.Select(x => x.ToTree())

.ToArray()),

new TreeNode("Methods", Methods

.Select(x => x.ToTree())

.ToArray())

});

}

}

}

using System;

using System.Windows.Forms;

namespace Lab3

{

class TypeDto

{

public string Name { get; set; }

public string BaseTypeName { get; set; }

public string Namespace { get; set; }

public string AssemblyName { get; set; }

public PublicInterfaceDto PublicInterface { get; set; }

public ImplementedInterfaceDto ImplementedInterface { get; set; }

public TypeDto BaseType { get; set; }

public TypeDto(Type type)

{

Name = type.Name;

BaseTypeName = type.BaseType?.Name;

Namespace = type.Namespace;

AssemblyName = type.Assembly.FullName;

PublicInterface = new PublicInterfaceDto(type);

ImplementedInterface = new ImplementedInterfaceDto(type);

if (type.BaseType != null)

{

BaseType = new TypeDto(type.BaseType);

}

}

public TreeNode ToTree()

{

return new TreeNode(Name, new[]

{

new TreeNode("Name", new[] { new TreeNode(Name) }),

new TreeNode("BaseTypeName", new[] { new TreeNode(BaseTypeName) }),

new TreeNode("Namespace", new[] { new TreeNode(Namespace) }),

new TreeNode("AssemblyName", new[] { new TreeNode(AssemblyName) }),

PublicInterface.ToTree(),

ImplementedInterface.ToTree(),

BaseType?.ToTree() ?? new TreeNode("Empty base type")

});

}

}

}

using System.Reflection;

using System.Windows.Forms;

namespace Lab3

{

class VirtualDto

{

public bool IsVirtual { get; set; }

public string DefinitionType { get; set; }

public VirtualDto(MethodInfo info)

{

IsVirtual = info.IsVirtual;

DefinitionType = info.DeclaringType.Name;

}

public TreeNode ToTree()

{

return new TreeNode("Virtuality", new[]

{

new TreeNode("IsVirtual", new[] {new TreeNode(IsVirtual.ToString())}),

new TreeNode("DefinitionType", new[] {new TreeNode(DefinitionType)}),

});

}

}

}

using System.Reflection;

namespace Lab3

{

public static class Access

{

public static string GetAccess(MethodBase x)

{

if (x == null)

{

return null;

}

if (x.IsPublic)

{

return "Public";

}

else if (x.IsPrivate)

{

return "Private";

}

else if (x.IsFamily)

{

return "Protected";

}

else if (x.IsFamilyAndAssembly)

{

return "Internal";

}

else if (x.IsFamilyOrAssembly)

{

return "Protected internal";

}

return null;

}

}

}

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Reflection;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace Lab3

{

public partial class MainWindow : Form

{

public MainWindow()

{

InitializeComponent();

}

private void OnGenerate(object sender, EventArgs e)

{

if(string.IsNullOrWhiteSpace(textBox1.Text))

{

MessageBox.Show("Empty string!");

return;

}

var type = AppDomain.CurrentDomain.GetAssemblies()

.Select(x => x.GetType(textBox1.Text))

.FirstOrDefault(x => x != null);

if(type == null)

{

MessageBox.Show("Type not found!");

return;

}

classTree.Nodes.Add(new TypeDto(type).ToTree());

}

}

}