DOI 10.31891/2307-5732-2023-323-4-262-267
УДК 004.624

ПОЗУР МИХАЙЛО
Вінницький національний технічний університет
ORCID ID: 0009-0003-5225-2453
e-mail: mixalchik545@gmail.com
ВОЙТКО ВІКТОРІЯ
Вінницький національний технічний університет
ORCID: 0000-0002-3329-7256
e-mail: defakfki@i.ua

АНАЛІЗ ПІДХОДІВ ДО МЕТАПРОГРАМУВАННЯ В .NET

Розглянуто та проаналізовано сучасні підходи до метапрограмування, що використовуються в .NET. Одним із підходів є використання механізму рефлексії платформи .NET, що дозволяє працювати з метаданими програми в процесі виконання та підключати нові модулі. Такий підхід дозволяє узагальнити певний функціонал, що спрощує процес розробки, але використання рефлексії має негативний вплив на швидкодію додатку. Іншим підходом є використання функціоналу платформи .NET, що дозволяє генерацію коду під час виконання додатку. Для цього використовуються класи простору імен System.Reflection.Emit, які дозволяють генерувати CIL код. Проте такий підхід потребує знання CIL та особливостей роботи CLR, що ускладнює процес розробки і підтримки програми. В .NET є можливість генерації коду в процесі виконання додатку з використанням декларативного підходу за допомогою механізму виразів. Вирази в .NET являють собою окремі блоки, кожен з яких описує операцію над даними. Такі блоки об'єднуються в дерева виразів, що дозволяє описувати складні операції над даними. Дерево виразів може бути сформоване в процесі виконання додатку та скомпільоване, в результаті чого буде отримано лямбда функцію, що виконує описані виразами операції. Проте такий підхід не надає можливості контролювати процес генерації коду. Окрім генерації CIL, у .NET є механізми генерації вихідного коду різними мовами. В статті розглянуто генерацію коду мови програмування C#. Одним із підходів є використання класів простору імен System.CodeDom, що дозволяє генерацію та компіляцію вихідного коду в процесі виконання програми. Проте, компіляція опирається на використання .NET Framework компілятора, що робить неможливим повноцінне використання такого підходу в нових версіях .NET. Підхід з використанням T4 дозволяє генерацію коду на основі шаблонів перед компіляцією додатку, що надає можливість виявляти синтаксичні помилки в згенерованому коді ще на етапі компіляції. Проте технологія T4 є частиною IDE Visual Studio, що робить її використання неможливим поза межами цієї IDE. Альтернативою T4 виступає технологія Source Generators, яка є частиною .NET Compiler Platform SDK. Оскільки Source Generators є частиною компілятора .NET, то процес генерації коду є складовою процесу компіляції додатку, що надає доступ до великої кількості метаданих. Проте використання Source Generators можливе лише в нових версіях .NET. Коректне поєднання підходів до метапрограмування дозволить створити ефективні інструменти для оптимізації процесу розробки .NET додатків.

Ключові слова: метапрограмування, .NET, C#, рефлексія, генерування коду, Reflection.Emit, Expression, System.CodeDom, T4, Source Generators, Roslyn.

POZUR MYKHAYLO
Vinnytsia National Technical University
VOITKO VIKTORIIA
Vinnytsia National Technical University

ANALYSIS OF METAPROGRAMMING APPROACHES IN .NET

Modern approaches to metaprogramming used in .NET are reviewed and analyzed. One approach is to use the reflection mechanism of the .NET platform, which allows to work with application metadata at runtime and plug in new modules. This approach allows to generalize certain functionality, which simplifies the development process, but the use of reflection has a negative effect on the application performance. Another approach is to use the functionality of the .NET framework, which allows code generation in runtime. For this, the System.Reflection.Emit namespace classes are used, which allow generating CIL code. However, this approach requires knowledge of CIL and internals of CLR, which complicates the process of software development and maintenance. In .NET, it is also possible to generate code during the execution of the application using a declarative approach by utilizing the expression mechanism. Expressions in .NET are separate blocks, each of which describes an operation on data. Such blocks are combined into expression trees, which allows describing complex operation. Expression tree can be formed during the execution of the application and compiled, resulting in a lambda function that performs the operations described by the expressions. However, this approach does not provide an opportunity to control the code generation process. In addition to CIL generation, .NET has mechanisms for generating source code in various languages. The article deals with the generation of C# programming language code. One approach is to use classes in the System.CodeDom namespace, which allows source code to be generated and compiled at runtime. However, the compilation relies on the use of the .NET Framework compiler, which makes it impossible to fully use this approach in new versions of .NET. The approach using T4 allows the generation of code based on templates before compiling the application, which provides an opportunity to detect syntax errors in the generated code at the compilation stage. However, T4 technology is part of the Visual Studio IDE, which makes it impossible to use it outside of that IDE. An alternative to T4 is the Source Generators, which is part of the .NET Compiler Platform SDK. Since Source Generators are part of the .NET compiler, the code generation process is a component of the application compilation process that provides access to a large amount of metadata. However, using Source Generators is only possible in newer versions of .NET. The correct combination of approaches to metaprogramming will allow creating effective tools for optimizing the development process of .NET applications.

Keywords: metaprogramming, .NET, C#, reflection, code generation, Reflection.Emit, Expression, System.CodeDom, T4, Source Generators, Roslyn.

Постановка проблеми

Сучасне програмне забезпечення часто складається з багатьох модулів та містить велику кількість функціоналу. Для автоматизації та спрощення розробки й підтримки програмного забезпечення використовують підходи метапрограмування [1].

У сфері розробки програмного забезпечення метапрограмування є одним із головних та важливих напрямків. Метапрограмування дозволяє як генерувати код під час розробки та компіляції, так і розробляти програмне забезпечення, що здатне змінювати та адаптувати свою поведінку під час виконання [2].

Таким чином, актуальним є питання вибору підходу метапрограмування, щоб забезпечити ефективне використання методів метапрограмування в процесі розробки масштабних програмних проєктів.

Аналіз останніх джерел

Source Generators [3] – інструмент метапрограмування, що є частиною .NET Compiler Platform SDK. Source Generators дозволяє аналізувати та генерувати новий код мови програмування C# під час компіляції. Згенерований код додається до черги компіляції та компілюється разом з іншим кодом програми.

У статті [4] розглянуто результати використання Source Generators у бібліотеці System.Text.Json. Ця бібліотека використовується для серіалізації та десеріалізації об’єктів у формат Json. Автори замінили використання рефлексії (System.Reflection) на Source Generators та провели порівняння швидкодії двох рішень. За результатами порівнянь рішення, що використовує Source Generators, виявилось у середньому на 40-50% більш ефективним з точки зору швидкодії та використання пам’яті, ніж рішення з використанням System.Reflection.

Метою роботи є аналіз підходів метапрограмування на платформі .NET та мові програмування C#, встановлення їх переваг та недоліків, що дозволить забезпечити ефективне використання розглянутих підходів для оптимізації процесу розробки .NET додатків.

Виклад основного матеріалу

Платформа .NET дозволяє використовувати низку підходів метапрограмування. У рамках дослідження важливо розглянути їх і виявити переваги та недоліки кожного. Це дозволить більш ефективно використовувати ті чи інші підходи та їх комбінації під час розробки додатків на .NET.

Найпоширенішим підходом до метапрограмування в .NET є рефлексія (System.Reflection) [1-2]. Рефлексія – це набір інструментів у платформі .NET для роботи з метаданими додатку. Механізм рефлексії має доступ до метаданих додатку та його коду під час виконання самого додатку, що дозволяє отримувати інформацію про класи, їх методи та атрибути, зчитувати та змінювати значення полів класу, викликати методи динамічно й змінювати певні властивості додатку. Це є можливим за рахунок того, що .NET додатки компілюються в Common Intermediate Language (CIL) та зберігаються разом з метаданими в Portable Executable.

Ієрархія метаданих у System.Reflection наведена на рис.1.

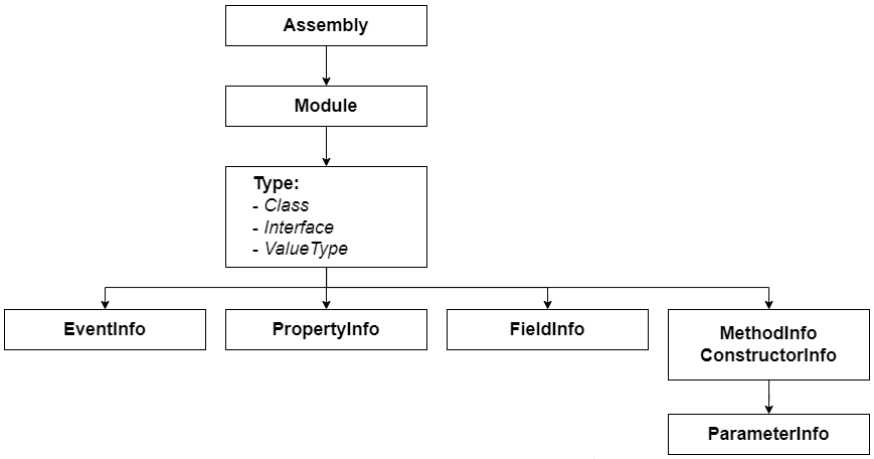

Рис. 1. Ієрархія метаданих у System.Reflection

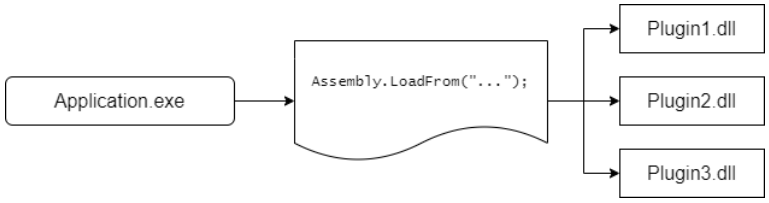

Рис. 2. Схема розширюваного додатку з використанням рефлексії

Зі схеми ієрархії метаданих (рис. 1) випливає, що рефлексія в .NET надає доступ до метаданих усіх рівнів додатку, починаючи з інформації про збірки (Assembly) і закінчуючи інформацією про аргументи

конструкторів та методів (ParameterInfo).

Доступ до метаданих на рівні збірок дозволяє створювати розширювані додатки, що є одним з типових сценаріїв використання рефлексії [2]. Наприклад, додаток може динамічно підключати плагіни, що реалізовані та зібрані у .dll файли (рис. 2).

Іншими типовими сценаріями використання рефлексії є динамічні виклики методів та конструкторів, зчитування та присвоєння значень полів і властивостей (рис. 3). Такий підхід використовують при розробці певного загального функціоналу, що повинен взаємодіяти з великою кількістю класів [1-2].

```
public static string ToStringExternal(this object obj) =>
    string.Join(
        "; ",
        obj.GetType()
           .GetProperties(BindingFlags.Instance | BindingFlags.Public)
           .Select(x => $"{x.Name}: {x.GetValue(obj)}"));
```

Рис. 3. Приклад використання рефлексії в .NET

Окрім вбудованих метаданих, .NET надає можливість створювати власні метадані. Для цього використовуються так звані атрибути (Attribute). Атрибути являють собою анотації, які можна призначити до певних елементів коду. Метадані, що надаються атрибутами, можуть бути зчитані під час виконання додатку за допомогою рефлексії для контролю поведінки тих чи інших модулів додатку (рис. 4). Сторонні інструменти, такі як компілятор чи статичний аналізатор коду, також можуть використовувати атрибути під час компіляції для того, щоб контролювати процеси взаємодії з кодом [5].

```
public class MyClass
{
    [Serializable]
    public string Name { get; set; }
    [Serializable]
    public int Age { get; set; }
    ...
}

public static class Extensions
{
    public static string Serialize(this object obj)
    {
        var propertiesToWrite = obj.GetType()
            .GetProperties()
            .Where(x => x.GetCustomAttribute<SerializableAttribute>() != null);

        return string.Join("; ", propertiesToWrite.Select(x => $"{x.Name}: {x.GetValue(obj)}"));
    }
}
```

Рис. 4. Використання атрибутів для керування логікою роботи додатку

Можливість створювати власні метадані за допомогою атрибутів дозволяє використовувати рефлексію в більшій кількості задач метапрограмування, адже так можна доволі просто контролювати поведінку модулів, що використовують рефлексію.

Варто зазначити, що сама собою рефлексія є лише набором API в .NET для роботи з метаданими та структурою додатку, що не опирається на генерацію коду. З цього випливає один із найбільших недоліків рефлексії, а саме доволі низькі показники швидкодії. Наприклад, у звичайному сценарії, коли необхідно отримати значення з властивості об'єкту, використання рефлексії є повільнішим у 923 рази [6] у порівнянні зі звичайним кодом. Хоча мова йде про наносекунди, проте використання рефлексії в широких масштабах може призвести до значного уповільнення роботи додатку.

Платформа .NET має вбудований механізм генерації CIL коду в процесі виконання додатку [2]. Усе необхідне для цього API подане в просторі імен System.Reflecion.Emit. Згенерований код може бути динамічно доданий до поточного додатку, використовуючи рефлексію, або збережений у вигляді Portable Executable (PE) файлу. Розглянемо приклад генерації CIL коду з використанням Reflection.Emit (рис. 5).

Приклад демонструє створення класу, що містить 2 поля типу int та конструктор, що приймає в якості аргументів 2 значення типу int. За допомогою AssemblyBuilder.DefineDynamicAssembly реалізується нова збірка, в якій далі створюється модуль, що містить новий тип даних.

З прикладу видно, що Reflection.Emit дозволяє оголошувати не тільки поля та властивості класів, а й методи та конструктори. Ще однією корисною особливість є можливість створення лямбда функцій у процесі виконання додатку (DynamicMethod). Для генерації коду методу, конструктора чи функції використовується клас IlGenerator, який дозволяє описувати порядок виконання команд CIL. Робота на такому низькому рівні має як переваги, так і недоліки. До переваг варто віднести можливість доступу до внутрішнього функціоналу Common Language Runtime (CLR) та доволі високий рівень варіативності. Ключовим недоліком такого підходу є необхідність працювати практично з іншою низькорівневою мовою програмування, що може значно ускладнювати процес розробки та підтримки програмного забезпечення.

```
var assemblyName = new AssemblyName("DynamicAssemblyExample");
var ab = AssemblyBuilder.DefineDynamicAssembly(
    assemblyName,
    AssemblyBuilderAccess.Run);

var mb = ab.DefineDynamicModule(assemblyName.Name);
var tb = mb.DefineType("ExampleType", TypeAttributes.Public);

var n1Field = tb.DefineField("_number1", typeof(int),  FieldAttributes.Private);
var n2Field = tb.DefineField("_number2", typeof(int),  FieldAttributes.Private);

var ctorParameters = new[] { typeof(int), typeof(int) };
ConstructorBuilder ctor1 = tb.DefineConstructor(
    MethodAttributes.Public,
    CallingConventions.Standard,
    ctorParameters);

var ilGen1 = ctor1.GetILGenerator();

ilGen1.Emit(OpCodes.Ldarg_0);
ilGen1.Emit(OpCodes.Ldarg_1);
ilGen1.Emit(OpCodes.Stfld, n1Field);
ilGen1.Emit(OpCodes.Ldarg_0);
ilGen1.Emit(OpCodes.Ldarg_2);
ilGen1.Emit(OpCodes.Stfld, n2Field);
ilGen1.Emit(OpCodes.Ret);

var type = tb.CreateType();
```

Рис. 5. Створення класу з використанням Reflection.Emit

Окрім Reflection.Emit, у .NET існує можливість генерації CIL коду без необхідності роботи з CLR на низькому рівні. Цього можна досягти за рахунок використання виразів (Expression). Самі собою вирази є описами простих операцій над даними, наприклад, порівняння, додавання, конкатенація рядків тощо. Вирази можна об'єднувати у складені структури, таким чином формуючи дерево виразів (Expression Tree), що дозволяє описувати комплексні операції над даними. Враховуючи розглянуті особливості, можна зазначити, що вирази дозволяють використовувати декларативний підхід до метапрограмування в .NET [2], що може значно спросити процес розробки. Розглянемо роботу з виразами на прикладі (рис. 6).

```
static void Main(string[] args)
{
    var addFunc = CalcExpression<decimal>("+").Compile();
    var res = addFunc(5, 5);
}

1 reference
static Expression<Func<T,T,T>> CalcExpression<T>(string operation)
{
    var op1 = Expression.Variable(typeof(T), "operand1");
    var op2 = Expression.Variable(typeof(T), "operand2");

    switch (operation)
    {
        case "+":
            return Expression.Lambda<Func<T, T, T>>(Expression.Add(op1, op2), op1, op2);
        case "-":
            return Expression.Lambda<Func<T, T, T>>(Expression.Subtract(op1, op2), op1, op2);
        case "*":
            return Expression.Lambda<Func<T, T, T>>(Expression.Multiply(op1, op2), op1, op2);
        case "/":
            return Expression.Lambda<Func<T, T, T>>(Expression.Divide(op1, op2), op1, op2);
        default:
            throw new NotSupportedException();
    }
}
```

Рис. 6. Використання виразів для побудови динамічних операцій

Результатом роботи методу CalcExpression є дерево виразів, яке виконує математичну операцію над даними залежно від переданого в метод параметру. Саме дерево складається з блоків виразів, кожен з яких описує певну операцію (створення змінної, додавання, віднімання тощо). В .NET вже реалізована низка готових блоків виразів, що дозволяють виконувати більшість типових задач. Окрім цього, MethodCallExpression може викликати сторонні методи, що дозволяє реалізовувати свої блоки виразів.

Для того, щоб вираз можна було перетворити у лямбда функцію, його потрібно обернути в LambdaExpression, при цьому необхідно явно задати тип делегату функції. Виклик методу Compile об'єкту

LambdaExpression скомпілює дерево виразів у лямбда функцію та поверне її делегат.

Таким чином, вирази дозволяють виконувати більшість задач метапрограмування, використовуючи декларативний підхід. Варто зазначити, що вирази мають певні обмеження [7], що не дозволить використовувати їх для низки задач.

Окрім генерації CIL, .NET надає інструменти для генерації звичайного вихідного коду. Відомо, що .NET не прив'язана до однієї мови програмування, але основною мовою програмування під платформу .NET є C# [1], тому наступні приклади розглядатимуть генерацію саме C# коду. Один із таких підходів опирається на використання System.CodeDom, що містить набір класів для роботи з елементами та структурою файлів коду. Так API дозволяє генерувати вихідний код у процесі роботи додатку та компілювати його. При цьому є можливість генерації не лише C# коду, а й інших мов, що підтримують .NET. Ключовими перевагами підходу є робота з C# кодом напряму та можливість отримати код в якості вихідних даних. До недоліків тут слід віднести те, що System.CodeDom опирається на використання .NET Framework компілятора, що робить неможливим компіляцію згенерованого коду при роботі з новими версіями .NET (.NET Core та .NET 5 і вище). При цьому можливість генерації коду за допомогою класів System.CodeDom все ще залишається.

Окрім генерації коду в процесі виконання додатку, часто використовуються підходи, що опираються на генерацію на етапі компіляції або перед нею. Використання таких підходів дозволяє автоматизувати написання певних повторюваних частин програмного коду, що може скоротити час, потрібний для розробки. Основним підходом у .NET, що дозволяє отримувати вихідний код до компіляції додатку, є підхід з використання Text Template Transformation Toolkit (T4) [2]. T4 – це інструмент для генерації тексту на основі шаблонів. T4 можна використовувати як для генерації звичайних текстових файлів, так і файлів, що містять програмний код. Важливою особливістю T4 є те, що шаблони можуть містити певну логіку, наприклад, цикли або операції порівняння, що дозволяє генерацію складних структур. Ключовими недоліками T4 є прив'язка до IDE Visual Studio та відсутність повної підтримки нових версій .NET [8].

В останніх версіях .NET все більше уваги приділяється швидкодії та незалежності платформи від сторонніх інструментів. Для цього в .NET SDK з'являється все більше інструментів, що дозволяють досягти поставлених цілей. Одним із таких інструментів є Source Generators. Він виступає в якості альтернативи T4 у задачах генерації коду на етапі компіляції [8]. Source Generators є частиною .NET Complier Platform SDK (Roslyn), що робить його використання незалежним від середовища розробки. У такому випадку генерація коду є складовою процесу компіляції (рис. 7), що дозволяє виявляти помилки у згенерованому коді ще на етапі компіляції.

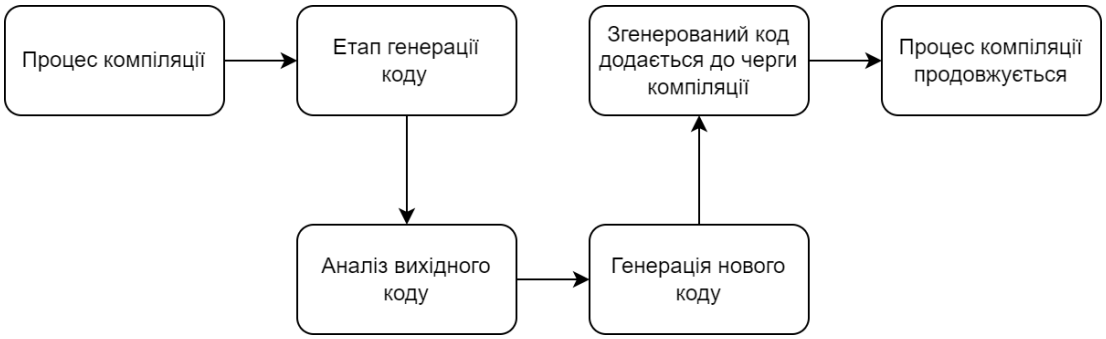

Рис. 7. Схема роботи Source Generators

Окрім генерації певних повторюваних частин коду, Source Generators можна використовувати в якості альтернативи підходам, що опираються на рефлексію або генерацію в процесі виконання програми. Наприклад, використання Source Generators для заміни рефлексії може значно підвищити швидкодію додатку [4]. Крім цього, генерація коду на етапі компіляції дозволить використання Native AOT компіляції. Такий підхід до компіляції опирається на генерацію відразу байт коду, а не CIL. Так можна уникнути використання JIT у процесі роботи додатку, що зменшує використання пам'яті та підвищує швидкодію додатку.

Результати порівняння підходів до метапрограмування в .NET зведено в таблицю 1.

Висновки

Для створення ефективних інструментів оптимізації процесу розробки .NET додатків з використанням метапрограмування необхідно сфокусуватися на комбінації різних підходів до метапрограмування з урахуванням їх переваг та недоліків.

Важливо розуміти специфіку сфери, оптимізація якої буде виконуватися за допомогою метапрограмування, оскільки певні задачі можуть потребувати змін у додатку в процесі виконання. У такому випадку варто розглядати поєднання підходів з використанням рефлексії та виразів. Для більш загальних випадків доцільно буде використовувати Source Generators з елементами виразів та рефлексії. Такий підхід дозволить написання загального функціоналу без необхідності додаткових операцій у процесі виконання додатку, що позитивно відобразиться на його швидкодії. Також це дозволить виявляти помилки згенерованого коду на етапі компіляції.

Таким чином, коректне поєднання підходів до метапрограмування дозволить створити ефективні інструменти для оптимізації процесу розробки .NET додатків.

Таблиця 1

Порівняння підходів до метапрограмування в .NET

Підхід	Підтримка нових версій .NET	Зміна програми під час виконання	Виявлення помилок під час компіляції	Вбудований в .NET	Сумісність з Native AOT
Рефлексія	Так	Так	Ні	Так	Частково
Reflection.Emit	Так	Так	Ні	Так	Ні
Expression	Так	Так	Так	Так	Частково
CodeDom	Частково	Так	Ні	Частково	Ні
T4	Частково	Ні	Так	Ні	Так
Source Generators	Так	Ні	Так	Так	Так

Література

1. Ingebrigtsen E. Metaprogramming in C#: Automate your .NET development and simplify overcomplicated code / Einar Ingebrigtsen. – Birmingham, 2023. – 352 с. – (Packt Publishing).

2. Hazzard K. Metaprogramming in .NET / K. Hazzard, J. Brock. – New York, 2013. – 360 с. – (Manning).

3. Source Generators [Електронний ресурс]. – Режим доступу до ресурсу: https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/source-generators-overview.

4. System.Text.Json Source Generators [Електронний ресурс] – Режим доступу до ресурсу: https://devblogs.microsoft.com/dotnet/try-the-new-system-text-json-source-generator.

5. Attribute Class [Електронний ресурс] – Режим доступу до ресурсу: https://learn.microsoft.com/en-us/dotnet/api/system.attribute?view=net-7.0.

6. Warren M. Why is reflection slow? [Електронний ресурс] / Matt Warren. – 2016. – Режим доступу до ресурсу: https://mattwarren.org/2016/12/14/Why-is-Reflection-slow/.

7. Expression trees [Електронний ресурс] – Режим доступу до ресурсу: https://learn.microsoft.com/en-us/dotnet/csharp/advanced-topics/expression-trees/.

8. Stropek R. C# Source Generators [Електронний ресурс] / Rainer Stropek. – 2022. – Режим доступу до ресурсу: https://devm.io/csharp/csharp-source-generators.

References

1. Ingebrigtsen E. Metaprogramming in C#: Automate your .NET development and simplify overcomplicated code / Einar Ingebrigtsen. – Birmingham, 2023. – 352 c. – (Packt Publishing).

2. Hazzard K. Metaprogramming in .NET / K. Hazzard, J. Brock. – New York, 2013. – 360 c. – (Manning).

3. Source Generators [Elektronnij resurs]. – Rezhim dostupu do resursu: https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/source-generators-overview.

4. System.Text.Json Source Generators [Elektronnij resurs] – Rezhim dostupu do resursu: https://devblogs.microsoft.com/dotnet/try-the-new-system-text-json-source-generator.

5. Attribute Class [Elektronnij resurs] – Rezhim dostupu do resursu: https://learn.microsoft.com/en-us/dotnet/api/system.attribute?view=net-7.0.

6. Warren M. Why is reflection slow? [Elektronnij resurs] / Matt Warren. – 2016. – Rezhim dostupu do resursu: https://mattwarren.org/2016/12/14/Why-is-Reflection-slow/.

7. Expression trees [Elektronnij resurs] – Rezhim dostupu do resursu: https://learn.microsoft.com/en-us/dotnet/csharp/advanced-topics/expression-trees/.

8. Stropek R. C# Source Generators [Elektronnij resurs] / Rainer Stropek. – 2022. – Rezhim dostupu do resursu: https://devm.io/csharp/csharp-source-generators.