УДК 004.624 DOI https://doi.org/10.35546/kntu2078-4481.2025.2.2.13

В. В. ВОЙТКО
кандидат технічних наук, доцент,
доцент кафедри програмного забезпечення
Вінницький національний технічний університет
ORCID: 0000-0002-3329-7256

М. Ю. ПОЗУР
аспірант кафедри програмного забезпечення
Вінницький національний технічний університет
ORCID: 0009-0003-5225-2453

МЕТОД ДИНАМІЧНИХ ВИКЛИКІВ В .NET ШЛЯХОМ ГЕНЕРАЦІЇ ВИХІДНОГО КОДУ НА ЕТАПІ КОМПІЛЯЦІЇ

У статті розглядається метод здійснення динамічних викликів в .NET шляхом генерації коду на етапі компіляції. Проаналізовано традиційний механізм здійснення динамічних викликів у .NET, що базується на використанні рефлексії. Визначено, що ключовим недоліком такого підходу є швидкодія, що обумовлює необхідність пошуку альтернативного рішення для здійснення динамічних викликів. Проведено огляд останніх публікацій на тему оптимізації динамічних викликів у .NET. У результаті аналізу визначено, що низка існуючих рішень описує лише сам процес здійснення виклику, але не розглядає зберігання згенерованих функцій та структуру, яка б дозволила забезпечити їх повторне використання. Розглянуто рішення, що опирається на генерацію метаданих на етапі компіляції та асоціацію цих метаданих з відповідними статичними викликами. Запропоновано рішення, що також базується на генерації метаданих під час компіляції та їх асоціації зі статичними викликами. Перевагою запропонованого рішення над існуючими є можливість здійснення динамічних викликів для об'єктів, які приведені до одного з базових типів, що розширює перелік сценаріїв застосування. Досягнути цього вдалося за рахунок використання структури метаданих, яка є подібною до рефлексії. Генерація метаданих реалізована з використанням Incremental Source Generators, що дозволяє забезпечити ефективну генерацію коду. Метод передбачає використання комбінації з двох генераторів коду, один з яких відповідає за генерацію самих метаданих, а інший – за генерацію коду об'єднання цих метаданих в одну структуру. Для визначення типів даних, для яких необхідно виконувати генерацію, запропоновано використовувати як атрибути метаданих, так і спеціально визначені методи, що в якості аргументів приймають тип даних. Це дозволило використання запропонованого методу як для типів даних, визначених у контексті поточного синтаксичного дерева, так і для типів даних, що належать до зовнішніх бібліотек. Швидкодію розробленого методу було перевірено з використанням бібліотеки Benchmark.NET, у результаті чого встановлено, що запропонований метод має вищу швидкодію в порівнянні з рефлексією.

Ключові слова: *метапрограмування, .NET, рефлексія, генерація коду, Source Generators.*

V. V. VOITKO
Candidate of Technical Sciences, Associate Professor,
Associate Professor at the Department of Software Engineering
Vinnytsia National Technical University
ORCID: 0000-0002-3329-7256

M. YU. POZUR
Postgraduate Student at the Department of Software Engineering
Vinnytsia National Technical University
ORCID: 0009-0003-5225-2453

METHOD OF EXECUTING DYNAMIC CALLS IN .NET USING SOURCE CODE GENERATION

The article discusses the development of a method for executing dynamic calls in .NET by generating code at the compilation. The traditional mechanism for making dynamic calls in .NET, based on the use of reflection, is analyzed. It is determined that the key disadvantage of such an approach is its performance, which necessitates the search for an alternative solution for making dynamic calls. A review of recent publications on the topic of optimizing dynamic calls in .NET is conducted. As a result of the analysis, it is determined that a number of existing solutions describe only the process of making a call, but do not consider the storage of generated functions and a structure that would allow for their reuse. A solution based on the generation of metadata at the compilation stage and the association of such metadata with corresponding static calls is considered. A solution is proposed that is also based on the generation

of the metadata during compilation and their association with static calls. The advantage of the proposed solution over the existing one is the ability to make dynamic calls for objects cast to one of the basic types, which expands the list of scenarios for the proposed method's application. This was achieved by using a metadata structure that is similar to the reflection. Metadata generation is implemented with Incremental Source Generators, which allows for efficient code generation. The method involves using a combination of two code generators, one of which is responsible for generating the metadata itself, and the other for generating code that combines such metadata into a single structure. To determine data types for which generation is required, it is proposed to use both metadata attributes and special methods that accept the data type as an argument. This allowed for the proposed method to be used for data types defined in the context of the current syntax tree and data types belonging to external libraries. The performance of the developed method was tested using the Benchmark.NET library, as a result of which it was found that the proposed method has better performance compared to reflection.

Key words: *metaprogramming, .NET, reflection, code generation, Source Generators.*

Постановка проблеми

Сучасне програмне забезпечення характеризується високим рівнем складності та масштабністю, що, у свою чергу, зумовлює зростання витрат ресурсів і часу на його розробку. У зв'язку з цим спостерігається тенденція до впровадження засобів метапрограмування в процесі створення програмних продуктів. До таких засобів належать як інструменти генерації вихідного коду, так і функціональні можливості мов програмування та фреймворків, що забезпечують реалізацію підходів метапрограмування на етапі виконання програмного застосунку [1]. У контексті платформи .NET одним із найпоширеніших інструментів метапрограмування є механізм рефлексії [2], який надає можливість отримання метаданих виконуваного застосунку та здійснення низки операцій з ними. Одним із найпоширеніших сценаріїв застосування рефлексії в .NET є виконання динамічних викликів [3], під якими розуміється виклик методів або доступ до членів типу за умов визначення імені під час виконання (dynamic name resolution), на відміну від статичного визначення на етапі компіляції (static name resolution). Такі виклики значно спрощують розробку узагальненого функціоналу, наприклад, серіалізація та десеріалізація даних. Водночас суттєвим недоліком механізму рефлексії є зниження продуктивності порівняно зі статичними викликами, що зумовлює необхідність пошуку більш ефективних альтернативних підходів. Проте, здійснення динамічних викликів з використанням рефлексії має вагомий недолік – швидкодію. Швидкість виконання такого виклику є значно повільнішою за статичний виклик [4]. Через це існує необхідність у пошуку альтернативних способів здійснення динамічних викликів.

Аналіз останніх досліджень і публікацій

Було проведено аналіз публікацій та розробок, пов'язаних з темою дослідження. В першу чергу було розглянуто низку підходів оптимізації динамічних викликів рефлексії, що опираються на генерацію коду в процесі виконання застосунку [5, 6]. В публікаціях [5, 6] описано використання механізму Expression для динамічної генерації статичних викликів. Такий підхід дозволяє згенерувати лямбда-функцію, що виконує відповідний статичний виклик. Використання такого підходу дозволяє значно покращити швидкодію викликів. Проте, в публікаціях [5, 6] розглядається лише сам процес створення функції та її використання в певному контексті, а не створення загального механізму, що дозволив би здійснювати такі виклики.

Ідею використання генерації коду для «компіляції викликів рефлексії» розглянуто в публікації [7]. Тут описано використання інструменту Source Generators [8] для генерації метаданих властивостей типів даних, що містять посилання на відповідні статичні виклики. Самі метадані генеруються на етапі компіляції. Ключовим недоліком запропонованого підходу є те, що асоціація між типом даних та його метаданими відбувається через генерацію методів розширень, що повертають метадані цього типу. Так як виклик методу розширення відбувається для типу даних, в який приведено об'єкт, то це унеможливлює використання запропонованого підходу для випадків, коли об'єкт приведений до одного з базових типів.

Ідея «компільованої рефлексії» розглядалася не лише в контексті платформи .NET. Наприклад автори публікації [9] розглядають підхід до реалізації функціоналу рефлексії як складової процесу компіляції в мові програмування Kotlin. На відміну від традиційної рефлексії, що працює під час виконання програми, такий підхід опирається на аналіз синтаксичного дерева та генерацію відповідних метаданих та викликів на етапі компіляції застосунку.

Формулювання мети дослідження

Метою дослідження є розробка методу здійснення динамічних викликів для платформи .NET шляхом генерації вихідного коду на етапі компіляції, що дозволить покращити швидкодію розроблюваного програмного забезпечення в сценаріях, які опираються на використання динамічних викликів.

Викладення основного матеріалу дослідження

Так як мета роботи полягає у розробці певного механізму, що б дозволив здійснювати динамічні виклики, то варто розглянути рефлексію, яка є основним інструментом для здійснення таких викликів у .NET. Процес здійснення динамічного виклику з використанням рефлексії потребує об'єкт, для якого відбуватиметься такий виклик.

Далі, використовуючи API рефлексії, отримуються метадані типу даних (*System.Type*), з яких можна отримати інформацію про всіх членів відповідного типу. Сам динамічний виклик здійснюється з метаданих відповідного члену. Ключова проблема швидкодії рефлексії полягає в останньому кроці. У публікації [4] розглядається здійснення такого виклику на рівні середовища виконання платформи .NET (Common Language Runtime або CLR). Виконання такого виклику рефлексії потребує великої кількості операцій на рівні CLR, що призводить до нижчої, у порівняні зі статичними викликами, швидкодії.

Отже, якщо об'єкт метаданих члену асоціювати з відповідним статичним викликом, то можна досягти значно кращих показників швидкодії. Саме це й вдалося зробити автору публікації [7]. Проте ключовим недоліком такого підходу є використання статичного зв'язку між типом даних та його метаданими. Для уникнення цього варто використовувати структуру метаданих, аналогічну до рефлексії (рис. 1).

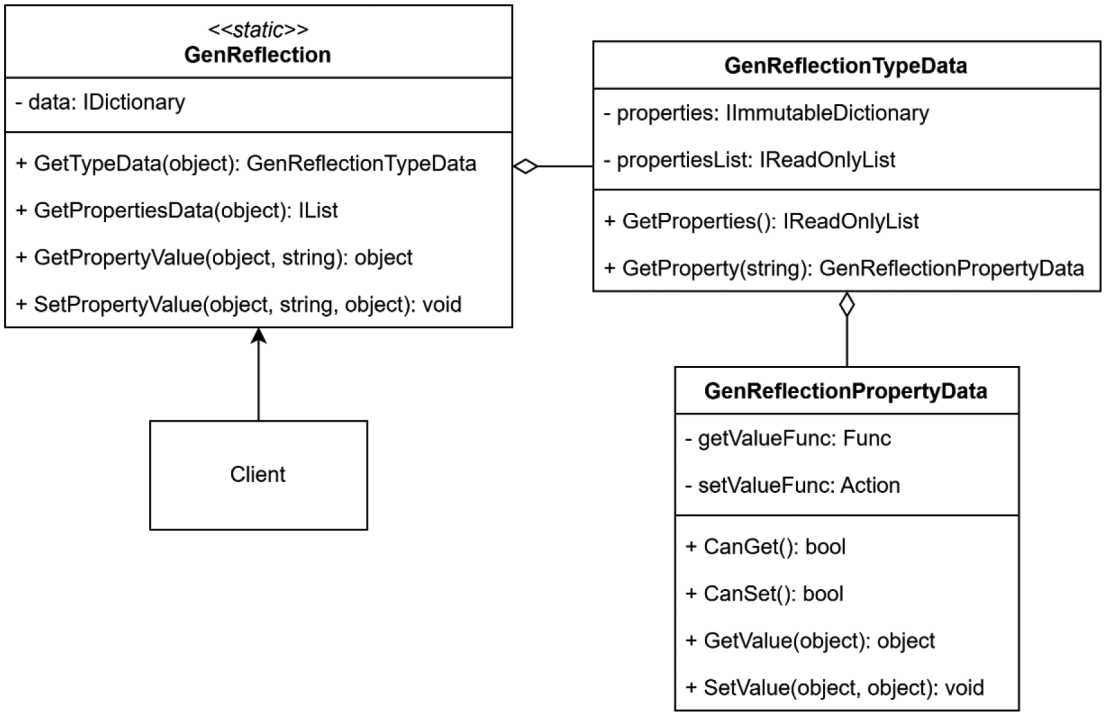

Рис. 1. UML-діаграма розроблюваної структури метаданих

Визначимо основні елементи такої структури:

– статичний клас, що зберігає колекцію метаданих типів та надає доступ до них;

– клас, що відповідає за метадані типу даних;

– клас, що відповідає за метадані властивості типу даних та містить посилання на відповідні статичні виклики.

Така структура дозволить виконувати динамічні виклики аналогічно до рефлексії, але з вищою швидкодією за рахунок наявності скомпільованого коду статичного виклику для членів типів. Проте такий підхід все ще потребує використання рефлексії для отримання типу даних об'єкту. Цей тип даних асоціюватиметься з метаданими типу розробленої структури через використання хеш-таблиць, щоб досягти найкращої швидкодії пошуку.

Варто зазначити, що розроблена модель розглядається в контексті динамічних викликів для властивостей класів (зчитування та присвоєння їх значень). Це обумовлено тим, що зчитування та присвоєння значень властивостей є одним із найпоширеніших сценаріїв використання динамічних викликів. З моделі видно, що метадані властивості асоціюються з відповідними статичними викликами.

Таким чином, генерація коду має забезпечити генерацію метаданих з відповідними статичними викликами так, щоб згенеровані метадані були сформовані у відповідну структуру (рис. 1). Для генерації коду було обрано Roslyn Source Generators [8], що є складовою .NET Complier Platform SDK. Ключовою перевагою цього інструменту є його інтеграція з компілятором Roslyn, що дозволяє отримувати доступ до всього синтаксичного дерева на рівні аналізатора коду. Одним із недоліків Source Generators є те, що генерація відбувається в контексті всього синтаксичного дерева, таким чином, при будь-яких змінах коду відбуватиметься повторна генерація. Для уникнення цієї проблеми потрібно використовувати Incremental Source Generators [10]. Головна перевага такого підходу полягає у використанні конвеєра перетворень, який дозволяє фільтрувати та трансформувати дані перед генерацією. Це дозволяє виконувати генерацію коду в контексті певної частини синтаксичного дерева. Окрім

цього, такий конвеєр перетворень використовує кешування, щоб уникнути генерацій для тих елементів синтаксичного дерева, в яких не відбулися зміни.

Для уникнення повторної генерації всього коду доцільно генерувати окремий файл для кожного типу даних. Для забезпечення визначеної структури (рис. 1), метадані потрібно генерувати таким чином, щоб їх можна було додати до однієї хеш-таблиці. Для цього можна генерувати часткові (partial) визначення статичного класу, кожне з яких міститиме змінну, що відповідатиме метаданим відповідного типу. Для формування єдиної структури даних потрібно визначити другий генератор, що генеруватиме код додавання всіх згенерованих змінних до хеш-таблиці. На рисунку 2 зображено загальну модель генерації коду.

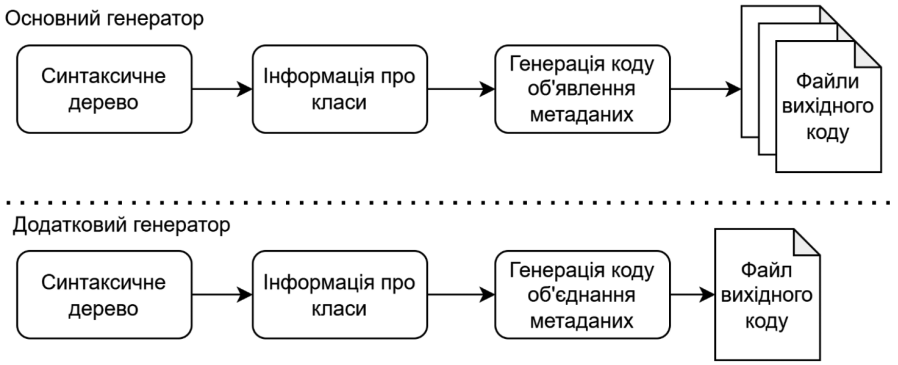

Рис. 2. Загальна модель генерації коду

Варто зазначити, що генерація часткових об’явлень статичного класу, визначеного в моделі (рис. 1), не є обов’язковою. Альтернативно можна генерувати об’явлення іншого типу даних або методів розширень, але разом із цим статичний клас, що містить метадані та надає доступ до структури, має забезпечити можливість додавання метаданих до структури. Проте в такому випадку також є необхідність визначення другого генератора, що генеруватиме код додавання метаданих до хеш-таблиці.

Наступна проблема полягає у виборі типів даних, для яких необхідно генерувати код. Для цього доцільно використовувати атрибути метаданих у комбінації зі спеціально визначеними методами, що в якості аргумента приймають тип даних. Це дозволить відмічати як типи даних, визначені в контексті поточного синтаксичного дерева, так і типи, що є частиною зовнішніх залежностей.

Сама генерація коду полягатиме в генерації об’явлення статичної змінної, якій відразу присвоєне значення об’єкту, що відповідає за метадані типу даних (рис. 3). Разом із цим генеруються також метадані властивостей цього типу даних з відповідними лямбда-функціями, що відповідають за статичні виклики.

```
public static GenReflectionTypeData _data_mytype1 = new GenReflectionTypeData(
    new Dictionary<string, GenReflectionPropertyData>
    {

        {
            "CreatedDt",
            new GenReflectionPropertyData(
                "CreatedDt",
                (x) => ((global::SourceGenTest.Data.MyType1)x).CreatedDt,
                (x, v) => ((global::SourceGenTest.Data.MyType1)x).CreatedDt = (global::System.DateTime)v
            )
        },
```

Рис. 3. Приклад згенерованих метаданих

Щодо додаткового генератора коду, то він має генерувати код додавання всіх попередньо згенерованих змінних до розробленої системи метаданих (рис. 4). В якості вхідних даних він приймає той самий набір типів даних, що й основний генератор. Сам код можна генерувати в контексті статичного конструктора, який завжди виконується при запуску додатку.

Таким чином, для реалізації запропонованого методу динамічних викликів потрібно:

1. Визначити власну структуру метаданих, що складається з метаданих членів типу, які містять посилання на відповідні статичні виклики, метаданих типів та статичного класу, що надає доступ до цих метаданих.
2. Визначити спеціальні атрибути метаданих та методи, що приймають в якості аргументу тип даних для відмічення класів, для яких потрібно виконувати генерацію.

```
static GenReflection()
{
    _reflectionData.TryAdd(typeof(global::SourceGenTest.Data.MyType1), _data_mytype1);
    _reflectionData.TryAdd(typeof(global::SourceGenTest.Data.BrandDto), _data_branddto);
    _reflectionData.TryAdd(typeof(global::SourceGenTest.Data.ArticleDto), _data_articledto);
    _reflectionData.TryAdd(typeof(global::SourceGenTest.Data.MyType2), _data_mytype2);

}
```

Рис. 4. Приклад коду, згенерованого додатковим генератором

3. Визначити генератор коду, який для кожного класу, відміченого відповідним атрибутом метаданих або викликом методу, генерує код об'явлення статичної змінної, якій присвоюється значення об'єкта, що відповідає метаданим цього класу.

4. Визначити генератор коду, який для всіх класів, відмічених відповідним атрибутом метаданих або викликом методу, генерує код додавання відповідних змінних, згенерованих на кроці 4, до визначеної на кроці 1 структури метаданих.

5. Для здійснення динамічних викликів потрібно використовувати розроблену систему метаданих.

Для перевірки швидкодії розробленого методу було використано бібліотеку Benchmark.NET [11]. Порівняння відбувалося як для зчитування значення властивості, так і для присвоєння. Оскільки процес отримання метаданих є частиною здійснення динамічного виклику, то його було включено до порівняння. Результати порівняння швидкодії розробленого методу динамічного виклику та швидкодії динамічного виклику з використанням рефлексії наведено у таблиці 1.

Таблиця 1

Порівняльний аналіз здійснення динамічних викликів присвоєння та зчитування значень властивості з використанням розробленого методу та рефлексії

Method	Mean	Error	StdDev	Ratio
GenReflectionGet	11.96 ns	0.499 ns	0.467 ns	0.31
ReflectionGet	38.50 ns	1.182 ns	1.106 ns	1.00
GenReflectionSet	13.42 ns	0.271 ns	0.161 ns	0.32
ReflectionSet	42.21 ns	0.279 ns	0.146 ns	1.00

Результати порівняння показують, що запропонований метод здійснення динамічних викликів є приблизно на 70 % швидшим у порівняні з використанням рефлексії.

Висновки

Розроблено метод здійснення динамічних викликів у .NET за рахунок генерації коду на етапі компіляції. Запропонований метод опирається на використання окремої системи метаданих, де метадані кожного члену типу даних асоціюються з відповідними статичними викликами. Для генерації об'явлення таких метаданих використовуються Incremental Source Generators. Швидкодія розробленого методу є значно вищою за швидкодію рефлексії в сценаріях здійснення динамічних викликів зчитування та присвоєння значень властивостей. Недоліком такого методу є те, що генерація відбувається на етапі компіляції, що унеможливлює його використання для динамічно доданих та анонімних типів даних.

Список використаної літератури

1. Bock J., Hazzard K. Metaprogramming In.NET. Manning Publications Co. LLC, 2012.

2. Ingebrigtsen E. Metaprogramming in C#: Automate Your.NET Development and Simplify Overcomplicated Code. Packt Publishing, Limited, 2023.

3. Beaumont A., Bakhtiari Bastaki B. An Investigation into the Prevalence of Reflection Techniques in Distributed Microsoft.Net NuGet Artefacts. *ICSCA 2022: 2022 11th International Conference on Software and Computer Applications*, м. Melaka Malaysia. New York, NY, USA, 2022. URL: https://doi.org/10.1145/3524304.3524329

4. Warren M. Why is reflection slow?. URL: https://mattwarren.org/2016/12/14/Why-is-Reflection-slow/ (дата звернення: 17.05.2025).

5. Haytam Z. C# Expression Trees: Property Getters. URL: https://blog.zhaytam.com/2020/11/17/expression-trees-property-getter/ (дата звернення: 17.05.2025).

6. Ricardo Peres. Using Generated Methods Instead of Reflection. URL: https://weblogs.asp.net/ricardoperes/using-generated-methods-instead-of-reflection (дата звернення: 17.05.2025).

7. Silin S. Reflection via source generators. URL: https://dev.to/byme8/aot-reflection-4ijb (дата звернення: 17.05.2025).

8. Microsoft. Introducing C# Source Generators. URL: https://devblogs.microsoft.com/dotnet/introducing-c-source-generators/ (дата звернення: 17.05.2025).

9. Reflekt: a Library for Compile-Time Reflection in Kotlin / A. Birillo та ін. *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, м. Pittsburgh, PA, USA, 22–24 трав. 2022 р. 2022. URL: https://doi.org/10.1109/icse-seip55303.2022.9793932

10. Gerr P. Incremental Roslyn Source Generators In.NET 6. URL: https://www.thinktecture.com/net/roslyn-source-generators-introduction/ (дата звернення: 17.05.2025).

11. BenchmarkDotNet. URL: https://benchmarkdotnet.org/index.html (дата звернення: 17.05.2025).

References

1. Bock, J., & Hazzard, K. (2012). *Metaprogramming In.NET*. Manning Publications Co. LLC.

2. Ingebrigtsen, E. (2023). *Metaprogramming in C#: Automate Your.NET Development and Simplify Overcomplicated Code*. Packt Publishing, Limited.

3. Beaumont, A., & Bakhtiari Bastaki B. (2022). An Investigation into the Prevalence of Reflection Techniques in Distributed Microsoft.Net NuGet Artefacts. In *ICSCA 2022: 2022 11th International Conference on Software and Computer Applications*. ACM. https://doi.org/10.1145/3524304.3524329

4. Warren, M. (2016, December 14). Why is reflection slow? https://mattwarren.org/2016/12/14/Why-is-Reflection-slow/

5. Haytam, Z. (2020, November 17). C# Expression Trees: Property Getters. https://blog.zhaytam.com/2020/11/17/expression-trees-property-getter/

6. Peres, R. (2022, January 31). Using Generated Methods Instead of Reflection. https://weblogs.asp.net/ricardoperes/using-generated-methods-instead-of-reflection

7. Silin, S. (2021, September 14). Reflection via source generators. https://dev.to/byme8/aot-reflection-4ijb

8. Microsoft. (2020, April 29). Introducing C# Source Generators. https://devblogs.microsoft.com/dotnet/introducing-c-source-generators/

9. Birillo, A., Lyulina, E., Malysheva, M., Tankov, V., & Bryksin, T. (2022). Reflekt: a Library for Compile-Time Reflection in Kotlin. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE. https://doi.org/10.1109/icse-seip55303.2022.9793932

10. Gerr, P. (2022, June 24). Incremental Roslyn Source Generators In.NET 6. https://www.thinktecture.com/net/roslyn-source-generators-introduction/

11. *BenchmarkDotNet*. https://benchmarkdotnet.org/index.html