Списание „Математика, компютърни науки и образование“ Том 4, кн. 1, 2021
Journal of Mathematics, Computer Science and Education Vol. 4, No. 1, 2021

DOI: 10.54664/LDTN7661

ТЕХНИКИ И ПОДХОДИ ЗА ГЕНЕРИРАНЕ НА АВТОМАТИЧЕН ПРОГРАМЕН КОД

Даниел Дамянов

TECHNIQUES AND APPROACHES IN AUTOMATIC PROGRAM CODE GENERATION

Daniel Damyanov

Abstract: *Code generation is an important success factor in using models in the software development process. We can efficiently generate code for an operating system or applications based on specific models, thus improving the consistency between the model and its application, as well as saving resources. CodeDOM, T4 and Reflection.Emit provide a mechanism for fast object generation with main classes.*

Keywords: *code generation, CodeDOM, .NET, T4, Reflection.Emit, metaprogramming.*

ВЪВЕДЕНИЕ

Генерирането на код е най-общо казано: писането на програми, които пишат програми [1, с. 3]. Използването на генератор на код има предимства в сравнение с конвенционалното програмиране. Използваният език за моделиране или програмиране става по-разбираем поради факта, че графичните елементи правят езика повече компактен и/или по-изчистен. Ефективността на разработката на софтуер също се повишава. Фактът, че по-малко код трябва да се пише, проверява и тества ръчно означава, че разработчиците могат да бъдат по-ефективни. Метапрограмирането е свързано с опростяването и повторното използване на софтуера. Вместо да зависи строго от езиковите функции, за да намали сложността на кода или да увеличи повторната употреба, метапрограмирането постига тези цели чрез различни библиотеки и техники за писане на код. Както беше споменато по-горе, класическото определение за метапрограма е „компютърна програма, която пише нови компютърни програми“ [1].

ИЗЛОЖЕНИЕ

В представянето на концептуалните основи при такъв тип изграждане на приложения, разгледани по-долу, следва да бъдат добавени и основните термини, използвани при генерирането на код [6]:

Конструктивен модел. Спецификация на системата, която се използва за генериране на код с помощта на автоматичен генератор. Промяната на конструктивен модел променя продукта директно. Езикът за моделиране също се приема като език за програмиране от високо ниво, тъй като в общия случай може да бъде изпълнен отделно.

Описателен модел. Спецификация, която се използва за описание на системата, без всъщност да се използва конструктивно при прилагането на системата. Описанията обикновено са

абстрактни и непълни. Тези модели се използват за ръчно изпълнение или се генерират само като документация, след като системата е разработена.

Тестов модел. Спецификация, която е подходяща за извличане на тестове ръчно или автоматично. Тестовият модел се компилира в изпълним код, който се използва за настройване на записи от тестови данни, като тестов драйвер или като очакван резултат от теста.

Конструктивен тестов модел. Тестов модел, който се превръща в изпълними тестове от генератора на код. За разлика от тях, описателните тестови модели се трансформират в тестове ръчно.

Генериране на код. Дейността по генериране на код от конструктивни модели.

Генератор на код. Програма, която трансформира конструктивен модел на език за програмиране от по-високо ниво в реализация. Генерираният код може да бъде част от продуктовата система или тестовия код.

Сценарий. Съдържа конструктивното управление за генериране на код. Сценариите параметризират генератора на код и по този начин позволяват създаване на код, специфичен за платформата и задачите.

Шаблон. Специална форма на скрипт. Той описва кодови модели, в които по време на генериране се вмъкват конкретни елементи от модела. Обикновено се използва механизъм за подмяна на макрониво.

Основните начини за генериране на краен код с C# и Visual Studio IDE са: **The Text Template Transformation Toolkit (T4), CodeDOM, Reflection.Emit.** Вместо да има „код, който пише код", шаблонът намалява сложността, като ви позволява просто да въведете кода, който искате да генерирате във файл [5, с. 249]. В средата на .NET продуктите, които генерират код, могат да бъдат най-често използваната форма на алгоритмична стандартизация.

Шаблоните T4 предоставят мощен начин за генериране на код по време на проектиране (а понякога и по време на компилиране, ако се настрои Visual Studio по подходящ начин). За да генерира текст, синтаксисът T4 предлага три типа елементи [3, с. 65]: **директиви, текстови блокове и контролни блокове**. Първите типове елементи, които са разгледани, са свързани с директивата T4. **Директивите** контролират механизма за шаблони по време на процеса на генериране, като позволяват да се добавят и параметри, разширението на изходния файл, програмният код на генерирания файл, реферираните проекти и импортирането на външни библиотеки, както и езикът, който ще се използва в крайните файлове. Вторият тип T4 елемент е **текстовият блок**. Текстовите блокове са редове от „суров" (plain) текст, който ще бъде вмъкнат в трансформирания изход. За разлика от директивите и контролните блокове, тези блокове от „суров" текст нямат специални разделители около тях. **Блокове за управление:** контролните блокове са в няколко разновидности. Следващият пример е стандартен контролен блок, който понякога се нарича блок-оператор. **Стандартни блокове за контрол**: извикването на вградените функции на T4 като WriteLine в стандартен контролен блок е добре познато. Стандартните контролни блокове могат да съдържат повече от един оператор в тях, поради което понякога се наричат блокове на оператори. Reflection.Emit поддържа динамичното създаване на нови типове по време на изпълнение. Може да се дефинира ново асембли, което да се изпълнява динамично или да се запише на диск, както може и да дефинира модули и нови типове с методи, които след това може да се извикат [5].

Като част от **Reflection** библиотеката **Reflection.Emit** пространство ни предоставя редица класове, които могат да се използват за изграждането на генератор и класове. Както видяхме, **Reflection.Emit** всъщност осигурява някои класове на Builder, като **AssemblyBuilder, ModuleBuilder, ConstructorBuilder, MethodBuilder, EventBuilder, PropertyBuilder** и други, което дава възможността да се изгради самостоятелен IL (Intermediate Language) динамично, по време на изпълнение. Това пространство на имена предлага следните програмни опции: позволява изграждането на модули и асемблита по време на изпълнение, създава класове и типове и генерира IL, стартира .NET компилатор за изграждане на приложения. Рефлексията (reflection) осигурява следните възможности: дефинира глобални методи и свойства по време на изпълнение, като

използва класа **DynamicMethod**, и ги изпълнява с помощта на делегати. Определя асемблитата по време на изпълнение и след това те може да се стартират и/или да се запишат на диск. Изпълнява ги и след това ги премахва от изпълнение, като позволява **на garbage collector**-а (**GC)** да възстанови ресурсите на паметта. Определя типовете на модулите по време на изпълнение, създава инстанции от тези типове и извиква техните методи. Определя метаинформация за дефинирани модули, които могат да се използват от инструменти като отстраняване на грешки и кодови профили. Основният проблем е, че изучаването на IL не е набор от умения, каквито имат повечето .NET разработчици, нито е такъв, който те биха искали да придобият, дори ако се интересуват от техники за метапрограмиране. Причината е проста: писането на код в IL може лесно да доведе до неправилни реализации и изисква сложен модел на изпълнение на код в .NET, което не е толкова интуитивно, колкото този, който предоставя език на високо ниво.

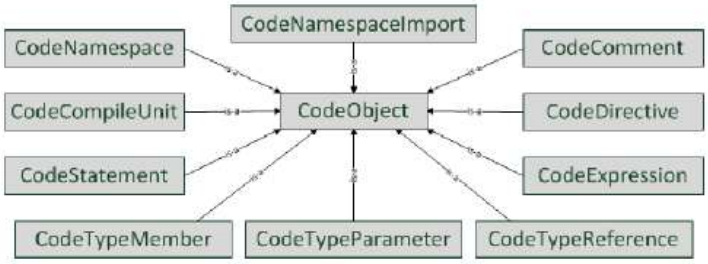
В .NET Framework средата се намира библиотека, наречена **Code Document Object Model** (CodeDOM), която позволява на разработчиците на програми, които искат да генерират изходен код, да генерират кода в множество програмни езици по време на изпълнение въз основа на един модел, който представлява кода за визуализация. За да представят изходния код, елементите на CodeDOM са свързани помежду си, за да образуват структура от данни, известна като граф на CodeDOM, която моделира структурата на изходния код. Пространството от имена System. CodeDom, дефинира типовете, които могат да представляват логическата структура на изходния код, независимо от конкретен програмен език. Пространството от имена, System.CodeDom. Compiler, дефинира типовете за генериране на изходен код от граф на CodeDOM и управление на компилацията на изходния код на поддържаните езици. Моделирането на изходния код, независимо от езика, може да бъде ценно, когато програмата трябва да генерира изходен код за програмен модел на множество езици или за несигурен целеви език. Например някои дизайнери използват CodeDOM като интерфейс за абстракция на определен програмен език, за да създадат изходен код на желания език за програмиране, ако е налична поддръжка от страна на CodeDOM. Програмите на мета ниво анализират, трансформират и генерират програми на ниво обект [4,1]. CodeDOM предоставя класове, които представляват много често срещани типове обекти в програмния код. Библиотеката е доста сложна колекция от класове, които могат да бъдат намерени в пространствата с имена System.CodeDom и System.CodeDom.Compiler [3, с. 102]. Може да се проектира програма, която изгражда модел на изходен код, използвайки елементи на CodeDOM, за да се сглобят всички елементи в един краен работещ код. Той може да бъде изобразен като изходен код, използвайки CodeDOM генератор за поддържан език за програмиране. CodeDOM може да се използва и за компилиране на изходен код в бинарно (двоично) асембли. Най-честите практики за използване на CodeDOM включват:

• Генериране на шаблонен код: генериране на **proxy** код за клиенти на ASP.NET, XML уеб услуги, съветници за кодове, дизайнери или други механизми за създаване на код.

• Динамична компилация: поддържа компилация на код на един или няколко езика.

Expression tree (дърво на израз) ≈ Code Graph [3, 102], като CodeDOM използва така наречените кодови графи за изразяване на кода като данни. Кодовият граф е йерархична структура на данни, която представлява логиката и структурата на алгоритъма. Може да се анализира от текста на изходния код или да се изпълнява стъпка по стъпка. Кодовите графи могат да бъдат превърнати в изпълним IL код, когато е време да се използват. Expression trees, въведени във версията 3.0 на .NET, също могат да представят кода като данни. Те също могат да бъдат изградени програмно или анализирани от изходния код и превърнати в IL за изпълнение. Дървото на израз представя код в дървовидна структура, където всеки връх е израз, например извикване на метод или двуаргументна операция, като x < y. Може да се компилира и стартира код, представен от дървета на изрази. Това дава възможност за динамична модификация на изпълним код, изпълнение на LINQ заявки в различни бази данни и създаване на динамични заявки. Пространството от имена System. CodeDOM е подредено в йерархична колекция от класове. В основата е клас, наречен CodeObject, който предоставя само речник, наречен User-Data. Всеки производен обект в **„namespace“** има

речник, който може да се използва за съхраняване на информация. Интересно е да се отбележи, че типът на речника за свойството UserData е ListDictionary от System.Collections.Specialized namespace, който е ефективен за списъци, които ще съдържат десет или по-малко елементи. При голям обем от елементи производителността драстично намалява. На **Фигура 1. Йерархия в CodeDom object** е показана основната графика на CodeDom обект:

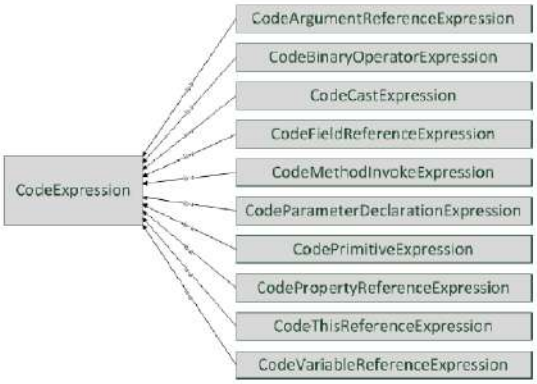


Фигура 1. Йерархия в CodeDom object [3, 102]

Повечето от имената на класовете, както се вижда на **Фигура 1. Йерархия в CodeDom object1**, сами говорят за функцията, която изпълняват. Те описват останалите основни компоненти на програмата: коментари, директиви, типове (интерфейси, класове и структури), членове (методи, полета и свойства), параметри и препратки към други обекти. Целта на **CodeCompileUnit** не е толкова ясно дефинирана. В по-голямата си част се приема, че дейността му е да сглобява всички останали класове в един общ. Класът **CodeExpression** служи като корен за всички типове изрази в **CodeDOM**. **CodeDOM** дефинира обект, наречен **CodeCompileUnit**, който се позовава на **CodeDOM** йерархията, която моделира изходния код за компилиране. **CodeCompileUnit** има свойства за съхраняване на препратки към атрибути, пространства от имена и асемблита. Класовете, които използват **CodeDOM**, произлизат от класа ***CodeDomProvider*** и съдържат методи, които обработват обектния граф, посочен от ***CodeCompileUnit***. За да се създаде обектен граф за обикновено приложение, трябва да се сглоби моделът на изходния код и да се препрати към ***CodeCompileUnit***.

```
CodeCompileUnit compileUnit = new CodeCompileUnit();
```

Фигура 2. Класове за създаване на референции, наследяващи CodeExpression class показва някои от по-често срещаните класове, които се използват в текущата разработка. Елементи на програмата, като аргументи, двоични оператори (добавяне, умножение, равенство и т.н.), операции на класове и извиквания на методи, могат да се използват за изграждане на алгоритъм като данни за един израз в даден момент. Типовете изрази с препратка в техните имена действат като препратки към други типове в CodeDOM.



Фигура 2. Класове за създаване на референции, наследяващи CodeExpression class [3, 104]

Когато се създава обект на CodeDOM, трябва да се посочат изразите, които ще бъдат използвани за описание на съставните части. Нека пример бъде следното изявление, написано на C#: **R = fn (A + B) / C (1).** Един от начините за неговото визуализиране е като набор от вложени или верижни извиквания на функции: **Assign(R, Divide(Invoke(fn, Add(A, B)), C)).** Този вид функционално разделение показва как точно трябва да се напише кодът като граф, като се използва CodeDOM. За да се определи пространство на имена, се създава ***CodeNamespace*** и се задава име за него чрез използване на подходящия конструктор или със задаване на свойството **Name**.

```
CodeNamespace samples = new CodeNamespace(“Samples”)[1];
```

За добавяне на инстанция за импортиране на пространство от имена, в общото пространството от имена, се добавя ***CodeNamespaceImport***, който посочва пространството от имена, което да се импортира в колекцията **CodeNamespace.*Imports***. Следният код добавя импортиране на системното пространство (System) от имена в колекцията Imports към CodeNames:

```
samples.Imports.Add(new CodeNamespaceImport(“System”));
```

Свързването на елементи от кода в обектния граф е етапът, когато всички елементи на кода, които формират графа на CodeDOM, трябва да бъдат свързани с ***CodeCompileUnit*** (който е коренният елемент на дървото) чрез поредица от препратки между елементи, директно посочени от свойствата на коренния обект на графа. Добавя се обект към свойството на контейнерния обект, за да се установи референция към обекта, който ги съдържа. Следващата функция добавя елементите на ***CodeNamespace*** към свойството за събиране на Namespaces на кода ***CodeCompileUnit***:

```
compileUnit.Namespaces.Add(samples);
```

За определянето на тип при деклариране на клас, структура, интерфейс или изброяване с помощта на CodeDOM, се създава нова ***CodeTypeDeclaration*** и му се присвоява име. Следващият пример демонстрира това с помощта на конструктор, за да се зададе име на класа към съответното свойство:

```
CodeTypeDeclaration classProduct = new CodeTypeDeclaration(“Product”);
```

При добавянето на тип в пространството от имена, се добавя ***CodeTypeDeclaration***, който представлява типа, който да се добави към пространството от имена в колекцията Types на ***CodeNamespace***.

Следващият пример демонстрира как да се добави клас с име classProduct към CodeNamespace:

```
samples.Types.Add(classProduct);
```

Добавянето на членове на клас към пространството от имена System.CodeDom предоставя различни елементи, които могат да бъдат използвани за представяне на членове на класа. Всеки член на класа може да бъде добавен към колекцията за членове на ***CodeTypeDeclaration***. При писане на код за изпълнима програма е необходимо да се посочи входната точка на програмата, като се създаде ***CodeEntryPointMethod,*** и да се представи метода, при който изпълнението на програмата трябва да започне.

```
CodeEntryPointMethod start = new CodeEntryPointMethod();
CodeMethodInvokeExpression cs1 = new CodeMethodInvokeExpression(
new CodeTypeReferenceExpression(“System.Console”),”WriteLine”, new
CodePrimitiveExpression(“Hello World!”));
start.Statements.Add(cs1);
```

Следващото изявление добавя метода на входната точка, наречен **Start**, към колекцията за членове на класа

```
class1.Members.Add( start );
```

Сега ***CodeCompileUnit*** с име compileUnit съдържа структурата на CodeDOM за създаване на проста програма „**Hello World**“.

[1] https://docs.microsoft.com/en-us/dotnet/framework/reflection-and-codedom/using-the-codedom

ЗАКЛЮЧЕНИЕ

Използването на генератор на код има предимства в сравнение с конвенционалното програмиране. Използваният език за моделиране или програмиране става по-разбираем поради факта, че графичните елементи правят езика повече компактен и/или по-изчистен. Ефективността при разработката на софтуер също се повишава. Фактът, че по-малко код трябва да се пише, проверява и тества ръчно означава, че разработчиците могат да станат по-ефективни. Допълнителни аспекти са например по-добрата повторна употреба на абстрактни модели от библиотеки, работа с моделите повишава ефективността на разработчиците още повече. Това намалява цялостното усилие и натоварването, необходимо за разработка на софтуер. Съществуват обаче и редица основни проблеми, които изискват подобрения в дизайна. Невинаги може да се постигне пълна удовлетвореност в крайния резултат, а това често води до допълнителна намеса от страна на програмиста и промени по кода. Създаденият код улеснява, но и прави така, че програмистът да не отеля голямо внимание на това как кодът функционира, а го използва машинално.

ЛИТЕРАТУРА

[1] **Jason Herrington,** Code Generation in Action, Manning, 2003.

[2] **Jesse Liberty,** Programming C# 3.0, O`Reilly, 2003.

[3] **Kevin Hazzard, Jason Bock,** Metaprogramming in .NET, Manning, 2012.

[4] **Martin Bravenboer**, Generalized Type-Based Disambiguation of Meta Programs with Concrete Object Syntax.

[5] **Peter Vogel,** PRACTICAL CODE GENERATION IN .NET. Addison-Wesley, 2010.

[6] **Shane Sendall,** Taming model round-trip engineering, In Proceedings of OOPSLA Workshop on Best Practices for Model-Driven Software Development, IBM Zurich Research Laboratory, 2004.

ИНФОРМАЦИЯ ЗА АВТОРА

Ас. Даниел Дамянов, Факултет „Математика и информатика“, Великотърновски университет „Св. св. Кирил и Методий“, E-mail: damyanovdaniel@yahoo.com.

ABOUT THE AUTHOR

Assist. Prof. Daniel Damyanov, Faculty of Mathematics and Informatics, Department of Information Technologies, “St. Cyril and St. Methodius” University of Veliko Tarnovo, E-mail: damyanovdaniel@yahoo.com.