https://doi.org/10.31891/2219-9365-2022-71-3-1
УДК 004.624

ПАВЛО СТАВИЦЬКИЙ
Вінницький національний технічний університет
https://orcid.org/0000-0002-9139-6076
e-mail: pavlo.stavytskyi@gmail.com
ВІКТОРІЯ ВОЙТКО
Вінницький національний технічний університет
https://orcid.org/0000-0002-3329-7256
e-mail: dekanfki@i.ua

МЕТОД ДЕКЛАРАТИВНОГО МЕТАПРОГРАМУВАННЯ НА ОСНОВІ РОЗШИРЕННЯ СИНТАКСИСУ ІСНУЮЧИХ МОВ ПРОГРАМУВАННЯ

Розглянуто метод декларативного метапрограмування, який орієнтований на генеруввання коду. Метод базується на розширенні синтаксису розповсюджених мов програмування загального призначення. Розширення синтаксису доповнює оригінальний синтаксис мови, надаючи додатковий функціонал генерування коду за допомогою використання шаблонів. Шаблон містить точну структуру отриманого згенерованого коду, але дозволяє вказувати деталі для кожного згенерованого фрагмента. Метод додає нові елементи синтаксису, які дозволяють змінювати контексти під час написання програми, що генерує код. Зовнішній контекст виконується миттєво під час компіляції програми. Внутрішній контекст містить код шаблону, який буде згенеровано. Головне — забезпечити плавний перехід між двома контекстами, щоб код усередині кожного з них був написаний за допомогою однієї мови програмування. Метод забезпечує два типи генерування коду, а саме: миттєве та відкладене генерування. Перший створює згенерований код миттєво під час компіляції. Другий дозволяє генерувати код під час виконання програми. У цьому випадку він генерує проміжне подання шаблону у вигляді публічної функції, яку можна викликати під час виконання програми. Щоб забезпечити належний досвід використання такого підходу, важливо реалізувати інтеграцію з інтегрованим середовищем розробки, яке включає підсвічування коду та перевірку синтаксису. Крім того, коректна інтеграція може бути забезпечена шляхом розробки плагіна для систем збірки Bazel або Gradle.

Ключові слова: декларативне програмування, метапрограмування, генерування коду.

PAVLO STAVYTSKYI, VIKTORIIA VOITKO
Vinnytsia National Technical University

METHOD OF THE DECLARATIVE METAPROGRAMMING BASED ON SYNTAX EXTENSIONS OF EXISTING PROGRAMMING LANGUAGES

A method of declarative metaprogramming that focuses on code generation is considered. The method is based on extending the syntax of commonly used general-purpose programming languages. The syntax extensions sit on top of the original syntax of a language by providing an additional code generation functionality based on using templates. The template holds the exact structure of the resulting generated code but allows placeholders to specify the details for each instance. The method adds new syntax elements that allow switching contexts when writing the program that generates code. The outer context is executed instantly when the program is being compiled. The inner context holds the template code which will be generated. When using control flow in the outer context it is possible to introduce a type of safety and compile time validation during the code generation. The key is to provide a seamless transition between two contexts, such that the code inside of each of them is written with the same programming language. This is how it is possible to switch between 2 contexts multiple times inside a single file. The method provides two types of code generation such as eager generation and embeddable templates. The former produces the generated code instantly, during the compilation time. The latter allows generating code at runtime when certain circumstances match. In this case, it generates an intermediate representation of a template represented with a public function that can be called at program runtime. In order to provide a decent developer experience, it is important to provide integration with integrated development environment which includes code highlighting and syntax validation. In addition, a seamless integration could be provided by introducing a plugin for Bazel or Gradle build systems.

Keywords: declarative programming, metaprogramming, code generation.

Постановка проблеми у загальному вигляді та її зв'язок із важливими науковими чи практичними завданнями

Генерування коду мов програмування є широко розповсюдженим підходом до вирішення низки проблем в інженерії програмного забезпечення. Для виконання задач генерування коду необхідно мати інструмент, який підтримує безпеку типів та декларативність. Цей інструмент повинен бути адаптований до широкого різноманіття розповсюджених мов програмування загального призначення. Такий підхід дозволяє застосувати метод декларативного метапрограмування для широкого спектру завдань для різних платформ та мов програмування.

Аналіз досліджень та публікацій

У даному дослідженні вперше запропоновано метод декларативного метапрограмування як розширення синтаксису розповсюджених мов програмування загального призначення. Цей підхід дозволяє

організувати програми для генерування коду мов програмування [1] таким чином, що їх вхідний та вихідний код є ідентичним, зберігаючи безпечну типізацію та валідацію на стадії компіляції.

Генерування коду відбувається шляхом застосування шаблонів коду [2], що необхідно згенерувати. Кожен шаблон може бути використано декілька разів для генерування коду, схожого за структурою, але різного за деталями. Серед статичних елементів шаблонів є загальна структура коду, функції, класи, змінні, вирази тощо. Проте назви функцій, змінних, класів є унікальними щоразу, шаблон використовується для генерування нового фрагмента коду.

Формулювання цілей статті

Метою дослідження є удосконалення процесу генерування коду розповсюджених мов програмування загального призначення шляхом розширення їх синтаксису, що дозволить забезпечити кросплатформеність програмного забезпечення з підтримкою безпеки типів та декларативності. В даній роботі зроблено акцент на інтеграцію концепції декларативного метапрограмування в існуючі мови програмування шляхом розширення їх синтаксису.

Виклад основного матеріалу

Розроблений метод декларативного метапрограмування призначений для застосування та розширення синтаксису будь-яких існуючих мов програмування загального призначення, таких як Java, Kotlin, Swift, Python та багато інших. Приклад шаблону для генерування коду для мови програмування Swift наведено на рисунку 1.

```
// my_template.swift.templ

#:
for i in 1...3 {
  ~:
  func myFunction#i() {
      let value = #i
  }
  ~
}
#
```

Рис. 1. Приклад шаблону для генерування коду мови програмування Swift

Цей шаблон використовує цикл, який для кожної ітерації генерує декларування функції, ім’я та тіло якої частково є унікальними відповідно до поточної ітерації циклу. Результуючий згенерований код наведено на рисунку 2.

```
// generated.swift

func myFunction1() {
  let value = 1
}

func myFunction2() {
  let value = 2
}

func myFunction3() {
  let value = 3
}
```

Рис. 2. Swift код, згенерований за допомогою шаблону

Особливістю запропонованого методу є його універсальність та можливість застосування до будь-якої з розповсюджених мов програмування загального призначення. Приклад шаблону, створеного на базі мови програмування Kotlin, наведено на рисунку 3.

```
// my_template.kt.templ

#:
for (i in 1..3) {
  ~:
  fun myFunction#i() {
      val value = #i
  }
  ~
}
#
```

Рис. 3. Приклад шаблону для генерування коду мови програмування Kotlin

Аналогічно приклад шаблону, створеного на базі мови Python, наведено на рисунку 4.

```
# my_template.py.templ

${
for i in range(3):
  ~{
  def my_function_$i():
      value = $i
  }
}
```

Рис. 4. Приклад шаблону для генерування коду мови програмування Python

З шаблонів, наведених вище, зрозуміло, що вони містять додаткові синтаксичні елементи, які відсутні в оригінальному синтаксисі відповідних мов програмування. Крім того, конкретні синтаксичні елементи можуть відрізнятися в кожній мові програмування для зручнішої адаптації, проте основний метод залишається незмінним.

Серед запропонованих синтаксичних розширень розробленого методу виділимо такі:

`#:`, `${` — початок блоку, який містить логіку генерування коду; цей блок виконується на етапі компіляції шаблону (конкретні символи можуть відрізнятися залежно від цільової мови програмування, щоб забезпечити коректну інтеграцію в її синтаксис);

`#,` `}` — закриття блоку генерування коду;

`~:`, `~{` — початок блоку шаблонів; код усередині цього блоку буде включено до згенерованого коду без змін; крім того, аргументи, передані з іншого блоку, будуть оцінені та включені в результуючий код;

`~,` `}` — закриття блоку шаблонів.

Компіляція розширеного синтаксису відбувається аналогічно до директив препроцесора мови C++[3]. Створений компілятор аналізує файли шаблонів, шукаючи наведені вище директиви. Немає необхідності виконувати повний аналіз коду і будувати абстрактне синтаксичне дерево. Пошук директив є достатнім.

Після виконання коду шаблону він перетворюється на код, що може бути виконано за допомогою оригінального компілятора або інтерпретатора цільової мови програмування.

Таким чином можна реалізувати два типи генерації коду:

- миттєве генерування;
- відкладене генерування.

Миттєве генерування означає, що процес генерування вихідного коду відбувається миттєво перед основним кроком компіляції коду. Коли оригінальний компілятор або інтерпретатор цільової мови програмування починає свою роботу, то генерування коду уже завершено. Приклад вихідного коду, згенерованого під час миттєвого генерування, наведено на рисунку 5.

ISSN 2219-9365

```
// my_template.swift.templ

#:
for i in 1...3 {
  ~:
  func myFunction#i() {
      let value = #i
  }
  ~
}
#
```

```
// generated.swift

func myFunction1() {
  let value = 1
}

func myFunction2() {
  let value = 2
}

func myFunction3() {
  let value = 3
}
```

Рис. 5. Приклад миттєвого генерування коду для мови Swift

При розробці програмного забезпечення може виникнути необхідність у генеруванні коду під час виконання програми в залежності від відповідної логіки. Таким чином, миттєвого генерування коду може бути не достатньо, тоді під час компіляції програми потрібно створити шаблонну функцію, що генеруватиме код за викликом [4]. Приклад відкладеного генерування коду наведено на рисунку 6.

```
// my_template.swift.templ

~template

#:
for i in 1...3 {
  ~:
  func myFunction#i() {
      let value = #i
  }
  ~
}
#
```

```
// generated.swift

func my_template() -> String {
  var template = ""

  for i in 1...3 {
      template += frag0(i)
  }
  return template
}

private func frag0(_ i: Int) -> String {
  return """
  func myFunction\(i)() {
     let value = \(i)
  }\n
  """
}
```

Рис. 6. Приклад відкладеного генерування коду для мови Swift

На рисунку 6 продемонстровано, як за шаблоном згенеровано проміжну функцію my_template (за ім'ям файлу шаблону), що повертає згенерований код як рядок, який у будь-який момент може бути записаний у відповідний файл. Ключовою відмінністю шаблону в такому випадку є оператор ~template. Кожен фрагмент вихідного коду, що написано в контексті ~: та ~ операторів, згенеровано в якості окремої функції з префіксом frag.

Зазвичай під час генерування коду один шаблон необхідно використати декілька разів з різним набором аргументів. Тому необхідно забезпечити можливість передачі аргументів у шаблони. Ця можливість реалізується таким чином, що оператор ~template використовується як сигнатура функції з аргументами. Для такого шаблона буде згенеровано відповідну функцію, яка приймає ідентичний набір аргументів, що може бути переданий під час виклику та генерування коду, як продемонстровано на рисунку 7.

```
// my_templ.swift.templ

~template(
  name: String
)

~:
let user = findUser(name: #name)
~
```

```
// generated.swift

func my_templ(name: String) -> String {
  var template = ""

  template += frag0(name)
  return template
}

private
func frag0(_ name: String) -> String {
  return """
  let user = findUser(name: \(name))
  """
}
```

Рис. 7. Приклад шаблону відкладеного генерування коду з аргументами для мови Swift

Загальний алгоритм обробки шаблонних файлів з розширенням .templ наведено на рисунку 8.

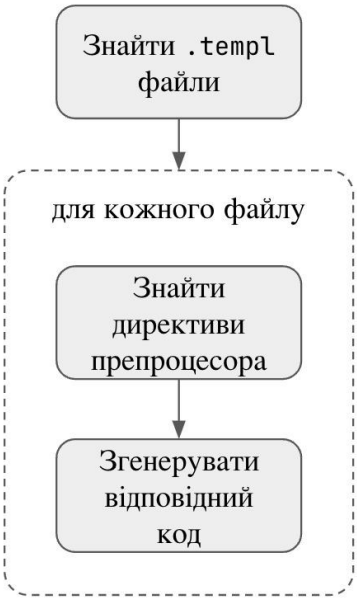

Рис. 8. Алгоритм обробки шаблонних файлів

Виклик шаблонної функції повертає рядок з кодом, що необхідно згенерувати. Після цього потрібно за замовчуванням використати механізм для створення та запису файлів тієї мови програмування, за допомогою якої створено програму для генерування коду.

Приклад створення файлу зі згенерованим кодом за допомогою шаблону відкладеного генерування коду наведено на рисунку 9.

```
// main.swift

import Foundation

let generatedCode = my_templ(name: "John Doe")


FileManager.default.createFile(
  atPath: "Users.swift",
  contents: generatedCode.data(using: .utf8),
  attributes: nil
)
```

Рис. 9. Приклад використання відкладеного шаблону коду мови Swift

Важливою складовою роботи з будь-якою мовою програмування є спеціалізований інструментарій, побудований навколо неї, зокрема: підтримка інтегрованого середовища розробки, підсвічування та валідація синтаксису тощо.

Для реалізації такого функціоналу необхідно врахувати наявність двох контекстів написаного програмного коду:

- ✓ зовнішній контекст – це код, що описує логіку генерування коду в рамках `#: , ${` та `# , }` синтаксичних елементів;
- ✓ внутрішній контекст – це код, що буде згенеровано в рамках `~: , ~{` та `~ , }` синтаксичних елементів.

Приклад розподілення контекстів коду шаблонів для мови програмування Swift продемонстровано на рисунку 10.

```
// my_template.swift.templ

#:
for i in 1...3 {
  ~:
  func myFunction#i() {
      let value = #i
  }
  ~
}
#
```

```
// my_template.swift.templ

#:
for i in 1...3 {
  ~:
  func myFunction#i() {
      let value = #i
  }
  ~
}
#
```

Рис. 10. Розподіл контекстів в шаблонах

При розробці інтеграції з зовнішнім інструментарієм необхідно розглянути такі компоненти системи декларативного метапрограмування:

- ✓ компілятор – містить логіку обробки додаткового синтаксису та генерування коду;
- ✓ плагіни для інтеграції з мовами програмування – містять логіку, специфічну для вибраної мови програмування; крім того, визначають спеціальні директиви препроцесора, які використовуються для цільової мови.
- ✓ інтерфейс командного рядка – використовується в основному системою збірки проєктів;
- ✓ компоненти для інтеграції з системами збірки проєктів – реалізують інтеграцію з системами збірки проєктів, такими як Gradle та Bazel, для автоматичної обробки додаткового синтаксису;
- ✓ інструментарій середовища розробки — реалізує функції підсвічування коду та аналізу синтаксису для файлів шаблонів.

Загальна модель роботи методу реалізації декларативного метапрограмування продемонстрована на рисунку 11.

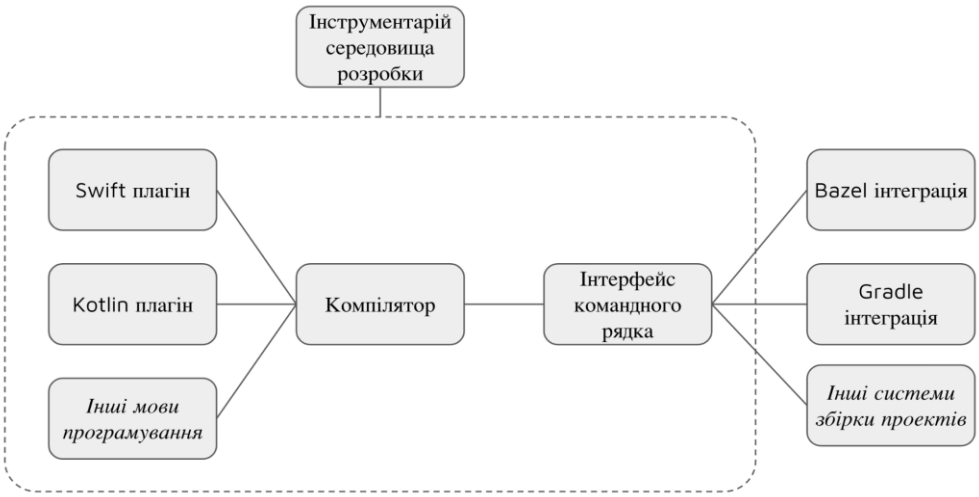

Рис. 11. Модель роботи методу реалізації декларативного метапрограмування

ISSN 2219-9365

Висновки з даного дослідження і перспективи подальших розвідок у даному напрямі

Запропонований метод декларативного метапрограмування базується на розширенні синтаксису існуючих розповсюджених мов програмування загального призначення. Такий підхід покращує процес генерування коду шляхом підтримки безпеки типізації та декларативності, що відповідає збереженню ідентичності вхідного та вихідного програмного коду. Завдяки універсальності розширюваного синтаксису розглянутий метод може бути рівноцінно застосований до будь-якої мови програмування, розширюючи спектр її застосування в області генерування коду.

Література

1. Herrington, J. (2003). Code generation in action. Manning Publications Co.

2. Gamma, E., Helm, R. and Johnson, R., 1994. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley.

3. Oualline, S. (2003). Practical C++ Programming, Second Edition (2nd ed.). O'Reilly Media.

4. SMITH, F., GROSSMAN, D., MORRISETT, G., HORNOF, L., & JIM, T. (2003). Compiling for template-based run-time code generation. Journal of Functional Programming, 13(3), 677-708. doi:10.1017/S095679680200463X

References

1. Herrington, J. (2003). Code generation in action. Manning Publications Co.

2. Gamma, E., Helm, R. and Johnson, R., 1994. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley.

3. Oualline, S. (2003). Practical C++ Programming, Second Edition (2nd ed.). O'Reilly Media.

4. SMITH, F., GROSSMAN, D., MORRISETT, G., HORNOF, L., & JIM, T. (2003). Compiling for template-based run-time code generation. Journal of Functional Programming, 13(3), 677-708. doi:10.1017/S095679680200463X