

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ЗАПОРІЗЬКА ПОЛІТЕХНІКА»

Факультет комп'ютерних наук та технологій
Кафедра комп'ютерних систем та мереж

Пояснювальна записка
до дипломного проєкту (роботи)
магістра
(ступінь вищої освіти)

на тему **СИСТЕМА АВТОМАТИЧНОЇ ГЕНЕРАЦІЇ КОДУ**
НА МОВІ ВИСОКОГО РІВНЯ

Виконав: студент 2 курсу, групи КНТ-622м
спеціальності _____
123 Комп'ютерна інженерія
(код і найменування спеціальності)

Освітня програма (спеціалізація)
«Спеціалізовані комп'ютерні системи»

ШКРЯБЕЦЬ В.І.
(ПРИЗВИЩЕ та ініціали)

Керівник КУДЕРМЕТОВ Р.К.
(ПРИЗВИЩЕ та ініціали)

Рецензент СТЕПАНЕНКО О.О.
(ПРИЗВИЩЕ та ініціали)

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет Комп'ютерних наук і технологій
Кафедра «Комп'ютерні системи та мережі»
Ступінь вищої освіти магістерський
Спеціальність 123 Комп'ютерна інженерія
(код і найменування)
Освітня програма (спеціалізація) «Спеціалізовані комп'ютерні системи»
(назва освітньої програми (спеціалізації))

ЗАТВЕРДЖУЮ

Зав. кафедри Кудерметов Р.К.

“ ” 2023 року

З А В Д А Н Н Я
НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА

ШКРЯБЕЦЬ Володимира Ігоровича

(ПРИЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Система автоматичної генерації коду
на мові високого рівня

керівник проєкту (роботи) к. т. н., доцент, КУДЕРМЕТОВ Равіль Камілович
(науковий ступінь, вчене звання, ПРИЗВИЩЕ, ім'я, по батькові)

затверджені наказом вищого навчального закладу від “24” жовтня 2023 року №400

2. Строк подання студентом проєкту (роботи) 10 грудня 2023 року

3. Вихідні дані до проєкту (роботи) синтаксис мов програмування високого рівня

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) _____

1) Аналіз підходів оптимізації мультиплатформеної розробки;

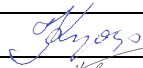
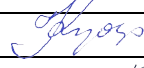
2) Проєктування системи автоматичної генерації коду;

3) Реалізація системи автоматичної генерації коду;

4) Впровадження системи.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)

6. Консультанти розділів проєкту (роботи)

Розділ	ПРИЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-4	КУДЕРМЕТОВ Р.К., доцент		
нормоконтроль	ЩЕРБАК Н.В., ст. викл.		

7. Дата видачі завдання 01.10.2023 р.

КАЛЕНДАРНИЙ ПЛАН

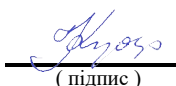
№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Аналітичний огляд аналогів та формування вимог до розроблюваної системи	10.10.2023 р.	
2	Проектування системи автоматичної генерації коду	25.10.2023 р.	
3	Реалізація системи автоматичної генерації коду	10.11.2023 р.	
4	Дослідження системи	15.11.2023 р.	
5	Розробка керівництва користувача	27.11.2023 р.	
6	Оформлення отриманих результатів у ПЗ	02.12.2023 р.	
7	Оформлення допоміжного матеріалу	10.12.2023 р.	

Студент


(підпис)

Володимир ШКРЯБЕЦЬ
(ім'я, ПРИЗВИЩЕ)

Керівник проєкту (роботи)


(підпис)

Равіль КУДЕРМЕТОВ
(ім'я, ПРИЗВИЩЕ)

РЕФЕРАТ

ПЗ: 71 с., 19 рис., 1 табл., 33 джерел.

GIT, GITHUB, GRADLE, KOTLIN, XML, МЕТАМОВА, АБСТРАКТНЕ СИНТАКСИЧНЕ ДЕРЕВО, ДЕРЕВО РОЗБОРУ, КОДОГЕНЕРАЦІЯ, ОС, СТИЛІСТИЧНЕ ДЕРЕВО

Об'єкт дослідження – синтаксис мов програмування високого рівня.

Предмет дослідження – система автоматичної генерації коду.

Мета роботи - спрощення розробки застосунків, які мають працювати на багатьох платформах, за допомогою використання системи машинного перекладу з однієї мови високого рівня на іншу.

У пояснювальній записці у першій частині проведено аналіз підходів для оптимізації мультиплатформеної розробки, таких як загальні мови програмування, скрипти, фреймворкі, методи кодогенерації та транскомпіляції, виявлено їх недоліки. Сформовано вимоги до розроблюваної системи.

У другій частині спроектовано систему автоматичної генерації коду. Розроблено та запропоновано архітектуру системи. Для подальшого масштабування вона має розділення на різні етапи обробки вхідних описів.

У третій частині описано реалізацію системи автоматичної генерації коду, яка з опису на метамові може генерувати структури та інтерфейси на мовах Kotlin, C/C++, Java, Swift.

У четвертій частині описано впровадження системи. Систему викладено у вільний доступ на платформі GitHub. Вона використовується для генерації коду у виробничому процесі для суміщення старої та нової частини комерційного продукту, написаного на різних мовах.

Галузь використання – компанії з розробки програмного забезпечення.

ABSTRACT

Explanatory note to the master's work: 71 p., 19 figures, 1 tables, 33 sources.

GIT, GITHUB, GRADLE, KOTLIN, XML, META LANGUAGE, ABSTRACT SYNTAX TREE, PARSE TREE, CODE GENERATION, OS, STYLIST TREE

The object of research is the syntax of high-level programming languages.

The subject of the study is a code generation system.

The goal of the work is to simplify the development of applications that must work on multiple platforms by using a machine translation system from one high-level language to another.

In the explanatory note, in the first part, the approaches for optimizing multi-platform development, such as common programming languages, scripting, frameworks, code generation and transcompilation methods, were analyzed and their shortcomings were identified. The requirements for the developed system have been formed.

The third part describes the implementation of the automatic code generation system, which can generate structures and interfaces in Kotlin, C/C++, Java, Swift languages from the metalanguage description.

The fourth part describes the possibilities of using the system. The system is freely available on the GitHub platform. It is used to generate code in the production process to combine the old and new parts of a commercial product written in different languages.

The field of use is software development companies.

ЗМІСТ

Перелік скорочень та умовних познач	8
Вступ	9
1 Аналіз підходів оптимізації мультиплатформеної розробки	11
1.1 Використання загальнодоступної мови та скриптів	11
1.2 Мультиплатформені фреймворки	13
1.3 Кодогенерація та транскompіляція	21
1.4 Вимоги до системи автоматичної генерації коду	22
1.5 Висновки до розділу 1	23
2 Проєктування системи автоматичної генерації коду	24
2.1 Загальний опис роботи системи машинного перекладу	24
2.2 Декларування на метамові	26
2.3 Побудова абстрактного синтаксичного дерева	29
2.4 Побудова дерева розбору	31
2.5 Стилiстичне дерево	33
2.6 Запис в файл	36
2.7 Висновки до розділу 2	39
3 Реалізація системи автоматичної генерації коду	39
3.1 Формати даних	39
3.2 Програмний інструментарій	40
3.3 Налаштування проєкту	44
3.4 Побудова AST-дерева	44
3.5 Побудова дерева розбору	47
3.6 Приклад трансляції вузлу DataClass на мову Kotlin	48
3.7 Приклад трансляції вузлу DataClass на мову C++	50
3.8 Побудова стилістичного дерева	52
3.9 Запис в файл	53
3.10 Висновки до розділу 3	55

4 Впровадження системи.....	55
4.1 Використані програмні засоби.....	55
4.2 Опис роботи системи кодогенерації та можливостей використання	58
4.3 Інтеграція системи генерації в систему збірки Gradle	63
4.4 Висновки до розділу 4	65
Висновки	66
Перелік джерел посилання	68
Додаток А.....	71

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАК

ANSI	– American national standards institute;
API	– Application programming interface;
AST	– Abstract syntax tree;
BSD	– Berkeley Software Distribution;
DSL	– Domain Specific Language;
Git	– Global Information Tracker;
IBM	– International Business Machines;
IDE	– Integrated Development Environment;
JNI	– Java Native Interface;
JSON	– JavaScript Object Notation;
JVM	– Java Virtual Machine;
KMP	– Kotlin Multiplatform;
PR	– Pull-Request (пропозиція зміни коду у репозиторії);
SDK	– Software Development Kit;
SSL	– Secure Sockets Layer;
XML	– eXtensible Markup Language;
OC	– операційна система.

ВСТУП

Розробка мультиплатформеного застосунку, тобто під різні операційні системи (ОС), зазвичай означає, що одну і ту ж логіку потрібно буде писати декілька разів для кожної з обраних ОС на притаманній мові програмування. Це призводить до зайвого витрачання людських та фінансових ресурсів.

Існує декілька підходів для оптимізації мультиплатформеної розробки, такі як загальна кодова база (фреймворки), загальні мови програмування (наприклад C++, C) або використання скриптів (наприклад lua/javascript). Але це працює допоки застосунок нескладний та й має свої недоліки. Загалом це неможливість, або складність використання таких підходів в умовах коли проєкт вже в розробці і треба частину перевести на загальні рейки. Доволі часто це призводить до збільшення часу розробки, або ж може вийти непритаманний інтерфейс користувача. Також крім спеціалістів з розробки під конкретну ОС потрібні ще додаткові розробники зі спеціалізованим досвідом.

Системи машинного перекладу з однієї мови високого рівня на іншу далі будемо називати транскompілятором [1-3].

Транскompіляція може бути використана для створення версій програмного забезпечення для різних мов або платформ. Це може полегшити підтримку і розширення програм у різних середовищах. Застосування транскompіляторів дозволяє об'єднати функціонал різних мов програмування в одному проєкті, що може бути корисним для певних завдань або команд розробників.

Транскompілятори можуть бути використані для перетворення вихідного коду для оптимізації його продуктивності чи роботи в іншому контексті. У разі, коли розробники хочуть перейти на нову мову програмування, транскompіляція може допомогти у перетворенні існуючого коду, щоб уникнути написання з нуля.

Великі проєкти можуть використовувати декілька мов програмування в різних частинах системи. Транскompіляція може полегшити спільну розробку і

підтримку таких проєктів. Транскомпіляція може використовуватися для внесення особливих функцій чи можливостей однієї мови в іншу. Таким чином, обрана тема магістерської роботи є актуальною.

Мета роботи - спрощення розробки застосунків, які мають працювати на багатьох платформах, за допомогою використання системи машинного перекладу з однієї мови високого рівня на іншу.

Для досягнення поставленої мети необхідно виконати наступні завдання:

- провести аналіз підходів оптимізації мультиплатформеної розробки;
- на базі проведеного аналізу визначити вимоги до розроблюваної системи автоматичної генерації коду;
- спроектувати систему автоматичної генерації коду;
- реалізувати систему;
- впровадити систему у компанію з розробки програмного забезпечення;
- провести аналіз отриманих результатів.

У роботі розглянуто можливість дати спеціалістам такий інструмент, що дозволить працювати в комфортній для них середі-мові та використовувати код один одного.

Оскільки загальна система машинного перекладу досить складна, то в рамках дипломного проєкту приділено основну увагу розробці блоку кодогенерації, тобто системі автоматичної генерації коду на мові високого рівня.

1 АНАЛІЗ ПІДХОДІВ ОПТИМІЗАЦІЇ МУЛЬТИПЛАТФОРМЕНОЇ РОЗРОБКИ

1.1 Використання загальнодоступної мови та скриптів

Існує декілька підходів для пришвидшення мультиплатформеної розробки. Наприклад, написання спільної бізнес-логіки використовуючи загальнодоступну мову (як-то C/C++). Але це працює допоки застосунок нескладний та й має свої недоліки. Неможливість, або складність використання такого підходу в умовах, коли проєкт вже в розробці, і треба частину перевести на загальні рейки.

1.1.1 Мова програмування C/C++

C++ [4-6] — мова програмування високого рівня загального призначення, створена датським комп'ютерним науковцем Б'ярне Страуструпом. Вперше випущений у 1985 році як розширення мови програмування C.

Переваги використання мови C++:

- C++ є доволі сучасним, остання версія стандарту датується 2020 роком, C++ 2023 також буде прийнятий найближчим часом;
- C++ є портативним, він дозволяє вам використовувати найновіші можливості мови на всіх платформах, навіть на старих платформах;
- скомпільований код на C++ швидкий, та займає небагато місця.

Недоліки використання мови C++:

- C++ має обширний функціонал. Стандартна бібліотека величезна, і навіть досвідчені розробники можуть не знати про неї всього;
- в мовах високого рівня зазвичай немає безшовної інтеграції з C++, що ускладнює використання цієї мови для побудови загальної логіки. Наприклад в Java/Kotlin потрібно писати додатковий код-обгортку (JNI), теж саме і в середовищі Apple/Swift;
- з C++ губиться можливість використання спеціалізованих зручних методів роботи з мережею, базою даних, структурами даних (JSON/XML), що також

збільшує час розробки;

- крім спеціалістів з розробки під конкретну ОС потрібні ще додаткові інженери зі знанням C/C++.

1.1.2 Скриптова мова програмування Lua

Lua [7] — це легка, високорівнева мова програмування, розроблена в основному для вбудованого використання в програмах. Lua є кросплатформною, оскільки інтерпретатор скомпільованого байт-коду написаний мовою ANSI C, а Lua має відносно простий C API для вбудовування його в програми.

Переваги використання мови Lua:

- кросплатформна - будується на будь-якій платформі з компілятором ANSI C;

- Lua настільки ж проста, як Python, з точки зору навчання написанню коду;

- вбудована та розширювана мова, яка забезпечує простий інтерфейс з C/C++/Objective-C;

- швидкий та якісний JIT-компілятор, який помітно покращує продуктивність багатьох завдань;

- добре задокументована - є довідковий посібник, книга, вікі;

- дружна та повна ентузіазму спільнота;

- чистий і простий синтаксис, підходить для початківців і доступний для не програмістів;

- інтегрований інтерпретатор - можна просто запустити lua з командного рядка;

- вбудована підтримка співпрограм для реалізації ітераторів і багатопоточності без випередження;

- інкрементний механізм зборки сміття в пам'яті з низькою затримкою, невеликою складністю реалізації та підтримкою слабких посилань;

- проста, але потужна бібліотека налагодження.

Недоліки використання мови Lua:

- в Lua немає 100% зворотної сумісності. Нові версії інтерпретатора часто

порушують роботу скриптів, написаних для попередніх версій;

- обмежена підтримка обробки помилок;
- немає підтримки Unicode;
- Lua підтримує лише обмежену кількість структур даних, як то масиви, списки та хеш-таблиці;
- немає фіналізаторів класів/об'єктів.

1.2 Мультиплатформені фреймворки

Далі розглянемо інший підхід - використання фреймворків.

1.2.1 Фреймворк Flutter

Flutter [8-9] програмне забезпечення з відкритим кодом, створене корпорацією Google. Призначено для розробки крос-платформних застосунків з єдиною кодовою базою для Web, Android, iOS, Windows, macOS, linux. Існує з 2017 року. Використовує мову Dart [10].

Переваги фреймворку Flutter:

- використання мови Dart. Спочатку Flutter можна було назвати сумішню JavaScript та C++, але у розробників з'явилися проблеми з кодуванням. Тому було ухвалено рішення знайти іншу мову, і нею став Dart. Його особливості в тому, що при збиранні видаляється весь невикористаний код, проводиться швидке виділення та очищення пам'яті, а також забезпечується висока швидкодія на пристрої користувача. Іншими словами, чудова мова для розробки застосунків;
- Flutter, в свою чергу, використовує менеджер пакетів dart, також відомий як інструмент pub. Пакети поставляються у вигляді вихідного коду і компілюються разом з основним застосунком;
- Flutter поставляється з компонентами Material і Cupertino з коробки;
- Flutter рендерить все, використовуючи власний 2D-рушій Skia, уникаючи

будь-якого спеціального з'єднання між представленням і іншим кодом. Це робить рендеринг неймовірно швидким. У нього можуть бути проблеми з iOS, але ця проблема повинна бути вирішена за допомогою новітньої підтримки Metal.

Недоліки фреймворку Flutter:

- фреймворк ще недостатньо популярний, отже знайти розробників здатних створювати якісні програми може бути нелегко;
- хоч стандартний Flutter має компоненти UI Material (для Android) та Cupertino (для iOS), але все одно бібліотека таких елементів не занадто велика;
- невелика кількість сторонніх бібліотек;
- з Flutter набагато складніше змінити застосунок, так як Dart код заздалегідь компілюється в бінарний код для цільової архітектури;
- для багатьох розробників мова Dart буде новою, тому доведеться витратити час на її вивчення. Попри те, що він дуже зручний для розробки мобільних застосунків, не всі фахівці хочуть вникати в нову мову програмування;
- інтерфейс доводиться створювати за допомогою коду, тому складніше знайти межу між дизайном та логікою. Іншими словами, не завжди можна зрозуміти, які елементи зручні, а які - ні, доки не буде написаний код та не скомпілюється програма;
- певна нестабільність. Фреймворк нещодавно вийшов із бета тестування, у ньому небагато бібліотек та деякі з них не завжди стабільно працюють, але з кожним днем цей недолік поступово зникає.

1.2.2 Технологія Kotlin Multiplatform (KMP)

Технологія Kotlin Multiplatform [11] розроблена для спрощення розробки кросплатформних проєктів. Це скорочує час, витрачений на написання та підтримку того самого коду для різних платформ, зберігаючи при цьому гнучкість і переваги рідного програмування.

На рисунку 1.1 продемонстровано архітектуру технології Kotlin Multiplatform.

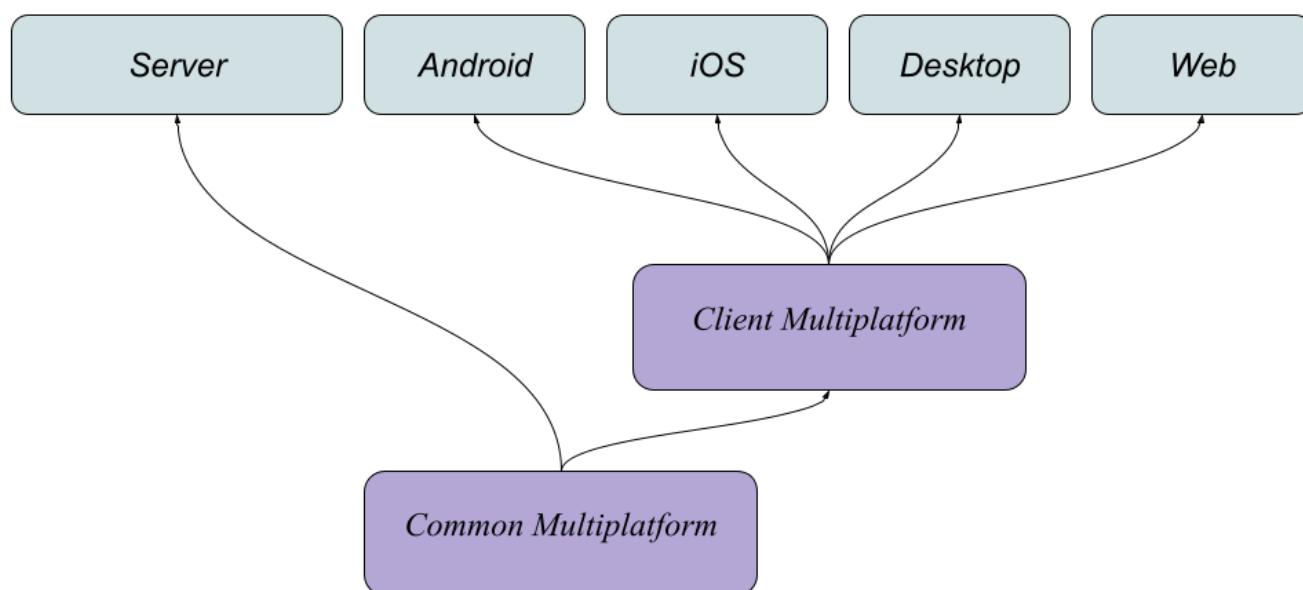


Рисунок 1.1 - Архітектура технології Kotlin Multiplatform

Переваги технології Kotlin Multiplatform:

1) загальний код - КМР дозволяє написати код один раз, і він працюватиме на різних типах пристроїв. Окрім того:

– КМР легко розширити. Щоб досягти певної поведінки на платформах Flutter і React Native, доводиться використовувати API, підготовлений розробниками фреймворку. Додавання специфічного для платформи коду може бути складним і внести додаткові помилки. Але в Kotlin Multiplatform можна з легкістю створити власний загальний API для деяких функціональних можливостей певної платформи. Наприклад, для малювання графіки Android використовує свій рідний Canvas, тоді як для Windows буде використано механізм візуалізації Skia. Якщо є потреба, ви можете створити загальний API, який буде відображати те саме для користувача, але під ним використовуватиметься технологія, яка підходить для даної платформи;

– є можливість розширення існуючий застосунок за допомогою КМР. Наприклад, якщо програма для Android і iOS вже існує, і для обох потрібно додати

певну нову функціональність, це можна зробити в Kotlin Multiplatform, щоб скоротити час розробки, необхідний для впровадження;

2) платформи специфічний код - у KMP надає більше можливостей для використання коду для конкретної платформи. Наприклад можна написати інтерфейс Android у Jetpack Compose, а інтерфейс iOS у SwiftUI та змусити їх обидва використовувати спільну кодову базу для бізнес-логіки. Але також можна використовувати спільну бібліотеку KMP, як-от Compose Multiplatform, для розробки інтерфейсу користувача та коду бізнес-логіки. Тож якщо якийсь функціонал взагалі неможливо реалізувати мультиплатформено - то Kotlin Multiplatform надасть вам простий спосіб вирішити їх незалежно та додати їх до вашої спільної кодової бази для майбутнього використання;

3) мова Kotlin [12-14]. Працюючи з KMP, Android та деякі бекенд-розробники почуватимуться як вдома, оскільки весь код написаний мовою Kotlin. Згідно з опитуванням Stack Overflow Survey 2022 [15], Kotlin більш популярний (9,16% порівняно з 6,54%) і більш улюблений, ніж Dart – мова фреймворка Flutter. Kotlin також забезпечує більшу безпеку під час компіляції, порівняно з JavaScript (React Native), тож розробники отримують помилки саме при компіляції програми, а не після її запуску. Завдяки цьому розробники можуть знаходити помилки на ранній стадії, що підвищує продуктивність.

Недоліки технології KMP:

- невелика спільнота – Kotlin Multiplatform все ще перебуває на початковій стадії, тому в Інтернеті помітно менше ресурсів та інформації по цьому фреймворку. Незважаючи на це, JetBrains регулярно розміщує на своєму офіційному сайті детальну документацію, де можна знайти відповіді на найважливіші питання та краще зрозуміти як працює KMP;

- обмеження Objective-C - під час використання KMP для програми iOS існують деякі обмеження та неочікувана поведінка, коли код Kotlin перекладається на Objective-C (мова iOS). Наприклад, параметри функцій за замовчуванням у Kotlin не працюють після перекладу, оскільки Objective-C взагалі не підтримує

параметри за замовчуванням. Процес перекладу з Kotlin на Objective-C на даний момент займає трохи часу, що може уповільнити розробку під iOS.

1.2.3 Фреймворк React Native

React Native [16] - фреймворк з відкритим програмним кодом, розроблений Meta. Використовується для розробки застосунків під операційні системи Android, iOS, macOS, tvOS, Web, Windows, UWP. Базується на використанні HTML5 та JavaScript.

Переваги фреймворку React Native:

- React-Native має велику бібліотеку компонентів інтерфейсу для користувача: календарі, засоби вибору дати, кнопки, компоненти матеріалів, підтримку SVG і багато іншого;

- React Native використовує пакети NPM для включення залежностей, включаючи сам React Native. Крім NPM, можна використовувати Yarn для установки і управління своїми залежностями. Залежності, які вимагають коду Kotlin/Java/Swift/Objective-C, зв'язуються за допомогою модулів gradle і пакетів CocoaPods. React Native включає функцію автоматичної лінковки, яка дозволяє уникнути ручної зміни файлів gradle і Podfile;

- React Native присутній на ринку вже більше 5 років і має в своєму розпорядженні тонни бібліотек для різних випадків, починаючи з push-повідомлень, відтворення відео і компонентів матеріалів, закінчуючи камерою (включаючи Firebase ML Kit) і інтеграціями Apple HealthKit і Google Fit.;

- React Native також сильно виграє від того, що він заснований на React і має можливість використовувати бібліотеки, створені для React, наприклад Redux або axios, або привносити сучасні функції, такі як React Hooks.

Недоліки фреймворку React Native:

- React Native використовує рідні віджети платформи (Native Views) і передає події через JavaScript. Це впливає на продуктивність рівня уявлення, однак 60 fps в секунду все ще досяжні, хоча продуктивність залежить від версії ОС і самого пристрою;

– React Native зав'язаний на JavaScript і застосунок на його основі містить js bundle. Але його завжди можна витягти і зрозуміти логіку програми, або змінити її, що є серйозною проблемою.

1.2.4 Фреймворк Xamarin

Xamarin [17] - являє собою подальше розвитку ідей фреймворку з відкритим кодом Mono [18]. Він дозволяє запускати .NET код на не-Windows платформах та в першу чергу призначений для розробки застосунків саме під мобільні платформи як-то Android, iOS.

Переваги фреймворку Xamarin:

– можливість писати єдиний код для всіх рішень є найочевиднішою перевагою Xamarin, але вона також є найважливішою. Точний обсяг коду, який можна спільно використовувати між різними платформами, здебільшого залежить від того, як код розподіляється між ядром і інтерфейсом користувача. Але загалом технологія Xamarin дозволяє групі розробників створювати близько 65-90 відсотків багаторазового коду. Ця перевага істотно спрощує і прискорює процес розробки. Для клієнтів це економить гроші та прискорює час виходу на ринок;

– продуктивність програми розроблених на Xamarin наближається до рівня програм написаних на рідній мові, і це є однією з найважливіших переваг Xamarin. Деякі інші багатоплатформні інструменти використовують інший шаблон для розробки мобільних додатків. Зокрема, вони дозволяють створювати мобільні додатки, такі як веб-сайти, тобто з використанням таких технологій, як HTML, CSS і JavaScript. В основному такі програми працюють у браузері, але веб-сторінки відображаються як нативна програма. Це зазвичай призводить до низької продуктивності та взаємодії з користувачем. За допомогою Xamarin програмісти можуть досягти показників продуктивності, які практично не відрізняються від власних програм. Завдяки цьому Xamarin виділяється з-поміж інших крос платформних засобів розробки;

– рідний для платформи UX - забезпечення звичного для користувача досвіду є однією з головних причин, чому компанії обирають індивідуальну мобільну

розробку для кожної платформи. Проте Xamarin надає програмістам усі інструменти, необхідні для створення рідного UX для кросплатформених програм. Xamarin.Forms дозволяє надавати спільний доступ до компонентів інтерфейсу користувача на різних платформах, показуючи їх як елементи інтерфейсу для певної платформи під час виконання. Розробники також можуть використовувати Xamarin.iOS і Xamarin.Android для створення рідного інтерфейсу користувача програми та отримання ще кращих результатів:

- підтримка Microsoft - Xamarin підтримується корпорацією Microsoft, і це величезна перевага для підприємств, яким потрібна надійна технологія. Перш за все, це означає, що ця платформа для мобільних розробок буде своєчасно оновлюватися і не припинить свою роботу в осяжному майбутньому. По-друге, Xamarin надає спільноті різноманітні навчальні ресурси. Корпорація Майкрософт вважає цю технологію своїм стратегічним продуктом і пропонує широкі можливості навчання для інженерів програмного забезпечення, які хочуть почати працювати з Xamarin;

- спрощена підтримка коду - обслуговування програм Xamarin зазвичай легше, ніж обслуговування застосунків написаних на окремих мовах для кожної платформи. Щоб оновити програми, розроблені для різних платформ, програмістам достатньо внести зміни у вихідний файл. Такі зміни автоматично відображаються в iOS, Android та інших продуктах. Ще один аспект – зручність співпраці з розробниками. Якщо продукт заснований на Xamarin, одна людина може виконувати весь обсяг роботи та підтримувати оновлення програми. Оскільки немає необхідності дублювати зміни на кожній платформі, це може заощадити багато часу та грошей;

- повна підтримка апаратного забезпечення - оскільки використання Xamarin дозволяє створювати функціональні можливості нативного рівня в кросплатформних програмах, ризик проблем із сумісністю апаратного забезпечення практично виключається. Використовуючи різні API та платформи, програмісти можуть створювати продукти, які безперебійно працюватимуть на всіх

цільових пристроях. Крім того, Xamarin забезпечує розширений рівень налаштування, увімкнувши підключення до рідних бібліотек.

Недоліки фреймворку Xamarin:

- ціна – хоч сам Xamarin є відкритим кодом. Але для комерційної розробки потрібна ліцензія на Visual Studio. Ціна починається від 1199 доларів США за перший рік підписки на професійний пакет;

- спільнота Xamarin постійно зростає, але все ще значно менша, ніж, наприклад, спільноти iOS або Android. Можна зіткнутися з проблемою пошуку професійного розробника Xamarin;

- більший розмір застосунків - застосунки розроблені з використанням Xamarin зазвичай трохи важчі аніж застосунки розроблені на SDK цільової платформи. Причина - додатковий код, який виникає через необхідність створення спільного функціоналу.

- затримка оновлень SDK. Коли iOS чи Android випускають деякі оновлення SDK, програмісти на Xamarin не можуть негайно використати ці оновлення. Перед цим команда Xamarin повинна інтегрувати їх у свою екосистему, наприклад, увімкнувши відповідну підтримку або запровадити новий плагін. Тому можуть виникнути деякі затримки. Але розробники Microsoft прагнуть надавати випуски в той же день, тому затримки іноді є проблемою.

1.2.5 Висновки

Таким чином, у кожного з фреймворків є свої переваги та недоліки, не існує ідеального та універсального рішення.

Взагалі ж при використанні фреймворків для мультиплатформеного програмування може вийти непритаманний інтерфейс користувача. Також, крім спеціалістів з розробки під конкретну ОС потрібні ще додаткові розробники зі знанням обраного фреймворку.

1.3 Кодогенерація та транскompіляція

Генерація коду [19] допомагає автоматизувати процес розробки застосунків. Існує багато систем для генерації коду [19, 20]. Кожна з них може використовуватися у певній області та має свої обмеження та недоліки. Зазвичай потрібно описати необхідний код за певними правилами або за допомогою графічних інструментів, або за допомогою деякого абстрактного опису. Нас цікавить можливість генерації вихідного опису на кілька мов високого рівня для автоматизації та спрощення написання застосунків для різних платформ.

Транскompілятор приймає вихідний код програми, написаної на мові програмування в якості вхідних даних і виконує еквівалентний вихідний код на тій самій або іншій мові програмування [3]. Транскompілятор виконує перетворення між мовами програмування, які працюють майже на одному рівні абстракції, в той час як традиційний компілятор перекладає з мови програмування більш високого рівня на мову програмування нижчого рівня. Наприклад, транскompілятор може виконувати переклад програми з Python на JavaScript, в той час як традиційний компілятор перекладає з мови C на асемблер чи з Java в байт-код.

Якщо розглянути існуючі транскompілятори (Source-to-source compiler) то можна побачити, що історично більшість з них була призначена:

- для міграції з однієї апаратної платформи на іншу. Наприклад Intel CONV86, SCP TRANS86, Sorcim TRANS86 - були призначені для перекладу асемблерного коду з платформи intel 8080 на intel 8086;

- для портування кодової бази з застарілої версії мови високого рівня на нову версію. Наприклад 2to3 - переклад Python 2 на Python 3; проєкти c2go, c2rust - призначені для трансляції з C на rust або Go;

- також існують комерційні проєкти для спеціалізованого перекладу з однієї мови високого рівня на іншу. Такі проєкти зазвичай мають досить обмежений набір підтримуваних мов, як то переклад з Fortran на Java, або Java на C#.

Як можна помітити всі такі проєкти призначені для одноразового перекладу кодової бази. І такий підхід неможливо використати в виробничих цілях для постійного перекладу коду в різних напрямках.

1.4 Вимоги до системи автоматичної генерації коду

Метою дипломного проєкту є спрощення розробки застосунків, які мають працювати на багатьох різних платформах, за допомогою використання системи машинного перекладу з однієї мови високого рівня на іншу. Можливість дати спеціалістам такий інструмент, що дозволить працювати в комфортній для них середі-мові та використовувати код один одного.

Об'єкт розробки – синтаксис мов високого рівня.

Предмет розробки – система автоматичної генерації коду.

Коректно спроектований застосунок зазвичай складається з багатьох незалежних або слабо зв'язаних блоків. Кожен з розробників може реалізувати якусь частину бізнес логіки на зручній йому мові: C/C++, Kotlin/Java, Swift, тощо. А система машинного перекладу (транскомпілятор) повинна транслювати цю частину на всі інші мови-платформи для подальшого використання вже під конкретну ОС на рідній для неї мові програмування з мінімальним втручанням людини.

Для досягнення поставленої мети позначимо проміжні результати розробки транскомпілятора:

- можливість транслювати структури та інтерфейси (примітиви), з мінімальним втручанням людини;
- можливість транслювати класи, функції та прості алгоритми, з мінімальним втручанням людини;
- можливість транслювати складні алгоритми, з мінімальним втручанням

людини.

Наведемо вимоги до системи автоматичної генерації коду:

- простота та зручність використання;
- коректність перекладу з мінімальним втручанням людини;
- можливість розширювати функціонал на інші мови програмування;
- достатній рівень швидкодії;
- прийнятний рівень вимог до апаратних ресурсів;

Окрім того для процесу розробки мультиплатформених застосунків є особливі вимоги, як то:

- можливість роботи системи на різних платформах - оскільки розробка таких застосунків зазвичай ведеться на багатьох операційних системах - Linux, Windows, Mac OS X;

- можливість “безшовної” інтеграції з існуючим кодом - неприпустимо щоб система транскompіляції перезаписала або якось змінила важливі частини вже існуючого коду розробленого інженерами;

- можливість двонаправленої трансляції, тобто з мови А на мову Б, а потім з мови Б на мову А;

- можливість масштабування та ітеративної компіляції - з часом кодова база зростає та процес транскompіляції може займати значний час, щоб уникнути проблем в майбутньому потрібно врахувати можливість розподіленого запуску системи на різних комп'ютерних системах зв'язаних мережею.

1.5 Висновки до розділу 1

Проведено аналіз підходів для оптимізації мультиплатформеної розробки, таких як загальні мови програмування, скрипти, фреймворкі, методи кодогенерації та транскompіляції.

У кожного з методів є свої прихильники та області використання, але ж не існує ідеального та універсального рішення. Взагалі це працює допоки застосунок нескладний та й має свої недоліки. Особливо неможливість, або складність використання таких підходів в умовах коли проєкт вже в розробці і треба частину перевести на загальні рейки. Доволі часто це призводить до збільшення часу розробки, або ж може вийти непритаманний інтерфейс користувача. Також крім спеціалістів з розробки під конкретну ОС потрібні ще додаткові розробники зі спеціалізованим досвідом.

Таким чином на основі аналізу підходів для оптимізації мультиплатформеної розробки було сформовано вимоги до розроблюваної системи.

2 ПРОЄКТУВАННЯ СИСТЕМИ АВТОМАТИЧНОЇ ГЕНЕРАЦІЇ КОДУ

2.1 Загальний опис роботи системи машинного перекладу

Оскільки різні мови програмування мають різну структуру проєктів та наповнення файлів з програмним кодом, то неможливо безпосередньо зробити переклад з однієї мови на іншу та отримати результуючий працюючий код. Цей процес має відбуватись поетапно [2] та наведено на рисунку 2.1.

Спочатку необхідно обробити початковий код на мові високого рівня та згенерувати його опис на метамові, для чого використовується нейронна мережа.

Метамова - це мова для опису загальної структури програми (логіки), яка базується на Kotlin Script. З опису на метамові формується абстрактне синтаксичне дерево, яке має наступну ієрархію рівнів: зверху простір імен; потім рівень класів, інтерфейсів; потім рівень методів та полей; рівень аргументів та логіки [1].

Для навчання нейронної мережі можливо використати Project CodeNet [21] від IBM. CodeNet - це великий набір даних, призначений для навчання систем штучного інтелекту тому, як перекладати код. Набір даних включає близько 14 млн

фрагментів коду та 500 млн рядків 55 мовами - від COBOL і FORTRAN до Java, C++ та Python. У ньому міститься близько 4 тисяч задач кодування. Дані CodeNet загальнодоступні, репозиторій розміщено на GitHub [22].

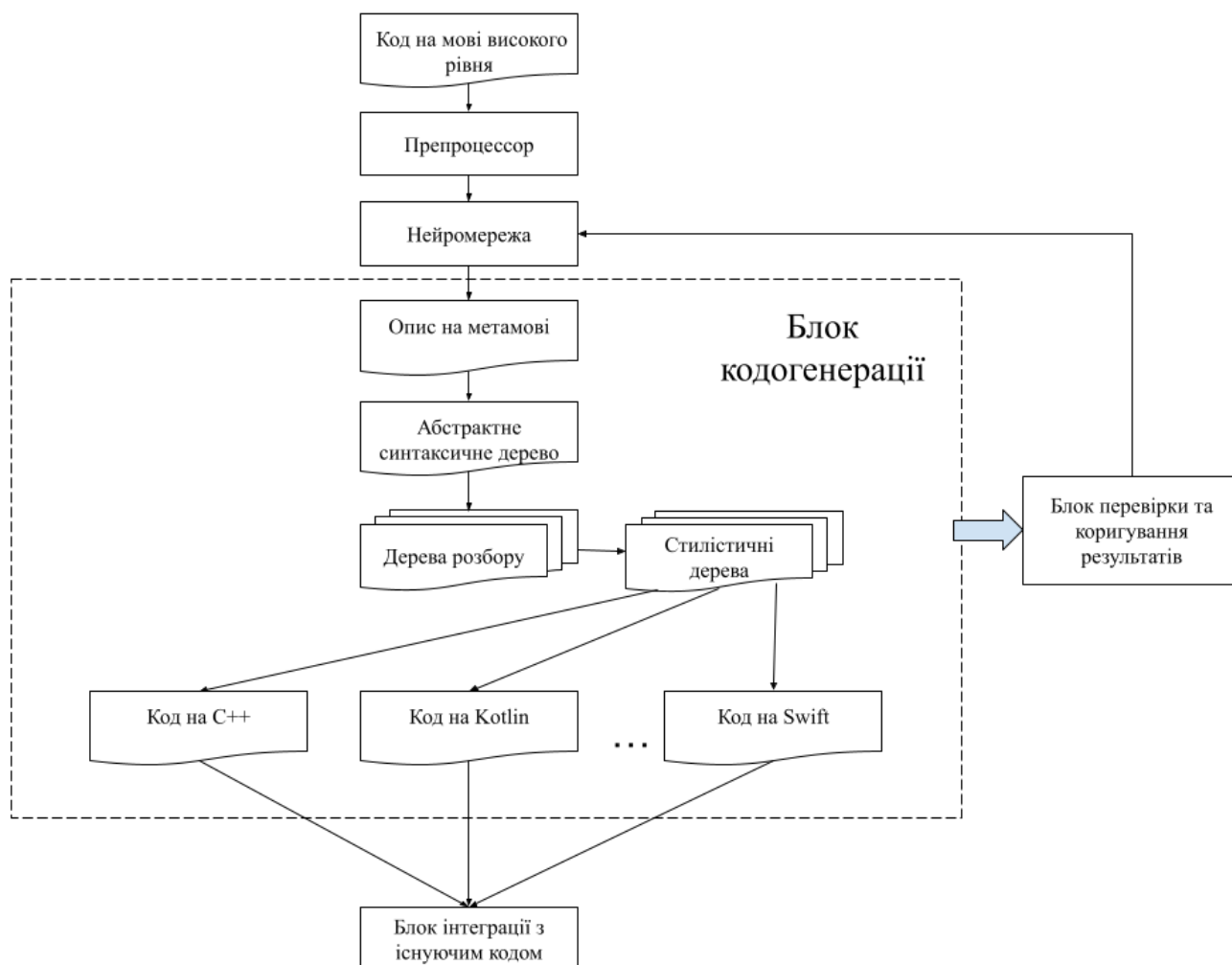


Рисунок 2.1 - Діаграма роботи системи машинного перекладу

Наступний етап - це трансляція та побудова вихідного дерева розбору за правилами потрібної мови високого рівня. На верхньому рівні воно має розбиття на файли, потім рівень просторів імен, далі класи та інтерфейси, потім рівень методів та полів; рівень аргументів та логіки.

З дерева розбору потрібно побудувати наступне дерево – стилістичне, при його створенні будуть враховані задані користувачем правила оформлення коду,

тобто стиль коду.

На останньому етапі кодогенерації виконується побудова вихідного коду за вихідним стилістичним деревом і запис його в файли.

Далі відбувається перевірка та коригування результатів, що допомагає з кожним разом все краще обучати нейронну мережу та поліпшувати результати машинного перекладу коду.

Блок інтеграції з існуючим кодом має автоматично або при мінімальному втручанні програмістів додавати згенерований код в існуючі частини проєкту. Цей функціонал базується на можливостях системи розподіленої система керування версіями файлів та спільної роботи Git [23].

Оскільки загальна система машинного перекладу досить складна, то в рамках дипломного проєкту приділимо основну увагу блоку кодогенерації, тобто системі автоматичної генерації коду на мові високого рівня.

2.2 Декларування на метамові

Блок кодогенерації починає роботу з обробки опису потрібного коду на метамові. Метамова описує загальну структуру програми та базується на Kotlin Script [24]. Вона була розроблена в процесі проєктування системи та являється проміжним етапом для відладки.

Метамова містить засоби для декларативного опису структур, класів, інтерфейсів, типів даних, методів, тощо та дозволяє програмісту вивчати і змінювати опис структур в найбільш зручний для людини спосіб.

В таблиці 2.1 наведено ключові слова метамови.

Таблиця 2.1 - Ключові слова метамови

constantsBlock	декларування блоку констант
namespace	декларування простору імен - декларативної області, у якій можуть розміщуватися різні елементи програми [4]
declareInterface	декларування інтерфейсу. Визначення протоколу, методів та типів даних, з яким об'єкти погоджуються для співпраці. Протокол є описом: - повідомлень, які розуміються об'єктом; - аргументів, якими можуть надаватися ці повідомлення; - типів результатів, до яких ці повідомлення повертаються; - інваріантів, які зберігаються попри модифікації стану об'єкта; - виняткових ситуацій, які вимагатимуть клієнтів обробляти об'єкт;
dataClass	декларування структури - складового типу даних, побудованого з інших типів [25]
addClass	декларування загального класу
enum	декларування переліку, тобто типу даних що складається з множини ідентифікаторів, що являють собою можливі значення змінних цього типу [25]
field	декларування змінної або поля даних
addMethod	додавання методу в клас або інтерфейс.
addBlockComment	додавання коментарю до будь якого декларування.

На прикладі з лістингу 2.1 показано, як виконується етап кодогенерації. На вході модулю генерації маємо декларативний опис потрібних блоків на метамові.

Листинг 2.1 - Декларування на метамові

```

1. namespace("com.goldman.data").apply {
2.     declareInterface("DataReader").apply {
3.         addMethod(
4.             "getSize",
5.             OutputList().apply {
6.                 output("size", DataType.int64)
7.             }, null
8.         )
9.         addMethod(
10.            "read",
11.            outputs = OutputList().apply {
12.                output("size", DataType.int64)
13.                outputReusable("array",
14.                    DataType.array(DataType.uint8))
15.            },
16.            inputs = InputList().apply {
17.                argument("offset", DataType.int64)
18.                argument("size", DataType.int32)
19.            }
20.        )
21.    }
22.    dataClass("GoldBuffer").apply {
23.        field("blockCount", DataType.int32)
24.        field("lastBlockSize", DataType.int32, 10)
25.        field("blockSize", DataType.int32)
26.    }
27. }
```

Як можна побачити опис починається з декларування простору імен за допомогою ключового слова `namespace` (рядок 1). В рядку 2 декларуємо сам інтерфейс `declareInterface("DataReader")`.

Далі іде опис методів інтерфейсу за допомогою ключового слова `addMethod` рядки 3 та 9. Розглянемо останній метод `read`. Опис вхідних аргументів в рядках 15-18 в блоці `InputList`. На вход приймає два аргумента `offset` та `size`, цілочисельне знакове розміром 64 біта та 32 біта відповідно. Результати на виході описані в блоці `OutputList` рядки 11-14. Цей метод повертає:

- `size` - цілочисельне знакове число розміром 64 біта;

- array - масив байт. Ключове слово outputReusable означає що метод приймає цю змінну зовні та кладе результат в неї. Така техніка потрібна для того щоб не створювати додатковий буфер, та не витратити зайву пам'ять.

Наступний приклад - декларування структури (клас даних в термінах Kotlin) GoldBuffer в рядках 22-26. Ця структура містить лише три поля даних:

- blockCount - цілочисельне знакове число розміром 32 біта;
- blockSize - цілочисельне знакове число розміром 32 біта;
- lastBlockSize - цілочисельне знакове число розміром 32 біта, зі значенням за замовчуванням 10.

2.3 Побудова абстрактного синтаксичного дерева

На основі опису на метамові система будує абстрактне синтаксичне дерево.

Абстрактне синтаксичне дерево (Abstract Syntax Tree або AST) є структурою даних, яка представляє синтаксичну структуру програми, відображаючи її синтаксичний аналіз у вигляді дерева. AST є інтермедіативним представленням програмного коду, яке дозволяє відокремити синтаксичну структуру від конкретних деталей синтаксису.

Кожен вузол в синтаксичному дереві представляє конструкцію мови програмування (таку як оператор, вираз, об'єкт чи інше) і має дочірні вузли, які представляють його складові частини або аргументи. Дерево може бути використано для виконання різних видів аналізу програми.

Абстрактне синтаксичне дерево важливе для компіляторів та інших інструментів аналізу програмного коду. Він допомагає розуміти структуру коду на рівні високого рівня, спрощує виконання оптимізацій, перетворень та інших операцій над програмним кодом.

Для прикладу з листингу 2.1 буде побудоване дерево на рисунку 2.2.

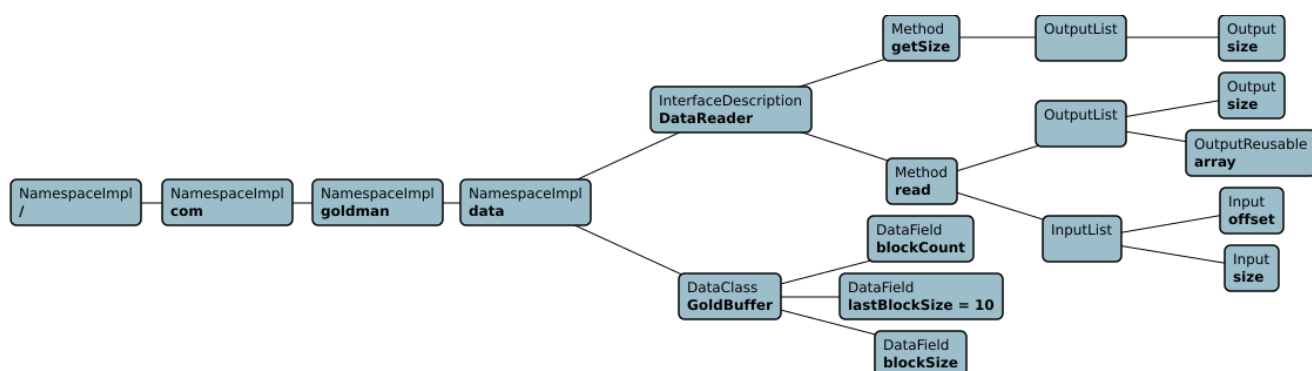


Рисунок 2.2 - Абстрактне синтаксичне дерево для інтерфейсу DataReader та структури GoldBuffer

Як можна побачити дерево будується від кореневого вузла Namespace, яке починає декларувати простори імен. Від нього по ланцюжку ідуть простори імен “com”, “goldman”, “data”. Така організація дає можливість групувати класи, інтерфейси чи структури з одного простору імен в якості дочірніх елементів вузла. У нашому прикладі маємо структуру GoldBuffer та інтерфейс DataReader які задекларовані в просторі імен “com.goldman.data”.

Вузол GoldBuffer має три дочірніх вузла:

- поле даних blockSize;
- поле даних blockCount;
- поле даних lastBlockSize,

як і було задекларовано на рівні метамови.

Вузол DataReader це опис інтерфейсу, в ньому два дочірніх вузла відповідні за методи getSize та read. Метод getSize не має вхідних аргументів, і має лише один результат це відповідно відображено на дереві - вузол OutputList має лише один дочірній Output.

Розглянемо метод read (вузол Method read), в його гілці можна побачити що вхідні аргументи та вихідні результати згруповані по відповідним дочірнім вузлам InputList та OutputList відповідно. Таким чином дерево повністю відповідає тому

що було описано на рівні метамови в лістингу 2.1.

Абстрактне синтаксичне дерево не можна використати для безпосередньої генерації вихідного коду оскільки воно ще занадто загальне та “віддалене” від результуючої мови. Приклад того що може бути різним:

- організація файлів - в C/C++, прийнято декларування робити в .h файлах, а реалізацію в .c/.cpp. В мовах java/kotlin/swift/go декларування та реалізація в одному файлі;

- різна структура файлів в Java/Kotlin простору імен декларується на початку файлу, потім йде блок імпорту. В мові C++ навпаки - в одному файлі може бути задекларовано багато просторів імен. А імпорт взагалі може бути в будь якому місці;

- структура директорій - в Kotlin/Java шлях до файлу на диску має співпадати з простором імен. В інших мовах таке не потрібно.

Тому потрібен наступний етап, на якому абстрактне синтаксичне дерево буде перетворено в дерево розбору.

2.4 Побудова дерева розбору

На цьому етапі система будує дерево розбору з урахуванням особливостей обраної мови, але без врахування стилістичних налаштувань. На рисунках 2.3 та 2.4 наведено вихідне дерево яке будує система для структури GoldBuffer та інтерфейсу DataReader описаних вище в розділах 2.2 та 2.3 для мови Kotlin.

Дерево розбору - це структура даних, яка представляє перетворення абстрактного дерева в дерево, що враховує особливості та правила граматики вихідної мови. В цьому дереві містяться синтаксичні елементи тексту та їх взаємозв'язки згідно з правил граматики.

Кожний вузол у дереві розбору відображує синтаксичний елемент, як то

оператор, вираз, ключове слово чи щось інше. Ребра це відносини між синтаксичними елементами, які визначають ієрархію та порядок вкладеності.

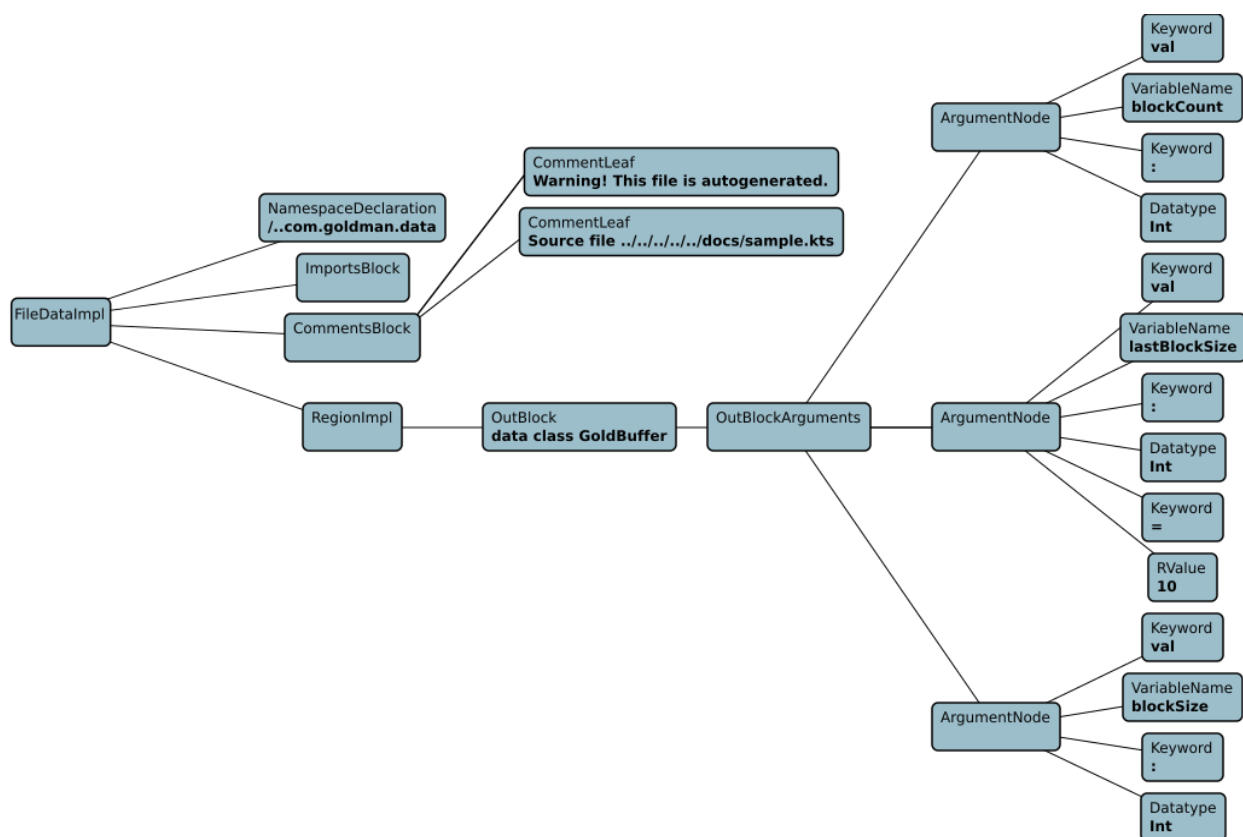


Рисунок 2.3 - дерево розбору для структури GoldBuffer

З рисунку 2.3 можна побачити що на цьому рівні система вже будує файло-орієнтоване дерево, оскільки на вищому рівні іде вузол FileData з вказанням шляху до вихідного файлу. На наступному рівні іде:

- декларування простору імен (NamespaceDeclaration);
- блок імпортів інших класів (ImportsBlock), в нашому випадку він пустий;
- блок з коментарями (CommentsBlock), містить автоматично додані коментарі;
- вузол з описом структури (OutBlock), в свою чергу він містить вхідні аргументи (OutBlockArguments), і далі на нижньому рівні бачимо опис трьох полів даних: blockSize, blockCount, lastBlockSize. Кожне з цих полів описано декількома

вузлами дерева.

На рисунку 2.4 інший приклад - як система формує дерево розбору для інтерфейсу `DataReader` та його методів. Початок також йде з вузла `OutBlock`. Він містить опис потрібний для декларування початку структури (`OutBlock`) і потім декларування двох методів (вузли `Method`). Вузол `Method` може містити:

- вузол `ResultLeaf` - в якому виконаний опис результату метода;
- вузол `InputList` - в якому декларуються вхідні аргументи, які отримує метод.

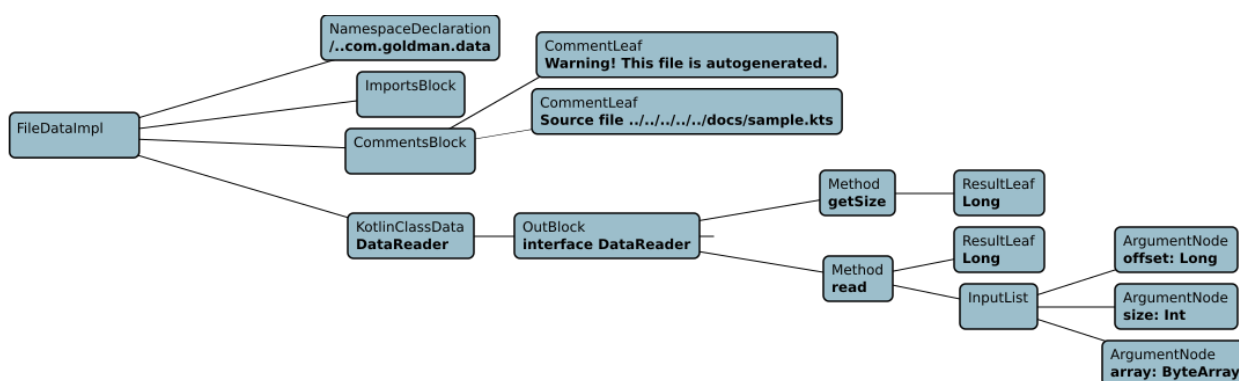


Рисунок 2.4 - Вихідне дерево розбору інтерфейсу `DataReader`

2.5 Стилiстичне дерево

Дерево розбору не враховує можливі налаштування стандарту оформлення коду на мові високого рівня. Для цього потрібен окремий етап трансформації дерева розбору в стилістичне дерево, яке містить елементи оформлення коду, такі як табуляція, коми, переноси рядків, елементи коментування, інші спеціальні символи. Таким чином стилістичне дерево – це дерево розбору доповнено вузлами оформлення коду.

На рисунку 2.5 приклад того, що буде в результаті трансформації дерева розбору структури `GoldBuffer` в стилістичне дерево.

Як можна побачити дерево у порівнянні з рис. 2.3 не змінило своєї структури, а лише доповнилось додатковими вузлами, які відповідають саме за оформлення вихідного файлу.

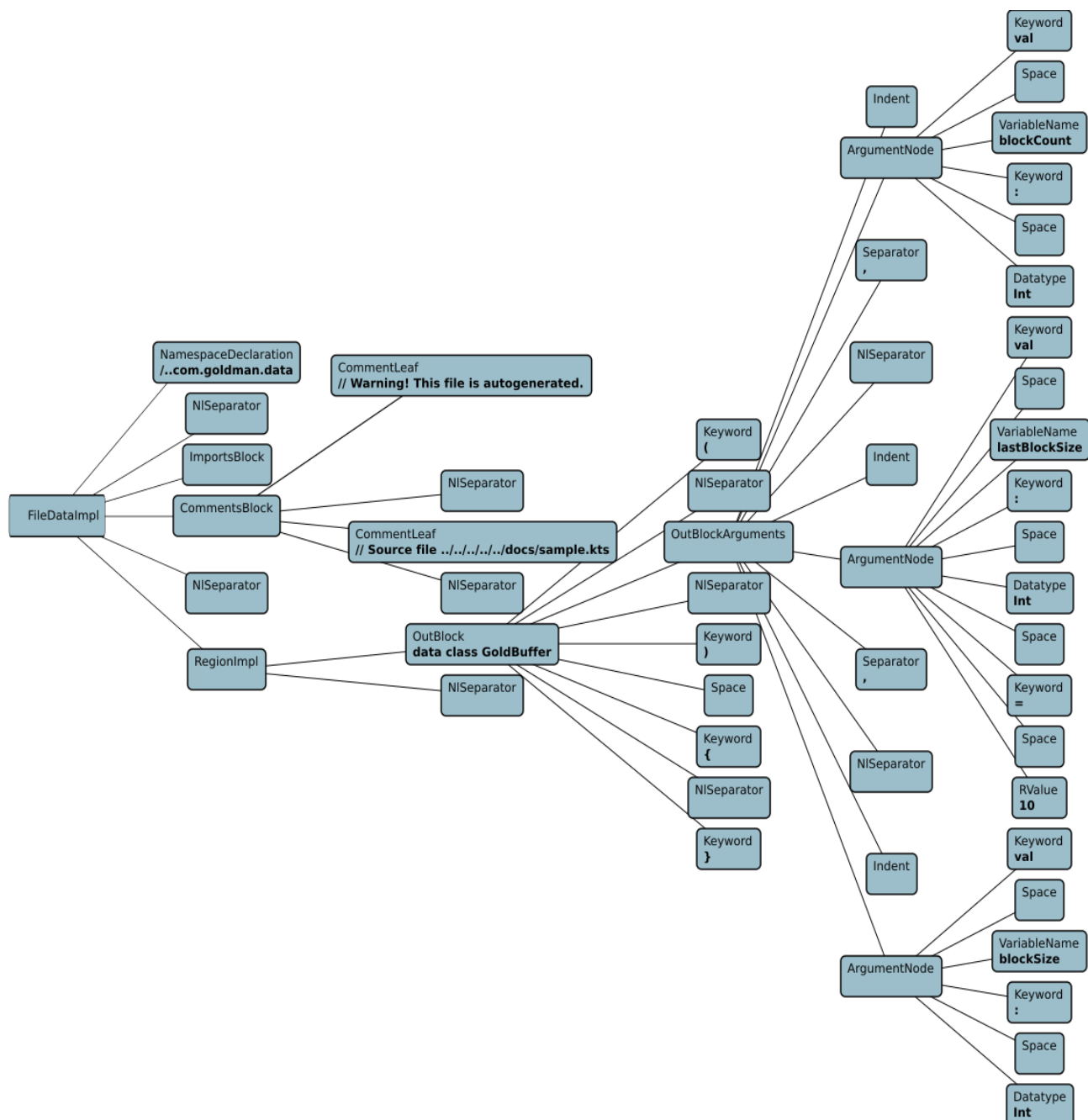


Рисунок 2.5 - Стилiстичне дерево для структури GoldBuffer

Далі розглянемо ці вузли:

- `nlSeparator` - відповідає за запис у вихідний файл символу переносу рядку;
- `Space` - звичайний символ пробілу - " ", приклад використання - відступ між типом даних та назвою змінної;
- `Indent` - символ табуляції, використовується для відступів у вкладених блоках коду;
- `Keyword` - вузли з символами "(", ")" для оформлення аргументів функцій, методів та конструкторів;
- крім того система на цьому етапі розставляє символи дужки для оформлення логічних блоків у місцях, які задекларовані стандартом оформлення коду.

Розглянемо два найбільш поширених прикладів оформлення логічних блоків в C-подібних мовах:

- `Kernighan & Ritchie` [26] стиль декларує розставлення фігурних дужок наступним чином - відкриваюча залишається на тому самому рядку, а закриваюча переноситься на окремий рядок, приклад в лістингу 2.2.

Лістинг 2.2 - `Kernighan & Ritchie` стандарт оформлення коду

```
1. while (x == y) {
2.     something();
3.     something_else();
4. }
```

- `BSD` стиль декларує розставлення фігурних дужок кожна на окремому рядку, приклад в лістингу 2.3.

Лістинг 2.3 - `BSD` стандарт оформлення коду

```
1. while (x == y)
2. {
3.     something();
4.     something_else();
5. }
```

На рисунку 2.6 зображено результат трансформації дерева розбору інтерфейсу `DataReader` в стилістичне дерево. Так само як і в попередньому випадку, структурних змін немає, лише додалися вузли для оформлення коду.

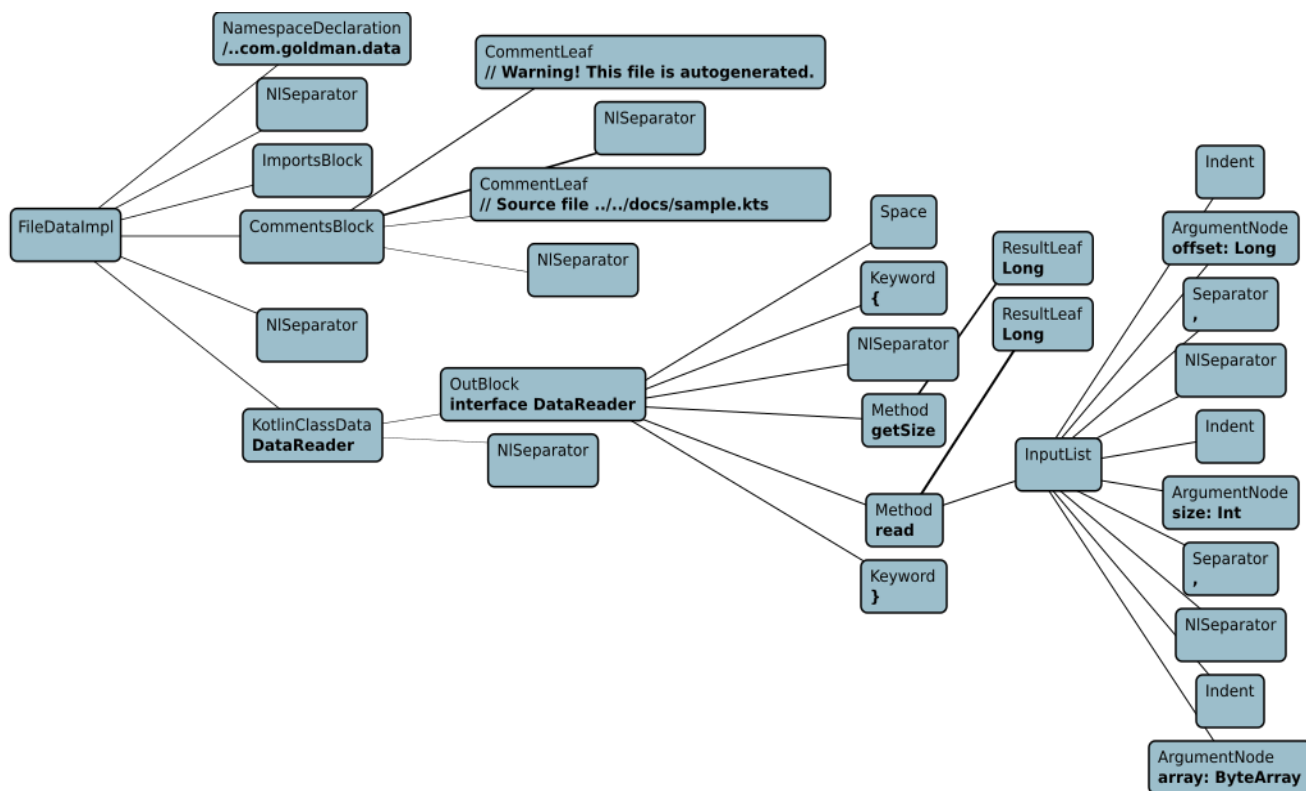


Рисунок 2.6 - Стилiстичне дерево для iнтерфейсу `DataReader`

2.6 Запис в файл

За стилістичним деревом система пише вихідні файли на обраній мові. В подальшому ці файли можна буде використовувати в проєктах користувача системи, тобто це і є фінальний результат системи кодогенерації.

Результуючі файли для прикладів, наведених у попередніх розділах, представлено у лістингах 2.4 та 2.5.

Лістинг 2.4 - Згенерований код структури GoldBuffer

```

1. package com.goldman.data
2. // Warning! This file is autogenerated.
3. // Source file ../../../../../../test/dataclass.kts
4.
5. data class GoldBuffer(
6.     val blockCount: Int,
7.     val lastBlockSize: Int = 10,
8.     val blockSize: Int,
9. ) {
10. }
```

Лістинг 2.5 - Згенерований код інтерфейсу DataReader

```

1. package com.goldman.data
2. // Warning! This file is autogenerated.
3. // Source file ../../../../../../test/interface.kts
4.
5. interface DataReader {
6.     fun getSize(): Long
7.     fun read(offset: Long, size : Int, array : ByteArray) :
8.     Long
9. }
```

Розглянемо більш детально як виконується запис в файл на прикладі GoldBuffer структури на рисунку 2.7. На ньому можна побачити що в файлі є три головні частини та відповідні їм вузли - декларування простору імен, блок коментарів файлу, опис класу даних.

Вузол NamespaceDeclaration записується на мові Kotlin [12-14] як один рядок. Цей вузол немає нащадків, тому на цьому його обробка системою закінчується.

Вузол CommentsBlock - позначає що в файл має бути записаний один або більше рядків з коментарями. Кожний рядок це окремий нащадок такого вузла. На рисунку 2.7 у цього блока два дочірніх вузла і два відповідних рядка з коментарями.

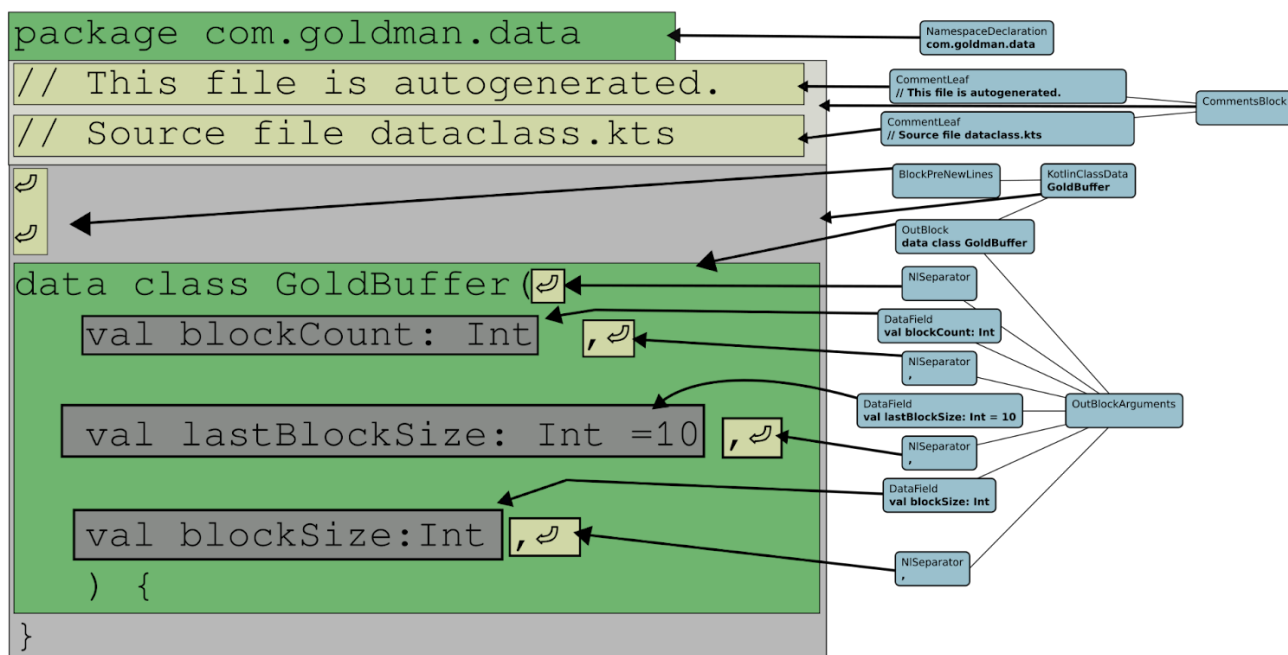


Рисунок 2.7 - Логіка перетворення дерева розбору в вихідний код

Наступний значний вузол - `KotlinClassData`. Він відповідає за описи Kotlin класів взагалі. Дочірніми до цього вузла можуть бути вузли з коментарями до класу, анотації класу, елементи стилістичного оформлення коду і таке інше. В нашому випадку в файл потрібно записати відступи, за це відповідає вузол `BlockPreNewLines`, який додає заздалегідь зконфігуровану кількість порожніх рядків.

Наступний дочірній вузол це `OutBlock` який відповідає за тіло класу/структури чи інтерфейсу. В цьому вузлі вказується що саме треба записати в файл, в нашому випадку це “`data class GoldBuffer`”. І далі йде опис полів структури у дочірньому вузлу `OutBlockArguments` за допомогою вузлів:

- `DataField` - який містить повний опис - ім'я поля, тип, значення за замовченням;
- `NISeparator` - такий вузол призводить до того що в вихідний файл записується символ закінчення рядка;
- `Space` - в вихідний файл буде записано один пробіл;

- Indent - в вихідний файл буде записано або символ табуляції або певну кількість пробілів, це визначається в налаштуваннях проєкту.

2.7 Висновки до розділу 2

Таким чином, була розроблена та запропонована архітектура системи кодогенерації. Для подальшого масштабування вона має розділення на різні етапи обробки вхідних описів.

Проміжні результати між етапами зберігаються у вигляді дерев - абстрактне синтаксичне дерево, дерево розбору, стилістичне дерево.

3 РЕАЛІЗАЦІЯ СИСТЕМИ АВТОМАТИЧНОЇ ГЕНЕРАЦІЇ КОДУ

3.1 Формати даних

Формат даних визначає, які типи даних можуть бути використані, як вони структуровані, та як можна взаємодіяти з цими даними, тобто це конкретний спосіб представлення або організації даних. Формати даних грають важливу роль в обміні та зберіганні інформації, вони встановлюють, як дані мають бути представлені та передані. Вибір певного формату залежить від потреб конкретного застосунку та вимог до обробки даних.

JSON (JavaScript Object Notation) [27] та XML (eXtensible Markup Language) [28] - це формати даних, які визначають правила для представлення структурованих даних у текстовому вигляді.

Формат JSON використовує пари "ключ-значення". Легко читається та редагується людиною в звичайних текстових редакторах, без використання

складних додаткових інструментів, тому часто використовується для обміну даними між мовами програмування, клієнтом та сервером тощо. JSON підтримує складні структури даних і може бути вкладений. Дані подаються у вигляді об'єктів, масивів, чисел, рядків, логічних значень та null.

Формат XML використовує теги, які можуть мати атрибути, для визначення елементів та їхньої структури. XML менш зручний для сприйняття через більш об'ємний синтаксис, але дозволяє створювати структуровані дані. Легко розширюється та підтримує ієрархічну структуру. Використовується в різних галузях, включаючи обмін даними між системами та зберігання конфігураційних файлів.

Таким чином, для нашої системи було обрано формат JSON - для простої конфігурації проєкту та формат XML - для збереження дерева на диску.

3.2 Програмний інструментарій

Система кодогенерації розроблена на мові Kotlin [9, 10] з використанням середовища розробки IntelliJ IDEA [29].

Kotlin — статично типізована, об'єктно-орієнтована мова програмування, яка працює поверх JVM. Може застосовуватися для розробки застосунків у багатьох проблемних областях. Розробляється компанією JetBrains.

Kotlin зазнав впливу багатьох мов: Java, Scala, Groovy, C#, JavaScript, Swift та дозволяє писати програми як в об'єктно-орієнтованому, так і в функціональному стилі. Мова Kotlin призначена для розробок промислового програмного забезпечення в багатьох областях. Kotlin може компілюватися і для JavaScript, внаслідок чого є серйозним варіантом для вибору при розробці клієнтських веб-застосунків. Також може компілюватися за допомогою технології Kotlin/Native безпосередньо у власні двійкові файли, які функціонують у системах за відсутності

віртуальної машини.

IntelliJ IDEA - це інтегроване середовище розробки (IDE) для багатьох мов програмування таких як Java, Kotlin, Groovy, Scala та інших. Створено компанією JetBrains та надає різноманітні інструменти для зручності розробки починаючи від редактора коду з підсвічуванням синтаксису та автодоповненням до потужних засобів рефакторингу та відлагодження. Перевагами IntelliJ IDEA є розумні підказки, автоматичне генерування коду, інтеграція з системами контролю версій, підтримка фреймворків та бібліотек, а також можливість роботи з численними технологіями, такими як Spring, Android та інші.

Проект може бути сконфігурований двома методами:

- з використанням опису в форматі JSON [27];
- з використанням скрипту на мові Groovy [30].

Groovy [30] - високорівнева мова програмування для середи Java, використовується у випадках коли налаштування проекту потрібно динамічно змінювати. Має більшу гнучкість у порівнянні з JSON [27], проте також має більший поріг входження, оскільки потрібно знати цю мову програмування.

В якості архітектурного підходу для нашої системи було обрано гексагональну архітектуру (Clean Architecture [31]), яка відповідає нашим вимогам та цілям як найкраще.

Гексагональна архітектура – це архітектура, побудована на використанні слабо пов'язаних компонентів. Тобто усі зв'язки між компонентами реалізовані за допомогою адаптерів та інтерфейсів. Використання такої архітектури робить компоненти легко замінними на будь-якому рівні, та полегшує тестування логіки. Цей підхід допомагає забезпечити високу якість коду, зручність підтримки та гнучкість системи для майбутніх змін та розширень.

Use Case сценарії в гексагональній архітектурі - це абстраговані та ізольовані частина, яка виконують лише одну певну функцію. Use Case сценарії мають притримуватись певних концепцій, розглянемо детальніше ці концепції.

Абстрагованість - кожний сценарій має окрему декларацію та реалізацію,

таким чином це полегшує можливість тестування та додає можливості гнучкого розширення системи. Для прикладу у лістингу 3.1 наведено опис інтерфейсу завантаження проєкту. Як можна побачити код, що використовує цей сценарій, не знає деталі того, яким чином відбувається завантаження налаштувань проєкту.

Лістинг 3.1 - Інтерфейс завантаження конфігурації проєкту

```
1. interface LoadProjectUseCase {
2.     operator fun invoke(projectPath: String): Project
3. }
```

Далі у лістингу 3.2 наведено одну з можливих імплементацій інтерфейсу LoadProjectUseCase, яка оперує локальними файлами в JSON форматі. Але за рахунок абстрактності можуть бути і інші імплементації, як-то завантаження Groovy скрипту з локальної мережі, або навіть завантаження JSON файлу конфігурації з будь-якого вузла в мережі Інтернет. Такий підхід додає гнучкості та можливості горизонтального масштабування логіки системи в подальшому для запуску в мережі чи в хмарі.

Лістинг 3.2 - Імплементація інтерфейсу завантаження проєкту яка оперує локальними файлами в JSON форматі

```
1. class LoadProjectUseCaseImpl : LoadProjectUseCase {
2.     operator fun invoke(projectPath: String): Project {
3.         val mapper =
4.             ObjectMapper().enable(SerializationFeature.INDENT_OUTPUT)
5.         val module = SimpleModule()
6.         module.addSerializer(DataType::class.java,
7.             DataTypeSerializer())
8.         module.addSerializer(DataValue::class.java,
9.             DataValueSerializer())
10.        mapper.registerModule(module)
11.        // load project file
12.        val projectJson = FileInputStream(projectPath)
13.        val project: Project =
14.            mapper.readValue(projectJson, Project::class.java)
15.        projectJson.close()
16.        return project
17.    }
18. }
```

Композиція з сценаріями нижнього рівня. Сценарій може використовувати інші сценарії нижнього рівня.

З лістингу 3.3 можна побачити, що сценарій отримує на ініціалізації чотири більш прості сценарії:

- сценарій LoadAstTreeUseCase для завантаження абстрактного дерева;
- сценарій LoadProjectUseCase для завантаження налаштувань проєкту;
- сценарій StoreOutTreeUseCase для збереження результуючого дерева розбору;
- сценарій TransformInTreeToOutTreeUseCase для перетворення абстрактного дерева в дерево розбору.

Лістинг 3.3 - Реалізація сценарію побудови дерева розбору

```

1.  class BuildOutTreeUseCaseImpl @Inject constructor(
2.  private val loadInTreeUseCase : LoadAstTreeUseCase,
3.  private val getProjectUseCase : LoadProjectUseCase,
4.  private val storeOutTreeUseCase : StoreOutTreeUseCase,
5.  private val transformInTreeToOutTreeUseCase :
TransformInTreeToOutTreeUseCase,
6.  ): BuildOutTreeUseCase {
7.  override operator fun invoke(projectFile: String,
inTreeFile: String, outTreeFile: String, target: Target) {
8.  val generatorsRepo =
GeneratorsRepo(getProjectUseCase(projectFile))
9.  val tree = loadInTreeUseCase(inTreeFile)
10. val outTree = transformInTreeToOutTreeUseCase(tree as
Node, generatorsRepo.get(target))
11. storeOutTreeUseCase(outTreeFile, outTree)
12. }
13. }
```

Не зберігає стан (stateless) - сценарії не мають зберігати будь які стани. Тобто будь яке використання сценарію не залежить від попереднього стану об'єкту. За рахунок цього досягається легка горизонтальна масштабованість. Системі не важливо де буде виконуватись цей сценарій - на локальній машині, в мережі чи в хмарі.

3.3 Налаштування проєкту

Налаштування проєкту можуть бути збережені в JSON файлі або Groovy скрипті. Розглянемо простіший варіант з використанням JSON в лістингу 3.4.

Лістинг 3.4 - Конфігураційний файл проєкту

```
1.  {
2.    "files": [
3.      "./classes.kts",
4.      "./interfaces.kts",
5.    ],
6.    "targets" : ["Kotlin", "Cxx", "Swift"],
7.    "addAutogeneratedCommentNotification" : true,
8.    "codeStyle" : {
9.      "newLinesBeforeClass" : 2,
10.     "tabSize" : 4
11.   },
12.   "outputFolder" : "./generated/"
13. }
```

В конфігурації проєкту можна задати:

- список `files` - вказує які файли на метамові потрібно обробити (рядки 3-4 лістинг 3.4);
- список `targets` - на яких мовах потрібно створити згенерований код (рядок 6 лістинг 3.4);
- `codeStyle` - стилістичні налаштування (рядки 8-11 лістинг 3.4);
- поле `outputFolder` - шлях до папки в яку потрібно записати результати (рядок 12 лістинг 3.4).

3.4 Побудова AST-дерева

Розглянемо як система будує абстрактні дерева для проєкту. Цей сценарій на вхід отримує шлях до файлу з конфігурацією проєкту. Блок-схема сценарію

побудови AST-дерева представлено на рисунку 3.1.

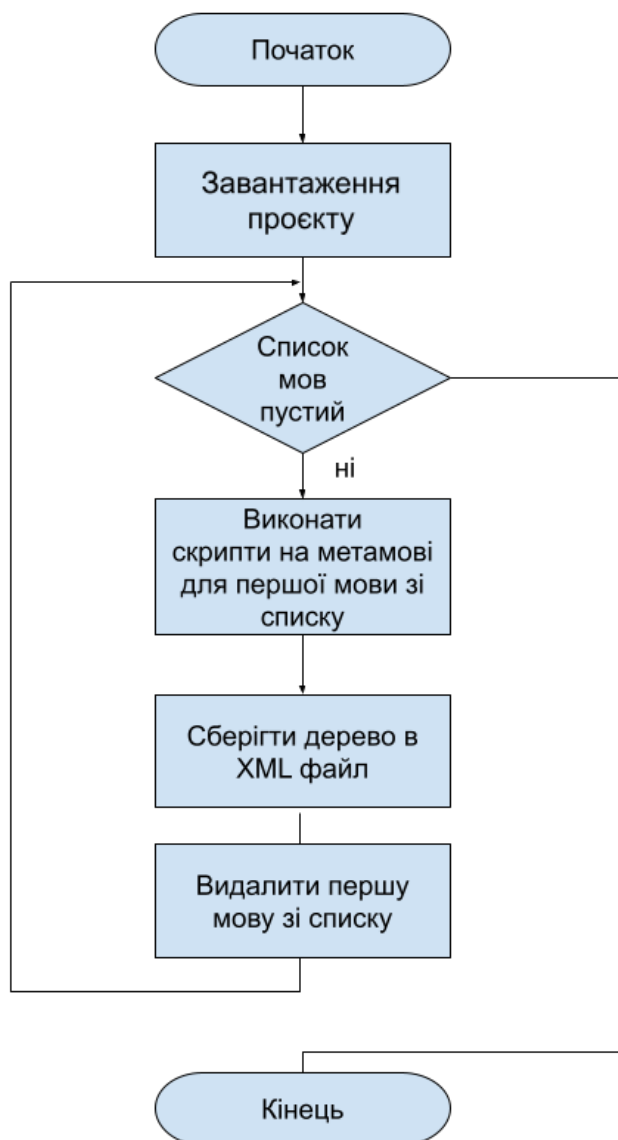


Рисунок 3.1 - Блок-схема сценарію побудови AST-дерева

Для початку роботи сценарію потрібно завантажити налаштування проєкту.

Наступний крок - ітеративне проходження списку мов, які вказані в конфігурації проєкту в налаштуваннях “targets” (лістинг 3.4 рядок 6).

Для кожної обраної мови буде виконано наступні кроки:

- завантажено та виконані скрипти на метамові;
- збережено результати виконання скриптів у вигляді XML дерева в зовнішні файли;

- мову для якої виконано всі кроки видаляємо з списку обробки.

Лістинг 3.5 - Абстрактне синтаксичне дерево в XML - форматі

```

1.  <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2.  <AstTree name="/">
3.      <NamespaceImpl name="">
4.          <NamespaceImpl name="com">
5.              <NamespaceImpl name="goldman">
6.                  <NamespaceImpl name="data">
7.                      <InterfaceDescription
baseFolder="./generated/" name="DataReader"
outFile="DataReader" srcFile="./docs/sample.kts">
8.                          <Method name="getSize">
9.                              <OutputList name="">
10.                                  <Output name="size"
type="int64"/>
11.                                  </OutputList>
12.                              </Method>
13.                              <Method name="read">
14.                                  <OutputList name="">
15.                                      <Output name="size"
type="int64"/>
16.                                      <OutputReusable
name="array" type="arrayOf_uint8"/>
17.                                  </OutputList>
18.                                  <InputList name="">
19.                                      <Input name="offset"
type="int64"/>
20.                                      <Input name="size"
type="int32"/>
21.                                  </InputList>
22.                              </Method>
23.                          </InterfaceDescription>
24.                          <DataClass baseFolder="./generated/"
name="GoldBuffer" outFile="GoldBuffer"
srcFile="/home/mrco/Documents/git/codegen/./docs/sample.kt
s">
25.                              <DataField name="blockCount"
type="int32"/>
26.                              <DataField name="lastBlockSize"
type="int32" value="10"/>
27.                              <DataField name="blockSize"
type="int32"/>
28.                          </DataClass>
29.                      </NamespaceImpl>
30.                  </NamespaceImpl>
31.              </NamespaceImpl>
32.          </NamespaceImpl>
33.  </AstTree>

```

Цей цикл повторюється для всіх вказаних в налаштуваннях мовах. Після чого алгоритм сценарію завершується. В результаті отримуємо XML - файл з деревом наступного вигляду як у лістингу 3.5.

3.5 Побудова дерева розбору

Розглянемо сценарій побудови дерева розбору для проєкту. Цей сценарій на вхід отримує шлях до абстрактного дерева, шлях до конфігурації проєкту та шлях куди зберігати дерево розбору. Блок-схема сценарію побудови дерева розбору представлено на рисунку 3.2.

На початку роботи сценарію потрібно завантажити налаштування проєкту.

Другий крок завантажити абстрактне дерево з вказаного файлу для вказаної в аргументах мови.

На третьому кроці аналізуємо вхідне дерево, та будуємо список результуючих файлів за правилами прийнятими в обраній мові програмування.

Четвертим кроком буде обхід абстрактного дерева, та трансляція ключових вузлів у відповідні структури обраної мови. Правила трансляції запрограмовані в коді для кожної мови та кожного виду блоків окремо. Приклад трансляції структури даних наведено в розділі 3.6.

На останньому кроці дерево зберігається на диску у виді XML файлу.



Рисунок 3.2 - Блок-схема сценарію побудови дерева розбору

3.6 Приклад трансляції вузлу `DataClass` на мову `Kotlin`

Розглянемо приклад трансляції наступного блоку (лістинг 3.6), в якому записана частина абстрактного дерева, що описує структуру даних `GoldBuffer` на

мову високого рівня Kotlin.

Лістинг 3.6 - Опис структури даних в абстрактному синтаксичному дереві в XML форматі

```

1.   <DataClass baseFolder="./generated/" name="GoldBuffer"
      outFile="GoldBuffer">
2.       <DataField name="blockCount" type="int32"/>
3.       <DataField name="lastBlockSize" type="int32"
      value="10"/>
4.       <DataField name="blockSize" type="int32"/>
5.   </DataClass>

```

Блок-схема алгоритму трансляції вузла DataClass на мову Kotlin представлено на рисунку 3.3.

Перший крок - знайти в дереві розбору вузол що відповідає потрібному файлу (або файлам для тих мов що мають окремі файли для опису та реалізації, як-то C++).

В мові Kotlin структури прийнято описувати за допомогою класу даних. Тому на другому кроці в вузол файлу додається новий дочірній вузол OutBlock з рядком даних “data class \$name”.

Третій крок - в OutBlock додаємо дочірній вузол OutBlockArguments, в якому будуть створені дочірні вузли нижнього рівня.

Крок четвертий - створюємо дочірні вузли в OutBlockArguments вузли Keyword(“val”), VarName(leaf.name), KeyWorrd(“:”) та DataType(leaf.type) у відповідному порядку для кожного елементу DataField з вхідного дерева.

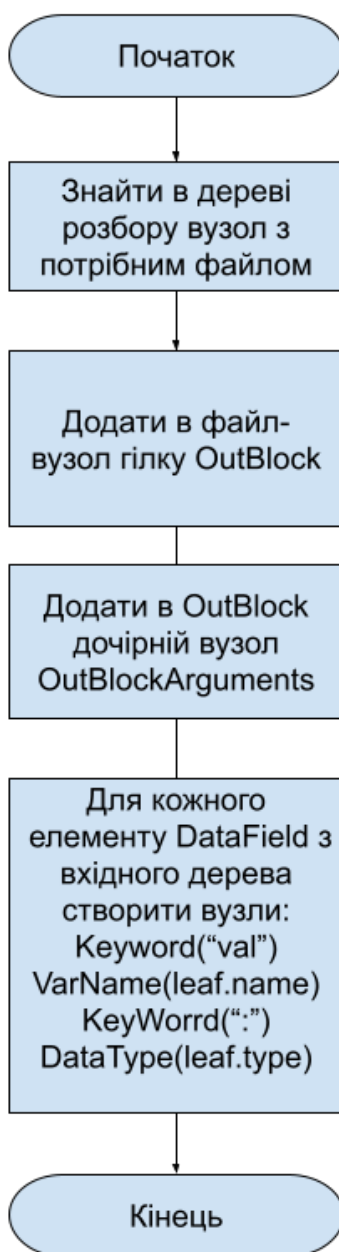


Рисунок 3.3 - Блок-схема алгоритму трансляції вузла DataClass на мову Kotlin

3.7 Приклад трансляції вузлу DataClass на мову C++

Розглянемо приклад трансляції вже наведеного блоку (лістинг 3.6), в якому записана частина абстрактного дерева, на мову високого рівня C++.

Блок-схема алгоритму трансляції вузла DataClass на мову C++ представлено на рисунку 3.4.

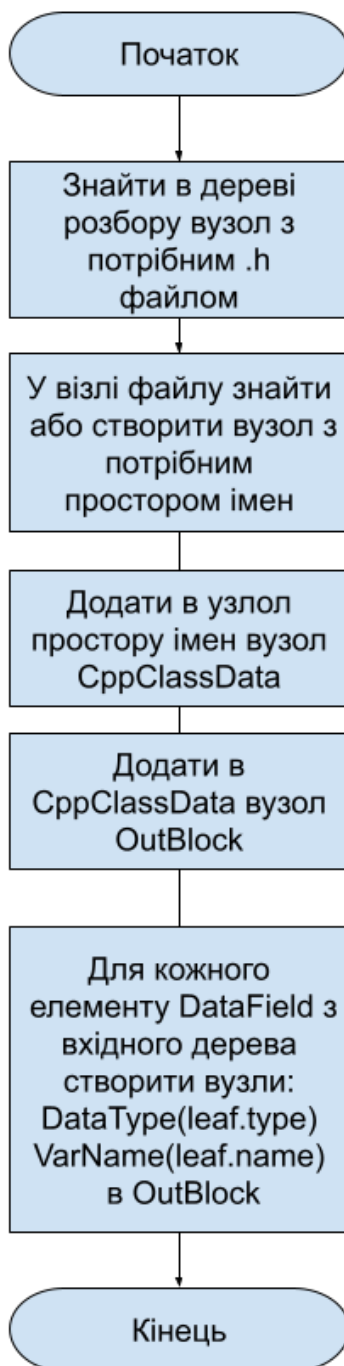


Рисунок 3.4 - Блок-схема алгоритму трансляції вузла DataClass на мову C++

В C++ декларування структур потрібно робити в окремо від реалізації в

header файлах. Тому на першому кроку алгоритм шукає в дереві розбору вузол, що відповідає потрібному header файлу.

В C++ в одному файлі можна змішувати різні простори імен, на відміну від Java/Kotlin. Тож на другому кроці шукаємо відповідний вузол з потрібним простором імен, якщо такого ще немає, то створюємо його.

Третій крок - додаємо вузол `CppClassData` - це загальне декларування класів та структур.

Четвертий крок - в `CppClassData` додаємо коментарі, якщо вони були задані, та дочірній `OutBlock` вузол.

Крок п'ятий - створюємо дочірні вузли в `OutBlock` - вузли `DataType(leaf.type)` та `VarName(leaf.name)` у відповідному порядку для кожного елементу `DataField` з вхідного дерева.

3.8 Побудова стилістичного дерева

Розглянемо сценарій побудови стилістичного дерева розбору для проєкту на рисунку 3.5.

На початку роботи сценарію потрібно завантажити налаштування проєкту.

На другому кроці відбувається завантаження дерева розбору з вказаного файлу для вказаної в аргументах мови.

На третьому кроці робимо обхід дерева розбору та створюємо копію дерева з додаванням стилістичних елементів. Єдиного алгоритму по додавання стилістичних елементів немає, тому система кодогенерації на цьому кроці приймає рішення в залежності від типу ключового вузла та обраної мови.

На останньому кроці дерево зберігається на диску у вигляді XML файлу.

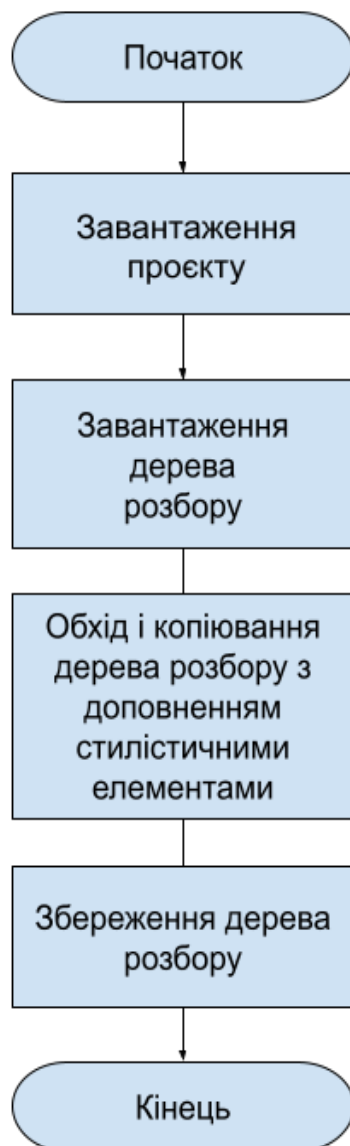


Рисунок 3.5 - Блок-схема сценарію побудови дерева розбору

3.9 Запис в файл

Останній сценарій системи кодогенерації - сценарій запису в файл, алгоритм наведено на рисунку 3.6.



Рисунок 3.6 - Блок-схема сценарію запису в файл

На початку роботи сценарію потрібно завантажити налаштування проекту.

На другому кроці відбувається завантаження стилістичного дерева з вказаного файлу для вказаної в аргументах мови.

На останньому кроці робимо обхід стилістичного дерева та записуємо кожний вузол у відповідний вихідний файл.

У вихідному файлу в результаті отримуємо згенерований програмний код.

3.10 Висновки до розділу 3

Таким чином було реалізовано систему автоматичної генерації коду. Система написана на мові Kotlin, та використовує XML формат для передачі проміжних результатів між різними етапами виконання генерації. Були розроблені сценарії для побудови абстрактного синтаксичного дерева, трансляції його в дерево розбору, сценарії для додавання стилістичної інформації та запису в вихідні файли згенерованого коду. Для пояснення роботи системи наведені та розібрані приклади роботи трансляторів структур даних на мови C++ та Kotlin. Система дозволяє генерувати структури та інтерфейси (примітиви) з мінімальним втручанням людини.

4 ВПРОВАДЖЕННЯ СИСТЕМИ

4.1 Використані програмні засоби

Репозиторій з кодом системи кодогенерації викладено на GitHub [11] за адресою <https://github.com/vshcryabets/codegen>.

GitHub [11] — один з найбільших веб-сервісів для спільної розробки програмного забезпечення. Існують безкоштовні та платні тарифні плани користування сайтом. GitHub базується на системі керування версіями Git [12] і розроблений на Ruby on Rails і Erlang компанією GitHub, Inc (раніше Logical Awesome). Сервіс безкоштовний для проєктів з відкритим вихідним кодом, з наданням користувачам усіх своїх можливостей (включаючи SSL), а для окремих індивідуальних проєктів пропонуються різні платні тарифні плани.

Система керування версіями Git [12] є однією з найефективніших, надійних та високопродуктивних. Надає гнучкі засоби нелінійної розробки, що базуються на

відгалуженні і злитті гілок. Для забезпечення цілісності історії та стійкості до змін заднім числом використовуються криптографічні методи, також можлива прив'язка цифрових підписів розробників до тегів і комітів.

На рисунку 4.1 наведено інтерфейс репозиторію.

The screenshot displays the GitHub interface for the repository 'codegen' by user 'vshcryabets'. The repository is public and has 4 branches, 0 tags, and 214 commits. The main branch is selected. A table lists the repository's files and folders, including source code, documentation, and build files. To the right, the 'About' section describes the project as 'code generation experiments' with a GPL-3.0 license. The 'Releases' section indicates no releases are published. The 'Packages' section shows one package, 'ce.cgen-lib'. The 'Languages' section shows a bar chart with Kotlin at 86.9%, C++ at 12.1%, and Other at 1.0%. The 'Suggested workflows' section is also visible.

File/Folder	Commit Message	Timestamp
.run	Restore doc generation	last month
bin	Build fat jar	8 months ago
buildSrc	Merge branch 'main' into feature/nn-parser	last month
cgen-console	Run samples from README	last month
cgen-example	Merge branch 'main' into feature/nn-parser	last month
cgen-lib	Restore doc generation	last month
cgen-nnparser	Add comments test to Tokenizer logic	2 months ago
cgen-parser	More tests	2 months ago
docs	Restore doc generation	last month
gradle/wrapper	Update gradle	3 months ago
test	Work over linearization	2 months ago
.gitignore	Update tokenizer logic	9 months ago
LICENSE	Initial commit	last year
README.md	Run samples from README	last month
TODO.txt	Working over C++ parser	8 months ago
build.gradle.kts	Update kotlin	5 months ago
gradle.properties	Coroutines dictionaries loader	2 months ago

Рисунок 4.1 – Репозиторій з вихідним кодом системи автоматичної генерації коду

Код системи є в публічному доступі. Кожен може використовувати його в

своїх проєктах за ліцензією GPL-3.0 (GNU General Public License version 3) [32]. Ліцензія GPL-3 є однією з найпопулярніших відкритих ліцензій програмного забезпечення. Вона була випущена Free Software Foundation у 2007 році і є третьою версією серії GPL.

Основні положення GPL-3 включають:

- вільність використання: ліцензія гарантує вільне використання програми для будь-яких цілей без обмежень;
- вільність розповсюдження: користувач може вільно розповсюджувати копії програми, змінювати її і розповсюджувати змінену версію;
- вільність доступу до вихідного коду: ліцензія вимагає, щоб користувачам надавалось доступ до вихідного коду програми або будь-яких змін, внесених до неї;
- збереження ліцензії: любі зміни чи похідні роботи також повинні бути ліцензовані на умовах GPL-3, забезпечуючи такий самий вільний характер використання;
- відсутність обмежень технологічних заходів: ліцензія забороняє використання технічних заходів (наприклад, технічних обмежень або шифрування) для обмеження прав, наданих користувачам ліцензії;
- авторське право: авторство і вказівка ліцензії повинні залишатися непорушними в усіх копіях і похідних роботах;
- відсутність гарантій: розповсюджувачі і розробники відмовляються від надання гарантій щодо програми, що постачається.

Також кожен користувач GitHub може долучитись до розробки проєкту - створюючи звіти з помилками, чи додаючи виправлення або новий функціонал через систему пулл-реквестів (пропозиції зміни коду у репозиторії).

Робота над системою продовжується, репозиторій постійно оновлюється, виправляються помилки та система поліпшується.

4.2 Опис роботи системи кодогенерації та можливостей використання

Система автоматичної генерації коду на даний час використовується для генерації коду у виробничому процесі для суміщення старої та нової частини комерційного продукту, написаного на різних мовах, що допомагає спростити розробку та зекономити час розробників, що засвідчує акт про впровадження у виробничий процес ТОВ “ЛОГІТЕК ЮКРЕЙН” результатів науково-дослідної магістерської роботи, який наведено у додатку А.

Система автоматичної генерації коду дозволяє генерувати загальні інтерфейси, блоки констант, перерахування та структури на мовах Kotlin [9, 10], C/C++ [4-6], Java, Swift. Планується додати підтримку інших мов високого рівня, зокрема Rust та Python.

Ведеться робота з удосконалення трансляторів для генерації класів, методів та логіки. Також планується розробка методики взаємодії та поширення коду між командою розробників за допомогою інструментарію git.

Для побудови абстрактного синтаксичного дерева приведемо конфігурацію запуску у середовищі IntelliJ Idea, рисунок 4.2.

У конфігурації запуску вказано шлях до:

- виконуючого класу, наразі це “`ce.entrypoints.PrepareAstTreeKt`”;
- файлу конфігурації проєкту (на прикладі це “`./test/p2.json`”);
- робочої директорії системи.

Після запуску цього сценарію будується абстрактне синтаксичне дерево. При цьому система видасть повідомлення про успішне завершення процесу, лістинг 4.1.

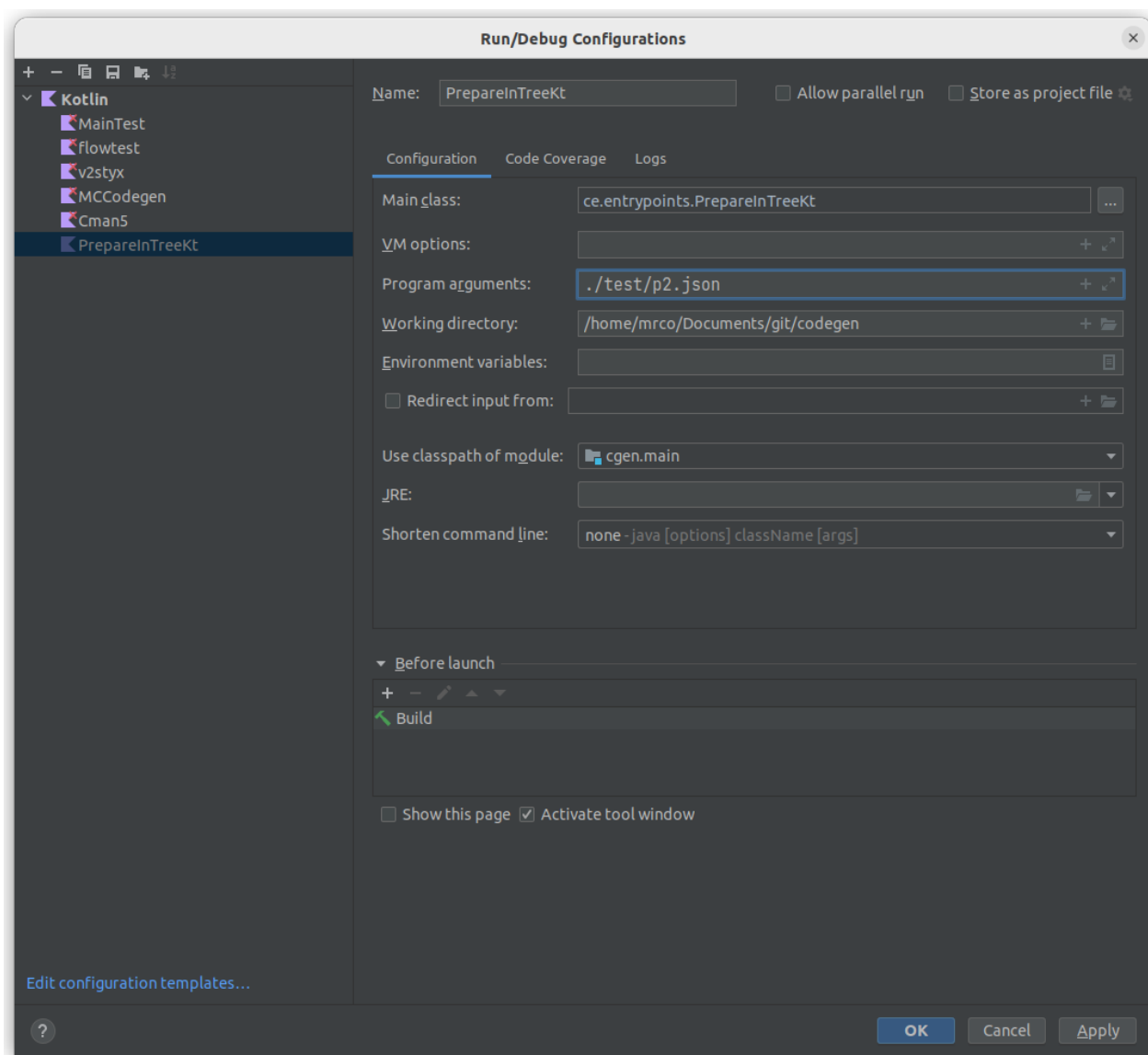


Рисунок 4.2 - Конфігурація запуску побудови абстрактного синтаксичного дерева

Лістинг 4.1 - Приклад виводу системи на етапі побудови абстрактного синтаксичного дерева

```

1. Processing Project(files=[./test/unit_test.kts],
   outputFolder=./generated/,
   codeStyle=CodeStyle(newLinesBeforeClass=2, tabSize=8),
   targets=[Kotlin], addAutogeneratedCommentNotification=true)
2. Target Kotlin
3. Processing ./test/ unit_test.kts
4. Block NewConstantsBlock = NewConstantsBlock

```

Для запуску створення дерева розбору (рисунок 4.3) потрібно в аргументах вказати:

- виконуючий клас, наразі це “ce.entrypoints.BuildOutTreeKt”;
- шлях до файлу с абстрактним синтаксичним деревом;
- шлях до файлу проєкту;
- шлях до результуючого файлу.

Після запуску етапу будується дерево розбору. При цьому система видасть повідомлення про успішне завершення процесу, лістинг 4.2.

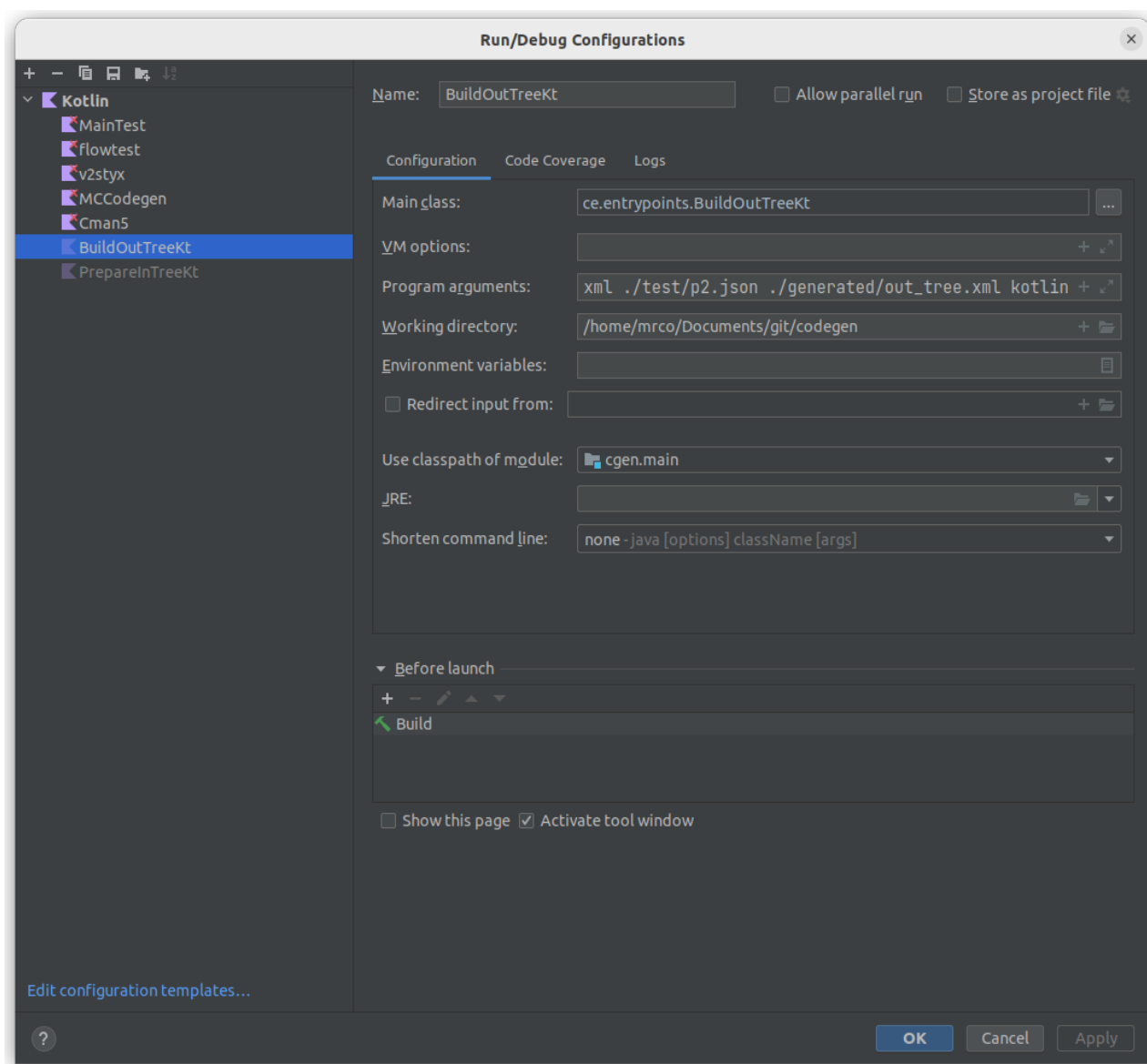


Рисунок 4.3 - Конфігурація запуску побудови дерева розбору

Лістинг 4.2 - Приклад виводу системи на етапі побудови дерева розбору

```

1.   Translating NewConstantsBlock
2.   Translating NewConstantsBlock2
3.   Translating NewConstantsBlock3
4.
5.   Process finished with exit code 0

```

Для запуску створення стилістичного дерева розбору (рисунок 4.4) потрібно в аргументах вказати:

- виконуючий клас, наразі це “ce.entrypoints.BuildCodeStyleTreeKt”;
- шлях до файлу с деревом розбору;
- шлях до файлу проєкту;
- шлях до результуючого файлу з стилістичним деревом.

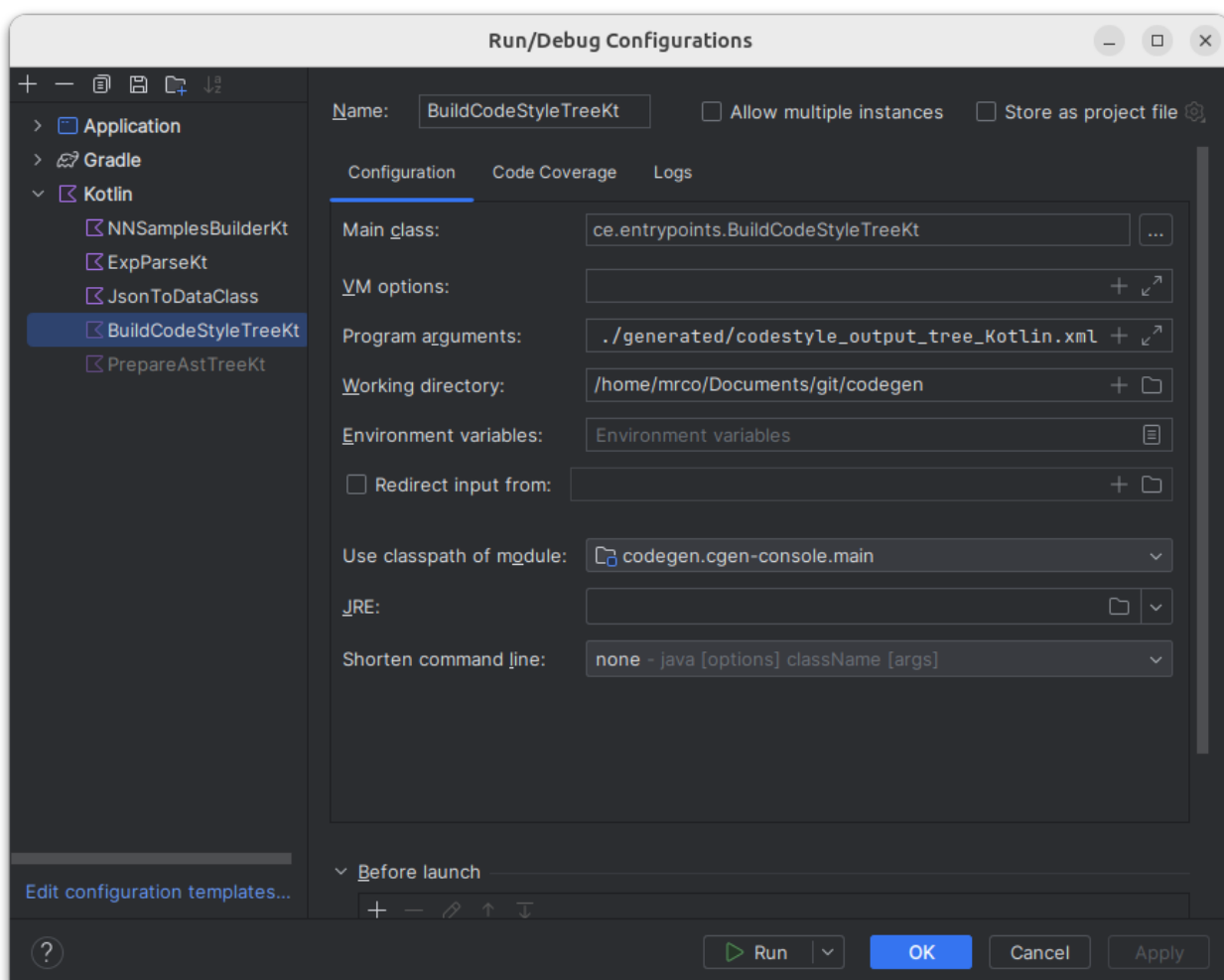


Рисунок 4.4 - Запуск трансляції дерева розбору в стилістичне дерево

На останньому кроку потрібно конфігурувати запуск етапу створення вихідних файлів на потрібних мовах програмування (рисунок 4.5).

Для цього треба в аргументах вказати:

- виконуючий клас, наразі це “ce.entrypoints.BuildOutFilesKt”;
- шлях до стилістичного дерева;
- шлях до файлу проєкту.

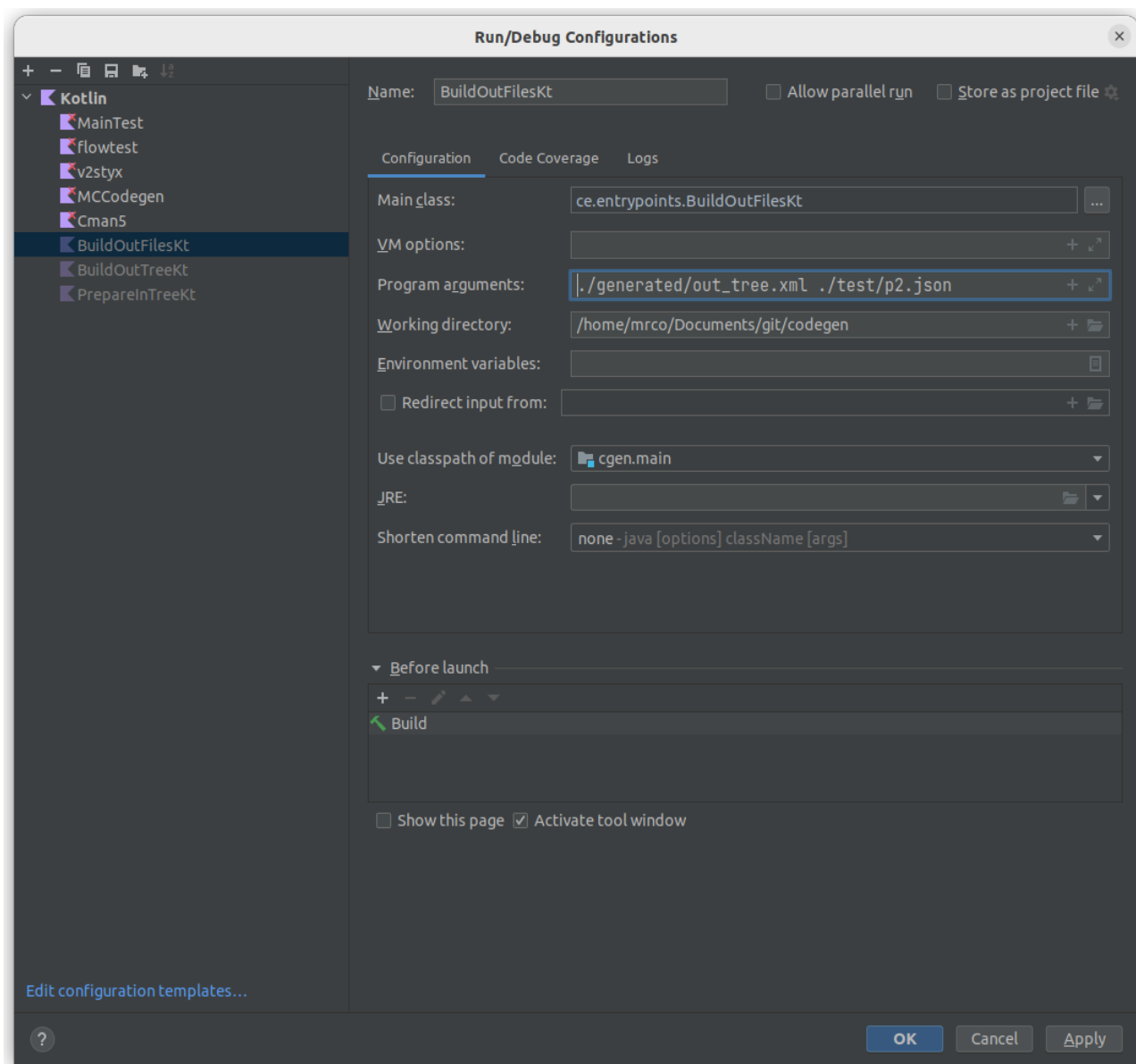


Рисунок 4.5 - Конфігурація запуску запису вихідних файлів

Після запуску запису в файли система видасть повідомлення про успішне завершення процесу та перелік файлів які були записані, лістинг 4.3.

Лістинг 4.3 - Приклад виводу системи на етапі запису вихідних файлів

```
1. Writing
   ./generated/com/knt622m/shkriabets/NewConstantsBlock.kt
2. Writing
   ./generated/com/knt622m/shkriabets/NewConstantsBlock2.kt
3. Writing
   ./generated/com/knt622m/shkriabets/NewConstantsBlock3.kt
4.
5. Process finished with exit code 0
```

Таким чином система генерує вихідні файли на заданих мовах високого рівня, які в подальшому можна використати при розробці мультиплатформених застосунків.

4.3 Інтеграція системи генерації в систему збірки Gradle

Генерацію коду можливо під'єднати до системи збірки Gradle [33].

Gradle [33] - це потужна система автоматизації збірки та керування залежностями, що використовується для розробки програмного забезпечення. Найбільшої популярності досягла в середі Java розробників. І в подальшому розповсюдилась на суміжні сфери - Android розробка, розробка на мовах Scala, Kotlin, Groovy тощо.

Головні функції Gradle:

- збірка проєкту: Gradle використовує DSL (Domain Specific Language) розроблений на основі Groovy або Kotlin для опису збірки проєкту. Завдяки цьому можна легко налаштовувати логіку збірки та задекларувати завдання та послідовність компіляції;
- керування залежностями: Gradle бере на себе логіку автоматичного

завантаження потрібних бібліотек, ресурсів та інших залежностей з репозитаріїв, це значно спрощує розгортання та підтримку робочого середовища розробників;

- розширюваність: Gradle дає можливість розширювати його функціональність за допомогою плагінів. Є багато вбудованих плагінів для різних мов програмування та інших завдань;

- підтримка багатомодульних проєктів: Gradle спроектовано з підтримкою багатомодульних проєктів, це дозволяє організовувати код у великих проєктах.

Gradle дуже популярний серед розробників завдяки своїй гнучкості, силі та здатності вирішувати різні завдання збірки та розгортання.

Для інтеграції системи генерації коду в систему збірки Gradle потрібно:

- під'єднати maven репозиторій jitpack.io, саме він використовується для розповсюдження скомпільованої версії кодогенератора;

- додати в список залежностей gradle-модуля, бібліотеку `com.github.vshcryabets:codegen`;

- створити gradle - завдання в якому потрібно налаштувати запуск класу `ce.entrypoints.BuildProjectKt` та вказати шлях до файлу з проєктом. Приклад такого завдання наведено в лістингу 4.4.

Лістинг 4.4 – Gradle-завдання для запуску генерації коду

```
1. task("cgeanTask", JavaExec::class) {
2.     workingDir(File("../"))
3.     args("./test/project.json")
4.     mainClass.set("ce.entrypoints.BuildProjectKt")
5.     classpath = sourceSets["test"].runtimeClasspath
6. }
```

В рядку 3 вказано шлях до файлу з проєктом. Рядок 2 відповідає за робочу директорію під час генерації коду.

4.4 Висновки до розділу 4

Таким чином, систему кодогенерації викладено у вільний доступ на платформі GitHub. Будь-хто може використовувати її у своїх проєктах по ліцензії GPL-3.0. Спільнота розробників може запропонувати свої правки та покращення системи за допомогою механізмів пул-реквестів або створення запитів про помилки на GitHub. Робота над системою продовжується, та планується додати підтримку інших мов високого рівня, зокрема Rust та Python. Ведеться робота з удосконалення трансляторів для генерації класів, методів та логіки. Планується розробка методики взаємодії та поширення коду між командою розробників за допомогою інструментарію git.

ВИСНОВКИ

Таким чином, у дипломному проєкті магістра було розроблено та запропоновано архітектуру системи кодогенерації. Для подальшого масштабування вона має розділення на різні етапи обробки вхідних описів. Проміжні результати між етапами зберігаються у вигляді дерев, таких як абстрактне синтаксичне дерево, дерево розбору, стилістичне дерево.

У роботі також було розроблено сценарії для побудови абстрактного синтаксичного дерева, трансляції його в дерево розбору, сценарії для додавання стилістичної інформації та запису в вихідні файли згенерованого коду. Для пояснення роботи системи наведені та розібрані приклади роботи трансляторів структур даних на мови C++ [4-6] та Kotlin [12-14].

Загалом у роботі було використано наступні матеріали, методи та технічні засоби:

- мова Kotlin використовується для опису та розробки системи автоматичної генерації коду;
- система автоматизації збірки Gradle;
- XML формат для передачі проміжних результатів між різними етапами виконання генерації;
- середа розробки IntelliJ Idea;
- система керування версіями файлів Git;
- GitHub - репозиторій для вихідного коду.

Таким чином, на даний час реалізовано кодогенератор, який з опису на метамові може генерувати структури та інтерфейси (примітиви) на мовах Kotlin, C/C++, Java, Swift з мінімальним втручанням людини.

Кодогенератор викладено у вільний доступ для ознайомлення на платформі GitHub [22]. Будь хто може використовувати розроблену систему у своїх проєктах по ліцензії GPL- 3.0. Спільнота розробників може запропонувати свої правки та

покращення системи за допомогою механізмів пул-реквестів або створення запитів про помилки на GitHub.

Робота над системою продовжується, репозиторій постійно оновлюється, виправляються помилки та система поліпшується. У систему планується додати підтримку інших мов високого рівня, зокрема Rust та Python. Ведеться робота з удосконалення трансляторів для генерації класів, методів та логіки. Також планується розробка методик взаємодії та поширення коду між командою розробників за допомогою інструментарію git.

На даний час кодогенератор використовується для генерації коду у виробничому процесі для суміщення старої та нової частини комерційного продукту, написаного на різних мовах програмування, що допомагає спростити розробку та зекономити час розробників. Акт про впровадження у виробничий процес ТОВ “ЛОГІТЕК УКРЕЙН” результатів науково-дослідної магістерської роботи представлено у додатку А.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1 Шкрябець В.І. Транскомпілятор як засіб мультиплатформленої розробки / В.І. Шкрябець, Т.С. Дьячук // Матеріали XI міжнародної науково-практичної конференції «Сучасні проблеми і досягнення в галузі радіотехніки, телекомунікацій та інформаційних технологій», 12-14 грудня 2022 року, Запоріжжя. – 2022. – С. 119-121

2 Шкрябець В.І. Система машинного перекладу з однієї мови високого рівня на іншу/ В.І. Шкрябець, Т.С. Дьячук // Тиждень науки-2023. Факультет комп'ютерних наук і технологій. Тези доповідей науково-технічної конференції, Запоріжжя, 24-28 квітня 2023 р. [Електронний ресурс] / Редкол. : Вадим ШАЛОМЄЄВ (відпов. ред.) Електрон. дані. – Запоріжжя : НУ «Запорізька політехніка», 2023. – С. 58-60

3 Source-to-source compiler. URL: https://en.wikipedia.org/wiki/Source-to-source_compiler/ (date of access: 21.11.2023).

4 Грицюк Ю.І., Рак Т.Є. Програмування мовою C++ : навчальний посібник. – Львів: Вид-во Львівського ДУ БЖД, 2011. – 292 с.

5 Bjarne Stroustrup. The C++ Programming Language (Third ed.): Addison-Wesley, 1997 - 912p.

6 Debbie Lafferty, Andrei Alexandrescu. Modern C++ Design: Generic Programming and Design Patterns Applied: Addison-Wesley, 2001 - 360p.

7 Lua Documentation. URL: <https://www.lua.org/docs.html> (date of access: 21.11.2023).

8 Flutter documentation. URL: <https://docs.flutter.dev/> (date of access: 21.11.2023).

9 Swav Kulinski. Flutter — The sky's the limit. URL: <https://medium.com/kinandcartacreated/flutter-the-skys-the-limit-84887c8f650d> (date of access: 05.12.2023).

10 Eric Windmill. Flutter in Action: Manning Publications, 2020 - 368p.

- 11 Kotlin Multiplatform. URL: <https://kotlinlang.org/docs/multiplatform.html> (date of access: 21.11.2023).
- 12 Iyanu Adelekan - Kotlin Programming By Example: Packt Publishing, 2018. - 500p.
- 13 Dmitry Jemerov, Svetlana Isakova - Kotlin in Action: Manning Publications, 2017 - 360p.
- 14 Pierre-Yves Saumont. The Joy of Kotlin: Manning Publications, 2019 - 480p.
- 15 2022 Developer Survey. URL: <https://survey.stackoverflow.co/2022/> (date of access: 21.11.2023).
- 16 React Native. URL: <https://reactnative.dev/> (date of access: 21.11.2023).
- 17 What is Xamarin? URL: <https://learn.microsoft.com/en-us/xamarin/get-started/what-is-xamarin> (date of access: 21.11.2023).
- 18 Mono Documentation. URL: <https://www.mono-project.com/docs/> (date of access: 21.11.2023).
- 19 5 AI Tools That Can Generate Code To Help Programmers. URL: <https://www.forbes.com/sites/janakirammsv/2022/03/14/5-ai-tools-that-can-generate-code-to-help-programmers/?sh=7f4865b35ee0> (date of access: 21.11.2023).
- 20 Comparison of code generation tools. URL: https://en.wikipedia.org/wiki/Comparison_of_code_generation_tools (date of access: 05.12.2023).
- 21 Project CodeNet. URL: <https://developer.ibm.com/exchanges/data/all/project-codenet/> (date of access: 22.11.2023).
- 22 GitHub Docs. URL: <https://docs.github.com/en> (date of access: 22.11.2023).
- 23 Git. URL: <https://git-scm.com/> (date of access: 22.11.2023).
- 24 Kotlin Script. URL: <https://www.jetbrains.com/help/teamcity/kotlin-script.html> (date of access: 22.11.2023).
- 25 Коротєєва Т.О. Алгоритми та структури даних: навч.посібник / Т.О.Коротєєва.- Львів: Видавництво Львівської політехніки, 2014. - 280 с.
- 26 Brian W. Kernighan; Dennis M. Ritchie. The C Programming Language. - Pearson; 2nd edition, 1988 - 272 p.

27 Вступ у JSON. URL: <https://www.json.org/json-uk.html> (дата звернення: 29.11.2023).

28 XML для новачків. URL: <https://support.microsoft.com/uk-ua/office/xml-для-новачків-a87d234d-4c2e-4409-9cbc-45e4eb857d44> (дата звернення: 29.11.2023).

29 IntelliJ IDEA. URL: <https://www.jetbrains.com/idea/> (дата звернення: 08.12.2023).

30 Groovy. URL: <https://groovy-lang.org/> (дата звернення: 30.11.2023).

31 Robert C. Martin. Clean Architecture: A Craftsman's Guide to Software Structure and Design. - Pearson; 1nd edition, 2017 - 352 p.

32 GNU GENERAL PUBLIC LICENSE. URL: <https://www.gnu.org/licenses/gpl-3.0.html> (дата звернення: 08.12.2023).

33 Gradle User Manual. URL: https://docs.gradle.org/current/userguide/userguide.html?_gl=1*11jmdvi*_ga*MTM3NDAxODAwMy4xNzAyMzYyODA5*_ga_7W7NC6YNPT*MTcwMjM2MjgwOS4xLjAuMTcwMjM2MjgwOS42MC4wLjA. (дата звернення: 12.12.2023).

ДОДАТОК А

ТОВАРИСТВО З ОБМЕЖЕНОЮ ВІДПОВІДАЛЬНІСТЮ «ЛОГІТЕК ІЮКРЕЙН»

Код за ЄДРПОУ 44196715

Адреса: Україна, 69005, Запорізька обл., місто Запоріжжя, пр.Соборний, будинок 160

Тел. +38(067)-617-01-26, e-mail: dvertynskiy@logitech.com, skirianova@logitech.com

Веб сторінка: www.logitech.com

Вих.№ 06/12-1 від 06 грудня 2023 року

м. Запоріжжя

АКТ

Про впровадження результатів науково-дослідної магістерської роботи

“Система автоматичної генерації коду на мові високого рівня”

Шкрябець Володимира Ігоровича

магістранта групи КНТ-622м НУ «Запорізька політехніка»

у виробничий процес

Комісія у складі:

Голови комісії

Директор ТОВ «ЛОГІТЕК ІЮКРЕЙН» Дмитро ВЕРТИНСЬКИЙ

Членів комісії

Старший інженер-програміст Олександр НОВАР

Головний бухгалтер Світлана КІР'ЯНОВА

склала дійсний акт про те, що результати науково-дослідної магістерської роботи “Система автоматичної генерації коду на мові високого рівня”, виконаної магістрантом ФКНТ НУ «Запорізька політехніка», гр. КНТ-622м Шкрябець В.І., науковий консультант – ст.викл. каф. КСМ Дьячук Т.С., впроваджені у робочий процес. Прототип системи “автоматичної генерації коду” викладено у вільний доступ на платформі GitHub та використовується для генерації коду у виробничому процесі ТОВ “ЛОГІТЕК ІЮКРЕЙН” для суміщення старої та нової частини комерційного продукту, написаного на різних мовах програмування, що дозволило спростити розробку та зекономити час розробників.

Голова комісії

Директор

ТОВ «ЛОГІТЕК ІЮКРЕЙН»

Підприємство працює без печатки

Члени комісії

Старший інженер-програміст

Головний бухгалтер

Дмитро

ВЕРТИНСЬКИЙ

Олександр

НОВАР

Світлана

КІР'ЯНОВА

Виконавець Світлана КІР'ЯНОВА +380 67 617 01 26

LOGITECH UKRAINE, LLC

160 Soborniy Avenue, Zaporizhzhia, 69005, Ukraine, phone +380612202840

ТОВ «ЛОГІТЕК ІЮКРЕЙН»

69005, Україна, м. Запоріжжя, проспект Соборний, 160, тел. +380612202840