**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ**

**«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ  
імені ІГОРЯ СІКОРСЬКОГО»**

**Факультет прикладної математики**

**Кафедра системного програмування і**

**спеціалізованих комп’ютерних систем**

|  |  |
| --- | --- |
| «На правах рукопису»  УДК 004.415 | До захисту допущено:  Завідувач кафедри  \_\_\_\_\_\_\_\_ Віталій РОМАНКЕВИЧ  «\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_2024р. |

**Магістерська дисертація**

**на здобуття ступеня магістра**

**за освітньо-професійною програмою «Системне програмування та**

**спеціалізовані комп’ютерні системи»**

**зі спеціальності 123 «Комп’ютерна інженерія»**

**на тему: «**Автоматизована система генерування UML діаграм**»**

Виконав (-ла):

студент (-ка) ІІ курсу, групи КВ-32мп

Цесля Антон Дмитрович \_\_\_\_\_\_\_\_\_\_

Науковий керівник:

Доцент кафедри СПіСКС, к. фіз..-мат. наук, доц.

Малежик Петро Михайлович \_\_\_\_\_\_\_\_\_\_

Рецензент:

Доцент кофедри ПМ, кандидат технічних наук

Маслянко Павло Павлович \_\_\_\_\_\_\_\_\_\_

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.

Студент (-ка) \_\_\_\_\_\_\_\_\_\_\_\_\_

Київ – 2024 року

**Національний технічний університет України**

**«Київський політехнічний інститут імені Ігоря Сікорського»**

**Факультет прикладної математики**

**Кафедра системного програмування і**

**спеціалізованих комп’ютерних сфистем**

Рівень вищої освіти – другий (магістерський)

Спеціальність – 123 «Комп’ютерна інженерія»

Освітньо-професійна програма **«Системне програмування та**

**спеціалізовані комп’ютерн системи»**

ЗАТВЕРДЖУЮ

Завідувач кафедри

\_\_\_\_\_\_\_ Віталій РОМАНКЕВИЧ

«\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_2023 р.

**ЗАВДАННЯ**

**на магістерську дисертацію студенту**

**Цеслі Антону Дмитровичу**

1. Тема дисертації « Автоматизована система генерування UML діаграм», науковий керівник дисертації Малежик Петро Михайлович, докт. пед.наук, проф., затверджені наказом по університету від «8 » листопада 2024 р. №5007-с

2. Термін подання студентом дисертації 9 грудня 2024

3. Об’єкт дослідження: алгоритм обходу абстрактного синтаксичного дерева згенерованого CLANG для автоматизованої генерації діаграм мовою PUML.

4. Вихідні дані: автоматично згенеровані діаграми в текстовому та графічному вигляді.

5. Перелік завдань, які потрібно розробити:

* Провести аналіз необхідності створення UML діаграм для проєктів.
* Порівняти ручний та автоматизований спосіб створнення діаграм.
* Оцінити наявні рішення створення UML діаграм
* Оцінити інструменти для автоматичного створення абстрактного синтаксичного дерева та його подальшого обходу та аналізу
* Розробити програмну систему для автоматизованої генерації UML діаграм з вихідного коду.
* Проаналізувати переваги системи і кращий спосіб використання

6. Орієнтовний перелік графічного (ілюстративного) матеріалу: презентація

7. Орієнтовний перелік публікацій:

* Тези доповіді «Автоматизована генерація UML діаграм за допомогою обходу абстрактного синтаксичного дерева» на науковій конференціїXVІІ науковій конференції магістрантів та аспірантів «Прикладна математика та комп’ютинг» ПМК-2024 (Київ, 20-22 листопада 2024 р.).
* Тези доповіді «Автоматизована генерація UML діаграм за допомогою обходу абстрактного синтаксичного дерева» на Х Міжнародній молодіжній науково-практичній інтернет-конференції «Наука і молодь в XXI сторіччі» (Полтава, 28 листопада 2024 р.)

8. Дата видачі завдання 25.10.2023

Календарний план

|  |  |  |  |
| --- | --- | --- | --- |
| № з/п | Назва етапів виконання  магістерської дисертації | Термін виконання етапів магістерської дисертації | Примітка |
| 1. | Узгодження керівника кваліфікаційної роботи (МД) та його затвердження | 15.09.2023 |  |
| 2. | Визначення тематики (напряму) дослідження магістерської дисертації | 15.10.2023 |  |
| 3. | Формулювання об`єкта і предмета дослідження | 01.12.2023 |  |
| 4. | Зміст та вступ до МД (робочі версія). Підготовка тез доповіді для виступу на конференції | 20.03.2023 |  |
| 5. | Перший розділ МД з висновками (робочі версія).  Підготовка тез доповіді для виступу на конференції | 15.04.2023 |  |
| 6. | Другий розділ МД з висновками (робочі версія) | 15.05.2023 |  |
| 7. | Тема магістерської дисертації, заява на ім’я завідувача кафедри з точною назвою МД | 10.09.2024 |  |
| 8. | Реферат до МД (українською мовою) та перший розділ з висновками | 08.10.2024 |  |
| 9. | Залік з науково-дослідної практики | 31.10.2024 |  |
| 10. | Попередній захист магістерської дисертації | 02.12.2024 |  |
| 11. | Надання керівнику МД остаточного варіанту дисертації для перевірки на запозичення та збіг/ідентичність/схожість | 05.12.2024 |  |
| 12. | Здача всіх документів магістерської дисертації секретарю комісії (Бояріновій Ю.Є.) | 9.12.2024 |  |
| 13. | Захисти МД | 16.12.2024 |  |

Студент Антон ЦЕСЛЯ

Науковий керівник Петро МАЛЕЖИК

**РЕФЕРАТ**

**Актуальність теми.** Документування програмного коду є критично важливою частиною процесу розробки та підтримки програмного продукту. В зв’язку зі стрімким збільшенням ролі інформаційних технологій в сучасному світі - складність та об’єми програмного коду в нових та існуючих проєктах постійно зростають. Людський ресурс та продуктивність праці є ключовим фактором розвитку продукту, це зумовлює те, що кожен учасник повинен мати можливість швидко дізнатись як працює той чи інший модуль проєкту для впровадження нового функціоналу чи відладки існуючого. Зважаючи на об’єми сучасних продуктів більшість розробників не знають досконально як працює вся інформаційна система, якою вони займаються. Для вирішення цього питання приходить різноманітна документація коду, яку, зазвичай, розробляють інженери, які займались ним. Зазвичай мова йде про різноманітні діаграми, які мовами розмітки на кшталт UML інженери створюють самі. Впровадження автоматичної системи генерації діаграм допоможе істотно прискорити процес розробки та підтримки продуктів та позитивно вплине на кінцевий результат.

**Об’єктом дослідження** є алгоритми обходу абстрактного синтаксичного дерева(AST) згенерованого з вихідного коду C++ для автоматизованої генерації різноманітних діаграм мовою UML.

**Предметом дослідження** є методи та технології обходу абстрактного синтаксичного дерева для автоматизованої генерації UML діаграм з вихідного коду C++.

**Метою роботи** є детальний аналіз та розробка методу обходу AST для генерації різноманітних UML діаграм

**Наукова новизна** полягає в автоматизованій генерації UML діаграм методом обходу абстрактного синтаксичного дерева з вихідного коду С++.

**Практична цінність** отриманих робіт полягає в тому, що при розробці нового продукту, чи підтримці існуючого команда розробки буде мати можливість швидко автоматизувати дороговартісний по ресурсам процес документації проєкту, а саме її частину з діаграмами класів по поведінки, що дозволить зосередити більше ресурсів на розробку та пришвидшить ознайомлення нових людей з кодовою базою.

**Апробація роботи.** Основні положення і результати роботи були представлені та обговорювались на XVІІ науковій конференції магістрантів та аспірантів «Прикладна математика та комп’ютинг» ПМК-2024 (Київ, 20-22 листопада 2024 р.), тези включені до дотатку А. Також вони були представлені на Х Міжнародній молодіжній науково-практичной інтернет-конференції «Наука і молодь в XXI сторіччі»(Полтава,28 листопада 2024р.), тези включені до додатку Б.

**Структура та обсяг роботи**. Магістерська дисертація складається з вступу, 4 розділів та висновків. В додатку В предсталено презентацію, а в додатку Г – лістинг розробленого програмного забезпечення.

*У вступі* подано обґрунтування актуальності теми, визначено об'єкт та предмет дослідження, сформульовано мету та завдання роботи, описано наукову новизну та практичну цінність отриманих результатів.

*У першому розділі* зроблено оцінку важливості документування програмного коду, та проаналізовані існуючі способи для цього.

*У другому розділі* проведено дослідження загальних молживостей UML діаграм та оглянуто інструменти для створення системи автоматичної генерації діаграм.

*У третьому розділі* оглянуто структуру програмної системи, її модулі та алгоритми.

*У четвертому розділі* розглянутий приклад генерації діаграм класів та поведінки

*У висновках* представлені результати проведеної роботи.

Работа представлена на 81 аркуші, містить 4 додатки та посилання на список використаних літературних джерел з 16 найменувань. У роботі наверено 8 рисунків та 6 таблиць

Ключові слова: UML, абстрактне синтаксичне дерево, AST, C++, автоматична генерація діаграм, обход AST, документація програмного коду.

**ABSTRACT**

**Topic relevance.** Documenting software code is a critically important part of the process of developing and supporting a software product. Due to the rapid increase in the role of information technology in the modern world, the complexity and volume of software code in new and existing projects are constantly growing. Human resources and labor productivity are key factors in product development, which means that each participant must be able to quickly learn how a particular project module works to implement new functionality or debug an existing one. Given the volume of modern products, most developers do not know in detail how the entire information system they are working on works. To solve this issue, various code documentation comes into play, which is usually developed by engineers who worked on it. Usually, we are talking about various diagrams that engineers create themselves in markup languages ​​​​such as UML. This takes up quite a lot of time for highly paid specialists, which they could spend on more priority tasks. The implementation of an automatic diagram generation system will help to significantly accelerate the process of product development and support and will positively affect the final result.

**The object of research** is the algorithms for traversing the abstract syntax tree (AST) generated from the C++ source code for the automated generation of various diagrams in the UML language.

**The subject of the research** is the methods and technologies for traversing the abstract syntax tree for the automated generation of UML diagrams from the C++ source code.

**The purpose of the work** is a detailed analysis and development of the AST traversal method for the generation of various UML diagrams.

**The scientific novelty** lies in the automated generation of UML diagrams by the method of traversing the abstract syntax tree from the C++ source code.

The practical value of the work obtained is that when developing a new product or supporting an existing one, the development team will be able to quickly automate the resource-intensive process of project documentation, namely it's part with behavior class diagrams, which will allow more resources to be focused on the development and will speed up the familiarization of new people with the code base.

**Approbation of the work**. The main provisions and results of the work were presented and discussed at the XVII Scientific Conference of Master's and PhD Students "Applied Mathematics and Computing" PMK-2024 (Kyiv, November 20-22, 2024), thesis was added to addition A. Also they were presented at the 10th International Youth Scientific and Practical Internet Conference "Science and Youth in the 21st Century" (Poltava, November 28, 2024), thesis was added to addition B.

**Structure and scope of the work.** The master's thesis consists of an introduction, 4 sections, and conclusions. Addition C contains presentation and addition D contains source code.

*The introduction* provides a justification for the relevance of the topic, defines the object and subject of the study, formulates the goal and objectives of the work, and describes the scientific novelty and practical value of the results obtained.

*The first section* assesses the importance of documenting program code and analyzes existing methods for this.

*The second section* studies the general capabilities of UML diagrams and reviews tools for creating a system for automatic diagram generation.

*The third section* reviews the structure of the software system, its modules, and algorithms.

*The fourth section* examines an example of generating class and behavior diagrams.

*The results* of the work are analyzed in the conclusions

The work is completed on 81 sheets, contains 4 appendices and links to the list of used literary sources from 16 titles. The work contains 8 figures and 6 tables.

**Keywords**: UML, abstract syntax tree, AST, C++, automatic diagram generation, AST traversal, program code documentation.

Зміст

[ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ОДИНИЦЬ 3](#_Toc184381149)

[ВСТУП 4](#_Toc184381150)

[1. Аналіз існуючих методів документування програмного коду 6](#_Toc184381151)

[1.1. Важливість документування програмного коду 6](#_Toc184381152)

[1.2. UML діаграми 8](#_Toc184381153)

[1.3. Створення UML діаграм без засобів автоматизації 10](#_Toc184381154)

[1.4. Автоматизована генерація UML діаграм 12](#_Toc184381155)

[1.5. Недоліки та невирішені задачі існуючих методів 14](#_Toc184381156)

[1.6 Переваги запропонованого підходу 14](#_Toc184381157)

[Висновки до розділу 17](#_Toc184381158)

[2. ДОСЛІДЖЕННЯ, ВИБІР ТА ОГЛЯД ОБРАНИХ МЕТОДІВ ДЛЯ АВТОМАТИЗАЦІЇ ГЕНЕРАЦІЇ UML ДІАГРАМ 18](#_Toc184381159)

[2.1. Можливості UML діаграм 18](#_Toc184381160)

[2.2. Використання Python 22](#_Toc184381161)

[2.3. Аналіз та огляд можливостей мови PlantUML 26](#_Toc184381162)

[2.4. Використання Cmake для отримання списку файлів 35](#_Toc184381163)

[2.5. Clang як інструмент аналізу коду C++ 36](#_Toc184381164)

[Висновки до розділу 40](#_Toc184381165)

[3. ОГЛЯД СТРУКТУРИ СИСТЕМИ ТА ВИКОРИСТАНИХ АЛГОРИТМІВ 42](#_Toc184381166)

[3.1. Загальна структура проекту 43](#_Toc184381167)

[3.2. Розгляд алгоритму генерації діаграм класів 49](#_Toc184381168)

[3.3 Огляд аналіз алгоритму генерації діаграм поведінки 56](#_Toc184381169)

[Висновки до розділу 64](#_Toc184381170)

[4. ТЕСТУВАННЯ РОЗРОБЛЕНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ТА АНАЛІЗ ОТРИМАНИХ РЕЗУЛЬТАТІВ 66](#_Toc184381171)

[4.1. Виконання програми на тестовому проєкті C++ 66](#_Toc184381172)

[4.2. Огляд згенерованої діаграми класів 67](#_Toc184381173)

[4.3. Огляд згенерованої діаграми поведінки 72](#_Toc184381174)

[Висновки до розділу 76](#_Toc184381175)

[Висновки 78](#_Toc184381176)

[Список використаних джерел 80](#_Toc184381177)

[Додаток А 82](#_Toc184381178)

[Додаток Б 87](#_Toc184381179)

[Додаток В 90](#_Toc184381180)

[Додаток Г 98](#_Toc184381181)

# ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ОДИНИЦЬ

UML — уніфікована мова моделювання

AST — абстрактне синтаксичне дерево

C++ — мова програмування C++

API — інтерфейс прикладного програмування

CI**/**CD — безперервна інтеграція та безперервна доставка

JSON — JavaScript Object Notation (текстовий формат обміну даними)

YAML — YAML Ain't Markup Language (формат серіалізації даних)

Clang — компілятор та інструментарій для мов сімейства C

LLVM — інфраструктура для розробки компіляторів та пов'язаних з ними технологій

Python — мова програмування високого рівня загального призначення

PlantUML — інструмент для створення UML-діаграм за допомогою текстових описів

CMake — кросплатформена система автоматизації збірки програмного забезпечення

Git — розподілена система керування версіями

SDK — набір засобів розробника програмного забезпечення

IDE — інтегроване середовище розробки

OOP — об'єктно-орієнтоване програмування

Doxygen — інструмент для генерації документації з вихідного коду

Graphviz — програмне забезпечення для візуалізації графів

HTTP — протокол передачі гіпертексту

URL — уніфікований локатор ресурсів

SDK — Software Development Kit, набір засобів розробника

CLI — командний інтерфейс

CSV — Comma-Separated Values, формат табличних даних

# ВСТУП

У сучасному світі інформаційних технологій програмне забезпечення відіграє ключову роль у різних сферах людської діяльності — від медицини та освіти до промисловості та фінансів. З розвитком технологій та збільшенням обсягів даних програмні системи стають усе більш складними, інтегрованими та багатофункціональними. Це ставить перед розробниками нові виклики щодо ефективного управління проєктами, забезпечення якості коду та підтримки високого рівня продуктивності команд.

Одним із найважливіших аспектів успішної розробки та підтримки складних програмних систем є якісна документація. Вона слугує не лише засобом передачі знань між членами команди, але й інструментом для аналізу, планування та прийняття рішень. Документація допомагає розробникам швидко орієнтуватися в кодовій базі, розуміти взаємозв'язки між компонентами та забезпечувати узгодженість у процесі розробки.

Особливу роль у документуванні програмного забезпечення відіграють UML (Unified Modeling Language) діаграми. Вони надають стандартизовані засоби для моделювання різних аспектів системи, включаючи структуру, поведінку, взаємодію компонентів та інші важливі характеристики. UML діаграми використовуються на всіх етапах життєвого циклу розробки — від аналізу вимог до впровадження та підтримки, сприяючи кращому розумінню системи як технічними спеціалістами, так і зацікавленими сторонами.

Проте на практиці часто виникають проблеми з підтримкою документації в актуальному стані. Ручне створення та оновлення UML діаграм є трудомістким процесом, який потребує значних ресурсів та часу. У великих проєктах з активною фазою розвитку документація може швидко застарівати, що знижує її корисність та може призвести до помилок і непорозумінь у команді. Це особливо актуально в умовах Agile-методологій, де швидкість реагування на зміни та гнучкість є ключовими факторами успіху.

Автоматизація процесу генерації UML діаграм стає все більш актуальною задачею. Вона дозволяє не лише знизити трудовитрати на документування, але й забезпечити актуальність та точність діаграм, що в свою чергу сприяє підвищенню якості програмного забезпечення та ефективності роботи команд розробників.

Наукова новизна роботи полягає в розробці методу автоматизованої генерації UML діаграм з вихідного коду C++ шляхом аналізу AST, що дозволяє обробляти складні конструкції мови та отримувати точну та повну інформацію про структуру та поведінку програмного забезпечення. Практична значущість полягає в створенні інструменту, який може бути використаний у реальних проєктах для підвищення ефективності документування, зниження витрат часу та ресурсів на підтримку документації, покращення комунікації в команді та якості програмного забезпечення.

# 1. Аналіз існуючих методів документування програмного коду

## 1.1. Важливість документування програмного коду

У сучасному світі програмне забезпечення стає все більш складним і багатофункціональним. Розробка та підтримка таких систем вимагає не лише високої кваліфікації розробників, але й ефективних методів управління знаннями про систему. Одним із ключових аспектів цього процесу є документування програмного коду, яке забезпечує візуалізацію архітектури та взаємодії компонентів системи.

Якісна документація є критичним компонентом успішних програмних проектів. Згідно з дослідженням, проведеним компанією Atlassian, розробники витрачають до 20% свого робочого часу на пошук інформації про код, який вони не писали [17]. Це свідчить про те, що наявність якісної документації може значно підвищити продуктивність команди. Також, за даними IEEE Software, в середньому 50% витрат на розробку програмного забезпечення припадає на підтримку та модифікацію існуючого коду [18]. Наявність актуальної документації може скоротити ці витрати до 30%.

**Роль документації в різних методологіях розробки**

У різних методологіях розробки програмного забезпечення документація відіграє важливу роль:

**Класичні методології (Waterfall)**

Ввелика увага приділяється детальній документації на кожному етапі, що забезпечує структурований підхід до розробки.

**Agile**

Хоча Agile акцентує увагу на робочому програмному забезпеченні більше, ніж на документації, проте вона все ще є важливою для передачі знань між членами команди.

**DevOps**

Документація сприяє швидкій передачі інформації між розробниками та операційними командами, що полегшує процеси розгортання та підтримки.

**Приклади з реального проекту**

Відомий випадок з компанією Knight Capital Group, коли відсутність актуальної документації призвела до втрати понад 440 мільйонів доларів за 45 хвилин. Помилка в коді, яку не виявили через відсутність належної документації та тестування, призвела до катастрофічних наслідків.

На даний момент вже практично та теоретично доведена цінність створення та підтримки якісної документації коду.

Розглянемо позитивний вплив якісної документації на базі декількох досліджень:

* de Souza, S. C. B., Anquetil, N., & de Oliveira, K. M. (2005). A study of the documentation essential to software maintenance. Proceedings of the 23rd Annual International Conference on Design of Communication: Documenting & Designing for Pervasive Information.[1]
* Treude, C., & Storey, M.-A. (2010). Work item tagging: Communicating concerns in collaborative software development. IEEE Transactions on Software Engineering.[2]
* Fagerholm, F., & Münch, J. (2012). Developer experience: Concept and definition. International Conference on Software and System Process, 73–77.[3]
* Tan, L., Yuan, D., Krishna, G., & Zhou, Y. (2007). / iComment: Bugs or bad comments? /. Proceedings of the 21st ACM Symposium on Operating Systems Principles.[4]

Якщо підсумувати результат всіх досліджень, стає зрозуміло, що якісна та актуальна документація є критичним компонентом успішних програмних проектів. Вона значно сприяє зменшенню технічного боргу та полегшує процеси підтримки і модернізації програмного забезпечення.[5] Коли документація добре структурована і детальна, розробники витрачають менше часу на розуміння існуючого коду, що підвищує загальну ефективність роботи.

Крім того, документація виступає ключовим засобом комунікації всередині команди. Вона допомагає уникнути непорозумінь, забезпечуючи узгоджене розуміння вимог і рішень між всіма учасниками проекту. Це особливо важливо для великих команд або проектів з географічно розподіленими учасниками.

Наявність детальної документації також значно скорочує час адаптації нових членів команди. Вона служить навчальним матеріалом і довідником, допомагаючи новачкам швидше інтегруватися в робочий процес і зрозуміти складні аспекти системи. Це особливо актуально в умовах високої плинності кадрів або швидкого розширення команди.

Документація позитивно впливає на якість програмного забезпечення. Вона допомагає виявляти та виправляти потенційні проблеми на ранніх етапах розробки, що підвищує надійність і стабільність системи. Зрештою, добре задокументований код полегшує процеси тестування та валідації, сприяючи випуску більш якісних продуктів.

Отже, інвестування в створення і підтримку якісної документації є стратегічно важливим для довгострокового успіху програмних проектів. Вона не тільки спрощує технічні аспекти розробки, але й підвищує ефективність командної роботи, покращує комунікацію та сприяє швидшій адаптації нових спеціалістів. У комплексі, це веде до більш ефективного управління проектом і підвищення задоволеності всіх зацікавлених сторін.

## 1.2. UML діаграми

UML (Unified Modeling Language) діаграми - це стандартизований набір графічних нотацій, що використовується в сфері розробки програмного забезпечення для моделювання, візуалізації, документування та спілкування про структуру та поведінку програмних систем[6].

UML був розроблений в 1990-х роках як засіб об'єднання різних методів моделювання. З моменту свого створення UML пройшов кілька версій, остання з яких — UML 2.5.1, випущена в 2017 році. Ця версія включає уточнення та покращення існуючих елементів, додає нові можливості для моделювання складних систем.

UML діаграми допомагають розробникам, архітекторам, аналітикам та іншим учасникам проекту зрозуміти та спроектувати різні аспекти системи, включаючи:

* Структуру системи: відображення класів, об'єктів, інтерфейсів та їх взаємозв'язків.
* Поведінку системи: моделювання процесів, робочих потоків, послідовностей дій та взаємодій між компонентами.
* Взаємодію між компонентами: показує, як різні частини системи співпрацюють між собою.

### Основні типи UML діаграм

* **Діаграми класів**: відображають класи системи, їх атрибути, методи та відносини між ними.
* **Діаграми послідовності**: показують порядок взаємодій між об'єктами для виконання певної функції.
* **Діаграми станів**: моделюють різні стани об'єкта та переходи між ними.
* **Діаграми активності**: відображають робочі процеси та потоки управління в системі.
* **Діаграми компонентів**: представляють фізичну архітектуру програмного забезпечення, включаючи модулі та їх взаємозв'язки.
* **Діаграми розгортання**: показують розподіл програмного забезпечення на фізичному обладнанні.

Найбільш інформативними та детальними є діаграми класів та послідовності.[7] В свою чергу, через те, що вони містять дуже багато інформації їх дуже важко підтримувати в актуальному стані. Коли швидкість розробки стає пріоритетом, підтримкою діаграм часто нехтують, відкладаючи на майбутнє. Часто дуже великий невиконаний об’єм роботи з документацією, тестами і т.д. може призвести до того, що потрібно буде переробляти їх з повного нуля, адже структура ПЗ та логіка вже сильно відрізняється від того, що було до цього[8].

## 1.3. Створення UML діаграм без засобів автоматизації

Існує безліч варіантів створення UML діаграм. Загалом вони діляться на:[9]

* Графічні
* Текстові

Обидва підходи мають свої переваги та недоліки, які варто враховувати при виборі інструменту або методу для моделювання системи.

Для прикладів графічних інструментів для створення UML діаграм можна виділити: Microsoft Visio, Lucidchart, StarUML, Enterprise Architect, Visual Paradigm.

Розглянемо переваги та недоліки даного підходу.

Переваги:

* Інтуїтивний інтерфейс: Більшість графічних інструментів надають зручний користувацький інтерфейс з перетягуванням та налаштуванням елементів.
* Візуальний контроль: Миттєво видно зміни та взаємодії між елементами діаграми, що полегшує розуміння структури.
* Широкі можливості налаштування: Можливість змінювати стиль, кольори, шрифти та інші графічні атрибути для покращення читабельності.
* Підтримка командної роботи: Деякі інструменти дозволяють одночасну роботу декількох користувачів над однією діаграмою.

Недоліки:

* Великий об’єм роботи для великих діаграм: Ручне додавання та налаштування великої кількості елементів може займати багато часу.
* Ризик помилок: Ручне створення діаграм може призвести до пропусків або неточностей, особливо в складних системах.

Текстові інструменти для створення UML діаграм використовують певний текстовий опис діаграми для подальшого перетворення у графічне зображення. Для прикладу текстових інструментів візьмемо: PlantUML, Mermaid, UMLet, TextUML, Yuml.

Переваги:

* Швидкість створення та оновлення: Можливість швидко змінювати діаграми шляхом редагування тексту.
* Контроль версій: Текстові файли легко інтегруються з системами контролю версій (Git, SVN).
* Легка інтеграція: Можливість вбудовувати діаграми в документацію, вихідний код або системи Wiki.
* Безкоштовність та відкритий код: Багато текстових інструментів є безкоштовними та мають відкритий вихідний код.

Недоліки:

* Вища крива навчання: Потрібно вивчити специфічний синтаксис для опису діаграм.
* Менший візуальний контроль: Попередній перегляд змін може вимагати додаткового кроку генерації зображення.
* Обмежені можливості налаштування графіки: Менша гнучкість у налаштуванні стилів та оформлення порівняно з графічними інструментами.
* Відсутність графічного інтерфейсу: Не можна безпосередньо маніпулювати графічними елементами мишею.

Таблиця 1.1. - Приклад використання PlantUML

|  |  |
| --- | --- |
| Код PlantUML: | Згенерована діаграма |
| @startuml  class User {      - String name      - String email      + login()      + logout()  }  class Admin {      + manageUsers()  }  User <|-- Admin  @enduml |  |

Підсумовуючи можна сказати, що графічні інструменти підходять для невеликих, або середніх проектів, де важлива візуальна привабливість та інтуїтивність інтерфейсу. Вони корисні, коли потрібно швидко створити діаграму без глибокого занурення в синтаксис або коли команда складається з людей, які не знайомі з текстовими мовами опису діаграм.

Текстові інструменти є кращими для великих проектів, де необхідна  інтеграція з процесами розробки. Вони дозволяють легше оновлювати діаграми при зміні коду, підтримують контроль версій та спрощують командну роботу. Незважаючи на вищий поріг входу, вони забезпечують більшу ефективність у довгостроковій перспективі.

## 1.4. Автоматизована генерація UML діаграм

Автоматизована генерація UML діаграм з вихідного коду є ефективним рішенням для подолання проблем, пов'язаних з ручним створенням та підтримкою документації.[10] Це особливо важливо для мов програмування, таких як C++, які характеризуються складним синтаксисом та потужними можливостями мета-програмування.

Існуючі методи автоматизованої генерації UML діаграм для C++

### Doxygen з Graphviz

Doxygen є популярним інструментом для генерації документації з вихідного коду, який може створювати діаграми залежностей за допомогою Graphviz.

Переваги:

* Підтримка багатьох мов програмування, включаючи C++.
* Автоматична генерація діаграм класів та їх взаємозв'язків.

Недоліки:

* Обмежена деталізація діаграм.
* Не підтримує діаграми поведінки (послідовності, станів).
* Складність обробки специфічних конструкцій C++ (шаблони, простори імен).

### Використання динамічного аналізу коду

Існують інструменти які безпосередньо використовують дані виконання програми для подальшої генерації діаграм поведінки. З переваг можна відмітити максимальну точність, так як використовується інформація про реальну поведінку програми, але це тягне за собою необхідність постійного запуску при перегенерації діаграм, що може потягнути за собою використання великої кількості ресурсів.[11] Також треба відмітити дуже високу складність налаштування та підтримки даних інструментів, через їх специфічний підхід.

### Використання парсерів та власних розробок для конкретного проєкту

Для великих і довготривалих проєктів можна розглядати розробку спеціалізованих програм, які аналізують вихідний код та генерують UML діаграми враховуючи специфічні необхідності конкретного проєкту.[12] Звісно це тягне за собою необхідність виділяти людські ресурси на підтримку даних інструментів, що не має сенсу для основної маси програмних систем.

## 1.5. Недоліки та невирішені задачі існуючих методів

### Неповна підтримка специфічних особливостей C++

Багато інструментів не обробляють шаблони, простори імен, перевантаження операторів та інші складні конструкції, що тягне за собою втрату важливої інформації, та неточність.

### Обмежена генерація діаграм поведінки

Наявні інструменти не мають повноцінного функціоналу автоматизованої генерації діаграм послідовності та станів. Часто такі діаграми потребують найбільшої уваги та можливості їх налаштування під конкретні вимоги, як глибина рекурсії викликів, вилучення з діаграм логіки сторонніх бібліотек, тощо.

### Складність інтеграції з сучасними процесами розробки

Багато інструментів не підтримують інтеграцію з системами контролю версій, CI/CD, а це тягне за собою ускладнення автоматизації процесів розробки та підтримки актуальності документації.[13]

### Висока вартість та складність використання

Комерційні інструменти можуть бути дорогими у впровадженні та складними для освоєння, а це тягне за собою обмежену доступність для невеликих команд або проектів з обмеженим бюджетом.

## 1.6 Переваги запропонованого підходу

### Використання абстрактного синтаксичного дерева Clang

Clang забезпечує доступ до повної інформації про вихідний код, включаючи новітні стандарти C++[14]. Інтерфейс Clang надає структуроване представлення коду, що дозволяє отримувати детальну інформацію про структуру та взаємозв'язки конструкцій. Основна перевага - це коректна обробка складних конструкцій (шаблони, простори імен) та відносно нескладний інтерфейс, який підтримує всі новітні стандарти С++ та постійно оновлюється.

### Інтеграція з Python

Бібліотека Clang доступна і для С++ і для Python. Для даної задачі більше підходить гнучкість та швидкість розробки, мови програмування Python, адже єдиний мінус цієї мови, швидкість, для нас тут не має значення, так-як це код, який використовується не користувачем продукту, а розробником. Також варто зазначити що у порівнянні з використанням С++ великої різниці по швидкості не буде, адже сама побудова синтаксичного дерева, як найскладніший та найбільш часозатратний написаний мовою С++, а через інтерфейс використовуються вже готові конструкції. Основною перевагою буде легкість написання та підтримки коду, велика кількість бібліотек. Також через те, що Python інтерпретована мова програмування, можлива більш легка інтеграція з іншими інструментами та процесами розробки.

### Генерація та зберігання діаграм мовою PlantUML

З точки зору автоматичної генерації текстовий формат є єдиним варіантом який слід розглядати. Діаграми в такому форматі займають невелику кількість пам’яті, а також цей формат ідеально підходить для зберігання та інтеграції в усі наявні системи контролю версій. Навіть через текстовий інтерфейс систем контролю версій розробник може чітко зрозуміти, як змінилась логіка виконання, чи зв’язків у програмі, використовуючи тільки інформацію про зміни в діаграмі, навіть не переглядаючи код. Також варто відмітити універсальність, цей формат підтримує різні типи діаграм, як структурні, так і поведінкові діаграми (класи, послідовності, стани).[15]

### Загальні переваги запропонованого підходу

Запропонований підхід має декілька суттєвих переваг. По-перше, він забезпечує актуальність діаграм завдяки автоматичному оновленню. Автоматична генерація UML діаграм з вихідного коду гарантує, що документація завжди відображає поточний стан системи. Це зменшує ризик людських помилок, пов'язаних з ручним оновленням, і запобігає накопиченню застарілої інформації, що спрощує підтримку та розвиток проєкту.

По-друге, підхід характеризується масштабованістю та продуктивністю. Він підходить для проектів будь-якого розміру, включаючи великі кодові бази з тисячами файлів. Використання Clang AST та оптимізованих алгоритмів дозволяє швидко аналізувати код та генерувати діаграми. Можливість розподілення аналізу коду на декілька потоків або вузлів також сприяє прискоренню процесу.

Третьою перевагою є відкритість та безкоштовність рішення. Використання інструментів з відкритим вихідним кодом, таких як Clang, Python та PlantUML, робить систему доступною без додаткових фінансових витрат. Широка спільнота розробників, що використовують ці інструменти, забезпечує наявність ресурсів, документації та прикладів. Крім того, відкритий код дозволяє модифікувати та розширювати функціонал системи під конкретні потреби проекту.

Четвертою перевагою є інтеграція в сучасні процеси розробки. Система може бути інтегрована в процеси безперервної інтеграції та доставки (CI/CD), автоматично генеруючи діаграми при кожному коміті або збірці. Текстовий формат діаграм легко відслідковується в системах контролю версій, що дозволяє бачити історію змін. Генеровані діаграми можуть бути автоматично включені в документацію проекту, зменшуючи витрати часу на ручне оновлення.

Нарешті, підхід сприяє покращенню якості програмного забезпечення. Використання AST дозволяє виявляти складні залежності та взаємозв'язки, що може допомогти в оптимізації та рефакторингу коду. Автоматично згенеровані діаграми відповідають стандартам UML, що сприяє уніфікації документації. Наявність актуальних діаграм також спрощує процес навчання нових членів команди та розуміння складних систем.

## Висновки до розділу

У цьому розділі було проведено детальний аналіз існуючих методів документування програмного коду та створення UML діаграм, зокрема для мови C++. Розглянуто переваги та недоліки ручних підходів до створення діаграм, включаючи графічні та текстові інструменти. Виявлено, що ручне створення та підтримка UML діаграм є трудомістким процесом, схильним до помилок та застарівання, що може негативно вплинути на ефективність розробки та якість програмного забезпечення.

Аналіз існуючих автоматизованих рішень показав, що вони мають ряд обмежень, зокрема неповну підтримку специфічних особливостей C++, обмежену можливість генерації діаграм поведінки, складність інтеграції з сучасними процесами розробки, а також високу вартість та складність освоєння деяких інструментів.

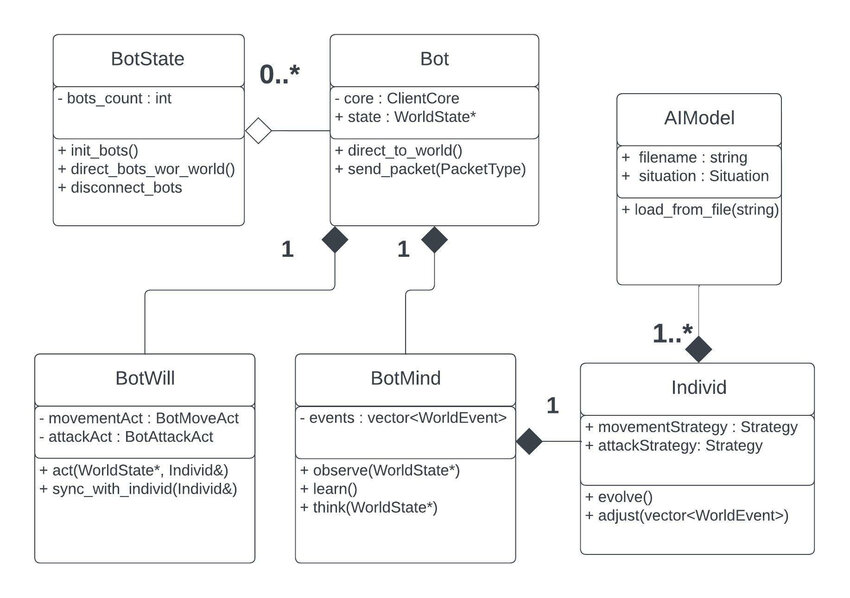
Запропонований підхід, що базується на використанні AST Clang, мови Python та PlantUML, надає можливість подолати ці недоліки. Він забезпечує повну підтримку сучасних стандартів C++ та коректну обробку складних конструкцій, гнучкість та швидкість розробки завдяки використанню Python, автоматичну генерацію та оновлення UML діаграм у текстовому форматі, що спрощує інтеграцію з системами контролю версій та процесами CI/CD, а також відкритість та безкоштовність завдяки використанню інструментів з відкритим вихідним кодом.

Таким чином, запропонований підхід має значний потенціал для покращення процесів документування та розробки програмного забезпечення на C++. Він сприяє підвищенню якості коду, ефективності командної роботи та зниженню технічного боргу. Подальший розвиток системи може включати розширення підтримки інших мов програмування, додавання нових типів діаграм, оптимізацію генерації та інтеграцію з іншими інструментами, що зробить її ще більш корисною та універсальною для розробників.

# 2. ДОСЛІДЖЕННЯ, ВИБІР ТА ОГЛЯД ОБРАНИХ МЕТОДІВ ДЛЯ АВТОМАТИЗАЦІЇ ГЕНЕРАЦІЇ UML ДІАГРАМ

## 2.1. Можливості UML діаграм

Unified Modeling Language (UML) є стандартизованою мовою моделювання, яка широко використовується в сфері розробки програмного забезпечення для візуалізації, специфікації, конструювання та документування компонентів програмних систем. UML надає набір графічних нотацій для представлення різних аспектів системи, що сприяє покращенню розуміння, комунікації та співпраці між членами команди розробників, аналітиками, архітекторами та зацікавленими сторонами.



**Рисунок 2.1.** - Приклад UML діаграми класів.

### Основні типи UML діаграм

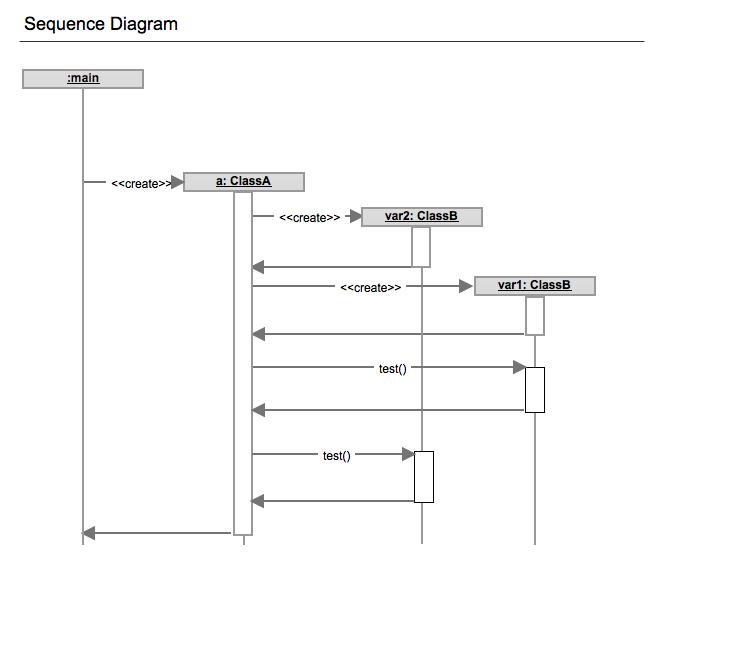
UML включає в себе кілька типів діаграм, які можна розділити на дві основні категорії: діаграми структури та діаграми поведінки

Діаграми структури відображають статичні аспекти системи.

* **Діаграми класів**: Показують класи, інтерфейси, їх атрибути, методи та відносини між ними, такі як наслідування, асоціації та агрегації.
* **Діаграми об'єктів**: Відображають конкретні екземпляри класів та їх стан на певний момент часу.
* **Діаграми компонентів**: Моделюють фізичні компоненти системи, такі як модулі, бібліотеки та їх взаємозв'язки.
* **Діаграми розгортання**: Показують фізичну архітектуру системи, включаючи вузли та з'єднання між ними.

Діаграми поведінки відображають динамічні аспекти системи.

* **Діаграми послідовності**: Відображають взаємодію між об'єктами у вигляді послідовності повідомлень.
* **Діаграми активності**: Моделюють робочі процеси та потоки управління в системі.
* **Діаграми станів**: Показують можливі стани об'єкта та переходи між ними залежно від подій.
* **Діаграми використання (use case diagrams)**: Відображають функціональні можливості системи з точки зору користувача та їх взаємодію з системою.



**Рисунок 2.2*. -***Діаграма послідовності.

### Роль UML діаграм у розробці програмного забезпечення

UML діаграми відіграють ключову роль на різних етапах життєвого циклу розробки програмного забезпечення:

* **Збір та аналіз вимог**: На цьому етапі діаграми випадків використання (Use Case Diagrams) допомагають виявити та зафіксувати функціональні вимоги до системи з точки зору користувача. Вони відображають взаємодію акторів з системою та визначають основні сценарії використання.
* **Проектування системи**: Діаграми класів (Class Diagrams) та діаграми компонентів (Component Diagrams) використовуються для моделювання статичної структури системи. Вони допомагають визначити класи, інтерфейси, атрибути, методи та взаємозв'язки між ними, що є основою для подальшої реалізації.
* **Реалізація та кодування**: Діаграми послідовності (Sequence Diagrams) та діаграми активності (Activity Diagrams) допомагають моделювати динамічну поведінку системи, відображаючи потоки управління та взаємодію між об'єктами під час виконання певних функцій. Це спрощує процес кодування та забезпечує узгодженість між дизайном та реалізацією.
* **Тестування та верифікація**: Діаграми станів (State Diagrams) можуть бути використані для моделювання можливих станів об'єктів та переходів між ними, що допомагає виявляти помилки та перевіряти коректність роботи системи.
* **Впровадження та підтримка**: Діаграми розгортання (Deployment Diagrams) допомагають планувати фізичну архітектуру системи, визначати розміщення компонентів на обладнанні, що є важливим для DevOps та інженерів з експлуатації.

### Комунікація між стейкхолдерами

UML діаграми є універсальним засобом комунікації між різними зацікавленими сторонами проекту:

* **Розробники** використовують діаграми для розуміння архітектури та взаємозв'язків у системі, що спрощує процес кодування та підтримки.
* **Тестувальники** можуть використовувати діаграми для розробки тестових сценаріїв та забезпечення покриття всіх функціональних вимог.
* **Менеджери проектів** отримують загальне уявлення про систему, що допомагає в управлінні ресурсами та плануванні.
* **Клієнти та користувачі** можуть краще зрозуміти, як система буде працювати, що сприяє узгодженню вимог та очікувань.

### Переваги використання UML діаграм

* **Візуалізація складних систем**: Діаграми дозволяють візуально представити складні структури та процеси, що сприяє кращому розумінню та аналізу.
* **Полегшення процесу розробки**: Чітке моделювання системи перед початком кодування допомагає виявити потенційні проблеми та оптимізувати архітектуру.
* **Підтримка документування**: UML діаграми слугують офіційною документацією, яка може бути використана протягом усього життєвого циклу системи.

### Проблеми при використанні UML діаграм

* **Підтримка актуальності**: У великих та швидко змінюваних проєктах діаграми можуть швидко застарівати, якщо не підтримуються в актуальному стані.
* **Часові витрати**: Створення та оновлення діаграм може бути трудомістким процесом, особливо при ручному підході.
* **Складність масштабування**: У великих системах діаграми можуть ставати надто складними для сприйняття, що вимагає їх розбиття на менші частини або абстрагування деталей.

### ****Приклади успішного використання UML діаграм****

У великих програмних проектах, таких як розробка систем керування банками, авіаційними системами або телекомунікаційними мережами, UML діаграми є незамінними інструментами. Наприклад, компанія IBM активно використовує UML у своїх проектах для моделювання складних систем та покращення комунікації між розподіленими командами.

## **2.2. Використання Python**

У цьому підрозділі буде детально розглянуто, чому для реалізації інструменту було обрано мову програмування Python, а також описано основні конструкції та теоретичні аспекти Python, які будуть використовуватися в розробці. Це включає огляд особливостей мови, стандартних бібліотек, а також сторонніх модулів, що є критичними для успішної реалізації поставлених завдань.

Python є однією з найбільш популярних мов програмування, відомою своєю простотою, зручним синтаксисом та широким спектром застосувань. Вибір Python для реалізації нашого інструменту був зумовлений наступними факторами:

* **Швидкість розробки**: Завдяки високому рівню абстракції та лаконичному синтаксису, Python дозволяє швидко розробляти програмне забезпечення, що є важливим для прототипування та реалізації складних алгоритмів.
* **Багата екосистема бібліотек**: Python має величезну кількість вбудованих та сторонніх бібліотек, що спрощують роботу з різними типами даних, мережевими протоколами, базами даних, а також надають можливості для інтеграції з іншими мовами та платформами.
* **Інтеграція з Clang**: Існує бібліотека libclang для Python, яка дозволяє взаємодіяти з Clang API, що є критичним для аналізу вихідного коду C++ та побудови AST.
* **Зручність роботи з текстом та файлами**: Python надає потужні інструменти для обробки текстових даних, регулярних виразів, роботи з файловою системою, що необхідно для генерації текстових описів діаграм та управління конфігураційними файлами.

### ****Основні конструкції Python, що будуть використані****

* У процесі розробки інструменту будуть активно використовуватися наступні конструкції та особливості мови Python:
* **Об'єктно-орієнтоване програмування (ООП)**: Python підтримує ООП, що дозволяє створювати класи, об'єкти, використовувати наслідування, поліморфізм та інші парадигми ООП. Це полегшує структурування коду та його підтримку.
* **Модулі та пакети**: Для організації коду на логічні частини будуть використовуватися модулі та пакети. Це спрощує управління залежностями та покращує масштабованість проєкту.
* **Декоратори**: Декоратори дозволяють модифікувати поведінку функцій або методів без зміни їхнього коду. Вони можуть бути використані для логування, кешування або інших цілей.
* **Генератори та вирази-генератори**: Генератори дозволяють створювати ітератори для послідовного доступу до елементів великих наборів даних без зайвого використання пам'яті.
* **Регулярні вирази**: Модуль re надає можливості для пошуку та обробки текстових шаблонів, що корисно при аналізі та трансформації текстових даних.
* **Робота з файлами та директоріями**: Використання модулів os, os.path, glob для управління файлами, отримання списків файлів, перевірки існування шляхів тощо.
* **Стандартні структури даних**: Списки, словники, множини та кортежі будуть використовуватися для зберігання та обробки зібраної інформації.

### ****Використання сторонніх бібліотек та модулів****

Окрім стандартної бібліотеки Python, у проєкті будуть використовуватися наступні сторонні бібліотеки:

* **clang.cindex**: Це модуль, що надає Python-інтерфейс до libclang. Він дозволяє взаємодіяти з Clang API, отримувати AST, обходити його та вилучати необхідну інформацію.
* **PyYAML**: Бібліотека для роботи з YAML-файлами. Використовується для завантаження та збереження конфігураційних параметрів з файлу diagrams.yaml.
* **PlantUML**: Хоча PlantUML не є бібліотекою Python, для взаємодії з ним можуть використовуватися бібліотеки, такі як plantuml, які дозволяють програмно викликати PlantUML для генерації діаграм.

### ****Використання словників для зберігання інформації про класи****

Словники дозволяють зручно зберігати та отримувати доступ до інформації про класи, їх атрибути та методи.

classes\_info = {}

classes\_info[class\_name] = {

'attributes': [],

'methods': [],

'base\_classes': []

}

**Завантаження конфігурації з YAML-файлу**

import yaml

with open('diagrams.yaml', 'r') as file:

config = yaml.safe\_load(file)

### ****Генерація текстових описів діаграм****

Використовуючи форматування рядків, можна легко створювати текстові файли з описами діаграм у синтаксисі PlantUML.

uml\_lines = []

uml\_lines.append('@startuml')

for class\_name, class\_info in classes\_info.items():

uml\_lines.append(f'class {class\_name} {{')

for attr in class\_info['attributes']:

uml\_lines.append(f' {attr}')

uml\_lines.append('}')

uml\_lines.append('@enduml')

with open('class\_diagram.puml', 'w') as file:

file.write('\n'.join(uml\_lines))

**Використання декораторів для логування**

def log\_execution(func):

def wrapper(\*args, \*\*kwargs):

print(f'Executing {func.\_\_name\_\_}')

return func(\*args, \*\*kwargs)

return wrapper

@log\_execution

def generate\_class\_diagram():

# Код генерації діаграми

pass

### ****Обробка помилок та виключень****

Python надає механізми для обробки помилок та виключень, що важливо для забезпечення надійності програми.

**Використання конструкцій** try-except

try:

# Код, що може викликати виключення

index = clang.cindex.Index.create()

except Exception as e:

print(f'An error occurred: {e}')

**Створення власних виключень**

class ParsingError(Exception):

pass

# Використання

if not source\_files:

raise ParsingError('No source files found.')

### ****Переваги та виклики використання Python****

**Переваги**

* **Швидкість розробки**: Python дозволяє швидко реалізовувати складні функціональності з меншим обсягом коду.
* **Читабельність коду**: Завдяки зрозумілому синтаксису, код на Python легко читати та підтримувати.
* **Велика спільнота та ресурси**: Наявність великої кількості ресурсів, документації та активної спільноти розробників.

**Виклики**

* **Швидкодія**: Python є інтерпретованою мовою, що може впливати на продуктивність при обробці великих обсягів даних.
* **Залежності від версій**: Необхідність контролювати сумісність версій Python та бібліотек.

## 2.3. Аналіз та огляд можливостей мови PlantUML

**PlantUML** — це відкритий інструмент для створення UML діаграм, який використовує простий текстовий синтаксис. Він дозволяє розробникам описувати діаграми у вигляді текстових скриптів, які автоматично перетворюються на графічні зображення.[16] Це спрощує процес моделювання, робить його більш гнучким та інтегрованим з процесами розробки програмного забезпечення.

### ****Переваги PlantUML****

* **Спрощення створення UML діаграм**: Замість використання графічних редакторів, розробники можуть швидко створювати діаграми, пишучи текст.
* **Інтеграція з контролем версій**: Текстові файли легко відстежуються в системах контролю версій, таких як Git.
* **Автоматизація**: Можливість автоматичного генерування діаграм в процесах збірки або документації.

### PlantUML підтримує широкий спектр UML діаграм:

* **Діаграми класів**: Моделюють структуру класів та їх взаємозв'язки.
* **Діаграми послідовності**: Відображають взаємодію між об'єктами у часі.
* **Діаграми станів**: Показують стани об'єктів та переходи між ними.
* **Діаграми активності**: Моделюють потоки управління та робочі процеси.
* **Діаграми компонентів**: Відображають фізичні компоненти системи та їх взаємодію.
* **Діаграми розгортання**: Показують розподіл системи на фізичному обладнанні.
* **Діаграми випадків використання**: Відображають взаємодію користувачів із системою.

### Переваги текстового підходу

* **Контроль версій**: Текстові файли легко відстежувати та зберігати історію змін.
* **Автоматизація**: Можливість генерувати діаграми автоматично під час збірки проєкту.
* **Простота обміну**: Текстові діаграми можна легко передавати та обговорювати в командах.

### Синтаксис та використання PlantUML

Кожен скрипт PlantUML починається з @startuml і завершується @enduml. Між цими директивами знаходиться опис діаграми.

**Приклад:**

@startuml

...опис діаграми...

@enduml

**Створення класу:**

class ClassName {

+ publicAttribute : Type

- privateAttribute : Type

# protectedMethod() : ReturnType

}

**Таблиця 2.1. - Модифікатори доступу**

|  |  |
| --- | --- |
| Модифікатор доступу в мовах програмування | Модифікатор доступу в мовах PlantUML |
| public | + |
| private | - |
| protected | # |
| package/private | ~ |

### ****Відносини між класами****

Використовуючи PlantUML можна дуже легко декларувати відносини між класами. Для цього в окремому рядку треба написати назву цих класів, та між ними вставити набір символів, що відобразять необхідні відносини, нариклад:

ClassA --> ClassB

Таблиця 2.2. - Основні відносини в UML.

|  |  |
| --- | --- |
| Тип відносин | Набір символів для позначення відносин мовою PlantUML |
| **Наслідування** | <|-- |
| **Реалізація інтерфейсу** | ..|> |
| **Асоціація** | --> |
| **Агрегація** | o-- |
| **Композиція** | \*-- |
| **Залежність** | ..> |

**Таблиця 2.3. - Приклад діаграми класів**

|  |  |
| --- | --- |
| Код діаграми | Згенерована діаграма |
| @startuml  class Vehicle {  + speed : int  + move() : void  }  class Car {  + numberOfDoors : int  + openDoor() : void  }  Vehicle <|-- Car  @enduml |  |

Таблиця 2.4. - **Основні елементи діаграм послідовності та приклади їх створення**

|  |  |  |
| --- | --- | --- |
| **Актори та учасники** | **Повідомлення** | **Активність учасника** |
| actor User  participant "System" as Sys | User -> Sys : request()  Sys --> User : response() | activate Sys  deactivate Sys |

**Таблиця 2.5. - Приклад діаграми послідовності**

|  |  |
| --- | --- |
| Код діаграми | Згенерована діаграма |
| @startuml  actor Customer  participant "Online Shop" as Shop  Customer -> Shop : selectProduct()  Shop -> Customer : displayDetails()  Customer -> Shop : placeOrder()  Shop -> Customer : confirmOrder()  @enduml |  |

### Розширені можливості PlantUML для використання в діаграмах послідовності

PlantUML надає потужний набір інструментів для моделювання складних сценаріїв у діаграмах послідовності. Особливо це стосується моделювання логіки програм, де важливо відобразити умовні переходи, цикли, паралельні процеси та інші конструкції. Розглянемо розширені можливості PlantUML, зосередившись на таких елементах:

* alt / else
* opt
* loop
* par
* break
* critical
* group

Елементи **alt** та **else** використовуються для моделювання альтернативних шляхів виконання, подібно до умовних операторів if-else у мовах програмування.

**Синтаксис:**

**alt [Condition]**

**Actions if the condition is true**

**else [Another condition]**

**Actions if the other condition is true**

**else**

**Actions if none of the conditions are true**

**end**

Елемент **opt** використовується для позначення опціональних дій, які можуть виконуватися або ні залежно від умови. Це аналог оператора if без else.

**Синтаксис:**

opt [Condition]

Actions if the condition is true

end

Елемент **loop** дозволяє моделювати повторювані дії або цикли.

**Синтаксис:**

loop [Condition or iterations]

Repeated actions

end

Елемент **par** використовується для моделювання паралельних процесів або потоків.

**Синтаксис:**

par

Branch 1

and

Branch 2

...

end

Елемент **break** використовується для позначення виходу з потоку або припинення поточної взаємодії при певній умові.

**Синтаксис:**

break [Condition]

Actions before breaking

end

Елемент **critical** використовується для позначення критичних секцій, де важливо забезпечити синхронізацію доступу.

**Синтаксис:**

critical [Description]

Actions in the critical section

end

Елемент **group** дозволяє об'єднувати декілька взаємодій під спільним описом або назвою, полегшуючи читання діаграми.

**Синтаксис:**

group [Group Name]

Actions within the group

end

### Комплексний приклад

Об'єднавши кілька розглянутих елементів у одному прикладі для моделювання складного сценарію отримаєму наступну, інформативну та комплексну діаграму, графічний варіант якої показаний на рисунку 2.3:

Код діаграми:

@startuml

participant "User" as User

participant "System" as System

participant "Payment Service" as PaymentService

participant "Delivery Service" as DeliveryService

User -> System : Place order

group "Order Validation"

opt [Check item availability]

System -> System : Check inventory

end

alt [Item in stock]

System -> User : Order confirmed

else

System -> User : Item out of stock

break

end

end

par

group "Payment"

User -> PaymentService : Make payment

PaymentService --> User : Payment confirmation

end

group "Preparation for Shipping"

Згенерована діаграма:

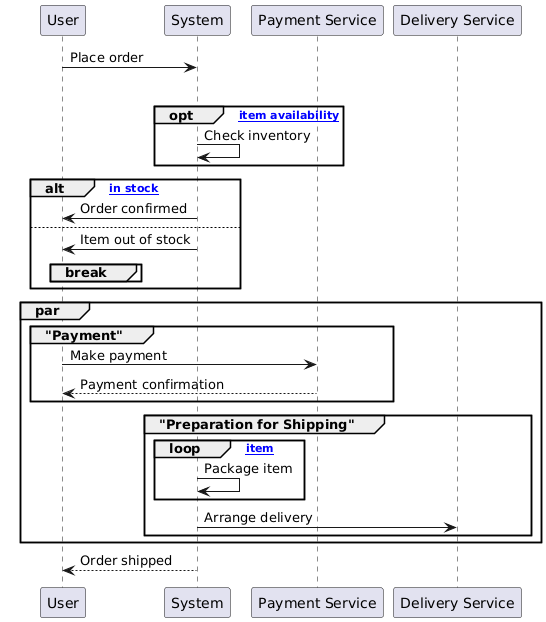


Рисунок 2.3. – Комплексний приклад

### Переваги використання PlantUML

* **Простота використання**: Легкий та зрозумілий синтаксис.
* **Текстовий формат**: Полегшує інтеграцію з системами контролю версій та автоматизацію.
* **Гнучкість**: Підтримка широкого спектру діаграм та можливість налаштування.
* **Безкоштовність**: Відкритий вихідний код та відсутність ліцензійних платежів.

## 2.4. Використання Cmake для отримання списку файлів

CMake - це кросплатформений інструмент для автоматизації процесу збірки програмного забезпечення. Він широко використовується в проектах на C та C++, надаючи гнучкий та потужний спосіб визначення процесу компіляції та зв'язування програм. В Процесі роботи Cmake генерує compile\_commands.json ,який містить інформацію про команди компіляції для кожного файлу в проекті. Він використовується інструментами для статичного аналізу коду, автодоповнення, лінтерів та інших цілей. Саме цей файл ми використаємо для збору інформації про файли, для яких нам необхідно генерувати діаграми.

### Генерація compile\_commands.json з CMake

За замовчуванням цього файлу в файловій системі немає, для його експорту з внутрішньої пам’яті CMake необхідно додати наступну опцію при конфігурації проекту. Це створить файл compile\_commands.json у каталозі build.

cmake -S . -B build -DCMAKE\_EXPORT\_COMPILE\_COMMANDS=ON

### Структура файлу compile\_commands.json

compile\_commands.json - це JSON-масив, де кожен елемент містить інформацію про компіляцію окремого файлу.

[

{

"directory": "/path/to/project/build",

"command": "/usr/bin/clang++ -I../include -std=c++17 -o CMakeFiles/myapp.dir/main.cpp.o -c /path/to/project/main.cpp",

"file": "/path/to/project/main.cpp"

},

{

"directory": "/path/to/project/build",

"command": "/usr/bin/clang++ -I../include -std=c++17 -o CMakeFiles/myapp.dir/utils.cpp.o -c /path/to/project/utils.cpp",

"file": "/path/to/project/utils.cpp"

}

]

### Значення полів

* **directory**: Каталог, у якому виконується команда компіляції.
* **command**: Повна команда компіляції файлу, включаючи всі параметри та опції.
* **file**: Шлях до вихідного файлу, який компілюється.

## 2.5. Clang як інструмент аналізу коду C++

### Огляд Clang та його можливостей

Clang є фронтендом компілятора для мов програмування C, C++ та Objective-C, який входить до складу проєкту LLVM. Він надає інструменти для парсингу, аналізу та трансформації вихідного коду. Однією з ключових особливостей Clang є наявність API для роботи з абстрактним синтаксичним деревом (AST), що дозволяє отримувати детальну інформацію про структуру програми.Переваги використання Clang AST

### Можливості Clang

Clang забезпечує:

* **Парсинг вихідного коду**: Перетворює вихідний код у внутрішнє представлення (AST), яке точно відображає структуру програми.
* **Підтримка сучасних стандартів C++**: Clang підтримує всі актуальні стандарти C++, включаючи C++11, C++14, C++17 та C++20.
* **API для роботи з AST**: Надає програмний інтерфейс для обходу та аналізу AST, що дозволяє розробляти інструменти для статичного аналізу коду, рефакторингу, генерації документації тощо.
* **Можливості розширення**: Clang дозволяє створювати плагіни та інструменти, які можуть інтегруватися з існуючими процесами розробки.

### Переваги використання Clang

* **Точність та деталізація**: AST Clang містить повну інформацію про кожен елемент коду, включаючи типи, шаблони, простори імен, макроси та інші конструкції.
* **Швидкість та ефективність**: Завдяки оптимізаціям у Clang, парсинг коду та побудова AST виконуються швидко, навіть для великих проєктів.
* **Відкрите програмне забезпечення**: Clang є проєктом з відкритим вихідним кодом, що дозволяє використовувати його безкоштовно та модифікувати під свої потреби.

### Мета використання Clang

Для імплементації запропонованого підходу Clang використовується для автоматизованого отримання інформації про структуру вихідного коду C++ з метою подальшої генерації UML діаграм. Використання Clang дозволяє коректно обробляти всі особливості мови C++, включаючи складні конструкції та останні стандарти.

### Використання libclang

Для інтеграції використовується бібліотеку **libclang**, яка надає C API для роботи з Clang. Крім того, існують прив'язки до інших мов програмування, зокрема Python, через модуль clang.cindex.

### Використання Python для обробки AST

Мова програмування Python була обрана для обробки AST через її простоту та швидкість розробки. Використовуючи модуль clang.cindex можливо викликати функції libclang безпосередньо з Python, що спрощує процес обходу AST та збору необхідної інформації.

### Побудова AST

Першим кроком є побудова AST для заданого файлу або проєкту. Це здійснюється шляхом виклику функції парсингу з необхідними параметрами.

### ****Приклад коду мовою Python****

###### import clang.cindex

###### # Ініціалізація індексу

###### index = clang.cindex.Index.create()

###### # Парсинг файлу

###### translation\_unit = index.parse('main.cpp', args=['-std=c++17'])

### Обхід AST та вилучення інформації

Після отримання AST необхідно обійти його вузли, збираючи інформацію про класи, методи, атрибути та їх взаємозв'язки.

**Основні типи вузлів, які будуть використані:**

* **Класи та структури** (CursorKind.CLASS\_DECL, CursorKind.STRUCT\_DECL): містять інформацію про визначення класів та структур.
* **Методи та функції-члени** (CursorKind.CXX\_METHOD, CursorKind.FUNCTION\_DECL): описують методи класів.
* **Атрибути та поля** (CursorKind.FIELD\_DECL): містять інформацію про змінні-члени класів.
* **Наслідування** (CursorKind.CXX\_BASE\_SPECIFIER): вказує на базові класи при наслідуванні.
* **Простори імен** (CursorKind.NAMESPACE): для коректної обробки просторів імен.
* **Шаблони** (CursorKind.CLASS\_TEMPLATE, CursorKind.FUNCTION\_TEMPLATE): для роботи з шаблонними класами та методами.

### Обробка специфічних конструкцій C++

**Шаблони:** для обробки шаблонних класів та методів використовуються вузли типу CursorKind.CLASS\_TEMPLATE та CursorKind.FUNCTION\_TEMPLATE. Це дозволяє отримати інформацію про параметри шаблонів та їх використання.

**Простори імен:** використовуючи вузли CursorKind.NAMESPACE, стає можливим зберігати ієрархію просторів імен, щоб забезпечити унікальність імен класів та їх коректне відображення в діаграмах.

**Наслідування:** вузли CursorKind.CXX\_BASE\_SPECIFIER містять інформацію про базові класи при наслідуванні. Це дозволяє будувати ієрархію класів та відображати відносини наслідування на діаграмах.

### Реалізація обходу AST ****мовою Python****

**Отримання кореневого вузла:**

root\_cursor = translation\_unit.cursor

**Рекурсивний обхід вузлів**

def visit\_node(node):

# Обробка вузла

for child in node.get\_children():

visit\_node(child)

**Перевірка типу вузла**

###### if node.kind == clang.cindex.CursorKind.CLASS\_DECL:

**Отримання інформації про вузол:**

name = node.spelling # Назва елемента

location = node.location # Місце в коді

### Робота з атрибутами та методами

**Отримання атрибутів класу:**

if node.kind == clang.cindex.CursorKind.FIELD\_DECL:

field\_name = node.spelling

field\_type = node.type.spelling

**Отримання методів класу:**

if node.kind == clang.cindex.CursorKind.CXX\_METHOD:

method\_name = node.spelling

return\_type = node.result\_type.spelling

# Отримання параметрів методу

parameters = [param.type.spelling for param in node.get\_arguments()]

### Повний приклад обходу AST

Припустимо, що маємо такий код, написаний мовою програмування C++:

namespace MyNamespace {

template <typename T>

class MyClass : public BaseClass {

T value;

void doSomething(T param);

};

}

Під час обходу AST отримаємо наступні вузли:

**NAMESPACE\_DECL**: MyNamespace

**CLASS\_TEMPLATE**: MyClass

**CXX\_BASE\_SPECIFIER**: BaseClass

**FIELD\_DECL**: value

**CXX\_METHOD**: doSomething

## Висновки до розділу

У цьому розділі було проведено детальний аналіз методів та інструментів, спрямованих на автоматизацію генерації UML діаграм з вихідного коду C++.

По-перше, було розглянуто можливості UML діаграм як інструменту для візуалізації, моделювання та документування програмних систем. Визначено, що UML діаграми відіграють ключову роль на всіх етапах розробки, від аналізу вимог до підтримки та розвитку програмного забезпечення. Однак традиційний підхід до їх створення може бути трудомістким і не завжди забезпечує актуальність діаграм у динамічних проєктах.

По-друге, проаналізовано мову PlantUML, яка пропонує текстовий синтаксис для опису UML діаграм. Використання PlantUML спрощує процес створення та оновлення діаграм, забезпечуючи інтеграцію з системами контролю версій і підтримку автоматизації. Наведено приклади синтаксису та можливостей PlantUML, що підтверджує її ефективність для швидкого моделювання складних систем.

Третім аспектом було використання CMake для отримання списку файлів проєкту через генерацію файлу compile\_commands.json. Цей файл містить детальну інформацію про команди компіляції кожного файлу, що є корисним для подальшого аналізу коду та автоматизованої генерації діаграм.

Нарешті, досліджено можливості Clang як інструменту для аналізу коду C++. Використання Clang та його бібліотеки libclang дозволяє отримувати абстрактне синтаксичне дерево (AST) програми, що містить повну інформацію про структуру коду. Це відкриває можливості для автоматизованого вилучення інформації про класи, методи, атрибути та їх взаємозв'язки.

Поєднання цих інструментів - Clang для аналізу коду, CMake для управління збіркою та PlantUML для генерації діаграм - створює ефективний підхід до автоматизації процесу створення UML діаграм. Це не лише зменшує часові витрати на документування, але й забезпечує актуальність та точність діаграм, що є критично важливим у сучасних динамічних проєктах. Такий підхід сприяє покращенню якості розробки програмного забезпечення та підвищенню продуктивності команд розробників.

# 3. ОГЛЯД СТРУКТУРИ СИСТЕМИ ТА ВИКОРИСТАНИХ АЛГОРИТМІВ

У цьому розділі ми детально розглянемо програмне забезпечення, розроблене для автоматичної генерації діаграм класів та послідовностей на основі вихідного коду C++. Ми проаналізуємо структуру коду, опишемо створені класи та їх функціональність, розглянемо використані бібліотеки та API.

Ключова задача – розробити додаток, легкий в користуванні, проте гарно налаштовуємий для максимально детального та зрозумілого генерування документації коду С++. Цей додаток повинен бути з відкритим вихідним кодом, для можливості допрацювання, при необхідності, користувачами. Для такого типу програмних систем дуже добре підходить мова програмування Python, через її простий синтаксис та швидкість розробки. Єдиним мінусом цієї мови програмування є швидкість та відсутність багатопоточності, але для такої задачі нам це не потрібно, бо це тільки ускладнить подалюшу розробку, а швидкість для утиліт розробки не має суттєвого значення. Система використовує наступні технології:

* **Python:** обрана як основна мова програмування завдяки її гнучкості та широкому набору бібліотек.
* **Clang (libclang):** використовується для парсингу вихідного коду C++ та побудови AST.
* **PlantUML:** використовується для генерації UML діаграм на основі текстового опису.
* **CMake:** використовується для отримання списку файлів проєкту та їх параметрів компіляції.

Також система поєдную в собі такі архітектурні рішення:

* **Модульність:** система побудована за принципами модульності, що забезпечує легкість розширення та підтримки.
* **Конфігурованість:** використання зовнішніх файлів конфігурації дозволяє налаштовувати поведінку програми без зміни коду.
* **Інтеграція з процесом розробки:** система може бути інтегрована в процеси CI/CD, що забезпечує автоматичне оновлення документації при кожній зміні коду.

## 3.1. Загальна структура проекту

Показане рішення гарно структуроване та складається з 3 модулів, які виконують певний перелік задач, для створення фінального результату:

* class\_diagram
* sequence\_diagram
* utils

### ****Модуль**** class\_diagram

**Цей модуль необхідний для побудови діаграми класів. На вході він отримує список файлів для обробки, використовуючи бібліотеку Clang створює абстрактне синтаксичне дерево для кожного з файлів. Після цього відбувається парсинг дерев, під час якого аналізуються вузл, та генерується код діаграми. Модуль має декілька класів, а саме:**

* **ClassCodeParser: виконує функцію генерації AST та його обходу. Він генерує проміжний результат в вигляді двух масивів типу Relationship та ClassInfo**
* **ClassUMLGenerator: виконує перетворення проміжного результату ClassCodeParser в код puml, перетворюючи відношення та інформацію про класи в структури puml.**
* **Relationshi: це структура, яка зберігає віндошення між двома класами.**
* **ClassInfo: це структура, яка зберігає необхідну інформацію про клас, наприклад їм’я**

### ****Модуль**** sequence\_diagram

**Цей модуль необхідний для побудови діаграм поведінки. На вході він також отримує список файлів для обробки, та точку входу для побудови конкретної діаграми. Також використовує бібліотеку Clang для створення AST для кожного з файлів. Після цього відбувається парсинг дерева, під час якого, рекурсивно, аналізуються саме виклики функцій та методів. Модуль має декілька класів, а саме:**

* **SequenceCodeParser: виконує функцію генерації AST та його обходу. В процесі обходу виконується збір всіх викликів функцій, починаючи з певної точки входу. Збираються параметри, значення які повертає функція, який клас її викликає і якому класу вона належить.**
* **SequenceDiagramGenerator: виконує перетворення проміжного результату SequenceCodeParser в код puml. Генерує послідовність викликів та перетворює інформацію з масиву FunctionCall в структури puml.**
* **FunctionCall: це структура, яка зберігаю повну та необхідну інформацію про виклик функції.**

### ****Модуль**** utils

**Цей модуль виконує багато задач, не зв’язаних з генерацією діаграм, але необхідними для коректної роботи алгоритму. Він містить наступні класи:**

* **ConfigLoader: Виконує парсинг налаштувань діаграм, отриманих з файлів diagrams.yaml. Він використовує модуль PyYaml для обходу конфігурації та збереженню необхідної інформації про майбутні діаграми.**
* **UMLRenderer: Використовуючи відкриту та офіційну бібліотеку PlantUML перетворює текстові репрезентації діаграм в графічні, якщо є така необхідність**

**Також цей модуль має декілька невеликих функцій для отримання списку файлів та коректної роботи clang, а саме:**

* **find\_libclang: використовується для пошуку бібліотеки clang в системі**
* **run\_cmake: використовується для запуску системі збірки CMake, а саме для генерації файлу compile\_commands.json, який відображає файли проєкту.**
* **load\_compile\_commands: використовується для парсингу файлу compile\_commands.json та створенню списку файлів для парсингу.**

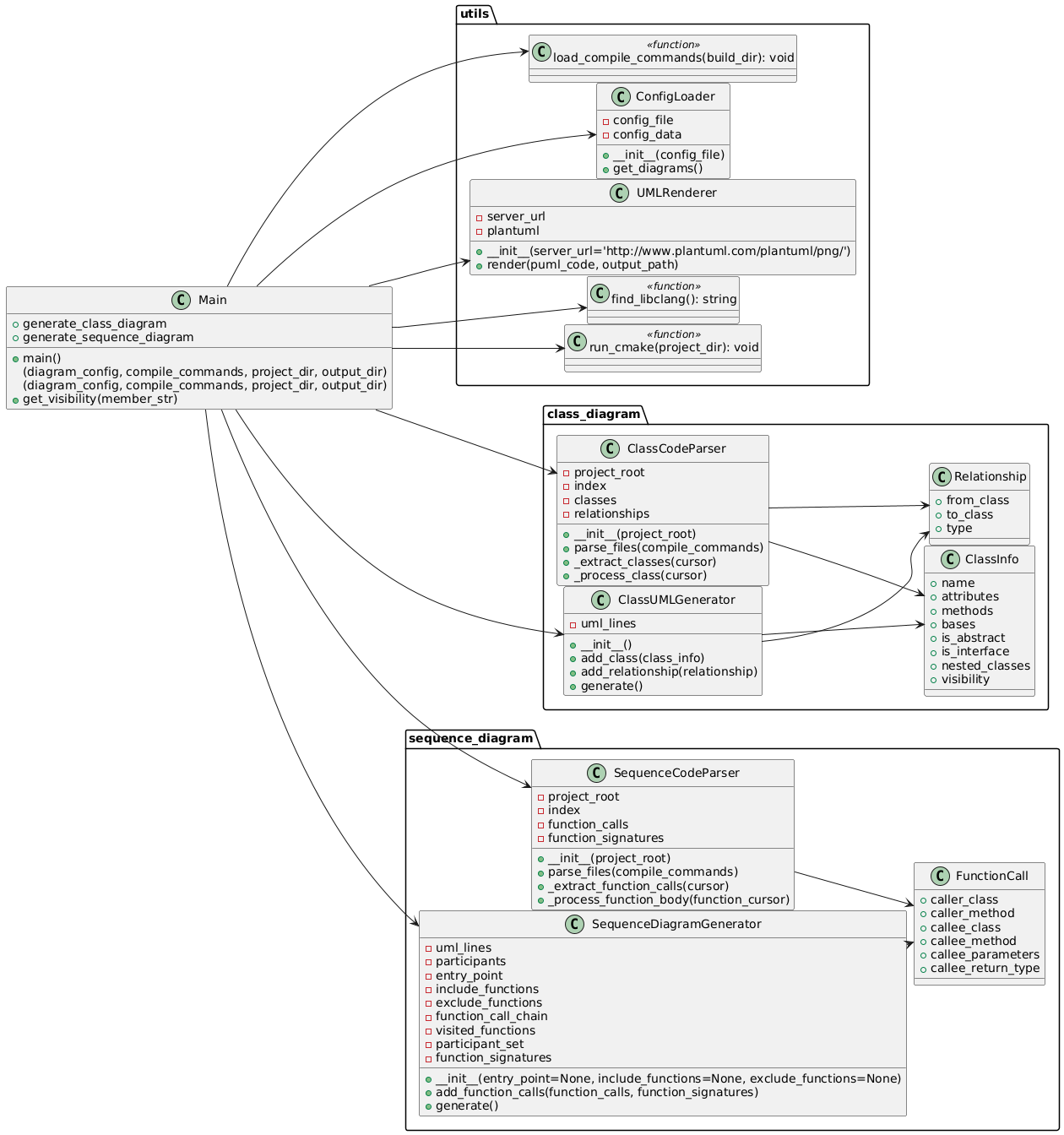


Рисунок 3.1. - Діаграма класів для проєкту

### Аналіз конфігурації diagrams.yaml

Конфігураційний файл diagrams.yaml використовується для визначення параметрів генерації діаграм. Він має такий формат:

diagrams:

- name: ClassDiagram

type: class

output: class\_diagram.png

options:

include\_classes:

exclude\_classes:

show\_relationships: true

- name: sequence\_diagram

type: sequence

output: sequence\_diagram.png

options:

include\_classes:

exclude\_classes:

include\_functions:

exclude\_functions:

entry\_point: main

У цьому файлі ми можемо визначити які діаграми нам необхідно згенерувати. Наразі підтримується два типи діаграм:

1. **ClassDiagram**: діаграма класів
2. **SequenceDiagram**: діаграма послідовності

### Параметри налаштування діаграм

Для всіх діаграм ми можемо відфільтрувати класи та функції які будуть згенеровані. Цього можна досягти за допомою параметрів:

* include\_classes: Масив класів які будуть включені в діаграми
* exclude\_classes: Масив класів які будуть виключені з діаграм..
* include\_functions: Масив функцій які будуть включені в діаграми.
* exclude\_functions: Масив функцій які будуть включені з діаграм.

Для цих параметрів також можна використати ключове слово “Global” щоб виключити всі стандартні та зовнішні виклики, оскільки для документування внутрішньої логіки системи вони не завжди потрібні. Також для діаграм класів є параметр show\_relationships, який відповідає за відображення зв’язків між класами, а для діаграм поведінки є обов’яковий параметр entry\_function, що показує з якої точки програми треба будувати діаграму.

### Завантаження конфігурації та парсинг

Скрипт config\_loader.py відповідає за завантаження та обробку конфігураційного файлу, це дозволяє програмі динамічно завантажувати налаштування діаграм та використовувати їх у подальшій обробці.

class ConfigLoader:

def \_\_init\_\_(self, config\_file):

self.config\_file = config\_file

def get\_diagrams(self):

with open(self.config\_file, 'r') as f:

config\_data = yaml.safe\_load(f)

return config\_data.get('diagrams', [])

### Структура main.py

Головний скрипт main.py відповідає за координацію всіх компонентів програми. Основні кроки, які виконує скрипт:

* **Обробка аргументів командного рядка**: отримання шляху до проєкту, конфігурації та ,за потреби, шляху до libclang.
* **Ініціалізація libclang**: встановлення шляху до бібліотеки перед імпортом clang.cindex.
* **Завантаження конфігурації**: використання ConfigLoader для отримання параметрів діаграм.
* **Запуск CMake та завантаження compile\_commands.json**: підготовка проєкту до аналізу.
* **Генерація діаграм**: виклик відповідних функцій для генерації діаграм класів та послідовностей.

### ****Обробка аргументів та ініціалізація**** libclang****:****

if len(sys.argv) > 3:

clang\_library\_path = sys.argv[3]

else:

clang\_library\_path = find\_libclang()

if not clang\_library\_path:

print("Unable to find libclang. Please specify the path explicitly.")

sys.exit(1)

clang.cindex.Config.set\_library\_file(clang\_library\_path)

### Виклик CMake та збір інформації про файли проєкту

Для коректного аналізу вихідного коду C++ необхідно мати файл compile\_commands.json, який містить інформацію про команди компіляції кожного файлу в проєкті. Цей файл генерується за допомогою CMake з опцією -DCMAKE\_EXPORT\_COMPILE\_COMMANDS=ON. Файл build\_tools.py, в модулі utils автоматизує процес виклику CMake та завантаження compile\_commands.json:

def run\_cmake(project\_dir):

build\_dir = os.path.join(project\_dir, 'build')

os.makedirs(build\_dir, exist\_ok=True)

subprocess.run(['cmake', '..', '-DCMAKE\_EXPORT\_COMPILE\_COMMANDS=ON'], cwd=build\_dir, check=True)

def load\_compile\_commands(build\_dir):

compile\_commands\_path = os.path.join(build\_dir, 'compile\_commands.json')

with open(compile\_commands\_path, 'r') as f:

compile\_commands = json.load(f)

return compile\_commands

У головному скрипті main.py виклик CMake та завантаження команд компіляції здійснюється наступним чином:

build\_tools.run\_cmake(project\_dir)

build\_dir = os.path.join(project\_dir, 'build')

compile\_commands = build\_tools.load\_compile\_commands(build\_dir)

Це забезпечує автоматизацію процесу підготовки проєкту до аналізу без необхідності ручного виклику CMake.

### ****Завантаження конфігурації та запуск генерації діаграм:****

config\_file = os.path.join(config\_dir, 'diagrams.yaml')

config\_loader = ConfigLoader(config\_file)

diagrams = config\_loader.get\_diagrams()

for diagram in diagrams:

diagram\_type = diagram.get('type')

if diagram\_type == 'class':

generate\_class\_diagram(diagram, compile\_commands, project\_dir, config\_dir)

elif diagram\_type == 'sequence':

generate\_sequence\_diagram(diagram, compile\_commands, project\_dir, config\_dir)

else:

print(f"Unknown diagram type: {diagram\_type}")

### Функції generate\_class\_diagram та generate\_sequence\_diagram

Ці функції відповідають за загальні виклики для створення діаграм. Вони визивають методи генерації з модулів class\_diagram та sequence\_diagram, виконують фільтрацію згідно з конфігурацією. В кінці, якщо є необхідність, використовують PlantUML для генерації графічного варіанту діаграми.

### Рендеринг діаграм та збереження зображень

Рендеринг діаграм здійснюється за допомогою бібліотеки PlantUml. Після генерації текстового представлення діаграми (файли .puml), рендерер перетворює їх у зображення (наприклад, .png).

class PlantUMLRenderer:

def \_\_init\_\_(self, server\_url='http://www.plantuml.com/plantuml/png/'):

self.server\_url = server\_url

self.plantuml = PlantUML(url=self.server\_url)

def render(self, puml\_code, output\_path):

try:

image\_data = self.plantuml.processes(puml\_code)

with open(output\_path, 'wb') as image\_file:

image\_file.write(image\_data)

except PlantUMLHTTPError as e:

print(f"Error rendering UML diagram: HTTP Error {e.response.status\_code} - {e.response.reason}")

print(f"Response content: {e.response.text}")

with open('debug\_diagram.puml', 'w') as debug\_file:

debug\_file.write(puml\_code)

except Exception as e:

print(f"Error rendering UML diagram: {str(e)}")

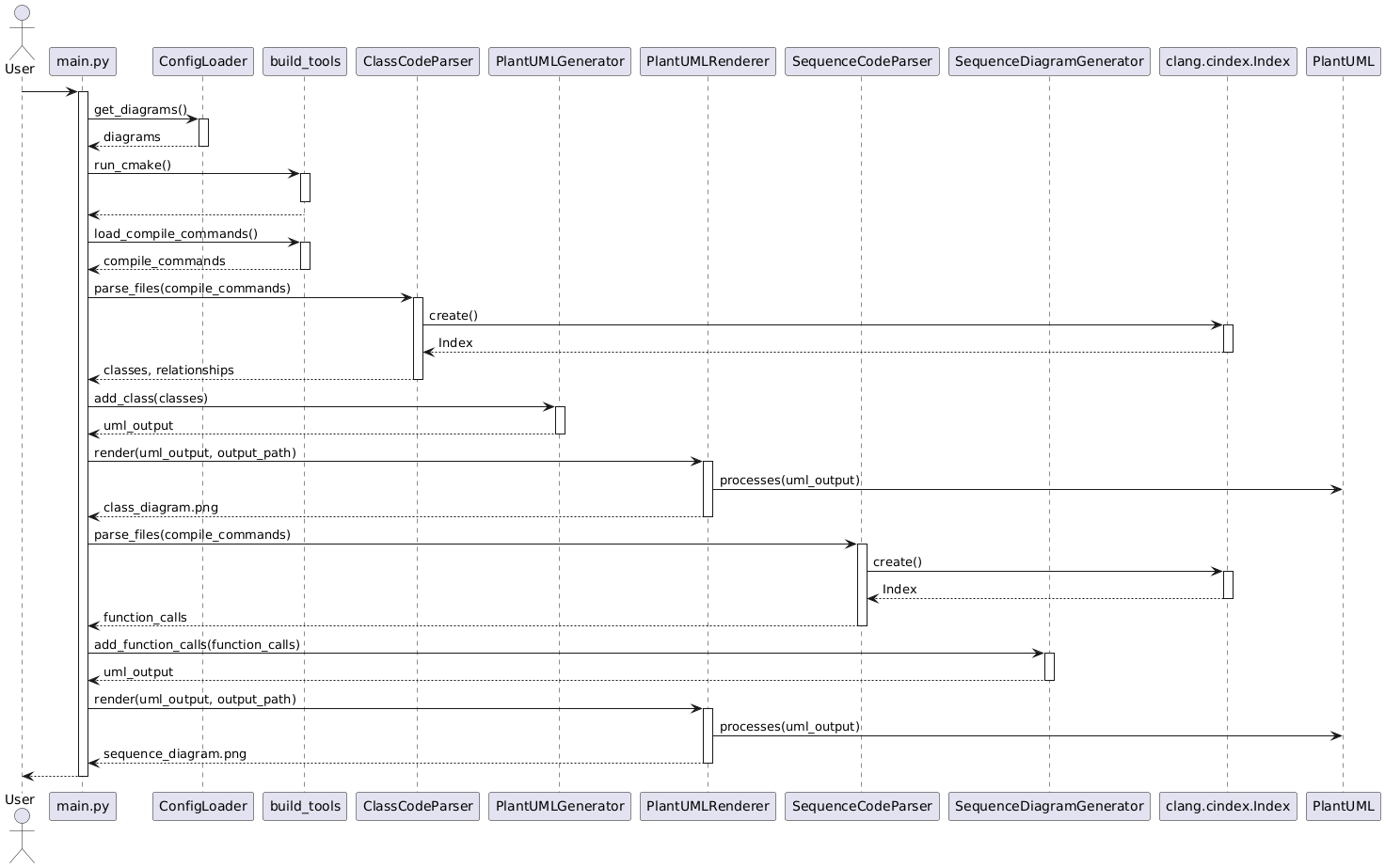


Рисунок 3.2. - Діаграма поведінки розробленої системи.

## 3.2. Розгляд алгоритму генерації діаграм класів

У цьому розділі ми детально розглянемо алгоритм генерації діаграм класів на основі вихідного коду C++. Особливу увагу буде приділено алгоритму обходу абстрактного синтаксичного дерева (AST), оскільки він є ключовим елементом у процесі аналізу коду та витягання необхідної інформації про класи, їх атрибути, методи та взаємозв'язки. Генерація діаграм класів базується на детальному аналізі вихідного коду та вилученні інформації про структуру класів, їх атрибути, методи та взаємозв'язки між ними. Для цього використовується AST, згенероване бібліотекою libclang.

Процес генерації діаграм класів складається з кількох основних етапів:

1. **Парсинг вихідного коду**: використання бібліотеки libclang для побудови AST вихідного коду C++.
2. **Обхід AST**: рекурсивний аналіз AST для виявлення оголошень класів, їх членів та взаємозв'язків.
3. **Збір інформації**: збереження зібраних даних у відповідних структурах даних для подальшої обробки.
4. **Генерація UML-коду**: створення текстового представлення діаграми у форматі PlantUML.
5. **Рендеринг діаграми**: перетворення текстового UML у графічне зображення.

### Парсинг вихідного коду та побудова AST

Для аналізу вихідного коду C++ ми використовуємо бібліотеку libclang, яка надає API для побудови та обходу AST.

**Ініціалізація індексу та парсинг файлів:**

class CodeParser:

def \_\_init\_\_(self, project\_root):

self.project\_root = Path(project\_root).resolve() # коренева директорія проєкту

self.index = clang.cindex.Index.create() # об'єкт Index, який використовується для парсингу вихідних файлів та побудови AST.

self.classes = {} # словник для зберігання інформації про класи

self.relationships = [] # список для зберігання взаємозв'язків між класами

**Парсинг файлів:**

def parse\_files(self, compile\_commands):

for entry in compile\_commands: список команд компіляції з compile\_commands.json

file\_path = entry['file']

command = entry.get('command')

if command:

arguments = command.split()

else:

continue # Пропустити файл без команди компіляції

filtered\_args = self.\_filter\_arguments(arguments) метод для фільтрації непотрібних аргументів

tu = self.index.parse(file\_path, args=filtered\_args) TranslationUnit, що представляє AST файлу

self.\_extract\_classes(tu.cursor) метод для початку обходу AST

### Алгоритм обходу AST для аналізу класів

Основна мета обходу AST — знайти всі оголошення класів, їх члени та взаємозв'язки з іншими класами. Обхід виконується рекурсивно, починаючи з кореневого вузла AST, використовуючи наступні правила:

* **Перевірка належності до проєкту**: ми аналізуємо тільки ті вузли, що належать до файлів проєкту.
* **Виявлення оголошень класів**: якщо вузол є оголошенням класу (CLASS\_DECL), ми передаємо його в метод \_process\_class.
* **Рекурсивний обхід**: для кожного дочірнього вузла ми повторюємо процес.
* **Перевірка на повторення**: якщо клас вже оброблений, ми його пропускаємо.
* **Створення ClassInfo**: об'єкт для зберігання інформації про клас.
* **Обробка базових класів**: виявлення успадкування та додавання взаємозв'язків.
* **Обробка членів класу**: збір атрибутів та методів з урахуванням їх видимості.

**Методи** \_extract\_classes та \_process\_class**:**

def \_extract\_classes(self, cursor):

for child in cursor.get\_children():

if child.location and child.location.file:

file\_path = Path(child.location.file.name).resolve()

try:

file\_path.relative\_to(self.project\_root)

except ValueError:

continue # Пропустити вузли, що не належать до проєкту

else:

continue

if child.kind == clang.cindex.CursorKind.CLASS\_DECL:

self.\_process\_class(child)

self.\_extract\_classes(child) # Рекурсивний виклик

def \_process\_class(self, cursor):

class\_name = cursor.spelling

if class\_name in self.classes:

return # Клас вже оброблений

class\_info = ClassInfo(name=class\_name, attributes=[], methods=[], bases=[], is\_abstract=False)

self.classes[class\_name] = class\_info

# Обробка базових класів (успадкування)

for base in cursor.get\_children():

if base.kind == clang.cindex.CursorKind.CXX\_BASE\_SPECIFIER:

base\_name = base.get\_definition().spelling

class\_info.bases.append(base\_name)

relationship = Relationship(from\_class=class\_name, to\_class=base\_name, type='inheritance')

self.relationships.append(relationship)

# Обробка членів класу

for member in cursor.get\_children():

if member.access\_specifier == clang.cindex.AccessSpecifier.PUBLIC:

visibility = '+'

elif member.access\_specifier == clang.cindex.AccessSpecifier.PROTECTED:

visibility = '#'

elif member.access\_specifier == clang.cindex.AccessSpecifier.PRIVATE:

visibility = '-'

else:

visibility = '~'

if member.kind == clang.cindex.CursorKind.FIELD\_DECL:

# Атрибут класу

type\_spelling = member.type.spelling

class\_info.attributes.append(f"{visibility} {member.spelling} : {type\_spelling}")

elif member.kind == clang.cindex.CursorKind.CXX\_METHOD:

# Метод класу

return\_type = member.result\_type.spelling

params = [param.type.spelling for param in member.get\_arguments()]

params\_str = ', '.join(params)

method\_signature = f"{visibility} {member.spelling}({params\_str}) : {return\_type}"

class\_info.methods.append(method\_signature)

### Збір інформації про взаємозв'язки

Взаємозв'язки між класами виявляються під час обробки базових класів та типів членів.

* **Успадкування:** Коли ми виявляємо, що клас успадковує від іншого, ми додаємо взаємозв'язок типу inheritance.
* **Агрегація та композиція:** Якщо атрибут класу має тип іншого класу, ми можемо додати взаємозв'язок типу aggregation або composition залежно від логіки програми.

**Приклад коду для додавання взаємозв'язку:**

if member.kind == clang.cindex.CursorKind.FIELD\_DECL:

type\_spelling = member.type.spelling

class\_info.attributes.append(f"{visibility} {member.spelling} : {type\_spelling}")

if type\_spelling in self.classes:

relationship = Relationship(from\_class=class\_name, to\_class=type\_spelling, type='aggregation')

self.relationships.append(relationship)

### Реалізація обробки областей видимості

Під час обходу абстрактного синтаксичного дерева (AST) ми аналізуємо кожен член класу та визначаємо його область видимості. Для цього ми використовуємо атрибут access\_specifier об'єкта Cursor, який представляє член класу.

**Частина коду з методу** \_process\_class відповідальна за обробку області видимості**:**

def \_process\_class(self, cursor):

# Обробка членів класу

for member in cursor.get\_children():

# Визначення області видимості

if member.access\_specifier == clang.cindex.AccessSpecifier.PUBLIC:

visibility = '+'

elif member.access\_specifier == clang.cindex.AccessSpecifier.PROTECTED:

visibility = '#'

elif member.access\_specifier == clang.cindex.AccessSpecifier.PRIVATE:

visibility = '-'

else:

visibility = '~' # Для випадків, коли область видимості не визначена

**Пояснення коду:**

**Отримання області видимості члена класу:**

Ми використовуємо member.access\_specifier, який може приймати такі значення: .AccessSpecifier.PUBLIC, AccessSpecifier.PROTECTED, AccessSpecifier.PRIVATE, AccessSpecifier.NONE. В залежності від значення access\_specifier, ми встановлюємо змінну visibility:

* '+' для public
* '#' для protected
* '-' для private

### ****Додавання інформації про атрибути та методи****

Для **атрибутів** (FIELD\_DECL) ми створюємо рядок, що включає:

* Символ видимості (visibility)
* Назву атрибуту (member.spelling)
* Тип атрибуту (type\_spelling)
* Формат: "{visibility} {name} : {type}"

Для **методів** (CXX\_METHOD) ми створюємо рядок в наступному форматі: "{visibility} {name}({parameters}) : {return\_type}" , що включає:

* Символ видимості (visibility)
* Назву методу (member.spelling)
* Список параметрів (params\_str)
* Тип повернення (return\_type)

### Визначення повних імен класів

Для уникнення колізій імен ми використовуємо повні імена класів, включаючи простори імен:

def \_get\_fully\_qualified\_name(self, cursor):

names = []

while cursor and cursor.kind != clang.cindex.CursorKind.TRANSLATION\_UNIT:

if cursor.kind in [clang.cindex.CursorKind.NAMESPACE, clang.cindex.CursorKind.CLASS\_DECL, clang.cindex.CursorKind.STRUCT\_DECL]:

name = cursor.spelling or cursor.displayname

if name:

names.append(name)

cursor = cursor.semantic\_parent

return '::'.join(reversed(names))

### Генерація UML-коду

Після того, як ми зібрали всю необхідну інформацію про класи та взаємозв'язки, ми використовуємо генератор UML для створення текстового представлення діаграми.

**Розглянемо реалізацію класу** PlantUMLGenerator, який має наступні методи**:**

* **Метод** add\_class: додає визначення класу до UML-коду.
* **Метод** add\_relationship: додає взаємозв'язки між класами з відповідною нотацією.
* **Метод** generate: повертає повний текст UML-коду.

class PlantUMLGenerator:

def \_\_init\_\_(self):

self.uml\_lines = []

def add\_class(self, class\_info):

self.uml\_lines.append(f"class {class\_info.name} {{")

for attribute in class\_info.attributes:

self.uml\_lines.append(f" {attribute}")

for method in class\_info.methods:

self.uml\_lines.append(f" {method}")

self.uml\_lines.append("}")

def add\_relationship(self, relationship):

if relationship.type == 'inheritance':

self.uml\_lines.append(f"{relationship.from\_class} --|> {relationship.to\_class}")

elif relationship.type == 'aggregation':

self.uml\_lines.append(f"{relationship.from\_class} o-- {relationship.to\_class}")

def generate(self):

return '\n'.join(self.uml\_lines)

### Рендеринг діаграми

Останнім кроком є рендеринг діаграми з текстового UML-коду у графічне зображення, реалізацію цього класу ми розглянули раніше, далі буде представлений приклад реалізації, де uml\_output - це текстове представлення діаграми у форматі PlantUML, а output\_image\_path - це шлях до файлу, де буде збережено зображення.

renderer = PlantUMLRenderer()

renderer.render(uml\_output, output\_image\_path)

Особливості реалізації:

* Обробка областей видимості:
  + Модифікатори доступу: використання символів '+', '-', '#' для відображення public, private, protected членів відповідно.
  + Відображення конструкторів та деструкторів: спеціальне позначення для цих методів.
* Підтримка шаблонних класів:
  + Параметри шаблонів: включення інформації про параметри шаблонів у назву класу.
  + Спеціалізації шаблонів: виявлення та обробка спеціалізованих версій шаблонних класів.
* Повне кваліфіковане ім'я класів:
  + Простори імен: включення інформації про простори імен для точного ідентифікування класів.
  + Уникнення колізій: забезпечення унікальності імен класів на діаграмі.
* Фільтрація та налаштування:
  + Виключення класів: можливість виключити певні класи або простори імен з аналізу.
  + Налаштування рівня деталізації: визначення, чи включати приватні атрибути та методи.

### Переваги підходу

* Точність: завдяки використанню AST забезпечується коректне вилучення інформації з вихідного коду.
* Актуальність: автоматична генерація діаграм при зміні коду гарантує, що документація завжди відповідає поточному стану проєкту.
* Гнучкість: можливість налаштовувати параметри генерації діаграм під потреби конкретного проєкту.

## 3.3 Огляд аналіз алгоритму генерації діаграм поведінки

У цьому розділі ми детально розглянемо алгоритм генерації діаграм поведінки, а саме діаграм послідовностей, на основі вихідного коду C++. Генерація діаграм послідовностей спрямована на відображення динамічної поведінки системи, показуючи, як різні об'єкти взаємодіють між собою під час виконання певного сценарію. Особливу увагу буде приділено алгоритму обходу абстрактного синтаксичного дерева (AST), який є ключовим елементом у процесі аналізу коду та витягання інформації про виклики функцій та їх взаємодію. Це дозволяє автоматично будувати діаграми послідовностей, що відображають динамічну поведінку системи під час виконання програми.

Процес генерації діаграм послідовностей складається з кількох основних етапів:

* **Парсинг вихідного коду**: використання бібліотеки libclang для побудови AST вихідного коду C++.
* **Обхід AST**: рекурсивний аналіз AST для виявлення викликів функцій та їх взаємозв'язків.
* **Збір інформації**: збереження зібраних даних про виклики функцій, їх параметри та типи повернення.
* **Побудова ланцюга викликів**: визначення послідовності викликів функцій, починаючи з точки входу (наприклад, функції main).
* **Генерація UML-коду**: створення текстового представлення діаграми послідовності у форматі PlantUML.
* **Рендеринг діаграми**: перетворення текстового UML у графічне зображення.

### Парсинг вихідного коду та побудова AST

Ініціалізація індексу:

class SequenceCodeParser:

def \_\_init\_\_(self, project\_root):

self.project\_root = Path(project\_root).resolve()

self.index = clang.cindex.Index.create()

self.function\_calls = []

self.function\_signatures = {}

Де: project\_root: - це коренева директорія проєкту. Об'єкт Index, використовується для парсингу вихідних файлів та побудови AST. function\_calls: список для зберігання інформації про виклики функцій. function\_signatures -це словник для зберігання сигнатур функцій (параметри та типи повернення).

**Реалізацція парсингу файлів:**

def parse\_files(self, compile\_commands):

for entry in compile\_commands:

file\_path = entry['file']

command = entry.get('command')

if command:

arguments = command.split()

else:

continue # Пропустити файл без команди компіляції

filtered\_args = self.\_filter\_arguments(arguments)

tu = self.index.parse(file\_path, args=filtered\_args)

self.\_extract\_function\_calls(tu.cursor)

Де: compile\_commands - це список команд компіляції з compile\_commands.json. \_filter\_arguments - це метод, який необхідний для фільтрації непотрібних аргументів компілятора. Змінна tu – це TranslationUnit, що представляє AST файлу. \_extract\_function\_calls - це метод для початку обходу AST.

### Алгоритм обходу AST для виявлення викликів функцій

Основна мета обходу AST — знайти всі виклики функцій та зібрати інформацію про них, включаючи класи та методи, що викликають та викликаються, параметри та типи повернення, за це відповідає **Метод** \_extract\_function\_calls**:**

def \_extract\_function\_calls(self, cursor):

for c in cursor.get\_children():

location = c.location

if location.file:

file\_path = Path(location.file.name).resolve()

try:

file\_path.relative\_to(self.project\_root)

except ValueError:

continue # Пропустити вузли, що не належать до проєкту

else:

continue

if c.kind in [clang.cindex.CursorKind.FUNCTION\_DECL, clang.cindex.CursorKind.CXX\_METHOD, clang.cindex.CursorKind.CONSTRUCTOR, clang.cindex.CursorKind.DESTRUCTOR]:

if c.is\_definition():

self.\_process\_function\_body(c)

self.\_extract\_function\_calls(c) # Рекурсивний виклик

**Виконуючи обхід ми притримуємось наступних правил:**

* **Перевірка належності до проєкту**: аналізуємо тільки ті вузли, що належать до файлів проєкту.
* **Виявлення визначень функцій та методів**: якщо вузол є визначенням функції або методу, передаємо його в метод \_process\_function\_body.
* **Рекурсивний обхід**: для кожного дочірнього вузла повторюємо процес.

**Розглянемо метод** \_process\_function\_body, який необхідний для**:**

* **Визначення класу викликача**: якщо функція належить класу або структурі, отримуємо повне ім'я класу; інакше позначаємо як 'Main' або 'Global'.
* **Збирання сигнатури функції**: отримуємо параметри та тип повернення функції.
* **Обхід тіла функції**: викликаємо метод \_visit\_function\_body для аналізу викликів всередині функції.

def \_process\_function\_body(self, function\_cursor):

# Визначення класу, до якого належить метод

if function\_cursor.semantic\_parent and function\_cursor.semantic\_parent.kind in [clang.cindex.CursorKind.CLASS\_DECL, clang.cindex.CursorKind.STRUCT\_DECL]:

caller\_class = self.\_get\_fully\_qualified\_name(function\_cursor.semantic\_parent)

else:

if function\_cursor.spelling == 'main':

caller\_class = 'Main' # Спеціальне позначення для функції main

else:

caller\_class = 'Global' # Для глобальних функцій

caller\_method = function\_cursor.spelling

# Збирання сигнатури функції

parameters = self.\_get\_function\_parameters(function\_cursor)

return\_type = self.\_get\_type\_spelling(function\_cursor.result\_type)

self.function\_signatures[(caller\_class, caller\_method)] = (parameters, return\_type)

# Обхід тіла функції для пошуку викликів

for c in function\_cursor.get\_children():

self.\_visit\_function\_body(c, caller\_class, caller\_method)

**Далі розглянемо** \_visit\_function\_body, що виконує наступні задачі**:**

* **Виявлення викликів функцій**: якщо вузол є викликом функції (CALL\_EXPR), отримуємо інформацію про викликану функцію.
* **Визначення класу викликаної функції**: аналогічно, отримуємо повне ім'я класу або позначаємо як 'Main' чи 'Global'.
* **Збирання параметрів та типу повернення**: отримуємо сигнатуру викликаної функції.
* **Збереження інформації про виклик**: створюємо об'єкт FunctionCall та додаємо його до списку function\_calls.
* **Рекурсивний обхід**: аналізує дочірні вузли для виявлення вкладених викликів.

def \_visit\_function\_body(self, cursor, caller\_class, caller\_method):

if cursor.kind == clang.cindex.CursorKind.CALL\_EXPR:

called\_cursor = cursor.referenced

if called\_cursor:

callee\_method = called\_cursor.spelling

if called\_cursor.semantic\_parent and called\_cursor.semantic\_parent.kind in [clang.cindex.CursorKind.CLASS\_DECL, clang.cindex.CursorKind.STRUCT\_DECL]:

callee\_class = self.\_get\_fully\_qualified\_name(called\_cursor.semantic\_parent)

else:

if callee\_method == 'main':

callee\_class = 'Main'

else:

callee\_class = 'Global'

# Збирання параметрів та типу повернення викликаної функції

callee\_parameters = self.\_get\_function\_parameters(called\_cursor)

callee\_return\_type = self.\_get\_type\_spelling(called\_cursor.result\_type)

# Збереження виклику функції

function\_call = FunctionCall(

caller\_class=caller\_class,

caller\_method=caller\_method,

callee\_class=callee\_class,

callee\_method=callee\_method,

callee\_parameters=callee\_parameters,

callee\_return\_type=callee\_return\_type

)

self.function\_calls.append(function\_call)

for c in cursor.get\_children():

self.\_visit\_function\_body(c, caller\_class, caller\_method)

**Допоміжні методи для отримання повних імен функцій, їх параметрів та типів ціх параметрів, а також типу повертаємого значення:**

def \_get\_fully\_qualified\_name(self, cursor):

names = []

while cursor and cursor.kind != clang.cindex.CursorKind.TRANSLATION\_UNIT:

if cursor.kind in [clang.cindex.CursorKind.NAMESPACE, clang.cindex.CursorKind.CLASS\_DECL, clang.cindex.CursorKind.STRUCT\_DECL]:

name = cursor.spelling or cursor.displayname

if name:

names.append(name)

cursor = cursor.semantic\_parent

return '::'.join(reversed(names)) if names else 'Global'

def \_get\_function\_parameters(self, func\_cursor):

param\_types = []

for param in func\_cursor.get\_arguments():

param\_type = self.\_get\_type\_spelling(param.type)

param\_types.append(param\_type)

return ', '.join(param\_types)

def \_get\_type\_spelling(self, ctype):

if ctype.kind == clang.cindex.TypeKind.POINTER:

pointee = ctype.get\_pointee()

return self.\_get\_type\_spelling(pointee) + '\*'

elif ctype.kind == clang.cindex.TypeKind.LVALUEREFERENCE:

pointee = ctype.get\_pointee()

return self.\_get\_type\_spelling(pointee) + '&'

elif ctype.kind == clang.cindex.TypeKind.RVALUEREFERENCE:

pointee = ctype.get\_pointee()

return self.\_get\_type\_spelling(pointee) + '&&'

else:

return ctype.spelling

Під час обходу AST ми зберігаємо інформацію про кожен виклик функції у вигляді об'єктів FunctionCall, які містять всі необхідні параметри для подальшої побудови діаграми, а саме:

* **caller\_class**: клас або контекст, з якого здійснюється виклик.
* **caller\_method**: метод або функція, яка здійснює виклик.
* **callee\_class**: клас, до якого належить викликана функція.
* **callee\_method**: назва викликаної функції.
* **callee\_parameters**: параметри викликаної функції.
* **callee\_return\_type**: тип повернення викликаної функції.

### Побудова ланцюга викликів

Після збирання всіх викликів функцій ми будуємо ланцюг викликів, починаючи з точки входу (наприклад, функції main). Це дозволяє визначити послідовність викликів та виключити непотрібні функції, які не пов'язані з основним потоком виконання, для цього був реалізований к**лас** SequenceDiagramGenerator**. Також, щ**об не перевантажувати діаграму викликами стандартних функцій або функцій бібліотек, ми можемо виключати їх на етапі парсингу або побудови ланцюга викликів.

class SequenceDiagramGenerator:

def \_\_init\_\_(self, entry\_point=None):

self.uml\_lines = []

self.participants = []

self.entry\_point = entry\_point

self.function\_call\_chain = []

self.visited\_functions = set()

self.function\_signatures = {}

def add\_function\_calls(self, function\_calls, function\_signatures):

self.function\_signatures = function\_signatures

if self.entry\_point:

self.\_build\_call\_chain(function\_calls, self.entry\_point)

calls\_to\_include = self.function\_call\_chain

else:

calls\_to\_include = function\_calls

# Збирання учасників у порядку виклику

for call in calls\_to\_include:

if call.caller\_class not in self.participants:

self.participants.append(call.caller\_class)

if call.callee\_class not in self.participants:

self.participants.append(call.callee\_class)

# Генерація UML-коду

self.uml\_lines.append('@startuml')

self.uml\_lines.append('actor Client')

for participant in self.participants:

self.uml\_lines.append(f'participant {participant}')

# Додатковий код для відображення викликів з параметрами та типами повернення

# ...

self.uml\_lines.append('@enduml')

**Для побудови ланцюга викликів** рекурсивно додаємо виклики функцій, починаючи з точки входу, також, для **уникнення повторів**: використовуємо множину visited\_functions для запобігання циклічним викликам та повторенням. За цю логіку відповідає **Метод** \_build\_call\_chain**:**

def \_build\_call\_chain(self, function\_calls, current\_function):

for call in function\_calls:

if (call.caller\_class, call.caller\_method) == current\_function:

if (call.callee\_class, call.callee\_method) not in self.visited\_functions:

self.function\_call\_chain.append(call)

self.visited\_functions.add((call.callee\_class, call.callee\_method))

self.\_build\_call\_chain(function\_calls, (call.callee\_class, call.callee\_method))

### Генерація UML-коду для діаграми послідовності

Після побудови ланцюга викликів ми генеруємо текстове представлення діаграми послідовності у форматі PlantUML, розглянемо процес д**одавання учасників та повідомлень:**

def add\_function\_calls(self, function\_calls, function\_signatures):

# ... код для побудови ланцюга викликів та збирання учасників ...

# Генерація UML-коду

self.uml\_lines.append('@startuml')

self.uml\_lines.append('actor Client')

for participant in self.participants:

self.uml\_lines.append(f'participant {participant}')

# Виклик точки входу

if self.entry\_point:

entry\_parameters, \_ = self.function\_signatures.get(self.entry\_point, ('', ''))

self.uml\_lines.append(f'Client -> {self.entry\_point[0]} : {self.entry\_point[1]}({entry\_parameters})')

self.uml\_lines.append(f'activate {self.entry\_point[0]}')

# Додавання викликів функцій

for call in self.function\_call\_chain:

callee\_parameters = call.callee\_parameters

callee\_return\_type = call.callee\_return\_type

message = f'{call.callee\_method}({callee\_parameters})'

return\_message = f'return {callee\_return\_type}' if callee\_return\_type else ''

self.uml\_lines.append(f'{call.caller\_class} -> {call.callee\_class} : {message}')

self.uml\_lines.append(f'activate {call.callee\_class}')

self.uml\_lines.append(f'{call.callee\_class} --> {call.caller\_class} : {return\_message}')

self.uml\_lines.append(f'deactivate {call.callee\_class}')

# Завершення діаграми

if self.entry\_point:

self.uml\_lines.append(f'{self.entry\_point[0]} --> Client')

self.uml\_lines.append(f'deactivate {self.entry\_point[0]}')

self.uml\_lines.append('@enduml')

**В процесі виконання даного алгоритму ми отримаємо наступні елементи діаграми**

* **Учасники**: додаємо акторів та учасників у порядку їх появи.
* **Повідомлення**: для кожного виклику функції додаємо стрілки з відповідними повідомленнями.
* **Активація та деактивація**: використовуємо activate та deactivate для відображення активації учасників.
* **Типи повернення**: додаємо повідомлення про повернення з функції з зазначенням типу.

### Особливості реалізації

* Виявлення викликів функцій та методів
  + Обробка статичних та динамічних викликів: включаючи виклики через вказівники на функції та лямбда-вирази.
  + Розпізнавання конструкторів та деструкторів: спеціальне відображення на діаграмі.
* Обробка умовних операторів та циклів
  + Використання конструкцій alt, loop у PlantUML: для відображення альтернативних шляхів виконання та повторюваних дій.
  + Аналіз AST для виявлення умовних конструкцій: ідентифікація IfStmt, ForStmt, WhileStmt та інших вузлів.
* Фільтрація викликів
  + Виключення викликів до стандартних бібліотек: щоб не перевантажувати діаграму незначними деталями.
  + Включення тільки важливих для аналізу функцій: на основі налаштувань конфігурації.

### Переваги підходу

* **Динамічне відображення поведінки системи:** допомагає зрозуміти, як компоненти взаємодіють у процесі виконання програми.
* **Автоматизація:** знімає потребу у ручному створенні складних діаграм послідовності.
* **Налаштовуваність:** дозволяє адаптувати діаграми під потреби конкретного сценарію або вимог.

### Виклики та рішення

* Складність аналізу великих кодових баз
  + Оптимізація алгоритмів: використання ефективних структур даних та алгоритмів для обробки великого обсягу інформації.
  + Паралелізація обробки: можливість розподілу аналізу між кількома потоками або процесами.
* Точність виявлення взаємодій
  + Глибокий аналіз AST: забезпечує коректне розпізнавання всіх викликів та умов.
  + Обробка специфічних конструкцій C++: таких як шаблони, перевантаження операторів, лямбда-вирази.

## Висновки до розділу

У цьому розділі було детально розглянуто структуру розробленої системи та використані алгоритми для автоматичної генерації UML діаграм на основі вихідного коду C++.

По-перше, було описано загальну архітектуру проєкту, який складається з трьох основних модулів: class\_diagram, sequence\_diagram та utils. Така модульна структура забезпечує гнучкість та розширюваність системи, дозволяючи легко додавати нові функціональні можливості або вдосконалювати існуючі.

По-друге, було детально проаналізовано кожен з модулів. Модуль class\_diagram відповідає за генерацію діаграм класів, використовуючи бібліотеку Clang для парсингу коду та збору інформації про класи, їх атрибути та взаємозв'язки. Модуль sequence\_diagram займається генерацією діаграм послідовностей, аналізуючи виклики функцій та методів для відображення динамічної поведінки системи. Модуль utils надає допоміжні функції для завантаження конфігурації, роботи з CMake та Clang, а також рендерингу діаграм за допомогою PlantUML.

Третім аспектом було розглянуто алгоритми генерації діаграм класів та послідовностей. Детально описано процеси парсингу вихідного коду, обходу абстрактного синтаксичного дерева (AST) та збору необхідної інформації. Особливу увагу приділено методам обробки областей видимості, визначенню повних імен класів та функцій, а також виявленню взаємозв'язків між класами.

Крім того, було досліджено методи генерації UML-коду та його рендерингу у графічні зображення. Використання PlantUML дозволяє легко конвертувати текстове представлення діаграм у зручні для аналізу та документування графічні формати.

Нарешті, було проаналізовано конфігураційні можливості системи, що забезпечують гнучкість та налаштовуваність генерації діаграм. Використання файлу diagrams.yaml дозволяє користувачам визначати параметри генерації, такі як типи діаграм, фільтри для включення або виключення певних класів та функцій, точки входу для діаграм послідовностей тощо.

Таким чином, розроблена система успішно реалізує поставлену задачу автоматичної генерації UML діаграм з вихідного коду C++. Вона поєднує в собі потужні інструменти аналізу коду (Clang), управління збіркою (CMake) та генерації діаграм (PlantUML), що дозволяє значно спростити процес документування та аналізу програмних систем. Відкрита архітектура та використання мови Python забезпечують легкість використання та можливість подальшого розширення системи відповідно до потреб користувачів.

# 4. ТЕСТУВАННЯ РОЗРОБЛЕНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ТА АНАЛІЗ ОТРИМАНИХ РЕЗУЛЬТАТІВ

## 4.1. Виконання програми на тестовому проєкті C++

Для демонстрації можливостей розробленої системи автоматизованої генерації UML діаграм необхідно обрати тестовий проєкт на мові C++, який містить різноманітні елементи мови, такі як класи, наслідування, шаблони, простори імен, перевантаження операторів та інше.

Створимо простий проєкт, який моделює систему управління унижною бібліотекою. Проєкт містить такі компоненти:

* **Простір імен** LibrarySystem: містить всі класи системи.
* **Клас** Person: базовий клас для представлення людини.
* **Клас** Member: наслідуєся від Person, представляє члена бібліотеки.
* **Клас** Librarian: наслідується від Person, представляє бібліотекаря.
* **Шаблонний клас** Book<T>: представляє книгу, де T — тип ідентифікатора книги.
* **Клас** Library: містить методи для додавання та видалення книг, реєстрації членів тощо.

### Налаштування середовища та діаграм

**Для виконання програми необхідно мати інтерпретатор python3 та встановлену бібліотеку CLANG. Для тестового використання розгянемо проєкт з 2 діаграмами, а саме діаграмою класів та викликів, з наступною конфігурацією diagrams.yaml:**

**diagrams:**

**- name: ClassDiagram**

**type: class**

**output: class\_diagram.png**

**options:**

**show\_relationships: true**

**- name: SequenceDiagram**

**type: sequence**

**output: sequence\_diagram.png**

**options:**

**include\_functions:**

**exclude\_functions:**

**exclude\_classes:**

**- Global**

**entry\_point: Library::** requestBook

**На виході очікуємо діаграму класів з всіма класами нашого проєкту і діаграму викликів, стартовою точкою якої є: Library::** requestBook.

**Використовуючи командий рядок запустимо генерацію, input/test2 – це шлях до нашого тестового проєкту, який містить в собі директорі. doc, в якій зберігається файл конфігурації, і туди будуть зберігатись діаграми, приклад запуску можна побачити на рисунку 4.1.**

****

Рисунок 4.1. - Запуск генерації

## 4.2. Огляд згенерованої діаграми класів

Для діаграми класів використовуємо в основному .h файли, тому розглянемо заголовочні файли тестового проєкту. Макроси включення файлів були прибрані, задля зменшення обсягу коду.

Person.h:

namespace LibrarySystem {

class Person {

protected:

std::string name;

int age;

public:

Person(const std::string& name, int age);

virtual ~Person();

virtual void displayInfo() const;

};

} // namespace LibrarySystem

Member.h:

namespace LibrarySystem {

class Member : public Person {

private:

int memberID;

public:

Member(const std::string& name, int age, int memberID);

void displayInfo() const override;

int getMemberID() const;

};

} // namespace LibrarySystem

Librarian.h:

namespace LibrarySystem {

class Librarian : public Person {

public:

Librarian(const std::string& name, int age);

void displayInfo() const override;

void manageLibrary();

};

} // namespace LibrarySystem

Book.h:

namespace LibrarySystem {

template <typename T>

class Book {

private:

T id;

std::string title;

std::string author;

public:

Book(T id, const std::string& title, const std::string& author)

: id(id), title(title), author(author) {}

T getID() const {

return id;

}

void displayInfo() const {

std::cout << "Book ID: " << id << ", Title: " << title

<< ", Author: " << author << std::endl;

}

};

} // namespace LibrarySystem

Library.h:

#pragma once

#include <vector>

#include <memory>

#include "Book.h"

#include "Member.h"

namespace LibrarySystem {

class Library {

private:

std::vector<Book<int>> books;

std::vector<std::shared\_ptr<Member>> members;

public:

void addBook(const Book<int>& book);

void removeBook(int bookID);

void registerMember(const std::shared\_ptr<Member>& member);

void displayBooks() const;

void displayMembers() const;

};

} // namespace LibrarySystem

### Наявні взаємозв'язки у проєкті

Member наслідується від Person: Це означає, що Member є підкласом Person і успадковує всі його властивості та методи. Таким чином, кожен Member має ім'я та вік, як і будь-яка Person, але також має додаткові атрибути та методи, специфічні для членів бібліотеки. Librarian наслідується від Person: Аналогічно, Librarian є підкласом Person, що представляє бібліотекаря. Він успадковує базові атрибути Person та додає специфічні методи, наприклад manageLibrary().

Library з Book<int> має тип зв’язку – агрегація. Зв'язок позначений як Library "1" \*-- "0..\*" "Book<int>", що означає, що один об'єкт Library може містити нуль або більше об'єктів Book<int>. Це вказує на те, що бібліотека містить колекцію книг, але книги можуть існувати незалежно від бібліотеки.

Library агрегація з Member: Зв'язок Library "1" \*-- "0..\*" Member показує, що бібліотека має нуль або більше зареєстрованих членів. Це відображає відносини між бібліотекою та її користувачами.

"Book<T>" залежить від T: Зв'язок "Book<T>" ..> T : template parameter вказує на те, що T є параметром шаблону для класу Book<T>. Це означає, що Book<T> може бути інстанційованим з різними типами T, що визначає тип ідентифікатора книги.

### Огляд згенерованого коду діаграми

@startuml

namespace LibrarySystem {

class Person {

- name : std::string

- age : int

+ Person(name : std::string, age : int)

+ ~Person()

+ displayInfo() const : void

}

class Member {

- memberID : int

+ Member(name : std::string, age : int, memberID : int)

+ displayInfo() const : void

+ getMemberID() const : int

}

class Librarian {

+ Librarian(name : std::string, age : int)

+ displayInfo() const : void

+ manageLibrary() : void

}

class "Book<T>" {

- id : T

- title : std::string

- author : std::string

+ Book(id : T, title : std::string, author : std::string)

+ getID() const : T

+ displayInfo() const : void

}

class Library {

- books : std::vector<Book<int>>

- members : std::vector<std::shared\_ptr<Member>>

+ addBook(book : Book<int>) : void

+ removeBook(bookID : int) : void

+ registerMember(member : std::shared\_ptr<Member>) : void

+ displayBooks() const : void

+ displayMembers() const : void

}

Person <|-- Member

Person <|-- Librarian

Library "1" \*-- "0..\*" "Book<int>"

Library "1" \*-- "0..\*" Member

"Book<T>" ..> T : template parameter

}

@enduml

Проаналізувавши код діаграми можемо зрозуміти, що генерація пройшла коректно та зрозуміти структуру тестового проєкту без детального аналізу коду. Оглянемо представлені в діаграмі класи та їх атрибути:

* Person: Базовий клас з атрибутами name та age, конструктором, деструктором та методом displayInfo().
* Member: Підклас Person з додатковим атрибутом memberID та методами getMemberID() та перевизначеним displayInfo().
* Librarian: Підклас Person з методом manageLibrary() та перевизначеним displayInfo().
* Book<T>: Шаблонний клас з атрибутами id, title, author та методами getID() та displayInfo().
* Library: Клас з атрибутами books та members, що представляють колекції книг та членів, та методами для управління цими колекціями.

Також можемо оглянути відносини між класами:

* Наслідування: Стрілки з порожнім трикутником (<|--) від Member та Librarian до Person вказують на відносини наслідування.
* Агрегація: Лінії з ромбом на боці Library (\*--) до Book<int> та Member показують, що Library агрегує множину книг та членів.
* Залежність: Пунктирна стрілка (..>) від "Book<T>" до T з позначкою template parameter вказує на те, що Book<T> залежить від параметра шаблону T.

Всі ці конструкції та взаємозв’язки можна побачити на згенерованій графічні версії діаграми на рисунку 4.2.

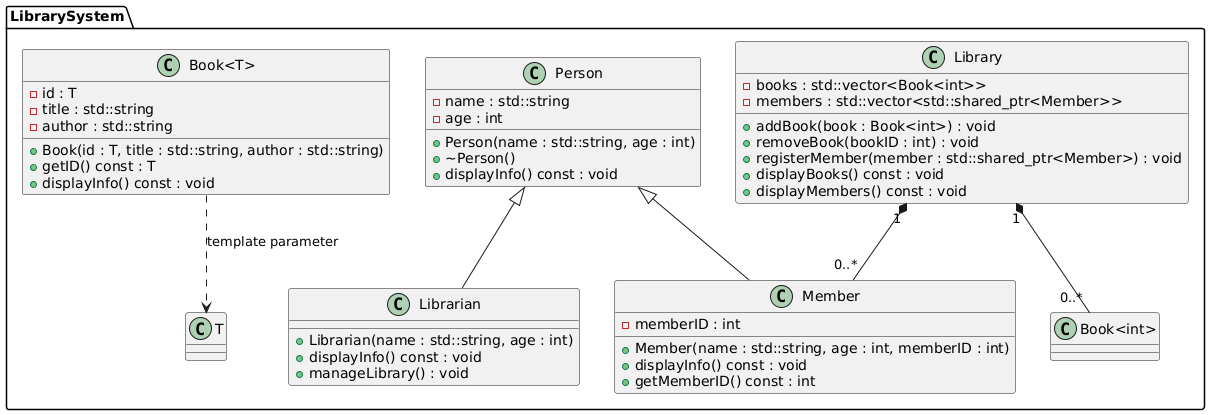


Рисунок 4.2. - Згенерована діаграма класів

## 4.3. Огляд згенерованої діаграми поведінки

Для розгляду генерації діаграм викликів нам необхідно оглянути .cpp файли з реалізацією цих викликів:

Person.cpp

namespace LibrarySystem {

Person::Person(const std::string& name, int age) : name(name), age(age) {}

Person::~Person() {}

void Person::displayInfo() const {

std::cout << "Name: " << name << ", Age: " << age << std::endl;

}

} // namespace LibrarySystem

Member.cpp

namespace LibrarySystem {

Member::Member(const std::string& name, int age, int memberID)

: Person(name, age), memberID(memberID) {}

void Member::displayInfo() const {

Person::displayInfo();

std::cout << "Member ID: " << memberID << std::endl;

}

int Member::getMemberID() const {

return memberID;

}

} // namespace LibrarySystem

Librarian.cpp

namespace LibrarySystem {

Librarian::Librarian(const std::string& name, int age)

: Person(name, age) {}

void Librarian::displayInfo() const {

Person::displayInfo();

std::cout << "Position: Librarian" << std::endl;

}

void Librarian::manageLibrary() {

std::cout << name << " is managing the library." << std::endl;

}

} // namespace LibrarySystem

Book.h: Оскільки шаблонні класи не можна розділяти на .h та .cpp файли, весь код шаблонного класу Book<T> повинен бути розміщений у файлі заголовку Book.h.

namespace LibrarySystem {

template <typename T>

class Book {

private:

T id;

std::string title;

std::string author;

public:

Book(T id, const std::string& title, const std::string& author)

: id(id), title(title), author(author) {}

T getID() const {

return id;

}

void displayInfo() const {

std::cout << "Book ID: " << id << ", Title: " << title

<< ", Author: " << author << std::endl;

}

};

} // namespace LibrarySystem

Library.cpp

namespace LibrarySystem {

void Library::addBook(const Book<int>& book) {

books.push\_back(book);

}

void Library::removeBook(int bookID) {

books.erase(std::remove\_if(books.begin(), books.end(),

[bookID](const Book<int>& b) { return b.getID() == bookID; }), books.end());

}

void Library::registerMember(const std::shared\_ptr<Member>& member) {

members.push\_back(member);

}

void Library::displayBooks() const {

for (const auto& book : books) {

book.displayInfo();

}

}

void Library::displayMembers() const {

for (const auto& member : members) {

member->displayInfo();

}

}

} // namespace LibrarySystem

Зазвичай для побудови діаграм поведінки використовують якусь функцію входу, з реалізацією певного функціоналу. В нашому випадку це функціонал, коли певний користувач хоче взяти книгу в бібліотеці. Тому точкою входу у нас є функція borrowBook, яка повинна видати книгу при її наявності, або повідомлення про відсутність. Нижче наведений код згенерованої діаграми для даної логіки.

### Огляд згенерованого коду діаграми

@startuml

actor Member as m

participant Library as l

participant "Book<int>" as b

m -> l : borrowBook(bookID)

activate l

l -> b : checkAvailability(bookID)

activate b

b --> l : availabilityStatus

deactivate b

alt [available]

l -> m : issueBook(bookID)

else [not available]

l -> m : informNotAvailable()

end

deactivate l

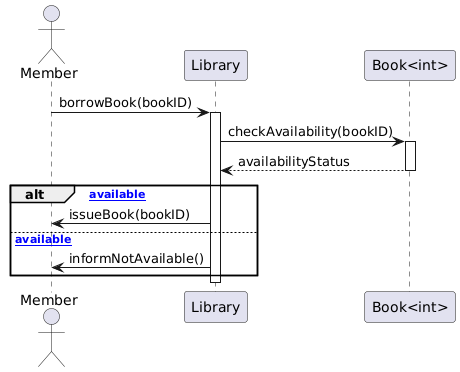
@enduml

**В даній діаграмі** Member — це член бібліотеки, який ініціює процес позичання книги. Також є 2 **учасники -** — бібліотека, що обробляє запит(Library) та необхідна книга з ідентифікатором (Book<int>).

**Послідовність дій виконання програми наступна:**

1. Member надсилає запит borrowBook до Library з ідентифікатором книги.
2. Library активується та звертається до Book<int> методом checkAvailability для перевірки наявності книги.
3. Book<int> повертає availabilityStatus до Library.
4. Якщо книга **доступна**, Library видає книгу Member методом issueBook.
5. Якщо книга **недоступна**, Library інформує Member про відсутність книги методом informNotAvailable.
6. Library деактивується після завершення процесу.

Згенеровану діаграму можна оглянути на Рисунку 4.3.



**Рисунок 4.3. -** Діаграма послідовності.

## Висновки до розділу

У цьому розділі було проведено практичне застосування розробленої системи на прикладі тестового проєкту, написаного мовою C++. Метою цього було продемонструвати реальну ефективність та функціональність інструменту в умовах, близьких до реальних. Були розглянуті вимоги до середовища, процес налаштування та запуску програмної системи, а також детально описано кроки, необхідні для інтеграції інструменту в існуючий проєкт.

Під час експерименту ми успішно згенерували UML діаграми класів та послідовностей, які точно відображають структуру та поведінку системи. Зокрема, діаграма класів відобразила всі основні класи проєкту, їх атрибути, методи та взаємозв'язки, включаючи наслідування, агрегацію та залежності. Діаграма послідовностей показала динамічну взаємодію між об'єктами під час виконання певного сценарію, відображаючи виклики методів та обмін повідомленнями між компонентами системи.

### Аналіз результатів

Результати підтверджують ефективність та практичність запропонованого підходу, заснованого на використанні Clang AST, Python та PlantUML. Використання AST дозволило точно вилучити необхідну інформацію з вихідного коду, включаючи складні конструкції C++, такі як шаблони, простори імен та перевантаження операторів. Інтеграція з PlantUML забезпечила швидку та гнучку генерацію діаграм у зручному для розробників форматі.

# Висновки

У сучасному світі програмного забезпечення все більшу увагу приділяють автоматизації процесів розробки та підтримки коду. Зі збільшенням складності програмних систем виникає необхідність у засобах, які дозволяють розробникам ефективно розуміти, аналізувати та документувати код. У цій роботі було розглянуто проблему автоматизації генерації UML-діаграм, що є актуальною задачею для багатьох проєктів.

У ході дослідження було проаналізовано існуючі методи документування коду та генерації UML-діаграм, визначено їхні недоліки та обмеження. Зокрема, виявлено, що ручне створення діаграм є трудомістким і схильним до помилок процесом, а наявні автоматизовані інструменти часто не підтримують специфічні особливості C++, такі як шаблони та простори імен.

Розроблене програмне забезпечення пропонує унікальний підхід до автоматизованої генерації UML-діаграм класів та послідовностей шляхом обходу абстрактного синтаксичного дерева (AST), згенерованого бібліотекою Clang. Використання мови програмування Python для обробки AST та інтеграція з PlantUML для генерації діаграм дозволили створити гнучку та ефективну систему. Запропонований підхід забезпечує коректну обробку складних конструкцій мови C++, підтримує сучасні стандарти та легко інтегрується з існуючими процесами розробки.

### Подальші перспективи

Перспективи подальших досліджень у цій галузі є досить обширними. По-перше, можливим є розширення системи для підтримки інших мов програмування, на основі С, що зробить її універсальним інструментом для різних команд розробників. По-друге, інтеграція з інструментами статичного аналізу коду може надати додаткові можливості для виявлення потенційних помилок та оптимізації коду.

Також перспективним є розвиток можливостей налаштування генерації діаграм, додавання підтримки інших типів діаграм UML, таких як діаграми станів або активності. Можна впровадити більш гнучкі механізми фільтрації та абстрагування, що дозволить розробникам зосередитися на найбільш важливих аспектах системи.

### Рекомендації щодо використання

Розроблене рішення найкраще підходить для середніх та великих проєктів на C++, де складність коду та кількість взаємозв'язків між компонентами ускладнюють ручне документування. Автоматизована генерація UML-діаграм буде особливо корисною для команд, які працюють над проєктами з активною фазою розвитку та частими змінами в кодовій базі. Інтеграція з системами контролю версій та процесами CI/CD дозволить підтримувати документацію в актуальному стані без додаткових зусиль.

Для освітніх проєктів та навчальних закладів це програмне забезпечення може служити інструментом для вивчення структури програм та принципів об'єктно-орієнтованого програмування. Воно допоможе студентам краще розуміти взаємозв'язки між класами та динаміку виконання програм.

### Підсумки

Загалом, розроблене програмне забезпечення вирішує актуальну проблему автоматизації документування коду та сприяє підвищенню ефективності розробки програмного забезпечення. Воно поєднує в собі сучасні інструменти та методи, що дозволяють зменшити технічний борг, покращити комунікацію в команді та забезпечити високу якість кінцевого продукту. Подальший розвиток системи та її адаптація до потреб різних проєктів відкривають широкі можливості для покращення процесів розробки в майбутньому.

# Список використаних джерел

1. de Souza, S. C. B., Anquetil, N., & de Oliveira, K. M. A study of the documentation essential to software maintenance // Proceedings of the 23rd Annual International Conference on Design of Communication: Documenting & Designing for Pervasive Information. — 2005. — С. 68–75.
2. Treude, C., & Storey, M.-A. Work item tagging: Communicating concerns in collaborative software development // IEEE Transactions on Software Engineering. — 2010. — Т. 36, №1. — С. 1–13.
3. Fagerholm, F., & Münch, J. Developer experience: Concept and definition // International Conference on Software and System Process. — 2012. — С. 73–77.
4. Tan, L., Yuan, D., Krishna, G., & Zhou, Y. iComment: Bugs or bad comments? // Proceedings of the 21st ACM Symposium on Operating Systems Principles. — 2007. — С. 145–158.
5. Pressman, R. S. Software Engineering: A Practitioner's Approach. 6-е вид. — McGraw-Hill, 2005. — 992 с.
6. Booch, G., Rumbaugh, J., & Jacobson, I. The Unified Modeling Language User Guide. 2-е вид. — Addison-Wesley, 2005. — 496 с.
7. Fowler, M. UML Distilled: A Brief Guide to the Standard Object Modeling Language. 3-е вид. — Addison-Wesley, 2004. — 208 с.
8. Object Management Group (OMG). Unified Modeling Language (UML) Specification, Version 2.5.1. — 2017. — 796 с.
9. Pilone, D., & Pitman, N. UML 2.0 in a Nutshell. — O'Reilly Media, 2005. — 214 с.
10. Czarnecki, K., & Eisenecker, U. W. Generative Programming: Methods, Tools, and Applications. — Addison-Wesley, 2000. — 832 с.
11. Stroustrup, B. The C++ Programming Language. 4-е вид. — Addison-Wesley, 2013. — 1368 с.
12. Chikofsky, E. J., & Cross, J. H. Reverse Engineering and Design Recovery: A Taxonomy // IEEE Software. — 1990. — Т. 7, №1. — С. 13–17.
13. Mens, T., & Tourwé, T. A Survey of Software Refactoring // IEEE Transactions on Software Engineering. — 2004. — Т. 30, №2. — С. 126–139.
14. Clang Team, LLVM Project. Clang: a C language family frontend for LLVM [Електронний ресурс]. — 2023. — Режим доступу: https://clang.llvm.org/
15. PlantUML Team. PlantUML: Open-Source Tool that Uses Simple Text Descriptions to Draw UML Diagrams [Електронний ресурс]. — 2023. — Режим доступу: <https://plantuml.com/>
16. Alhir, S. S. Learning UML 2.0. — O'Reilly Media, 2003. — 252 с.