НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет прикладної математики

Кафедра програмного забезпечення комп'ютерних систем

«На правах рукопису»
УДК 004.912

«До захисту допущено»
Завідувач кафедри
_____ Євгенія СУЛЕМА
«___»_____2022 р.

Магістерська дисертація

на здобуття ступеня магістра
за освітньо-науковою програмою
«Інженерія програмного забезпечення мультимедійних та інформаційно-пошукових систем»
зі спеціальності 121 Інженерія програмного забезпечення
на тему: «Алгоритмічно-програмний метод пошуку запозичень у програмному коді»

Виконав:
студент ІІ курсу, групи КП-01мн
Свинарчук Максим Владиславович _____

Керівник:
Доцент кафедри ПЗКС, к.т.н., доцент,
Заболотня Тетяна Миколаївна _____

Консультант з нормоконтролю:
Доцент кафедри ПЗКС, к.т.н., доцент,
Онай Микола Володимирович _____

Рецензент:
Доцент кафедри ММСА ІПСА, к.т.н., доцент,
Дідковська Марина Віталіївна _____

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

Київ – 2022 року

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет прикладної математики

Кафедра програмного забезпечення комп'ютерних систем

Рівень вищої освіти – другий (магістерський)

Спеціальність (спеціалізація) – 121 «Інженерія програмного забезпечення»

Освітньо-наукова програма «Інженерія програмного забезпечення мультимедійних та інформаційно-пошукових систем»

ЗАТВЕРДЖУЮ

Науковий керівник кафедри

_____ Іван ДИЧКА

«___»_____2020 р.

ЗАВДАННЯ
на магістерську дисертацію студенту

Свинарчуку Максиму Владиславовичу

1. Тема дисертації «Алгоритмічно-програмний метод пошуку запозичень у програмному коді», науковий керівник дисертації Заболотня Тетяна Миколаївна, к.т.н., доцент, затверджені наказом по університету від «05» травня 2022 р. № НС/201/2022.

2. Термін подання студентом дисертації «10» червня 2022 р.

3. Об'єкт дослідження: процес автоматизованого пошуку запозичень у текстових даних.

4. Предмет дослідження: методи, алгоритми та програмні засоби автоматизованого пошуку запозичень у програмному коді.

5. Перелік завдань, які потрібно розробити:
- провести збір даних для тестування та порівняння методів пошуку запозичень у програмному коді;
- провести аналіз існуючих методів пошуку запозичень у програмному коді та виявити їх недоліки;
- розробити комбінований метод для виявлення запозичень у програмному коді;
- розробити модель бази даних для зберігання та подальшого використання результатів пошуку запозичень програмного коду розробленим методом;
- виконати програмну реалізацію розробленого методу;
- виконати тестування розробленого програмного забезпечення;
- провести аналіз отриманих результатів.

6. Орієнтовний перелік графічного (ілюстративного) матеріалу:
- структура бази даних (плакат);
- схема роботи комбінованого методу пошуку запозичень у програмному коді (плакат);

- діаграма прецедентів програмного забезпечення для виявлення запозичень у програмному коді (плакат);
- структурна схема програмного забезпечення (плакат);
- схема модулів програмного забезпечення (плакат);
- схема використання багатопоточності для оптимізації швидкодії роботи комбінованого методу (плакат).

7. Орієнтовний перелік публікацій:
- Тези доповіді “Комбінований підхід для пошуку запозичень у програмному коді”.

8. Консультанти розділів дисертації

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Онай М.В., доцент кафедри ПЗКС		

9. Дата видачі завдання «04» жовтня 2020 р.

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1.	Ознайомлення з предметною галуззю	17.10.2020	
2.	Визначення структури магістерської дисертації; вивчення літератури, пошук додаткової літератури, патентний пошук	04.12.2020	
3.	Робота над першим розділом магістерської дисертації; проведення наукового дослідження	15.02.2021	
4.	Проведення наукового дослідження; робота над другим розділом магістерської дисертації; розроблення програмного забезпечення;	05.04.2021	
5.	Проведення наукового дослідження; робота над статтею за результатами наукового дослідження; підготовка матеріалів доповіді на конференції ПМК-2021	05.11.2021	
6.	Проведення наукового дослідження; робота над третім розділом магістерської дисертації	20.02.2022	
7.	Завершення роботи над основною частиною магістерської дисертації; підготовка ілюстративного матеріалу	16.05.2022	
8.	Оформлення текстової та графічної частини магістерської дисертації	08.06.2022	

Студент Максим СВИНАРЧУК

Науковий керівник Тетяна ЗАБОЛОТНЯ

РЕФЕРАТ

Актуальність теми. Існуючі системи для пошуку запозичень у програмному коді не здатні гарантувати виявлення плагіату, а можуть лише вказати загальний відсоток подібного коду та виокремити його для перевірки користувачем. Реалізовані методи для аналізу подібності програмного коду здебільшого намагаються виділити одну конкретну характеристику коду (кількість атрибутів, структурне представлення програми, представлення у вигляді графу тощо) та використовувати її для аналізу. На практиці це дає непогані результати, але налаштованість лише на одну характеристику коду дозволяє знайти спосіб приховування плагіату для кожного методу окремо. Це створює проблему, коли використання одного підходу для аналізу може давати як добрі, так і незадовільні результати.

Розроблення більш ефективних методів для пошуку запозичень у програмному коді є актуальною задачею та має практичне застосування.

Об’єктом дослідження є процес автоматизованого пошуку запозичень у текстових даних.

Предметом дослідження є методи, алгоритми та програмні засоби автоматизованого пошуку запозичень у програмному коді.

Мета роботи: підвищення ефективності виявлення запозичень у програмному коді за критерієм повноти пошуку шляхом розроблення алгоритмічно-програмного методу, який поєднує переваги різних підходів до аналізу запозичень у програмному коді.

Методи дослідження. В роботі використовуються методи оброблення текстових даних, методи порівнянь текстів, теорія програмних систем, теорія програмування, теорія алгоритмів.

Наукова новизна. Запропоновано алгоритмічно-програмний метод пошуку запозичень у програмному коді, заснований на комбінації

текстового, атрибутного та структурного аналізів програмного коду, який відрізняється від існуючих методів проведенням багатоетапного та комплексного аналізу запозичень та забезпеченням можливості роботи з базою програмних кодів на кожному з етапів методу, що дозволяє підвищити ефективність виявлення запозичень за критерієм повноти пошуку.

Практична значущість. Запропонований алгоритмічно-програмний метод пошуку запозичень у програмному коді дозволив підвищити ефективність виявлення таких запозичень за критерієм повноти пошуку та розробити відповідні програмні засоби, придатні до використання в навчальній і комерційній сферах з метою перевірки програмних кодів на наявність плагіату.

Визначений у дослідженні формат подання даних програмної структури для кожного з етапів аналізу комбінованого методу може бути врахований при розробленні програмних засобів для реалізації ефективного за критерієм часу пошуку запозичень в базі програмних кодів.

Апробація роботи. Основні положення і результати роботи були представлені та обговорювалися на XIV науковій конференції магістрантів та аспірантів «Прикладна математика та комп'ютинг» ПМК-2021 (17-19 листопада 2021 р., м. Київ).

Структура та обсяг роботи. Магістерська дисертація складається з вступу, чотирьох розділів, висновків та додатків.

У вступі надано загальну характеристику роботи, зроблено оцінку стану проблеми, обґрунтовано актуальність напрямку досліджень.

У першому розділі розглянуто основні чотири типи запозичень програмного коду з різними методиками для приховування факту запозичення. Наведено загальну класифікацію алгоритмів за методами, які лежать в їх основі та розглянуто найпопулярніші алгоритми, які використовується для виявлення запозичень у програмному коді.

У другому розділі запропоновано комбінований метод пошуку запозичень у програмному коді, який складається з шести етапів: нормалізація, токенізація, структурний аналіз, аналіз коментарів, аналіз атрибутів коду та формування результатів. На етапах аналізу структури коду та коментарів використано алгоритм просіювання з алгоритмом відбитків документа. Для визначення схожості між порівнюваними програмами на етапі аналізу атрибутів описано алгоритм обрахунку на основі оцінки кореляції. Окремо розглянуто формат представлення програмної структури, який використовується на кожному з етапів аналізу.

У третьому розділі проведено аналіз обраних технологій розроблення. Описано архітектуру розробленого програмного забезпечення, структуру бази даних та особливості реалізації, зокрема використання можливостей багатопотоковості для оптимізації швидкодії пошуку запозичень.

У четвертому розділі наведено приклад використання реалізованого програмного забезпечення. Проведено аналіз ефективності запропонованого методу за критеріями повноти пошуку та часу пошуку запозичень у програмному коді. Порівняння з існуючими системами демонструє, що запропонований метод дозволяє отримати більшу точність повноти пошуку, але вимагає більше часу для аналізу. Також описано напрямки для подальшої роботи в рамках даного дослідження.

У висновках проаналізовано отримані результати дослідження.

У додатках наведено графічні матеріали до пояснювальної записки, скріншоти презентації та лістинг коду розробленого програмного забезпечення.

Робота виконана на 86 аркушах, містить 3 додатки та посилання на список використаних літературних джерел з 26 найменувань. У роботі наведено 34 рисунки, 8 таблиць, 12 лістингів.

Ключові слова: плагіат, пошук запозичень, нормалізація, токенізація, атрибутні характеристики коду, структурний аналіз, мітки документа.

ABSTRACT

Theme urgency. Existing borrowing systems in source code are not able to guarantee the detection of plagiarism, they can only specify the total percentage of such code and select it for user verification. Implemented methods for analyzing the similarity of program code basically try to identify one specific characteristic of the code (number of attributes, structural representation of the program, representation in the form of a graph, etc.) and use it for analysis. In practice, this gives good results, but using only one characteristic of the code allows finding a way to hide plagiarism for each of the methods separately. This creates a problem when using one approach for analysis can give both good and unsatisfactory results.

Developing more effective methods for finding borrowings in source code is relevant and has practical usage.

Object of research is the process of automated search for borrowings in text data.

Subject of research is the methods, algorithms, and software for the automated search for borrowings in the source code.

Research objective is to improve the efficiency of detecting borrowings in the source code by the criterion of completeness of the search by developing an algorithmic-software method that combines the advantages of different approaches to the analysis of borrowings in the program code.

Research methods. The thesis uses the methods of text data processing, methods of text comparisons, theory of software systems, theory of programming, theory of algorithms.

Scientific novelty is that an algorithmic-program method of finding borrowings in source code is proposed, based on a combination of textual, attribute and structural analysis of source code, which differs from existing methods by conducting multi-stage and complex analysis of borrowings and

providing the ability to work with a source code database, which allows increasing the efficiency of borrowing detection by the criterion of completeness of the search.

Practical value. The proposed algorithmic-software method of searching for borrowings in the source code allowed to increase the efficiency of searching for such borrowings by the criterion of completeness and to develop appropriate software suitable for use in educational and commercial spheres to check source codes for plagiarism.

The format of data presentation of the program structure determined in the research for each of the stages of the analysis of the combined method can be considered when developing software for the implementation of time-efficient borrowing in the database of program codes.

Approbation. The basic points and outcomes of the research have been presented and discussed at the 14th scientific conference for students and postgraduates «Applied mathematics and computing» PMK-2021 (November 17-19, 2021, Kyiv).

Structure and content of the thesis. The master thesis consists of the introduction, four chapters, conclusions, and appendixes.

The introduction provides a general description of the work, assesses the current state of the problem, substantiates the relevance of research.

The first section discusses the four main types of source code borrowing with different techniques for concealing the fact of borrowing. The general classification of algorithms according to the methods underlying them is given and the most popular algorithms used to detect borrowings in the source code are considered.

The second section proposes a combined method of finding borrowings in source code, which consists of six stages: normalization, tokenization, structural analysis, comment analysis, analysis of code attributes and the formation of results. At the stages of analysis of the code structure and comments, a winnowing

algorithm and a document fingerprint algorithm were used. To determine the similarity between the compared programs at the stage of attribute analysis, a calculation algorithm based on correlation estimation was described. The format of the program structure presentation used at each stage of the analysis is considered separately.

The third section analyses the selected development technologies. The architecture of the developed software, the structure of the database and the features of implementation are described, for example, the use of multithreading to optimize the speed of borrowing search.

The fourth section gives an example of using the implemented software. The efficiency of the proposed method was analyzed according to the criteria of completeness of search and time of search of borrowings in the source code. Comparison with existing systems shows that the proposed method allows to obtain greater accuracy of search completeness but requires more time for analysis. Directions for further work in this study were also described.

The results of the work are analyzed in the conclusions.

The appendices provide graphic materials for the explanatory note, screenshots of the presentation and code listing of the developed software.

The work is performed on 86 sheets, contains 3 appendices and links to the list of used literature sources from 26 titles. The paper presents 34 figures, 8 tables, 12 listings.

Keywords: plagiarism, borrowing search, normalization, tokenization, code attribute characteristics, structural analysis, document fingerprints.

ЗМІСТ

СПИСОК ТЕРМІНІВ, СКОРОЧЕНЬ ТА ПОЗНАЧЕНЬ

Лексема – token – послідовність машинних символів вихідного коду програми, що мають певне сукупне значення.

Метасимвол (в регулярному виразі) – спеціальний символ (“.”, “+”, “?”, “*”, “{“, “}”, “|”, “(” та “)”) або комбінація символів (“\d”, “\w”, “\B”, тощо), які відповідають певній множині інших символів, або мають логіку співставлення підрядків та дозволяють створювати складні пошукові запити в регулярних виразах.

Оператор – спеціальний символ в мовах програмування, аналогічний запису математичним операціям, який повідомляє транслятору про операцію з операндами. Окрім арифметичних операцій, оператори можуть виконувати операції з логічними значеннями, рядками та перевірку рівності.

Операнд – аргумент операції та дані, що обробляються оператором чи командою. Іноді операндом називають місце, позицію в тексті, де має стояти аргумент операції.

Оператор керування – це оператор, який визначає порядок та спосіб виконання інших операторів; оператори керування визначають потік (логіку) виконання програми та діляться на декілька типів: логічний оператор (if), оператор циклу (do, do-while, for) та оператор switch.

CLI – Command Line Interface – тип інтерфейсу для взаємодії користувача і комп’ютера, де команди подаються через введення текстових інструкцій.

GUI – Graphical User Interface – тип графічного інтерфейсу для взаємодії користувача з електронними пристроями, що заснований на поданні доступних команд (функцій та системних об'єктів) у вигляді графічних елементів екрана (піктограм, вікон, меню, кнопок, списків, тощо).

ANTLR – ANother Tool for Language Recognition – потужний генератор синтаксичних аналізаторів для читання, оброблення, виконання та

перекладу структурованого тексту або двійкових файлів, який широко використовується для створення мов, інструментів і фреймворків.

HTML – HyperText Markup Language – мова розмітки гіпертекстових документів, базовий інструмент для створення всіх веб-сторінок, визначає зміст та структуру веб-контенту.

JVM – Java Virtual Machine – віртуальна машина Java, основна частина виконуючої системи Java, яка виконує байт-код, попередньо створений із програмного коду компілятором. JVM також може використовуватися для виконання програм, написаних іншими мовами програмування, якщо вони були скомпільовані у байт-код Java.

JAR – Java Archive – це архів, створений мовою програмування Java. Може бути окремим виконуваним файлом, тобто представляти собою програму, для запуску якої необхідна встановлена JVM.

ORM – Object-Relational Mapping – технологія програмування, яка пов'язує бази даних із концепціями об'єктно-орієнтованих мов програмування, створюючи «віртуальну об'єктну базу даних».

JPA – Java Persistence API – це специфікація Java SE і Java EE, стандартизований інтерфейс для Java ORM фреймворків, що описує спосіб управління та збереження Java-об'єктів в таблицях реляційних баз даних.

Кластеризація баз даних – розбиття бази даних на кілька спеціалізованих об’єктів (кластерів), які використовується для спільного зберігання декількох таблиць, що часто використовуються разом у SQL запитах. Це дозволяє скоротити кількість операцій вводу-виводу та покращити час доступу до таблиць в рамках одного кластера.

Реплікація баз даних – одна з технік масштабування бази даних, яка полягає в тому, що дані з одного сервера бази даних постійно копіюються (реплікуються) на один або кілька інших серверів (реплік).

SSL – Secure Sockets Layer – рівень захищених сокетів, який забезпечує безпечний канал зв'язку між двома машинами або пристроями, що працюють через інтернет чи внутрішню мережу.

ER – Entity-relationship – схема, яка показує відношення сутностей, що зберігаються в базі даних.

API – Application Programming Interface – прикладний програмний інтерфейс, який описує способів взаємодії з програмним забезпеченням.

DAO – Data Access Object – програмний інтерфейс для взаємодії з базою даних або будь-якого іншого механізму зберігання, що приховує деталі реалізації.

DTO – Data Transfer Object – шаблон проєктування, який використовується для передачі даних між додатками. DTO на відміну від бізнес об’єктів (моделей) або DAO не повинен містити жодної логіки поведінки.

CRUD – Create, Read, Update, Delete – базові функції управління даними.

СУБД – система управління базами даних.

ВСТУП

У сучасному технологічному світі права інтелектуальної власності мають не меншу цінність, ніж будь-які інші об’єкти приватної власності, а інколи, навіть, є більш важливими. Висока популярність та поширеність мережі Інтернет дозволили людству створити необмежену бібліотеку знань. А легка доступність всіх можливих типів інформації є потужним акселератором в розвитку будь-чого. Однак, разом з доступністю до авторських джерел почастішали випадки запозичення чужих робіт та присвоєння їх результатів. Таке явище називається плагіатом.

Плагіат часто можна спостерігати у навчальних роботах, у комерційній і навіть науково-дослідницькій сферах. Будь-який прояв запозичень чи намагання привласнити чужий результат інтелектуальної роботи завжди несе шкоду як власнику, так і «крадію». Саме тому з’явилася потреба у захисті авторських прав інтелектуальної власності. Але обсяги і кількість різних типів робіт з кожним роком збільшуються в геометричній прогресії, відповідно встежити за всіма порушеннями прав власності стає складніше, а інколи взагалі неможливо. Тому захист авторських прав, інспектування та перевірки авторства стає дедалі важчим процесом.

Звідси можна зробити висновок, що необхідні різного роду системи для порівняння робіт та виявлення плагіату. Крім того, таки системи мають бути максимально автоматизованими, адже однотипних текстів буває дуже багато, шукати та перевіряти вручну всі роботи нераціонально. Головна задача, яка стоїть перед такого роду системами – як шукати плагіат?

Існує безліч вже реалізованих і діючих програм та програмних комплексів для пошуку плагіату. Але у всіх подібних системах є один важливий недолік: вони працюють тільки з текстами, написаними природною мовою. Звідси виникає проблема з важливою та найбільш активною в тенденціях розвитку сферою в сучасному світі – IT-технологіями. Головна особливість даної сфери в тому, що інтелектуальна

власність виражається в програмному коді – тексті, написаному спеціальними мовами програмування для опису інструкцій роботи електронно-обчислювальних машин. І тут всі класичні детектори плагіату безсилі. У програмний текст входять імена змінних, функцій, методів, класів, структур, але все це ніяк не впливає на сутність програми. Їх можна перейменувати як завгодно, переставляти місцями, писати одну і ту ж інструкцію декількома різними способами, але програма буде працювати як і раніше. Тобто код може бути змінений до невпізнанності зі збереженням результату роботи.

Отже, розроблення спеціалізованих методів для пошуку запозичень у програмному коді є актуальною задачею. Існуючі рішення не здатні гарантувати виявлення наявності або відсутності запозичень, можуть лише вказати загальний відсоток подібного коду та виокремити його для перевірки користувачем. Реалізовані методи для аналізу подібності програмного коду у своїй основі намагаються виділити одну конкретну характеристику коду (кількість атрибутів, структурне представлення програми, представлення у вигляді графу тощо) та використовувати її для аналізу. На практиці це дає непогані результати, але налаштованість лише на одну характеристику коду дозволяє знайти спосіб приховування плагіату для кожного з методів окремо. Це створює проблему, коли використання одного підходу для аналізу може давати як високі, так і незадовільні результати.

Дослідження в рамках даної магістерської дисертації зосереджено на розробленні комбінованого алгоритмічно-програмного методу пошуку запозичень у програмному коді, який аналізує різні частини програмного тексту та є більш ефективним за критерієм повноти пошуку.

1. АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ ТА ІСНУЮЧИХ РІШЕНЬ

1.1. Типи запозичень програмного коду

Запозичення коду – це використання вже розробленої та ліцензованої програмної структури або її реалізації будь-якою мовою програмування без відповідного дозволу з боку автора або власника, якщо він таким не вважається.

Можна виділити декілька типів запозичень програмного коду [1].

1.1.1. Перший тип запозичення програмного коду

Перший тип запозичення програмного коду – це цілком скопійований фрагмент коду, що, фактично, і є повною копією оригіналу. Проте можуть бути деякі варіації у пробільних символах (різна кількість пробілів, перенесення рядків, символи табуляції тощо), коментарях та/або розмітці. Програми створенні за першим типом запозичення можна вважати точними клонами. Розглянемо наступний фрагмент коду (лістинг 1.1):

Лістинг 1.1. Оригінальний фрагмент коду

```
if (i <= j) {
   x = z + j; // Коментар 1
} else {
   x = z - i; // Коментар 2
}
```

Та порівняємо його з точною копією (лістинг 1.2).

Лістинг 1.2. Точна копія фрагмента коду

```
if (i<=j){
 // Розширений Коментар 1
 x=z+j;
}else{
 // Розширений Коментар 2
 x=z-i;
}
```

Легко помітити, що ці два фрагменти є схожими після видалення пробілів та коментарів, якщо порівнювати кожний рядок окремо. Але, навіть після видалення пробільних символі та коментарів, наступний фрагмент коду (лістинг 1.3) відрізняється від попередніх при порівняння рядків починаючи з символу "{" і закінчуючи "}". При цьому цей фрагмент також вважається запозиченням першого типу і є точною копією двох інших.

Лістинг 1.3. Копія коду з перенесенням символів на нові рядки

```
if ( i <= j )
{ // Новий Коментар 1
   x=z+j;
}
 else
{ // Новий Коментар 2
   x=z-i;
}
```

Типові методи пошуку запозичень на основі попарного порівняння рядків не зможуть виявити такі ділянки ідентичного коду.

1.1.2. Другий тип запозичення програмного коду

Другий тип запозичення програмного коду (лістинг 1.4). – це структурно/синтаксично ідентичні фрагменти коду, в яких присутні відмінності в іменах ідентифікаторів визначених користувачем (імен змінних, констант, класів, методів тощо), типах, розмітці, коментарях. Зарезервовані слова та синтаксична структура залишається такою ж, як у оригіналі. Також допускаються всі зміни з першого типу.

Лістинг 1.4. Запозичення програмного коду другого типу

```
if (i <= j)
   x = z + j; // Коментар 1
   x = z + 3;
else
   x = z - i; // Коментар 2
```

```
if (t <= k)
{              // Змінений Коментар 1
   a = b + k;
   a = b + 7; // Новий Коментар 3
} else
{
   a = b - t; // Змінений Коментар 2
}
```

1.1.3. Третій тип запозичення програмного коду

Для даного типу запозичення характерно, щоб певний фрагмент програмного коду отриманий за допомогою копіювання іншого фрагмента коду, але з видаленням і/або додаванням операторів та операндів (лістинг 1.5). При цьому зберігаються всі зміни властиві другому типу запозичення.

Лістинг 1.5. Запозичення програмного коду третього типу

```
if (i <= j)
   x = z + j; // Коментар 1
   x = z + 3;
else
   x = z - i; // Коментар 2
```

```
if (t <= k)
{              // Змінений Коментар 1
   a = b + k;
   r = -4;    // Новий Вираз
   a = b + r + 7; // Новий Коментар 3
} else
{
   a = b - t; // Змінений Коментар 2
}
```

1.1.4. Четвертий тип запозичення програмного коду

Цей тип запозичення є результатом семантичної подібності між двома чи більше фрагментами коду. У цьому типі фрагменти не обов'язково скопійовані з оригіналу. Два фрагменти коду можуть бути розроблені двома різними програмістами, синтаксично реалізовані по-різному, але реалізовувати однакову логіку та аналогічні за своєю функціональністю алгоритми. В ширшому сенсі, четвертий тип описує запозичення самої ідеї алгоритму. Так, наприклад, розрахунок факторіалу числа можна реалізувати двома шляхами (лістинг 1.6). З семантичної точки зору це різні ділянки коду, але вони однакові за своєю функціональністю та відносяться до четвертого типу запозичення.

Лістинг 1.6. Запозичення програмного коду четвертого типу

```
int res, i=1;
for (res=1; i<=VALUE; i++){
   res=res*i;
}
```

```
int factorial(int n) {
   if (n == 0)
      return 1;
   else
      return n * factorial(n-1);
}
```

Проаналізувавши описані вище типи запозичення програмного коду та характерні їм модифікації, можна стверджувати, що перший і другий типи належать до незначних змін. Третій тип знайти складніше, до нього можна віднести фрагменти програмного коду, які спеціально намагалися замаскувати, вносячи додатковий «шум». Четвертий тип більше властивий комерційному коду в силу складності своєї реалізації. Зазвичай, для того щоб реалізувати ділянку коду інакше з точки зору синтаксису в ньому необхідно добре розібратися. До четвертого типу також можна віднести переписування коду з однієї мови програмування в іншу. Тому цей тип запозичень коду необов'язковий для виявлення.

1.2. Класифікація методів для пошуку запозичень у програмному коді

Існує велика кількість способів представлення та подальшого аналізу програмного коду. В основному всі вони складаються з фази перетворення та фази порівняння. На першому етапі вихідний текст перетворюється на внутрішній формат, який дозволяє використовувати ефективніший алгоритм порівняння. У наступній фазі відбувається безпосередній пошук запозичень та виведення деякої кількісної міри, що характеризує їх кількість. Тому доцільно класифікувати методи виявлення плагіату програмного коду відповідно до їх внутрішнього формату представлення програмного тексту [1].

1.2.1. Пошук запозичень методом текстового порівняння

Використовується для виявлення звичайного текстового плагіату (рис. 1.1). Програмний код жодним чином не модифікується і розглядається «як є». Метод не враховує структуру програми та її синтаксичні особливості, тому перейменування функцій та змінних чи несуттєві зміни у коді можуть стати серйозною перешкодою для ефективної роботи цього методу. Тексти програм порівнюються дослівно. Очевидно, що такий підхід досить неефективний для пошуку запозичень програмних

структур. Модифікації, характерні запозиченням першого та другого типів забезпечать унікальність програмного коду при перевірці в системах, які використовують метод текстового порівняння.

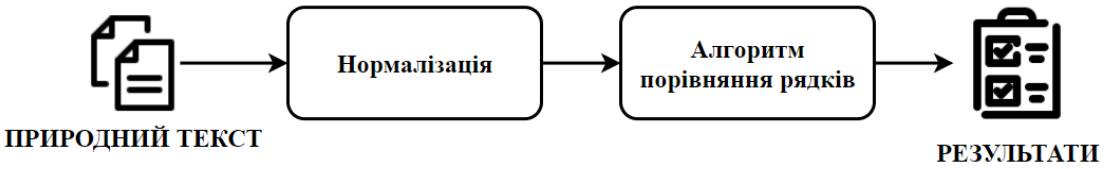


Рис. 1.1. Схема роботи методів на основі текстового порівняння

До плюсів методу текстового порівняння можна віднести досить просту реалізацію та можливість пошуку запозичень не лише в програмному коді, а й у звичайному тексті (наприклад, в коментарях програми). Крім того, перед аналізом можна провести деяку нормалізацію тексту та видалити несуттєві символи та/або слова. Також слід зазначити, що даний метод не залежить від мови програмування, а отже добре підходить для початкового аналізу запозичень.

1.2.2. Пошук запозичень на основі атрибутів коду

Один із перших методів пошуку запозичень програмного коду (рис. 1.2). Програмний код також не зазнає жодних змін. Але на відміну від виявлення запозичень методом текстового порівняння просте перейменування функцій та/або змінних, а також незначні маніпуляції з вихідним кодом не вплинуть на роботу цього методу.

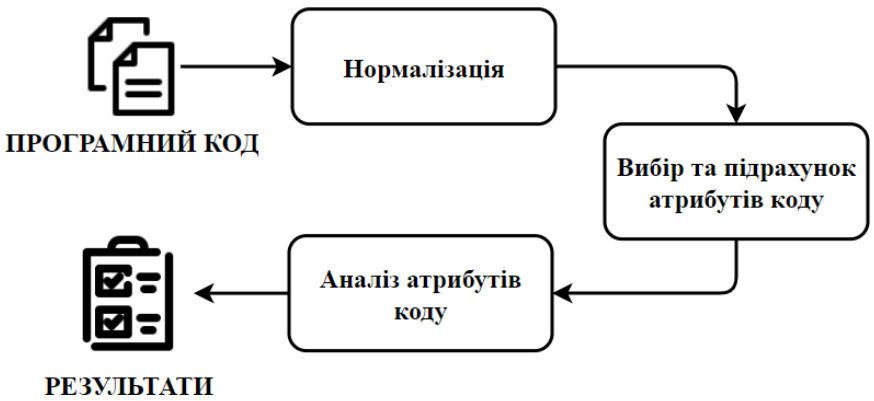


Рис. 1.2. Схема роботи методів на основі атрибутів коду

Як відправна точка для виявлення запозичень вибираються деякі кількісні характеристики програми – атрибути. Атрибутами можуть бути конкретні ознаки програми, такі як середня довжина рядків, кількість символів у програмі, кількісні характеристики операторів та операндів, змінних, команд присвоєння, ключових слів, цикломатична складність, тощо. Атрибути також можуть бути побудовані з декількох ознак, тоді програма буде представлена у вигляді послідовності чисел або у вигляді точки n-мірного простору, що підвищить якість порівняння програмних кодів. Дві програми можуть вважатися схожими, якщо відповідні числа атрибутів їх наборів близькі або збігаються. В результаті схожість програмних кодів може бути встановлено за допомогою нескладних порівнянь чисел (кількісних ознак коду). А оскільки в значній кількості мов програмування використовуються однакові ключові слова для задання циклів/умов, то цей метод також досить універсальний і майже не залежить від мови програмування.

Метод виявлення запозичень на основі атрибутів коду має серйозний недолік: якщо атрибути не пов'язані один з одним, результат може містити багато хибної інформації. Також недоцільно використовувати даний метод на невеликих програмах, оскільки кількість основних ключових конструкцій у них відрізняється незначно. Запозичення третього типу дозволить зменшити шанс виявити копію (лістинг 1.7).

Лістинг 1.7. Приклад виявлення запозичень на основі атрибутів коду. Програма ліворуч: операторів – 2; операцій – 7; операндів – 11. Програма праворуч: операторів – 2; операцій – 8; операндів – 13.

```
if (i <= j)
   x = z + j; // Коментар 1
   x = z + 3;
else
   x = z - i; // Коментар 2
```

```
if (t <= k)
{ // Змінений Коментар 1
   r = y + k;
   q = 33; // Новий вираз
   r = y + 6; // Новий Коментар 3
} else
{
   r = y - t; // Змінений Коментар 2
}
```

1.2.3. Пошук запозичень на основі структури коду

Метод виявлення запозичень на основі структури коду використовує структуру програми. Відмінна риса даного підходу – це дослідження не окремо кожної ознаки програмного коду, а в загальному контексті, встановлення зв'язків різних його характеристик.

Щоб не брати до уваги зайву інформацію в програмному коді, структурний метод дозволяє привести його до компактного вигляду, в якому враховуються лише важливі деталі, що впливають на оригінальність програми. Можна розглядати програмний код таким, як він написаний, і, наприклад, порівнювати попарно рядки. Детектори плагіату, що працюють зі звичайним текстом та які реалізовані на основі методу текстового порівняння, так і надійдуть. Але це вкрай неефективно для виявлення запозичень в програмному коді. Одним із шляхів вирішення даної проблеми може бути представлення коду в параметризованому вигляді, а саме виконання процесу токенізації (лістинг 1.8). Це дозволить проігнорувати ті властивості, що легко модифікуються, такі як коментарі, порожні символи, назви ідентифікаторів, що ніяк не впливають на виконання програми.

Лістинг 1.8. Приклад токенізації програмного коду

```
01: #include <stdio.h>
02: int main(void){
03:    for (int i = 0; i < 10; i++) {
04:       printf("Hello World!");
05:    }
06:    return 0;
07: }
```

```
01:
02: I @ ( ){
03: F (I @ = @ @ < @ @ + + ) {
04: @ ( " )
05: }
06: r @
07: }
```

Загальний підхід, характерний для всіх реалізацій структурного методу, можна розбити на три етапи (рис. 1.3):

1. Нормалізація програмного коду для видалення коментарів, пробільних символів, тощо (детальніше див. підрозділ 2.2).
2. Токенізація програмного коду для представлення коду в параметризованому вигляді (детальніше див. підрозділ 2.3).

3. Використання алгоритму пошуку запозичень, який використовує вже підготовлений програмний код та аналізує його структуру.

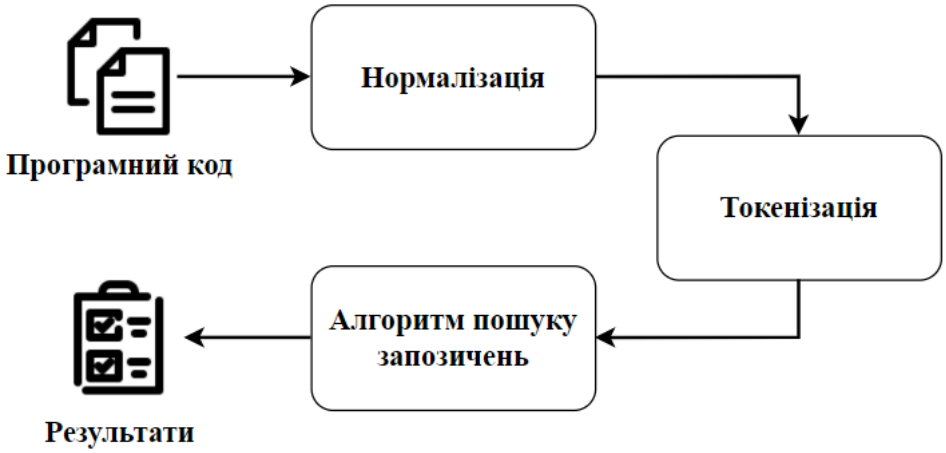


Рис. 1.3. Схема роботи методів на основі структури коду

Якщо порівняти метод на основі структури коду з методом на основі атрибутів коду, то перший має перевагу за якістю представлення програмного коду. Недоліком цього є складність у реалізації, і навіть трудомісткість роботи. Наочним прикладом можуть бути побудова дерев та графів програми та порівняння з подібними структурами в інших програмних кодах. Ще одним недоліком є сильна прив'язка структурного методу до мови програмування. Необхідно адаптувати створений метод для кожної конкретної мови окремо, що потрібує додаткових зусиль. Однак, не дивлячись на перераховані недоліки, структурний метод дає значний виграш у точності.

1.3. Аналіз існуючих алгоритмів для пошуку запозичень у програмному коді

Нижче наведено аналіз наступних алгоритмів:

- алгоритм локального вирівнювання рядків;
- алгоритм Левенштейна;
- алгоритм жадібного рядкового замощення;
- алгоритм з використанням Колмогорової складності;
- алгоритм відбитків документа.

1.3.1. Алгоритм локального вирівнювання рядків [2]

Нехай є два тексти програмного коду, представлених у вигляді рядків токенів (можливо, різної довжини). Вирівнювання двох токенізованих рядків відбувається за допомогою вставки в них пробілів таким чином, щоб їх довжини стали однаковими. Нехай A^* і B^* – нові рядки, отримані після вирівнювання вхідних рядків *a* і *b*, відповідно. Розглянемо A_i^* і B_i^*: вартість їх збігу – *s*; якщо тільки один із символів пропуск, то – *t;* вартість невідповідності – *d;* де *s* – додатне, *t* і *d* – невід'ємні.

Ціна вирівнювання – це сума індивідуальних вартостей всіх пар A_i^* та B_i^*, найбільше значення цієї цільової функції для всіх *i* та *j*, таких що $i \le j \le \left|A_i^*\right| = \left|B_i^*\right|$ на рядках $A_i^*\left[i..j\right]$ і $B_i^*\left[i..j\right]$ – *величина вирівнювання*.

Оптимальне вирівнювання між двома рядками – максимальне значення цільової функції серед усіх комбінацій вирівнювання. Це значення може бути обчислено за допомогою динамічного програмування.

Застосування алгоритму виглядає наступним чином (рис. 1.4):

1. Отримуються токенізовані рядки *A* і *B* двох програм.
2. *B* ділиться на секції, кожна з яких представляє модуль програми.
3. Для кожної секції і рядка *A* отримуються значення оптимального локального вирівнювання.
4. Результати комбінуються.

Рис. 1.4. Приклад локального вирівнювання

1.3.2. Алгоритм Левенштейна

Алгоритм Левенштейна [3] дозволяє обчислити найкоротшу відстань редагування (Levenshtein disstance – LD), яка відображає подібність між двома токенізованими рядками програмних кодів та вимірюється як кількість замін, вставок та видалення символів, необхідних для перетворення одного рядка на інший.

Лістинг 1.9. Псевдокод алгоритму Левенштейна

```
0   LevenshteinDistance(String A, String B) {
1     n = length(A);
2     m = length(B);
3     if (n = = 0)
4        return m;
5     else if (m = = 0)
6        return n;
7     d = [0..n, 0..m];
8     // i та j використовуються для індексування позиції у рядках A та B
9     i, j, cost;
10
11     For i = 0 … n
12        d[i, 0] = i;
13     For j = 0 … m
14        d[0, j] := j;
15
16     For i = 1 … n
17        For j = 1 … m
18           if A[i] = B[j]
19              cost = 0;
20           else
21              cost = 1;                        // заміна
22           d[i, j] = minimum(
23                              d[i-1,j] + 1,     // вилучення
24                              d[i ,j-1] + 1,    // вставка
25                              d[i-1,j-1] + cost  // заміна або однакові
26                           );
27     // значення LD в останній клітинці матриці
28     return d[n,m];
29  }
```

Реалізація алгоритму виглядає наступним чином (лістинг 1.9):

1. Встановлюється змінна *n* рівною довжині рядка *A*.
2. Встановлюється змінну *m* рівною довжині рядка *B*.
3. Якщо *n* = 0, результат – *m*.

   Якщо *m* = 0, результат – *n*.
4. Ініціалізується перший рядок від 0 до *n*.

   Ініціалізується перший стовпець від 0 до *m*.

5. У подвійному циклі за змінними n і m виконується порівняння.
6. Якщо $s[i]$ дорівнює $t[j]$, вартість операції – 0.

   Якщо $s[i]$ не дорівнює $t[j]$, вартість операції – 1.
7. Заповнюється комірка матриці (відстань Левенштейна) $d[i, j]$ значенням мінімальним з:
   a. Комірки зверху + 1: $d[i - 1, j] + 1$
   b. Комірки зліва + 1: $d[i, j - 1] + 1$
   c. Комірки зліва та зверху по діагоналі + вартість операції з п. 6: $d[i - 1, j - 1] + cost$.
8. Після всіх ітерацій кроків 5-7, відстань Левенштейна буде дорівнювати значенню в останній комірці: $d[n, m]$.

Розглянемо приклад роботи алгоритму Левенштейна для рядків «SCIENCE» та «SCIENTIFIC». Першим кроком розташовуємо ці слова по горизонталі та вертикалі до таблиці. При цьому неважливо, яке з них буде в рядку, а яке – в колонці. Також знадобиться один додатковий рядок та один стовпець. Пронумеруємо рядок від 0..n та стовпець від 0..m. Результат підрахунку відстані Левенштейна відповідно до описаного вище алгоритму наведено в табл. 1.1.

Таблиця 1.1

Приклад розрахунку відстані Левенштейна

		S	**C**	**I**	**E**	**N**	**C**	**E**
	0	1	2	3	4	5	6	7
S	1	0	1	2	3	4	5	6
C	2	1	0	1	2	3	4	5
I	3	2	1	0	1	2	3	4
E	4	3	2	1	0	1	2	3
N	5	4	3	2	1	0	1	2
T	6	5	4	3	2	1	1	2
I	7	6	5	4	3	2	2	2
F	8	7	6	5	4	3	3	3
I	9	8	7	6	5	4	4	4
C	10	9	8	7	6	5	4	**5**

Значення відстані Левенштейна використовується для обчислення функції подібності між вдома програмними кодами за наступною формулою:

$$sim(A,\ B\ ) = \left(1 - \frac{LD(A, B)}{max(A_{length}, B_{length})}\right) \cdot 100\% , \qquad (1.1)$$

де $LD(A, B)$ – відстань Левенштейна для програм A, B;

A_{length} – довжина токенізованого представлення програми A;

B_{length} – довжина токенізованого представлення програми B.

1.3.3. Алгоритм жадібного рядкового замощення

Розглянемо евристичний алгоритм отримання жадібного рядкового замощення [4]. На вхід подаються два рядки токенізованого представлення програмного коду, на виході отримуємо набір їхніх спільних непересічних підрядків, близький до оптимального.

Алгоритм використовує дві евристики:

1. Більш довгі послідовні збіги при порівнянні рядків токенів – це краще, ніж набір менших і непослідовних збігів.
2. Алгоритм ігнорує збіги, які менші порогового значення.

Друга евристика сприяє фільтрації шумів, тобто невеликих ділянок коду, які випадково збігаються в перевірених вихідних кодах.

Результатом застосування жадібного алгоритму для рядків токенів буде набір їх спільних підрядків, що не перетинаються. Підрядок, що входить у цей набір називається тайлом. Чим більша нормалізована сума довжин отриманих тайлів, тим більше схожість даних підрядків.

Функцію подібності можна представити у наступному вигляді:

$$sim(A, B) = \frac{2 \cdot |tiles|}{|A| + |B|} \cdot 100\% , \qquad (1.2)$$

де $|tiles|$ – кількість символів знайденого набору тайлів; $|A|$, $|B|$ – відповідні довжини рядків, що порівнюються.

Псевдокод алгоритму представлений на рис. 1.5.

У структурі алгоритму можна виділити 4 цикли:

1. Зовнішній цикл. Його результатом є додавання у набір чергового тайлу. Тайли додаються у порядку зменшення.
2. Внутрішній цикл по всім символам першого рядку.
3. Внутрішній цикл по всім символам другого рядку.
4. Цикл по співпадаючим символам, починаючи від першого співпадіння поточних символів даних рядків.

```
0   Greedy-String-Tiling(String A, String B) {
1       tiles = {};
2       do {
3           maxmatch = MinimumMatchLength;
4           matches = {};
5           Forall unmarked tokens A_a in A {
6               Forall unmarked tokens B_b in B {
7                   j = 0;
8                   while (A_{a+j} == B_{b+j} &&
9                           unmarked(A_{a+j}) && unmarked(B_{b+j}))
10                      j++;
11                  if (j == maxmatch)
12                      matches = matches ⊕ match(a, b, j);
13                  else if (j > maxmatch) {
14                      matches = {match(a, b, j)};
15                      maxmatch = j;
16                  }
17              }
18          }
19          Forall match(a, b, maxmatch) ∈ matches {
20              For j = 0 . . . (maxmatch − 1) {
21                  mark(A_{a+j});
22                  mark(B_{b+j});
23              }
24              tiles = tiles ∪ match(a, b, maxmatch);
25          }
26      } while (maxmatch > MinimumMatchLength);
27      return tiles;
28  }
```

Рис. 1.5. Псевдокод алгоритму жадібного рядкового замощення

Оптимізації алгоритму жадібного рядкового замощення

Використовується ідея алгоритму Рабіна-Карпа – порівняння підрядків за їх хеш-значеннями:

1. Хеш-значення обчислюються для всіх підрядків довжини *MinimumMatchLength* в *A* і *B*.
2. Кожне хеш-значення з рядка *A* потім порівнюється з кожним з хеш-значенням рядка *B*. Якщо вони однакові, то знайдено можливий збіг двох підрядків, що починаються з цього токена.
3. Використовується хеш-таблиця для розміщення підрядків з *B*, які мають те саме значення хешу, що й заданий підрядок з *A*. Це зменшує обчислення до лінійної кількості кроків.

1.3.4. Алгоритм з використанням Колмогорової складності

У алгоритмі використовується відстань між послідовностями, заснована на теорії інформації (an information based sequence distance) [1]:

$$d(x, y) = 1 - \frac{K(x) - K(x \mid y)}{K(xy)}\ , \qquad (1.3)$$

де $K(x)$ – Колмогорова складність послідовності x. Вона показує, скільки даних містить послідовність x. За визначенням, $K(x)$ – довжина найкоротшого програмного коду, який на порожній ввід друкує рядок x. Аналогічно, $K(x \mid y)$ – це довжина найкоротшого програмного коду, який на введення рядка y друкує рядок x. Варто відмітити, що якщо y не містить жодних даних для розуміння будови x, то $K(x \mid y) = K(x)$. Тоді вираз $K(x) - K(x \mid y)$ можна трактувати як скільки y «відомо» про x. Докладніше про Колмогорову складність можна переглянути в [5].

Основна відмінність наведеної метрики від інших, що використовуються в задачах знаходження запозичень програмного коду, полягає в її універсальності: дві програми, близькі до будь-якої іншої метрики, будуть близькими і щодо даної [6]. Теоретично детектор, заснований на такій метриці, неможливо обдурити.

Чим ближче функція відстані $d(x, y)$ до 0, тим більше загальних даних містять дві програми x та y. Але, як відомо, Колмогорова складність не обчислювана [9]. У статті [8] наводиться евристичне наближення $d(x, y)$, що базується на застосуванні алгоритму стиснення до токенізованого представлення програми:

$$d(x, y) \approx 1 - \frac{Comp(x) - Comp(x \mid y)}{Comp(xy)} , \qquad (1.4)$$

де $Comp(x)$ – довжина стисненого рядка, отриманого з вихідного за допомогою будь-якого алгоритму стиснення. У роботі [8] було розроблено спеціальний алгоритм стиснення (TokenCompress), який задовольняє специфічним вимогам цього алгоритму.

На основі роботи алгоритму рахується величина $d(x, y)$, яка потім використовується для визначення ймовірності, що одна з цих програм є копією іншої за наступною формулою:

$$sim(A, B) = (1 - d(A, B)) \cdot 100\% . \qquad (1.5)$$

1.3.5. Алгоритм відбитків документа

У алгоритмі відбитків [7] токенізований програмний код подається у вигляді послідовності міток так, щоб для схожих програм їх набори перетиналися. Послідовність дій даного алгоритму можна представити у наступному вигляді:

1. Послідовно хешуються підрядки довжиною k (фіксований параметр) токенізованої програми P.
2. Виділяється деяка підмножина їх хеш-значень (міток), що добре характеризує P. Ті ж кроки повторюються для програм $T_1, T_2 ... T_n$, їх обрані мітки зберігаються в таблицю (базу даних).
3. За допомогою таблиці отримується набір ділянок рядку P, підозрілих на співпадіння.

4. Отримані на попередньому кроці дані використовується для обчислення функції подібності за формулою (1.6).

$$sim(A, B) = \frac{|S(A) \cap S(B)|}{|S(A) \cup S(B)|} \cdot 100\% , \qquad (1.6)$$

де $S(A)$ – послідовність міток програми A;

$|A|$ – розмір множини A.

Змістовна частина алгоритму – це вибір значення k та другий крок. Число k обмежує довжину найменшого підрядка, з яким може працювати даний алгоритм. Невелике його значення сприяє ігноруванню «шумів», в іншому випадку алгоритм може пропустити випадок запозичення.

Для виділення множини міток серед хеш-значень токенізованої програми існує декілька способів**.** Більш детально кожен із них буде розглянуто в підрозділі 2.4.

1.4. Огляд існуючих систем для пошуку запозичень у програмному коді

Нижче представлена загальна таблиця деяких існуючих систем для виявлення запозичень програмного коду. Фактично, тільки чотири з наведених нижче працездатні на даний момент. Детально вони будуть описані далі у цьому пункті.

Таблиця 1.2

Системи для виявлення запозичень у програмному коді

Система	Автор	Модель представлення	Алгоритм
Accuse	Donaldson [8]	характеристика атрибутів	порівняння атрибутів (функція кореляційної оцінки)
Bandit	West [9]	токени	порівняння токенів - за допомогою Dotpot

Продовження табл. 1.2

JPlag	Tichy, Malhopl, Prechelt [4]	токени	алгоритм жадібного рядкового замощення з модифікацією алгоритмом Рабіна-Карпа
MOSS	Aiken [7]	мітки документа	алгоритм просіювання (реалізація алгоритму відбитків)
Plague	Whale [10]	токени	аналіз профіля структури, алгоритм Хескеля
SID	Chen, Francia, Lin, Mckinnon, Seker [1]	токени	метрика, заснована на Колмогоровій складності, EToken Compress
SIM	Grune [11]	токени	алгоритм локального вирівнювання рядків
YAP	Wise [12]	токени	порівняння токенів, алгоритм Хескеля
YAP3	Wise [13]	токени	алгоритм жадібного рядкового замощення

1.4.1. SIM

Для виявлення запозичень у програмному коді система SIM реалізує класичний метод на основі структури коду, а саме токенізує вихідний код, та порівнює токенізовані рядки за допомогою *алгоритму локального вирівнювання рядків* (рис. 1.6-1.7).

SIM підтримує Java, C, C++, Pascal, Modula-s, Miranda та текст на природній мові.

Головний недолік – неможливість відслідковувати переміщення блоків програмного коду.

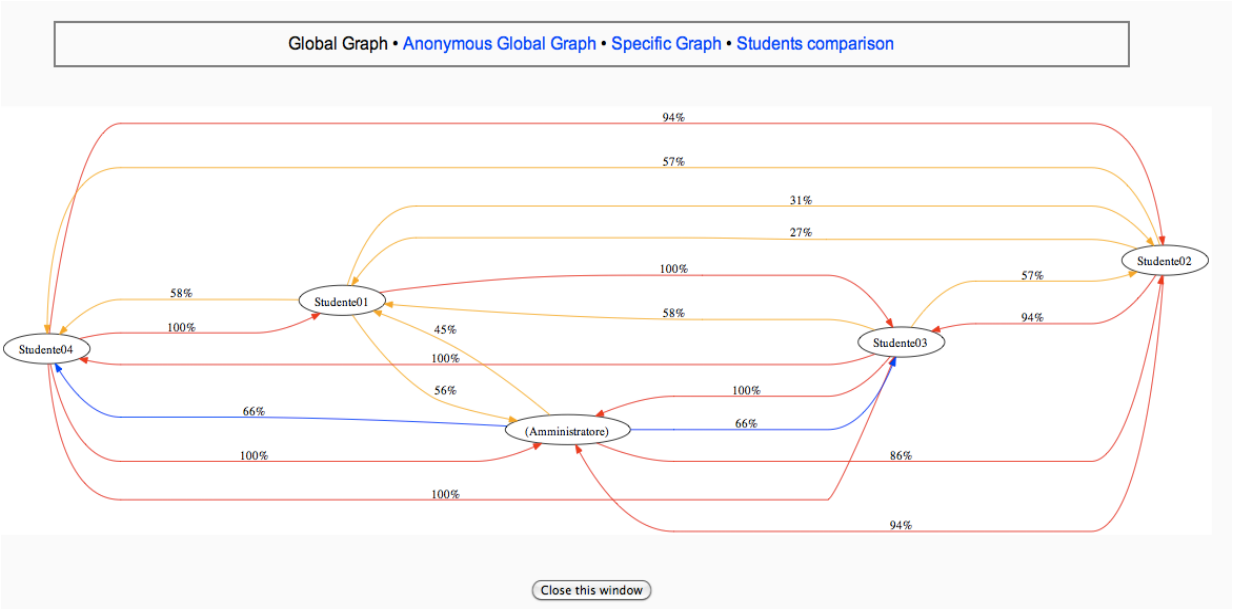


Рис. 1.6. Приклад графу зі статистикою запозичень у системі SIM

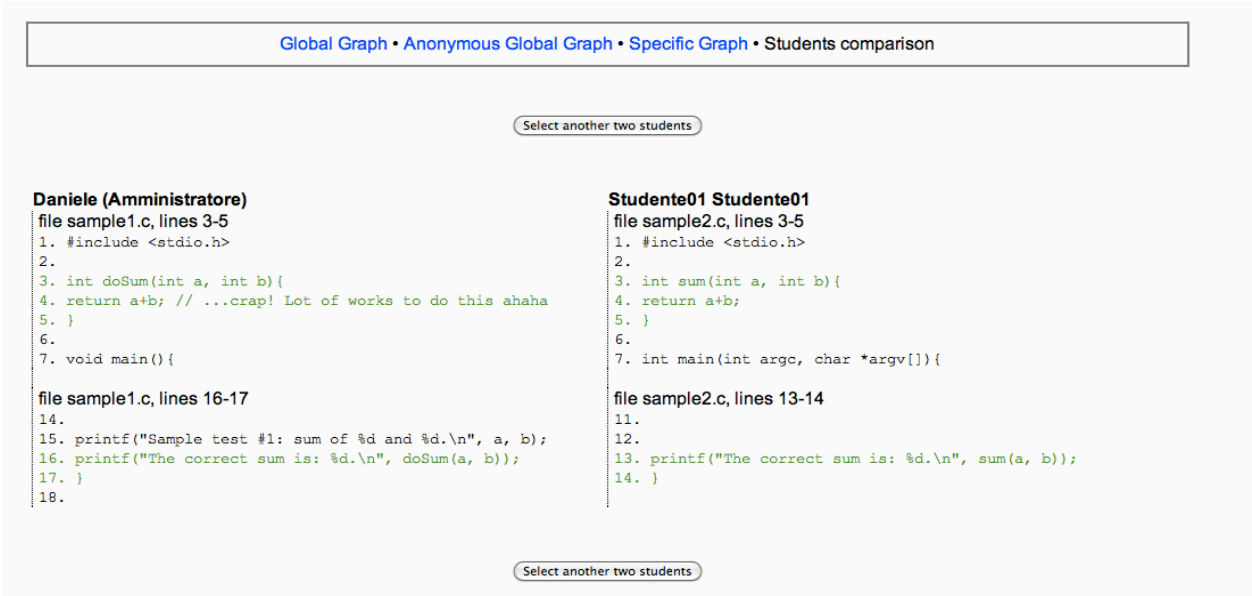


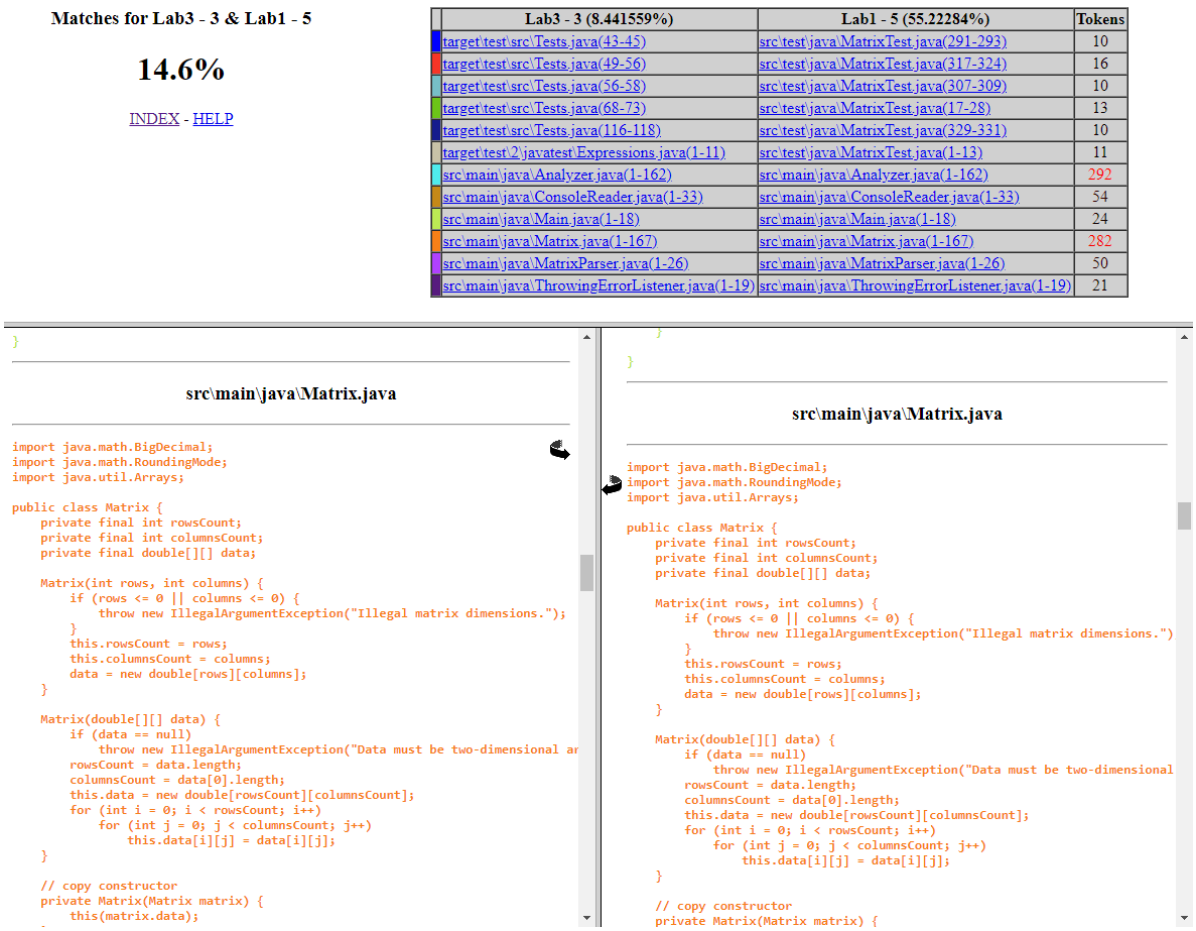
Рис. 1.7. Перегляд запозичень програмного коду у системі SIM

1.4.2. JPlag

JPlag – система для аналізу запозичення програмного коду між множиною наборів файлів (рис.1.8-1.9). Детектор призначався для перевірки студентських робіт, на них же і тестувався. Система підтримує Java, C#-1.2, C, C++, Python 3, Scheme та текст на природній мові.

JPlag працює в два етапи: спочатку перетворює програми в рядки токенів, які представляють структуру програми, потім попарно порівнює

рядки токенів за допомогою алгоритму *жадібного рядкового замощення*, з додатковими оптимізаціями.



Matches for Lab3 - 3 & Lab1 - 5

14.6%

INDEX - HELP

	Lab3 - 3 (8.441559%)	Lab1 - 5 (55.22284%)	Tokens
	target\test\src\Tests.java(43-45)	src\test\java\MatrixTest.java(291-293)	10
	target\test\src\Tests.java(49-56)	src\test\java\MatrixTest.java(317-324)	16
	target\test\src\Tests.java(56-58)	src\test\java\MatrixTest.java(307-309)	10
	target\test\src\Tests.java(68-73)	src\test\java\MatrixTest.java(17-28)	13
	target\test\src\Tests.java(116-118)	src\test\java\MatrixTest.java(329-331)	10
	target\test\2\javatest\Expressions.java(1-11)	src\test\java\MatrixTest.java(1-13)	11
	src\main\java\Analyzer.java(1-162)	src\main\java\Analyzer.java(1-162)	292
	src\main\java\ConsoleReader.java(1-33)	src\main\java\ConsoleReader.java(1-33)	54
	src\main\java\Main.java(1-18)	src\main\java\Main.java(1-18)	24
	src\main\java\Matrix.java(1-167)	src\main\java\Matrix.java(1-167)	282
	src\main\java\MatrixParser.java(1-26)	src\main\java\MatrixParser.java(1-26)	50
	src\main\java\ThrowingErrorListener.java(1-19)	src\main\java\ThrowingErrorListener.java(1-19)	21

src\main\java\Matrix.java

```
import java.math.BigDecimal;
import java.math.RoundingMode;
import java.util.Arrays;

public class Matrix {
    private final int rowsCount;
    private final int columnsCount;
    private final double[][] data;

    Matrix(int rows, int columns) {
        if (rows <= 0 || columns <= 0) {
            throw new IllegalArgumentException("Illegal matrix dimensions.");
        }
        this.rowsCount = rows;
        this.columnsCount = columns;
        data = new double[rows][columns];
    }

    Matrix(double[][] data) {
        if (data == null)
            throw new IllegalArgumentException("Data must be two-dimensional ar
        rowsCount = data.length;
        columnsCount = data[0].length;
        this.data = new double[rowsCount][columnsCount];
        for (int i = 0; i < rowsCount; i++)
            for (int j = 0; j < columnsCount; j++)
                this.data[i][j] = data[i][j];
    }

    // copy constructor
    private Matrix(Matrix matrix) {
        this(matrix.data);
```

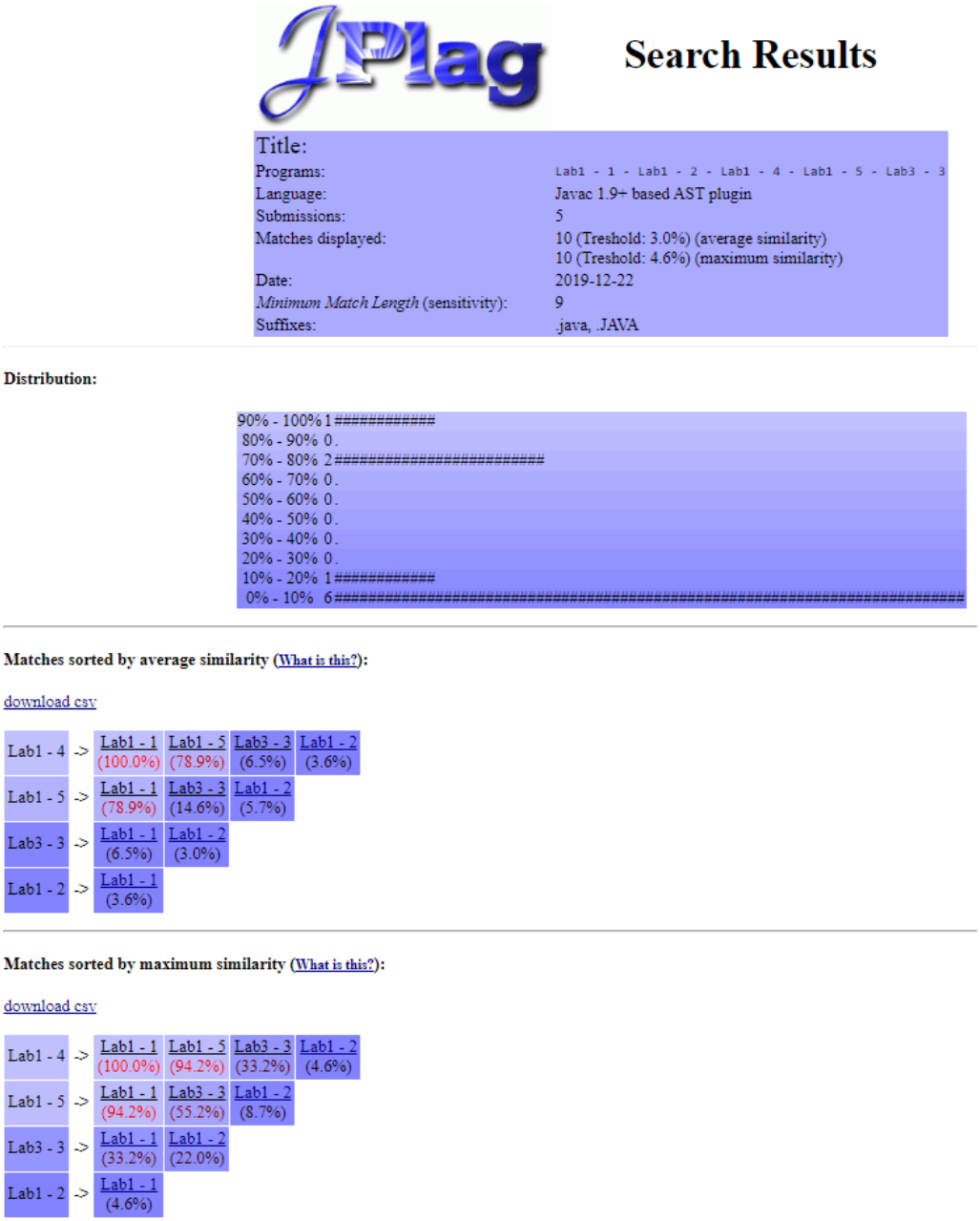
src\main\java\Matrix.java

```
import java.math.BigDecimal;
import java.math.RoundingMode;
import java.util.Arrays;

public class Matrix {
    private final int rowsCount;
    private final int columnsCount;
    private final double[][] data;

    Matrix(int rows, int columns) {
        if (rows <= 0 || columns <= 0) {
            throw new IllegalArgumentException("Illegal matrix dimensions.")
        }
        this.rowsCount = rows;
        this.columnsCount = columns;
        data = new double[rows][columns];
    }

    Matrix(double[][] data) {
        if (data == null)
            throw new IllegalArgumentException("Data must be two-dimensional
        rowsCount = data.length;
        columnsCount = data[0].length;
        this.data = new double[rowsCount][columnsCount];
        for (int i = 0; i < rowsCount; i++)
            for (int j = 0; j < columnsCount; j++)
                this.data[i][j] = data[i][j];
    }

    // copy constructor
    private Matrix(Matrix matrix) {
```

Рис. 1.8. Перегляд запозичень програмного коду у системі JPlag

Остання версія JPlag представляє собою бібліотеку, яка може бути використана за допомогою Command Line Interface (CLI) або програмним чином за допомогою Java API, що дозволяє легко інтегрувати детектор в інші проєкти Java. Але в будь-якому випадку для використання необхідно мати встановлену JVM.

Для великих наборів даних JPlag пропонує стратегію паралельного порівняння, що дозволяє пришвидшити час виконання, використовуючи всі доступні ядра центрального процесора. Для цього необхідно задати аргумент “-c parallel”.



JPlag

Search Results

Title:

Programs:	Lab1 - 1 - Lab1 - 2 - Lab1 - 4 - Lab1 - 5 - Lab3 - 3
Language:	Javac 1.9+ based AST plugin
Submissions:	5
Matches displayed:	10 (Treshold: 3.0%) (average similarity) 10 (Treshold: 4.6%) (maximum similarity)
Date:	2019-12-22
Minimum Match Length (sensitivity):	9
Suffixes:	.java, .JAVA

Distribution:

90% - 100% 1 ##########
80% - 90% 0.
70% - 80% 2 ########################
60% - 70% 0.
50% - 60% 0.
40% - 50% 0.
30% - 40% 0.
20% - 30% 0.
10% - 20% 1 ##########
0% - 10% 6 ##

Matches sorted by average similarity (What is this?):

download csv

Lab1 - 4	->	Lab1 - 1 (100.0%)	Lab1 - 5 (78.9%)	Lab3 - 3 (6.5%)	Lab1 - 2 (3.6%)
Lab1 - 5	->	Lab1 - 1 (78.9%)	Lab3 - 3 (14.6%)	Lab1 - 2 (5.7%)	
Lab3 - 3	->	Lab1 - 1 (6.5%)	Lab1 - 2 (3.0%)		
Lab1 - 2	->	Lab1 - 1 (3.6%)			

Matches sorted by maximum similarity (What is this?):

download csv

Lab1 - 4	->	Lab1 - 1 (100.0%)	Lab1 - 5 (94.2%)	Lab3 - 3 (33.2%)	Lab1 - 2 (4.6%)
Lab1 - 5	->	Lab1 - 1 (94.2%)	Lab3 - 3 (55.2%)	Lab1 - 2 (8.7%)	
Lab3 - 3	->	Lab1 - 1 (33.2%)	Lab1 - 2 (22.0%)		
Lab1 - 2	->	Lab1 - 1 (4.6%)			

Рис. 1.9. Загальний результат аналізу системи JPlag

1.4.3. SID

SID (Software Integrity Diagnosis) – система для виявлення запозичень у програмному коді, яка використовує алгоритм заснований на використанні Колмогорової складності та алгоритму TokenCompress (детальніше див. пункт 1.3.4). Підтримує мови: Java та C++.

SID працює у два етапи (рис. 1.10):

- на першому етапі програмний код синтаксично аналізуються для створення послідовностей токенів;

- на другому етапі за допомогою алгоритму TokenCompress для всіх програм попарно обчислюється $d(x, y)$ – міра загальної інформації. Потім, всі пари сортуються за відстанями подібності.

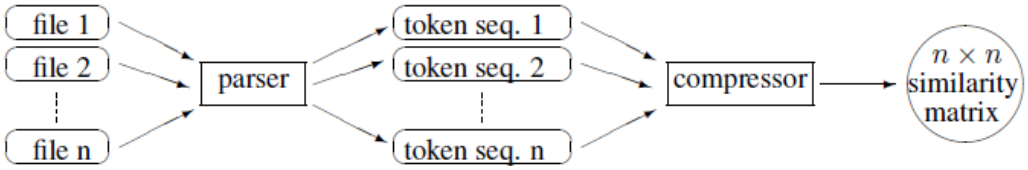


Рис. 1.10. Схема роботи системи SID

Автори SID стверджують, що обдурити систему дуже складно.

1.4.4. MOSS

MOSS (Measure Of Software Similarity) – система розроблена Стенфордським університетом в 1994 році. Для пошуку запозичень у програмному коді система використовує варіант реалізації *алгоритму відбитків документа, а саме – алгоритм просіювання.*

MOSS представлений веб-сервісом та підтримує до 23 мов програмування. Система генерує HTML-сторінки, які представляють результати з функціональним графічним інтерфейсом (рис. 1.10-1.11).

Moss Results

Tue Sep 8 23:29:31 PDT 2015

Options -l python -d -m 10

[How to Read the Results | Tips | FAQ | Contact | Submission Scripts | Credits]

File 1	File 2	Lines Matched
/home/ubuntu/Projects/work/2015/uct-csc1010h/tutorials/6/raw/[redacted]/ (99%)	/home/ubuntu/Projects/work/2015/uct-csc1010h/tutorials/6/raw/[redacted]/ (99%)	86
/home/ubuntu/Projects/work/2015/uct-csc1010h/tutorials/6/raw/[redacted]/ (76%)	/home/ubuntu/Projects/work/2015/uct-csc1010h/tutorials/6/raw/[redacted]/ (66%)	91
/home/ubuntu/Projects/work/2015/uct-csc1010h/tutorials/6/raw/[redacted]/ (81%)	/home/ubuntu/Projects/work/2015/uct-csc1010h/tutorials/6/raw/[redacted]/ (82%)	69
/home/ubuntu/Projects/work/2015/uct-csc1010h/tutorials/6/raw/[redacted]/ (70%)	/home/ubuntu/Projects/work/2015/uct-csc1010h/tutorials/6/raw/[redacted]3/ (61%)	70
/home/ubuntu/Projects/work/2015/uct-csc1010h/tutorials/6/raw/[redacted]/ (69%)	/home/ubuntu/Projects/work/2015/uct-csc1010h/tutorials/6/raw/[redacted]2/ (40%)	71
/home/ubuntu/Projects/work/2015/uct-csc1010h/tutorials/6/raw/[redacted]0/ (56%)	/home/ubuntu/Projects/work/2015/uct-csc1010h/tutorials/6/raw/[redacted]3/ (50%)	43
/home/ubuntu/Projects/work/2015/uct-csc1010h/tutorials/6/raw/[redacted]/ (62%)	/home/ubuntu/Projects/work/2015/uct-csc1010h/tutorials/6/raw/[redacted]9/ (55%)	67
/home/ubuntu/Projects/work/2015/uct-csc1010h/tutorials/6/raw/[redacted]/ (55%)	/home/ubuntu/Projects/work/2015/uct-csc1010h/tutorials/6/raw/[redacted]/ (48%)	40
/home/ubuntu/Projects/work/2015/uct-csc1010h/tutorials/6/raw/[redacted]0/ (54%)	/home/ubuntu/Projects/work/2015/uct-csc1010h/tutorials/6/raw/[redacted]0/ (55%)	40

Рис. 1.11. Загальний результат аналізу системи MOSS

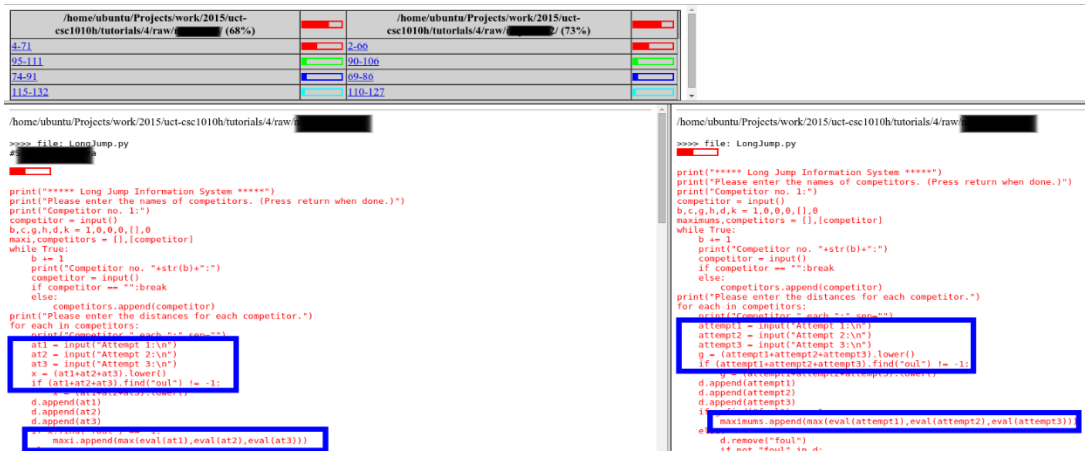


Рис. 1.12. Перегляд запозичень вихідного коду у системі MOSS

1.5. Висновки до першого розділу

У даному розділі розглянуто основні типи запозичень програмного коду з різними методиками для приховування факту запозичення. Всього було виділено 4 типи, кожен з яких є розширенням попереднього, за виключенням 4 типу, у якому два фрагменти синтаксично реалізовані по-різному, але виконують однакові дії. Цей тип видозміни коду необов'язковий для виявлення і не розглядається у даній роботі.

Наведено загальну класифікацію алгоритмів за методами, які лежать в їх основі:

- пошук запозичень методом текстового порівняння;
- пошук запозичень на основі атрибутів коду;
- пошук запозичень на основі структури коду.

Також було розглянуто найпопулярніші алгоритми, які використовується для виявлення запозичень у програмному коді:

- алгоритм локального вирівнювання рядків;
- алгоритм Левенштейна;
- алгоритм жадібного рядкового замощення;
- алгоритм з використанням Колмогорової складності в задачі пошуку запозичень у програмному коді;
- алгоритм відбитків документа.

Наведено загальний список систем для виявлення запозичень у програмному коді та окремо більш детально розглянуто чотири з них, які залишаються працездатними на даний момент:

- SIM;
- JPlag;
- SID;
- MOSS.

Проаналізовано три методи виявлення запозичень у програмному коді, можна зробити висновок, що кожний має свої недоліки і переваги. Виходячи з цього, було прийнято рішення розробити комбінований метод та реалізувати всі три підходи в рамках одного методу, що дозволить компенсувати недоліки кожного з них.

2. КОМБІНОВАНИЙ МЕТОД ПОШУКУ ЗАПОЗИЧЕНЬ У ПРОГРАМНОМУ КОДІ

2.1. Опис запропонованого методу для пошуку запозичень у програмному коді

Комбінований метод для пошуку запозичень програмного коду передбачає комплексний аналіз (див. Додаток 1. Схема роботи комбінованого методу пошуку запозичень у програмному коді) та включає в себе наступні етапи:

1. Нормалізація програмного коду.
2. Токенізація програмного коду.
3. Аналіз коментарів – реалізує метод текстового порівняння.
4. Аналіз структури коду – реалізує підхід на основі структури коду.
5. Аналіз атрибутів коду – реалізує підхід на основі атрибутів коду.
6. Формування результатів.

Варто зазначити, що кожний з етапів аналізу (етапи 3, 4 та 5) застосовується не до всього програмного коду, а лише до виділених його окремих частин на етапах нормалізації (етап 1) та токенізації (етап 2). Це дозволяє частково позбутися недоліків кожного з підходів до аналізу запозичень та зосередитися на їхніх перевагах.

Етапи нормалізації та токенізації необхідні для приведення програмного коду у вигляд, який полегшить пошук запозичень на етапах аналізу та нівелює вплив модифікацій характерних для першого та другого типу запозичень програмного коду (див. пункти 1.1.1-1.1.2).

Оскільки запропонований метод включає три етапи аналізу різних частин програмного коду, то виявлення запозичень займе більше часу, ніж окреме виконання одного типу аналізу. При цьому етапи аналізу незалежні та можуть виконуватися паралельно, що дозволить оптимізувати роботу методу та, за наявності обчислювальних потужностей, зменшити час виконання методу до часу виконання найдовшого з трьох етапів аналізу і

звести вплив такого об’єднання до мінімуму. Дана оптимізація методу є одним з напрямків роботи щодо його реалізації (див. пункт 3.4.2).

Однією з особливостей комбінованого методу пошуку запозичень у програмному коді є запропонований формат подання програмної структури, який дозволяє реалізувати можливість роботи з базою програмних кодів. Для цього для програмної структури на кожному етапі аналізу виділяються окремі характеристики, які потім будуть використовуватися для майбутніх порівнянь (детальніше див. підрозділ 2.8). В свою чергу, робота з базою програмних кодів вимагає більше ресурсних затрат та спеціальних алгоритмів на певних етапах аналізу.

2.2. Нормалізація програмного коду

Нормалізація – це процес перетворення всіх вхідних текстових даних, таким чином, щоб прибрати всі несуттєві частини. Даний етап виконує механічне оброблення програмного коду, тобто розглядає його як звичайний текст та виконує наступні модифікації (рис. 2.1):

- буквені символи переводяться в нижній регістр;
- символи порожнього простору (включаючи пробіл, табуляцію, прогін сторінки, крім символу нового рядка) замінюються одиничним пробільним символом;
- символ нового рядка або їх послідовність замінюється на токен <NEW_LINE>, який потім буде видалено на етапі токенізації.

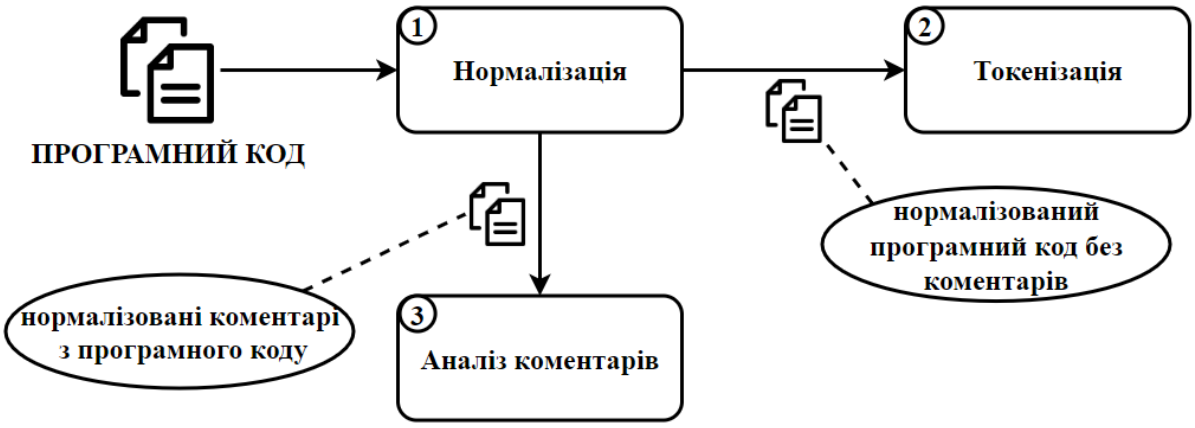


Рис. 2.1. Етап нормалізації програмного коду

На етапі нормалізації програмний код розбивається на дві частини (рис. 2.2):

- коментарі в нормалізованому форматі, які використовуються на етапі аналізу коментарів;
- програмний код в нормалізованому форматі з видаленими коментарями, який далі буде використовуватися на етапі токенізації.

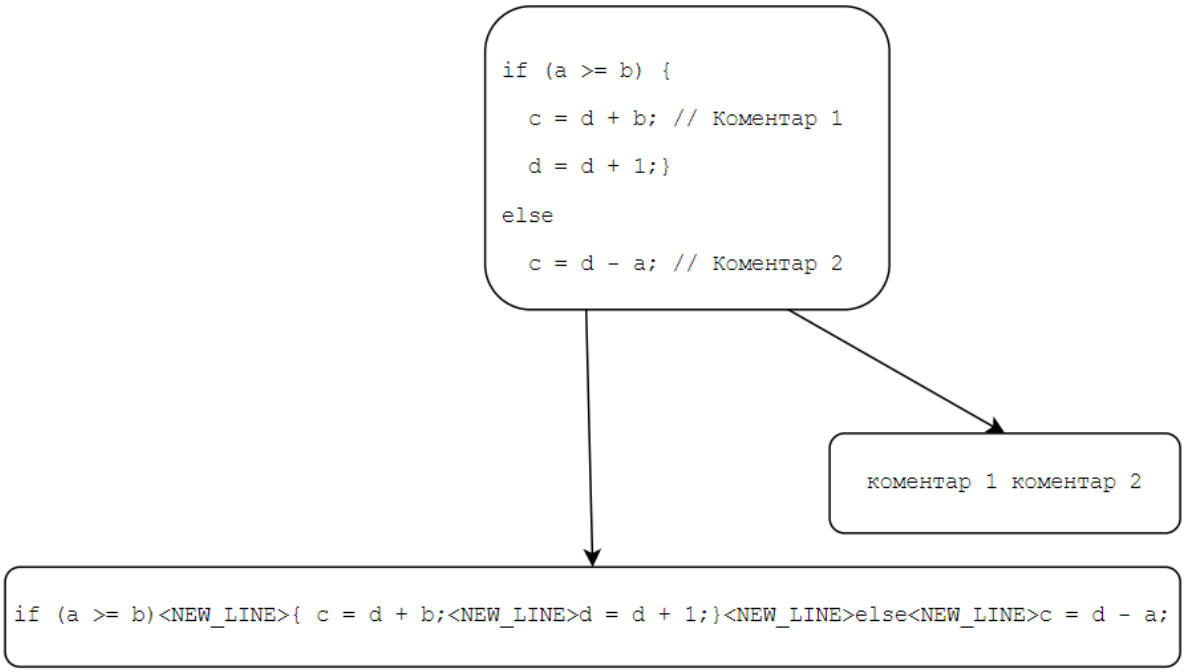


Рис. 2.2. Приклад нормалізації та виділення коментарів програмного коду

Для виділення коментарів у програмному коді на етапі нормалізації використовуються регулярні вирази [14]. Регулярні вирази – це формальна мова пошуку та здійснення модифікацій з рядками тексту за допомогою спеціальних синтаксичних правил. Регулярний вираз представляє собою правило пошуку (рис. 2.3), що складається із символів та метасимволів. Він дозволяє знайти у вхідному тексті всі підрядки, які задовольняють вказаному правилу. Далі за необхідністю над знайденими підрядками можна виконувати різні операції модифікації.

Регулярні вирази застосовуються в багатьох програмах для роботи з текстом, але якщо потрібно написати програму, що дозволяє обробляти або порівнювати текст для пошуку, регулярні висловлювання практично незамінні, їх функціональність підтримують всі сучасні мови програмування. Також регулярні вирази використовуються більшістю текстових редакторів.

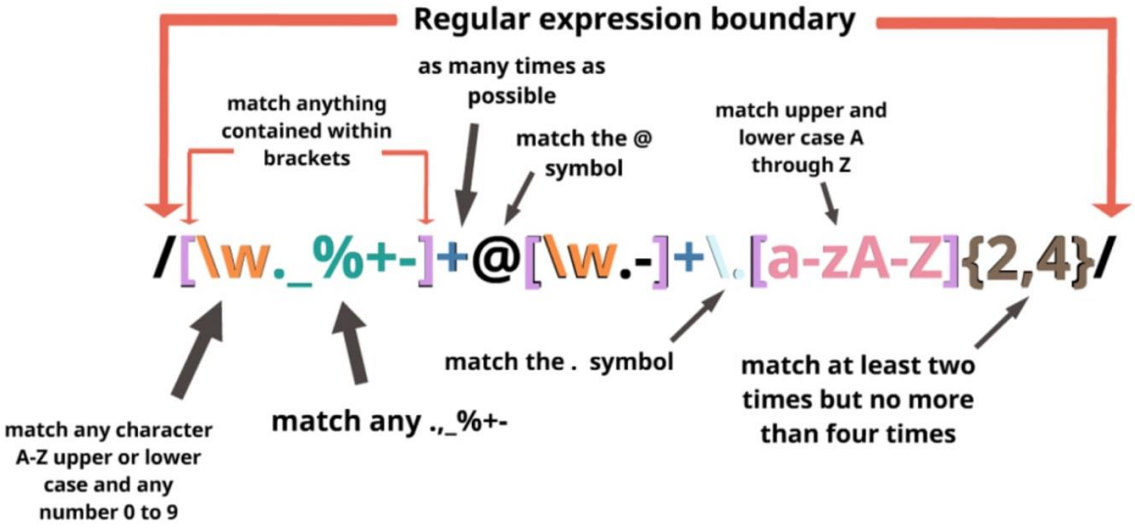


Рис. 2.3. Приклад регулярного виразу

У різних мовах програмування коментарі можуть мати декілька типів та різний синтаксис. Так наприклад сімейство С-подібних мов програмування (Java, C#, C++, JavaSxript) має однаковий спосіб задання та типи коментарів. Це багаторядкові коментарі (рис. 2.4), що призначені для коментування одного чи більше рядків.

```
/*
 *
 * Example of a multiline comment
 */
```

Рисунок 2.4 – Приклад багаторядкового коментарю в С-подібних мовах програмування

Також існують однорядкові коментарі, що призначені для коментування тільки одного рядка (рис. 2.5).

```
//Example of a single-line comment
```

Рисунок 2.5 – Приклад однорядкового коментарю в С-подібних мовах програмування

Для виділення різних типів коментарів для кожної мови програмування необхідно задати окремі регулярні вирази (табл. 2.1).

Таблиця 2.1

Приклад регулярних виразів для виділення коментарів в коді в С-подібних мовах програмування

Тип коментарю	Регулярний вираз
Багаторядковий	\/*(.|[\r\n])*?*\/
Однорядковий	\/{2,}.*

2.3. Токенізація програмного коду

Етап токенізації програмного коду перетворює нормалізоване представлення програмної структури методом виділення та подальшої заміни наступних видів лексем (рис. 2.6):

- ідентифікатори (назви змінних);
- ключові слова характерні для програмного коду;
- літерали (константи);
- оператори (знаки операцій);
- роздільники (пунктуаційні знаки) та парні роздільники (наприклад, фігурні дужки).

Процес токенізації залежить від мови програмування вихідного коду. Для вирішенням цього недоліку необхідно створити таблицю токенів для різних мов програмування і налаштувати роботи з цими таблицями. Створення нової таблиці не є складним процесом, так як багато мов мають схожий синтаксис (реалізацію процесу токенізації див. пункт 3.4.1).

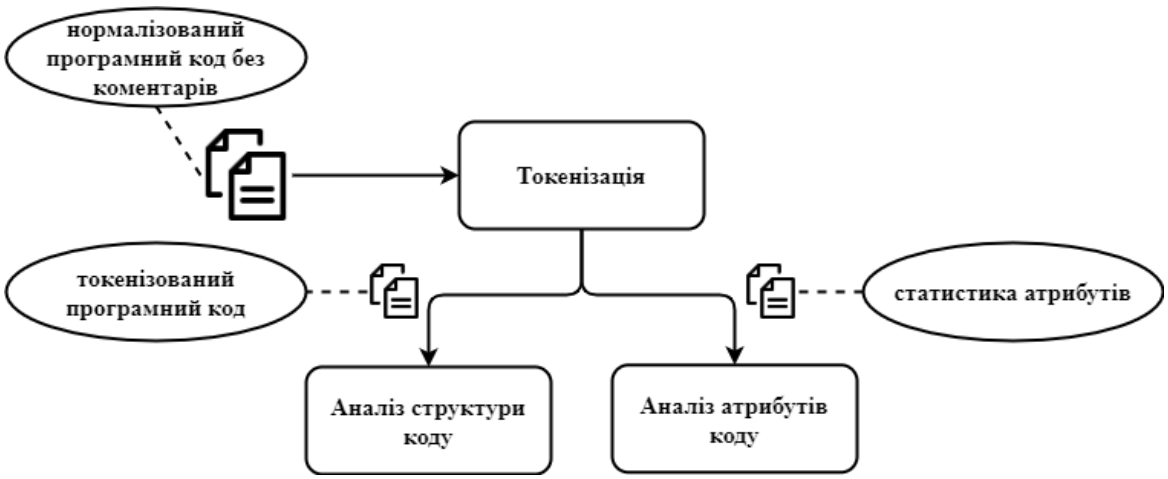


Рис. 2.6. Етап токенізації програмного коду

Токенізоване подання програмного коду використовується на етапі аналізу структури коду. Також під час процесу токенізації окремо збирається характеристика атрибутів програмного коду (див. підрозділ 2.6), яка потім використовується на етапі аналізу атрибутів коду.

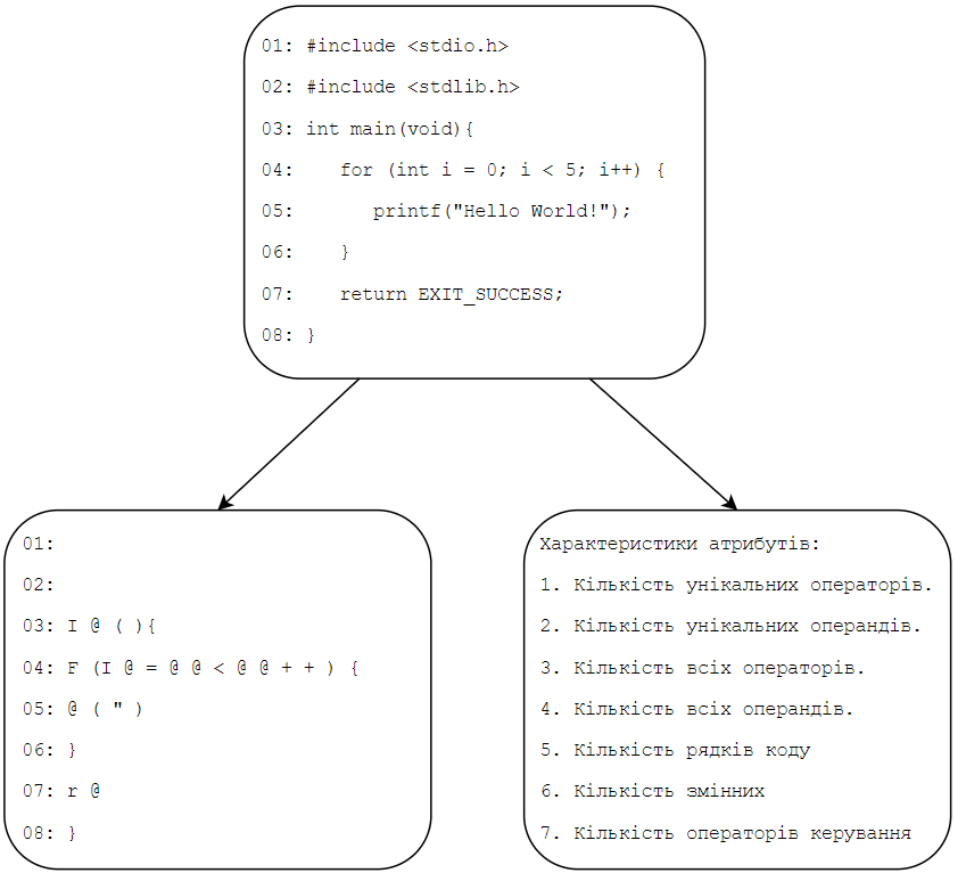


Рис. 2.7. Приклад токенізації та аналізу характеристик атрибутів програмного коду (* для наглядності процесу токенізації використовується не нормалізований програмний код)

2.4. Структурний аналіз програмного коду

Структурний аналіз коду представляє собою реалізацію підходу пошуку запозичень на основі структури коду.

2.4.1. Алгоритм відбитків для структурного аналізу

Задля забезпечення пошуку запозичень вхідного програмного коду в базі програмних кодів на даному етапі необхідно реалізувати структурний алгоритм, який передбачає таку можливість. Попарно порівнювати токенізоване представлення програми з всією базою є досить ресурсоємною операцію. Пропонується використати реалізацію алгоритму відбитків документа для структурного аналізу (див. пункт 1.3.5). Даний алгоритм дозволяє для токенізованого представлення програмного коду підрахувати послідовність міток (відбитків), які будуть характерні для конкретної програми. Такий підхід дозволить порівнювати не токенізовані представлення програм, а їх набори міток. Таким чином, по кількості однакових відбитків можна буде вирахувати подібність порівнюваних програм.

2.4.2. Алгоритм для вибору множини міток

У пункті 1.3.5 описано загальний алгоритм відбитків документу. Змістовною частиною цього алгоритму є другий крок – вибір множини міток серед послідовності хеш-значень. Нехай хеш-функція – h, а $h_1, h_2 \ldots h_n$ – послідовність значень хеш-функції, отриманих на першому кроці алгоритму відбитків. Розглянемо деякі можливі реалізації другого кроку:

1. Наївний підхід полягає у виборі кожного i-го з n хеш-значень. Але такий спосіб не стійкий до модифікацій коду. Якщо на початок файлу додати один зайвий символ, то в результаті вибору хеш-значень вийде зовсім інша множина міток.

2. Призначати мітками p мінімальних хеш-значень [15], їхня кількість для всіх документів завжди буде постійною. За допомогою цього методу не можна знайти часткові запозичення, але він добре працює на програмах приблизно одного розміру.
3. Манбер [7] запропонував вибирати в якості міток тільки ті хеш-значення, для яких $h \equiv 0 \ mod \ p$, так залишиться тільки n / p міток (об'єм ідентифікаційного набору для різних програм буде відрізнятися). Однак тоді відстань між послідовно вибраними хеш-значеннями не обмежена і може бути досить великою. У цьому випадку запозичення, що опинилися між мітками, не будуть враховані.
4. Алгоритм просіювання (winnowing) [7] не має недоліку методу Манбера. Алгоритм гарантує, що якщо у двох програмах є хоча б один досить довгий загальний підрядок, то як мінімум одна мітка в їх наборах має збігатися.

Щоб бути ефективнішим за інші реалізацій, алгоритм для вибору множини міток повинен гарантувати наступне:

- якщо у двох токенізованих програмах є спільний підрядок довжиною як мінімум t, то його буде знайдено;
- загальні підрядки, що коротші за шумовий поріг k повинні ігноруватися.

Дані вимоги забезпечуються алгоритмом просіювання.

Хід алгоритму просіювання (табл. 2.2)*:*

1. Обирається значення k, $k \le t$.
2. Вхідний рядок розбивається на підрядки довжиною k, так звані k-грами. Розраховуються хеш-значення підрядків: $h_1, h_2 \dots h_n$. Чим більше значення k, тим менш ймовірні випадкові збіги. Але із зростанням k також падає стійкість методу до перестановок.
3. Вздовж послідовності $h_1, h_2 \dots h_n$ просувається вікно розміром: $w = t - k + 1$.

4. На кожному кроці вікно рухається на одну позицію вправо. Міткою призначається мінімальне h_j у вікні. Якщо мінімальних елементів декілька, то призначається крайній правий.

Якщо в послідовних вікнах хеш-значення від одного і того ж підрядка є мінімальними, то міткою призначається тільки одне з них (немає сенсу зберігати послідовні абсолютно однакові мітки).

Таблиця 2.2

Приклад вибору міток за алгоритмом просіювання

Початковий рядок:	A do run run run, a do run run
Нормалізований рядок:	adorunrunrunadorunrun
Обираємо значення k, $k \leq t$	$t = 8$, $k = 5$
Послідовність 5-грам	adoru dorun orunr runru unrun nrunr runru unrun nruna runad unado nador adoru dorun orunr runru unrun
Послідовність хеш-значень для 5-грам	77 74 42 17 98 50 17 98 8 88 67 39 77 74 42 17 98
Вікна хеш значень з довжиною 4 ($w = t – k + 1$)	(77, 74, 42, **17**) (74, 42, 17, 98) (42, 17, 98, 50) (17, 98, 50, **17**) (98, 50, 17, 98) (50, 17, 98, **8**) (17, 98, 8, 88) (98, 8, 88, 67) (8, 88, 67, 39) (88, 67, **39**, 77) (67, 39, 77, 74) (39, 77, 74, 42) (77, 74, 42, **17**) (74, 42, 17, 98)
Мітки обрані алгоритмом просіювання	17 17 8 39 17
** Мітки згруповані з номером позиції в послідовності хеш-значень для 5-грам*	[17,3] [17,6] [8,8] [39,11] [17,15]

Ефективність алгоритму можна охарактеризувати показником щільності d – частка хеш-значень, обраних алгоритмом як мітки, серед усіх хеш-значень документа. Для методу просіювання: $d = \frac{2}{w+1}$ [7].

Інші реалізації алгоритму відбитків також можуть бути змінені таким чином, щоб дотримувалися описані вище вимоги, але для них показник d буде значно вищим за отриманий для методу просіювання. Тобто, для однієї і тієї ж програми, алгоритм просіювання гарантує найменшу множину міток. Це забезпечує найменшу кількість операцій порівнянь і вимагає менше ресурсів для роботи з базою даних, що і вимагається для етапу структурного аналізу програмного коду.

2.5. Аналіз коментарів програмного коду

В основі даного етапу лежить підхід виявлення запозичень методом текстового порівняння. Для виділення коментарів (як простих, так і багаторядкових) з програмного коду застосовується функціональність регулярних виразів на етапі нормалізації (див. підрозділ 2.2).

Для пошуку запозичень серед коментарів використовується алгоритм відбитків документа з алгоритмом просіювання, як і на етапі структурного аналізу програмного коду (див. підрозділ 2.4). Даний алгоритм гарантує аналогічну ефективність при застосовані до природнього тексту. Також він забезпечить можливість пошуку запозичень серед коментарів в базі програмних кодів.

Відсутність збігів серед коментарів не є гарантом унікальності програмних кодів, оскільки коментарі піддаються легкій модифікації або можуть бути повністю видаленими. Однак, досить високий відсоток їх схожості практично точно свідчить про наявність плагіату в програмах.

2.6. Аналіз атрибутів програмного коду

Етап реалізує підхід пошуку запозичень на основі аналізу атрибутів коду.

2.6.1. Вибір характеристик атрибутів програмного коду

Кожна програма має певну структуру даних, яка може бути використана в якості однієї з характеристик програмного коду. Для доведення факту запозичення ця характеристика повинна слабко змінюватися у випадку модифікації програмного коду або включення частин однієї програми в іншу. Для аналізу атрибутів коду пропонується використати характеристики, які застосовувалися для розробки системи пошуку запозичань на основі атрибутів коду Accuse [17]:

1. Кількість унікальних операторів.
2. Кількість унікальних операндів.
3. Кількість всіх операторів.
4. Кількість всіх операндів.
5. Кількість рядків коду.
6. Кількість змінних.
7. Кількість операторів керування.

Оператори керування визначають потік (логіку) виконання програми та діляться на декілька типів:

1. *Оператор if* – вирішує, чи виконувати наступний оператор, або який із двох операторів виконати.
2. *Оператор циклу* – вирішує, скільки разів виконувати настпуний оператор. Існує три види операторів циклу: 1) *цикл while* – перевіряє, чи є виконується задана умова істинною, перш ніж виконувати тіло циклу; 2) *цикл do-while* – перевіряє, чи є умова істинною після виконання тіла циклу; 3) *цикл for* – використовуються для виконання тіла циклу задану кількість разів.

3. *Оператор switch* – вирішує, який з *n*-ої кількості заданих операторів виконати згідно вказаної умови.

З метою запобігти модифікаціям програмного коду, характерним для третього типу запозичень (див. пункт 1.1.3), які можуть значно вплинути на коректність результату аналізу на основі атрибутів коду, для даного етапу введені додаткові оптимізації при підрахунку обраних характеристик:

- при підрахунку *Кількості всіх операторів* не рахувати оператори присвоєння (=);
- для кожного знайденого оператора присвоєння віднімати два операнди від значення *Кількість всіх операндів*, а також зменшити *Кількість рядків коду* на одиницю;
- для підрахунку *Кількість рядків коду* використовується спеціальний токен “<NEW_LINE>”, введений на етапі нормалізації замість символу нового рядка. При цьому послідовності з символами початку нового блоку коду не рахуються взагалі. Для С-подібних мов програмування: <NEW_LINE>{<NEW_LINE>,<NEW_LINE>}<NEW_LINE>, <NEW_LINE>else<NEW_LINE>. Після підрахунку *Кількість рядків коду* токен <NEW_LINE> видаляється з токенізованого представлення програмного коду.

Ці оптимізації гарантують ігнорування операцій ініціалізації (одна операція одночасно впливає на три характеристики) та хаотично розкиданого коду на нових рядках, що може бути спеціально зроблено для приховування факту плагіату і при цьому ніяк не впливати на результат роботи програмного коду.

2.6.2. Алгоритм знаходження кореляції на основі характеристик атрибутів програмного коду

На етапі токенізації програмного коду для кожної програми обчислюються значення семи характеристик. Тепер необхідно порівняти попарно кожну з характеристик та на основі результатів визначити відсоток подібності порівнюваних програм.

Для порівняння використовується функція кореляційної оцінки [203]:

$$sim(A, B) = \frac{CORRELATION_{AB}}{MAX\_CORRELATION} \cdot 100\% , \qquad (2.1)$$

де $CORRELATION_{AB}$ – коефіцієнт кореляції між програмами *A* і *B*;
MAX_CORRELATION – максимально можливий коефіцієнт кореляції.

Коефіцієнт кореляції залежить від двох констант, які призначаються кожній із семи характеристик коду (табл. 2.3):

- розмір вікна (“*window*”) – максимальна допустима різниця між двома значеннями певної характеристики коду, при якій програми можуть вважатися копіями;
- коефіцієнту важливості (“*importance*”) – вага певної характеристики коду в загальному коефіцієнті кореляції.

Коефіцієнт кореляції між програмами *A* і *B* обчислюється за таким алгоритмом:

1. Значення кореляції для пари програм *A* і *B* спочатку дорівнює нулю*:* $CORRELATION_{AB} = 0$.
2. Для кожної з характеристик атрибутів обчислюється модуль різниці: якщо A_i – значення *i*-ї характеристики програми *A*, а B_i – відповідна характеристика програми *B*, то $diff_i = |A_i – B_i|$.
3. Якщо $diff_i \le window_i$, тобто різниця допустима для *i*-ї характеристики, то $CORRELATION_{AB} \mathrel{+}= importance_i – diff_i$.

Значення констант розміру вікна та коефіцієнту важливості наведені в роботі [17]. Але для реалізації зазначеного підходу в рамках

комбінованого методу використано значення коефіцієнту важливості наведеного в праці [18], які було отримано емпіричним методом на основі робіт студентів. Різні студенти виконували одні і ті ж завдання на початку першого курсу, наприкінці першого та на початку другого курсу відповідно. Їх результати ілюструють різні навички програмування та все більш складні способи для прикриття факту плагіату.

Таблиця 2.3

Значення розміру вікна та коефіцієнту важливості для характеристик

№	Характеристика коду	Розміру вікна *window*	Коефіцієнту важливості *importance*
1	Кількість унікальних операторів	5	5
2	Кількість унікальних операндів	5	5
3	Кількість всіх операторів	3	7
4	Кількість всіх операндів	3	7
5	Кількість рядків коду	3	6
6	Кількість змінних	2	4
7	Кількість операторів керування	1	3
$MAX\_CORRELATION = \sum_{i=1}^{7} importance_i = 37$			

2.6.3. Приклад аналізу атрибутів програмного коду

Розглянемо покрокове виконання аналізу на основі атрибутів коду. Нехай порівнюється дві програми *A* та *B*. Після етапів нормалізації та токенізації характеристики їх атрибутів виглядають наступним чином (табл. 2.4):

Таблиця 2.4

Характеристики атрибутів програм *A* та *B*

№	Характеристика коду	Програма *A*	Програма *B*
1	Кількість унікальних операторів	25	24
2	Кількість унікальних операндів	13	12
3	Кількість всіх операторів	118	110
4	Кількість всіх операндів	106	90
5	Кількість рядків коду	63	64
6	Кількість змінних	11	11
7	Кількість операторів керування	4	4

Коефіцієнт кореляції між парами програм *A* і *B* рахується наступним чином:

1. Кількість унікальних операторів: $diff_1 = |25 - 24| = 1 \leq 5$ – програми знаходяться в межах розміру вікна, і для цієї пари програм розраховується приріст коефіцієнту кореляції: $CORRELATION_{AB}^1 = importance_1 - diff_1 = 5 - 1 = 4$.
2. Кількість унікальних операндів: $diff_2 = |13 - 12| = 1 \leq 5$ – програми знаходяться в межах розміру вікна, і для цієї пари програм розраховується приріст коефіцієнту кореляції: $CORRELATION_{AB}^2 = CORRELATION_{AB}^1 + importance_2 - diff_2 = 4 + 5 - 1 = 8$.
3. Кількість всіх операторів: $diff_3 = |118 - 110| = 8 > 3$ – програми знаходяться за межами розміру вікна, коефіцієнт залишається попереднім: $CORRELATION_{AB}^3 = CORRELATION_{AB}^2$.

4. Кількість всіх операндів: $diff_4 = |106 - 90| = 16 > 3$ – програми знаходяться за межами розміру вікна, коефіцієнт залишається попереднім: $CORRELATION^4_{AB} = CORRELATION^3_{AB}$.
5. Кількість рядків коду: $diff_5 = |63 - 64| = 1 \leq 3$ – програми знаходяться в межах розміру вікна, і для цієї пари програм розраховується приріст коефіцієнту кореляції: $CORRELATION^5_{AB} = CORRELATION^4_{AB} + importance_5 - diff_5 = 8 + 6 - 1 = 13$.
6. Кількість змінних: $diff_6 = |11 - 11| = 0 \leq 2$ – програми знаходяться в межах розміру вікна, і для цієї пари програм розраховується приріст коефіцієнту кореляції: $CORRELATION^6_{AB} = CORRELATION^5_{AB} + importance_6 - diff_6 = 13 + 4 - 0 = 17$.
7. Кількість операторів керування: $diff_7 = |4 - 4| = 0 \leq 1$ – програми знаходяться в межах розміру вікна, і для цієї пари програм розраховується приріст коефіцієнту кореляції: $CORRELATION^7_{AB} = CORRELATION^6_{AB} + importance_7 - diff_7 = 17 + 3 - 0 = 20$.

Отримавши коефіцієнт кореляції між парами програм, рахується відсоток подібності за формулою (2.1), що і буде результатом пошуку запозичень на основі атрибутів коду:

$$sim(A, B) = \frac{CORRELATION_{AB}}{MAX\_CORRELATION} \cdot 100\% = \frac{20}{37} \cdot 100\% \approx 54.05\%.$$

2.7. Формування результатів роботи методу

Формування результатів виявлення запозичень програмного коду передбачає:

1. Отримання відсотку схожості між програмами за кожним з етапів аналізу окремо – аналіз коментарів, аналіз атрибутів коду, структурний аналіз.

2. Встановлення порогових значень для відсоткового збігу програмних текстів окремо для кожного з етапів аналізу. В результатах пошуку запозичень будуть відображені лише ті програми, для яких відсоток схожості більше вказаного порогового значення хоча б для одного з етапів аналізу.
3. Виділення ділянок програмного коду, підозрілих в запозиченні.

Даний формат представлення результатів роботи методу було обрано з метою надати користувачу загальну статистику запозичень між програмами. Як і існуючі реалізації різних підходів до запозичень у програмному методі, запропонований метод не дає власної оцінки чи є програма плагіатом. При цьому він дає більшу повноту результатів, що дозволяє точніше визначити, чи є конкретна програма плагіатом, але таке рішення є суб'єктивним і повинно виноситися користувачем.

2.8. Формат представлення програмної структури

На кожному з етапів аналізу (підрозділи 2.4-2.6) запропоновано спеціальний формат подання програмної структури, який далі використовується різними алгоритмами для порівняння програм та визначення відсотку співпадіння між ними. Програмна структура подається наступним чином (рис. 2.8):

- для етапу «аналізу атрибутів коду» зберігаються кількісні характеристики атрибутів;
- для етапу «аналізу структури коду» послідовність міток для токенізованого програмного коду;
- для етапу «аналізу коментарів» послідовність міток для коментарів.

Такий формат подання програмної структури необхідний для алгоритмів, що використовуються в рамках комбінованого методу. Користь від його застосування можна розширити, наприклад, за рахунок використання бази даних програмних кодів.

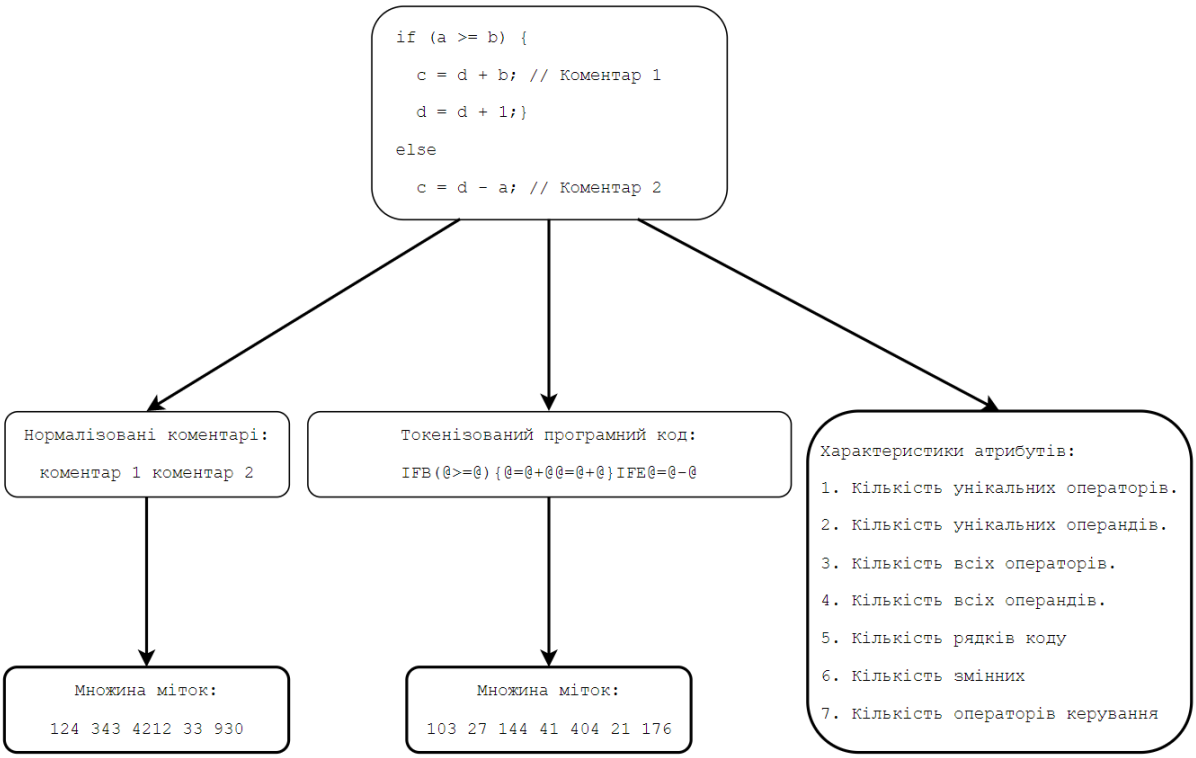


Рис. 2.8. Приклад подання програмної структури

Класичний пошук запозичень у програмному коді передбачає, що на вхід методу подається множина програм, вони обробляються певним чином, аналізуються попарно між собою, та в результаті рахується статистика запозичень між вхідною множиною програм. Для систем аналізу запозичень у програмному коді робота з базою даних є важливою опцією. З підтримкою даної функціональності для вхідної програми можна виконати пошук запозичень серед збережених раніше в базі даних інших програмних структур. При цьому виконувати пошук запозичень між програмами в «сирому» вигляді не ефективно, адже для однієї і тієї ж програмної структуру з бази даних доведеться постійно виконувати етапи підготовки коду (нормалізація, токенізація). Формат подання програмної структури, що застосовується в рамках запропонованого комбінованого методу дозволяє позбутися цього недоліку, адже пошук запозичень відбувається серед множини міток та характеристик програм. Зберігання програми в даному форматі дозволяє використовувати її в майбутніх порівняннях відразу на

етапах аналізу, пропустивши підготовчі етапи нормалізації та токенізації (рис. 2.9), що пришвидшить пошук запозичень.

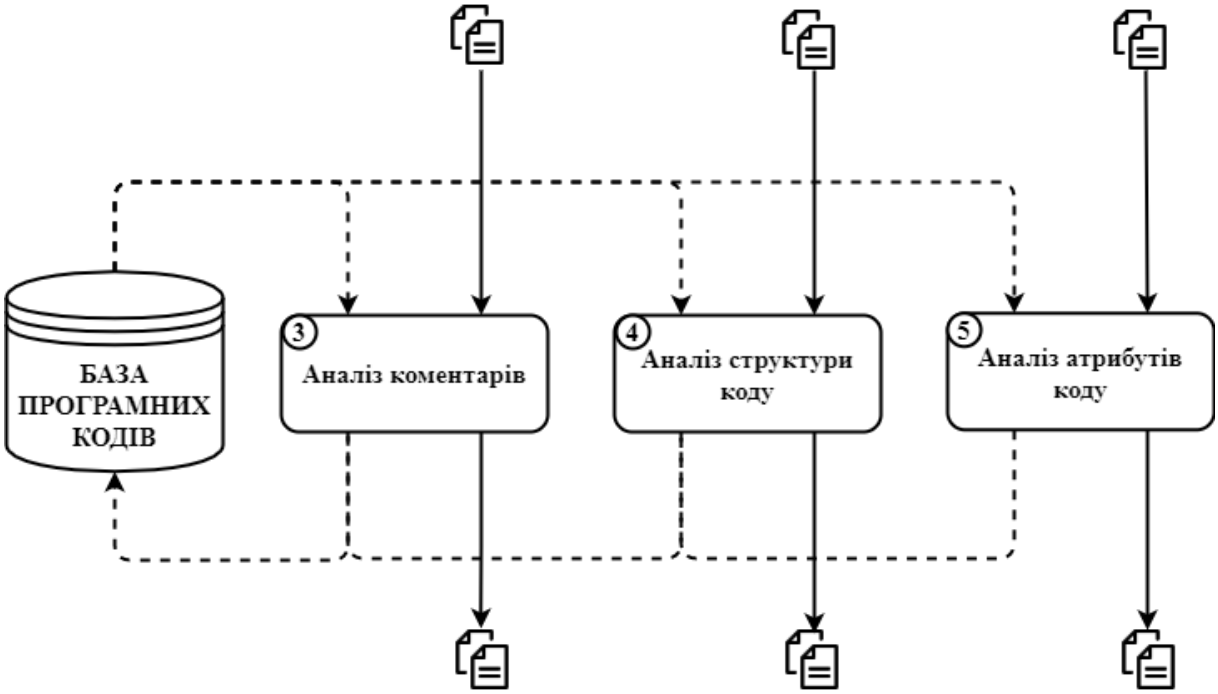


Рис. 2.9. Взаємодія з базою даних на етапах аналізу

2.9. Висновки до другого розділу

У даному розділі було розглянуто запропонований комбінований методу пошуку запозичень у програмному коді, який складається з шести етапів:

1. Нормалізація програмного коду.
2. Токенізація програмного коду.
3. Структурний аналіз програмного коду.
4. Аналіз коментарів програмного коду.
5. Аналіз атрибутів програмного коду.
6. Формування результатів роботи методу.

Кожний з етапів було розглянуто окремо та описаного його ключові особливості. На етапах аналізу структури коду та коментарів для пошуку запозичень було використано алгоритм просіювання з алгоритмом відбитків документа. Для етапу аналізу атрибутів коду було обрано сім характеристик

атрибутів програмного коду: кількість унікальних операторів, кількість унікальних операндів, кількість всіх операторів, кількість всіх операндів, кількість рядків коду, кількість змінних, кількість операторів керування. Для визначення схожості між порівнюваними програмами за обраними атрибутами було описано алгоритм обрахунку на основі оцінки кореляції.

Окремо описано запропонований формат представлення програмної структури, який використовується на кожному з етапів аналізу. Описано переваги використання даного формату, зокрема використання для організації пошуку в базі даних програмних кодів.

3. ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ ДЛЯ АВТОМАТИЗОВАНОГО ВИЯВЛЕННЯ ЗАПОЗИЧЕНЬ У ПРОГРАМНОМУ КОДІ

3.1. Обґрунтування вибору засобів реалізації програмного забезпечення

Для демонстрації роботи запропонованого комбінованого методу пошуку запозичень у програмному коді було розроблено програмне забезпечення у вигляді бібліотеки з можливістю повноцінного її використання за допомогою інтерфейсу командного рядка (CLI). Було використано наступні технології:

1. В якості основного інструменту розроблення використано мову програмування Java [20].
2. В якості СУБД було обрано SQL базу даних, а саме H2 [21].
3. В якості допоміжного інструменту для токенізації програмного коду використано генератор синтаксичних аналізаторів ANTLR [22].

3.1.1. Тип реалізації програмного забезпечення

Реалізація запропонованого методу у вигляді бібліотеки носить практичний характер. Справа в тому, що існуючі системи для пошуку запозичень у програмному коді реалізовані як веб- або десктопні додатки. Вони мають веб-/GUI-інтерфейс для використання та жодних можливостей для інтеграції з стороннім ПЗ. Створення бібліотеки з реалізацією пошуку запозичень у програмному коді дозволяє позбавитися цього недоліку та використовувати її можливості для розробки багатьох сервісів, які в рамках запропонованої функціональності виконують перевірку програмного коду на плагіат. Більш того, оскільки бібліотека лише надає результати аналізу, це дозволить сервісам, що її використовують, самостійно реалізовувати інтерфейс користувача, налаштовувати власні модифікації.

Бібліотека для пошуку запозичень у програмному може бути легко інтегрована в інші типи додатків, зокрема десктопні та веб-додатки, також на її основі можна реалізувати веб-плагіни чи бот-сервіси для різних платформ (наприклад, Telegram). Як ідея для прямого використання розробленої бібліотеки може слугувати сервіс GitHub Classroom або веб-додаток для автоматизації виявлення плагіату в навчальних програмних проєктах розроблений в рамках бакалаврського дипломного проєкту [19].

Окремою особливістю розробленої бібліотеки є її можливість роботи за допомогою CLI. При використанні бібліотеки таким чином буде згенерований HTML-звіт з елементарним інтерфейсом для перегляду результатів пошуку запозичень у програмному коді. Даний спосіб взаємодії з бібліотекою розроблений лише для демонстраційних цілей роботи методи, без необхідності реалізовувати власне ПЗ.

3.1.2. Мова програмування Java

Java – це високорівнева, об'єктно-орієнтована мова програмування, яка розроблена так, щоб мати якомога менше залежностей в її реалізації. Java представляється як мова програмування загального призначення дозволяє програмістам писати код один раз та запускати його в будь-якому середовищі. Тобто це означає, що скомпільований код Java може працювати на всіх платформах, які її підтримують Java, без необхідності повторної компіляції.

Мова програмування Java має наступні характеристики:

1. *Об'єктно-орієнтованість.* Для забезпечення ефективного функціонування систем у все більш складних мережевих умовах мови програмування стали приймати об'єктно-орієнтовані концепції. Технологія Java забезпечує платформу розробки додатків на основі об'єктів (екземплярів класів).
2. *Архітектурна незалежність (крос-платформеність).* Компілятор генерує об'єктні файли, формат яких не залежить від

архітектури комп'ютера. Скомпільована програма може виконуватися на будь-яких процесорах, а для її роботи потрібно лише виконуюча система Java. Код, що генерується компілятором Java, називається байт-кодом. Він розроблений таким чином, щоб його можна було легко інтерпретувати на будь-якій машині або перетворити в власний машинний код.

3. *Надійність.* Java забезпечує перевірку часу компіляції і перевірку часу виконання. Виконання додатка контролює віртуальна машина Java (JVM), що не допускає таких ситуацій, як пошкодження ділянок пам'яті за межами адресного простору процесу. Мова має просту модель управління пам'яттю: об'єкти створюються за допомогою оператора *new* зі строгими посиланнями на них, немає вказівників і є процес автоматичного видалення непотрібних об'єктів (прибирання сміття).
4. *Безпечність.* Безпека Java має значення, оскільки технологія призначена для роботи в розподіленому середовищі. Java дозволяє створювати додатки, в які не можна вторгнутися ззовні.
5. *Висока продуктивність.* Інтерпретатор Java може працювати без необхідності перевірки середовища виконання. А байт-код можна транслювати під час виконання програми в машинний код для того процесора, на якому виконується дана програма.
6. *Багатопотоковість.* Java підтримує програми, які виконують кілька операцій одночасно, що сприяє швидкодії програмного забезпечення та високій інтерактивності для кінцевого користувача. Java підтримує багатопоковість на рівні мови. Більш того, високорівневі системні бібліотеки в Java є потокобезпечними, тобто надають безконфліктний доступ до функціональності конкуруючим потокам виконання.
7. *Динамічність.* Мова Java забезпечує зв'язування класів тільки в разі потреби. Новий код може бути доданий з різних джерел. Java

базується на роботі з бібліотеками, що робить мову легко адаптованою до постійно мінливого середовища.

Однією з головних перевагою мови програмування Java при її виборі є її архітектурна незалежність. Це дуже важлива перевага при розробці програмного забезпечення у вигляді бібліотеки з можливістю повноцінного її використання за допомогою CLI. Бібліотеки для мови програмування Java представлені архівними файлами в форматі JAR. При цьому JAR-файл може бути виконаний як окрема незалежна програма за допомогою JVM, що і вимагається для роботи з бібліотекою за допомогою CLI.

3.1.3. СУБД H2 та технологія взаємодії з обраною мовою програмування

H2 – це легковісна база даних з відкритим кодом, написана на мові програмування Java. H2 позиціонує себе як тестова база даних для використання під час розробки чи для демонстраційних цілей і не призначена для комерційного використання.

Головні особливості бази даних:

1. Надзвичайно швидкий механізм роботи транзакцій.
2. Невеликий розмір – розміру файлу JAR, близько 1.5 МБ.
3. Два режими роботи – клієнт-сервер, вбудований.
4. Два режими зберігання даних – файлова система та оперативна пам'ять.
5. Підтримка стандартну SQL 2003 і JDBC API.
6. Підтримка кластеризації і реплікації.
7. Підтримка потужних функцій безпеки – шифрування бази даних (AES), шифрування пароля SHA-256, функції шифрування та SSL.

База даних H2 була обрана для використання бібліотекою через малий розмір, можливість вбудованого режиму роботи та високу продуктивність взаємодії з мовою програмування Java. Варто відмітити, що дана база даних

використовується лише для демонстраційних цілей в режимі роботи через CLI. Якщо бібліотеку використовувати для розробки стороннього сервісу, то для цього передбачена можливість налаштування взаємодії бібліотеки з будь-якою зовнішньою SQL базою даних.

Для зручності роботи з БД на мові програмування Java використана технологія Spring Data JPA – одна з частин фреймворку Spring Data, яка реалізує стандарт JPA та використовує ORM фреймворк Hibernate [23].

3.1.4. Інструмент генерації синтаксичних аналізаторів ANTLR

ANTLR – генератор синтаксичних аналізаторів, що надає можливість автоматично створювати програму-парсер та лексичний аналізатор однією з декількох мов програмування (Java, C++, C#, Python, Ruby) за описом LL-граматики (лістинг 3.1). Дозволяє конструювати компілятори, транслятори, інтерпретатори для різних формальних мов. Дана технологія була використана при розробці як допоміжний інструмент на етапі токенізації програмного коду.

Лістинг 3.1. Коротка довідка елементів LL-граматики

```
(...) підправило
(...)* повторення підправила 0 або більше разів
(...)+ Повторення підправила 1 або більше разів
(...)? підправило, може бути відсутнім
{...} семантичні дії (на вхідній мові програмування, наприклад, Java)
[...] параметри правила
| оператор альтернативи
.. оператор діапазону
~ заперечення
. будь-який символ
= привласнення
: мітка початку правила
; мітка кінця правила
```

Переваги:

1. Вільне програмне забезпечення.
2. Використання єдиної нотації для опису лексичних та синтаксичних аналізаторів.

3. Зручність роботи з абстрактним синтаксичним деревом.
4. Надання повідомлень про помилки та відновлення після них.
5. Наявність візуальних середовищ розробки (ANTLR Works, ANTLR Studio, плагінів до Eclipse та IntelliJ IDEA), які дозволяють створювати та налагоджувати граматику, підтримують підсвічування синтаксису, автодоповнення, візуальне відображення граматик, що будується в реальному часі.

Лістинг 3.2. Приклад граматику для лексичного та синтаксичного аналізу арифметичних виразів з невід'ємними числами

```
grammar ArithmExpression;

// визначення правил граматики
stat : stat (MUL|DIV) stat
     | stat (PLUS|MINUS) stat
     | INTEGER
     | REAL
     | OP_BR stat CL_BR;

// визначення токенів
OP_BR: '(';
CL_BR: ')';
PLUS: '+';
MINUS: '-';
MUL: '*';
DIV: '/';
INTEGER: ('0' .. '9')+;                  // невід'ємні цілі числа
REAL: ('0' .. '9')+ ',' ('0' .. '9')+;  // невід'ємні дійсні числа

// пропуск пробілу чи табуляції у виразі під час лексичного аналізу
WHITESPACE: [ \t] -> skip;
```

3.2. Архітектура розробленого програмного забезпечення

Програмне забезпечення для пошуку запозичень у програмному коді реалізоване у вигляді бібліотеки. Основні компоненти ПЗ та зв’язки між ними зображено на структурній схемі бібліотеки (див. Додаток 1. Структурна схема бібліотеки). В рамках розробленого програмного забезпечення можна виділити наступні модулі (рис. 3.1):

1. *Java API.* Контролює роботу всієї бібліотеки. Надає публічний інтерфейс для використання можливостей розробленого ПЗ та задає необхідний формат вхідних і вихідних даних.

2. *Інтерпретатор командного рядка.* Модуль для взаємодії з бібліотекою за допомогою CLI. Відповідальний за переформатування введеної команди користувача в програмні класи та використання функціональності модулю Java API для запуску процесу пошуку запозичень.
3. *Модуль попередньої обробки програмних текстів.* Реалізує логіку етапів нормалізації і токенізації програмного коду. На основі вхідних даних з Java API разом з модулем для взаємодії з файловою системою зчитує файли програмного коду та виконує їх обробку.
4. *Модуль пошуку запозичень у програмному коді.* Виконує пошук запозичень програмного коду на основі даних, отриманих з модулю попередньої обробки програмних текстів. Також використовує модуль для взаємодії з базою даних для пошуку запозичень в базі програмних кодів та збереження формату подання програмних структур, який використовується для аналізу.
5. *Модуль для взаємодії з базою даних.* Допоміжний модуль у випадку, якщо використовується функціональність пошуку запозичень у базі програмних кодів. Даний модуль також дозволяє налаштувати підключення до будь-якої SQL бази даних в рамках Java-програми, що використовує розроблену бібліотеку. Без вказання налаштувань (лістинг 3.3) буде використовуватися вбудована в бібліотеку база даних H2.

Лістинг 3.3. Налаштування підключення до зовнішньої SQL бази даних

```
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.username=mysqluser
spring.datasource.password=mysqlpass
spring.datasource.url=jdbc:mysql://localhost:3306/myDb
```

6. *Модуль для формування результатів аналізу*. На основі отриманих даних формує результати пошуку запозичень у програмному коді, генерує та зберігає файли для перегляду результатів на файловій системі для користувача CLI чи створює програмні класи з результатами аналізу для Java-програм.
7. *Модуль для взаємодії з файловою системою.* Відповідальний за зчитування файлів програмного коду для пошуку запозичень та збереження результатів аналізу на файловій системі.

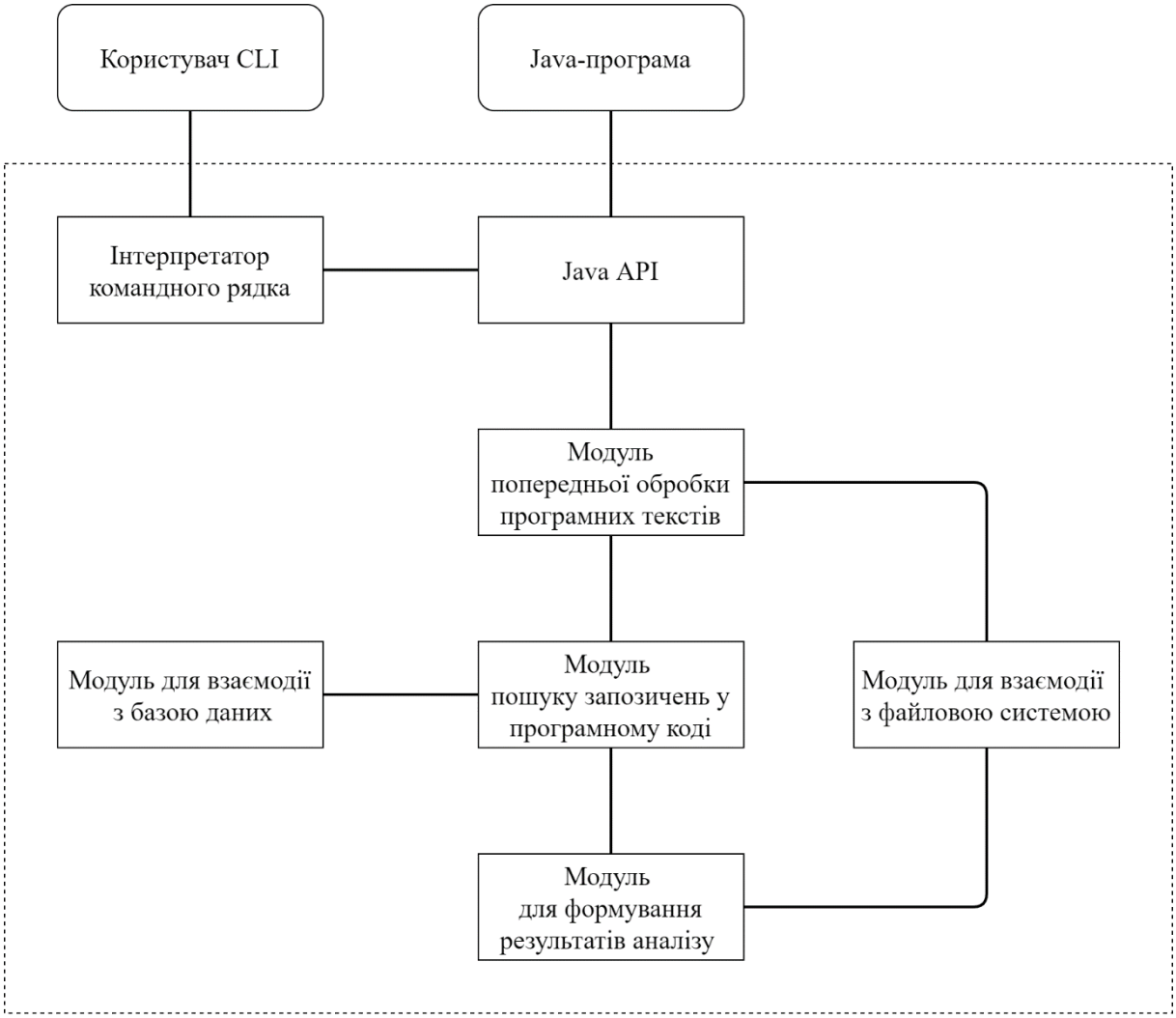


Рис. 3.1. Схема модулів розробленого програмного забезпечення

Для більш детального розуміння взаємодії програмних класів в рамках розробленого програмного забезпечення пропонується розглянути її архітектуру. Бібліотека для пошуку запозичень у програмному коді має класичну багаторівневу архітектуру [24]. Кожен з модулів, описаних вище належить певному рівню. Всі елементи організовані в горизонтальні шари, кожен рівень виконує конкретну роль та несе певну відповідальність у застосуванні. Рівні архітектури бібліотеки (рис. 3.2):

1. *Рівень представлення (presentation layer).* Розміщуються модулі Інтерпретатору командного рядка та Java API. Відповідальний за обробку запитів від користувача та перевірку вхідних даних. Описує публічний API бібліотеки.
2. *Рівень фасадів (facade layer).* Розміщуються класи фасадів та конвертори. Даний рівень відповідальний за конвертацію об’єктів DTO з рівня представлення в об’єкти моделей та навпаки. Моделі використовуються на рівні бізнес-логіки та зберігаються в базі даних. DTO необхідні для передачі даних між бібліотекою і Java-програмою.
3. *Рівень сервісів (service layer).* Розміщуються сервіси, які містять всю логіку роботи комбінованого методу пошуку запозичень у програмному коді. На даному рівні розміщуються модулі попередньої обробки програмних текстів, пошуку запозичень у програмному коді та модуль для формування результатів аналізу. Окремо варто відмітити, що рівень сервісів взаємодіє зі спеціальними підмодулями, які реалізують підтримку конкретної мови програмування.
4. *Рівень зберігання та доступу до даних (persistence layer).* Розміщуються модулі бібліотеки, які взаємодіють з файловою системою та базою даних для отримання, зберігання та обробки даних.

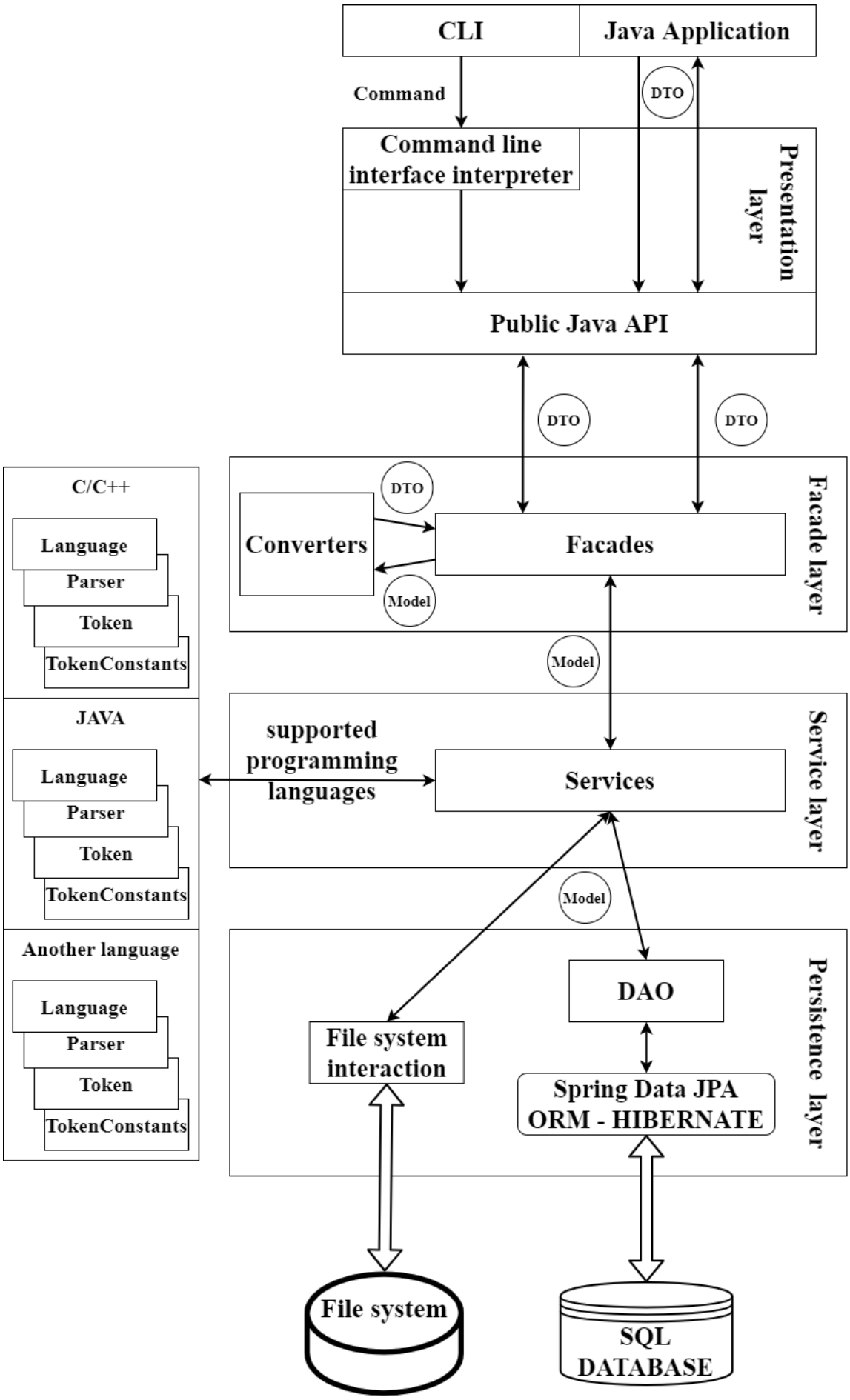


Рис. 3.2. Архітектура програмного забезпечення

3.3. Структура бази даних

Для підтримки можливості роботи комбінованого методу з базою програмних кодів в рамках розробленої бібліотеки спроєктовано схему реляційної бази даних. Таблиці БД, їх зв'язки, первинні і зовнішні ключі та опис полів сутностей наведено у вигляді ER-діаграми (див. Додаток 1. Структура бази даних). Таблиці приведено у першу, другу та третю нормальні форми [25]. Нижче детально описано сутності спроєктованої ER-діаграми з описом їх призначення та атрибутів.

Програма (табл. programs) – програма збережена в базі даних:

1. Унікальний ідентифікатор програми (program_id) – первинний ключ.
2. Унікальний ідентифікатор мови програмування (programming_ language_id) – зовнішній ключ.
3. Назва програми (name).
4. Дата зберігання (storage_date).

Мова програмування (табл. programming_languages) – мова програмування відповідної програмної структури:

1. Унікальний ідентифікатор мови програмування (programming_ language_id) – первинний ключ.
2. Назва мови програмування (name).

Значення атрибутної характеристики (табл. attribute_characteristic_ values) – кількісне значення конкретної характеристики атрибутів для збереженої програми:

1. Унікальний ідентифікатор значення атрибутної характеристики (attribute_characteristic_value_id) – первинний ключ.
2. Унікальний ідентифікатор програми (program_id) – зовнішній ключ.
3. Унікальний ідентифікатор атрибутної характеристики (attribute_ characteristic_id) – зовнішній ключ.

4. Значення (value).

Атрибутна характеристика (табл. attribute_characteristics) – одна з семи атрибутних характеристик для аналізу атрибутів програмного коду (див. пункт 2.6.1):

1. Унікальний ідентифікатор атрибутної характеристики (attribute_characteristic_id) – первинний ключ.
2. Назва (name).
3. Значення константи вікна (window).
4. Значення коефіцієнта важливості (importance).

Послідовність міток документа (табл. document_fingerprints) – подання програми або її коментарів у форматі послідовності міток документа:

1. Унікальний ідентифікатор послідовності міток документа (document_fingerprints_id) – первинний ключ.
2. Унікальний ідентифікатор програми (program_id) – зовнішній ключ.
3. Текст (text) – програмний текст або коментарі, на основі яких отримано дану послідовність міток документа.
4. Довжина підрядка (substring_length) – величина параметра *k* алгоритму просіювання.
5. Довжина спільного підрядок (common_substring_length) – величина параметра *t* алгоритму просіювання.

Мітка (табл. fingerprints) – мітка з послідовності міток документа:

1. Унікальний ідентифікатор мітки (fingerprints_id) – первинний ключ.
2. Унікальний ідентифікатор послідовності міток документу (document_fingerprints_id) – зовнішній ключ.
3. Хеш-значення мітки (value).
4. Позиція мітки у вікні (position).
5. Порядок мітки у загальній послідовності мітків документу.

3.4. Особливості реалізації програмного забезпечення

В реалізації запропонованого комбінованого методу пошуку запозичень у програмному коді було реалізовано ряд структурних особливостей для покращення ефективності роботи ПЗ, додано функціональність для ігнорування шаблонного коду під час аналізу та спроєктовано систему таким чином, щоб в майбутньому можна було легко додавати підтримку нових мов програмування. Пропонується розглянути дані особливості більш детально.

3.4.1. Реалізація підтримки нової мови програмування

Етап токенізації програмного коду в рамках комбінованого методу залежить від мови програмування, якою написані порівнювані програмні структури. Тобто щоб запропонований метод повноцінно міг аналізувати програмну структуру на певній мові програмування, необхідно реалізувати її підтримку в рамках процесу токенізації. Розроблена бібліотека може виконувати пошук запозичень для наступних мов програмування: Java, C, C++, C#, Python3 .

Додати підтримку нової мови програмування досить просто. Для початку необхідно ознайомитися з модулем проєкта frontend-utils, який містить наступні абстрактні класи:

- *TokenConstants* – інтерфейс в якому описані константні значення всіх можливих в мові програмування типів токенів.
- *Token* – абстрактний клас, містить всю необхідну інформацію про токен, створений парсером. Має абстрактний метод «*String type2string()*», який реалізується в класах-наслідниках та повертає окреме строкове значення для кожного типу токена.
- *AbstractParser* – парсер мови програмування. Має абстрактний метод «*TokenizationResult parse(String programText)*», який реалізовується в класах-наслідниках та повертає список токенів і статистику по атрибутним характеристикам.

- *Language* – інтерфейс мови програмування. Містить список методів, які необхідно реалізувати окремо для кожної мови програмування. Наприклад, метод, щоб отримати назву мови програмування, розширення файлів, тощо.

Далі необхідно створити новий модуль в рамках Java-проєкту з назвою frontend.<мова програмування>, наслідувати описані вище абстрактні класи й інтерфейси, та описати для нової мови програмування константи токенів, реалізувати абстрактні методи. Як приклад можна розглянути вже реалізовані frontend модулі (рис. 3.3-3.4).

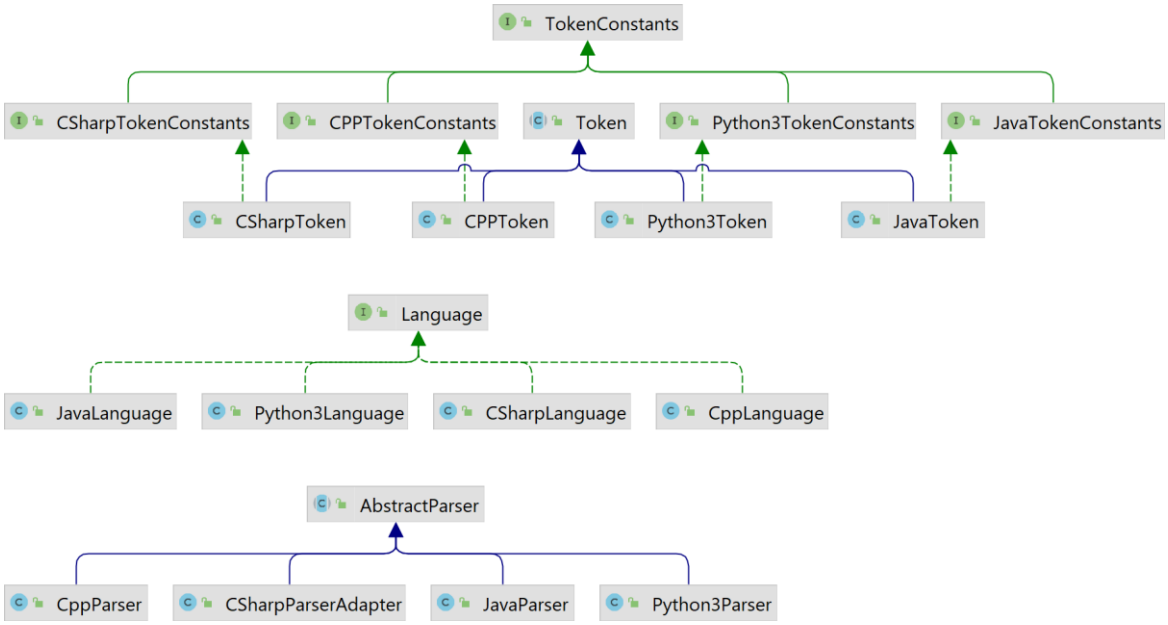


Рис. 3.3. Ієрархія класів, що реалізують підтримку різних мов програмування

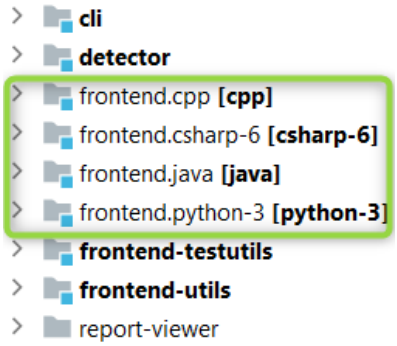


Рис. 3.4. Модулі, що реалізують підтримку різних мов програмування

3.4.2. Використання багатопотоковості

Запропонований комбінований метод включає два етапи оброблення програмного коду та три етапи аналізу його різних частин. Якщо порівнювати з існуючими системами пошуку запозичень (див. підрозділ 1.4), вони виконують лише один етап аналізу. Тобто комбінований метод має значний програш в часі виконання аналізу. Але цього недоліку можна позбутися, або хоча б зменшити час виконання трьох етапів аналізу до мінімуму за рахунок оптимізації виконання методу, а саме використання можливостей багатопотоковості в рамках мови програмування Java. В першу чергу, це можливо, тому що всі три етапи аналізу не залежать один від одного та можуть виконуватися паралельно. З цією метою в рамках бібліотеки було реалізовано підтримку процесу пошуку запозичень в режимі багатопотоковості.

Порядок кроків в режимі багатопотоковості наступний (рис 3.5):

1. При запуску процесу пошуку запозичень створюється контролюючий потік – Control Thread.
2. В потоці Control Thread виконується етап нормалізації. Після його завершення створюється потік Comment-analysis Thread, в рамках якого починається аналіз коментарів програмного коду.
3. Не очікуючи завершення аналізу коментарів контролюючий потік виконує етап токенізації, потім створюється два потоки: Attribute-analysis Thread з процесом аналізу атрибутів коду та Code-structure-analysis Thread з процесом аналізу структури коду.
4. Контролюючий потік очікує завершення трьох решти потоків з процесами аналізу, потім збирає результати і на їх основі формує звіт.

Важливо відмітити, що використання багатопотоковості дає виграш в часі виконання лише в тому випадку, коли середовище виконання програмного забезпечення має досить обчислювальних потужностей (оперативна пам’ять, кількість ядер).

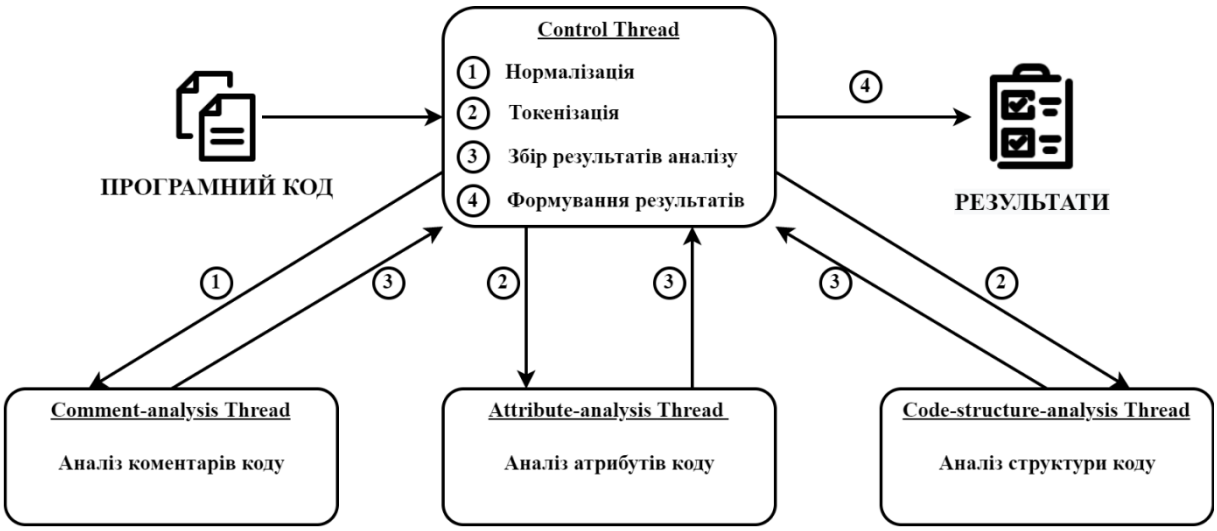


Рис. 3.5. Використання багатопотоковості для оптимізації швидкодії роботи комбінованого методу

3.4.3. Ігнорування шаблонного коду

Для зручності користування системою для пошуку запозичень у програмному коді важливо передбачити можливість виключення з аналізу шаблонного коду, який може бути обов'язковою частиною порівнюваних програм. Дану функціональність було реалізовано в рамках бібліотеки.

Пропонується наступний принцип виключення шаблонного коду з аналізу:

1. Користувач вказує директорію з шаблонним кодом.
2. Бібліотека виконує етапи нормалізації і токенізації для шаблонного коду і запам'ятовує значення атрибутних характеристик, послідовність токенів та послідовність коментарів.
3. Під час етапу нормалізації в виділених коментарях порівнюваних програм видаляються коментарі з шаблонного коду.
4. Під час етапу токенізації в токенізованих рядках порівнюваних програм видаляються послідовність токенів з шаблонного коду.
5. Під час етапу токенізації від значень атрибутних характеристик порівнюваних програм віднімаються значення атрибутних

характеристик шаблонного коду, але тільки для тих програм в яких було знайдено послідовність токенів шаблонного коду.

3.5. Висновки до третього розділу

У даному розділі описано обрані засоби реалізації програмного забезпечення:

1. Тип розробленого програмного забезпечення – бібліотека з можливістю повноцінного її використання за допомогою інтерфейсу командного рядка (CLI).
2. Мова програмування – Java.
3. SQL база даних – H2.
4. Допоміжний інструмент для токенізації коду – ANTLR.

Розглянуто схему взаємодії модулів в розробленому програмному забезпеченні, описано елементи багаторівневої архітектури та структуру бази даних.

Наведено особливості реалізації програмного забезпечення, а саме:

1. Реалізація підтримки нової мови програмування.
2. Використання багатопотоковості для оптимізації часу виконання процесу пошуку запозичень у програмному коді.
3. Ігнорування шаблонного коду.

4. АНАЛІЗ ЕФЕКТИВНОСТІ КОМБІНОВАНОГО МЕТОДУ ПОШУКУ ЗАПОЗИЧЕНЬ У ПРОГРАМНОМУ КОДІ

4.1. Приклад використання реалізованого програмного забезпечення

Програмне забезпечення для пошуку запозичень у програмному коді передбачає два способи взаємодії: за допомогою інтерфейсу командного рядка та використання в програмному коді в якості бібліотеки. Основна функціональність доступна користувачеві зображена на UML-діаграма прецедентів (див. Додаток 1. Функціональність бібліотеки).

4.1.1. Взаємодія через інтерфейс командного рядка

Для використання бібліотеки через інтерфейс командного рядка необхідно, щоб в системі була встановлена JVM 11 версії або вище (рис. 4.1).

```
C:\Users\Maksym_Svynarchuk>java --version
java 11.0.9 2020-10-20 LTS
Java(TM) SE Runtime Environment 18.9 (build 11.0.9+7-LTS)
Java HotSpot(TM) 64-Bit Server VM 18.9 (build 11.0.9+7-LTS, mixed mode)
```

Рис. 4.1. Перевірка версії JVM в командному рядку

Для пошуку запозичень у програмному коді користувач має виконати команду запуску JAR-файлу (рис 4.2) та вказати обов’язкові і додаткові параметри для налаштування, якщо необхідно (табл. 4.1).

```
C:\Users\Maksym_Svynarchuk>java -jar detector.jar -l java -r results -bc templateCode
 -db save -p parallel directoryWithPrograms
```

Рис. 4.2. Приклад команди запуску JAR-файлу

Таблиця 4.1

Приклад вибору міток за алгоритмом просіювання

Аргумент CLI	Допустимі значення	Опис
-l	java, python3, cpp, csharp	Мова програмування порівнюваних програмних структур. *Обов'язковий параметр.*
-r	шлях до директорії на файловій системі	Директорії, в якій будуть збережений звіт з результатами порівняння. *Обов'язковий параметр.*.
-bc	шлях до директорії на файловій системі	Директорія з шаблонним кодом.
-n	додатне число або -1	Максимальна кількість порівнянь, які будуть показані в створеному звіті. Якщо встановлено значення -1, то відображатимуться всі порівняння. Стандартне значення: 30.
-m	0-100	Порогове значення для співпадіння. Усі порівняння для яких хоча б одне значення аналізу вище порогу будуть збережені. Стандартне значення: 0.
-db	save / nosave	Наявність параметра вмикає пошук запозичень у базі програмних кодів. save – зберігати нові порівнювані програми в базу, nosave – не зберігати.
-p	normal / parallel	Режим порівняння, що використовується для порівняння програм. normal – виконання аналізу в одному потоці, parallel – використання багатопотоковості. Стандартне значення: normal.
-kt	два натуральних значення більше 0 через кому	Значення *k, t* для алгоритму просіювання.

Results					
First Program	Second Program	Similarity			Show code
		Comments	Attributes	Structure	
Program A	Program B	0.0%	92.7%	85.6%	compare
Program A	Program C	0.0%	40.3%	50.8%	compare
Program B	Program C	83.3%	70.5%	65.9%	compare

Рис. 4.3. Приклад результатів пошуку запозичень між програмами

INDEX

Program A - module1.c

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
int count_pairs(int n, int *array);
int main()
{
    int size = 0;
    fscanf(stdin, "%i", &size);
    int array[size];
    for (int i = 0; i < size; i++)
    {
        fscanf(stdin, "%i", &array[i]);
    }
    int pairs = count_pairs(size, array);
    fprintf(stdout, "%i", pairs) ;
}
int count_pairs(int n, int *array)
{
    int max = array[1];
    for (int i = 0; i < n; i++)
    {
        if (array[i] > max)
        {
            max = array[i];
        }
    }
    int count_array[max];
    for (int i = 0; i < max; i++)
    {
        count_array[i] = 0;
    }
    for (int i = 0; i < n; i++)
    {
        count_array[array[i]-1]++;
    }
    int pairs = 0;
    int plus = 0;
    for (int i = 0; i < max; i++)
    {
        if (count_array[i] > 1)
        {
            plus = count_array[i] / 2;
            pairs = pairs + plus ;
        }
    }
    return pairs ;
}
```

Program A - module2.c

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include <stdbool.h>
#include <inttypes.h>
#include <ctype.h>
#include <string.h>
#include <progbase.h>
#include <progbase/console.h>
#include <progbase/canvas.h>
#include <assert.h>
int aver (int *arr, int num)
{
    int min=0;
    for(int i=0; i<num; i++)
    {
        if (min<arr[i])
        {
            min=arr[i];
        }
    }
    int count[min+1];
    for(int i=0; i<min+1; i++)
    {
        count[i]=0;
    }
```

Program B - module1.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int maxnum(int *arr, int size)
{
    int max = 0;
    for (int i = 0; i < size; i++)
    {
        if (arr[i] > max)
        {
            max = arr[i];
        }
    }
    int count[max + 1];
    for (int i = 0; i < max + 1; i++)
    {
        count[i] = 0;
    }

    for (int i = 0; i < size; i++)
    {
        count[arr[i]]++;
    }
    int sum = 0;
    int num = 0;
    for (int i = 0; i < max + 1; i++)
    {
        if (count[i] > sum)
        {
            sum = count[i];
            num = i;
        }
        if (count[i] == sum)
        {
            num = i;
        }
    }
    return num;
}
int main()
{
    int N = 0;
    fscanf(stdin, "%i", &N);
    int arr[N];
    for (int i = 0; i < N; i++)
    {
        fscanf(stdin, "%i", &arr[i]);
    }
    int max = maxnum(arr, N);
    fprintf(stdout, "%i", max);
}
```

Program B - module2.c

```
#include <stdio.h>
#include <stdlib.h>

int result(int *arr, int length)
{
    int max = arr[0];
    for (int i = 0; i < length; i++)
    {
        if (arr[i] > max)
        {
            max = arr[i];
        }
    }
    int result[max + 1];
    for (int i = 0; i < max + 1; i++)
    {
        result[i] = 0;
    }
    for (int i = 0; i < length; i++)
    {
        result[arr[i]]++;
    }

    int answer = 0;
    for (int i = 0; i < max + 1; i++)
```

Рис. 4.4. Перегляд запозичених ділянок коду

4.1.2. Використання в програмному коді

Реалізована бібліотека для пошуку запозичень надає аналогічну функціональність для використання в програмному коді. При цьому, налаштування задаються за допомогою об'єктів класів налаштувань (рис. 4.5).

```
PlagOptions options = new PlagOptions("/path/to/rootDir/with/programs", LanguageOption.JAVA);
options.setMaximumNumberOfComparisons(30);
options.setSimilarityThreshold(40);
options.isUsedDatabase(true);
options.databaseMode(DatabaseOption.SAVE);

Plag plag = new Plag(options);
PlagResult result = plag.run();

List<PlagComparison> comparisons = result.getComparisons();

// Optional
File outputDir = new File("/path/to/output");
Report report = new Report(outputDir, options);
report.writeResult(result);
```

Рис. 4.5. Приклад використання бібліотеки в програмному коді

Якщо необхідно, користувач може згенерувати звіт як при використанні бібліотеки через інтерфейс командного рядка, або працювати з отриманими об'єктами класів результатів для більш глибокої інтеграції з програмою, що розробляється (рис. 4.6).

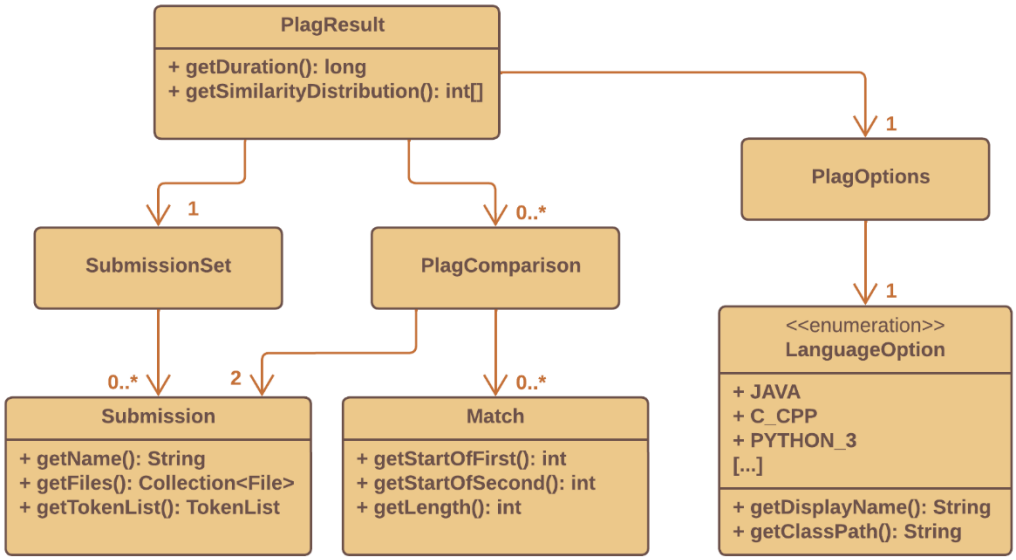


Рис. 4.6. Ієрархія класів для збереження результатів аналізу

4.2. Критерії оцінки та аналіз ефективності запропонованого методу

Оцінювання ефективності реалізованого програмного забезпечення на основі запропонованого комбінованого методу проводилось за наступними критеріями:

1. Повнота пошуку запозичень у програмному коді.
2. Час пошуку запозичень у програмному коді.

Порівняння проводилось між наступними системами:

1. Програмна реалізація запропонованого комбінованого методу.
2. Система MOSS, що побудована на основі структурного методу.
3. Система Accuse, що побудована на основі атрибутного методу.

4.2.1. Повнота пошуку запозичень у програмному коді

В області плагіату програмних кодів повнота пошуку має якісне та кількісне представлення.

Якісно повнота пошуку виражається сукупністю даних в результатах аналізу, на основі яких можна зробити висновок, чи є конкретна програмна структура запозиченою. Комбінований метод має перевагу по даному критерію над іншими аналогами (див. підрозділ 1.4), оскільки він виконує три етапи аналізу різних частин програми (структури коду, атрибутів, коментарів), надає користувачеві відсоток схожості по кожній з частин та можливість для перегляду підозрілого на плагіат коду. Існуючі аналоги, у свою чергу, в результатах вказують лише одне значення відсотку схожості по певній частині коду та дозволяють попарно переглядати ділянки запозичень програмного коду. Отже, запропонований метод надає більшу повноту пошуку для визначення запозичень у програмному коді.

Кількісне подання повноти пошуку є більш конкретним критерієм, який можна виміряти. Будемо вважати, що повнота пошуку запозичень є частиною знайдених релевантних програм (програм які насправді є копіями один одного) у загальному числі релевантних програм [26]:

$$повнота = \frac{|\{релевантні\ документи\} \cap \{знайдені\ документи\}|}{|\{релевантні\ документи\}|} \cdot 100\% \quad (4.1)$$

де $\{релевантні\ документи\}$ – фактичні копії програм;

$\{знайдені\ документи\}$ – програми, які метод визначив як копії;

$|\{знайдені\ документи\}|$ – кількість програм.

Для порівняння було зібрано 4 тестових набори програмних кодів. Кожен з наборів містить 40 унікальних програм та 10 програм, які є копіями з різними модифікаціями:

1. Модифікації характерні для запозичень першого типу.
2. Модифікації характерні для запозичень другого типу.
3. Модифікації характерні для запозичень третього типу.
4. Даний набір містить копії, які були переписані різними способами з однієї програми. Фактично це відповідає четвертому типу запозичень, тобто програмні структури відрізняються настільки, наскільки це можливо в рамках задачі.

Для аналізу було виставлено порогове значення – програмні структури вважаються копіями, якщо відсоток співпадіння більше 50%.

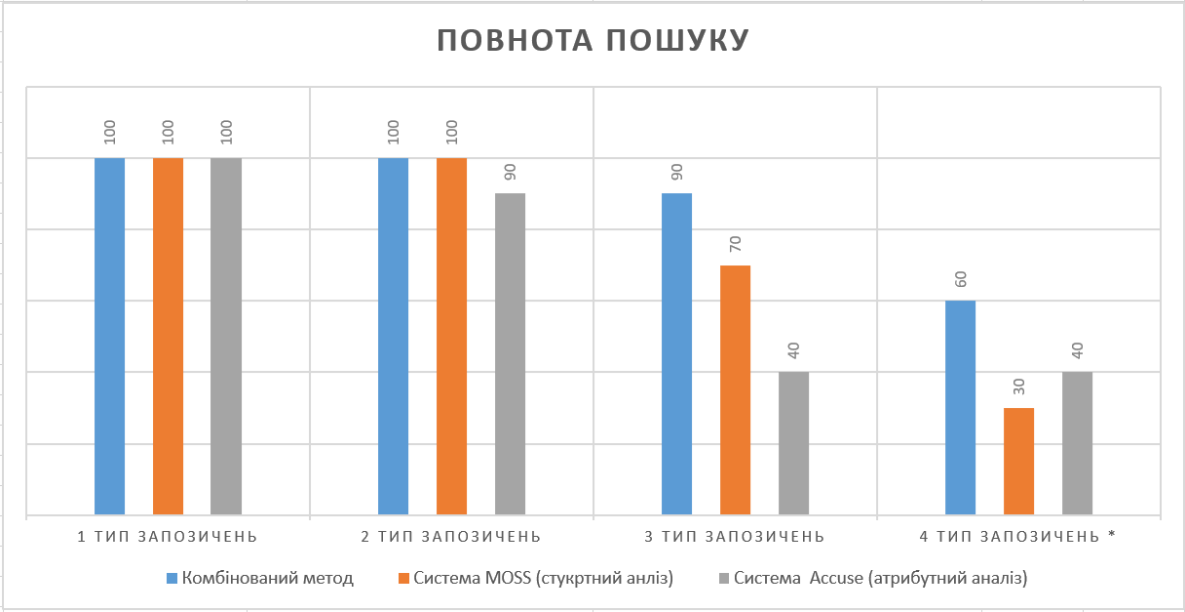


Рис. 4.7. Гістограма порівняння значення повноти пошуку запозичень у програмному коді

За результатами отриманих значень критерію повноти пошуку (рис. 4.7) для порівнюваних систем можна відмітити наступне:

1. Модифікації першого і другого типу запозичень практично повністю визначаються всіма порівнюваними системами.
2. Модифікації третього типу запозичень майже не визначаються системою на основі атрибутів (лише 4 з 10 релевантних програм було визнано копіями), добре аналізуються системою на основі структури коду (7 з 10 копій ідентифіковано). Запропонований метод за рахунок комбінації різних етапів аналізів показав найкращий результат (9 з 10).
3. Модифікації четвертого типу, як і очікувалось, складно визначити будь-якими системами з розглянутих, при цьому комбінований метод знову видав найкращий результат (6 з 10 копій було виявлено).

На основі отриманих результатів порівнянь, можна стверджувати, що запропонований комбінований метод для пошуку запозичень у програмному коді є кращим за критерієм повноти пошуку.

4.2.2. Час пошуку запозичень у програмному коді

Час виконання є важливим критерієм для будь-якої системи пошуку запозичень. Програмна реалізація комбінованого методу підтримує роботу в одно- та багатопотоковому режимі роботи (див. пункт 3.4.2), аналіз ефективності проводилось для кожного них.

Для порівняння часу пошуку запозичень різними системами було підготовлено 6 тестових наборів по 10, 50, 100, 250, 500 та 1000 програм відповідно. Середній розмір кожної програмної структури складає 100-200 рядків коду. Важливо відмітити, що на час роботи систем не впливає наявність чи відсутність копій між програмами.

Аналіз ефективності за критерієм часу пошуку запозичень у програмному коді виконувався на обладнанні з наступними характеристиками:

- операційна система: Windows 10 Enterprise, 64-bit;
- процесор: Intel® Core™ i7-8665U CPU 1.9-2.11 GHz – 4 фізичні ядра, 8 логічних процесорів;
- оперативна пам’ять: 32 GB, 2400MHz.

Таблиця 4.2

Порівняння швидкості пошуку запозичень

Програм	Порівнянь	Комбінований метод	Комбінований метод (режим багатопотоковості)	MOSS	Accuse
10	45	0.14	0.07	0.05	0.04
50	1225	1.05	0.41	0.37	0.32
100	4950	3.00	1.44	1.20	0.81
250	31125	16.05	7.29	5.98	4.75
500	124750	41.42	18.69	15.34	12.18
1000	499500	128.14	56.11	47.67	39.26

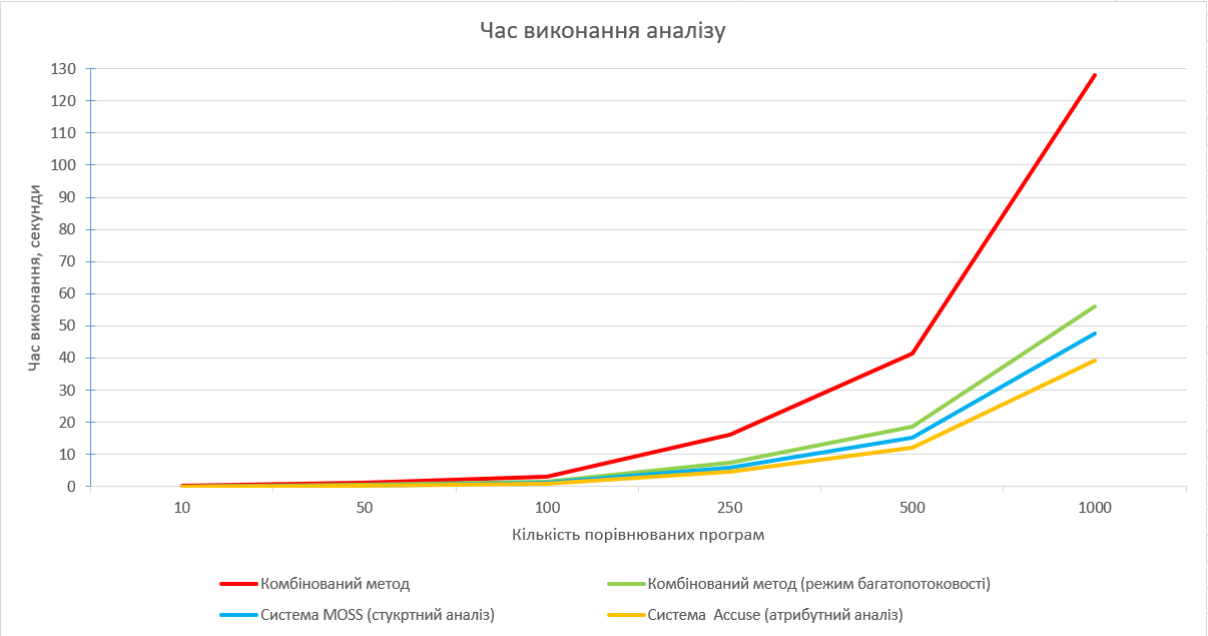


Рис. 4.8. Діаграма порівняння часу пошуку запозичень у програмному коді

За результатами отриманих значень критерію часу пошуку запозичень (табл. 4.2, рис. 4.8) для порівнюваних систем можна відмітити наступне:

1. Аналіз комбінованого методу в однопотоковому режимі виконання займає в середньому в 2.7 рази більше часу, ніж структурний аналіз системою MOSS; та в 3.4 рази більше, ніж атрибутний аналіз системою Accuse.
2. Оптимізація комбінованого методу з використанням можливостей багатопотоковості дозволяє значно зменшити час аналізу. Так в середньому час пошуку запозичень запропонованим методом більший в 1.2 рази, ніж в системі MOSS; та в 1.6 разів, ніж в системі Accuse.

На основі отриманих результатів порівнянь можна стверджувати, що запропонований комбінований метод для пошуку запозичень у програмному коді потребує більше часу для виконання аналізу, ніж аналогічні системи. При цьому використання багатопотоковості дозволяє оптимізувати роботу методу та, за наявності обчислювальних потужностей, максимально наблизити час аналізу до показників конкурентних систем.

4.3. Напрямки подальшої роботи

Напрямки подальшої роботи над вдосконаленням запропонованого комбінованого методу зосереджені на більш глибокому дослідженні використаних підходів для покращення точності пошуку запозичень у програмному коді на різних етапах аналізу.

Для етапу аналізу атрибутів кодів:

1. Проведення досліджень атрибутних характеристик на стійкість до різних типів модифікацій. Розглянутий метод наразі використовує 7 атрибутних характеристик (див. пункт 2.6.1), цей список може бути змінений або розширений. Для цього

необхідно провести емпіричні дослідження на тестових наборах з різними типами модифікацій програмного коду.

2. Проведення досліджень значення констант розміру вікна та коефіцієнту важливості для використаних атрибутних характеристик на залежність від мови програмування. Якщо така залежність буде встановлена, то для підвищення точності атрибутного аналізу є сенс сформувати окремі таблиці значень даних констант.
3. Використання нейронних мереж для класифікації та визначення плагіату на основі стилю програмування. В такому разі необхідно аналізувати зовсім інші атрибути (відступи, регістр літер в назвах змінних, розміщення елементів відкриття / закриття блоків коду, тощо), які можна вважати характеристиками стилю програмування. Загалом тема використання нейронної мережі для класифікації та визначення плагіату програмного коду ще не має практичної реалізації і вимагає більш глибоких досліджень.

Для етапу аналізу структури коду та аналізу коментарів:

1. Оптимізація вибору параметрів k (довжина підрядка для якого рахуються хеш-значення) і t (мінімальна довжина спільного підрядка який точно буде знайдено) для алгоритму просіювання в залежності від мови програмування.
2. Проведення досліджень альтернативних методів вибору послідовності міток документа для підвищення стійкості до модифікацій програмного коду.

4.4. Висновки до четвертого розділу

У даному розділі розглянуто приклади використання розробленого програмного забезпечення:

- за допомогою інтерфейсу командного рядка;
- використання в програмному коді в якості бібліотеки.

Проаналізовано ефективність реалізованого комбінованого методу пошуку запозичень у програмному коді за критеріями повноти пошуку та часу виконання аналізу з системами MOSS (структурний аналіз) та Accuse (атрибутний аналіз). В результаті даного порівняння з'ясовано, що запропонований комбінований метод випереджає інші системи за повнотою пошуку, але вимагає більше часу для аналізу.

Описано напрямки подальшої роботи, які зосереджені в дослідженні алгоритмів, що використовуються на різних етапах аналізу для покращення точності пошуку запозичень у програмному коді.

ВИСНОВКИ

Дана дисертація присвячена підвищенню ефективності виявлення запозичень у програмному коді за критерієм повноти пошуку шляхом розроблення алгоритмічно-програмного методу, який поєднує переваги різних підходів до аналізу запозичень у програмному коді.

В ході дослідження розглянуто основні типи запозичень програмного коду та модифікації, характерні кожному з них. Наведено загальну класифікацію підходів для пошуку програмного плагіату та розглянуто найпопулярніші алгоритми, які їх реалізують. Проаналізувавши підходи для виявлення запозичень у програмному коді, зроблено висновок, що кожний має свої недоліки і переваги.

Для компенсації недоліків існуючих систем прийнято рішення розробити комбінований метод та реалізувати в його рамках три проаналізовані підходи для пошуку запозичень у програмному коді, що дозволить компенсувати їх недоліки. Описано всі етапи запропонованого методу та наведено приклади виконання кожного з них. Окремо розглянуто формат представлення програмної структури для трьох етапів аналізу в рамках комбінованого методу, що дозволяє реалізувати ефективний пошук запозичень програмного коду у базі даних.

Створено програмне забезпечення у вигляді бібліотеки мови програмування Java, яке реалізує запропонований метод та може бути використане як для інтеграції в програмному коді інших проєктів, так і для демонстрації роботи методу за допомогою інтерфейсу командного рядка. В рамках реалізації використано інструменти багатопотоковості для оптимізації часу виконання процесу пошуку запозичень у програмному коді.

Проведено аналіз ефективності запропонованого методу за критеріями повноти пошуку та часу пошуку запозичень у програмному коді. Порівняння з існуючими системами демонструє, що запропонований метод

дозволяє отримати більшу точність повноти пошуку, але вимагає більше часу для аналізу. Втім, оптимізація з використанням багатопотоковості дозволяє наблизити час виконання пошуку комбінованим методом до значень порівнюваних систем.

Подальші напрямки роботи над запропонованим методом зосереджені в дослідженні алгоритмів, що використовуються на різних етапах аналізу для покращення точності пошуку запозичень у програмному коді, а саме: проведення досліджень атрибутних характеристик на стійкість до різних типів модифікацій, проведення досліджень значення констант розміру вікна та коефіцієнту важливості для використаних атрибутних характеристик на залежність від мови програмування, використання нейронних мереж для класифікації та визначення плагіату на основі стилю програмування, оптимізація вибору параметрів k і t для алгоритму просіювання в залежності від мови програмування та проведення досліджень альтернативних методів вибору послідовності міток документа для підвищення стійкості до модифікацій програмного коду.

СПИСОК ВИКОРИСТАНИХ ЛІТЕРАТУРНИХ ДЖЕРЕЛ

1. Chen, X. Shared information and program plagiarism detection [Text] / Xin Chen; Brent Francia; Ming Li, Brian McKinnon, Amit Seker. // IEEE Transactions on Information Theory. – 2004. – Vol. 50, № 7. – P. 1545-1551 – ISSN: 0018-9448.
2. Huang X. A space-efficient algorithm for local similarities [Text] / Huang X., Hardison R.C., Miller W. // Computer Applications in the Biosciences 6. – 1990 – P. 373-381.
3. Желудков А. В. Особенности алгоритмов нечёткого поиска [Текст] / Желудков А. В., Макаров Д. В., Фадеев П. В. // Электронный научно-технический журнал «Инженерный вестник». – грудень 2014 – с. 501-510.
4. Prechelt, L JPlag: Finding plagiarisms among a set of programs [Text] / Lutz Prechelt, Guido Malpohl, Michael Philippsen // Journal of Universal Computer Science. – 2002. – Vol. 8, № 11 – P. 1016-1038 – ISNN: 0948-695X.
5. Li M., Vitanyi P. An introduction to Kolmogorov complexity and its applications. 2nd Ed., Springer, New York. 1997.
6. Li M. The similarity metric. [Text] / Li M., Chen X., Li X., Ma B., Vitanyi P. // Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms. – 2003. – P. 863–872.
7. Schleimer S. Winnowing: local algorithms for document fingerprinting [Text] / Saul David Schleimer, Daniel S. Wilkerson, Alex Aiken // Proceedings of the ACM SIGMOD international conference on Management of data. – 2003. – P. 76-85 – ISBN: 978-1-58113-634-0.
8. Donaldson L. A plagiarism detection system [Text] / Donaldson L., Lancaster A., Sposato P.H. // ACM SIGSCI Bulletin 13(1) – February 1981 – P. 21-25.

9. West A. Copying with plagiarism in Computer Science teaching laboratories [Text] / West A. // Computers in Teaching Conference. – Dublin, 1995.
10. Whale G. Identification of program Similarity in Large Populations [Text] / Whale G. // The Computer Journal, Vol.33. – Number 2, 1990.
11. Grune D., Huntjens M. Het detecteren van kopieen bij informatica-practica [Text] / Dick Grune, Matty Huntjens // Informatie 13. – Nov 1989. – P. 864-867.
12. Michael J. Wise Detection of similarities in student programs: YAP'ing may be prefarable to Plague'ing [Text] / Michael J. Wise // SIDSCI Technical Symposium. – Kansas City, USA, March 5-6, 1992. – P. 268-271.
13. Michael J. Wise String similarity via Greedy Tiling and Running Karp-Rabin Matching [Електронний ресурс]. – Режим доступу: https://www.researchgate.net/publication/262763983_String_Similarity_via_Greedy_String_Tiling_and_Running_Karp-Rabin_Matching
14. Jeffrey F. Mastering Regular Expressions [Text] / Jeffrey Friedl – O'Reilly Media, Inc., August 2006 – ISBN: 9780596528126.
15. Heintze N. Scalable document fingerprinting [Text] / Nevin Heintze // USENIX Workshop on Electronic Commerce – October 1996.
16. Grier S. A Tool that Detects Plagiarism in Pascal Programs [Text] / Sam Grier // Twelfth SIGCSE Technical Symposium, St Louis, Missouri – . 15–20 (February 26-27, 1981) (SIGCSE Bulletin Vol. 13, No. 1, February 1981).
17. Grier S. A Tool that Detects Plagiarism in Pascal Programs [Text] / Sam Grier // Twelfth SIGCSE Technical Symposium, St Louis, Missouri – February 1981. – SIGCSE Bulletin Vol. 13, № 1 – P. 15–20.
18. Verco K. Software for detecting suspected plagiarism: comparing structure and attribute counting systems [Text] / Verco K. and Wise M. // In Proc. First Australian Conf. on Computer Science Education, Sydney Australia – July 3-5 1996. – P. 86-95.

19. Свинарчук М. В., Гадиняк Р. А. Веб-додаток для автоматизації виявлення плагіату в навчальних програмних проєктах: дипломний проєкт бакалавра: 121 Інженерія програмного забезпечення. – Київ, 2020. – 158 с. [Електронний ресурс]. – https://ela.kpi.ua/handle/123456789/35658.
20. Features of Java [Електронний ресурс]. – Режим доступу: https://www.javatpoint.com/features-of-java.
21. H2 Database Engine [Електронний ресурс]. – Режим доступу: https://www.h2database.com/html/main.html.
22. ANTLR [Електронний ресурс]. – Режим доступу: https://www.antlr.org/.
23. Spring Data JPA [Електронний ресурс]. – Режим доступу: https://spring.io/projects/spring-data-jpa
24. Richards M. Software Architecture Patterns [Text] / Mark Richards – O'Reilly Media, Inc. – 2015 – P. 46 – ISBN: 9781491924242.
25. Database Normalization Explained [Електронний ресурс]. – Режим доступу: https://towardsdatascience.com/database-normalization-explained-53e60a494495.
26. Сухий О. Л. АЛГОРИТМИ ПОШУКУ В ІНФОРМАЦІЙНИХ СИСТЕМАХ [Текст] / Сухий О. Л., Міленін В. М., Тарадайнік В. М // Інститут обдарованої дитини НАПН України. – 2015. – с. 13.

ДОДАТКИ

Додаток 1

Копії графічних матеріалів

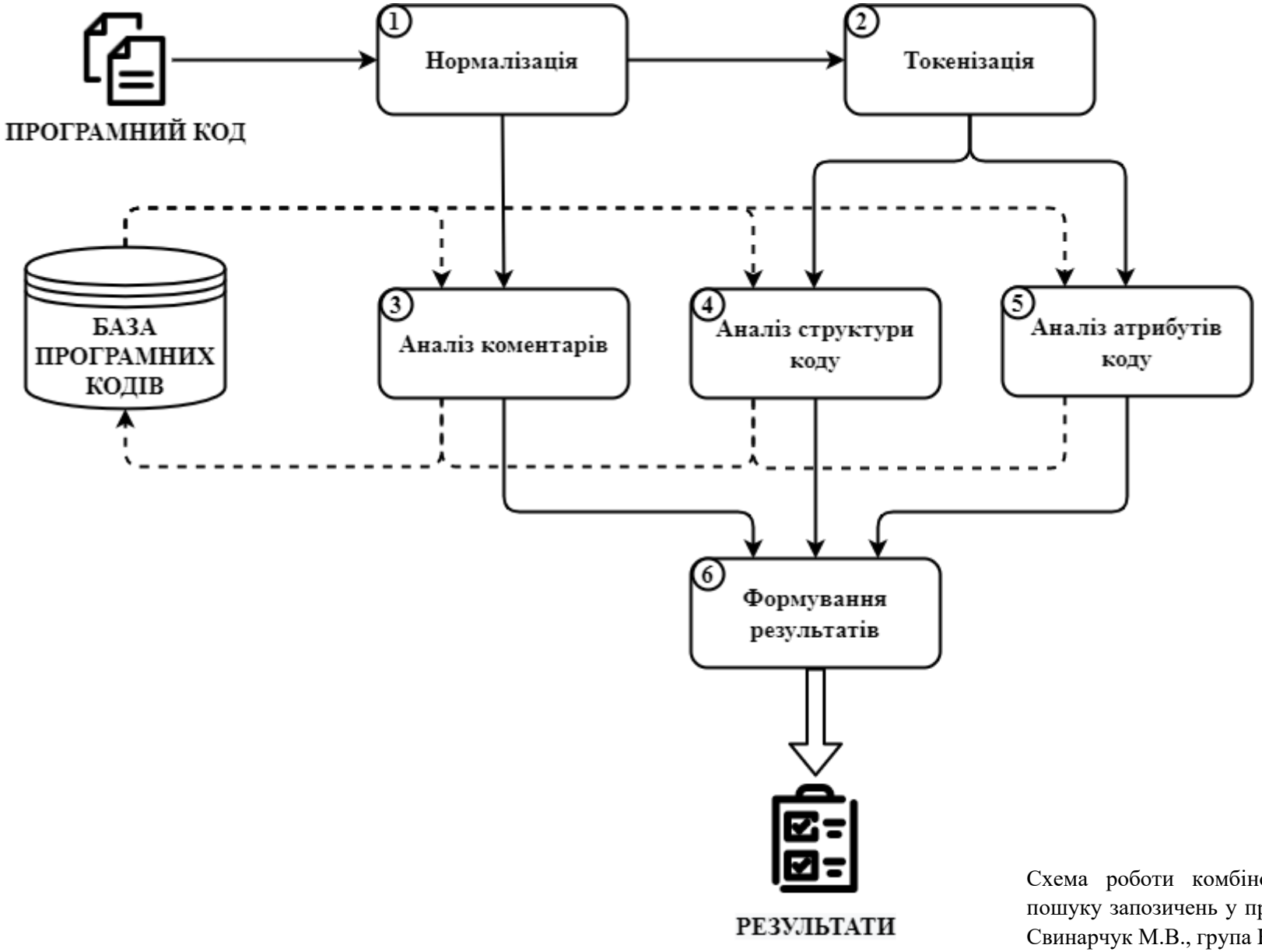
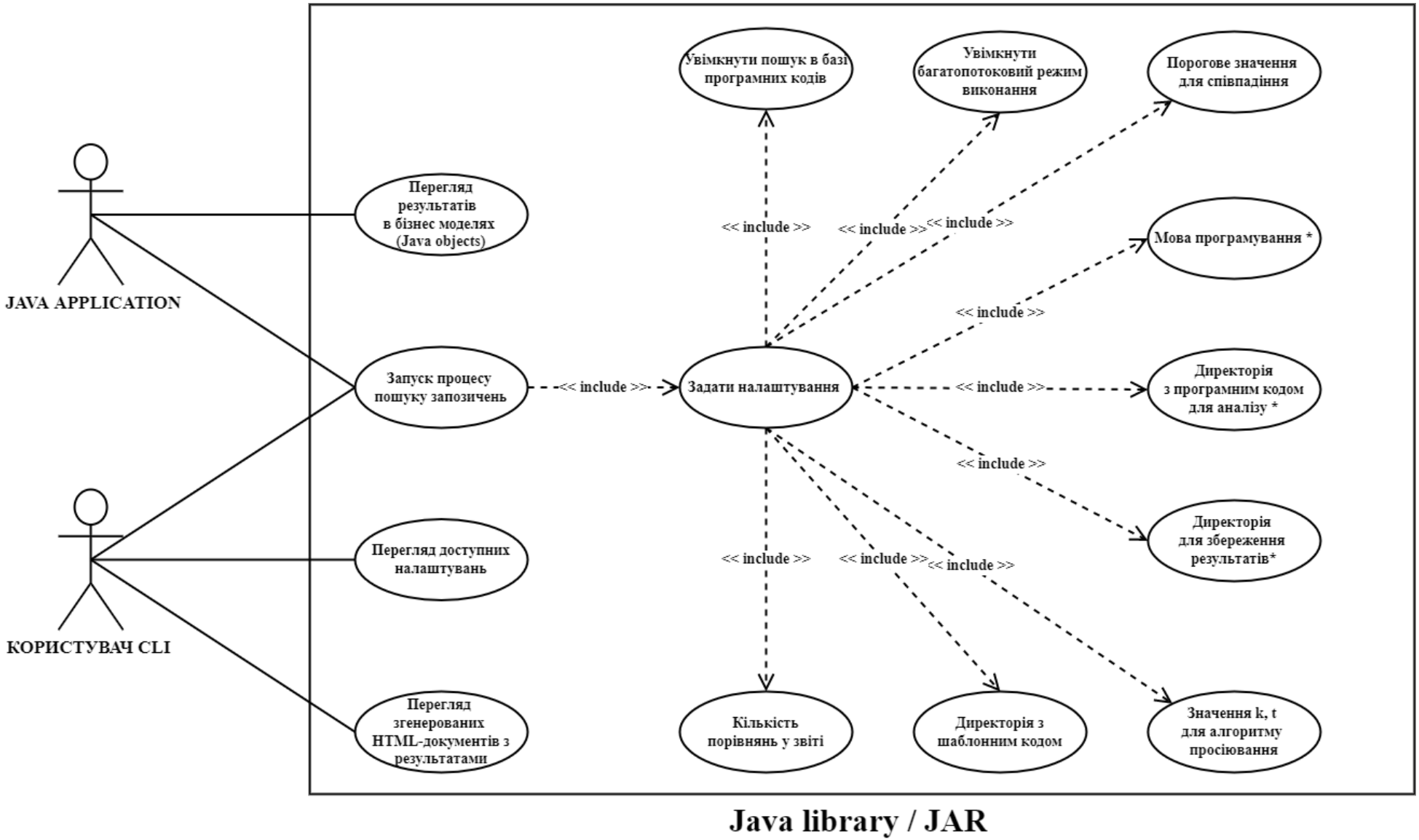
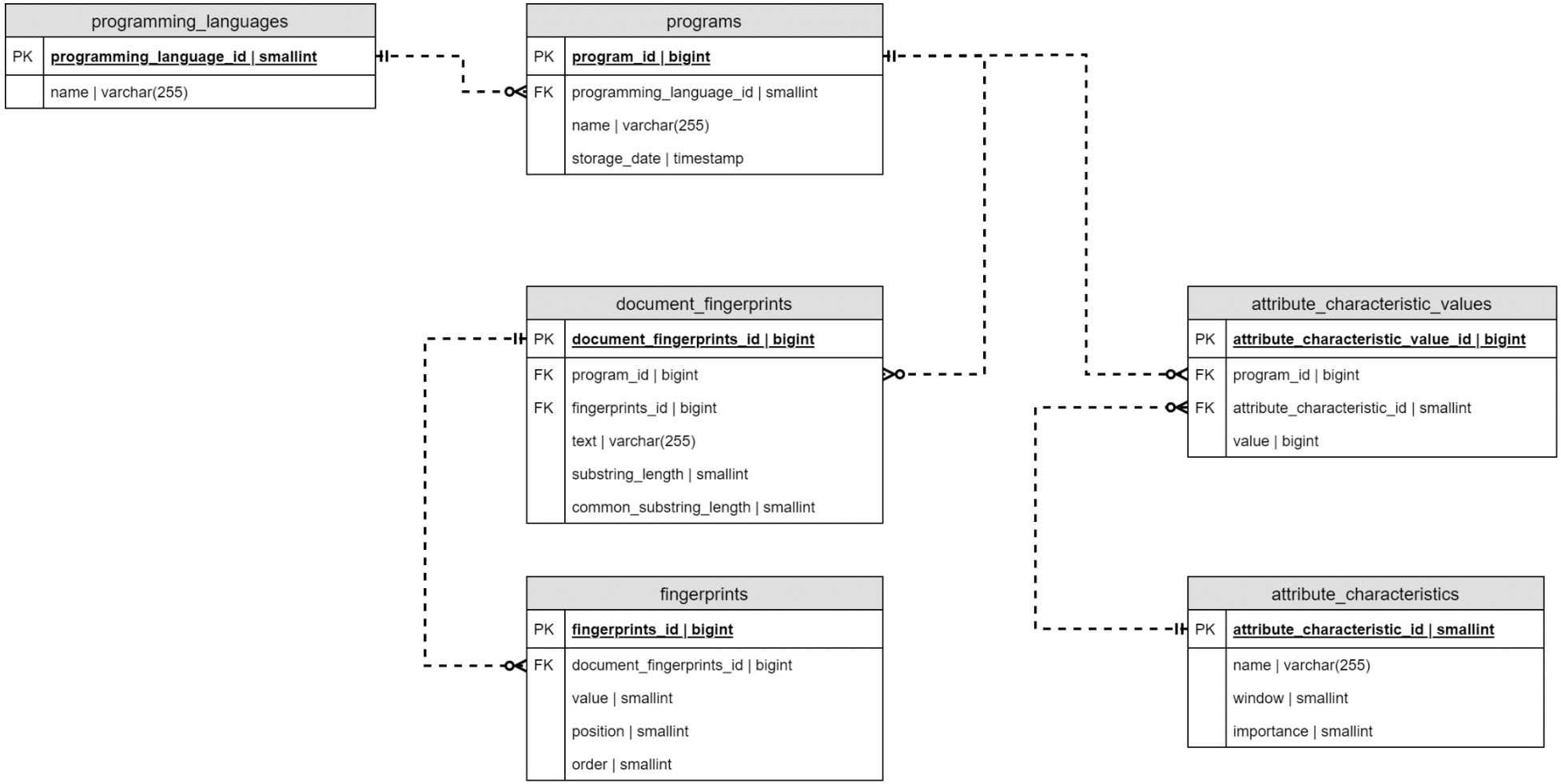


Схема роботи комбінованого методу пошуку запозичень у програмному коді
Свинарчук М.В., група КП-01мн

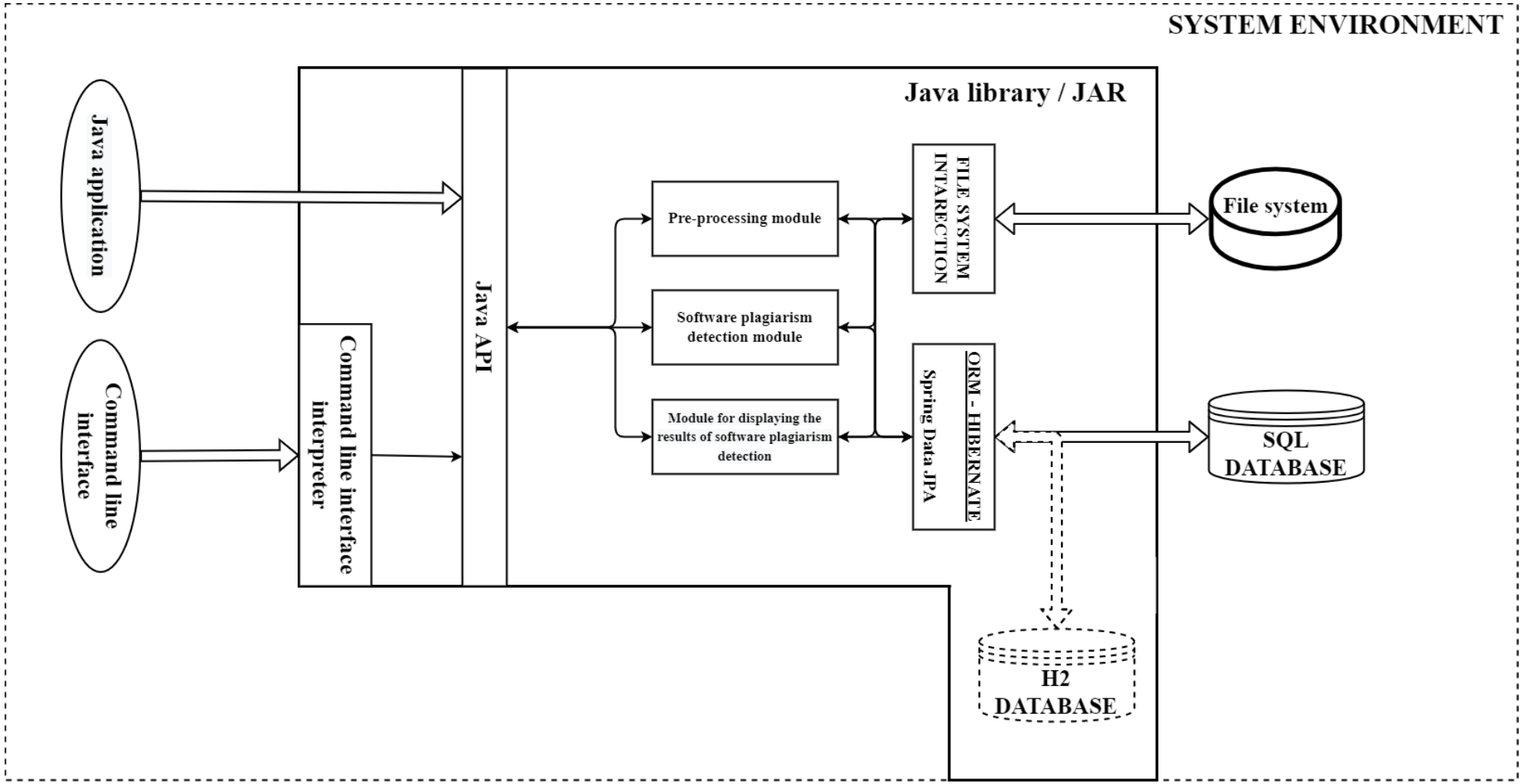


Бібліотека для автоматизованого пошуку запозичень у програмному коді. Функціональність бібліотеки. UML-діаграма прецедентів

Свинарчук М.В., група КП-01мн



Бібліотека для автоматизованого пошуку запозичень у програмному коді. Структура бази даних. ER-діаграма
Свинарчук М.В., група КП-01мн



Структурна схема бібліотеки для автоматизованого пошуку запозичень у програмному коді
Свинарчук М.В., група КП-01мн

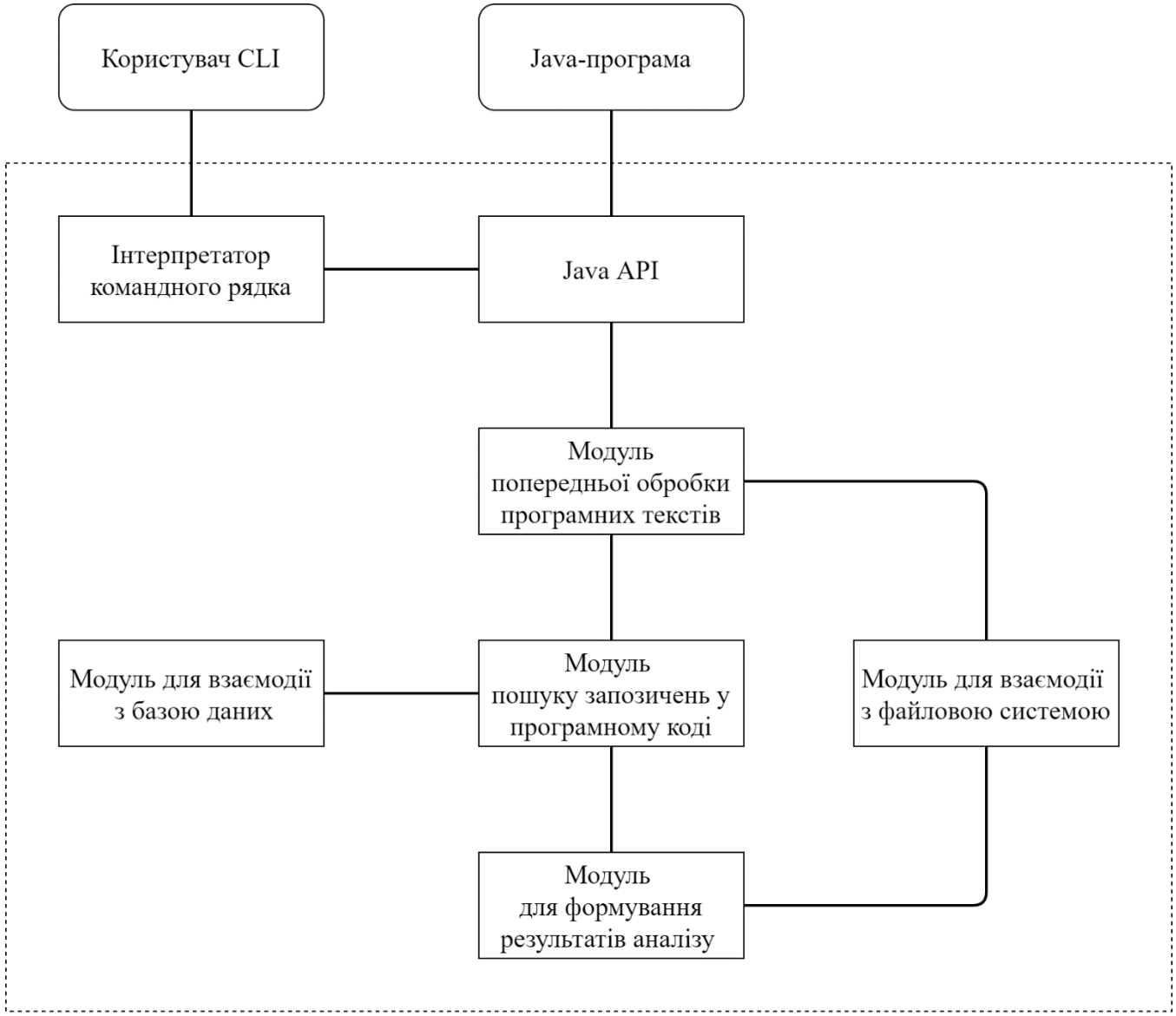


Схема модулів бібліотеки для автоматизованого пошуку запозичень у програмному коді
Свинарчук М.В., група КП-01мн

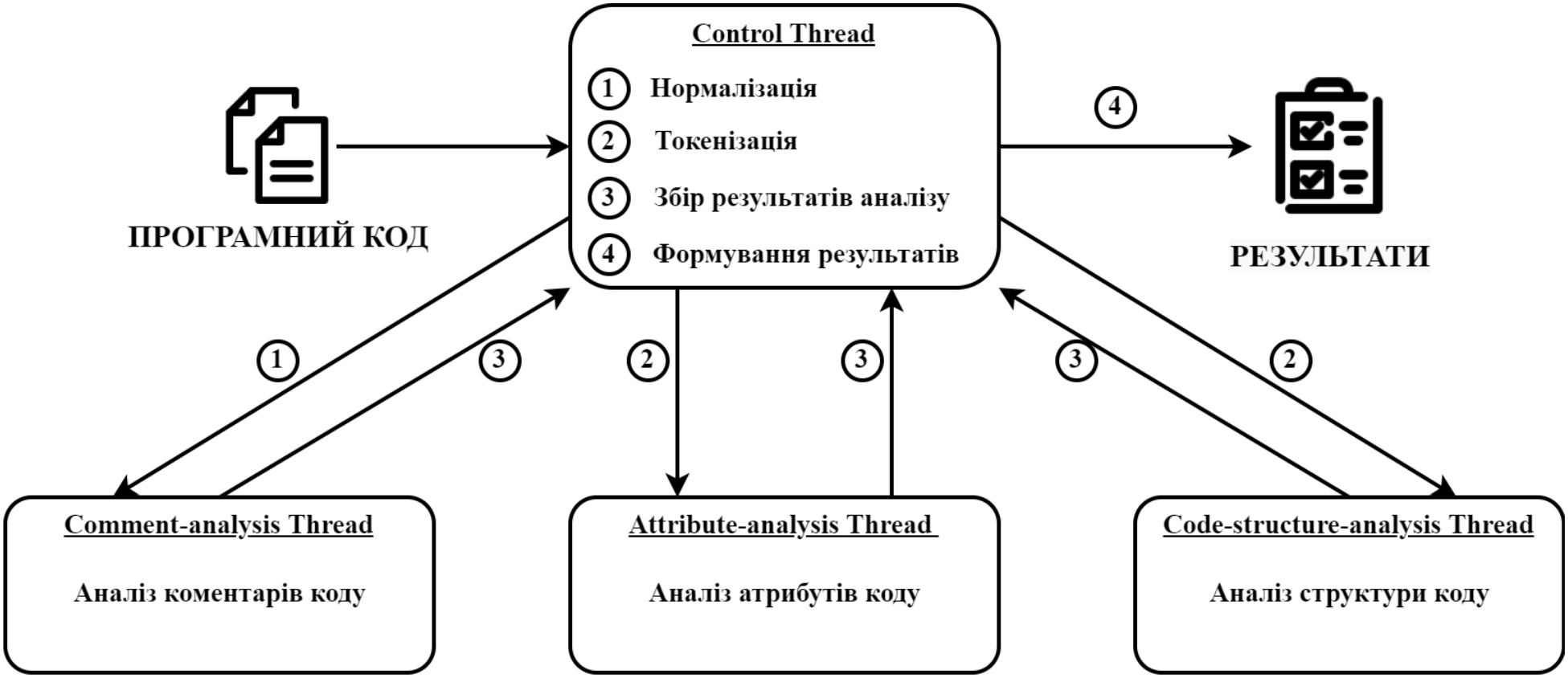


Схема використання багатопоточності для оптимізації швидкодії роботи комбінованого методу в бібліотеці для автоматизованого пошуку запозичень у програмному коді
Свинарчук М.В., група КП-01мн

Додаток 2

Копія презентації

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО”

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

КАФЕДРА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ КОМП’ЮТЕРНИХ СИСТЕМ

АЛГОРИТМІЧНО-ПРОГРАМНИЙ МЕТОД ПОШУКУ ЗАПОЗИЧЕНЬ У ПРОГРАМНОМУ КОДІ

Виконав: студент групи КП-01мн Свинарчук Максим Владиславович

Науковий керівник: к.т.н, доцент Заболотня Тетяна Миколаївна

Київ – 2022

АКТУАЛЬНІСТЬ ДОСЛІДЖЕННЯ

- значне зростання кількості програмного коду;
- запозичення програмного коду в навчальній та комерційній сферах;
- проблема виявлення запозичень програмного коду;
- відсутність комплексного підходу.

ОБ’ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Об’єкт дослідження: процес автоматизованого пошуку запозичень у текстових даних.

Предмет дослідження: методи, алгоритми та програмні засоби автоматизованого пошуку запозичень у програмному коді.

ПОСТАНОВКА ЗАДАЧІ

Мета дослідження: підвищити ефективність виявлення запозичень у програмному коді за критерієм повноти пошуку шляхом розроблення алгоритмічно-програмного методу, який поєднує переваги різних підходів до аналізу запозичень у програмному коді.

1. Дослідити типи запозичень програмного коду та характерні їм модифікації.
2. Класифікувати існуючі методи для пошуку запозичень у програмному коді та проаналізувати алгоритми їх реалізацій.
3. Розробити комбінований метод виявлення запозичень, який передбачає поетапний аналіз програмного коду текстовим, атрибутним та структурним методами для пошуку запозичень.
4. Розробити програмне забезпечення для пошуку запозичень у програмному коді на основі запропонованого комбінованого методу.
5. Проаналізувати ефективність запропонованого комбінованого методу.

ТИПИ ЗАПОЗИЧЕНЬ КОДУ. ПЕРШИЙ ТИП

Модифікації: коментарі, пробільні символи, відступи

```
if (i <= j) {
   x = z + j; // Коментар 1
} else {
   x = z - i; // Коментар 2
}
```

```
if ( i <= j )
{ // Новий Коментар 1
   x=z+j;
}
 else
{ // Новий Коментар 2
   x=z-i;
}
```

ТИПИ ЗАПОЗИЧЕНЬ КОДУ. ДРУГИЙ ТИП

Модифікації: імена ідентифікаторів, літерали, типи змінних, розмітка

```
if (i <= j)
  x = z + j; // Коментар 1
  x = z + 3;
else
  x = z - i; // Коментар 2
```

```
if (t <= k)
{    // Змінений Коментар 1
  a = b + k;
  a = b + 7; // Новий Коментар 3
} else
{    // Змінений Коментар 2
  a = b - t;
}
```

ТИПИ ЗАПОЗИЧЕНЬ КОДУ. ТРЕТІЙ ТИП

Модифікації: додавання і/або видалення операторів та операндів

```
if (i <= j)
  x = z + j;
  x = z + 3;
else
  x = z - i;
```

```
if (t <= k)
{
  a = b + k;
  r = -4;   // Новий Вираз
  a = b + r + 7;
} else
{
   a = b - t;
}
```

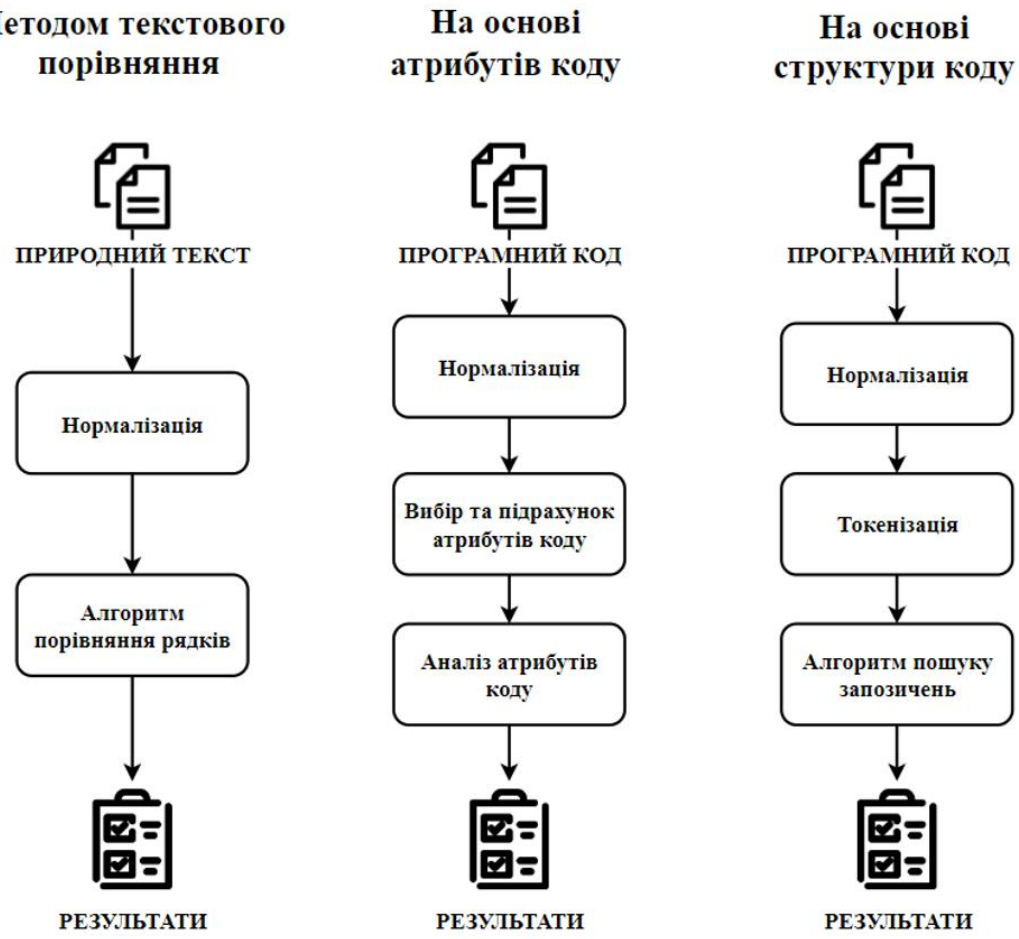
ТИПИ ЗАПОЗИЧЕНЬ КОДУ. ЧЕТВЕРТИЙ ТИП

Модифікації: два фрагменти коду синтаксично реалізовані по різному, але виконують однакові дії.

```
int res, i=1;
for (res=1; i<=VALUE; i++){
   res=res*i;
}
```

```
int factorial(int n) {
   if (n == 0)
      return 1;
   else
      return n * factorial(n-1);
}
```

КЛАСИФІКАЦІЯ ІСНУЮЧИХ ПІДХОДІВ



ІСНУЮЧІ СИСТЕМИ ТА АЛГОРИТМИ ПОШУКУ ЗАПОЗИЧЕНЬ У ПРОГРАМНОМУ КОДІ

Система	Метод	Модель представлення	Алгоритм
Accuse	атрибутний	характеристика атрибутів	порівняння атрибутів (функція кореляційної оцінки)
YAP	структурний	токени	порівняння токенів, алгоритм Хескеля
JPlag	структурний	токени	алгоритм жадібного рядкового замощення з модифікацією алгоритмом Рабіна-Карпа
MOSS	структурний	мітки документа	алгоритм просіювання (реалізація алгоритму відбитків)
SID	структурний	токени	метрика, заснована на Колмогоровій складності
SIM	структурний	токени	алгоритм локального вирівнювання рядків

ЗАПРОПОНОВАНИЙ КОМБІНОВАНИЙ МЕТОД

1. Нормалізація програмного коду.
2. Токенізація програмного коду.

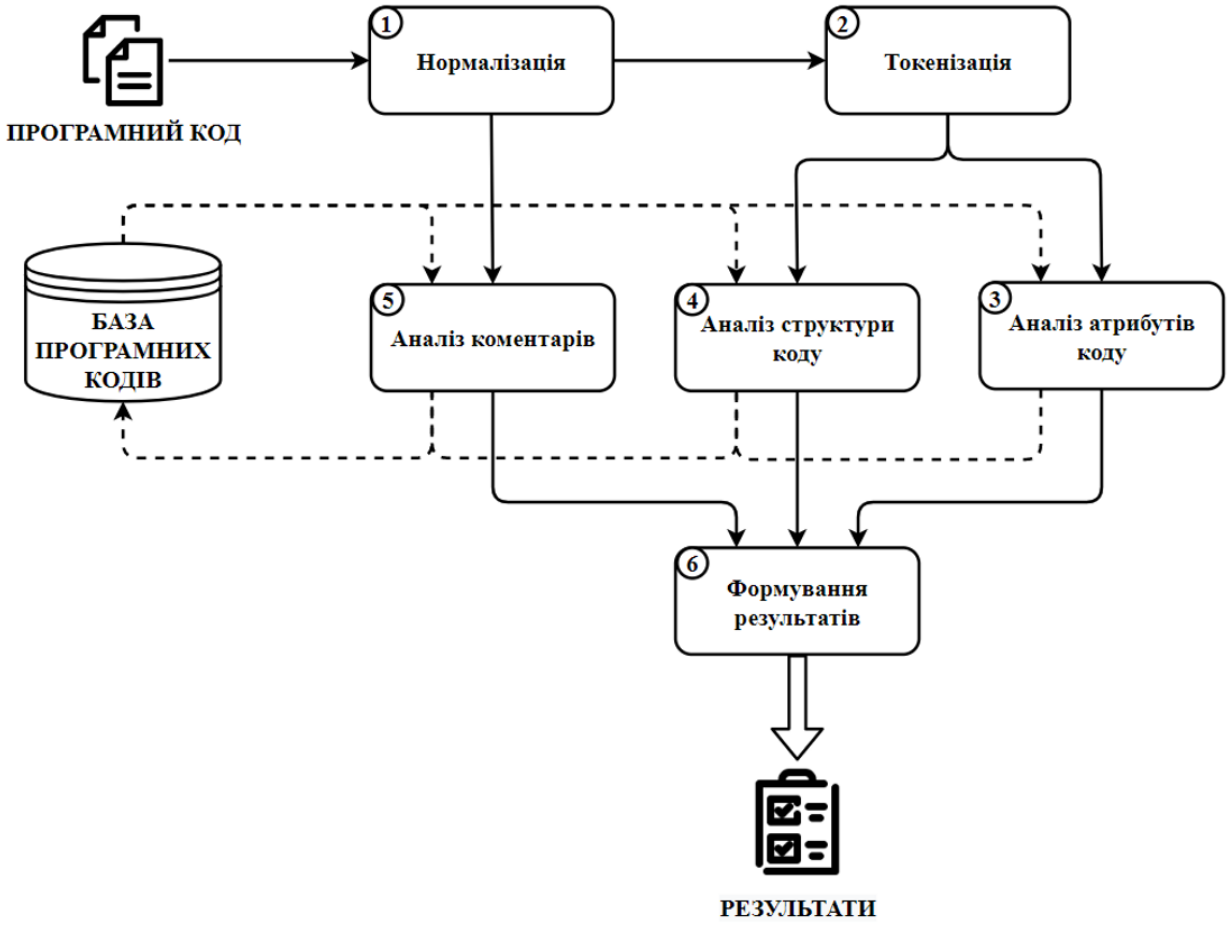
} Етапи обробки

3. Аналіз атрибутів коду.
4. Аналіз структури коду.
5. Аналіз коментарів.

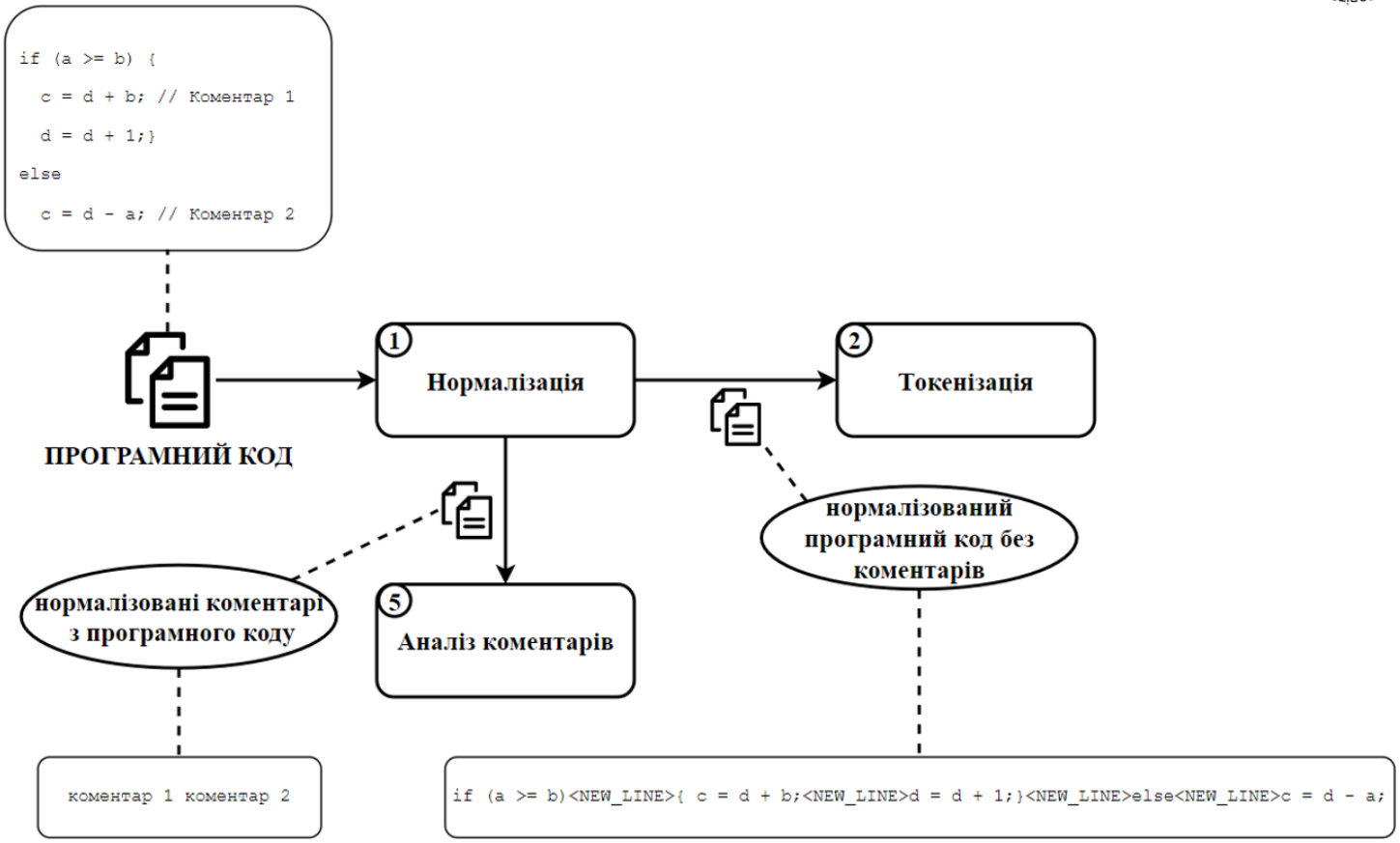
} Етапи аналізу

6. Формування результатів.

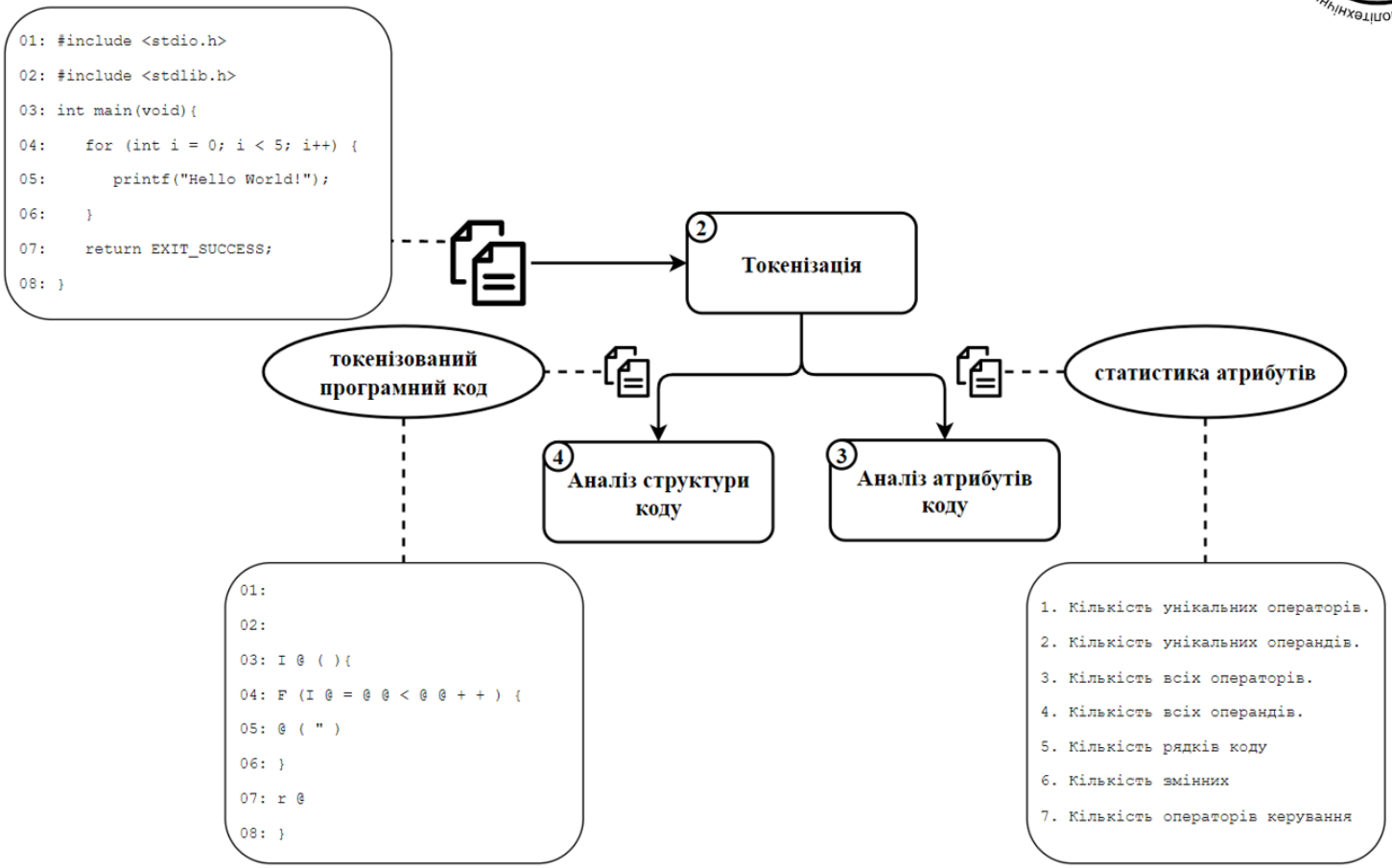
СХЕМА РОБОТИ КОМБІНОВАНОГО МЕТОДУ



ЕТАП 1. НОРМАЛІЗАЦІЯ



ЕТАП 2. ТОКЕНІЗАЦІЯ



ЕТАП 3. АНАЛІЗ АТРИБУТІВ КОДУ

1. Кількість унікальних операторів.
2. Кількість унікальних операндів.
3. Кількість всіх операторів*.
4. Кількість всіх операндів*.
5. Кількість рядків коду*.
6. Кількість змінних.
7. Кількість операторів керування.

! при підрахунку *Кількості всіх операторів* не рахувати оператори присвоєння;

! для кожного знайденого оператора присвоєння віднімати 2 операнди від значення *Кількість всіх операндів*, зменшити *Кількість рядків коду* на 1.

ЕТАП 3. АНАЛІЗ АТРИБУТІВ КОДУ

$$sim(A, B) = \frac{CORRELATION_{AB}}{MAX\_CORRELATION} \cdot 100\%$$

1. $CORRELATION_{AB} = 0$.
2. Для кожної з характеристик атрибутів: $diff_i = |A_i - B_i|$.
3. Якщо $diff_i \le window_i$, то $CORRELATION_{AB} \mathrel{+}= importance_i - diff_i$.

№	Характеристика коду	Розміру вікна *window*	Коефіцієнту важливості *importance*
1	Кількість унікальних операторів	5	5
2	Кількість унікальних операндів	5	5
3	Кількість всіх операторів	3	7
4	Кількість всіх операндів	3	7
5	Кількість рядків коду	3	6
6	Кількість змінних	2	4
7	Кількість операторів керування	1	3
$MAX\_CORRELATION = \sum_{i=1}^{7} importance_i = 37$			

ЕТАП 3. АНАЛІЗ АТРИБУТІВ КОДУ

1. $diff_1 = |25 - 24| = 1 \leq 5$: $CORRELATION_{AB}^1 = importance_1 - diff_1 = 5 - 1 = 4$.
2. $diff_2 = |13 - 12| = 1 \leq 5$: $CORRELATION_{AB}^2 = CORRELATION_{AB}^1 + importance_2 - diff_2 = 4 + 5 - 1 = 8$
3. $diff_3 = |118 - 110| = 8 > 3$: $CORRELATION_{AB}^3 = CORRELATION_{AB}^2$
4. $diff_4 = |106 - 90| = 16 > 3$: $CORRELATION_{AB}^4 = CORRELATION_{AB}^3$
5. $diff_5 = |63 - 64| = 1 \leq 3$: $CORRELATION_{AB}^5 = CORRELATION_{AB}^4 + importance_5 - diff_5 = 8 + 6 - 1 = 13$.
6. $diff_6 = |11 - 11| = 0 \leq 2$:$CORRELATION_{AB}^6 = CORRELATION_{AB}^5 + importance_6 - diff_6 = 13 + 4 - 0 = 17$.
7. $diff_7 = |4 - 4| = 0 \leq 1$: $CORRELATION_{AB}^7 = CORRELATION_{AB}^6 + importance_7 - diff_7 = 17 + 3 - 0 = 20$.

$$sim(A, B) = \frac{CORRELATION_{AB}}{MAX\_CORRELATION} \cdot 100\% = \frac{20}{37} \cdot 100\% \approx 54.05\%$$

№	Характеристика коду	Програма *A*	Програма *B*
1	Кількість унікальних операторів	25	24
2	Кількість унікальних операндів	13	12
3	Кількість всіх операторів	118	110
4	Кількість всіх операндів	106	90
5	Кількість рядків коду	63	64
6	Кількість змінних	11	11
7	Кількість операторів керування	4	4

ЕТАП 4. СТРУКТУРНИЙ АНАЛІЗ КОДУ АЛГОРИТМ ВІДБИТКІВ ДОКУМЕНТА

1. Хешуються підрядки довжиною k токенізованих програм A і B.
2. Для кожної програми обирається послідовність хеш-значень (міток) за алгоритмом просіювання.
3. Отримані послідовності міток програм A і B використовуються для обчислення функції подібності:

$$sim(A, B) = \frac{|S(A) \cap S(B)|}{|S(A) \cup S(B)|} \cdot 100\% ,$$

де $S(A)$ – послідовність міток програми A;

$|A|$ – розмір множини A.

ЕТАП 4. СТРУКТУРНИЙ АНАЛІЗ КОДУ. АЛГОРИТМ ПРОСІЮВАННЯ

Початковий рядок:	A do run run run, a do run run
Нормалізований рядок:	adorunrunrunadorunrun
Обираємо значення *k*, *k* ≤ *t*	t = 8, k = 5
Отримуємо послідовність 5-грам	adoru dorun orunr runru unrun nrunr runru unrun nruna runad unado nador adoru dorun orunr runru unrun
Гіпотетична послідовність хеш-значень для 5-грам	77 74 42 17 98 50 17 98 8 88 67 39 77 74 42 17 98
Вікна хеш-значень з довжиною 4 (*w = t – k + 1*)	(77, 74, 42, **17**) (74, 42, 17, 98) (42, 17, 98, 50) (17, 98, 50, **17**) (98, 50, 17, 98) (50, 17, 98, **8**) (17, 98, 8, 88) (98, 8, 88, 67) (8, 88, 67, 39) (88, 67, **39**, 77) (67, 39, 77, 74) (39, 77, 74, 42) (77, 74, 42, **17**) (74, 42, 17, 98)
Мітки обрані алгоритмом просіювання	17 17 8 39 17
*** Мітки згруповані з номером позиції в послідовності хеш-значень для 5-грам**	[17,3] [17,6] [8,8] [39,11] [17,15]

ЕТАП 5. АНАЛІЗ КОМЕНТАРІВ

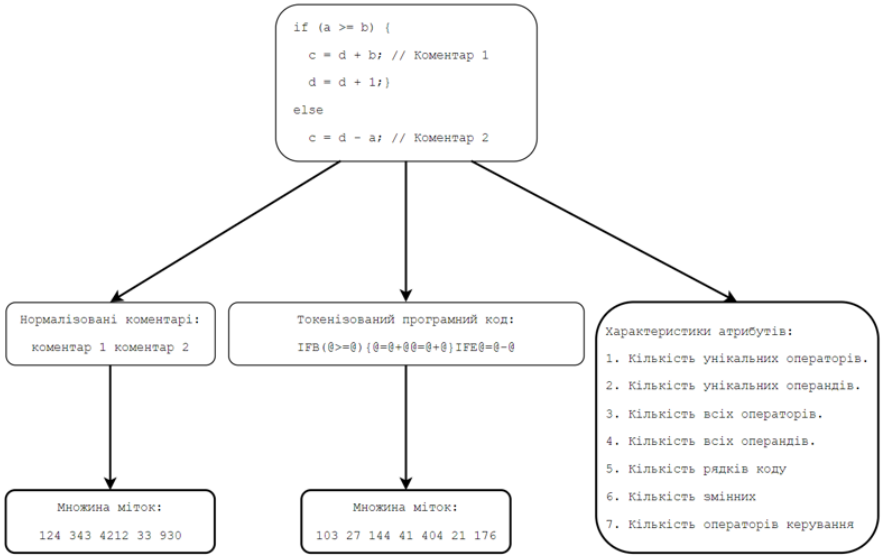
- коментарі виділяються з програмного коду за допомогою регулярних виразів;
- використовується алгоритм відбитків документа з алгоритмом просіювання;
- високий відсоток схожості свідчить про наявність плагіату в програмах.

ЕТАП 6. ФОРМУВАННЯ РЕЗУЛЬТАТІВ

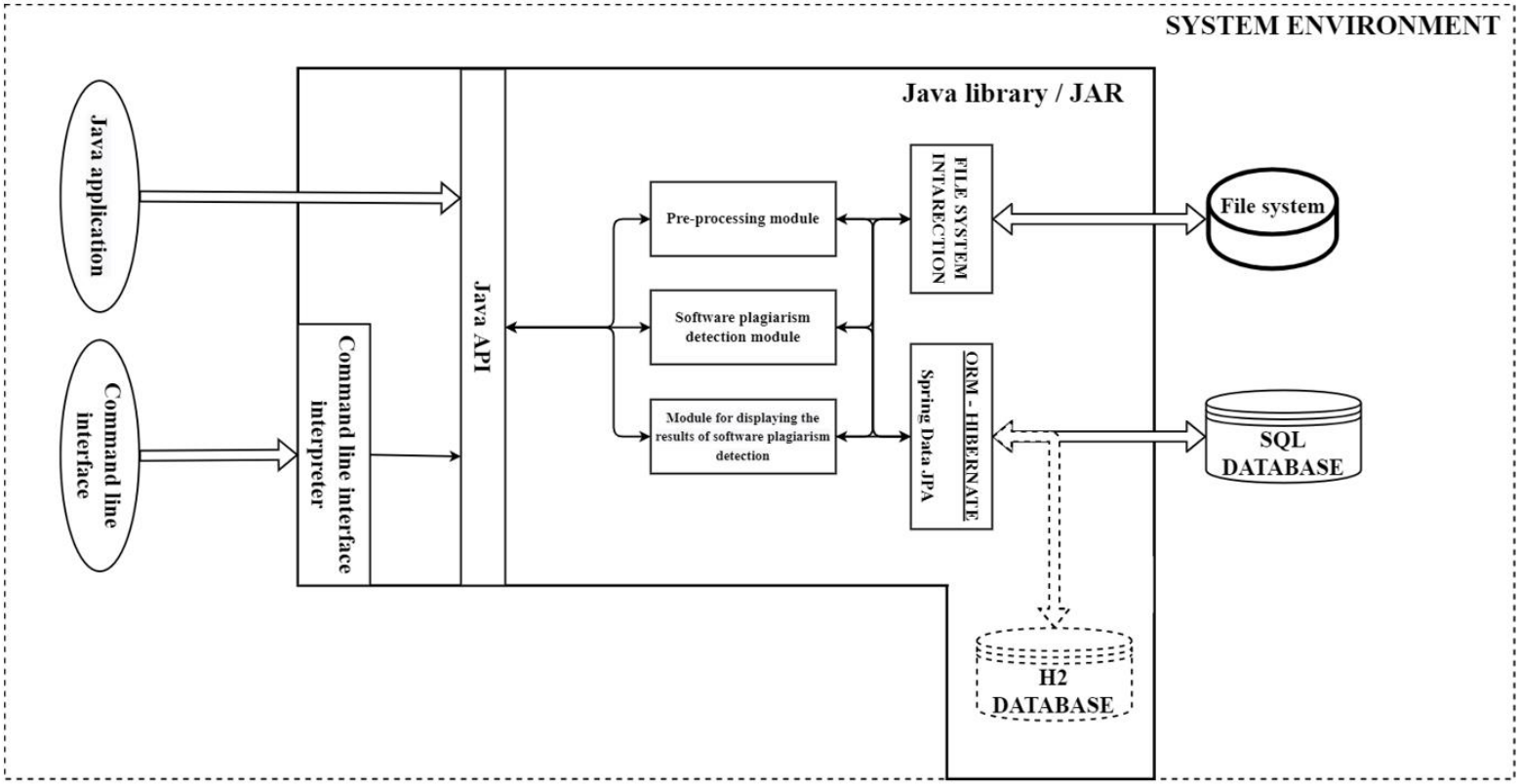
- отримання відсотку схожості між програмами по кожному з етапів аналізу окремо (коментарі, атрибути коду, структурний аналіз);
- задання порогових значень для процентного збігу програмних текстів для кожного з етапів аналізу;
- виділення ділянок програмного коду, підозрілих в запозиченні.

ФОРМАТ ПРЕДСТАВЛЕННЯ ПРОГРАМНОЇ СТРУКТУРИ

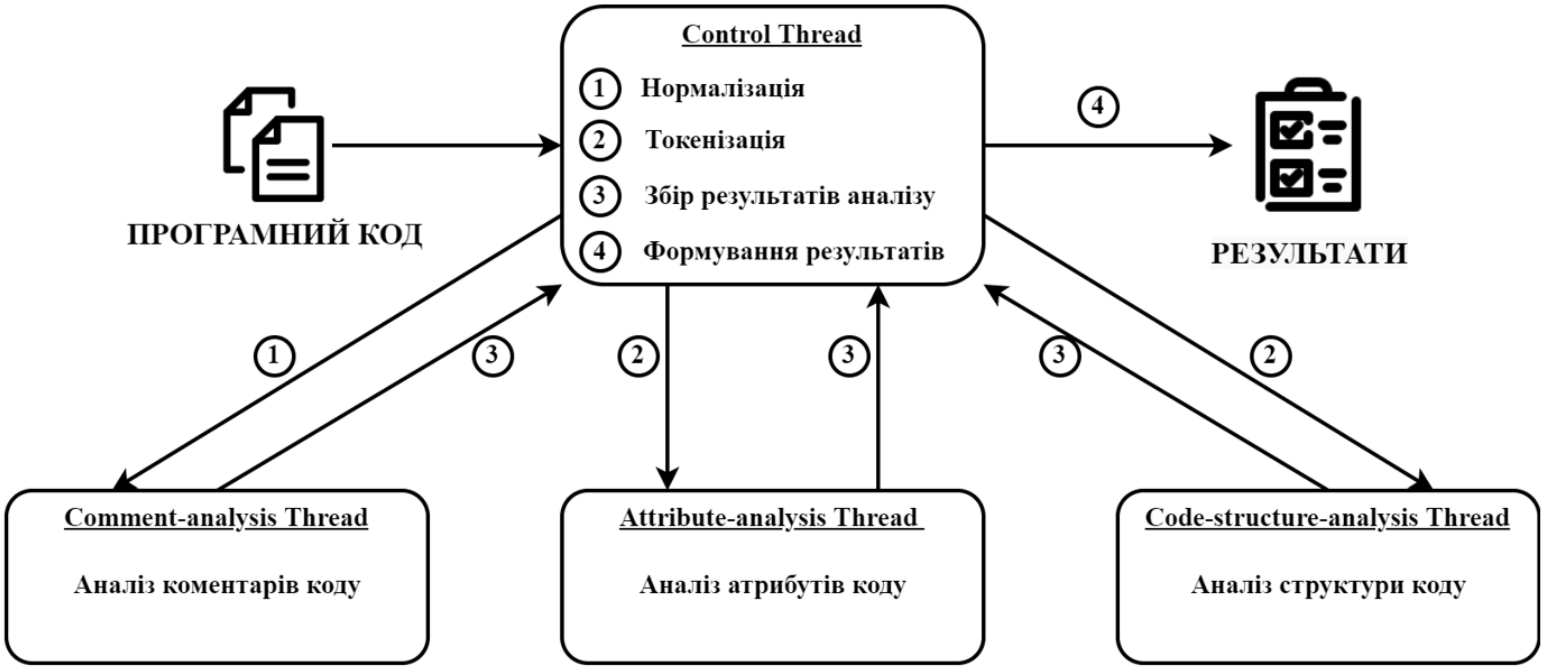
- **аналіз атрибутів коду** – кількісні характеристики атрибутів;
- **аналіз структури коду** – послідовність міток для токенізованого програмного коду;
- **аналіз коментарів** – послідовність міток для коментарів.



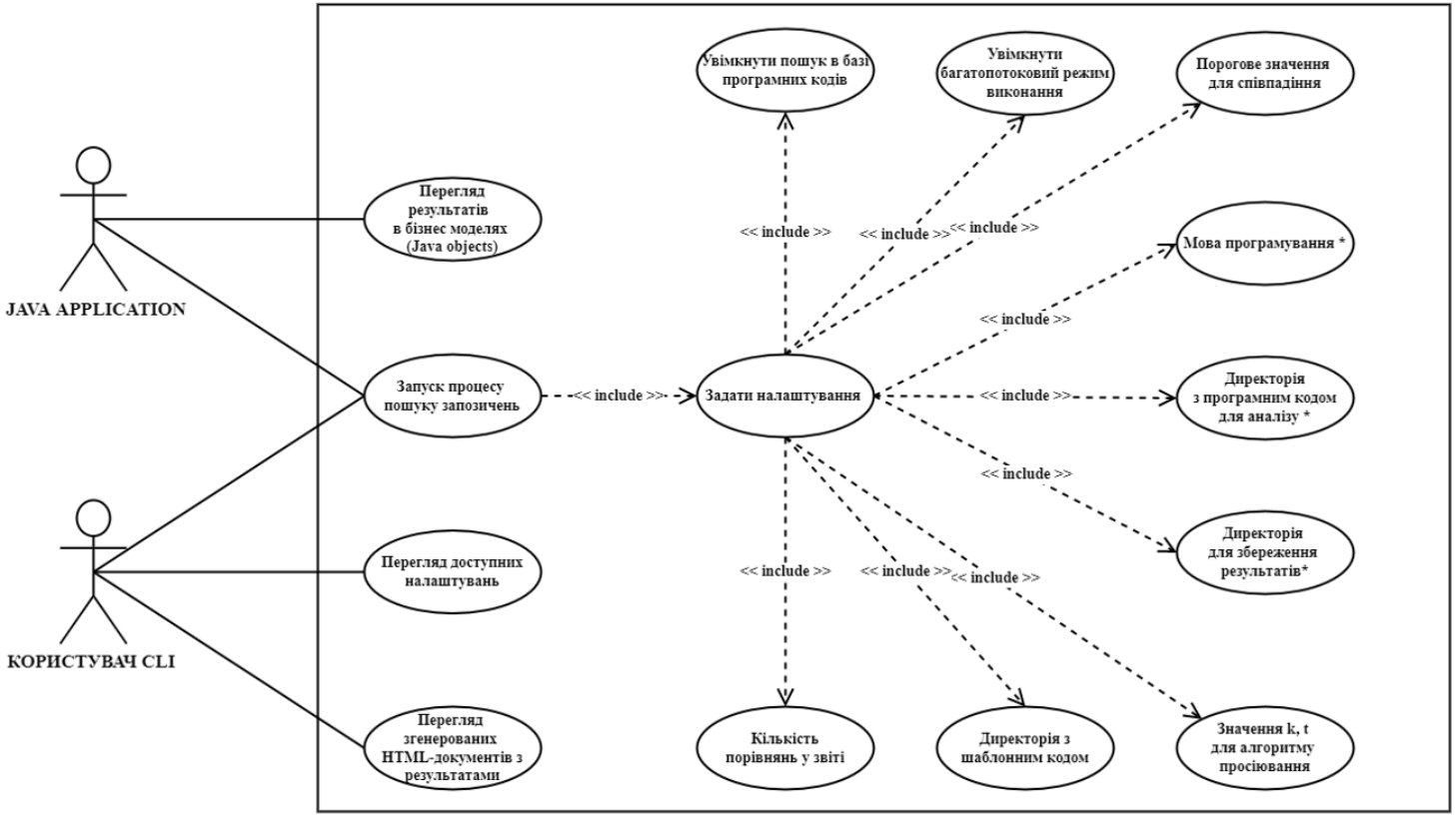
ПРОГРАМНА РЕАЛІЗАЦІЯ. АРХІТЕКТУРА



ПРОГРАМНА РЕАЛІЗАЦІЯ. ВИКОРИСТАННЯ БАГАТОПОТОКОВОСТІ



ПРОГРАМНА РЕАЛІЗАЦІЯ. ФУНКЦІОНАЛЬНІСТЬ БІБЛІОТЕКИ



ПРИКЛАД РОБОТИ БІБЛІОТЕКИ. ІНТЕРФЕЙС КОМАНДНОГО РЯДКА


```
C:\Users\Maksym_Svynarchuk>java -jar detector.jar -l java -r results -bc templateCode
 -db save -p parallel directoryWithPrograms
```

INDEX

Program A - module1.c

```c
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
int count_pairs(int n, int *array);
int main()
{
    int size = 0;
    fscanf(stdin, "%i", &size);
    int array[size];
    for (int i = 0; i < size; i++)
    {
        fscanf(stdin, "%i", &array[i]);
    }
    int pairs = count_pairs(size, array);
    fprintf(stdout, "%i", pairs) ;
}
int count_pairs(int n, int *array)
{
    int max = array[1];
    for (int i = 0; i < n; i++)
    {
        if (array[i] > max)
        {
            max = array[i];
        }
    }
    int count_array[max];
    for (int i = 0; i < max; i++)
    {
        count_array[i] = 0;
    }
    for (int i = 0; i < n; i++)
    {
        count_array[array[i]-1]++;
    }
    int pairs = 0;
    int plus = 0;
    for (int i = 0; i < max; i++)
    {
        if (count_array[i] > 1)
        {
            plus = count_array[i] / 2;
            pairs = pairs + plus ;
        }
    }
    return pairs ;
}
```

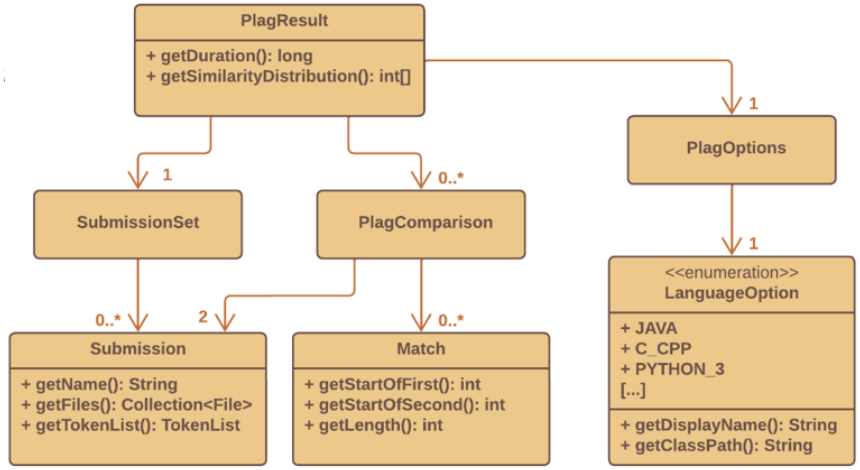
Program B - module1.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int maxnum(int *arr, int size)
{
    int max = 0;
    for (int i = 0; i < size; i++)
    {
        if (arr[i] > max)
        {
            max = arr[i];
        }
    }
    int count[max + 1];
    for (int i = 0; i < max + 1; i++)
    {
        count[i] = 0;
    }

    for (int i = 0; i < size; i++)
    {
        count[arr[i]]++;
    }
    int sum = 0;
    int num = 0;
    for (int i = 0; i < max + 1; i++)
    {
        if (count[i] > sum)
        {
            sum = count[i];
            num = i;
        }
        if (count[i] == sum)
        {
            num = i;
        }
    }
    return num;
}
int main()
{
    int N = 0;
    fscanf(stdin, "%i", &N);
    int arr[N];
    for (int i = 0; i < N; i++)
    {
        fscanf(stdin, "%i", &arr[i]);
    }
    int max = maxnum(arr, N);
    fprintf(stdout, "%i", max);
```

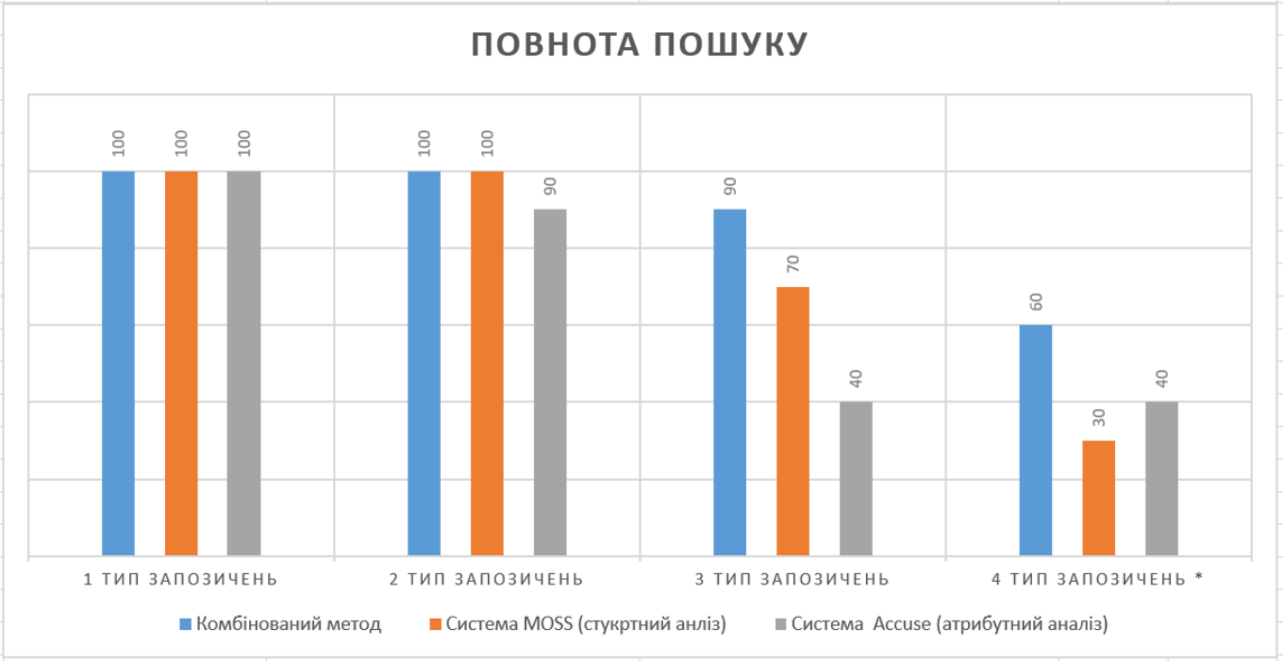
Results					
First Program	Second Program	Similarity			Show code
		Comments	Attributes	Structure	
Program A	Program B	0.0%	92.7%	85.6%	compare
Program A	Program C	0.0%	40.3%	50.8%	compare
Program B	Program C	83.3%	70.5%	65.9%	compare

ПРИКЛАД РОБОТИ БІБЛІОТЕКИ. ВИКОРИСТАННЯ В ПРОГРАМНОМУ КОДІ


```
PlagOptions options = new PlagOptions("/path/to/rootDir/with/programs", LanguageOption.JAVA);
options.setMaximumNumberOfComparisons(30);
options.setSimilarityThreshold(40);
options.isUsedDatabase(true);
options.databaseMode(DatabaseOption.SAVE);

Plag plag = new Plag(options);
PlagResult result = plag.run();

List<PlagComparison> comparisons = result.getComparisons();

// Optional
File outputDir = new File("/path/to/output");
Report report = new Report(outputDir, options);
report.writeResult(result);
```



ЕФЕКТИВНІСТЬ. ПОВНОТА ПОШУКУ



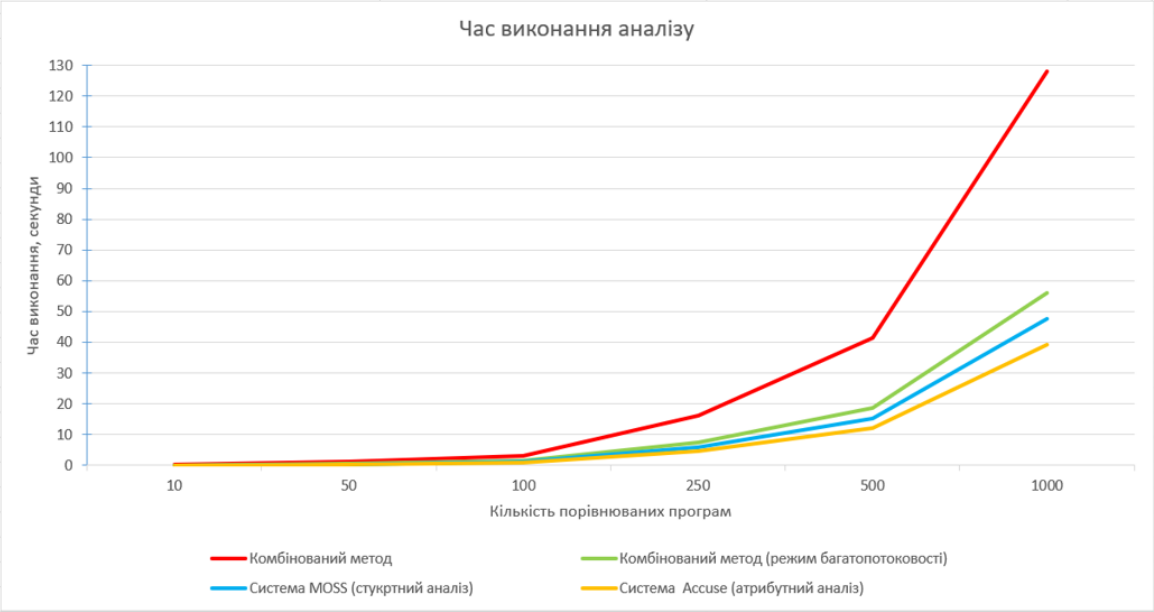
$$повнота = \frac{|\{релевантні\ документи\} \cap \{знайдені\ документи\}|}{|\{релевантні\ документи\}|} \cdot 100\% ,$$

де $\{релевантні\ документи\}$ – фактичні копії програм;

$\{знайдені\ документи\}$ – програми, які метод визначив як копії;

$|\{знайдені\ документи\}|$ – кількість програм

ЕФЕКТИВНІСТЬ.
ЧАС ПОШУКУ ЗАПОЗИЧЕНЬ



Програм	Порівнянь	Комбінований метод	Комбінований метод (режим багатопотоковості)	MOSS (стукртний аналіз)	Accuse (атрибутний аналіз)
10	45	0.14	0.07	0.05	0.04
50	1225	1.05	0.41	0.37	0.32
100	4950	3.00	1.44	1.20	0.81
250	31125	16.05	7.29	5.98	4.75
500	124750	41.42	18.69	15.34	12.18
1000	499500	128.14	56.11	47.67	39.26

НАУКОВА НОВИЗНА

Запропоновано алгоритмічно-програмний метод пошуку запозичень у програмному коді, заснований на комбінації текстового, атрибутного та структурного аналізів програмного коду, який відрізняється від існуючих методів проведенням багатоетапного та комплексного аналізу запозичень та забезпеченням можливості роботи з базою програмних кодів на кожному з етапів методу, що дозволяє підвищити ефективність виявлення запозичень за критерієм повноти пошуку.

ПРАКТИЧНА ЗНАЧУЩІСТЬ

Запропонований алгоритмічно-програмний метод пошуку запозичень у програмному коді дозволив підвищити ефективність пошуку таких запозичень за критерієм повноти та розробити відповідні програмні засоби, придатні до використання в навчальній і комерційній сферах з метою перевірки програмних кодів на наявність плагіату.

Визначений у дослідженні формат подання даних програмної структури для кожного з етапів аналізу комбінованого методу може бути врахований при розробленні програмних засобів для реалізації ефективного за критерієм часу пошуку запозичень в базі програмних кодів.

АПРОБАЦІЯ

Заболотня Т.М., Свинарчук М.В., Алгоритмічно-програмний метод пошуку запозичень у програмному коді, Збірник XIV наукової конференції магістрантів та аспірантів «Прикладна математика та комп’ютинг» ПМК-2021 (17-19 листопада 2021 р., м. Київ).

УНІКАЛЬНІСТЬ РОБОТИ

UNICHECK

Ім'я користувача:
Заболотня Тетяна Миколаївна

Дата перевірки:
13.06.2022 00:34:12 EEST

Дата звіту:
13.06.2022 01:08:01 EEST

ID перевірки:
1011554016

Тип перевірки:
Doc vs Internet + Library

ID користувача:
83364

Назва документа: Свинарчук_Unicheck-2

Кількість сторінок: 53 **Кількість слів:** 9957 **Кількість символів:** 76291 **Розмір файлу:** 116.88 KB **ID файлу:** 1011423871

4.58%
Схожість

Найбільша схожість: 2.06% з джерелом з Бібліотеки (ID файлу: 1004047619)

0.18% Джерела з Інтернету	2	Сторінка 55
4.4% Джерела з Бібліотеки	90	Сторінка 55

0.89% Цитат

Цитати	3	Сторінка 56

Вилучення списку бібліографічних посилань вимкнене

0%
Вилучень

НАПРЯМКИ ПОДАЛЬШОЇ РОБОТИ

1. Аналіз атрибутів коду:
   a) дослідження атрибутних характеристик на стійкість до різних типів модифікацій;
   b) дослідження значення констант розміру вікна та коефіцієнту важливості для атрибутів на залежність від мови програмування;
   c) використання нейронних мереж для класифікації та визначення плагіату на основі стилю програмування.
2. Аналіз структури коду:
   a) оптимізація вибору параметрів k і t для алгоритму просіювання в залежності від мови програмування;
   b) дослідження альтернативних методів вибору послідовності міток для підвищення стійкості до модифікацій програмного коду.

ВИСНОВКИ

1. Проведено дослідження чотирьох типів запозичень програмного коду.
2. Класифіковано існуючі методи для пошуку запозичень у програмному коді: метод текстового порівняння, метод на основі атрибутів коду та на основі структури коду. Проаналізовано існуючі алгоритми їх реалізацій.
3. Розроблено комбінований метод пошуку запозичень, який передбачає поетапний аналіз програмного коду текстовим, атрибутним та структурним методами для пошуку запозичень.
4. Розроблено програмне забезпечення у вигляді Java-бібліотеки, яке реалізує запропонований комбінований метод з використанням багатопотоковості для оптимізації швидкодії.
5. Запропонований метод дозволив підвищити ефективність виявлення запозичень за критерієм повноти пошуку, але показав більший час аналізу.
6. Визначено напрямки подальшого дослідження.

Дякую за увагу!

Додаток 3

Лістинг програми

JavacAdapter.java

```
import java.io.File;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Paths;
import java.util.Collections;

import javax.tools.Diagnostic;
import javax.tools.DiagnosticCollector;
import javax.tools.JavaCompiler;
import javax.tools.JavaCompiler.CompilationTask;
import javax.tools.StandardJavaFileManager;
import javax.tools.ToolProvider;

import com.sun.source.tree.CompilationUnitTree;
import com.sun.source.tree.LineMap;
import com.sun.source.util.JavacTask;
import com.sun.source.util.SourcePositions;
import com.sun.source.util.Trees;

import de.plag.ErrorConsumer;
import de.plag.TokenConstants;

public class JavacAdapter {

    private static final JavaCompiler javac = ToolProvider.getSystemJavaCompiler();

    public int parseFiles(File directory, Iterable<File> pathedFiles, final JavaParser parser) {
        final StandardJavaFileManager fileManager = javac.getStandardFileManager(null, null, StandardCharsets.UTF_8);
        var listener = new DiagnosticCollector<>();
        var javaFiles = fileManager.getJavaFileObjectsFromFiles(pathedFiles);
        final CompilationTask task = javac.getTask(null, fileManager, listener, null, null, javaFiles);
        final Trees trees = Trees.instance(task);
        final SourcePositions positions = trees.getSourcePositions();
        for (final CompilationUnitTree ast : executeCompilationTask(task)) {
            final String filename = fileNameOf(directory, ast);
            final LineMap map = ast.getLineMap();
            ast.accept(new JavaTokenGeneratingTreeScanner(filename, parser, map, positions, ast), null);
            parser.add(TokenConstants.FILE_END, filename, 1, -1, -1);
        }
        return processErrors(parser.getErrorConsumer(), listener);
    }

    private Iterable<? extends CompilationUnitTree> executeCompilationTask(final CompilationTask task) {
        Iterable<? extends CompilationUnitTree> abstractSyntaxTrees = Collections.emptyList();
        try {
            abstractSyntaxTrees = ((JavacTask) task).parse();
```

```
            } catch (IOException exception) {
                exception.printStackTrace();
            }
            return abstractSyntaxTrees;
        }

        private String fileNameOf(File directory, final CompilationUnitTree
ast) {
            if (directory == null)
                return ast.getSourceFile().getName();
            else {
                return
Paths.get(directory.toURI()).relativize(Paths.get(ast.getSourceFile().toUri
())).toString();
            }
        }

        private    int    processErrors(final    ErrorConsumer    consumer,
DiagnosticCollector<Object> listener) {
            int errors = 0;
            for (Diagnostic<?> diagnosticItem : listener.getDiagnostics()) {
                if                (diagnosticItem.getKind()                ==
javax.tools.Diagnostic.Kind.ERROR) {
                    consumer.addError(diagnosticItem.toString());
                    errors++;
                }
            }
            return errors;
        }

    }
```

JavaLanguage.java

```
    package de.plag.java;

    import java.io.File;

    import de.plag.ErrorConsumer;
    import de.plag.TokenList;

    /**
     * Language for Java 9 and newer.
     */
    public class JavaLanguage implements de.plag.Language {
        private final JavaParser parser;

        public JavaLanguage(ErrorConsumer errorConsumer) {
            parser = new JavaParser(errorConsumer);
        }

        @Override
        public String[] suffixes() {
            return new String[] {".java", ".JAVA"};
        }

        @Override
        public String getName() {
```

```
        return "Javac based AST plugin";
    }

    @Override
    public String getShortName() {
        return "java";
    }

    @Override
    public int minimumTokenMatch() {
        return 9;
    }

    @Override
    public boolean supportsColumns() {
        return true;
    }

    @Override
    public boolean isPreformatted() {
        return true;
    }

    @Override
    public boolean usesIndex() {
        return false;
    }

    @Override
    public int numberOfTokens() {
        return JavaTokenConstants.NUM_DIFF_TOKENS;
    }

    @Override
    public TokenList parse(File directory, String[] files) {
        return this.parser.parse(directory, files);
    }

    @Override
    public boolean hasErrors() {
        return this.parser.hasErrors();
    }

}
```

JavaParser.java

```
package de.plag.java;

import static java.util.stream.Collectors.toList;

import java.io.File;
import java.util.Arrays;

import de.plag.AbstractParser;
import de.plag.ErrorConsumer;
import de.plag.TokenList;
```

```
public class JavaParser extends AbstractParser {
    private TokenList tokens;

    /**
     * Creates the parser.
     * @param errorConsumer is the consumer for any occurring errors.
     */
    public JavaParser(ErrorConsumer errorConsumer) {
        super(errorConsumer);
    }

    public TokenList parse(File directory, String[] files) {
        tokens = new TokenList();
        errors = 0;
        var pathedFiles = Arrays.stream(files).map(it -> new
File(directory, it)).collect(toList());
        errors += new JavacAdapter().parseFiles(directory, pathedFiles,
this);
        return tokens;
    }

    public void add(int type, String filename, long line, long column,
long length) {
        tokens.addToken(new JavaToken(type, filename, (int) line, (int)
column, (int) length));
    }

    public void increaseErrors() {
        errors++;
    }
}
```

JavaToken.java

```
package de.plag.java;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import de.plag.Token;
import de.plag.TokenConstants;

public class JavaToken extends Token implements JavaTokenConstants {

    private final Logger logger =
LoggerFactory.getLogger(JavaToken.class);

    public JavaToken(int type, String file, int line, int column, int
length) {
        super(type, file, line, column, length);
    }

    @Override
    protected String type2string() {
        switch (type) {
            case TokenConstants.FILE_END:
                return "EOF";
```

```
case TokenConstants.SEPARATOR_TOKEN:
    return "METHOD_SEPARATOR";
case J_PACKAGE:
    return "PACKAGE";
case J_IMPORT:
    return "IMPORT";
case J_CLASS_BEGIN:
    return "CLASS{";
case J_CLASS_END:
    return "}CLASS";
case J_METHOD_BEGIN:
    return "METHOD{";
case J_METHOD_END:
    return "}METHOD";
case J_VARDEF:
    return "VARDEF";
case J_SYNC_BEGIN:
    return "SYNC{";
case J_SYNC_END:
    return "}SYNC";
case J_DO_BEGIN:
    return "DO{";
case J_DO_END:
    return "}DO";
case J_WHILE_BEGIN:
    return "WHILE{";
case J_WHILE_END:
    return "}WHILE";
case J_FOR_BEGIN:
    return "FOR{";
case J_FOR_END:
    return "}FOR";
case J_SWITCH_BEGIN:
    return "SWITCH{";
case J_SWITCH_END:
    return "}SWITCH";
case J_CASE:
    return "CASE";
case J_TRY_BEGIN:
    return "TRY{";
case J_CATCH_BEGIN:
    return "CATCH{";
case J_CATCH_END:
    return "}CATCH";
case J_FINALLY:
    return "FINALLY";
case J_IF_BEGIN:
    return "IF{";
case J_ELSE:
    return "ELSE";
case J_IF_END:
    return "}IF";
case J_COND:
    return "COND";
case J_BREAK:
    return "BREAK";
case J_CONTINUE:
    return "CONTINUE";
```

```
case J_RETURN:
    return "RETURN";
case J_THROW:
    return "THROW";
case J_IN_CLASS_BEGIN:
    return "INCLASS{";
case J_IN_CLASS_END:
    return "}INCLASS";
case J_APPLY:
    return "APPLY";
case J_NEWCLASS:
    return "NEWCLASS";
case J_NEWARRAY:
    return "NEWARRAY";
case J_ASSIGN:
    return "ASSIGN";
case J_INTERFACE_BEGIN:
    return "INTERF{";
case J_INTERFACE_END:
    return "}INTERF";
case J_CONSTR_BEGIN:
    return "CONSTR{";
case J_CONSTR_END:
    return "}CONSTR";
case J_INIT_BEGIN:
    return "INIT{";
case J_INIT_END:
    return "}INIT";
case J_VOID:
    return "VOID";
case J_ARRAY_INIT_BEGIN:
    return "ARRINIT{";
case J_ARRAY_INIT_END:
    return "ARRINIT}";
case J_ENUM_BEGIN:
    return "ENUM{";
case J_ENUM_CLASS_BEGIN:
    return "ENUM_CLA";
case J_ENUM_END:
    return "}ENUM";
case J_GENERIC:
    return "GENERIC";
case J_ASSERT:
    return "ASSERT";
case J_ANNO:
    return "ANNO";
case J_ANNO_MARKER:
    return "ANNOMARK";
case J_ANNO_M_BEGIN:
    return "ANNO_M{";
case J_ANNO_M_END:
    return "}ANNO_M";
case J_ANNO_T_BEGIN:
    return "ANNO_T{";
case J_ANNO_T_END:
    return "}ANNO_T";
case J_ANNO_C_BEGIN:
    return "ANNO_C{";
```

```
            case J_ANNO_C_END:
                return "}ANNO_C";
            case J_MODULE_BEGIN:
                return "MODULE{";
            case J_MODULE_END:
                return "}MODULE";
            case J_EXPORTS:
                return "EXPORTS";
            case J_PROVIDES:
                return "PROVIDES";
            case J_REQUIRES:
                return "REQUIRES";
            case J_TRY_WITH_RESOURCE:
                return "TRY_RES";
            case J_YIELD:
                return "YIELD";
            case J_DEFAULT:
                return "DEFAULT";
            case J_RECORD_BEGIN:
                return "RECORD{";
            case J_RECORD_END:
                return "}RECORD";

            default:
                logger.error("UNKNOWN: " + type);
                return "<UNKNOWN" + type + ">";
        }
    }
}
```

JavaTokenConstants.java

```
package de.plag.java;

import de.plag.TokenConstants;

public interface JavaTokenConstants extends TokenConstants {

    int J_PACKAGE = 2;
    int J_IMPORT = 3;
    int J_CLASS_BEGIN = 4;
    int J_CLASS_END = 5;
    int J_METHOD_BEGIN = 6;
    int J_METHOD_END = 7;
    int J_VARDEF = 8;
    int J_SYNC_BEGIN = 9;
    int J_SYNC_END = 10;
    int J_DO_BEGIN = 11;
    int J_DO_END = 12;
    int J_WHILE_BEGIN = 13;
    int J_WHILE_END = 14;
    int J_FOR_BEGIN = 15;
    int J_FOR_END = 16;
    int J_SWITCH_BEGIN = 17;
    int J_SWITCH_END = 18;
    int J_CASE = 19;
    int J_TRY_BEGIN = 20;
```

```
int J_CATCH_BEGIN = 21;
int J_CATCH_END = 22;
int J_FINALLY = 23;
int J_IF_BEGIN = 24;
int J_ELSE = 25;
int J_IF_END = 26;
int J_COND = 27;
int J_BREAK = 28;
int J_CONTINUE = 29;
int J_RETURN = 30;
int J_THROW = 31;
int J_IN_CLASS_BEGIN = 32;
int J_IN_CLASS_END = 33;
int J_APPLY = 34;
int J_NEWCLASS = 35;
int J_NEWARRAY = 36;
int J_ASSIGN = 37;
int J_INTERFACE_BEGIN = 38;
int J_INTERFACE_END = 39;
int J_CONSTR_BEGIN = 40;
int J_CONSTR_END = 41;
int J_INIT_BEGIN = 42;
int J_INIT_END = 43;
int J_VOID = 44;
int J_ARRAY_INIT_BEGIN = 45;
int J_ARRAY_INIT_END = 46;

// new in 1.5:
int J_ENUM_BEGIN = 47;
int J_ENUM_CLASS_BEGIN = 48;
int J_ENUM_END = 49;
int J_GENERIC = 50;
int J_ASSERT = 51;

int J_ANNO = 52;
int J_ANNO_MARKER = 53;
int J_ANNO_M_BEGIN = 54;
int J_ANNO_M_END = 55;
int J_ANNO_T_BEGIN = 56;
int J_ANNO_T_END = 57;
int J_ANNO_C_BEGIN = 58;
int J_ANNO_C_END = 59;

// new in 1.7
int J_TRY_WITH_RESOURCE = 60;

// new in 1.9
int J_REQUIRES = 61;
int J_PROVIDES = 62;
int J_EXPORTS = 63;
int J_MODULE_BEGIN = 64;
int J_MODULE_END = 65;

// new in 13
int J_YIELD = 66;

// new in 17
int J_DEFAULT = 67;
```

```
        int J_RECORD_BEGIN = 68;
        int J_RECORD_END = 69;

        final static int NUM_DIFF_TOKENS = 70;
    }
```

JavaTokenGeneratingTreeScanner.java

```
    package de.plag.java;

    import com.sun.source.tree.AnnotationTree;
    import com.sun.source.tree.AssertTree;
    import com.sun.source.tree.AssignmentTree;
    import com.sun.source.tree.BlockTree;
    import com.sun.source.tree.BreakTree;
    import com.sun.source.tree.CaseTree;
    import com.sun.source.tree.CatchTree;
    import com.sun.source.tree.ClassTree;
    import com.sun.source.tree.CompilationUnitTree;
    import com.sun.source.tree.ConditionalExpressionTree;
    import com.sun.source.tree.ContinueTree;
    import com.sun.source.tree.DefaultCaseLabelTree;
    import com.sun.source.tree.DoWhileLoopTree;
    import com.sun.source.tree.EnhancedForLoopTree;
    import com.sun.source.tree.ErroneousTree;
    import com.sun.source.tree.ExportsTree;
    import com.sun.source.tree.ForLoopTree;
    import com.sun.source.tree.IfTree;
    import com.sun.source.tree.ImportTree;
    import com.sun.source.tree.LineMap;
    import com.sun.source.tree.MethodInvocationTree;
    import com.sun.source.tree.MethodTree;
    import com.sun.source.tree.ModuleTree;
    import com.sun.source.tree.NewArrayTree;
    import com.sun.source.tree.NewClassTree;
    import com.sun.source.tree.PackageTree;
    import com.sun.source.tree.ProvidesTree;
    import com.sun.source.tree.RequiresTree;
    import com.sun.source.tree.ReturnTree;
    import com.sun.source.tree.SwitchExpressionTree;
    import com.sun.source.tree.SwitchTree;
    import com.sun.source.tree.SynchronizedTree;
    import com.sun.source.tree.ThrowTree;
    import com.sun.source.tree.Tree;
    import com.sun.source.tree.TryTree;
    import com.sun.source.tree.TypeParameterTree;
    import com.sun.source.tree.VariableTree;
    import com.sun.source.tree.WhileLoopTree;
    import com.sun.source.tree.YieldTree;
    import com.sun.source.util.SourcePositions;
    import com.sun.source.util.TreeScanner;

    final class JavaTokenGeneratingTreeScanner extends TreeScanner<Object,
Object> {
        private final String filename;
        private final JavaParser parser;
```

```
        private final LineMap map;
        private final SourcePositions positions;
        private final CompilationUnitTree ast;

        public  JavaTokenGeneratingTreeScanner(String  filename,  JavaParser
parser, LineMap map, SourcePositions positions, CompilationUnitTree ast) {
            this.filename = filename;
            this.parser = parser;
            this.map = map;
            this.positions = positions;
            this.ast = ast;
        }

        /**
         * Convenience method that adds a specific token.
         * @param tokenType is the type from {@link JavaTokenConstants}.
         * @param position is the start position of the token.
         * @param length is the length of the token.
         */
        private void addToken(int tokenType, long position, int length) {
            parser.add(tokenType,   filename,   map.getLineNumber(position),
map.getColumnNumber(position), length);
        }

        /**
         * Convenience method that adds a specific token.
         * @param tokenType is the type from {@link JavaTokenConstants}.
         * @param start is the start position of the token.
         * @param end is the end position of the token for the calculation of
the length.
         */
        private void addToken(int tokenType, long start, long end) {
            parser.add(tokenType,     filename,     map.getLineNumber(start),
map.getColumnNumber(start), (end - start));
        }

        @Override
        public Object visitBlock(BlockTree node, Object p) {
            long start = positions.getStartPosition(ast, node);
            long end = positions.getEndPosition(ast, node) - 1;
            addToken(JavaTokenConstants.J_INIT_BEGIN, start, 1);
            Object result = super.visitBlock(node, p);
            addToken(JavaTokenConstants.J_INIT_END, end, 1);
            return result;
        }

        @Override
        public Object visitClass(ClassTree node, Object p) {
            long start = positions.getStartPosition(ast, node);
            long end = positions.getEndPosition(ast, node) - 1;

            if (node.getKind() == Tree.Kind.ENUM) {
                addToken(JavaTokenConstants.J_ENUM_BEGIN, start, 4);
            } else if (node.getKind() == Tree.Kind.INTERFACE) {
                addToken(JavaTokenConstants.J_INTERFACE_BEGIN, start, 9);
            } else if (node.getKind() == Tree.Kind.RECORD) {
                addToken(JavaTokenConstants.J_RECORD_BEGIN, start, 1);
            } else if (node.getKind() == Tree.Kind.ANNOTATION_TYPE) {
```

```
                addToken(JavaTokenConstants.J_ANNO_T_BEGIN, start, 10);
            } else if (node.getKind() == Tree.Kind.CLASS) {
                addToken(JavaTokenConstants.J_CLASS_BEGIN, start, 5);
            }
            Object result = super.visitClass(node, p);
            if (node.getKind() == Tree.Kind.ENUM) {
                addToken(JavaTokenConstants.J_ENUM_END, end, 1);
            } else if (node.getKind() == Tree.Kind.INTERFACE) {
                addToken(JavaTokenConstants.J_INTERFACE_END, end, 1);
            } else if (node.getKind() == Tree.Kind.RECORD) {
                addToken(JavaTokenConstants.J_RECORD_END, end, 1);
            } else if (node.getKind() == Tree.Kind.ANNOTATION_TYPE) {
                addToken(JavaTokenConstants.J_ANNO_T_END, end, 1);
            } else if (node.getKind() == Tree.Kind.CLASS) {
                addToken(JavaTokenConstants.J_CLASS_END, end, 1);
            }
            return result;
        }

        @Override
        public Object visitImport(ImportTree node, Object p) {
            long start = positions.getStartPosition(ast, node);
            addToken(JavaTokenConstants.J_IMPORT, start, 6);
            return super.visitImport(node, p);
        }

        @Override
        public Object visitPackage(PackageTree node, Object p) {
            long start = positions.getStartPosition(ast, node);
            addToken(JavaTokenConstants.J_PACKAGE, start, 7);
            return super.visitPackage(node, p);
        }

        @Override
        public Object visitMethod(MethodTree node, Object p) {
            long start = positions.getStartPosition(ast, node);
            long end = positions.getEndPosition(ast, node) - 1;
            addToken(JavaTokenConstants.J_METHOD_BEGIN,                start,
node.getName().length());
            Object result = super.visitMethod(node, p);
            addToken(JavaTokenConstants.J_METHOD_END, end, 1);
            return result;
        }

        @Override
        public Object visitSynchronized(SynchronizedTree node, Object p) {
            long start = positions.getStartPosition(ast, node);
            long end = positions.getEndPosition(ast, node) - 1;
            addToken(JavaTokenConstants.J_SYNC_BEGIN, start, 12);
            Object result = super.visitSynchronized(node, p);
            addToken(JavaTokenConstants.J_SYNC_END, end, 1);
            return result;
        }

        @Override
        public Object visitDoWhileLoop(DoWhileLoopTree node, Object p) {
            long start = positions.getStartPosition(ast, node);
            long end = positions.getEndPosition(ast, node) - 1;
```

```
            addToken(JavaTokenConstants.J_DO_BEGIN, start, 2);
            Object result = super.visitDoWhileLoop(node, p);
            addToken(JavaTokenConstants.J_DO_END, end, 1);
            return result;
        }

        @Override
        public Object visitWhileLoop(WhileLoopTree node, Object p) {
            long start = positions.getStartPosition(ast, node);
            long end = positions.getEndPosition(ast, node) - 1;
            addToken(JavaTokenConstants.J_WHILE_BEGIN, start, 5);
            Object result = super.visitWhileLoop(node, p);
            addToken(JavaTokenConstants.J_WHILE_END, end, 1);
            return result;
        }

        @Override
        public Object visitForLoop(ForLoopTree node, Object p) {
            long start = positions.getStartPosition(ast, node);
            long end = positions.getEndPosition(ast, node) - 1;
            addToken(JavaTokenConstants.J_FOR_BEGIN, start, 3);
            Object result = super.visitForLoop(node, p);
            addToken(JavaTokenConstants.J_FOR_END, end, 1);
            return result;
        }

        @Override
        public Object visitEnhancedForLoop(EnhancedForLoopTree node, Object
p) {
            long start = positions.getStartPosition(ast, node);
            long end = positions.getEndPosition(ast, node) - 1;
            addToken(JavaTokenConstants.J_FOR_BEGIN, start, 3);
            Object result = super.visitEnhancedForLoop(node, p);
            addToken(JavaTokenConstants.J_FOR_END, end, 1);
            return result;
        }

        @Override
        public Object visitSwitch(SwitchTree node, Object p) {
            long start = positions.getStartPosition(ast, node);
            long end = positions.getEndPosition(ast, node) - 1;
            addToken(JavaTokenConstants.J_SWITCH_BEGIN, start, 6);
            Object result = super.visitSwitch(node, p);
            addToken(JavaTokenConstants.J_SWITCH_END, end, 1);
            return result;
        }

        @Override
        public Object visitSwitchExpression(SwitchExpressionTree node, Object
parameterValue) {
            long start = positions.getStartPosition(ast, node);
            long end = positions.getEndPosition(ast, node) - 1;
            addToken(JavaTokenConstants.J_SWITCH_BEGIN, start, 6);
            Object      result      =      super.visitSwitchExpression(node,
parameterValue);
            addToken(JavaTokenConstants.J_SWITCH_END, end, 1);
            return result;
        }
```

```
        @Override
        public Object visitCase(CaseTree node, Object p) {
            long start = positions.getStartPosition(ast, node);
            addToken(JavaTokenConstants.J_CASE, start, 4);
            return super.visitCase(node, p);
        }

        @Override
        public Object visitTry(TryTree node, Object p) {
            long start = positions.getStartPosition(ast, node);
            if (node.getResources().isEmpty())
                addToken(JavaTokenConstants.J_TRY_BEGIN, start, 3);
            else
                addToken(JavaTokenConstants.J_TRY_WITH_RESOURCE, start, 3);
            if (node.getFinallyBlock() != null)
                addToken(JavaTokenConstants.J_FINALLY, start, 3);
            return super.visitTry(node, p);
        }

        @Override
        public Object visitCatch(CatchTree node, Object p) {
            long start = positions.getStartPosition(ast, node);
            long end = positions.getEndPosition(ast, node) - 1;
            addToken(JavaTokenConstants.J_CATCH_BEGIN, start, 5);
            Object result = super.visitCatch(node, p);
            addToken(JavaTokenConstants.J_CATCH_END, end, 1);
            return result;
        }

        @Override
        public Object visitIf(IfTree node, Object p) {
            long start = positions.getStartPosition(ast, node);
            long end = positions.getEndPosition(ast, node) - 1;
            addToken(JavaTokenConstants.J_IF_BEGIN, start, 2);
            node.getCondition().accept(this, p);
            node.getThenStatement().accept(this, p);
            if (node.getElseStatement() != null) {
                start           =           positions.getStartPosition(ast,
node.getElseStatement());
                addToken(JavaTokenConstants.J_ELSE, start, 4);
                node.getElseStatement().accept(this, p);
            }
            addToken(JavaTokenConstants.J_IF_END, end, 1);
            return null;
        }

        @Override
        public Object visitBreak(BreakTree node, Object p) {
            long start = positions.getStartPosition(ast, node);
            addToken(JavaTokenConstants.J_BREAK, start, 5);
            return super.visitBreak(node, p);
        }

        @Override
        public Object visitContinue(ContinueTree node, Object p) {
            long start = positions.getStartPosition(ast, node);
            addToken(JavaTokenConstants.J_CONTINUE, start, 8);
```

```
            return super.visitContinue(node, p);
        }

        @Override
        public Object visitReturn(ReturnTree node, Object p) {
            long start = positions.getStartPosition(ast, node);
            addToken(JavaTokenConstants.J_RETURN, start, 6);
            return super.visitReturn(node, p);
        }

        @Override
        public Object visitThrow(ThrowTree node, Object p) {
            long start = positions.getStartPosition(ast, node);
            addToken(JavaTokenConstants.J_THROW, start, 5);
            return super.visitThrow(node, p);
        }

        @Override
        public Object visitNewClass(NewClassTree node, Object p) {
            long start = positions.getStartPosition(ast, node);
            if (node.getTypeArguments().size() > 0) {
                addToken(JavaTokenConstants.J_GENERIC,     start,     3     +
node.getIdentifier().toString().length());
            }
            addToken(JavaTokenConstants.J_NEWCLASS, start, 3);
            return super.visitNewClass(node, p);
        }

        @Override
        public Object visitTypeParameter(TypeParameterTree node, Object p) {
            long start = positions.getStartPosition(ast, node);
            // This is odd, but also done like this in Java17
            addToken(JavaTokenConstants.J_GENERIC, start, 1);
            return super.visitTypeParameter(node, p);
        }

        @Override
        public Object visitNewArray(NewArrayTree node, Object arg1) {
            long start = positions.getStartPosition(ast, node);
            long end = positions.getEndPosition(ast, node) - 1;
            addToken(JavaTokenConstants.J_NEWARRAY, start, 3);
            if        (node.getInitializers()        !=        null        &&
!node.getInitializers().isEmpty()) {
                start           =           positions.getStartPosition(ast,
node.getInitializers().get(0));
                addToken(JavaTokenConstants.J_ARRAY_INIT_BEGIN, start, 1);
                addToken(JavaTokenConstants.J_ARRAY_INIT_END, end, 1);
            }
            return super.visitNewArray(node, arg1);
        }

        @Override
        public Object visitAssignment(AssignmentTree node, Object p) {
            long start = positions.getStartPosition(ast, node);
            addToken(JavaTokenConstants.J_ASSIGN, start, 1);
            return super.visitAssignment(node, p);
        }
```

```
        @Override
        public Object visitAssert(AssertTree node, Object p) {
            long start = positions.getStartPosition(ast, node);
            addToken(JavaTokenConstants.J_ASSERT, start, 6);
            return super.visitAssert(node, p);
        }

        @Override
        public Object visitVariable(VariableTree node, Object p) {
            long start = positions.getStartPosition(ast, node);
            addToken(JavaTokenConstants.J_VARDEF,                    start,
node.toString().length());
            return super.visitVariable(node, p);
        }

        @Override
        public  Object  visitConditionalExpression(ConditionalExpressionTree
node, Object p) {
            long start = positions.getStartPosition(ast, node);
            addToken(JavaTokenConstants.J_COND, start, 1);
            return super.visitConditionalExpression(node, p);
        }

        @Override
        public Object visitMethodInvocation(MethodInvocationTree node, Object
p) {
            long start = positions.getStartPosition(ast, node);
            addToken(JavaTokenConstants.J_APPLY,                     start,
positions.getEndPosition(ast, node.getMethodSelect()) - start);
            return super.visitMethodInvocation(node, p);
        }

        @Override
        public Object visitAnnotation(AnnotationTree node, Object p) {
            long start = positions.getStartPosition(ast, node);
            addToken(JavaTokenConstants.J_ANNO, start, 1);
            return super.visitAnnotation(node, p);
        }

        @Override
        public Object visitModule(ModuleTree node, Object p) {
            long start = positions.getStartPosition(ast, node);
            long end = positions.getEndPosition(ast, node) - 1;
            addToken(JavaTokenConstants.J_MODULE_BEGIN, start, 6);
            Object result = super.visitModule(node, p);
            addToken(JavaTokenConstants.J_MODULE_END, end, 1);
            return result;
        }

        @Override
        public Object visitRequires(RequiresTree node, Object p) {
            long start = positions.getStartPosition(ast, node);
            addToken(JavaTokenConstants.J_REQUIRES, start, 8);
            return super.visitRequires(node, p);
        }

        @Override
        public Object visitProvides(ProvidesTree node, Object p) {
```

```
            long start = positions.getStartPosition(ast, node);
            addToken(JavaTokenConstants.J_PROVIDES, start, 8);
            return super.visitProvides(node, p);
        }

        @Override
        public Object visitExports(ExportsTree node, Object p) {
            long start = positions.getStartPosition(ast, node);
            addToken(JavaTokenConstants.J_EXPORTS, start, 7);
            return super.visitExports(node, p);
        }

        @Override
        public Object visitErroneous(ErroneousTree node, Object p) {
            parser.increaseErrors();
            return super.visitErroneous(node, p);
        }

        @Override
        public Object visitYield(YieldTree node, Object p) {
            long start = positions.getStartPosition(ast, node);
            long end = positions.getEndPosition(ast, node);
            addToken(JavaTokenConstants.J_YIELD, start, end);
            return super.visitYield(node, p);
        }

        @Override
        public Object visitDefaultCaseLabel(DefaultCaseLabelTree node, Object
p) {
            long start = positions.getStartPosition(ast, node);
            long end = positions.getEndPosition(ast, node);
            addToken(JavaTokenConstants.J_DEFAULT, start, end);
            return super.visitDefaultCaseLabel(node, p);
        }
    }
```

Plag.java

```
package de.plag;

import static de.plag.options.Verbosity.LONG;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;
import java.util.Collections;
import java.util.Optional;
import java.util.Set;
import java.util.stream.Collectors;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import de.plag.clustering.ClusteringFactory;
import de.plag.exceptions.ExitException;
```

```
   import de.plag.exceptions.SubmissionException;
   import de.plag.options.PlagOptions;
   import de.plag.options.LanguageOption;
   import de.plag.strategy.ComparisonMode;
   import de.plag.strategy.ComparisonStrategy;
   import de.plag.strategy.NormalComparisonStrategy;
   import de.plag.strategy.ParallelComparisonStrategy;

   /**
    * This class coordinates the whole errorConsumer flow.
    */
   public class Plag {
       private static final Logger logger = LoggerFactory.getLogger("Plag");

       private final PlagOptions options;

       private final Language language;
       private final ComparisonStrategy comparisonStrategy;
       private final GreedyStringTiling coreAlgorithm; // Contains the
comparison logic.
       private final ErrorCollector errorCollector;
       private final Set<String> excludedFileNames;

       /**
        * Creates and initializes a Plag instance, parameterized by a set of
options.
        * @param options determines the parameterization.
        */
       public Plag(PlagOptions options) {
           this.options = options;
           errorCollector = new ErrorCollector(options);
           coreAlgorithm = new GreedyStringTiling(options);
           language = initializeLanguage();
           comparisonStrategy                                              =
initializeComparisonStrategy(options.getComparisonMode());
           excludedFileNames                                               =
Optional.ofNullable(this.options.getExclusionFileName()).map(this::readExcl
usionFile).orElse(Collections.emptySet());
           options.setExcludedFiles(excludedFileNames); // store for report
       }

       /**
        * If an exclusion file is given, it is read in and all strings are
saved in the set "excluded".
        * @param exclusionFileName the file name or path
        */
       private Set<String> readExclusionFile(final String exclusionFileName)
{
           try    (BufferedReader    reader    =    new    BufferedReader(new
FileReader(exclusionFileName, PlagOptions.CHARSET))) {
               final           var           excludedFileNames           =
reader.lines().collect(Collectors.toSet());
               if (options.getVerbosity() == LONG) {
                   errorCollector.print(null, "Excluded files:");
                   for (var excludedFilename : excludedFileNames) {
                       errorCollector.print(null, " " + excludedFilename);
                   }
               }
```

```
                return excludedFileNames;
            } catch (IOException e) {
                logger.error("Could not read exclusion file: " +
e.getMessage(), e);
                return Collections.emptySet();
            }
        }

        /**
         * Main procedure, executes the comparison of source code submissions.
         * @return the results of the comparison, specifically the submissions
whose similarity exceeds a set threshold.
         * @throws ExitException if the Plag exits preemptively.
         */
        public PlagResult run() throws ExitException {
            // Parse and validate submissions.
            SubmissionSetBuilder builder = new SubmissionSetBuilder(language,
options, errorCollector, excludedFileNames);
            SubmissionSet submissionSet = builder.buildSubmissionSet();

            if (submissionSet.hasBaseCode()) {

coreAlgorithm.createHashes(submissionSet.getBaseCode().getTokenList(),
options.getMinimumTokenMatch(), true);
            }

            int submissionCount = submissionSet.numberOfSubmissions();
            if (submissionCount < 2) {
                throw new SubmissionException("Not enough valid submissions!
(found " + submissionCount + " valid submissions)");
            }

            // Compare valid submissions.
            PlagResult                          result                          =
comparisonStrategy.compareSubmissions(submissionSet);
            errorCollector.print("\nTotal time for comparing submissions: " +
TimeUtil.formatDuration(result.getDuration()), null);


result.setClusteringResult(ClusteringFactory.getClusterings(result.getCompa
risons(), options.getClusteringOptions()));

            return result;
        }

        private     ComparisonStrategy     initializeComparisonStrategy(final
ComparisonMode comparisonMode) {
            return switch (comparisonMode) {
                case   NORMAL   ->   new   NormalComparisonStrategy(options,
coreAlgorithm);
                case  PARALLEL  ->  new  ParallelComparisonStrategy(options,
coreAlgorithm);
            };
        }

        private Language initializeLanguage() {
            LanguageOption languageOption = this.options.getLanguageOption();
```

```
            try {
                Constructor<?>                  constructor                  =
Class.forName(languageOption.getClassPath()).getConstructor(ErrorConsumer.c
lass);
                Object[] constructorParams = {errorCollector};

                Language           language           =           (Language)
constructor.newInstance(constructorParams);

                this.options.setLanguage(language);
                this.options.setLanguageDefaults(language);
                logger.info("Initialized language " + language.getName());
                return language;
            }    catch    (NoSuchMethodException    |    SecurityException    |
ClassNotFoundException | InstantiationException | IllegalAccessException
                    | IllegalArgumentException | InvocationTargetException e)
{
                e.printStackTrace();
                throw   new   IllegalStateException("Language   instantiation
failed:" + e.getMessage());
            }

        }
    }
```

PlagComparison.java

```
    package de.plag;

    import java.util.ArrayList;
    import java.util.Comparator;
    import java.util.List;

    /**
     * This method represents the whole result of a comparison between two
submissions.
     */
    public class PlagComparison implements Comparator<PlagComparison> {

        private static final int ROUNDING_FACTOR = 10;

        private final Submission firstSubmission;
        private final Submission secondSubmission;

        private final List<Match> matches;

        public    PlagComparison(Submission    firstSubmission,    Submission
secondSubmission) {
            this.firstSubmission = firstSubmission;
            this.secondSubmission = secondSubmission;
            matches = new ArrayList<>();
        }

        /**
```

```
        * Add a match to the comparison (token indices and number of tokens),
if it does not overlap with the existing matches.
        * @see Match#Match(int, int, int)
        */
       /* package-private */ final void addMatch(int startOfFirst, int
startOfSecond, int length) {
           for (Match match : matches) {
               if (match.overlap(startOfFirst, startOfSecond, length)) {
                   return;
               }
           }
           matches.add(new Match(startOfFirst, startOfSecond, length));
       }

       @Override
       public int compare(PlagComparison comparison1, PlagComparison
comparison2) {
           return                    Float.compare(comparison2.similarity(),
comparison1.similarity()); // comparison2 first!
       }

       @Override
       public boolean equals(Object other) {
           if (!(other instanceof PlagComparison)) {
               return false;
           }
           return (compare(this, (PlagComparison) other) == 0);
       }

       /**
        * @return the base code matches of the first submission.
        */
       public PlagComparison getFirstBaseCodeMatches() {
           return firstSubmission.getBaseCodeComparison();
       }

       /**
        * @return the first of the two submissions.
        */
       public Submission getFirstSubmission() {
           return firstSubmission;
       }

       /**
        * @return all matches between the two submissions.
        */
       public List<Match> getMatches() {
           return matches;
       }

       /**
        * Get the total number of matched tokens for this comparison.
        */
       public final int getNumberOfMatchedTokens() {
           int numberOfMatchedTokens = 0;

           for (Match match : matches) {
               numberOfMatchedTokens += match.getLength();
```

```
            }

            return numberOfMatchedTokens;
        }

        /**
         * @return the base code matches of the second submissions.
         */
        public PlagComparison getSecondBaseCodeMatches() {
            return secondSubmission.getBaseCodeComparison();
        }

        /**
         * @return the second of the two submissions.
         */
        public Submission getSecondSubmission() {
            return secondSubmission;
        }

        /**
         * @return Maximum similarity in percent of both submissions.
         */
        public final float maximalSimilarity() {
            return Math.max(similarityOfFirst(), similarityOfSecond());
        }

        /**
         * @return Minimum similarity in percent of both submissions.
         */
        public final float minimalSimilarity() {
            return Math.min(similarityOfFirst(), similarityOfSecond());
        }

        /**
         * @return Similarity in percent (what percentage of tokens across
both submissions are matched).
         */
        public final float similarity() {
            boolean subtractBaseCode = firstSubmission.hasBaseCodeMatches()
&& secondSubmission.hasBaseCodeMatches();
            float                               sa                              =
firstSubmission.getSimilarityDivisor(subtractBaseCode);
            float                               sb                              =
secondSubmission.getSimilarityDivisor(subtractBaseCode);
            return (200 * getNumberOfMatchedTokens()) / (sa + sb);
        }

        /**
         * @return Similarity in percent for the first submission (what percent
of the first submission is similar to the
         * second).
         */
        public final float similarityOfFirst() {
            int divisor = firstSubmission.getSimilarityDivisor(true);
            return (divisor == 0 ? 0f : (getNumberOfMatchedTokens() * 100 /
(float) divisor));
        }
```

```
	/**
	 * @return Similarity in percent for the second submission (what
percent of the second submission is similar to the
	 * first).
	 */
	public final float similarityOfSecond() {
	    int divisor = secondSubmission.getSimilarityDivisor(true);
	    return (divisor == 0 ? 0f : (getNumberOfMatchedTokens() * 100 /
(float) divisor));
	}

	/**
	 * @return Similarity in percent rounded down to the nearest tenth.
	 */
	public final float roundedSimilarity() {
	    return ((int) (similarity() * ROUNDING_FACTOR)) / (float)
ROUNDING_FACTOR;
	}

	/**
	 * @return Similarity of the first submission to the basecode in
percent rounded down to the nearest tenth.
	 */
	public final float basecodeSimilarityOfFirst() {
	    return ((int) (firstBasecodeSimilarity() * ROUNDING_FACTOR)) /
(float) ROUNDING_FACTOR;
	}

	/**
	 * @return Similarity of the second submission to the basecode in
percent rounded down to the nearest tenth.
	 */
	public final float basecodeSimilarityOfSecond() {
	    return ((int) (secondBasecodeSimilarity() * ROUNDING_FACTOR)) /
(float) ROUNDING_FACTOR;
	}

	@Override
	public String toString() {
	    return    firstSubmission.getName()    +    "    <->    "    +
secondSubmission.getName();
	}

	private float firstBasecodeSimilarity() {
	    float sa = firstSubmission.getSimilarityDivisor(false);
	    PlagComparison              firstBaseCodeMatches              =
firstSubmission.getBaseCodeComparison();
	    return  firstBaseCodeMatches.getNumberOfMatchedTokens()  *  100  /
sa;
	}

	private float secondBasecodeSimilarity() {
	    float sb = secondSubmission.getSimilarityDivisor(false);
	    PlagComparison              secondBaseCodeMatches              =
secondSubmission.getBaseCodeComparison();
	    return  secondBaseCodeMatches.getNumberOfMatchedTokens()  *  100  /
sb;
	}}
```

PlagResult.java

```
package de.plag;

import java.util.List;

import de.plag.clustering.ClusteringResult;
import de.plag.options.PlagOptions;

/**
 * Encapsulates the results of a comparison of a set of source code submissions.
 */
public class PlagResult {

    private List<PlagComparison> comparisons; // comparisons whose similarity was about the specified threshold

    private final SubmissionSet submissions;

    private final PlagOptions options;

    private final long durationInMillis;

    private final int[] similarityDistribution; // 10-element array representing the similarity distribution of the detected matches.

    private List<ClusteringResult<Submission>> clusteringResult;

    public PlagResult(List<PlagComparison> comparisons, SubmissionSet submissions, long durationInMillis, PlagOptions options) {
        this.comparisons = comparisons;
        this.submissions = submissions;
        this.durationInMillis = durationInMillis;
        this.options = options;
        similarityDistribution = calculateSimilarityDistribution(comparisons);
        comparisons.sort((first, second) -> Float.compare(second.similarity(), first.similarity())); // Sort by percentage (descending).
    }

    /**
     * Drops elements from the comparison list to free memory. Note, that this affects the similarity distribution and is
     * only meant to be used if you don't need the information about comparisons with lower match percentage anymore.
     * @param limit the number of comparisons to keep in the list
     */
    public void dropComparisons(int limit) {
        this.comparisons = this.getComparisons(limit);
    }

    public void setClusteringResult(List<ClusteringResult<Submission>> clustering) {
        this.clusteringResult = clustering;
    }
```

```
        /**
         * @return a list of all comparisons sorted by percentage (descending)
         */
        public List<PlagComparison> getComparisons() {
            return comparisons;
        }

        /**
         * Returns the first n comparisons (sorted by percentage, descending),
limited by the specified parameter.
         *  @param numberOfComparisons specifies the number of requested
comparisons. If set to -1, all comparisons will be
         * returned.
         * @return a list of comparisons sorted descending by percentage.
         */
        public List<PlagComparison> getComparisons(int numberOfComparisons) {
            if (numberOfComparisons == -1) {
                return comparisons;
            }
            return    comparisons.subList(0,    Math.min(numberOfComparisons,
comparisons.size()));
        }

        /**
         * @return the duration of the comparison in milliseconds.
         */
        public long getDuration() {
            return durationInMillis;
        }

        /**
         * @return the submission set that contains both the valid submissions
and the invalid ones.
         */
        public SubmissionSet getSubmissions() {
            return submissions;
        }

        /**
         * @return the total number of submissions that have been compared.
         */
        public int getNumberOfSubmissions() {
            return  submissions.numberOfSubmissions(); // Convenience method
to preserve API
        }

        /**
         * @return the Plag options with which the Plag run was configured.
         */
        public PlagOptions getOptions() {
            return options;
        }

        /**
         * Returns the similarity distribution of detected matches in a 10-
element array. Each entry represents the absolute
         * frequency of matches whose similarity lies within the respective
interval. Intervals: 0: [0% - 10%), 1: [10% - 20%),
```

```
         * 2: [20% - 30%), ..., 9: [90% - 100%]
         * @return the similarity distribution array.
         */
        public int[] getSimilarityDistribution() {
            return similarityDistribution;
        }

        public List<ClusteringResult<Submission>> getClusteringResult() {
            return this.clusteringResult;
        }

        @Override
        public String toString() {
            return String.format("PlagResult { comparisons: %d, duration: %d
ms, language: %s, submissions: %d }", getComparisons().size(),
                    getDuration(),          getOptions().getLanguageOption(),
submissions.numberOfSubmissions());
        }

        /**
         * Note: Before, comparisons with a similarity below the given
threshold were also included in the similarity matrix.
         */
        private  int[]  calculateSimilarityDistribution(List<PlagComparison>
comparisons) {
            int[] similarityDistribution = new int[10];

            comparisons.stream().map(PlagComparison::similarity).map(percent
-> percent / 10).map(Float::intValue).map(index -> index == 10 ? 9 : index)
                    .forEach(index -> similarityDistribution[index]++);

            return similarityDistribution;
        }
    }
```

Match.java

```
    package de.plag;

    /**
     * Represents two sections of two submissions that are similar.
     */
    public class Match {

        private final int startOfFirst;
        private final int startOfSecond;
        private final int length;

        /**
         * Creates a match.
         * @param startOfFirst is the starting token index in the first
submission.
         * @param startOfSecond is the starting token index in the second
submission.
         * @param length is the length of these similar sections (number of
tokens).
         */
```

```
        public Match(int startOfFirst, int startOfSecond, int length) {
            this.startOfFirst = startOfFirst;
            this.startOfSecond = startOfSecond;
            this.length = length;
        }

        /**
         * @return the starting index in the first submission.
         */
        public int getStartOfFirst() {
            return startOfFirst;
        }

        /**
         * @return the starting index in the second submission.
         */
        public int getStartOfSecond() {
            return startOfSecond;
        }

        /**
         * @return the length of the similar sections, meaning the number of
tokens.
         */
        public int getLength() {
            return length;
        }

        /**
         * Checks if two matches overlap.
         * @return true if they do.
         */
        public   final   boolean   overlap(int   otherStartOfFirst,   int
otherStartOfSecond, int oLength) {
            if (startOfFirst < otherStartOfFirst) {
                if ((otherStartOfFirst - startOfFirst) < length) {
                    return true;
                }
            } else {
                if ((startOfFirst - otherStartOfFirst) < oLength) {
                    return true;
                }
            }

            if (startOfSecond < otherStartOfSecond) {
                return (otherStartOfSecond - startOfSecond) < length;
            } else {
                return (startOfSecond - otherStartOfSecond) < oLength;
            }
        }
    }s
```