ДВНЗ «ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ»
Факультет комп'ютерно-інформаційних технологій та автоматизації
Кафедра прикладної математики та інформатики

«До захисту допущено»
В.о. завідувача кафедри
Наталія МАСЛОВА
(підпис) (Ім'я та ПРІЗВИЩЕ)
«_____»_____2023 р.

Випускна кваліфікаційна робота

бакалавра
(освітній ступінь)

на тему
«Створення аналізаторів програмного коду»
зі спецчастиною
«Розробка програмних модулів, інтерфейсу засобами API сервісів»

Виконав (-ла): студент (-ка) 4 курсу, групи ІПЗ-19
(шифр групи)
спеціальності 121 Інженерія програмного забезпечення
освітньої програми 121 «Інженерія програмного забезпечення»
(шифр і назва спеціальності)
Чуб Анастасія Олександрівна _____
(прізвище та ініціали) (підпис)

Керівник доц. каф. ПМІ, к.т.н., доцент Наталія МАСЛОВА _____
(посада, науковий ступінь, вчене звання, Ім'я та ПРІЗВИЩЕ) (підпис)

Рецензент доцент каф. АТ, к.т.н., доцент Анна ВОРОПАЄВА _____
(посада, науковий ступінь, вчене звання, Ім'я та ПРІЗВИЩЕ) (підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент _____
(підпис)

Луцьк – 2023р.

ДВНЗ «Донецький національний технічний університет»
Факультет комп'ютерно-інформаційних технологій та автоматизації
Кафедра прикладної математики та інформатики
Освітній ступень бакалавр
Спеціальність 121 Інженерія програмного забезпечення
(шифр і назва)
Освітня програма 121 Інженерія програмного забезпечення
(шифр і назва)

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри
_____ Наталія МАСЛОВА
(підпис) (Ім'я та ПРІЗВИЩЕ)
«____»_____2023 року

ЗАВДАННЯ
НА ДИПЛОМНУ РОБОТУ СТУДЕНТУ

Чуб Анастасії Олександрівні
(прізвище, ім'я, по батькові)

1. Тема роботи «Створення аналізаторів програмного коду»
Спецчастина «Розробка програмних модулів, інтерфейсу засобами API сервісів»
Керівник роботи Маслова Н. О., к.т.н., доцент
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом від « « 5» травня 2023 року № 175
2. Строк подання студентом роботи 08 червня 2023 року
3. Вихідні дані до роботи результати переддипломної практики

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1) аналіз предметної області та метрик коду
2) опис методів аналізу програмного коду та оцінки якості програмного забезпечення
опис процесу проектування програмного засобу та архітектурних особливостей
3) опис алгоритму, розробленого для рішення задачі,
опис парсеру, подробиці реалізації окремих модулів та підсистем
4) опис роботи розробленої програмної системи, засоби безпеки;
5) тестування програмної розробки.
5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
робота містить 20 рисунків, 15 слайдів презентації та інший графічний матеріал, у кількості, достатній для демонстрації сутності роботи

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Назарова І.А. доцент каф. ПМІ		7.06.2023

7. Дата видачі завдання 5 травня 2023 року

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломної роботи	Строк виконання етапів роботи	Примітка
1	Огляд літератури з предметної області	05.05.2023 – 10.05.2023	
2	Збір інформації, аналіз ринку	11.05.2023 – 15.05.2023	
3	Розробка структури програмного забезпечення, вибір даних для обробки та аналізу.	16.05.2023 – 20.05.2023	
4	Тестування системи	21.05.2023 – 01.06.2023	
5	Оформлення пояснювальної записки	01.06.2023 – 07.06.2023	
6	Нормоконтроль пояснювальної записки, її перевірка антиплагіатною системою	08.06.2023- 10.06.2023	
7	Оформлення супутніх матеріалів (розробка графічного матеріалу, написання доповіді, тощо) та рецензування роботи	11.06.2023- 21.06.2023	
8	Захист роботи	23.06.2023	

Студент _____ Анастасія ЧУБ
(підпис) (Ім'я та ПРІЗВИЩЕ)

Керівник роботи _____ Наталія МАСЛОВА
(підпис) (Ім'я та ПРІЗВИЩЕ)

А Н О Т А Ц І Я

Чуб Анастасія Олександрівна. Створення аналізаторів програмного коду. Випускна кваліфікаційна робота на здобуття освітнього ступеня «бакалавр» за спеціальністю 121 «Інженерія програмного забезпечення». – ДВНЗ ДонНТУ, Луцьк, 2023.

Об’єкт дослідження - технології аналізу програмного коду та його метрик.

Предмет дослідження – методи аналізу безпеки програмного коду.

Метою роботи є аналіз застосування методів та розробка програмного інструменту аналізу програмного коду за допомогою API.

Методи досліджень базуються на сучасних положеннях статичного аналізу коду, інтелектуального аналізу даних, сучасних застосуваннях в кібербезпеці та кібераналізі.

Наукова новизна полягає в створенні аналітичного огляду застосування сучасних методів аналізу коду.

Практичне значення полягає у можливості використання розробленого інструменту для допомоги програмістам в написанні коду, покращення безпеки програмних продуктів та аналізу архітектури проектів.

Ключові слова: аналіз коду, аналізатор, Java, метрики коду, якість програмного забезпечення, лексичний аналізатор.

ЗМІСТ

ВСТУП

Зміст дослідження. Інформаційні інтелектуальні технології (ІТ) на сучасному етапі розвитку суспільства є основним каталізатором розвитку низки галузей науково-виробничої діяльності, які пов’язані з математичним моделюванням різноманітних процесів.

Для створення ефективних і високопродуктивних програм одним із важливих компонентів програмування є оптимізація коду та зменшення зайвого коду. Багато факторів впливають на якість вихідного коду; ці включають мову програмування, знання та досвід програміста та інші фактори. Як показав аналіз вихідного коду, значна частина якості програми залежить від дотримання програмних параметрів [1].

Компілятори вихідного коду шукають лише помилки, а те, що відповідає метрикам, залишається за програмістом.

Вже давно прийнято, що «поганий або гарний код» не пов’язаний з ефективністю використання обчислювальних ресурсів (швидкість програми, обсяг оперативної пам’яті), а з налагодженням і модифікацією програмного забезпечення на етапах власної розробки та супроводу. Якість коду в цьому випадку оцінюється за допомогою таких критеріїв, як правильне розбиття програми на модулі, обмеження використання потенційно небезпечних мовних конструкцій і чітке оформлення вихідного коду [2]. Створення універсальних критеріїв, які дозволяють визначити, чи є програма надійною, є надзвичайно складним завданням, навіть якщо ви визначите, що складають ключові показники якості коду. Як би там не було, ця оцінка надто індивідуальна, і вона може відрізнятися від одного програміста до іншого в залежності від типу проекту. Отже, існуючі методи визначення метрик коду в основному обмежуються обчисленням відповідних значень, які повністю залежать від людини.

Щоб пришвидшити процес підготовки висококваліфікованих фахівців, практика показує, що вивчення метрик програмування необхідно розпочати негайно [3]. Ось чому створення програмного забезпечення, яке відповідає цим метрикам, є проблемою, яка є актуальною.

Завдання:

- розглянути теоретичні засади оцінювання якості коду в програмній інженерії;

- визначити критерії якості програмного забезпечення;

- дослідити методи та засоби розробки системи для оцінки якості програмного коду користувача;

- розробити та впровадити систему для оцінки якості програмного коду користувача.

Методи дослідження – методи системного аналізу, аналіз наукової літератури, спостереження, абстрагування, узагальнення.

Джерела дослідження представлені науковими працями таких науковців, як В. Антонюк, Е. Аллен, В. Бабак, К. Бек, Дж. Блох, Д. Поліщук, Л. Ситник, Є. Скулиш, К. Асановіч, Б. Чепмен, С. Кім, М. Тейлор та інших.

Результати мають практичне значення, оскільки вони вказують на те, що дослідження ґрунтується на результатах поглибленого вивчення особливостей розробки та впровадження системи для оцінки якості програмного коду користувача. Крім того, вони запропонували новий метод аналізу коду проектів, заснований на багатопараметричній обробці рядків коду, що є новим підходом до аналізу якості програмного коду.

У дипломній роботі міститься вступ, чотири розділи, висновки, список джерел посилання та додатки. На 97 сторінках викладено основні ідеї роботи. У списку використаних джерел є двадцять назв.

1 ОГЛЯД ПРЕДМЕТНОЇ ГАЛУЗІ І ПОСТАНОВКА. ЗАДАЧІ ДОСЛІДЖЕННЯ

1.1. Принципи оцінювання якості коду в програмній інженерії

У контексті розробки програмного забезпечення якість програмного забезпечення відноситься до двох пов’язаних, але різних понять [2]:

- функціональна якість програмного забезпечення;
- структурна якість програмного забезпечення.

Функціональна якість програмного забезпечення відбиває те, наскільки добре воно відповідає заданому проекту з урахуванням функціональних вимог чи специфікацій. Цей атрибут також може бути описаний як придатність програмного забезпечення для певної мети або його порівняння з конкурентами на ринку як продукт, що стоїть. Це ступінь, у якій було зроблено правильне програмне забезпечення.

Структурна якість програмного забезпечення стосується того, наскільки воно відповідає нефункціональним вимогам, які підтримують виконання функціональних вимог, таких як надійність або ремонтопридатність. У більшою мірою це пов’язано зі ступенем, у якому програмне забезпечення працює так, як необхідно.

Багато аспектів структурної якості можуть бути оцінені лише статично за допомогою аналізу внутрішньої структури програмного забезпечення, його вихідного коду, на рівні модуля, системного рівня (іноді званого наскрізним тестуванням) фактично відповідає тому, як його архітектура дотримується надійних принципів архітектури програмного забезпечення, викладених у документі на тему, підготовлену Object Management Group (OMG).

Однак деякі структурні якості, такі як зручність використання, можуть бути оцінені тільки динамічно (користувачі або інші особи, що діють від їх імені, взаємодіють з програмним забезпеченням або, принаймні, з деяким

прототипом або частковою реалізацією; навіть взаємодія з макетом, зробленим у картоні, є динамічним тестом, тому що таку версію можна вважати прототипом). Інші аспекти, такі як надійність, можуть стосуватися як програмного забезпечення, а й базового устаткування, тому його можна оцінювати як статично, і динамічно (навантажувальне тестування).

Варто зазначити, що функціональна якість зазвичай оцінюється динамічно, але можна використовувати статичні тести (наприклад, огляди програмного забезпечення).

Історично склалося так, що структура, класифікація та термінологія атрибутів та показників, що застосовуються до управління якістю програмного забезпечення, були отримані або вилучені із стандарту ISO 9126 та наступного стандарту ISO/IEC 25000 [6].

На основі цих моделей Консорціум з якості IT-програмного забезпечення (CISQ) визначив п’ять основних бажаних структурних характеристик, необхідних для того, щоб частина програмного забезпечення забезпечувала цінність для бізнесу:

- надійність;
- ефективність;
- безпека;
- ремонтопридатність;
- розмір.

Слід зазначити, що вимірювання якості програмного забезпечення кількісно визначає, якою мірою програма або система оцінюється по кожному з цих п’яти параметрів. Агрегований захід якості програмного забезпечення може бути розрахований за допомогою якісної або кількісної схеми оцінки або їх поєднання, а потім за допомогою системи зважування, що відображає пріоритети [7].

Цей погляд на якість програмного забезпечення в лінійному континуумі доповнюється аналізом «критичних помилок програмування», які за певних

обставин можуть призвести до катастрофічних збоїв або зниження продуктивності, що робить цю систему непридатною для використання незалежно від рейтингу, що базується на агрегованих вимірах. Такі помилки програмування, виявлені на системному рівні, становлять до 90 відсотків виробничих проблем.

Як наслідок, якість коду без контексту всієї системи, як його описав У. Едвардс Демінг, має обмежену цінність.

Для перегляду, дослідження, аналізу та передачі вимірювань якості програмного забезпечення концепції та методи візуалізації інформації надають візуальні інтерактивні засоби, корисні, зокрема, якщо кілька показників якості програмного забезпечення мають бути пов’язані один з одним або з компонентами програмного забезпечення чи системи.

Наприклад, карти програмного забезпечення являють собою спеціалізований підхід, який «може виражати та об’єднувати інформацію про розробку програмного забезпечення, якість програмного забезпечення та системну динаміку».

Якість програмного забезпечення також грає роль етапі випуску програмного проекту. Зокрема, якість та встановлення процесів випуску (також процесів виправлення), управління конфігурацією є важливими частинами загального процесу розробки програмного забезпечення.

Вимірювання якості програмного забезпечення полягає в кількісному визначенні того, якою мірою система або програмне забезпечення має бажані характеристики. Це може бути виконано за допомогою якісних чи кількісних засобів чи їх поєднання.

В обох випадках для кожної бажаної характеристики існує набір вимірних атрибутів, існування яких у частині програмного забезпечення або системи, як правило, корелює та асоціюється з цією характеристикою. Наприклад, атрибут, пов’язаний із переносимістю, - це кількість операторів, що залежать від мети, в програмі.

Точніше, під час використання підходу «Розгортання функції якості» ці вимірні атрибути є «як», які необхідно забезпечити, щоб включити «що» у наведеному вище визначенні якості програмного забезпечення.

Дерево залежності між характеристиками якості програмного забезпечення та їх вимірними атрибутами представлене на діаграмі праворуч, де кожна з 5 характеристик, що мають значення для користувача (праворуч) або власника бізнес-системи, залежить від вимірних атрибутів (ліворуч):

- практика архітектури додатків;
- практика кодування;
- складність програми;
- документація;
- портативність;
- технічний та функціональний обсяг.

Кореляція між помилками програмування та виробничими дефектами показує, що основні помилки коду становлять 92 відсотки від загальної кількості помилок у вихідному коді. Ці численні проблеми лише на рівні коду зрештою становлять лише 10 відсотків дефектів у виробництві. Погана практика розробки програмного забезпечення на рівні архітектури складає всього 8 відсотків від загальної кількості дефектів, але вимагає більше половини зусиль, що витрачаються на усунення проблем, і призводить до 90 відсотків серйозних проблем з надійністю, безпекою та ефективністю у виробничому середовищі.

1.2. Критерії якості програмного забезпечення

Показники надійності використовуються для кількісного визначення надійності програмного продукту. Вибір метрики залежить від типу системи, до якої вона застосовується, а також від умов предметної області, яка її використовує.

Рисунок 1.1 показує показники надійності, які можна використовувати для кількісної оцінки надійності програмного продукту.

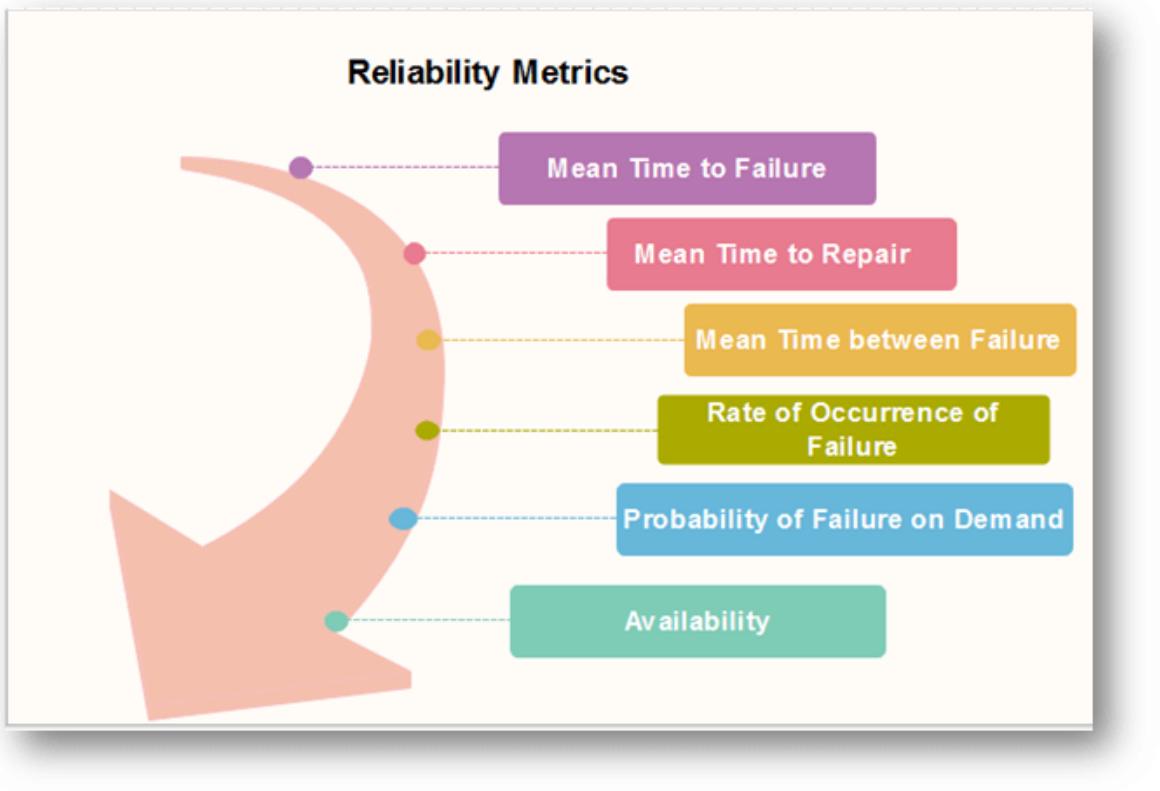


Рисунок 1.1 – Показники надійності ПЗ

Середній час для відмови (MTTF).

Інтервал часу між двома одночасними невдачами називається MTTF. Середній час безвідмовної роботи 200 означає, що можна очікувати один збій на 200 одиниць часу. Одиниці часу повністю залежні від системи і можуть навіть відображатися у кількості транзакцій. Система великих транзакцій підходить MTTF [13].

Наприклад, він підходить для систем автоматизованого проектування, де дизайнер витрачає кілька годин на розробку дизайну, а також для систем текстового процесора.

Середня тривалість ремонту (MTTR).

Збій займає час, щоб виправити. Середній час, необхідний для відстеження та виправлення помилок, що викликають збій, визначається MTTR.

Середній загальний час напрацювання на відмову (MTBR).

Щоб отримати показник MTBF, можна об’єднати показники MTTF і MTTR. Середній час безвідмовної роботи більше середнього часу безвідмовної роботи. Таким чином, середній час безвідмовної роботи 300 означає, що через 300 годин після відмови очікується наступна відмова. У цьому випадку всі методи вимірювання часу використовуються в часі, а не в часі виконання, як це робить MTTF.

Частота помилок (ROCOF).

Це кількість відмов за один інтервал часу. кількість непередбачених подій, які відбулися за певний період часу на робочому місці. ROCOF - це частота виникнення, з якою ймовірно виникнення ролі. ROCOF дорівнює 0,02, що означає, що можуть статися дві помилки на кожні 100 кроків в одиницю робочого часу. Його також називають метрикою ступеня відмови.

Існування ймовірності відмови на запит (POFOD).

Можливість відмови системи при запиті на послуги називається POFOD. Це кількість недоліків, які має система за наявності кількох системних входів.

POFOD означає ймовірність того, що система зупиниться під час запиту на обслуговування. POFOD становить 0,1, що означає, що 1% запитів на обслуговування може завершитися помилкою. POFOD є важливим кроком для важливих систем захисту. POFOD необхідний для систем захисту, де час від часу потрібні послуги.

Доступність.

Імовірність того, що система застосовна для використання в даний момент, називається доступністю. Він розраховує час перезапуску та ремонту системи. Якщо доступність становить 0,995, система може мати доступ до 995 кожні 1000 одиниць часу. Відсоток часу, протягом якого система може

працювати, включаючи заплановані та непередбачені простої доступність системи становить 96%, якщо вона не працює в середньому чотири години зі 100 годин [18].

Визначення областей вимог є методом, який використовується для підвищення надійності системи (рисунок 1.2).

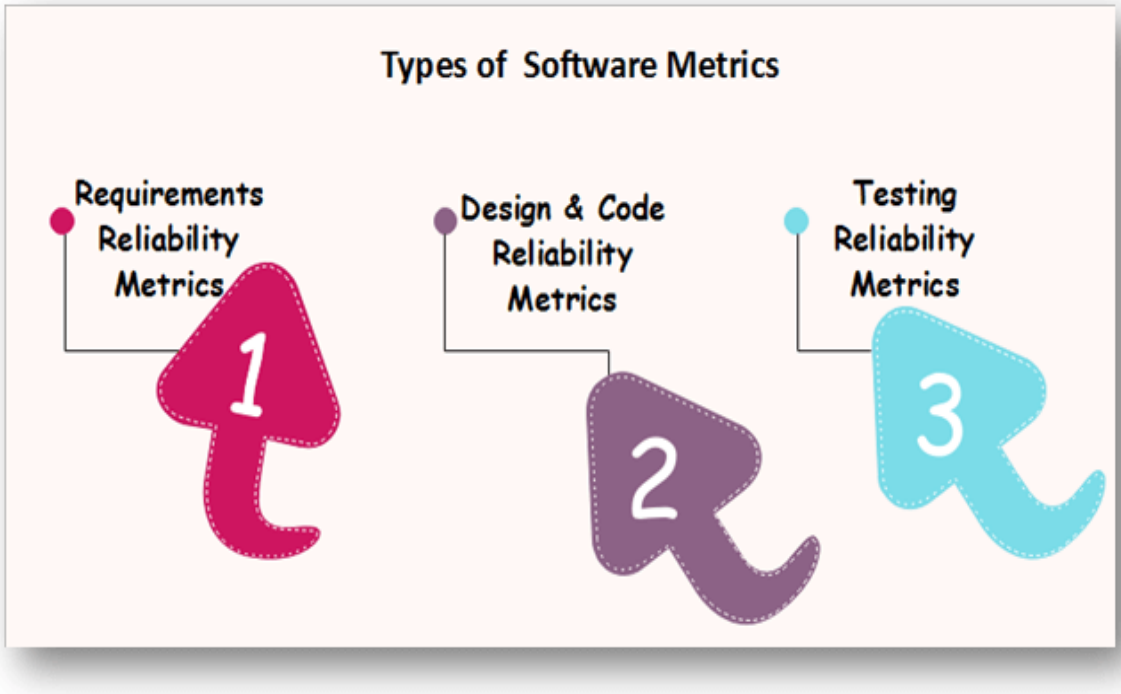


Рисунок 1.2 – Різні типи показників програмного забезпечення

Метрики надійності стандартів.

Програмне забезпечення має містити певні функції, згідно з вимогами. Вони визначають функціональні можливості програми. Якщо вимоги написані таким чином, щоб користувач і розробник не мали проблем. Щоб запобігти втрати цінних даних, вимоги повинні включати прийнятну структуру.

Щоб спростити етап проектування, вимоги мають бути ретельними та детальними. Недостатні дані повинні бути включені в вимоги. Мета «Надійність вимоги» використовується для оцінки вищезазначених елементів якості документа.

Показники надійності коду та дизайну.

Складність, розмір і модульність – це методи якості, які використовуються в проектуванні та кодуванні.

Складні модулі складні для розуміння та схильні до помилок. У поєднанні великого розміру та високої складності або малого розміру надійність буде нижчою. Хоча ці метрики можна використовувати в об'єктно-орієнтованому коді, оцінка якості вимагає додаткових метрик.

Перевірка надійності показників. Ці показники розраховуються за допомогою двох методів. По-перше, система повинна бути забезпечена завданнями, зазначеними у вимогах. Через відсутність функціоналу менше багів.

Обчислення коду, пошук помилок і виправлення є другим методом. Плани тестування включають кілька тестових випадків, щоб переконатися, що система виконує зазначену функціональність.

Кожен метод тестування базується на стані системи та тестує певний набір вимог.

Основною метою ефективної програми перевірки є гарантування того, що кожен компонент пройшов тестування та що система функціонує належним чином і відповідає вимогам.

З іншого боку, всі критерії якості поділяються на «зовнішні», які можуть бачити користувачі програмного забезпечення, і «внутрішні», які можуть бачити тільки розробники програмного забезпечення.

Відповідно до стандарту ICO 9126 зовнішні фактори включають наступне:

- точність;
- стійкість;
- розширюваність;
- повторне використання;
- сумісність, ефективність;

- переносимість;
- простота використання;
- функціональність;
- своєчасність.

Таким чином, кінцевий користувач не бачить елементів, таких як розширюваність і повторне використання.

1.3. Постановка задачі дослідження

Мета полягає в тому, щоб створити систему, яка може оцінювати якість програмного коду користувача.

У нас є три завдання, які ми повинні виконати:

1. Дослідити теоретичні основи оцінювання якості кода в програмній інженерії;
2. Визначити стандарти якості програмного забезпечення;
3. Досліджувати методи та інструменти розробки системи з метою оцінки якості програмного коду користувача;
4. Створити та запустити систему для оцінки якості програмного коду користувача.

1.4. Висновки до розділу 1

Підсумовуючи перший розділ, можна зробити наступні висновки:

1) визначено стандарти оцінювання якості коду в програмній інженерії;

2) окреслено вимоги до якості програмного забезпечення щодо надійності коду;

3) створено завдання для дослідження.

2 МЕТОДИ ТА ЗАСОБИ РОЗРОБКИ СИСТЕМИ ДЛЯ ОЦІНКИ ЯКОСТІ ПРОГРАМНОГО КОДУ КОРИСТУВАЧА

2.1. Дослідження метричних характеристик програмного забезпечення

Забезпечення якості програмного забезпечення (SQA) — це процедура, яка гарантує, що всі процедури, методи, дії та робочі компоненти, які використовуються для розробки програмного забезпечення, були відстеженні та відповідають встановленим стандартам. Це можуть бути стандарти, такі як ISO 9000, модель CMMI та ISO15504.

Тестування якості програмного забезпечення (SQA) включає всі етапи розробки програмного забезпечення, починаючи з визначення вимог і закінчуючи кодуванням і випуском. Якість є його основною метою [11].

План забезпечення якості програмного забезпечення, який також називають SQAP, включає процедури, методи та інструменти, що використовуються для забезпечення відповідності продукту або послуги вимогам, визначеним у SRS (специфікація вимог до програмного забезпечення) (рисунок 2.1).

Рисунок 2.1 – Складові SQAP

Документ плану SQA містить наступні розділи [21]:

- мета;
- посилання;
- організація конфігурації програмного забезпечення;
- повідомлення про проблему та заходи щодо її вирішення;
- інструменти, методи та технології;
- управління кодом;
- документація: збір, обслуговування та зберігання;
- методи тестування.

У процесі життєвого циклу розробки програмного забезпечення, особливо SQA, може бути необхідно дотримуватися стандартів якості.

На основі семи принципів управління якістю ISO 9000 допомагає компаніям гарантувати, що їхні товари та послуги відповідають потребам клієнтів. Рисунок 2.2 показує основні принципи ISO 9000.

Рисунок 2.2 – Принципи ISO 9000

Степінь CMMI. Термін CMMI означає «інтеграція моделі зрілості можливостей». Ця модель виникла в програмному забезпеченні. Його можна використовувати для управління процесами покращення в проекті, відділі чи в організації в цілому. Рисунок 2.3 показує п’ять рівнів CMMI та їхні особливості.

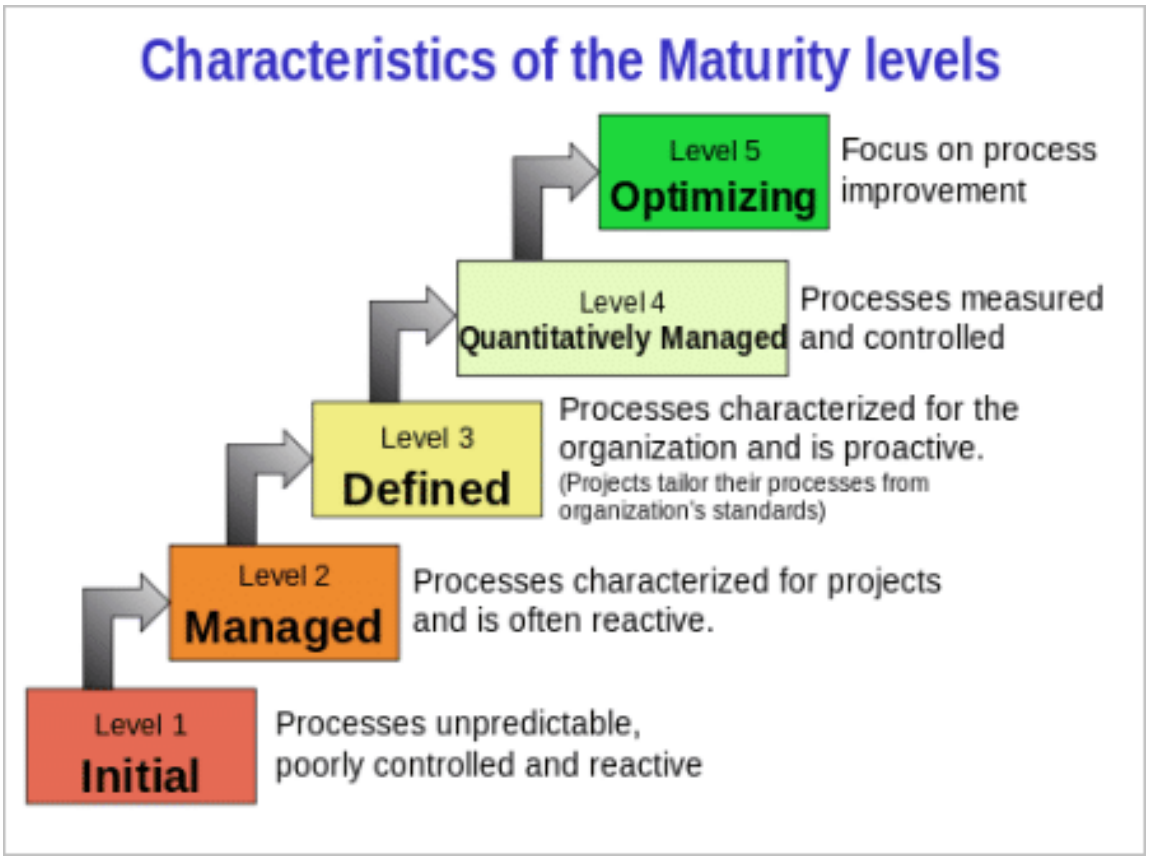


Рисунок 2.3 – П’ять рівнів CMMI

Інтеграція з моделлю зрілості тестування (TMMi): ця модель, заснована на CMMi, спрямована на рівні зрілості в тестуванні програмного забезпечення та управлінні якістю. Організація має більше можливостей виробляти високоякісну продукцію з меншою кількістю дефектів і відповідати вимогам бізнесу, коли зростає рівень зрілості. Рисунок 2.4 показує п’ять рівнів TMMi.

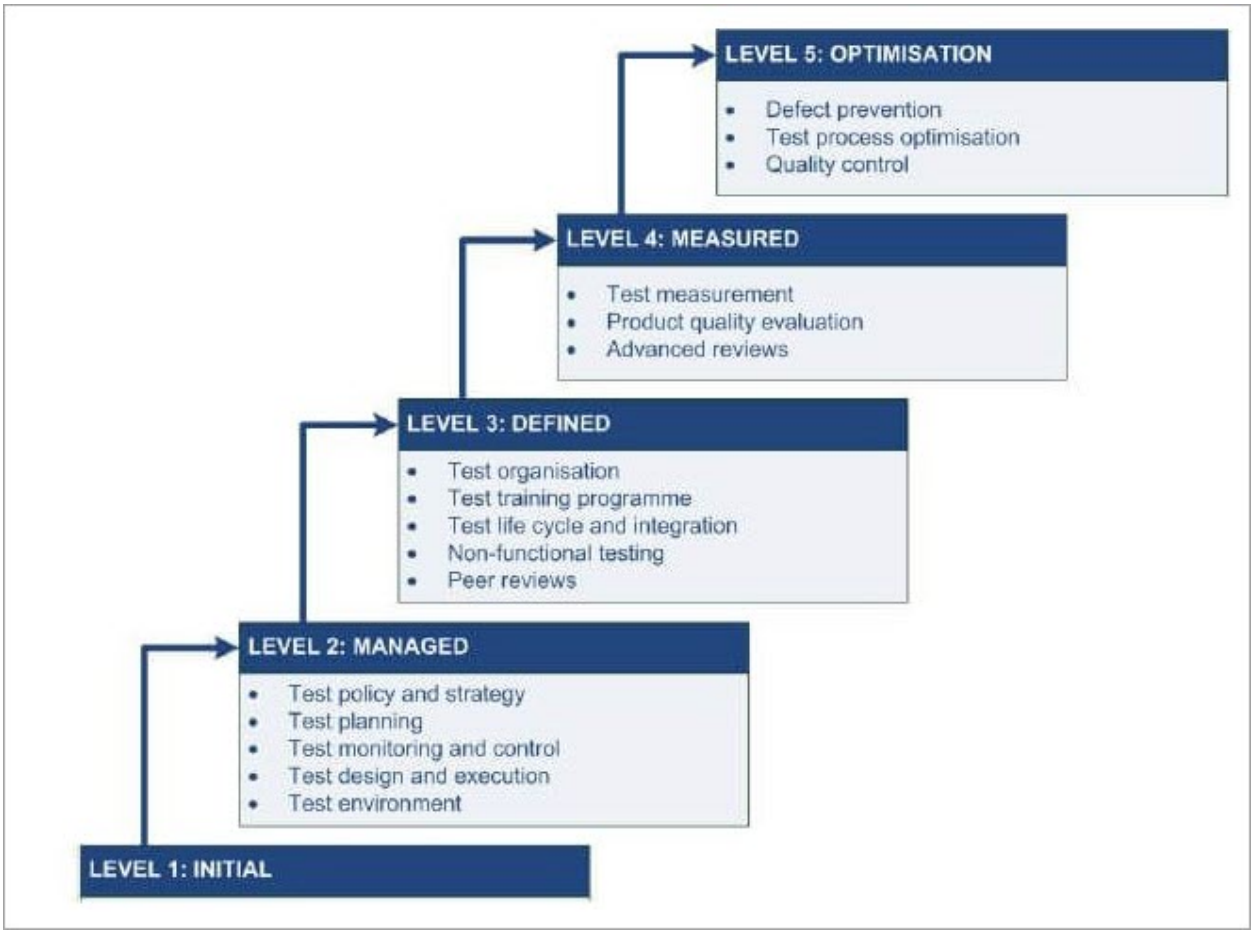


Рисунок 2.4 – Рівні TMMi

Стандарти оцінювання якості включають [23] аудит, інспекцію коду, перевірку дизайну, моделювання, функціональне тестування, стандартизацію, покрокове керівництво, модульне тестування, стрес-тестування.

Аудит – це перевірка робочих продуктів і пов'язаної з ними інформації, щоб визначити, чи дотримуються стандартні процедури. Рецензування – процес, під час якого внутрішні та зовнішні зацікавлені сторони розглядають програмний продукт для отримання їхніх коментарів і схвалення.

Інспекція коду – це найбільш формальний тип перевірки, при якому проводиться статичне тестування для пошуку помилок і запобігання виявлення помилок на більш пізніх стадіях. Це створюється навченим посередником або рівнем і базується на правилах, контрольних списках,

критеріях входу та виходу. Рецензент не обов’язково повинен бути автором коду.

Перевірка дизайну проводиться за допомогою контрольного списку, який перевіряє загальні вимоги та такі характеристики:

- конструкція програмного забезпечення;
- функціональні та інтерфейсні функції;
- угоди;
- вивчення вимог;
- інтерфейси та структура;
- логіка;
- продуктивність;
- обробка та відновлення помилок;
- розширюваність, тестованість;
- зв’язок і єдність.

Моделювання використовує реальну ситуацію для віртуального вивчення поведінки системи. Симулятори є чудовою альтернативою пісочниці, якщо реальну систему неможливо протестувати безпосередньо.

Функціональне тестування — це спосіб контролю за якістю, який перевіряє роботу системи без огляду на те, як вона її виконує. Тестування «чорної скриньки» зосереджено на перевірці специфікацій або функцій системи.

Стандартизація є важливою частиною гарантії якості. Це забезпечує якість завдяки зменшенню сумнівів і припущень.

Статичний аналіз — це аналіз програмного забезпечення, який виконується автоматичним інструментом без використання програми. Деякі популярні методи статичного аналізу включають програмні метрики та реверс-інжиніринг. Нові команди використовують такі інструменти для статичного аналізу коду, як SonarCube і VeraCode.

Покрокове керівництво. Покроковий посібник із програмного забезпечення, також відомий як покроковий посібник, є експертною перевіркою, під час якої розробник допомагає членам команди розробників ознайомитися з продуктом, ставить запитання, пропонує альтернативи та робить коментарі щодо потенційних помилок, стандартних помилок або інших проблем.

Модульне тестування — це метод тестування білої скриньки, при якому повне покриття коду гарантується виконанням кожної незалежної гілки, шляху та умови хоча б один раз.

Стрес-тестування. Цей тип тестування проводиться для перевірки надійності системи, випробовуючи її під надзвичайним навантаженням, тобто поза нормальними умовами.

Метрика програмного забезпечення є стандартом, який визначає, наскільки програмна система або процес має певні властивості в інженерії та розробці програмного забезпечення.

Ці слова часто використовуються як синоніми, незважаючи на те, що метрики є функціями, а виміри — це числа, отримані шляхом використання метрик.

Практики та теоретики інформатика постійно працюють над застосуванням подібних підходів до розробки програмного забезпечення, оскільки всі науки потребують кількісних вимірювань.

Метою є отримання кількісної оцінки вимірювань, яка є об’єктивним, відтворювальним і підданим. Ці оцінки можуть бути дуже корисними для планування розкладу та бюджету, оцінки витрат, забезпечення якості, тестування, налагодження програмного забезпечення, оптимізації продуктивності програмного забезпечення та оптимізації розподілу завдань персоналу.

Щоб визначити якість програмного забезпечення, необхідно кількісно визначити, наскільки система або програмне забезпечення має необхідні функції [10].

Це можна досягти за допомогою кількісних або якісних ресурсів, або їх поєднання.

В обох випадках існує набір вимірних атрибутів, які, як правило, пов'язані та пов'язані з бажаною характеристикою.

Наприклад, атрибут переносимості означає кількість операторів у програмі, що залежать від мети.

Точніше кажучи, ці характеристики є «як», необхідні для включення «що» у вищезазначене визначення якості програмного забезпечення, коли використовується підхід «Розгортання функції якості» (рисунок 2.5).

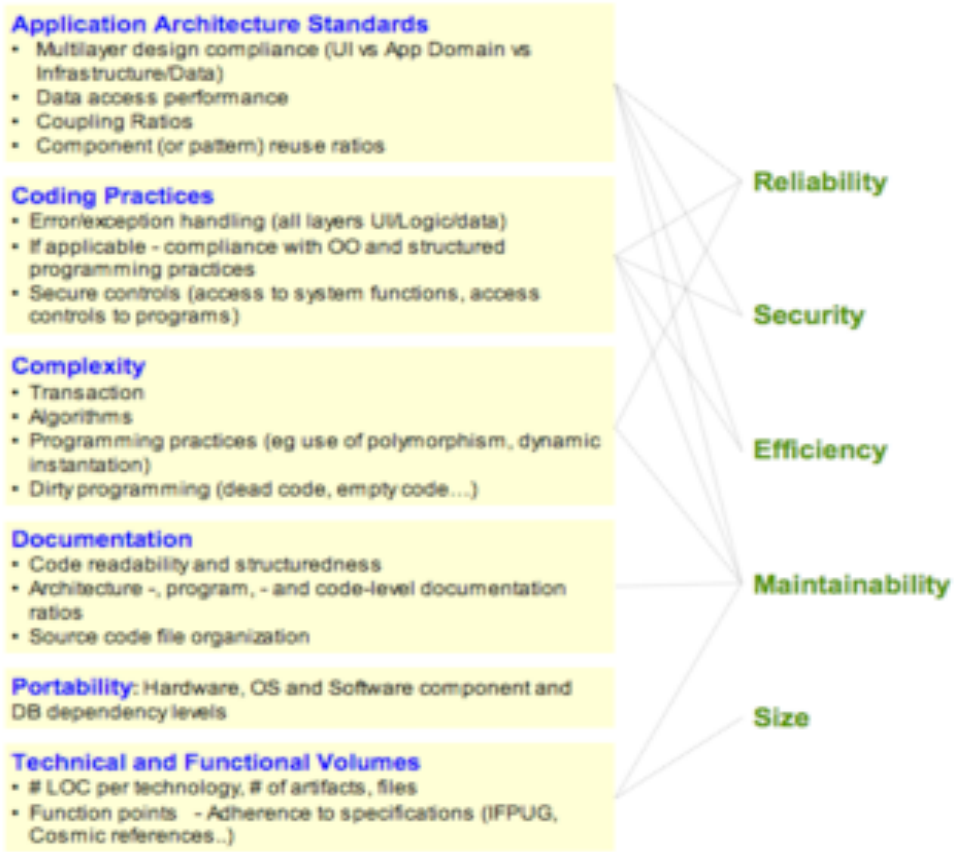


Рисунок 2.5 – Взаємозв'язок між бажаними характеристиками програмного забезпечення (праворуч) та вимірними атрибутами (ліворуч)

Ефективний процес вимірювання складається з наступних елементів [17]: чітко визначити проблеми розробки програмного забезпечення та елементів даних, необхідних для розуміння цих проблем; обробляти зібрані дані для допомоги в аналізі проблем; аналізувати показники, щоб зрозуміти проблеми розвитку; і використовувати результати аналізу для вдосконалення процесу та виявлення нових проблем і проблем (рисунок 2.6).

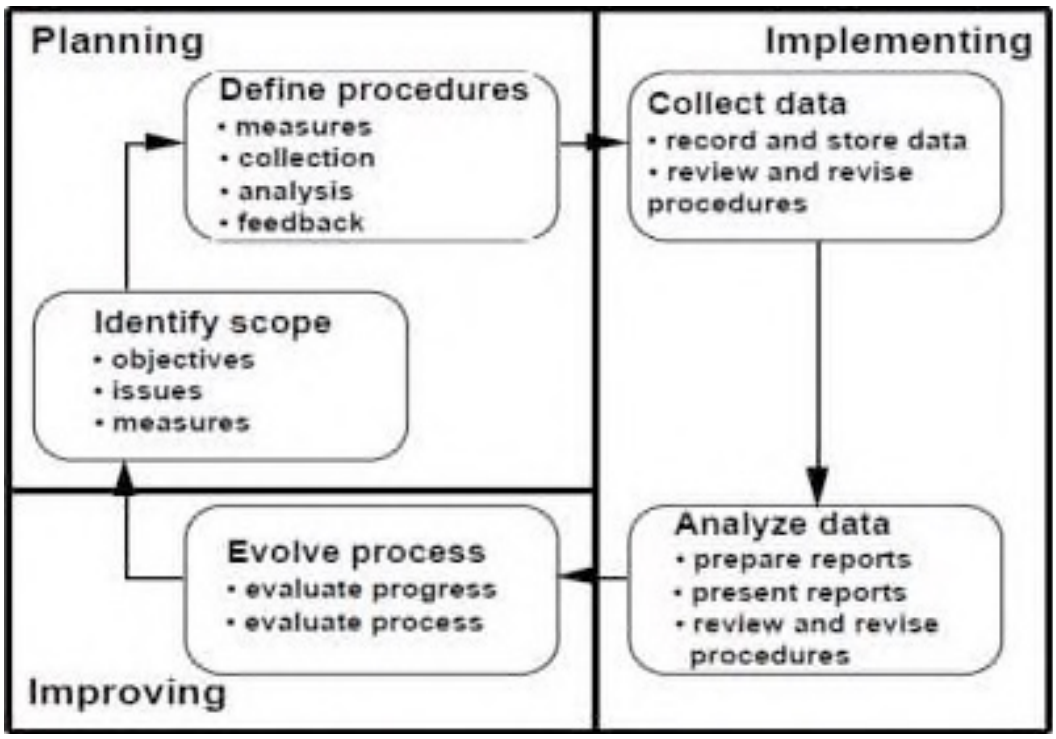


Рисунок 2.6 – Алгоритм вимірювання ПЗ

Для проектування процесу вимірювання з використанням цієї архітектури необхідно виконати наступні завдання [18]:

– створення процесу вимірювання, який повинен бути доступним як частина стандартного програмного процесу організації;

– планування процесу за проектами та документування процедур шляхом зміни та адаптації активу процесу;

– здійснення процесу на проектах шляхом виконання планів і процедур;

– удосконалення процесу шляхом створення планів і процедур, коли проекти розвиваються та змінюють свої вимірювальні потреби.

З точки зору метричних вимірювань, вони можуть бути класифіковані таким чином:

– пряме вимірювання атрибуту суб’єкта господарювання, який не залежить від будь-яких інших атрибутів чи сутності. Пряме вимірювання – це оцінка того, що вже є. Наприклад, кількість рядків коду. Непряме або похідне вимірювання означає використання математичної моделі для обчислення інших атрибутів або сутностей. Це завжди включає обчислення принаймні двох показників. Наприклад, щільність дефекту = ні. дефектів програмного продукту/загальний розмір продукту;

– система прогнозування складається з математичної моделі та набором процедур прогнозування для пошуку невідомих параметрів і інтерпретації результатів. Наприклад, якість програмного забезпечення.

Часто показники програмного забезпечення класифікують у більш широкому розумінні як показники процесу [4]:

– показники процесу включають показники процесу розробки програмного забезпечення, такі як середній рівень досвіду програмістів, тип методології та загальний час розробки. Композитні моделі, емпіричні, статистичні, теоретичні та інші варіанти є доступними;

– показники продукту включають показники програмного продукту на кожному етапі розвитку, починаючи з вимог до встановленої системи. Метаметри продукту можуть включати складність дизайну програмного забезпечення, розмір кінцевої програми (вихідного або об’єктного коду) або кількість створених сторінок документації. Вони часто класифікуються за розміром, складністю, якістю та залежністю даних.

Крім того, показники програмного забезпечення можна класифікувати таким чином:

– об’єктивні метрики гарантують, що для даної метрики завжди будуть однакові значення, виміряні двома або більше кваліфікованими спостерігачами;

– суб’єктивні метрики, з іншого боку, можуть вимірювати різні значення для даної метрики, оскільки їхні суб’єктивні судження беруть участь у вимірюванні.

Що стосується показників продукту, розмір товару, виміряний у рядках коду (LOC), є об’єктивним показником, який дозволяє будь-якому поінформованому спостерігачу, який використовує однакове визначення LOC, отримати однакову величину для конкретної програми.

Класифікація програмного забезпечення як «органічне», «напіввід’єднане» або «вбудоване» відповідно до вимог моделі оцінки витрат COCOMO є прикладом суб’єктивної метрики продукту.

Незважаючи на те, що програмні показники можна легко класифікувати як примітивні об’єктивні показники товару або примітивні суб’єктивні показники продукту тощо, цей модуль не дотримується стандартів цієї організації.

Програма повинна відповідати наступним вимогам [5]:

1) мета метрики. Кожен показник повинен мати чітку назву;

2) опис. Кожен показник повинен містити коротку характеристику;

3) умови межі. Числа, порівняні з результатом під час аналізу вихідного коду;

4) виділення. Результати вимірювання порівнюються з межами, а високі результати відзначені червоним або зеленим кольором, що означає, що вони відповідають межам чи ні;

5) Розширюваність. Розширення програми дозволить завантажувати метрики до існуючої програми (рисунок 2.7).

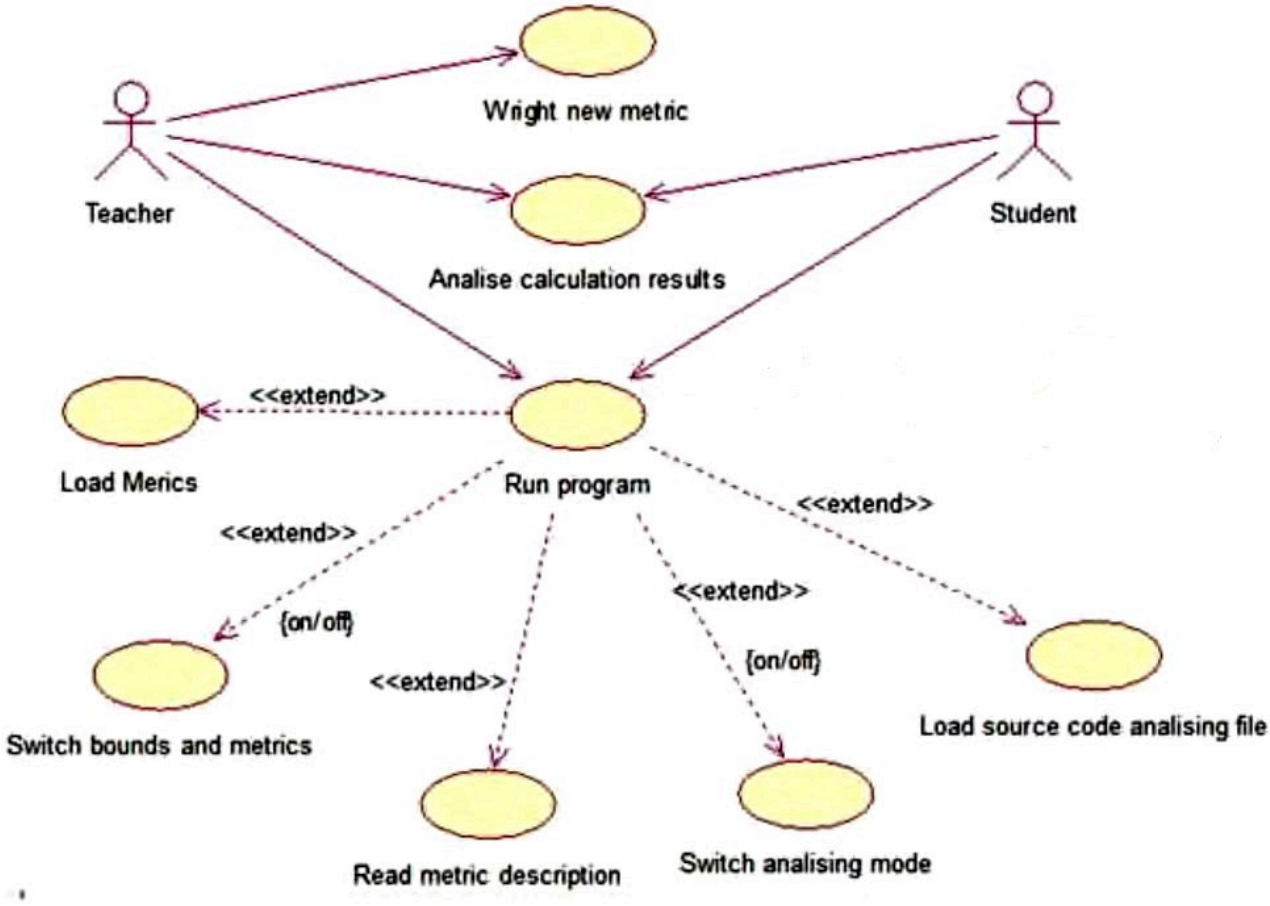


Рисунок 2.7 – Діаграма випадків використання програмного застосунку

Таким чином, користувач може розробити метрики та внести їх до програми. Під час роботи програми користувач може маніпулювати нею, щоб отримати розраховані показники.

2.2. Вимірювання надійності характеристик програмного забезпечення

Система, яка була розроблена, повинна мати можливість вимірювати метричні характеристики вихідного коду. Крім того, вона повинна розширюватися, щоб кожен міг писати метрику та включати її в інтерфейс програм. Таким чином, лише невелика кількість метрик буде вбудована в нову систему, а додаткові метрики можна буде додати під час роботи системи [2].

Мета системи полягає в тому, щоб вивчити код користувача з точки зору метрик і безпеки коду навчання. Крім того, ви зможете вимкнути режим навчання та деякі показники.

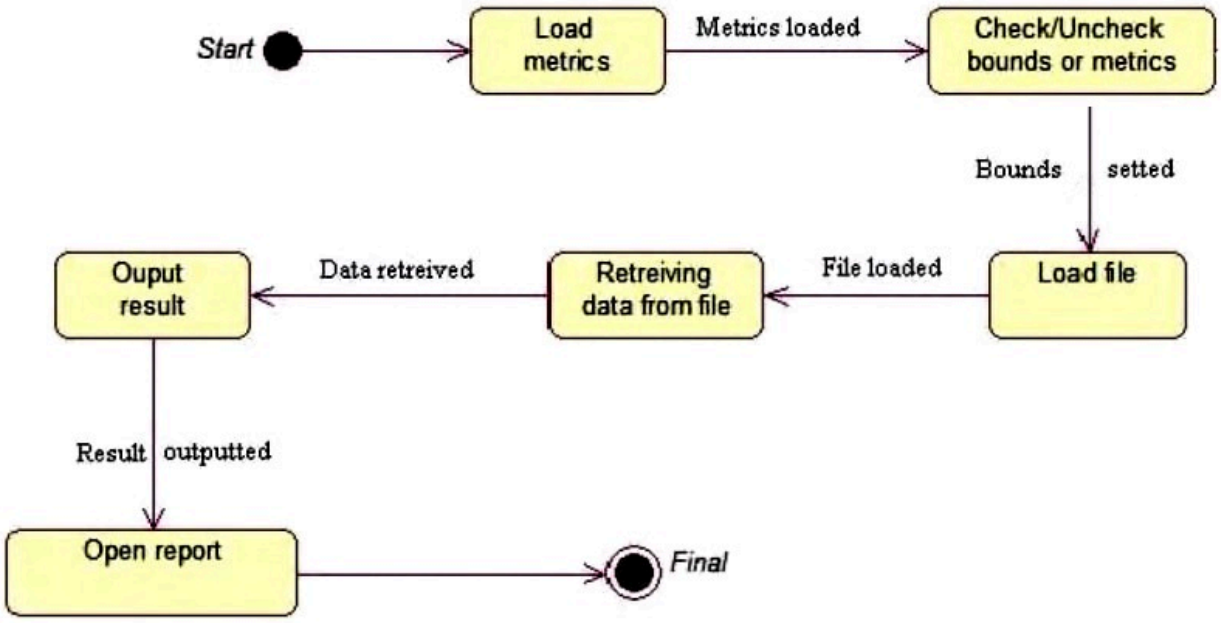


Рисунок 2.8 – Діаграма станів програми

Діаграма стану показує етапи роботи програми. Першим кроком є завантаження метрик.

Для аналізу вихідного коду завантажте файл.

Після вибору файл буде проаналізований, і програма видасть результат, який можна переглянути в звіті.

Варто зазначити, що більшість сьогоднішнього програмного забезпечення обчислювальних систем є бібліотеками та API, а не програмними продуктами. Ці продукти включають не тільки бібліотеки стандартних класів, такі як STL або Collection Framework, але й ядра операційних систем, наприклад.

Таким чином, величезна кількість програмного забезпечення, створеного до нього, все ще використовується розробником, навіть якщо він працює над ним поодинці. Більшість бібліотек програмного забезпечення створюється під час роботи в команді, і ці бібліотеки використовуються розробником або колегами.

Бібліотеки, отримані користувачем, не так жорстко регулюються, як API ядра операційної системи, тому їх важко змінювати між версіями. Тим не менш, вимоги до якості проектування внутрішніх інтерфейсів ніхто не відміняв, і чим краще вони будуть розроблені, тим менше проблем буде при їх використанні. Крім того, багато програм мають зовнішні бібліотеки для розширення, які дуже схожі на API операційних систем і відомі як plug-in.

2.3. Парсери та лексичні аналізатори

Парсер є важливою складовою частиною процесу аналізу мови в області комп'ютерних наук. Він використовується для аналізу послідовності символів або токенів, що складають мову, і визначення синтаксичної структури цієї послідовності. Парсер перетворює вхідну послідовність символів на дерево синтаксичного розбору (або іншу структуру даних), яка відображає структуру вхідного рядка згідно з граматикою мови.

В залежності від виду парсера, є декілька підходів до аналізу мови, таких як LL (пряма ліва рекурсія), LR (пряма права рекурсія) та їх варіації. Вони використовують різні алгоритми та стратегії для синтаксичного аналізу мови.

Лексер, також відомий як лексичний аналізатор, є першим етапом компілятора або системи аналізу коду. Його основна функція полягає в розбитті вхідного вихідного коду на послідовність токенів [15]. Ось як зазвичай працює лексер.

Сканування: Лексер сканує вихідний код посимвольно, читаючи кожен символ і групуючи їх в значущі одиниці.

Пропуск пропусків та коментарів: Лексер ігнорує пропуски (такі як пробіли, табуляції та символи нового рядка) і пропускає коментарі (наприклад, однорядкові коментарі, починаючи з "//", або багаторядкові коментарі, заключені у "/″ *і* ″/").

Розпізнавання токенів: Лексер розпізнає шаблони вихідного коду та групує символи, щоб сформувати токени. Токени представляють основні будівельні блоки коду, такі як ідентифікатори, ключові слова, літерали (константи), оператори, розділові знаки та інше.

Класифікація токенів: Кожен токен має свій тип, такий як "ідентифікатор", "ключове слово", "літерал" або "оператор". Лексер класифікує токени згідно цих типів.

Генерація токенів: Якщо лексер знаходить токен, він генерує відповідний об'єкт або структуру токена, яка зазвичай містить інформацію, таку як тип токена та його значення (якщо таке є). Лексер переходить до наступного символу і повторює процес до того моменту, поки не просканує весь вихідний код.

Потік токенів: Лексер формує потік токенів, який слугує вхідними даними для наступного етапу компіляції або аналізу, такого як парсер.

Парсер - це компонент, який приймає потік токенів, створений лексером, і проводить докладний аналіз їх структури та взаємозв'язків згідно з визначеною граматикою або набором правил. Ось детальний опис операцій парсера.

Визначення граматики: Парсер ґрунтується на визначенні граматики, яка визначає правила синтаксису для аналізованої мови програмування або мови з обмеженим набором функцій. Граматика вказує, як різні токени можуть комбінуватися, щоб створити правильні вирази, оператори та інші мовні конструкції.

Синтаксичний аналіз: Парсер виконує синтаксичний аналіз, щоб визначити, чи відповідає послідовність токенів граматичним правилам. Він

аналізує їх порядок, структуру та взаємозв'язки, щоб перевірити їх правильність.

Дерево розбору / Абстрактне синтаксичне дерево (AST): Якщо токени відповідають граматичним правилам, парсер будує дерево розбору або абстрактне синтаксичне дерево (AST). Дерево розбору / AST відображає ієрархічну структуру коду, відображаючи взаємозв'язки між різними мовними конструкціями (наприклад, оператори, вирази, функційні виклики та інше). Дерево розбору / AST слугує проміжним представленням коду, яке можна використовувати для подальшого аналізу або генерації коду.

Виявлення помилок: Якщо токени не відповідають граматичним правилам, парсер виявляє синтаксичні помилки і може надавати інформативні повідомлення про помилки для відлагодження та виправлення.

Семантичний аналіз (опціонально): Окрім синтаксичного аналізу, деякі парсери також виконують семантичний аналіз. Це включає перевірку семантичної коректності поза синтаксисом. Наприклад, перевірка, що змінні оголошені перед використанням, перевірка сумісності типів та дотримання мовних правил. Проміжне представлення: Парсер може створити проміжне представлення коду на основі дерева розбору / AST. Проміжне представлення (IR) є більш структурованим та абстрактним представленням, яке може використовуватися для різних оптимізацій або перетворень перед генерацією остаточного виконавчого коду.

Подальша компіляція або аналіз: Вихідні дані парсера, такі як дерево розбору / AST або IR, використовуються як вхідні дані для наступних етапів компіляції або аналізу, таких як оптимізація, генерація коду, статичний аналіз або будь-яка інша обробка, необхідна для конкретної мети системи.

2.4. Проектування системи для оцінки якості програмного коду

Успіх програми на ринку зазвичай залежить від нездатності компанії виконати низку конкретних запитів через обмежені ресурси. В результаті така

бізнес-модель зазвичай не враховує функції, які використовує підмножина клієнтів. Компанії зазвичай надають API, або програмний інтерфейс програми, щоб клієнти могли найняти розробників програмного забезпечення, щоб вони могли розширювати свою програму за допомогою функціональних можливостей, унікальних для клієнтів. Як тільки компанія запускає додаток за допомогою API, воно стає предметно-орієнтованою платформою, яка дозволяє стороннім розробникам створювати додаткові збірки або мости для більш важливих додатків [16].

Розроблений аналізатор є потужним інструментом для аналізу програмного коду. Його структура розроблена таким чином, що додавання нових мов для аналізу вимагає лише незначних модифікацій. Це досягається за допомогою використання інтерфейсу та перерахування, що представляє мови, які програма може аналізувати.

Схема системи для оцінки якості програмного коду користувача була створена в результаті процесу проектування, як показано на рисунку 2.9.

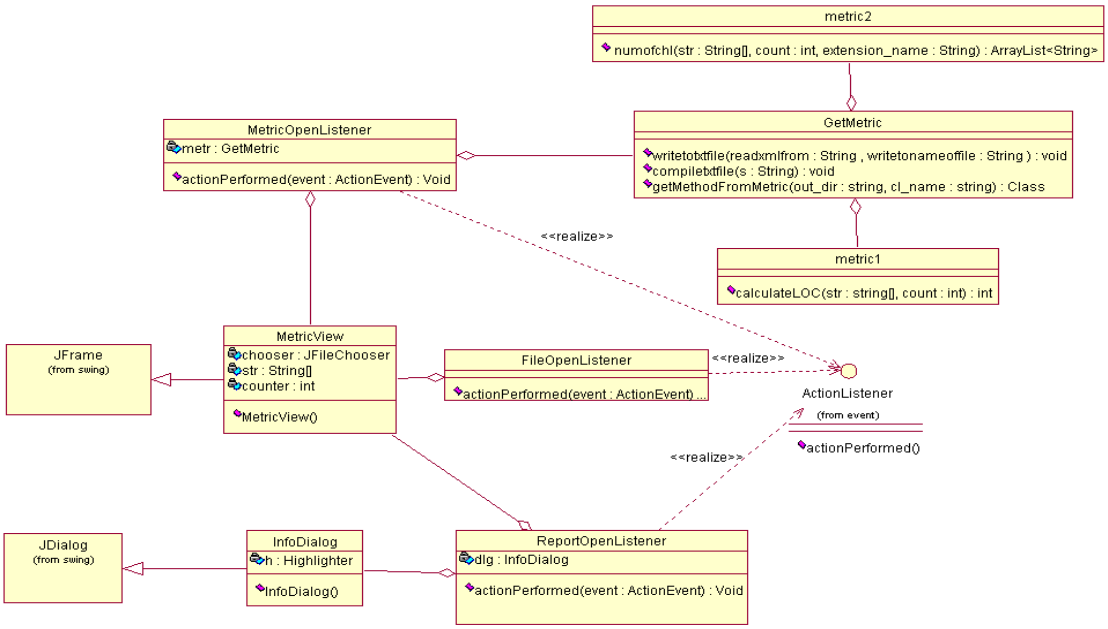


Рисунок 2.9 - Діаграма класів системи для оцінки якості програмного коду користувача

Ми використовуємо записи для зберігання всіх даних, які ми отримуємо з файлів коду. Вони також мають метод для виведення структури класів, що були проаналізовані. Ми також маємо деякі публічні перерахування, які представляють різні лексеми, які можуть повертати лексичний аналізатор.

Робота програми схожа на перші кроки компілятора. Першим кроком є лексичний аналізатор, який приймає весь вхідний код і повертає лексеми. У нашому випадку лексичний аналізатор працює як ітератор з одним визначеним методом next. При виклику цього методу ми переглядаємо рядок, який містить код, пропускаючи будь-які пробіли та коментарі. Потім виконуємо парсинг.

Таблиця 1.1 – Алгоритм роботи лексичного аналізатору

Символ	Дія
Пробіли	Пропускаємо
/	Перевіряємо наступний символ, пропускаючи його до кінця рядка або до */ у випадку коментаря, або повертаємо токен DIVIDE.
Букви або _	Конструюємо рядок з будь-яких буквено-цифрових символів або _. Потім робимо пошук в хеш-таблиці, щоб перевірити, чи наступне слово є ключовим словом. Якщо так, повертаємо токен KEYWORD, інакше повертаємо токен IDENTIFIER
Число	Виконуємо парсинг число до кінця послідовності цифр, плюс одну нецифрову, щоб дозволити, наприклад, 10f як число з плаваючою точкою.
'	Зчитуємо один символ, або два, якщо перший символ - , і створюємо токен CHARACTER.
"	Зчитуємо лексеми до неквотованої " і повертаємо токен STRING.
Все інше	Проходить через оператор переключення, де можливо перевіряється наступний символ (у випадках, наприклад, ++ або +=) або миттєво повертається токен.

Одну лексему та повертаємо її в кодовому вигляді, закодовану за допомогою перерахування токенів. Лексичний аналізатор має просту структуру, переміщуючись символ за символом. Цей ітератор використовується парсером і зберігається у ньому. Парсер набагато складніший, хоча його вихід простіший, ніж повноцінне синтаксичне дерево AST (Abstract syntax tree).

Парсер пройде через лексичний аналізатор, доки не залишиться більше токенів, додаючи до виводу на основі токенів і контексту. Основні етапи роботи парсера наступні: Перевірка поточного токену. Застосування модифікаторів, якщо вони є (наприклад, static, final і т. д.). Застосування модифікаторів доступу, якщо вони є (наприклад, public, private і т. д.). Якщо токен - class, enum або interface, виконуємо парсинг клас. Отримання наступного токену, якщо не кінець файлу.

Парсинг класу: Отримання імені класу. Перевірка наявності extends або implements. Якщо це enum, виконуємо парсинг всі варіанти enum. Повторюємо цикл токенів до знаходження закриваючої фігурної дужки (RCURLY). Якщо токен - class, enum або interface, виконуємо парсинг вкладений клас. Якщо токен – тип, то виконуються описані нижче кроки.

Отримуємо ім'я. Перевіряємо наступний токен: Якщо це (-> виконуємо парсинг метод. В іншому випадку це змінна. Виконуємо парсинг метод: Оскільки нам не потрібні внутрішні дані для аналізу, ми можемо пропустити все тіло. Виконуємо парсинг аргументи методу. Перевіряємо, чи є тіло методу (для виведення його, якщо воно є, ми виводимо { ... }, в іншому випадку просто ;). Пропускаємо тіло, якщо є. Це головна частина парсера. Є ще кілька важливих функцій, таких як парсинг типів таким чином, що дозволяє загальні типи та невідомі типи (наприклад, T).

Це досягається за допомогою рекурсивного обходу типу, використовуючи < та > для збільшення та зменшення рівня рекурсії відповідно. Він також перевіряє наявність будь-якої кількості [] та одного ... в

кінці, що вказує на тип масиву. Інший важливий блок - це пропуск анотацій, що вимагає перевірки, чи має анотація тіло, і пропускується, оскільки вона не використовується в нашому аналізі.

Ця програма не використовує зовнішніх бібліотек, крім вбудованого JDK, де ми отримуємо базові типи, такі як List<T> і Map<T>. Для графічного інтерфейсу ми використовуємо Java Swing API, яке спрощує процес створення графічних меню з вбудованими компонентами, такими як вибір файлів.

Рефакторинг коду в комп'ютерному програмуванні та розробці програмного забезпечення — це процес зміни структури комп'ютерного коду шляхом факторингу, при цьому зберігаючи його зовнішню поведінку. Рефакторинг — це процес покращення дизайну, структури та/або реалізації програмного забезпечення (його нефункціональних частин), при цьому зберігаючи його функціональність [18].

Рефакторинг може мати багато переваг. Це може покращити читаність коду та знизити його складність; це також може покращити ремонтопридатність вихідного коду та створити простішу, зрозумілішу або більш виразну внутрішню архітектуру або об'єктну модель для поліпшення розширюваності.

Підвищення продуктивності — це ще одна потенційна мета рефакторингу, оскільки програмісти-інженери постійно стикаються з проблемою написання програм, які працюють швидше або потребують менше пам'яті.

Рефакторинг зазвичай включає кілька стандартизованих базових мікрорефакторингів, кожен із яких вносить невелику зміну у вихідний код комп'ютерної програми, яка або зберігає поведінку програмного забезпечення, або принаймні не змінює його відповідність функціональним вимогам.

Багато середовищ розробки пропонують автоматизовану підтримку виконання механічних компонентів цих основних рефакторингів.

Переваги рефакторингу поділяються на дві основні групи [4].

Ремонтопридатність. У зв’язку з тим, що вихідний код легко читати, а цілі автора легко зрозуміти, помилки легше виправляти. Це можна досягти, зв’язавши великі монолітні підпрограми до набору індивідуально коротких, добре названих методів, призначених для досягнення певного мети. Перенесення методу в більш відповідний клас або видалення оманюючих коментарів можуть допомогти досягти цього.

Розширюваність. Використання відомих шаблонів проектування та надання деякої гнучкости, якої раніше не було, полегшує розширення потенціалу програми.

Інженерія продуктивності може вирішити проблему, відому як роздування програмного забезпечення, яка виникає через традиційні методи розробки програмного забезпечення, які намагаються скоротити час розробки програми, а не час, необхідний для її запуску.

Використовуючи переваги паралельних процесорів і векторних блоків, інженерія продуктивності також може адаптувати програмне забезпечення до обладнання, на якому воно працює.

Зміна сигнатури – це найпоширеніший метод рефакторинга. Найбільш вживані методи рефакторинга:

- зміна сигнатури (change method signature);
- інкапсуляція поля (encapsulate field);
- виділення класу (extract class);
- виділення інтерфейсу (extract interface);
- виділення локальної змінної (extract local variable);
- виділення методу (extract method);
- генералізація типу (generalize type);
- вбудовування (inline);
- введення фабрики (introduce factory);

– введення параметра (introduce parameter);

– підйом методу (pull up method);

– спуск методу (push down method);

– перейменування методу (rename method);

– переміщення методу (move method);

– заміна умовного оператора поліморфізмом (replace conditional with polymorphism);

– заміна успадкування делегуванням (replace inheritance with delegation);

– заміна коду типу підкласами (replace type code with subclasses).

2.5. Java Swing API

Java Swing є набором бібліотек та інструментів, які дозволяють розробникам створювати графічний інтерфейс користувача (GUI) для програм, написаних на Java. Він надає широкий спектр компонентів і контролів, які можна використовувати для створення різноманітних вікон, кнопок, полів введення, таблиць, списків та багатьох інших елементів GUI.

Основні компоненти Swing API наведено нижче.

JFrame: Це основний контейнер для створення вікна програми. Він надає рамку, заголовок, панель меню та інші основні елементи управління вікном.

JPanel: Це контейнер, який можна використовувати для групування інших компонентів разом. Він дозволяє організувати компоненти в макеті, наприклад, розташування їх у вигляді сітки або в стековій формі.

JButton: Це кнопка, на яку можна натиснути. Вона виконує дії або запускає функціональність програми, коли користувач клікає на неї.

JLabel: Це текстовий елемент, який використовується для відображення нередагованого тексту або зображень на GUI.

JTextField: Це поле введення, в яке користувач може вводити текст або дані.

JCheckBox: Це елемент вибору, який має два стани: обраний і необраний.

JRadioButton: Це елемент вибору, який дозволяє користувачеві вибрати один з декількох варіантів.

JTable: Це компонент, який дозволяє відображати дані у вигляді табличної форми з рядками і стовпцями. Він підтримує сортування, фільтрацію та редагування даних.

JComboBox: Це комбінований список, який дозволяє користувачу вибрати один елемент із списку доступних варіантів.

JScrollPane: Це контейнер, який дозволяє прокручувати вміст в межах іншого контейнера, якщо його розмір перевищує доступний простір.

LayoutManager: Swing надає різні менеджери макету, такі як BorderLayout, GridLayout, FlowLayout, які допомагають керувати розташуванням компонентів на GUI.

Event Handling: Swing API надає механізм обробки подій, який дозволяє реагувати на дії користувача, такі як натискання кнопки, введення тексту, переміщення миші та інші події.

Це лише кілька основних компонентів та можливостей, які надає Java Swing API. Завдяки своїй гнучкості і розширюваності, Swing є потужним інструментом для створення візуальних інтерфейсів в програмах на Java.

2.6. Висновки до розділу 2

З другого розділу ми можемо зробити такі висновки:

1) досліджено метрику програмного забезпечення;
2) здійснено оцінку надійності функцій програмного забезпечення;
3) описується процес проектування системи, який використовується для оцінки якості програмного коду.

3 ПРАКТИЧНА РЕАЛІЗАЦІЯ СИСТЕМИ ДЛЯ ОЦІНКИ ЯКОСТІ ПРОГРАМНОГО КОДУ КОРИСТУВАЧА

3.1. Розробка структурної схеми та архітектури прототипу

Рисунок 3.1 показує структурну схему системи для оцінки якості програмного коду користувача.

Тепер ви можете включити обчислювальні показники. Наступним кроком є ввімкнення та вимкнення кнопок перевірки, які керують режимом вивчення, межами та підсвічуванням. Крім того, у вас є можливість ввімкнути/вимкнути всю метрику.

Наступним кроком є включення файлу вихідного коду для аналізу.

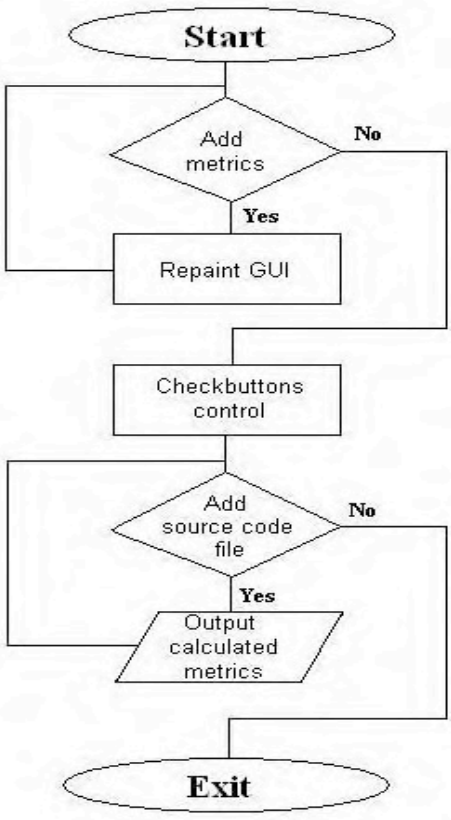


Рисунок 3.1 – Узагальнена схема алгоритму роботи створеної програмної системи

Далі запропонуємо реалізацію системи на прикладі створення метрики LOC (рисунок 3.2).

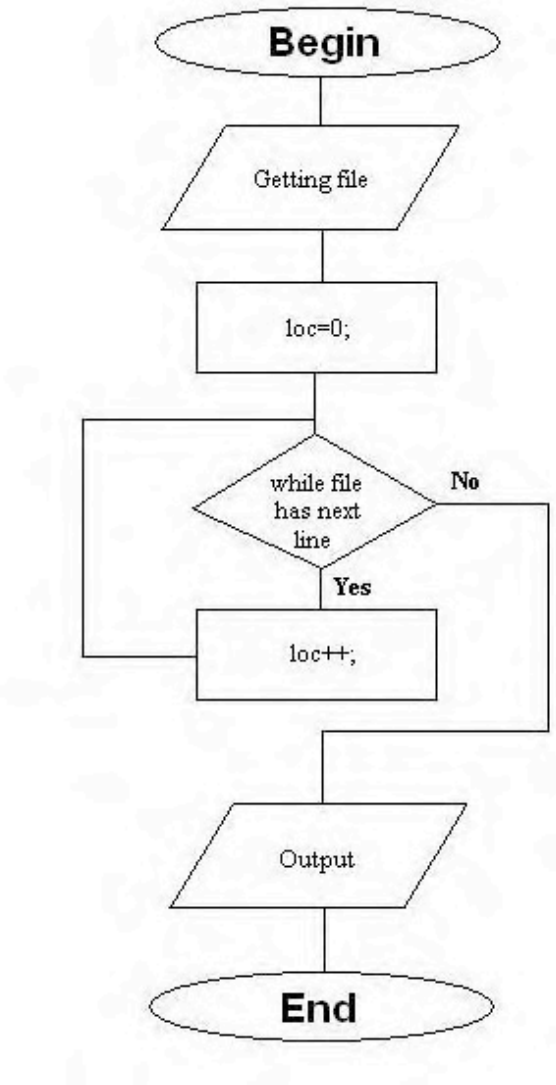


Рисунок 3.2 – Схема алгоритму розрахунку метрики LOC

3.2. Опис класів полів та методів

Пакунок `analyzer` (dev.mqxf.analyzer) містить весь код, необхідний для аналізу файлів коду. Це починається з інтерфейсу `Analyzer`. Він містить дві функції, одна з яких є абстрактною, а інша - за замовчуванням. Абстрактна функція має бути реалізована будь-якою реалізацією інтерфейсу аналізатора, тоді як функція за замовчуванням дозволяє користувачеві цього коду аналізувати будь-яку мову без необхідності шукати конкретну реалізацію

аналізатора, використовуючи перечислення enum `Language`. Хоча наразі цей код дозволяє аналізувати лише код на Java, ця функція дозволяє легко розширити його до більшої кількості мов.

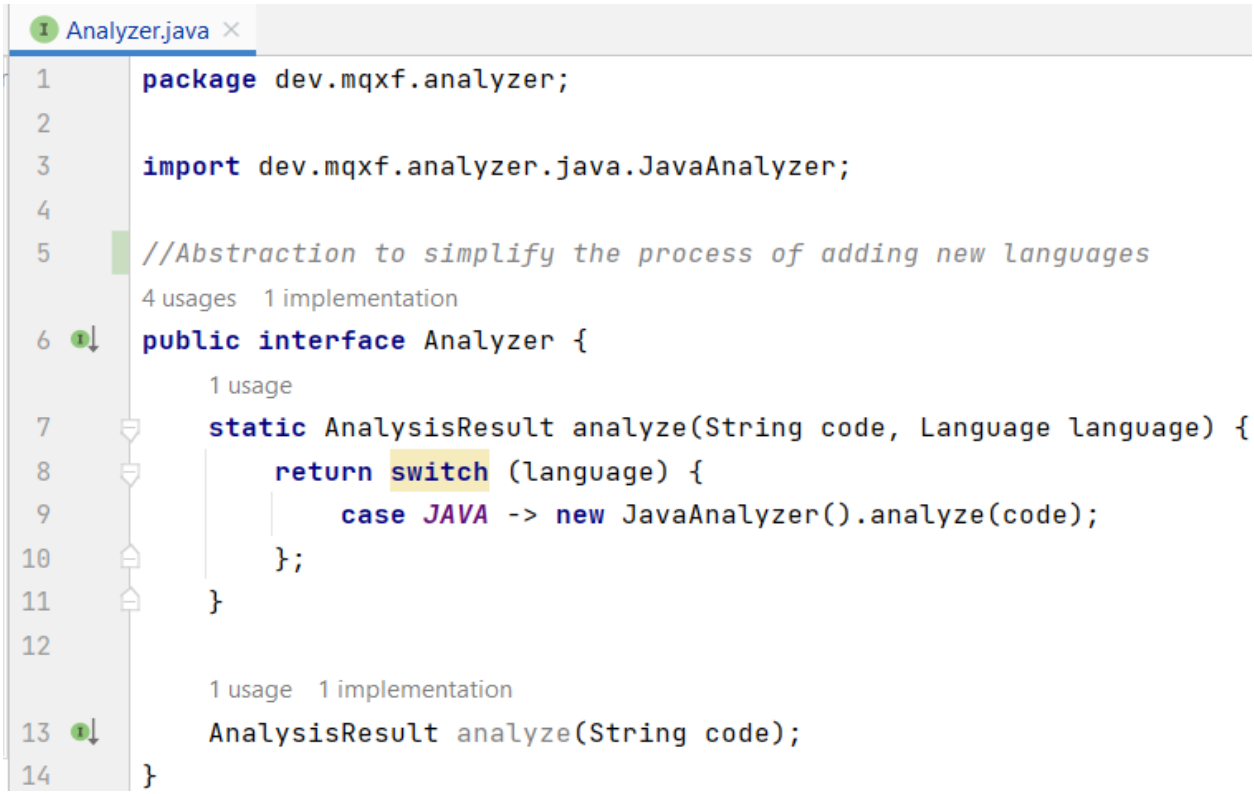


Рисунок 3.3 – Файл Analyzer.java в середі розробки

Інші перечислення enum, що містяться у пакунку `analyzer` - AccessModifier для таких ключових слів, як `public`, `private` і `protected`, ClassType для розрізнення `class`, `interface` і `enum` та Modifier для розбору `static`, `final`, `abstract` і `default`.

Нарешті, в пакеті `analyzer` знаходяться всі записи, які використовуються для зберігання результатів аналізу. Починаючи з найпростіших і піднімаючись вгору.

Поле: Представляє собою поле в класі, може мати будь-який модифікатор доступу і може бути `final` та/або `static`. Запис Field містить у собі 4 поля: ім'я, тип, модифікатор доступу та модифікатори. Ім'я та тип є

`String`, оскільки нам потрібно зберігати тип лише як рядок для відображення його користувачеві.

AccessModifier - це, очевидно, модифікатор доступу, і тільки один, оскільки ви не можете мати `public private String a`, оскільки це не має сенсу, а Modifiers - це `java.util.List<Modifier>`. Ми використовуємо `java.util.List` для зберігання декількох модифікаторів у списку динамічного розміру, оскільки ми можемо мати декілька модифікаторів, наприклад, `static final`, або жодного. У записі Field є два методи, обидва є методами типу toString. Один з них є методом перевизначення за замовчуванням, а інший генерує його з додатковими пробілами, щоб вкладені класи мали гарний вигляд у вікні перегляду структури класів.

Метод: Представляє метод всередині класу, може мати будь-який модифікатор доступу та будь-який звичайний модифікатор. Запис Method містить 7 полів: ім'я, параметри, тип повернення, модифікатор доступу, модифікатори, типи викидів та тіло. Ім'я - це `Рядок`, що представляє ідентифікатор. Параметри - це список рядків, кожен з яких зберігається як `type name` всередині `String`. Тут також використовується `java.util.List` для зберігання змінної кількості параметрів у списку динамічного розміру.

ReturnType – це рядок, що містить один тип, щоб дозволити відобразити його користувачеві. accessModifier і модифікатори такі ж, як і для поля, але модифікатори можуть також містити `за замовчуванням` і `абстрактний`, оскільки методи можуть бути за замовчуванням і абстрактними. ThrowsTypes – це список рядків, кожен з яких є просто типом, що представляє всі типи, які декларує функція, і які можуть бути викинуті зсередини функції. Наприклад, функція типу `public void foo() throws IOException { ... }` буде містити список з одним рядком `IOException`. Нарешті, hasBody - це булевий параметр, який вказує на те, чи є у функції тіло, чи ні, що дозволяє зробити акуратну особливість при відображенні функції користувачеві: функція, яка має тіло,

буде показуватися з закінченням `{ ... }`, а та, що не має, буде закінчуватись символом ';'.

Method.java

```
1   package dev.mqxf.analyzer;
2   import java.util.List;
    5 usages
3   public record Method(
        1 usage
4       String name,
        2 usages
5       List<String> parameters,
        1 usage
6       String returnType,
        3 usages
7       AccessModifier accessModifier,
        1 usage
8       List<Modifier> modifiers,
        2 usages
9       List<String> throwsTypes,
        1 usage
10      boolean hasBody
11  ) {
12
13      @Override
14  @  public String toString() { return toClassString( tab: ""); }
17
        2 usages
18  @   public String toClassString(String tab) {
19          StringBuilder builder = new StringBuilder(tab);
20
21          if (accessModifier != AccessModifier.PACKAGE) {
22              builder.append(accessModifier.toString().toLowerCase());
```

Рисунок 3.4 – Файл Method.java в середі розробки

ClassStructure: Представляє собою клас. Він містить в собі 10 полів: className, accessModifier, модифікатори, методи, поля, classType, subClasses, extendedClass, implementedInterfaces та enumValues. ClassName – це `String`, accessModifier та модифікатори такі самі, як і для Field та Method. Поля, методи та підкласи є `java.util.List` їхніх елементів-представників. Оскільки

клас може містити в собі поля, методи та інші класи, ми повинні використовувати списки з динамічним розміром, щоб показати, що насправді містить клас. Це дозволяє відображати порожні класи та класи з іншими класами всередині. ExtendedClass та ImplementedInterfaces – це `String` та `List<String>` відповідно, щоб показати, який клас розширено і які інтерфейси реалізовано. Оскільки ви можете розширити лише один клас, extendedClass може бути просто `String`, будучи нульовим, якщо жоден клас не розширено. Тим часом, реалізовані інтерфейси – це `java.util.List`, оскільки ви можете розширити будь-яку кількість інтерфейсів. `enumValues` – це список `String`, що представляє кожне значення Enum, доступне для використання, тільки якщо клас має тип `enum`. Цей список дозволяє нам показати, які значення доступні всередині перечислень.

AnalysisResult: Це повний результат аналізу. Він містить все, що було знайдено під час аналізу. Він має 6 полів: classStructures, classWord, interfaceWord, maxDepth, publicClassVariables і commentPercent. classStructures – це список `java.util.List` структур класів, що представляє всі класи верхнього рівня, знайдені в аналізі. ClassWord і interfaceWord дозволяють нам відображати інформацію користувачеві правильною мовою. Це потрібно для перенесення на інші мови. Наприклад, classWord для java – це `class`, а для rust – `struct`. MaxDepth - це цілочисельна змінна, що представляє максимальну глибину вкладеності коду, яка рахується, коли ми пропускаємо тіло методу. publicClassVariables – це список змінних, які є повністю загальнодоступними, це означає, що вони знаходяться в загальнодоступних класах і самі є загальнодоступними. Це може допомогти знайти змінні, які можуть бути вразливими до зміни зловмисником. Нарешті, commentPercent – це число з плаваючою комою, яке показує, яку частину коду складають коментарі.

У пакеті `java` (dev.mqxf.analyzer.java) написано аналізатор для коду на мові Java. Він містить 4 класи `Analyzer`, `Lexer`, `Parser` та `Token`. Клас `Token` - це запис, який має деякі внутрішні перечислення для представлення

різних типів токенів, які можуть з'являтися в Java. Аналізатор є простою обгорткою навколо лексера і синтаксичного аналізатора, яка об'єднує їх в одну функцію.

Лектор працює як ітератор, надаючи новий токен з методу `next`, він має 4 приватні змінні: char, яка зберігає весь код, який він аналізує, length, яка є довжиною коду, index, яка зберігає поточний індекс, щоб дозволити обхід коду, і comment, яка зберігає, скільки символів було всередині коментаря.

Синтаксичний аналізатор `Parser` повертає весь результат аналізу від лексера. Він має приватну змінну, що представляє поточну лексему, і лексер.

Рисунок 3.5 – Файл JavaParser.java в середі розробки

Це дозволяє декільком функціям мати спільний доступ до одного лексера і лексеми. Вона також має приватну змінну, що представляє максимальну глибину вкладеності, яка буде призначена лише у тому випадку, якщо вона може бути більшою за попередню максимальну глибину. Він імпортує `java.util.ArrayList`, як реалізацію `java.util.List`, який

використовується для зберігання будь-яких змінних, що мають бути динамічно розміщеними списками у результатах аналізу.

3.3. Висновки до розділу

Проаналізувавши предметну область, були розглянуті такі теоретичні аспекти, необхідні для розробки аналізатора коду:

– принципи та положення модульного та об'єктно-орієнтованого програмування;

– патерни проектування;

– основи статичного аналізу коду;

– правила та рекомендації щодо якості коду;

– засоби автоматизованого аналізу;

– аналіз та виявлення потенційних помилок та недоліків у програмному коді;

– розробка алгоритмів та методів аналізу програмного коду;

– реалізація системи аналізатора коду з використанням обраного програмного середовища;

– виявлення потенційних проблем у програмному коді, таких як синтаксичні помилки, недотримання стандартів програмування, поганий стиль коду, вразливості безпеки тощо;

– забезпечення зручного інтерфейсу для візуалізації та звітування результатів аналізу коду;

– розробка модулів для автоматизованого тестування та оцінки покриття коду.

4 ВПРОВАДЖЕННЯ

4.1. Впровадження системи автоматизованих метрик

Тож запустимо спроектовану систему. При запуску програми з’являється вікно (рисунок 4.1). Одразу ми бачимо кнопку для завантаження файлу, нижче ми бачимо два перемикача “Show fields” та “Only show public members”.

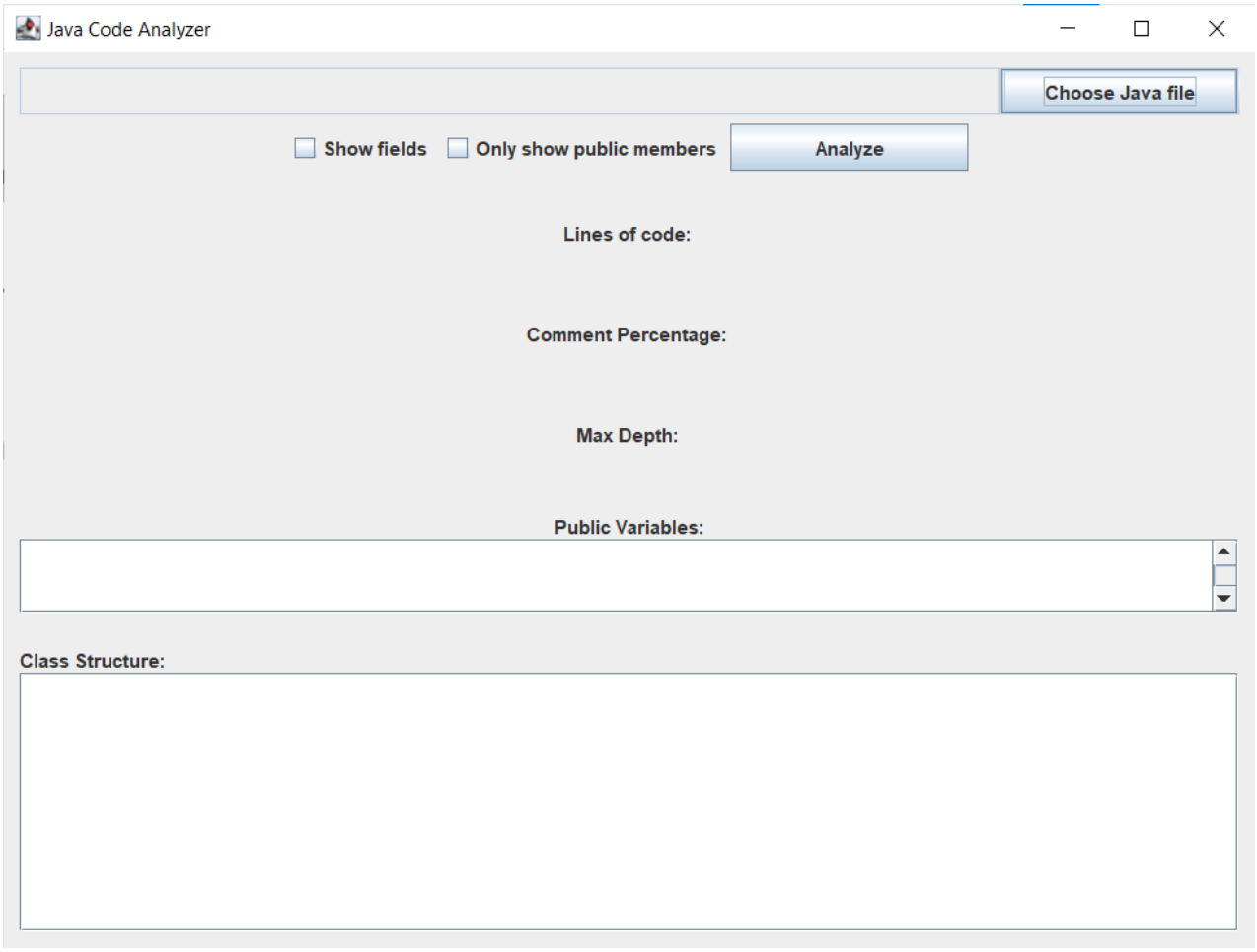


Рисунок 4.1 – Зовнішній вигляд інтерфейсу додатку

Перший перемикач “Show fields” при його активації демонструє всі поля класів обраного Java файлу у рамці “Class Structure”. Другий перемикач “Only show public members” допомагає сховати всі непублічні структури в класах

обраного файлу. Зміни можна побачити також у текстовому полі “Class Structure” після активації перемикача і натиснення кнопки “Analyze”.

Нижче перемикачів ми можемо бачити строку “Lines of code”, вона відображає результати однойменної метрики або LOC, тобто кількість строчок обраного для аналізу файлу.

Наступним ми бачимо строку “Comment Percentage”, яка аналізує який процент строк коду має в собі коментарі.

Тож спробуємо додати файл Java для аналізу (рисунок 4.2 - 4.3).

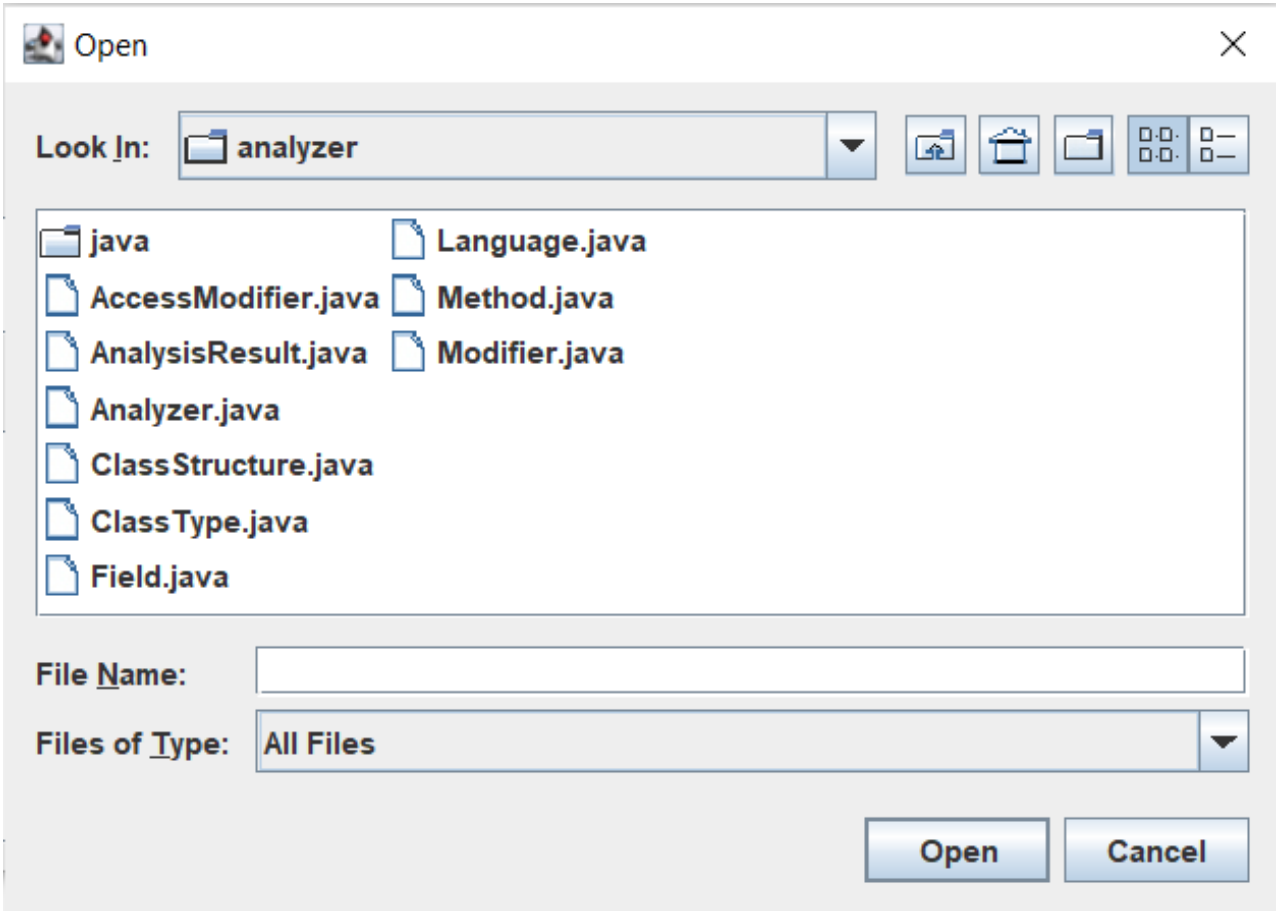


Рисунок 4.2 – Вікно вибору файлу для аналізу

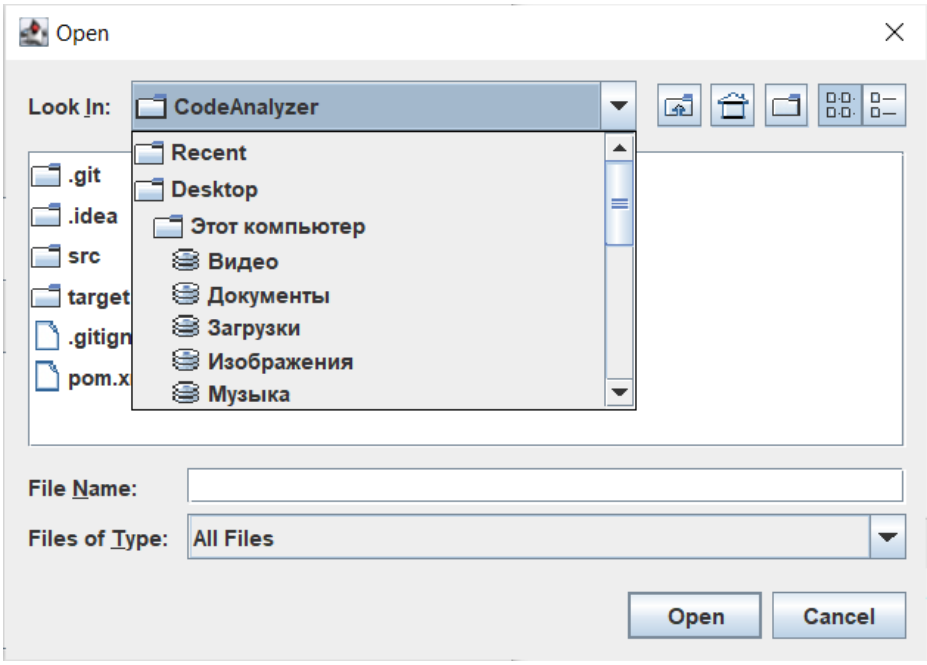


Рисунок 4.3 – Вибір директорії

Файл обраний і доданий до аналізатору (рисунок 4.4 – 4.5).

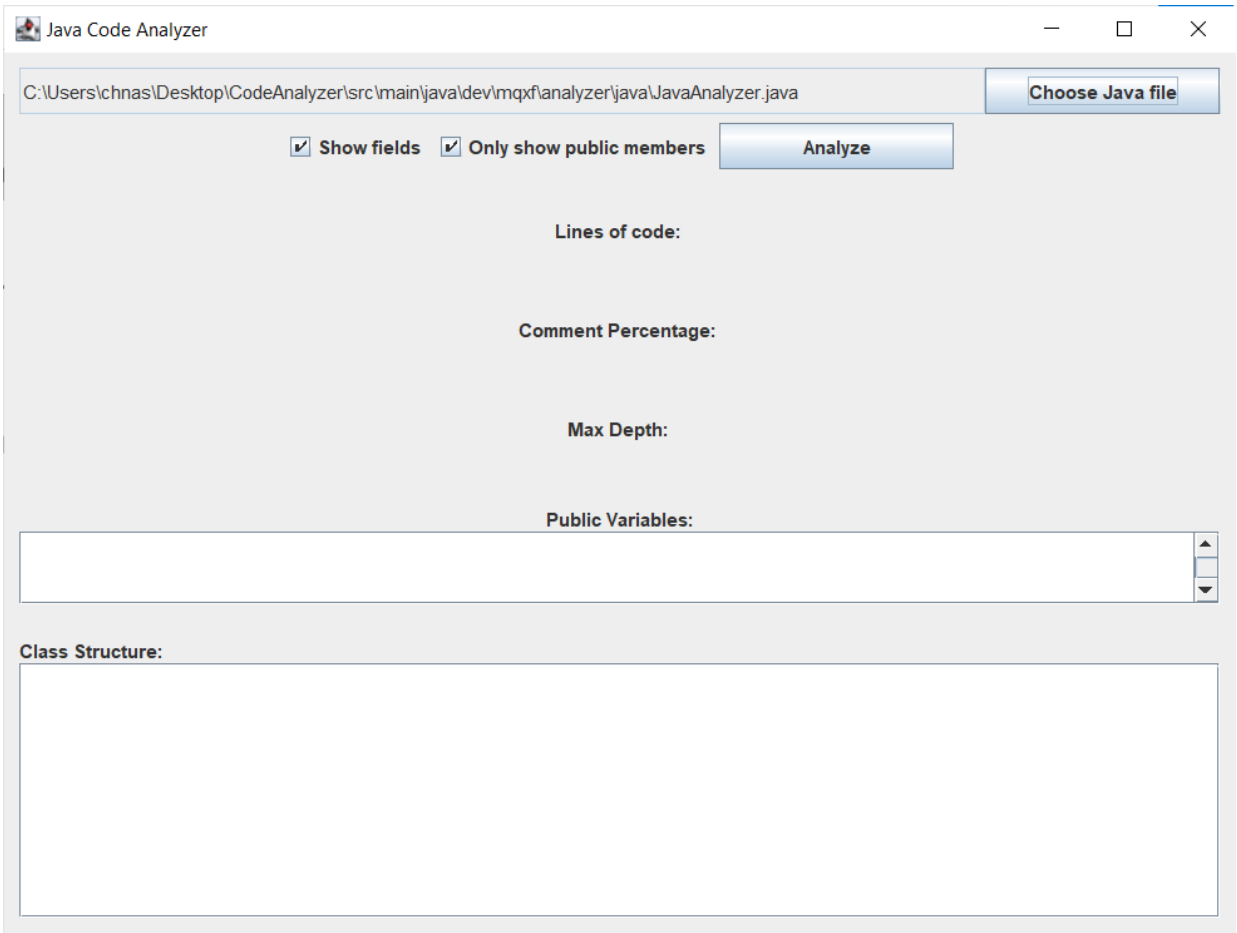


Рисунок 4.4 – Додавання файлу

Після вибору файлу для аналізу натискаємо на кнопку “Analyze”, після чого виконується аналіз, результати якого ми одразу можемо бачити у відповідних полях.

Також ми бачимо строку “Max Depth”, яка відображає вкладеність або рекурсивність коду, що аналізується. Нижче поле можемо бачити “Public Variables“, в якому демонструються всі змінні публічного ступеню доступності файлу, який аналізується. “Class Structure” відображає структуру файлу, тобто всі його складові частини: класи, функції і так далі.

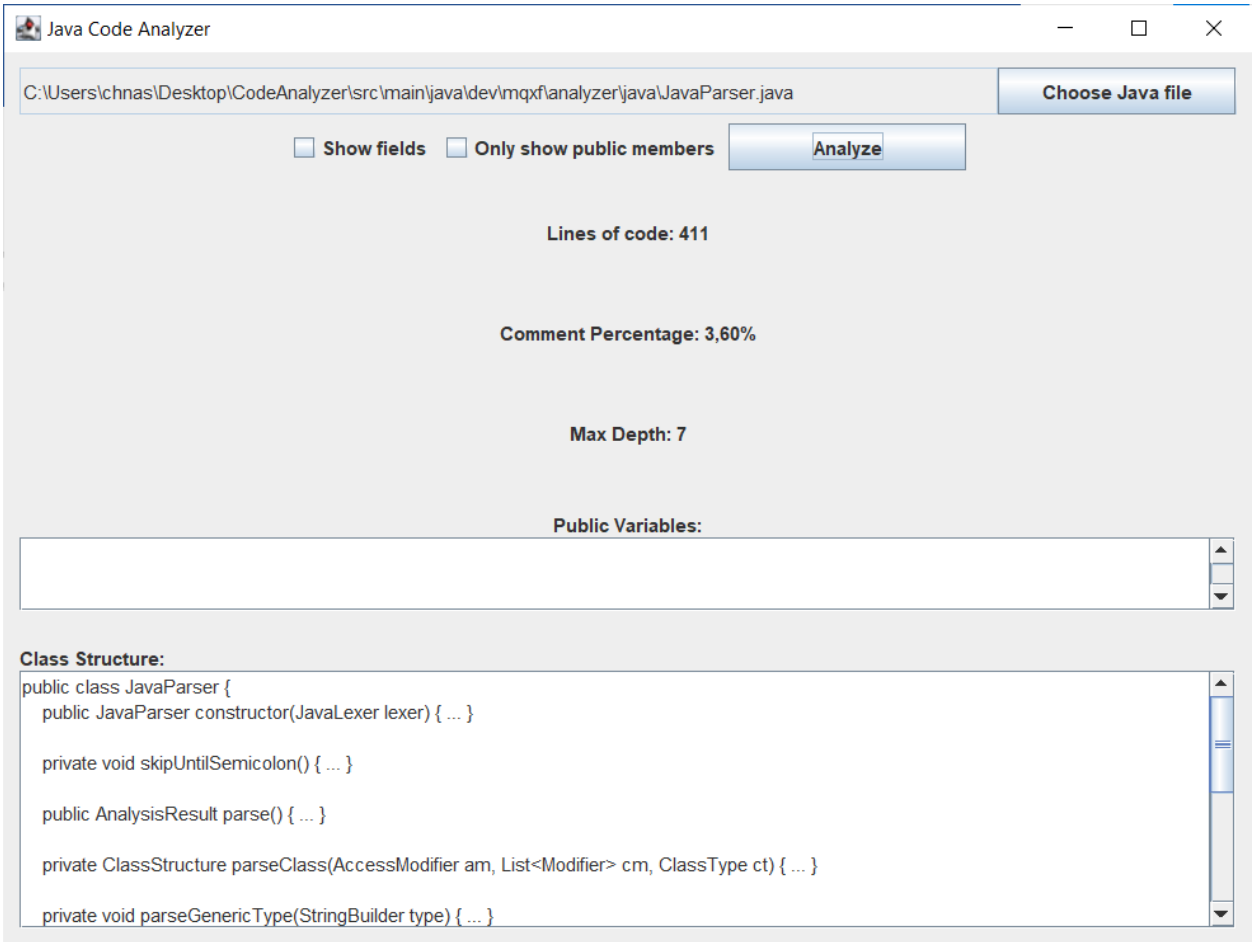


Рисунок 4.5 – Робочий функціонал розробленої системи для аналізу програмного коду користувача

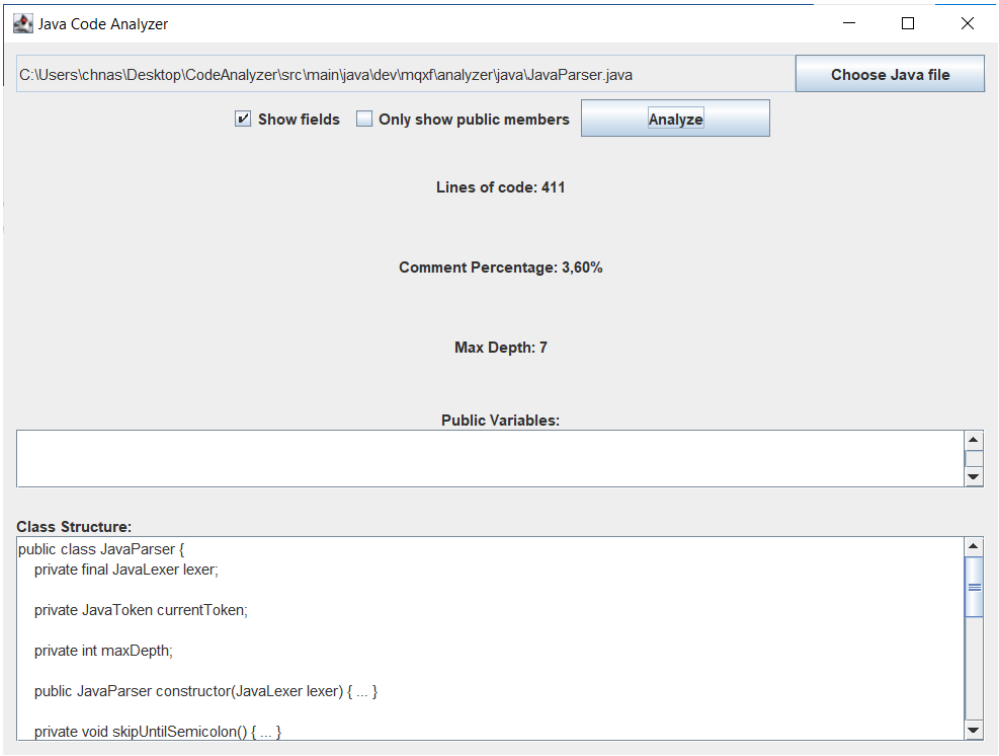


Рисунок 4.6 – Результат роботи при ввімкненому “Show fields”

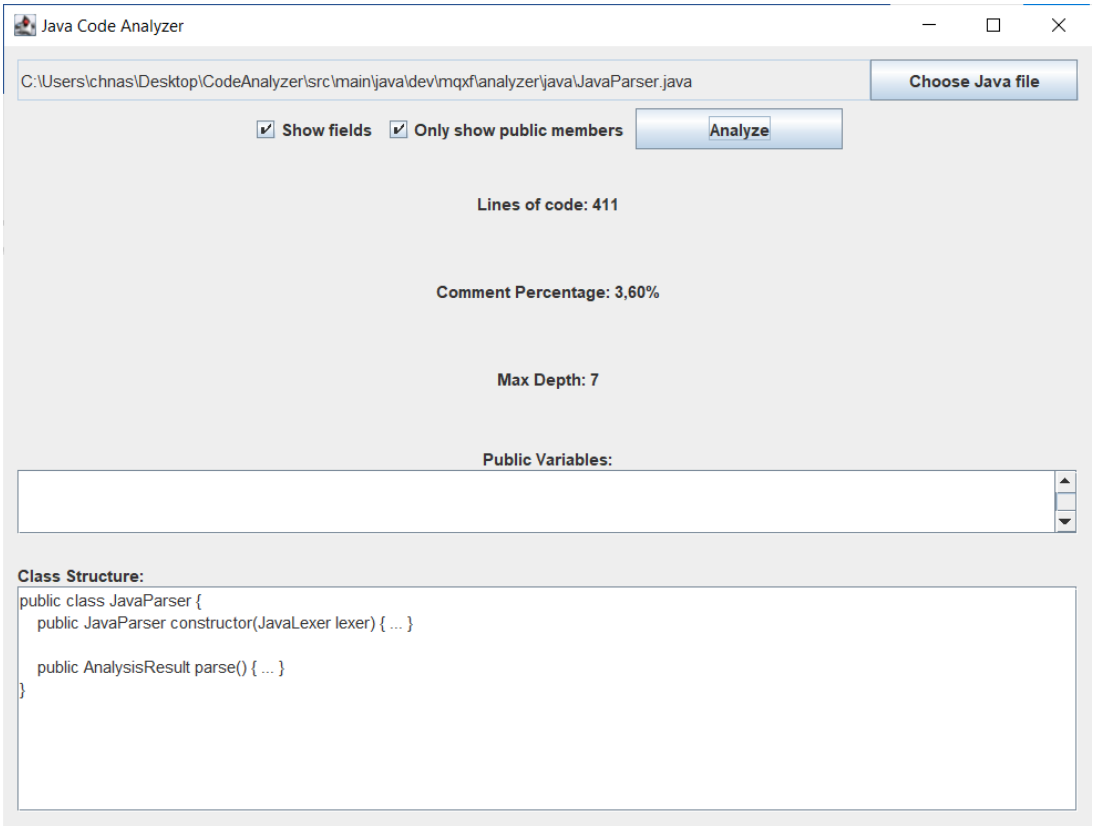


Рисунок 4.7 – Результат роботи при ввімкнених “Show fields” та “Only show public members”

4.2. Аналіз ефективності розробленої системи

Порівняємо розроблену систему з аналогами програмного забезпечення, які вже існують.

Проаналізуємо наступне:

– програму метричних вимірювань;

– розробку автоматизованої системи вимірювань.

Крім того, ми проведемо конкурентний аналіз розробленої системи порівняно з існуючими альтернативами.

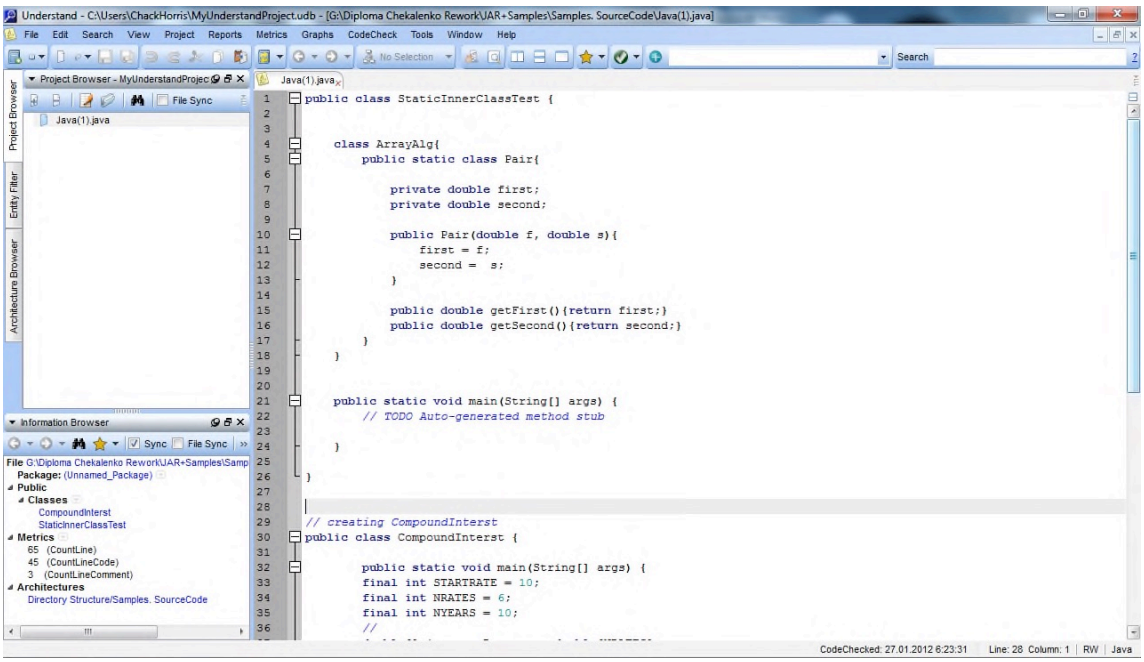


Рисунок 4.8 - Аналізатор коду

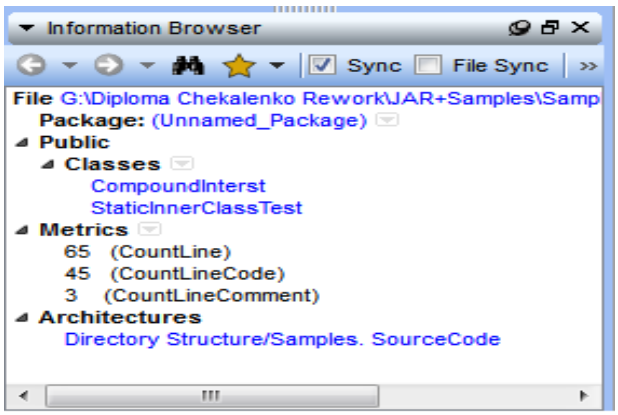


Рисунок 4.9 – Вікно метрик

Як показано, загальна кількість показників обмежена.

Порівняно з розробленим додатком, він має обмежену кількість метрик і не може бути розширеним.

У нього немає інших варіантів ніяких інших метрик.

Окрім того, що стосується навчального процесу, він не може показати безліч меж і має безліч варіантів, які не корисні для навчального процесу.

4.3. Висновки до розділу 4

Підводячи підсумок, можна сказати наступне щодо загального змісту роботи:

1. «Поганий або гарний код» не пов’язаний з ефективністю використання обчислювальних ресурсів (швидкість програми, обсяг оперативної пам’яті), а з налагодженням і зміною програмного забезпечення на етапах власної розробки та супроводу. Якість коду в цьому випадку оцінюється за допомогою таких критеріїв, як правильне розбиття програми на модулі, обмеження використання потенційно небезпечних мовних конструкцій і чітке оформлення вихідного коду. Створення універсальних критеріїв, які дозволяють визначити, чи є програма надійною, є надзвичайно складним завданням, навіть якщо ви визначите, що складають ключові показники якості коду. Як би там не було, ця оцінка надто індивідуальна, і вона може відрізнятися від одного програміста до іншого в залежності від типу проекту. Отже, існуючі методи визначення метрик коду в основному обмежуються обчисленням відповідних значень, які повністю залежать від людини.

2. Процес забезпечення якості програмного забезпечення (SQA) – це процес, який гарантує, що всі процеси, методи, дії та робочі компоненти, які використовуються для розробки програмного забезпечення, були відстежені та відповідають встановленим стандартам. Це можуть бути стандарти, такі як ISO 9000, модель CMMI та ISO15504. Тестування якості програмного

забезпечення (SQA) включає всі етапи розробки програмного забезпечення, починаючи з визначення вимог і закінчуючи кодуванням і випуском. Якість є його головною метою. План забезпечення якості програмного забезпечення, який також називають SQAP, складається з процедур, методів і інструментів, що використовуються для гарантування, що продукт або послуга відповідає вимогам, визначеним у SRS (специфікація вимог до програмного забезпечення). Метрика програмного забезпечення є стандартом, який визначає, наскільки програмна система або процес має певні властивості в інженерії та розробці програмного забезпечення.

3. Ефективний процес вимірювання складається з таких частин:

– чітко визначити проблеми розробки програмного забезпечення та елементів даних, необхідних для розуміння цих проблем;

– обробляти зібрані дані для допомоги в аналізі проблем;

– аналізувати показники, щоб зрозуміти проблеми розвитку;

– використовувати результати аналізу для вдосконалення процесу та виявлення нових проблем.

4. Компанії зазвичай надають API, або програмний інтерфейс програми, щоб клієнти могли найняти розробників програмного забезпечення, щоб вони могли розширювати свою програму за допомогою функціональних можливостей, унікальних для клієнтів. Як тільки компанія запускає додаток за допомогою API, воно перетворюється на предметно-орієнтовану платформу, яка дозволяє стороннім розробникам створювати додаткові збірки або мости для інших додатків, які мають велике значення для підмножини клієнтів.

5. Система, яка була розроблена, повинна мати можливість вимірювати метричні характеристики вихідного коду. Крім того, вона повинна розширюватися, щоб кожен міг писати метрику та включати її в інтерфейс програм. Таким чином, лише невелика кількість метрик буде вбудована в розроблювальну систему, а додаткові метрики можна буде додавати під час

роботи системи. Мета системи полягає в тому, щоб вивчити код користувача з точки зору метрик і безпеки коду навчання.

6. Перегляд поточних програмних аналізаторів метрик показує, що вони не потребують функцій навчального процесу. Програмне забезпечення, розроблене в цій роботі, аналізує всі переваги, недоліки та переваги. Серед них:

– багатомовність;
– легкість;
– відкритість джерела;
– розширюваність;
– простота інтерфейсу;
– багатофункціональність.

7. Для створення ефективних і високопродуктивних програм одним із важливих компонентів програмування є оптимізація коду та зменшення зайвого коду. Багато факторів впливають на якість вихідного коду; ці включають мову програмування, знання та досвід програміста та інші фактори. Як показав аналіз вихідного коду, значна частина якості програми залежить від дотримання програмних параметрів. Компілятори вихідного коду шукають лише помилки, а те, що відповідає метрикам, залишається за програмістом.

Підводячи підсумок третього розділу, ми можемо зробити такі висновки:

1) розроблено архітектуру прототипу та структурну схему;
2) впроваджено автоматизовану систему вимірювань метрик;
3) описано основні компоненті і класи програми;
4) проведено аналіз ефективності системи, щоб визначити якість програмного коду користувача.

ВИСНОВКИ

Під час виконання проекту зі створення аналізатора коду було проведено аналіз предметної області, надано обґрунтування обраної теми, визначену мету, об'єкт, предмет та методи дослідження. Також був проведений аналіз конкурентів.

В результаті виконання проекту зі створення аналізатора коду були досягнуті такі цілі:

- розроблені алгоритми та методи аналізу програмного коду;
- реалізована система аналізатора коду з використанням обраного програмного середовища;
- виявлені потенційні проблеми у програмному коді;
- забезпечено зручний інтерфейс для візуалізації та звітування результатів аналізу коду;
- розроблені модулі для автоматизованого тестування та оцінки покриття коду.

Після завершення аналітичної роботи була розроблена система аналізатора коду, включаючи вибір практичних інструментів та розробку архітектури системи.

Під час розробки аналізатора коду були описані особливості реалізації модулів системи, визначена структура бази даних та розроблена блок-схема алгоритму аналізу коду.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Антонюк В.С. Методологія наукових досліджень: [Текст] : навч. посіб./ В.С. Антонюк, Л.Г. Полонський, В.І. Аверченков, Ю.А. Малахов. К.: НТУУ «КПІ», 2015. 286 с.

2. Аллен Е. Типові помилки проектування. Бібліотека програміста / Е. Аллен. СПб., 2003. С. 22-29.

3. Бабак В.П. Інформаційна безпека та сучасні мережеві технології : Англо-українсько-російський словник термінів / В.П. Бабак, О. Г. Корченко. Київ : Издательство НАУ, 2003. С. 230-255.

4. Бек К. Екстремальне програмування / К. Бек. СПб., 2002. С. 3-17.

5. Блох Дж. Java™. Ефективне програування / Д. Блох. М. : Лорі, 2002. С. 44-49.

6. Вороновський Г. К., Махотило К. В., Петрашев С. Н., Сергєєв С. А. Генетичні алгоритми, штучні нейроні мережі і проблеми віртуальної реальності. - Замовне. Х.: ОСНОВА, 1997. С. 99-112.

7. Головач Ю. Складні мережі / Ю. Головач, О. Олємской, К. фон Фербер та ін. // Журнал фізичних досліджень. - 2006. 10, № 4. С. 247-289.

8. Дейч А. М. Методи ідентифікації динамічних об’єктів. М: Енергія, 1979. 240 с.

9. Поліщук Д. О. Комплексне детерміноване оцінювання складних ієрархічно-мережевих систем: І. Опис методики / Д. О. Поліщук, О. Д. Поліщук, М. С. Яджак // Системні дослідження та інформаційні технології. 2015. № 1. С. 21–31.

10. Ситник Л. Ю. Система оцінки якості програмного коду веб-проектів// Тези доповідей наук.-практ. конф. “Сучасні тенденції розвитку системного програмування” (25-26 листопада 2020 р.). К.: НАУ, 2020. С. 28.

11. Скулиш Є. Д. Засоби аналізу та оцінка ризиків інформаційної безпеки /Є. Д. Скулиш, О. Г. Корченко, Ю. І. Горбенко, С. В. Казмирчук // Інформаційна безпека. Людина, суспільство, держава 2011. №3 (7). – С. 31-48.

12. API. URL: https://ru.wikipedia. org/wiki/ API.

13. Asanovic, Krste et al. (Jan 18, 2016). The Landscape of Parallel Computing cResearch: A View from Berkeley University of California, Berkeley. Technical cReport No. UCB/EECS-2016-183.

14. Barbara Chapman, Gabriele Jost, Ruud van der Pas. Using OpenMP: portable shared memory parallel programming (Scientific and Engineering Computation). Cambridge, Massachusetts: The MIT Press., 2008. 353 pp.

15. Code Conventions for the Java™ Programming Language / URL: http://www.oracle.com/technetwork/java/index.html

16. Lexer and parser generators. URL:https://v2.ocaml.org/manual/lexyacc.html.

17. Hong S.G., Kim S.W. and Lee J.J., 2015. The Minimum Cost Path Finding Algorithm Using a Hopfield Type Neural Network, Proceedings IEEE International Conference on Fuzzy Systems 4. P.719–726.

18. Токен. URL: https://uk.wikipedia.org/wiki/Токен.

19. Рефакторинг. URL:https://uk.wikipedia.org/wiki/Рефакторинг.

20. Java Swing. URL:Режим доступу: https://www.javatpoint.com/java-swing.

Додаток А
Зауваження нормоконтролера

Таблиця А.1 – Зауваження нормоконтролера

Сторінка	Позначення	Зміст зауваження
С. 17, 26-27, 54-56		Невірні переліки
ПЗ		Нещільне заповнення сторінок, номери підрозділів,… мають зайву крапку
Розділ 4		Не на усі рисунки є посилання

Додаток Б.
Лістинг програмної розробки

Фрагмент основного візуального модуля

```
package dev.mqxf.gui;

import dev.mqxf.analyzer.AnalysisResult;
import dev.mqxf.analyzer.java.JavaLexer;
import dev.mqxf.analyzer.java.JavaParser;
import dev.mqxf.analyzer.Analyzer;
import dev.mqxf.analyzer.Language;

import javax.swing.*;
import java.awt.*;
import java.io.File;
import java.nio.file.Files;
import java.nio.file.Path;

public class GUI extends JFrame {
    private JTextField chosenFileField;
    private JButton chooseFileButton;
    private JButton analyzeButton;
    private JTextArea resultTextArea;
    private JScrollPane scrollPane;
    private JLabel commentPercentLabel;
    private JLabel maxDepthLabel;
    private JLabel linesLabel;
    private JTextArea publicVarsTextArea;
    private JScrollPane publicVarsScrollPane;
    private JCheckBox fields;
    private JCheckBox onlyPublic;

    public GUI() {
        setTitle("Java Code Analyzer");
        setSize(800, 600);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setLayout(new BorderLayout());

        JPanel fileChooserPanel = new JPanel(new BorderLayout());
        fileChooserPanel.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10)); // Adding
padding

        chosenFileField = new JTextField();
        chosenFileField.setEditable(false);
        chooseFileButton = new JButton("Choose Java file");
        chooseFileButton.setPreferredSize(new Dimension(150, 30));
        chooseFileButton.setMaximumSize(new Dimension(150, 30));
```

```
        chooseFileButton.addActionListener(e -> {
            JFileChooser fileChooser = new JFileChooser();
            int returnValue = fileChooser.showOpenDialog(null);
            if (returnValue == JFileChooser.APPROVE_OPTION) {
                File selectedFile = fileChooser.getSelectedFile();
                chosenFileField.setText(selectedFile.getAbsolutePath());
            }
        });
        fileChooserPanel.add(chosenFileField, BorderLayout.CENTER);
        fileChooserPanel.add(chooseFileButton, BorderLayout.EAST);

        analyzeButton = new JButton("Analyze");
        analyzeButton.setPreferredSize(new Dimension(150, 30));
        analyzeButton.setMaximumSize(new Dimension(150, 30));
        analyzeButton.addActionListener(e -> {
            String filePath = chosenFileField.getText();
            if (filePath.isEmpty()) {
                JOptionPane.showMessageDialog(null, "Please choose a file first.");
                return;
            }
            try {
                AnalysisResult analysisResult = Analyzer.analyze(Files.readString(Path.of(filePath)),
Language.JAVA);
                resultTextArea.setText(analysisResult.toClassString(onlyPublic.isSelected(),
fields.isSelected()));
                commentPercentLabel.setText(String.format("Comment Percentage: %.2f%%",
analysisResult.commentPercent() * 100));
                maxDepthLabel.setText("Max Depth: " + analysisResult.maxDepth());
                linesLabel.setText("Lines of code: " + analysisResult.lines());
                publicVarsTextArea.setText(String.join("\n", analysisResult.publicClassVariables()));
            } catch (Exception ex) {
                ex.printStackTrace();
                resultTextArea.setText("Error while analyzing file: " + ex.getMessage());
            }
        });

        fields = new JCheckBox("Show fields");
        onlyPublic = new JCheckBox("Only show public members");

        JPanel buttonPanel = new JPanel();
        buttonPanel.add(fields);
        buttonPanel.add(onlyPublic);
        buttonPanel.add(analyzeButton);

        fileChooserPanel.add(buttonPanel, BorderLayout.SOUTH);

        JPanel resultPanel = new JPanel(new BorderLayout());
        resultPanel.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));
        JLabel resultLabel = new JLabel("Class Structure:");
        resultTextArea = new JTextArea();
```

```
        resultTextArea.setEditable(false);
        resultTextArea.setRows(10);
        scrollPane = new JScrollPane(resultTextArea);
        resultPanel.add(resultLabel, BorderLayout.NORTH);
        resultPanel.add(scrollPane, BorderLayout.CENTER);

        // Extra analysis result details
        JPanel extraDetailsPanel = new JPanel(new GridLayout(4, 1));
        extraDetailsPanel.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));

        JPanel linesPanel = new JPanel(new FlowLayout(FlowLayout.CENTER));
        linesLabel = new JLabel("Lines of code: ");
        linesPanel.add(linesLabel);
        extraDetailsPanel.add(linesPanel);

        JPanel commentPercentPanel = new JPanel(new FlowLayout(FlowLayout.CENTER));
        commentPercentLabel = new JLabel("Comment Percentage: ");
        commentPercentPanel.add(commentPercentLabel);
        extraDetailsPanel.add(commentPercentPanel);

        JPanel maxDepthPanel = new JPanel(new FlowLayout(FlowLayout.CENTER));
        maxDepthLabel = new JLabel("Max Depth: ");
        maxDepthPanel.add(maxDepthLabel);
        extraDetailsPanel.add(maxDepthPanel);

        JPanel publicVarsPanel = new JPanel(new BorderLayout());
        JLabel publicVarsLabel = new JLabel("Public Variables:");
        publicVarsLabel.setHorizontalAlignment(SwingConstants.CENTER);
        publicVarsTextArea = new JTextArea();
        publicVarsTextArea.setEditable(false);
        publicVarsTextArea.setRows(5);
        publicVarsScrollPane = new JScrollPane(publicVarsTextArea);
        publicVarsPanel.add(publicVarsLabel, BorderLayout.NORTH);
        publicVarsPanel.add(publicVarsScrollPane, BorderLayout.CENTER);
        extraDetailsPanel.add(publicVarsPanel);

        add(fileChooserPanel, BorderLayout.NORTH);
        add(extraDetailsPanel, BorderLayout.CENTER);
        add(resultPanel, BorderLayout.SOUTH);

        setVisible(true);
    }
}
```

Фрагмент модуля виведення результатів

```
    package dev.mqxf.analyzer;
    import java.util.List;
```

```
public record AnalysisResult(
    List<ClassStructure> classStructures,
    String classWord,
    String interfaceWord,
    int maxDepth,
    List<String> publicClassVariables,
    float commentPercent,
    int lines
) {

    public String toString() {
        StringBuilder builder = new StringBuilder();
        for (ClassStructure clazz : classStructures) {
            builder.append(clazz.toString()).append("\n");
        }
        return builder.toString().replaceAll(" class ", ' ' + classWord + ' ').replaceAll(" interface ", '
' + interfaceWord + ' ');
    }

    public String toClassString(boolean onlyPublic, boolean fields) {
        StringBuilder builder = new StringBuilder();
        for (ClassStructure clazz : classStructures) {
            builder.append(clazz.toClassString("", onlyPublic, fields)).append("\n");
        }
        return builder.toString().replaceAll(" class ", ' ' + classWord + ' ').replaceAll(" interface ", '
' + interfaceWord + ' ');
    }
}
```

Блок організації класів

```
package dev.mqxf.analyzer;
import java.util.List;

public record ClassStructure(
    String className,
    AccessModifier accessModifier,
    List<Modifier> modifiers,
    List<Method> methods,
    List<Field> fields,
    ClassType classType,
    List<ClassStructure> subClasses,
    String extendedClass,
    List<String> implementedInterfaces,
    List<String> enumValues
) {
    public void getPublicVariables(String path, List<String> buffer) {
        for (Field field : fields) {
            if (field.accessModifier() == AccessModifier.PUBLIC) {
```

```
            buffer.add(path + className + "." + field.name());
        }
    }
    for (ClassStructure clazz : subClasses) {
        if (clazz.accessModifier() == AccessModifier.PUBLIC) {
            clazz.getPublicVariables(path + className + ".", buffer);
        }
    }
}

@Override
public String toString() {
    return toClassString("", false, false);
}

public String toClassString(String tab, boolean onlyPublic, boolean showFields) {
    StringBuilder builder = new StringBuilder(tab);

    if (accessModifier != AccessModifier.PACKAGE) {
        builder.append(accessModifier.toString().toLowerCase());
        builder.append(' ');
    }

    for (Modifier modifier : modifiers) {
        builder.append(modifier.toString().toLowerCase());
        builder.append(' ');
    }

    builder.append(classType.toString().toLowerCase());
    builder.append(' ');
    builder.append(className);

    if (extendedClass != null) {
        builder.append(" extends ");
        builder.append(extendedClass);
    }

    if (implementedInterfaces.size() > 0) {
        builder.append(" implements ");
        for (String implementedInterface : implementedInterfaces) {
            builder.append(implementedInterface);
            builder.append(", ");
        }
        builder.delete(builder.length() - 2, builder.length());
    }

    builder.append(" {\n");

    if (classType == ClassType.ENUM) {
        for (String enumValue : enumValues) {
```

```
                builder.append(enumValue);
                builder.append(",\n");
            }
            builder.delete(builder.length() - 2, builder.length());
            builder.append(";\n");
        }

        if (showFields) {
            for (Field field : fields) {
                if (!onlyPublic || field.accessModifier() == AccessModifier.PUBLIC) {
                    builder.append(field.toClassString(tab + "    "));
                    builder.append('\n');
                }
            }
        }

        for (Method method : methods) {
            if (!onlyPublic || method.accessModifier() == AccessModifier.PUBLIC) {
                builder.append(method.toClassString(tab + "    "));
                builder.append('\n');
            }
        }

        if (methods.size() > 0 && subClasses.size() == 0) {
            builder.delete(builder.length() - 1, builder.length());
        }

        for (ClassStructure clazz : subClasses) {
            if (!onlyPublic || clazz.accessModifier() == AccessModifier.PUBLIC) {
                builder.append(clazz.toClassString(tab + "    ", onlyPublic, showFields));
                builder.append('\n');
            }
        }

        if (subClasses.size() > 0) {
            builder.delete(builder.length() - 1, builder.length());
        }

        builder.append(tab);
        builder.append("}\n");
        return builder.toString();
    }
}
```

Блок виконання парсингу

```
package dev.mqxf.analyzer.java;
import dev.mqxf.analyzer.*;

import java.util.ArrayList;
```

```
import java.util.List;

public class JavaParser {
   private final JavaLexer lexer;
   private JavaToken currentToken;
   private int maxDepth = 0;

   public JavaParser(JavaLexer lexer) {
      this.lexer = lexer;
      this.currentToken = lexer.next();
   }

   private void skipUntilSemicolon() {
      while (currentToken.type() != JavaToken.TokenType.SEMICOLON &&
currentToken.type() != JavaToken.TokenType.EOF) {
         currentToken = lexer.next();
      }
   }

   public AnalysisResult parse() {
      List<ClassStructure> classStructures = new ArrayList<>();
      List<String> publicClassVariables = new ArrayList<>();
      AccessModifier accessModifier = AccessModifier.PACKAGE;
      List<Modifier> modifiers = new ArrayList<>();
      StringBuilder packageName = null;

      //go to the end of the file
      while (currentToken.type() != JavaToken.TokenType.EOF) {
         //we don't care about annotations
         if (currentToken.type() == JavaToken.TokenType.ANNOTATION) skipAnnotation();
         if (currentToken.type() == JavaToken.TokenType.KEYWORD) {
            switch ((JavaToken.Keyword) currentToken.value()) {
               case PUBLIC, PRIVATE, PROTECTED ->
                  accessModifier =
AccessModifier.valueOf(currentToken.value().toString().toUpperCase());
               case STATIC, ABSTRACT, FINAL, DEFAULT ->

modifiers.add(Modifier.valueOf(currentToken.value().toString().toUpperCase()));
               case CLASS -> {
                  classStructures.add(parseClass(accessModifier, modifiers, ClassType.CLASS));
                  accessModifier = AccessModifier.PACKAGE;
                  modifiers = new ArrayList<>();
               }
               case INTERFACE -> {
                  classStructures.add(parseClass(accessModifier, modifiers,
ClassType.INTERFACE));
                  accessModifier = AccessModifier.PACKAGE;
                  modifiers = new ArrayList<>();
               }
               case ENUM -> {
```

```
                    classStructures.add(parseClass(accessModifier, modifiers, ClassType.ENUM));
                    accessModifier = AccessModifier.PACKAGE;
                    modifiers = new ArrayList<>();
                }
                case RECORD -> {
                    //record parsing is not implemented
                    do {
                        currentToken = lexer.next();
                    } while (currentToken.type() != JavaToken.TokenType.LCURLY);
                    skipBody();
                }
                case PACKAGE -> {
                    //a file can only have one package
                    if (packageName != null) {
                        throw new RuntimeException("Duplicate package definition in one file!");
                    }
                    packageName = new StringBuilder();
                    accessModifier = AccessModifier.PACKAGE;
                    modifiers = new ArrayList<>();
                    currentToken = lexer.next();
                    while (currentToken.type() != JavaToken.TokenType.SEMICOLON) {
                        if (currentToken.type() == JavaToken.TokenType.IDENTIFIER) {
                            packageName.append(currentToken.value().toString());
                        } else if (currentToken.type() == JavaToken.TokenType.DOT) {
                            packageName.append(".");
                        }
                        currentToken = lexer.next();
                    }
                }
                case IMPORT -> skipUntilSemicolon();
                default -> {
                }
            }
        }

        currentToken = lexer.next();
    }

    if (packageName == null) {
        packageName = new StringBuilder();
    }

    for (ClassStructure clazz : classStructures) {
        clazz.getPublicVariables(packageName.toString() + ".", publicClassVariables);
    }

    return new AnalysisResult(classStructures, "class", "interface", maxDepth,
publicClassVariables, (float) lexer.getComment() / (float) lexer.getLength(), lexer.getLines());
}
private ClassStructure parseClass(AccessModifier am, List<Modifier> cm, ClassType ct) {
```

```
//skip the `class` or equivalent keyword
currentToken = lexer.next();
String className;
List<Method> methods = new ArrayList<>();
List<Field> fields = new ArrayList<>();
List<ClassStructure> subClasses = new ArrayList<>();
AccessModifier accessModifier = AccessModifier.PACKAGE;
List<Modifier> modifiers = new ArrayList<>();
String extendedClass = null;
List<String> implementedInterfaces = new ArrayList<>();
List<String> enumValues = new ArrayList<>();

if (currentToken.type() == JavaToken.TokenType.IDENTIFIER) {
   StringBuilder name = new StringBuilder(currentToken.value().toString());
   //classnames can have generics
   parseGenericType(name);
   className = name.toString();
} else {
   throw new RuntimeException("Expected class name!");
}

//check if we have extends or implements
while (currentToken.type() == JavaToken.TokenType.KEYWORD &&
   (currentToken.value() == JavaToken.Keyword.EXTENDS ||
      currentToken.value() == JavaToken.Keyword.IMPLEMENTS)) {
   if (currentToken.value() == JavaToken.Keyword.EXTENDS) {
      currentToken = lexer.next();
      if (currentToken.type() == JavaToken.TokenType.IDENTIFIER) {
         StringBuilder name = new StringBuilder(currentToken.value().toString());
         parseGenericType(name);
         extendedClass = name.toString();
      } else {
         throw new RuntimeException("Expected class name after extends!");
      }
   } else {
      currentToken = lexer.next();
      while (currentToken.type() == JavaToken.TokenType.IDENTIFIER) {
         StringBuilder name = new StringBuilder(currentToken.value().toString());
         parseGenericType(name);
         implementedInterfaces.add(name.toString());
         if (currentToken.type() == JavaToken.TokenType.COMMA) {
            currentToken = lexer.next();
         }
      }
   }
}

if (currentToken.type() == JavaToken.TokenType.LCURLY) {
   currentToken = lexer.next();
} else {
```

```
            throw new RuntimeException("Expected {!");
        }

        //enum values
        if (ct == ClassType.ENUM && currentToken.type() == JavaToken.TokenType.IDENTIFIER) {
            while (true) {
                String val = currentToken.value().toString();
                currentToken = lexer.next();
                if (currentToken.type() == JavaToken.TokenType.LPAREN) {
                    enumValues.add(val + "( ... )");
                    int depth = 1;
                    while (depth > 0 && currentToken.type() != JavaToken.TokenType.EOF) {
                        currentToken = lexer.next();
                        if (currentToken.type() == JavaToken.TokenType.LPAREN) {
                            depth++;
                        } else if (currentToken.type() == JavaToken.TokenType.RPAREN) {
                            depth--;
                        }
                    }
                    currentToken = lexer.next();
                    if (currentToken.type() == JavaToken.TokenType.SEMICOLON) {
                        break;
                    } else if (currentToken.type() == JavaToken.TokenType.RCURLY) {
                        return new ClassStructure(className, am, cm, methods, fields, ct, subClasses, extendedClass, implementedInterfaces, enumValues);
                    }
                    currentToken = lexer.next();
                } else if (currentToken.type() == JavaToken.TokenType.SEMICOLON) {
                    break;
                } else if (currentToken.type() == JavaToken.TokenType.RCURLY) {
                    return new ClassStructure(className, am, cm, methods, fields, ct, subClasses, extendedClass, implementedInterfaces, enumValues);
                }
                else {
                    enumValues.add(val);
                    currentToken = lexer.next();
                }
            }
        }

        //continue until the end of the class
        while (currentToken.type() != JavaToken.TokenType.RCURLY &&
            currentToken.type() != JavaToken.TokenType.EOF) {
            if (currentToken.type() == JavaToken.TokenType.ANNOTATION) skipAnnotation();
            if (currentToken.type() == JavaToken.TokenType.KEYWORD) {
                switch ((JavaToken.Keyword) currentToken.value()) {
                    case PUBLIC, PRIVATE, PROTECTED -> {
                        accessModifier = AccessModifier.valueOf(currentToken.value().toString().toUpperCase());
```

```
                currentToken = lexer.next();
            }
            case STATIC, ABSTRACT, FINAL, DEFAULT -> {

modifiers.add(Modifier.valueOf(currentToken.value().toString().toUpperCase()));
                currentToken = lexer.next();
            }
            case CLASS -> {
                subClasses.add(parseClass(accessModifier, modifiers, ClassType.CLASS));
                accessModifier = AccessModifier.PACKAGE;
                modifiers = new ArrayList<>();
            }
            case INTERFACE -> {
                subClasses.add(parseClass(accessModifier, modifiers,
ClassType.INTERFACE));
                accessModifier = AccessModifier.PACKAGE;
                modifiers = new ArrayList<>();
            }
            case ENUM -> {
                subClasses.add(parseClass(accessModifier, modifiers, ClassType.ENUM));
                accessModifier = AccessModifier.PACKAGE;
                modifiers = new ArrayList<>();
            }
            default -> {
                StringBuilder type = new
StringBuilder(currentToken.value().toString().toLowerCase());
                parseGenericType(type);
                if (currentToken.type() == JavaToken.TokenType.IDENTIFIER) {
                    String name = currentToken.value().toString();
                    currentToken = lexer.next();
                    if (currentToken.type() == JavaToken.TokenType.LPAREN) {
                        methods.add(parseMethodPrototype(name, type.toString(), accessModifier,
modifiers));
                    } else {
                        fields.add(new Field(name, type.toString(), accessModifier, modifiers));
                        skipUntilSemicolon();
                        currentToken = lexer.next();
                    }
                }
                accessModifier = AccessModifier.PACKAGE;
                modifiers = new ArrayList<>();
            }
        }
    } else if (currentToken.type() == JavaToken.TokenType.IDENTIFIER) {
        StringBuilder type = new StringBuilder(currentToken.value().toString());
        parseGenericType(type);
        if (currentToken.type() == JavaToken.TokenType.IDENTIFIER) {
            String name = currentToken.value().toString();
            currentToken = lexer.next();
            if (currentToken.type() == JavaToken.TokenType.LPAREN) {
```

```
                methods.add(parseMethodPrototype(name, type.toString(), accessModifier,
modifiers));
            } else {
                fields.add(new Field(name, type.toString(), accessModifier, modifiers));
                skipUntilSemicolon();
                currentToken = lexer.next();
            }
        } else {
            methods.add(parseMethodPrototype("constructor", type.toString(), accessModifier,
modifiers));
        }
        accessModifier = AccessModifier.PACKAGE;
        modifiers = new ArrayList<>();
    }
    else if (currentToken.type() == JavaToken.TokenType.LCURLY) {
        accessModifier = AccessModifier.PACKAGE;
        modifiers = new ArrayList<>();
        skipBody();
    }
    else if (currentToken.type() == JavaToken.TokenType.OPERATOR &&
currentToken.value() == JavaToken.Operator.LESS) {
        skipGeneric();
    }
}

if (currentToken.type() == JavaToken.TokenType.RCURLY) {
    currentToken = lexer.next();
}

return new ClassStructure(className, am, cm, methods, fields, ct, subClasses,
extendedClass, implementedInterfaces, enumValues);
}

private void parseGenericType(StringBuilder type) {
    currentToken = lexer.next();
    if (currentToken.type() == JavaToken.TokenType.OPERATOR &&
        currentToken.value() == JavaToken.Operator.LESS) {
        type.append("<");
        int genericDepth = 1;
        //we need to loop through nested generics, and lists of generics
        while (genericDepth > 0 && currentToken.type() != JavaToken.TokenType.EOF) {
            currentToken = lexer.next();
            if (currentToken.type() == JavaToken.TokenType.OPERATOR || currentToken.type()
== JavaToken.TokenType.COMMA) {
                if (currentToken.value() == JavaToken.Operator.LESS) {
                    type.append("<");
                    genericDepth++;
                } else if (currentToken.value() == JavaToken.Operator.GREATER) {
                    type.append(">");
                    genericDepth--;
```

```
            } else if (currentToken.value() == JavaToken.Operator.RIGHT_SHIFT) {
                //here shifts mean multiple nested generics end
                type.append(">>");
                genericDepth -= 2;
            } else if (currentToken.value() == JavaToken.Operator.UNSIGNED_RIGHT_SHIFT) {
                type.append(">>>");
                genericDepth -= 3;
            } else if (currentToken.type() == JavaToken.TokenType.COMMA) {
                type.append(", ");
            }
        }
        //we also need to handle <T super E> or <? super E> and extends aswell
        else if (currentToken.type() == JavaToken.TokenType.KEYWORD && currentToken.value() == JavaToken.Keyword.EXTENDS) {
            type.append(" extends ");
        }
        else if (currentToken.type() == JavaToken.TokenType.KEYWORD && currentToken.value() == JavaToken.Keyword.SUPER) {
            type.append(" super ");
        }
        else if (currentToken.type() == JavaToken.TokenType.KEYWORD ||
            currentToken.type() == JavaToken.TokenType.IDENTIFIER) {
            type.append(currentToken.value().toString());
        }
      }
      currentToken = lexer.next();
    }
    //arrays can also be included here, List<T>[]
    while (currentToken.type() == JavaToken.TokenType.LSQUARE) {
      currentToken = lexer.next();
      type.append('[');
      if (currentToken.type() != JavaToken.TokenType.RSQUARE) {
        throw new RuntimeException("Expected ]!");
      }
      currentToken = lexer.next();
      type.append(']');
    }
    //and ...
    if (currentToken.type() == JavaToken.TokenType.TRIPLE_DOT) {
      currentToken = lexer.next();
      type.append("...");
    }
    if (currentToken.type() == JavaToken.TokenType.DOT) {
      currentToken = lexer.next();
      type.append(".");
      type.append(currentToken.value().toString());
      parseGenericType(type);
    }
  }
```

```
    private Method parseMethodPrototype(String name, String returnType, AccessModifier
accessModifier, List<Modifier> modifiers) {
        List<String> parameters = new ArrayList<>();
        List<String> throwsTypes = new ArrayList<>();

        currentToken = lexer.next();

        //we only care about the prototype, so check the parameters
        while (currentToken.type() != JavaToken.TokenType.RPAREN &&
            currentToken.type() != JavaToken.TokenType.EOF) {
            if (currentToken.type() == JavaToken.TokenType.ANNOTATION) skipAnnotation();
            if (currentToken.type() == JavaToken.TokenType.KEYWORD ||
                currentToken.type() == JavaToken.TokenType.IDENTIFIER) {
                StringBuilder paramType = new StringBuilder(currentToken.type() ==
JavaToken.TokenType.KEYWORD ? currentToken.value().toString().toLowerCase() :
currentToken.value().toString());
                parseGenericType(paramType);
                if (currentToken.type() == JavaToken.TokenType.IDENTIFIER ||
currentToken.type() == JavaToken.TokenType.KEYWORD) {
                    String paramName = currentToken.value().toString();
                    parameters.add(paramType + " " + paramName);
                    currentToken = lexer.next();
                    if (currentToken.type() == JavaToken.TokenType.COMMA) {
                        currentToken = lexer.next();
                    }
                }
                else if (currentToken.type() == JavaToken.TokenType.LPAREN ||
currentToken.type() == JavaToken.TokenType.COMMA) {
                    currentToken = lexer.next();
                }
                else {
                    throw new RuntimeException("Expected parameter type!");
                }
            } else {
                throw new RuntimeException("Expected parameter type!");
            }
        }

        if (currentToken.type() == JavaToken.TokenType.RPAREN) {
            currentToken = lexer.next();
        } else {
            throw new RuntimeException("Expected )!");
        }

        boolean hasBody = false;

        if (currentToken.type() == JavaToken.TokenType.KEYWORD && currentToken.value()
== JavaToken.Keyword.THROWS) {
            currentToken = lexer.next();
```

```
            while (true) {
                StringBuilder type = new StringBuilder(currentToken.type() == JavaToken.TokenType.KEYWORD ? currentToken.value().toString().toLowerCase() : currentToken.value().toString());
                parseGenericType(type);
                throwsTypes.add(type.toString());
                if (currentToken.type() != JavaToken.TokenType.COMMA) {
                    break;
                }
                currentToken = lexer.next();
            }
        }

        //if we have a body, skip it
        if (currentToken.type() == JavaToken.TokenType.LCURLY) {
            hasBody = true;
            skipBody();
        } else if (currentToken.type() == JavaToken.TokenType.SEMICOLON) {
            currentToken = lexer.next();
        } else {
            throw new RuntimeException("Expected { or ;!");
        }

        return new Method(name, parameters, returnType, accessModifier, modifiers, throwsTypes, hasBody);
    }

    //skip an annotation including the parenthesis it may have
    private void skipAnnotation() {
        lexer.next();
        currentToken = lexer.next();
        if (currentToken.type() == JavaToken.TokenType.LPAREN) {
            int depth = 1;
            while (depth > 0 && currentToken.type() != JavaToken.TokenType.EOF) {
                currentToken = lexer.next();
                if (currentToken.type() == JavaToken.TokenType.LPAREN) {
                    depth++;
                } else if (currentToken.type() == JavaToken.TokenType.RPAREN) {
                    depth--;
                }
            }
            currentToken = lexer.next();
        }
    }

    //skip a whole curly bracket body, counting the max depth aswell
    private void skipBody() {
        int depth = 1;
        if (maxDepth < depth) {
            maxDepth = depth;
```

```
            }
            while (depth > 0 && currentToken.type() != JavaToken.TokenType.EOF) {
                currentToken = lexer.next();
                if (currentToken.type() == JavaToken.TokenType.LCURLY) {
                    depth++;
                    if (depth > maxDepth) {
                        maxDepth = depth;
                    }
                } else if (currentToken.type() == JavaToken.TokenType.RCURLY) {
                    depth--;
                }
            }
            currentToken = lexer.next();
        }

        //skip a list of generics before functions, like <T, R>
        private void skipGeneric() {
            int depth = 1;
            if (maxDepth < depth) {
                maxDepth = depth;
            }
            while (depth > 0 && currentToken.type() != JavaToken.TokenType.EOF) {
                currentToken = lexer.next();
                if (currentToken.type() == JavaToken.TokenType.OPERATOR &&
    currentToken.value() == JavaToken.Operator.LESS) {
                    depth++;
                    if (depth > maxDepth) {
                        maxDepth = depth;
                    }
                } else if (currentToken.type() == JavaToken.TokenType.OPERATOR &&
    currentToken.value() == JavaToken.Operator.GREATER) {
                    depth--;
                }
            }
            currentToken = lexer.next();
        }

    }
```

Лістинг лексеру

```
package dev.mqxf.analyzer.java;
    import static dev.mqxf.analyzer.java.JavaToken.Operator.*;
    import static dev.mqxf.analyzer.java.JavaToken.TokenType.*;

    public class JavaLexer {

        private final char[] chars;
        private final int length;
        private int index;
```

```
//Comment characters counter
private int comment;
//Lines of code counter
private int lines;

public JavaLexer(String code) {
    chars = code.toCharArray();
    this.index = 0;
    this.length = chars.length;
}

//Look at the next char without incrementing the index
private char peek() {
    if (index < chars.length) {
        return chars[index];
    }
    return 0;
}

private char nextChar() {
    if (index < chars.length) {
        return chars[index++];
    }
    return 0;
}

public int getComment() {
    return comment;
}

public int getLength() {
    return length;
}

public int getLines() {
    return lines;
}

public JavaToken next() {
    //a token can only be on one line of code, so we can only add 1 to the lines of code counter
    boolean added = false;
    char c = nextChar();
    //if c == 0, we have reached the end of the file
    while (c != 0) {
        //skip whitespaces
        if (Character.isWhitespace(c)) {
            if (c == '\n' && !added) {
                added = true;
                lines++;
            }
```

```
    c = nextChar();
} else if (c == '/') {
    //this could be a comment
    switch (peek()) {
        case '/' -> {
            //skip until newline
            do {
                comment++;
            } while (nextChar() != '\n');
            c = nextChar();
        }
        case '*' -> {
            //skip until closing deliminer
            nextChar();
            while (true) {
                comment++;
                if (nextChar() == '*') {
                    if (nextChar() == '/') {
                        c = nextChar();
                        break;
                    }
                }
            }
        }
        //next two are not comments, so we return the token that they are
        case '=' -> {
            nextChar();
            return new JavaToken(OPERATOR_ASSIGN, DIVIDE);
        }
        default -> {
            return new JavaToken(OPERATOR, DIVIDE);
        }
    }
}
//This is any identifier or keyword
else if (Character.isAlphabetic(c) || c == '_') {
    StringBuilder builder = new StringBuilder(String.valueOf(c));
    while (Character.isAlphabetic(peek()) || Character.isDigit(peek()) || peek() == '_') {
        builder.append(nextChar());
    }
    String value = builder.toString();
    //check if it is a keyword. We use get instead of containsKey to only lookup once
    JavaToken.Keyword word = JavaToken.keywords.get(value);
    if (word != null) {
        return new JavaToken(KEYWORD, word);
    }
    return new JavaToken(IDENTIFIER, value);
}
//number
else if (Character.isDigit(c)) {
```

```
            StringBuilder builder = new StringBuilder(String.valueOf(c));
            while (Character.isDigit(peek()) || peek() == '_' || peek() == '.') {
                if (peek() == '_') continue;
                builder.append(nextChar());
            }
            String value = builder.toString();
            //number with a . must be a float or double
            if (value.contains(".")) {
                if (value.endsWith("f")) {
                    return new JavaToken(FLOAT, Float.parseFloat(value));
                } else {
                    return new JavaToken(DOUBLE, Double.parseDouble(value));
                }
            } else {
                if (value.endsWith("b")) {
                    return new JavaToken(BYTE, Byte.parseByte(value));
                } else if (value.endsWith("s")) {
                    return new JavaToken(SHORT, Short.parseShort(value));
                } else if (value.endsWith("l")) {
                    return new JavaToken(LONG, Long.parseLong(value));
                } else if (value.endsWith("f")) {
                    return new JavaToken(FLOAT, Float.parseFloat(value));
                } else if (value.endsWith("d")) {
                    return new JavaToken(DOUBLE, Double.parseDouble(value));
                } else {
                    return new JavaToken(INT, Integer.parseInt(value));
                }
            }
        } else {
            //just look at what the operator is
            return switch (c) {
                case '\'' -> {
                    c = nextChar();
                    c = escapeChar(c);
                    if (nextChar() != '\'') {
                        throw new RuntimeException("Unexpected character '" + c + "', expected '''");
                    }
                    yield new JavaToken(CHAR, c);
                }
                case '\"' -> {
                    StringBuilder builder = new StringBuilder();
                    c = nextChar();
                    while (c != '\"') {
                        c = escapeChar(c);
                        builder.append(c);
                        c = nextChar();
                    }
                    yield new JavaToken(STRING, builder.toString());
                }
                case '(' -> new JavaToken(LPAREN, null);
```

```
case ')' -> new JavaToken(RPAREN, null);
case '{' -> new JavaToken(LCURLY, null);
case '}' -> new JavaToken(RCURLY, null);
case '[' -> new JavaToken(LSQUARE, null);
case ']' -> new JavaToken(RSQUARE, null);
case ';' -> new JavaToken(SEMICOLON, null);
case ':' -> {
   if (peek() == ':') {
      nextChar();
      yield new JavaToken(DOUBLE_COLON, null);
   }
   yield new JavaToken(COLON, null);
}
case ',' -> new JavaToken(COMMA, null);
case '.' -> {
   // ... is an operator
   if (peek() == '.' & index + 1 < chars.length && chars[index + 1] == '.') {
      nextChar();
      nextChar();
      yield new JavaToken(TRIPLE_DOT, null);
   }
   yield new JavaToken(DOT, null);
}
case '+' -> switch (peek()) {
   //check for ++ and +=
   case '+' -> {
      nextChar();
      yield new JavaToken(OPERATOR, PLUSPLUS);
   }
   case '=' -> {
      nextChar();
      yield new JavaToken(OPERATOR_ASSIGN, PLUS);
   }
   default -> new JavaToken(OPERATOR, PLUS);
};
case '-' -> switch (peek()) {
   case '-' -> {
      nextChar();
      yield new JavaToken(OPERATOR, MINUSMINUS);
   }
   case '=' -> {
      nextChar();
      yield new JavaToken(OPERATOR_ASSIGN, MINUS);
   }
   case '>' -> {
      nextChar();
      yield new JavaToken(ARROW, null);
   }
   default -> new JavaToken(OPERATOR, MINUS);
};
```

```
                case '*' -> {
                    if (peek() == '=') {
                        nextChar();
                        yield new JavaToken(OPERATOR_ASSIGN, MULTIPLY);
                    }
                    yield new JavaToken(OPERATOR, MULTIPLY);
                }
                case '%' -> {
                    if (peek() == '=') {
                        nextChar();
                        yield new JavaToken(OPERATOR_ASSIGN, MODULO);
                    }
                    yield new JavaToken(OPERATOR, MODULO);
                }
                case '!' -> {
                    if (peek() == '=') {
                        nextChar();
                        yield new JavaToken(OPERATOR, NOT_EQUAL);
                    }
                    yield new JavaToken(OPERATOR, NOT);
                }
                case '<' -> switch (peek()) {
                    case '<' -> {
                        nextChar();
                        if (peek() == '=') {
                            nextChar();
                            yield new JavaToken(OPERATOR_ASSIGN, LESS_EQUAL);
                        }
                        yield new JavaToken(OPERATOR, LEFT_SHIFT);
                    }
                    case '=' -> {
                        nextChar();
                        yield new JavaToken(OPERATOR_ASSIGN, LESS_EQUAL);
                    }
                    default -> new JavaToken(OPERATOR, LESS);
                };
                case '>' -> switch (peek()) {
                    case '>' -> {
                        nextChar();
                        yield switch (peek()) {
                            case '>' -> {
                                nextChar();
                                if (peek() == '=') {
                                    nextChar();
                                    yield new JavaToken(OPERATOR_ASSIGN,
UNSIGNED_RIGHT_SHIFT);
                                }
                                yield new JavaToken(OPERATOR, UNSIGNED_RIGHT_SHIFT);
                            }
                            case '=' -> {
```

```
                        nextChar();
                        yield new JavaToken(OPERATOR_ASSIGN, RIGHT_SHIFT);
                    }
                    default -> new JavaToken(OPERATOR, RIGHT_SHIFT);
                };
            }
            case '=' -> {
                nextChar();
                yield new JavaToken(OPERATOR_ASSIGN, GREATER_EQUAL);
            }
            default -> new JavaToken(OPERATOR, GREATER);
        };
        case '=' -> {
            if (peek() == '=') {
                nextChar();
                yield new JavaToken(OPERATOR, EQUAL);
            }
            yield new JavaToken(OPERATOR, ASSIGN);
        }
        case '&' -> {
            if (peek() == '&') {
                nextChar();
                yield new JavaToken(OPERATOR, AND);
            }
            yield new JavaToken(OPERATOR, BITWISE_AND);
        }
        case '|' -> {
            if (peek() == '|') {
                nextChar();
                yield new JavaToken(OPERATOR, OR);
            }
            yield new JavaToken(OPERATOR, BITWISE_OR);
        }
        case '^' -> {
            if (peek() == '=') {
                nextChar();
                yield new JavaToken(OPERATOR_ASSIGN, XOR);
            }
            yield new JavaToken(OPERATOR, XOR);
        }
        case '@' -> new JavaToken(ANNOTATION, null);
        case '?' -> new JavaToken(QUESTION, null);
        //char must be an EOF
        default -> new JavaToken(EOF, null);
      };
    }
  }
  return new JavaToken(EOF, null);
}
```

```
    private char escapeChar(char c) {
        if (c == '\\') {
            char n = nextChar();
            c = switch (n) {
                case 'b' -> '\b';
                case 'f' -> '\f';
                case 'n' -> '\n';
                case 'r' -> '\r';
                case 't' -> '\t';
                case 'u' ->
                    (char)  Integer.parseInt(String.valueOf(nextChar()) + nextChar() + nextChar() +
nextChar(), 16);
                default -> n;
            };
        }
        return c;
    }
}
```

Приклад файлу з токенами

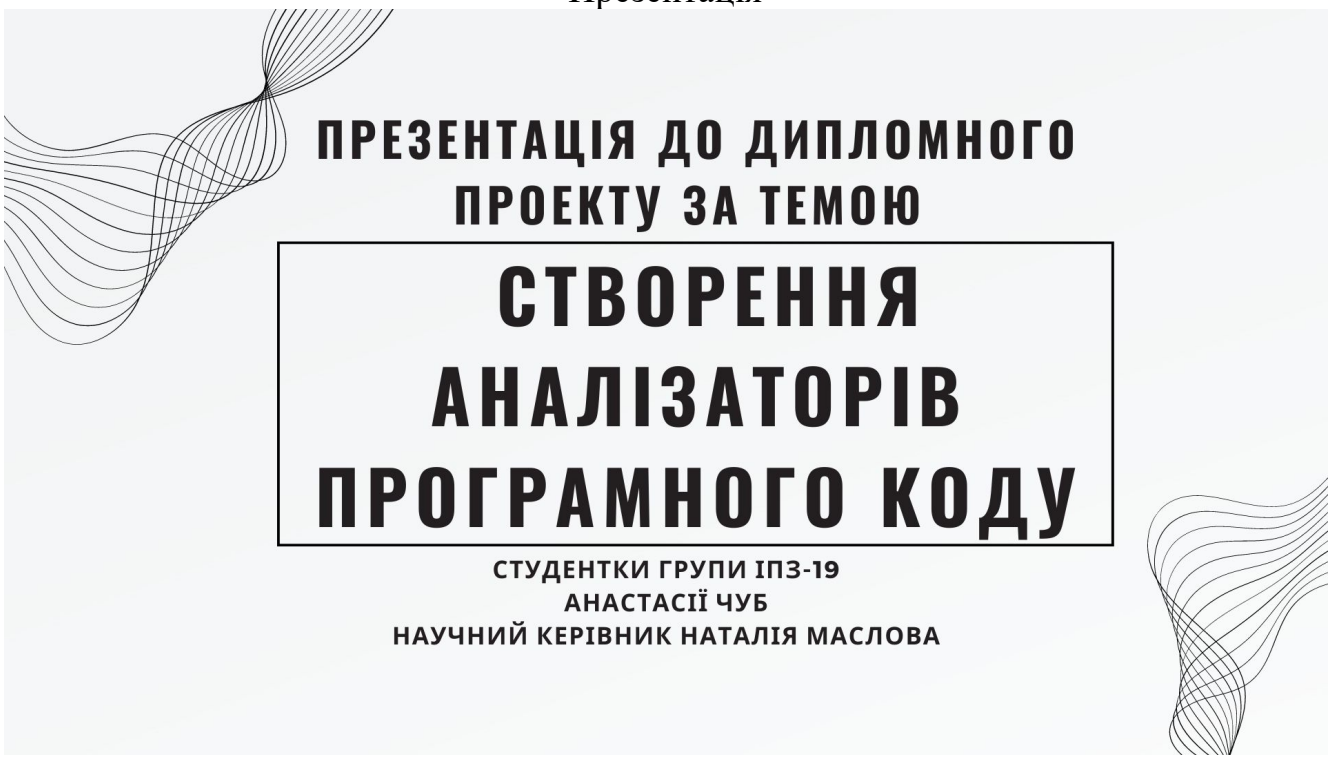
```
package dev.mqxf.analyzer.java;
import java.util.HashMap;
public record JavaToken(TokenType type, Object value) {

    //A HashMap is easier to work with to store keywords
    public static final HashMap<String, Keyword> keywords = new HashMap<String,
Keyword>() {
        {
            put("public", Keyword.PUBLIC);
            put("private", Keyword.PRIVATE);
            put("protected", Keyword.PROTECTED);
            put("static", Keyword.STATIC);
            put("final", Keyword.FINAL);
            put("abstract", Keyword.ABSTRACT);
            put("strictfp", Keyword.STRICTFP);
            put("synchronized", Keyword.SYNCHRONIZED);
            put("transient", Keyword.TRANSIENT);
            put("volatile", Keyword.VOLATILE);
            put("native", Keyword.NATIVE);
            put("class", Keyword.CLASS);
            put("interface", Keyword.INTERFACE);
            put("enum", Keyword.ENUM);
            put("extends", Keyword.EXTENDS);
            put("super", Keyword.SUPER);
            put("implements", Keyword.IMPLEMENTS);
            put("import", Keyword.IMPORT);
            put("package", Keyword.PACKAGE);
            put("if", Keyword.IF);
```

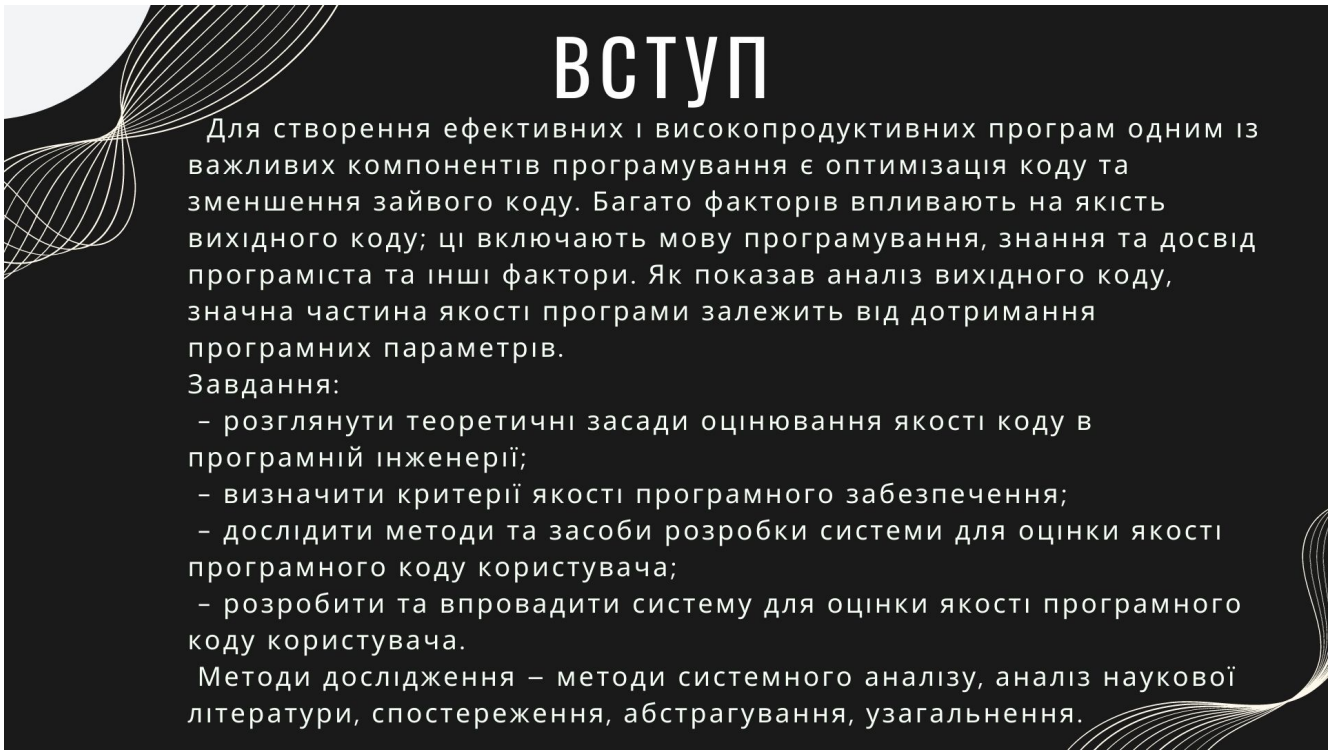
```
            put("else", Keyword.ELSE);
            put("while", Keyword.WHILE);
            put("do", Keyword.DO);
            put("for", Keyword.FOR);
            put("switch", Keyword.SWITCH);
            put("case", Keyword.CASE);
            put("default", Keyword.DEFAULT);
            put("break", Keyword.BREAK);
            put("continue", Keyword.CONTINUE);
            put("return", Keyword.RETURN);
            put("throw", Keyword.THROW);
            put("throws", Keyword.THROWS);
            put("assert", Keyword.ASSERT);
            put("new", Keyword.NEW);
            put("void", Keyword.VOID);
            put("int", Keyword.INT);
            put("short", Keyword.SHORT);
            put("long", Keyword.LONG);
            put("float", Keyword.FLOAT);
            put("double", Keyword.DOUBLE);
            put("boolean", Keyword.BOOLEAN);
            put("char", Keyword.CHAR);
            put("byte", Keyword.BYTE);
            put("catch", Keyword.CATCH);
            put("finally", Keyword.FINALLY);
            put("true", Keyword.TRUE);
            put("false", Keyword.FALSE);
            put("null", Keyword.NULL);
            put("record", Keyword.RECORD);
        }
    };
    public enum TokenType {
        KEYWORD,
        IDENTIFIER,
        OPERATOR,
        OPERATOR_ASSIGN,
        STRING,
        CHAR,
        FLOAT,
        DOUBLE,
        BYTE,
        SHORT,
        INT,
        LONG,
        LPAREN,
        RPAREN,
        LSQUARE,
        RSQUARE,
        LCURLY,
        RCURLY,
```

```
    SEMICOLON,
    COMMA,
    DOT,
    TRIPLE_DOT,
    COLON,
    DOUBLE_COLON,
    ARROW,
    ANNOTATION,
    QUESTION,
    EOF
}

public enum Keyword {
    PUBLIC,
    PRIVATE,
    PROTECTED,
    STATIC,
    FINAL,
    ABSTRACT,
    STRICTFP,
    SYNCHRONIZED,
    TRANSIENT,
    VOLATILE,
    NATIVE,
    CLASS,
    INTERFACE,
    ENUM,
    EXTENDS,
    SUPER,
    IMPLEMENTS,
    IMPORT,
    PACKAGE,
    IF,
    ELSE,
    WHILE,
    DO,
    FOR,
    SWITCH,
    CASE,
    BREAK,
    CONTINUE,
    RETURN,
    THROW,
    THROWS,
    CATCH,
    DEFAULT,
    ASSERT,
    NEW,
    VOID,
    INT,
```

```
        SHORT,
        LONG,
        FLOAT,
        DOUBLE,
        BOOLEAN,
        CHAR,
        BYTE,
        FINALLY,
        TRUE,
        FALSE,
        NULL,
        RECORD
    }

    public enum Operator {
        PLUS,
        PLUSPLUS,
        MINUS,
        MINUSMINUS,
        MULTIPLY,
        DIVIDE,
        MODULO,
        AND,
        OR,
        NOT,
        XOR,
        ASSIGN,
        EQUAL,
        NOT_EQUAL,
        GREATER,
        LESS,
        GREATER_EQUAL,
        LESS_EQUAL,
        AND_EQUAL,
        OR_EQUAL,
        LEFT_SHIFT,
        RIGHT_SHIFT,
        UNSIGNED_RIGHT_SHIFT,
        BITWISE_AND,
        BITWISE_OR
    }
}
```

Додаток Г.
Презентація

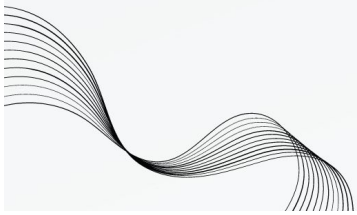




ПРЕДМЕТНА ОБЛАСТЬ

Вимірювання якості програмного забезпечення полягає в кількісному визначенні того, якою мірою система або програмне забезпечення має бажані характеристики. Це може бути виконано за допомогою якісних чи кількісних засобів чи їх поєднання.

Показники надійності використовуються для кількісного визначення надійності програмного продукту. Вибір метрики залежить від типу системи, до якої вона застосовується, а також від умов предметної області, яка її використовує.

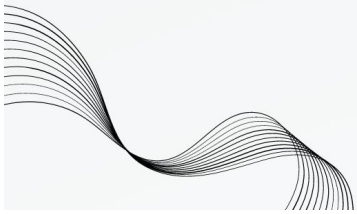


ПОСТАНОВКА ЗАДАЧІ

Мета полягає в тому, щоб створити систему, яка може оцінювати якість програмного коду користувача.

У нас є три завдання, які ми повинні виконати:

1. Дослідити теоретичні основи оцінювання якості кода в програмній інженерії;
2. Визначити стандарти якості програмного забезпечення;
3. Досліджувати методи та інструменти розробки системи з метою оцінки якості програмного коду користувача;
4. Створити та запустити систему для оцінки якості програмного коду користувача.



МЕТОДИ РОЗРОБКИ СИСТЕМИ АНАЛІЗУ КОДУ

Стандарти оцінювання якості включають аудит, інспекцію коду, перевірку дизайну, моделювання, функціональне тестування, стандартизацію, покрокове керівництво, модульне тестування, стрес-тестування.

Статичний аналіз — це аналіз програмного забезпечення, який виконується автоматичним інструментом без використання програми. Деякі популярні методи статичного аналізу включають програмні метрики та реверс-інжиніринг. Нові команди використовують такі інструменти для статичного аналізу коду, як SonarCube і VeraCode.

Модульне тестування — це метод тестування білої скриньки, при якому повне покриття коду гарантується виконанням кожної незалежної гілки, шляху та умови хоча б один раз.

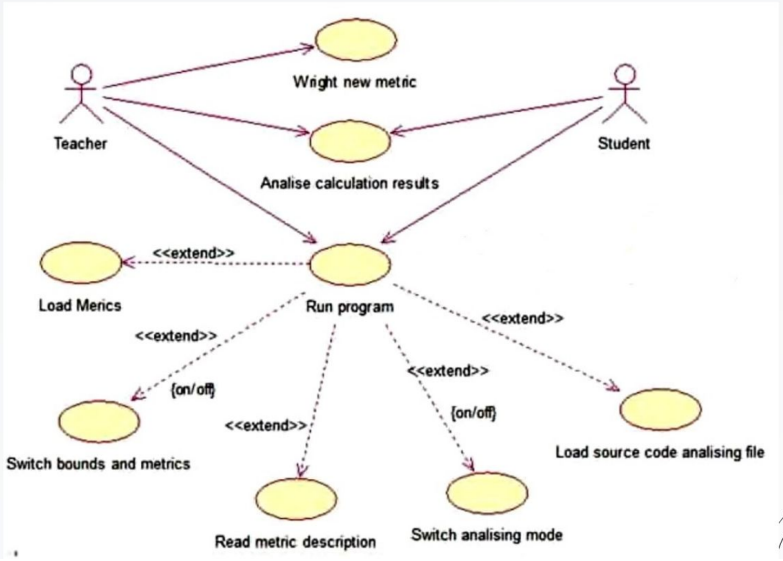
Стрес-тестування. Цей тип тестування проводиться для перевірки надійності системи, випробовуючи її під надзвичайним навантаженням, тобто поза нормальними умовами.

ПАРСЕРИ ТА ЛЕКСИЧНІ АНАЛІЗАТОРИ

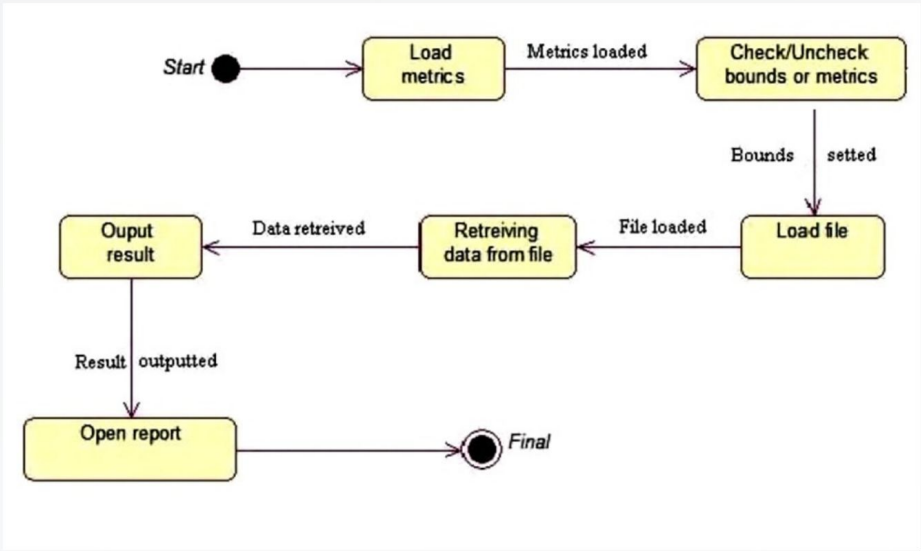
Парсер є важливою складовою частиною процесу аналізу мови в області комп'ютерних наук. Він використовується для аналізу послідовності символів або токенів, що складають мову, і визначення синтаксичної структури цієї послідовності. Парсер перетворює вхідну послідовність символів на дерево синтаксичного розбору (або іншу структуру даних), яка відображає структуру вхідного рядка згідно з граматикою мови.

Лексер, також відомий як лексичний аналізатор, є першим етапом компілятора або системи аналізу коду. Його основна функція полягає в розбитті вхідного вихідного коду на послідовність токенів.

ДІАГРАМА ВИПАДКІВ ВИКОРИСТАННЯ ПРОГРАМНОГО ЗАСТОСУНКУ



ДІАГРАМА СТАНІВ ПРОГРАМИ



АЛГОРИТМ РОБОТИ ЛЕКСИЧНОГО АНАЛІЗАТОРУ

Символ	Дія
Пробіли	Пропускаємо
/	Перевіряємо наступний символ, пропускаючи його до кінця рядка або до */ у випадку коментаря, або повертаємо токен DIVIDE.
Букви або _	Конструюємо рядок з будь-яких буквено-цифрових символів або _. Потім робимо пошук в хеш-таблиці, щоб перевірити, чи наступне слово є ключовим словом. Якщо так, повертаємо токен KEYWORD, інакше повертаємо токен IDENTIFIER
Число	Виконуємо парсинг число до кінця послідовності цифр, плюс одну нецифрову, щоб дозволити, наприклад, 10f як число з плаваючою точкою.
'	Зчитуємо один символ, або два, якщо перший символ - , і створюємо токен CHARACTER.
"	Зчитуємо лексеми до неквотованої " і повертаємо токен STRING.
Все інше	Проходить через оператор переключення, де можливо перевіряється наступний символ (у випадках, наприклад, ++ або +=) або миттєво повертається токен.

JAVA SWING API

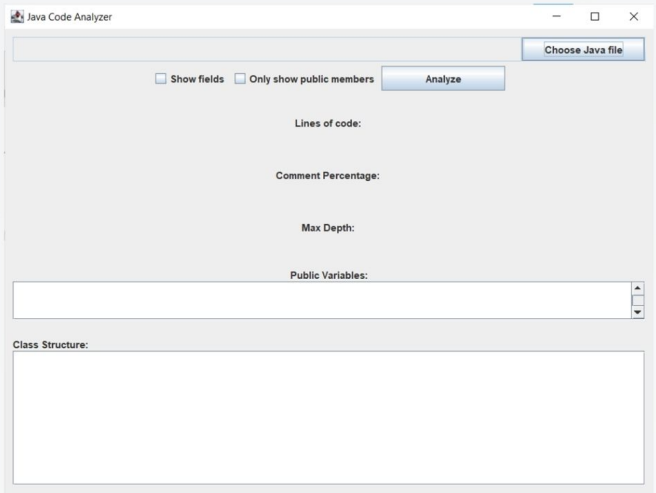
Java Swing є набором бібліотек та інструментів, які дозволяють розробникам створювати графічний інтерфейс користувача (GUI) для програм, написаних на Java. Він надає широкий спектр компонентів і контролів, які можна використовувати для створення різноманітних вікон, кнопок, полів введення, таблиць, списків та багатьох інших елементів GUI.

Основні компоненти Swing API: JFrame, JPanel, JButton, JLabel, JTextField і тд.

ОПИС КЛАСІВ ПОЛІВ ТА МЕТОДІВ

Пакунок **`analyzer`** (dev.mqxf.analyzer) містить весь код, необхідний для аналізу файлів коду. Це починається з інтерфейсу `Analyzer`. Він містить дві функції, одна з яких є абстрактною, а інша – за замовчуванням. Абстрактна функція має бути реалізована будь-якою реалізацією інтерфейсу аналізатора, тоді як функція за замовчуванням дозволяє користувачеві цього коду аналізувати будь-яку мову без необхідності шукати конкретну реалізацію аналізатора, використовуючи перечислення enum `Language`. Хоча наразі цей код дозволяє аналізувати лише код на Java, ця функція дозволяє легко розширити його до більшої кількості мов. **AnalysisResult**: Це повний результат аналізу. Він містить все, що було знайдено під час аналізу. Він має 6 полів: classStructures, classWord, interfaceWord, maxDepth, publicClassVariables і commentPercent. classStructures – це список `java.util.List` структур класів, що представляє всі класи верхнього рівня, знайдені в аналізі.

ВПРОВАДЖЕННЯ СИСТЕМИ



Тож запустимо спроектовану систему. При запуску програми з'являється вікно. Одразу ми бачимо кнопку для завантаження файлу, нижче ми бачимо два перемикача "Showfields" та "Only show public members".

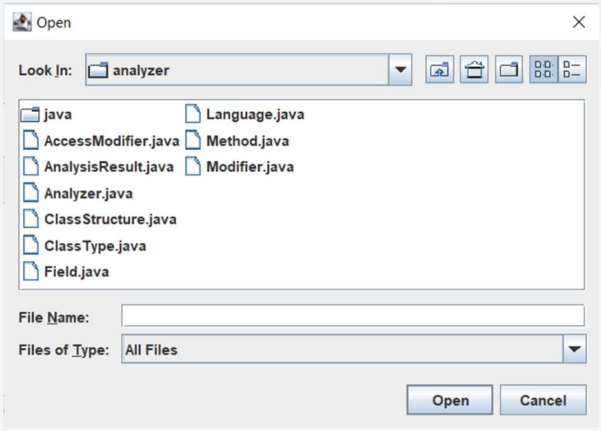
Перший перемикач "Showfields" при його активації демонструє всі поля класів обраного Java файлу у рамці "ClassStructure". Другий перемикач "Onlyshow public members" допомагає сховати всі непублічні структури в класах обраного файлу. Зміни можна побачити також у текстовому полі "ClassStructure" після активації перемикача і натиснення кнопки "Analyze".

Нижче перемикачів ми можемо бачити строку "Lines of code", вона відображає результати однойменної метрики або LOC, тобто кількість строчок обраного для аналізу файлу.

Наступним ми бачимо строку "CommentPercentage", яка аналізує який процент строк коду має в собі коментарі.

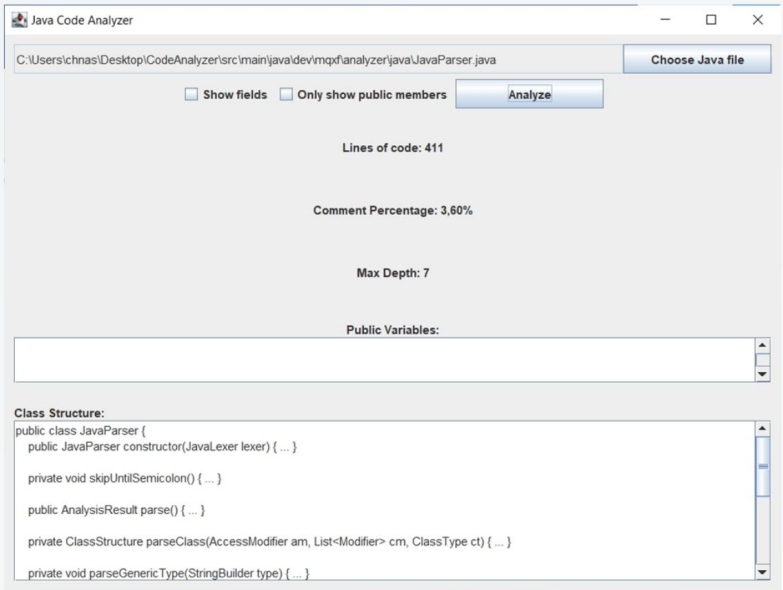
Тож спробуємо додати файл Java для аналізу

ВПРОВАДЖЕННЯ СИСТЕМИ



Після вибору файлу для аналізу натискаємо на кнопку “Analyze”, після чого виконується аналіз, результати якого ми одразу можемо бачити у відповідних полях. Також ми бачимо строку “MaxDepth”, яка відображає вкладеність або рекурсивність коду, що аналізується. Нижче поле можемо бачити “PublicVariables”, в якому демонструються всі змінні публічного ступеню доступності файлу, який аналізується. “Class Structure” відображає структуру файлу, тобто всі його складові частини: класи, функції і так далі.

ВПРОВАДЖЕННЯ СИСТЕМИ



ВИСНОВКИ

Під час виконання проекту зі створення аналізатора коду було проведено аналіз предметної області, надано обґрунтування обраної теми, визначену мету, об'єкт, предмет та методи дослідження. Також був проведений аналіз конкурентів.

В результаті виконання проекту зі створення аналізатора коду були досягнуті такі цілі:

–розроблені алгоритми та методи аналізу програмного коду;

–реалізована система аналізатора коду з використанням обраного програмного середовища;

–виявлені потенційні проблеми у програмному коді;

–забезпечено зручний інтерфейс для візуалізації та звітування результатів аналізу коду;

–розроблені модулі для автоматизованого тестування та оцінки покриття коду.

Після завершення аналітичної роботи була розроблена система аналізатора коду, включаючи вибір практичних інструментів та розробку архітектури системи.

Під час розробки аналізатора коду були описані особливості реалізації модулів системи, визначена структура бази даних та розроблена блок–схема алгоритму аналізу коду.

ДЯКУЮ ЗА УВАГУ