Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)
Кафедра програмної інженерії
(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

бакалавр
(назва освітнього ступеня)

на тему: Методи та програмні механізми генерації візуальних моделей структури Android-додатків

Виконав: студент IV курсу, групи СП-41

спеціальності 121 Інженерія програмного забезпечення
(шифр і назва спеціальності)

_____ Окушко Д.О.
(підпис) (прізвище та ініціали)

Керівник _____ Гладьо Ю.Б.
(підпис) (прізвище та ініціали)

Нормоконтроль _____ Стоянов Ю.М.
(підпис) (прізвище та ініціали)

Завідувач кафедри _____ Петрик М.Р.
(підпис) (прізвище та ініціали)

Рецензент _____ _____
(підпис) (прізвище та ініціали)

Тернопіль - 2025

Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)

Кафедра програмної інженерії
(повна назва кафедри)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Петрик М.Р.
(підпис) (прізвище та ініціали)
«__» _____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня Бакалавр
(назва освітнього ступеня)

за спеціальністю 121 Інженерія програмного забезпечення
(шифр і назва спеціальності)

Студенту Окушку Денису Олександровичу
(прізвище, ім'я, по батькові)

1. Тема роботи Методи та програмні механізми генерації візуальних моделей структури Android-додатків

Керівник роботи Гладьо Юрій Богданович, к.т.н., доц.
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Затверджені наказом ректора від « 18 » 04 2025 року № 4/7-333

2. Термін подання студентом завершеної роботи 17.06.2025р.

3. Вихідні дані до роботи наукові літературні джерела

4. Зміст роботи (перелік питань, які потрібно розробити)
Вступ
1. Огляд предметної області.
2. Теоретична частина
3. Практична частина
4. Безпека життєдіяльності, основи охорони праці
Висновки

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, слайдів)
1. Титулка. 2. Мета, задачі дослідження. 3. Порівняння аналогів
4. Концепція програмного рішення. 5. ПЗ, яке використано в роботі
6. Послідовність дій роботи плагіна. 7. Послідовність роботи з модулями у межах діаграми класів. 8. Послідовність збору інформації для діаграми класів.
9. Приклади результуючих діаграм Android –компонентів та пакетів.
10. Алгоритм роботи DiagramGenerator. 11. Скріншоти вікон роботи плагіна.
12. Приклад результату роботи плагіна.
13. Висновки. Основні результати проведеного дослідження.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Безпека життєдіяльності, основи охорони праці	Лазарюк В.В., доц.каф. МТ		

7. Дата видачі завдання 18.04.2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1.	Ознайомлення з завданням до кваліфікаційної роботи	18.04 – 20.04.25	*Виконано*
2.	Підбір джерел про генерацію візуальних моделей		
	структури Android-додатків	21.04 – 24.04.25	*Виконано*
3.	Опрацювання джерел про у генерацію візуальних		
	моделей структури Android-додатків	25.04 – 30.04.25	*Виконано*
4.	Проведення дослідження щодо генерації візуальних		
	моделей структури Android-додатків	01.05 – 04.05.25	*Виконано*
5	Розроблення програмного коду	05.05 – 12.05.25	*Виконано*
6.	Оформлення розділу «Огляд предметної області»	13.05 – 18.05.25	*Виконано*
7.	Оформлення розділу «Теоретична частина»		*Виконано*
8.	Оформлення розділу «Практична частина»	19.05 – 24.05.25	*Виконано*
9.	Виконання завдання до підрозділу «Безпека		
	життєдіяльності, основи хорони праці»	25.05 – 31.05.25	*Виконано*
10.	Оформлення кваліфікаційної роботи		*Виконано*
11.	Нормоконтроль	01.06 – 05.06.25	*Виконано*
12.	Перевірка на плагіат	06.06 – 09.06.25	*Виконано*
13.	Попередній захист кваліфікаційної роботи	10.06 – 14.06.25	*Виконано*
14.	Захист кваліфікаційної роботи	24.06.25	

Студент _____ Окушко Д.О.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Гладьо Ю.Б.
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Кваліфікаційна робота бакалавра. Тернопільський національний технічний університет імені Івана Пулюя, кафедра програмної інженерії, спеціальність 121 «Інженерія програмного забезпечення». ТНТУ, 2025, Сторінок 53, рисунків 25, джерел 41.

Ключові слова: візуалізація, діаграма, клас, плагін, Android, Java, Kotlin, PSI, UML

У кваліфікаційній роботі бакалавра спроектовано та розроблено спеціалізований плагін для автоматичної генерації візуальних моделей Android - застосунків як UML -діаграм.

Проаналізовано діаграми для моделювання структури вихідного коду, котрі найчастіше використовуються. Побудовано тип діаграм, котрий відтворює взаємозв'язки основних елементів Android –додатків.

Розроблено методи збирання інформації щодо класів для Kotlin і Java – файлів та щодо модулів у проекті за допомогою аналізу зв'язків. Втілено механізм кластеризації модулів і їх посилань через детектування модулів з ідентичними посиланнями. Втілено метод рекурсивного збирання транзитивних посилань на класи-спадкоємці.

Розроблено генератор діаграм, котрий дає змогу відображати діаграми класів, пакетів і Android -компонентів при допомозі обробки інформації від класів аналізаторів.

ABSTRACT

Bachelor's thesis. Ivan Puluj Ternopil National Technical University, Department of Software Engineering, specialty 121 "Software Engineering". TNTU, 2025, Pages 53, figures 25, sources 41.

Key words: visualization, diagram, class, plug-in, Android, Java, Kotlin, PSI, UML

In the bachelor's thesis, a specialized plugin for automatic generation of visual models of Android applications as UML diagrams was designed and developed.

The most commonly used diagrams for modeling the structure of the source code were analyzed. A type of diagram was constructed that reproduces the relationships of the main elements of Android applications.

Methods for collecting information about classes for Kotlin and Java files and about modules in the project using link analysis were developed. A mechanism for clustering modules and their links through the detection of modules with identical links was implemented. A method for recursively collecting transitive links to inherited classes was implemented.

A diagram generator was developed that allows you to display diagrams of classes, packages and Android components using information processing from analyzer classes.

ПЕРЕЛІК СКОРОЧЕНЬ І ТЕРМІНІВ

DBSCAN (англ. Density-Based Spatial Clustering of Applications with Noise) – алгоритм кластеризації даних.

IDE (англ. Integrated Development Environment) – sнтегроване середовище розробки.

IntelliJ IDEA – комерційне інтегроване середовище розробки для різних мов програмування (Java, Python, Scala, PHP та ін.) від компанії JetBrains.

PSI (англ. Program Structure Interface) – шар на платформі IntelliJ, що відповідає за розбір файлів і створення синтаксичної та семантичної моделі коду.

UML (англ. Unified Modeling Language) – уніфікована мова моделювання/

ОС – операційна система.

ПЗ – програмне забезпечення.

ЗМІСТ

ВСТУП

Актуальність теми дослідження. З кожним днем програмні системи, що розробляються, стають все більшими і складнішими через зростаючі вимоги клієнтів і вдосконалення технологій. Для управління складністю дизайну програмних продуктів на початку 1990-х років було запропоновано використання поняття архітектури ПЗ. Цей підхід пропонує поділ великих та складних систем на керовані компоненти та сполучні елементи, а також забезпечення можливості їхнього високорівневого аналізу та принципового розуміння.

Якщо розробники не дотримуються цього підходу і не вживають активних заходів щодо забезпечення архітектури проекту, то внаслідок постійного збільшення розміру кодової бази з часом часто погіршуються масштабованість, рівень читання та загальна якість коду.

Розбиття ПЗ на модулі і створення архітектури, що розширюється, зменшують проблеми обслуговування, але зростаюча функціональність ускладнює його підтримку та вивчення вихідного коду, яке може займати до 70% від загального часу, що витрачається на розробку.

Методи візуалізації допомагають зрозуміти програмні системи та знизити витрати на їх розробку, показуючи ключові елементи системи. Зокрема структурні діаграми можуть відображати статичну структуру ПЗ та різні рівні абстракції та реалізації.

Структурні діаграми UML, такі як діаграми класів та компонентів, є основними в об'єктно-орієнтованому проектуванні та широко використовуються для документування та розуміння системних архітектур. Вони дозволяють зобразити ієрархію компонентів або модулів, їх наповнення і те, як вони пов'язані та взаємодіють між собою. Але оскільки архітектура проекту змінюється з часом, необхідно постійно актуалізувати діаграми та вносити до них зміни, що може бути ресурсомістким, якщо робити це вручну.

Метою роботи є проектування та розробка плагіна для автоматичної генерації

візуальних моделей Android -додатків у вигляді UML -діаграм.

Для досягнення мети потрібно вирішити наступні завдання:

– аналіз та вибір реалізованих діаграм;

– розробка типу діаграм, що характеризують особливості Android -додатків;

– розробка методу збору інформації про класи для Kotlin та Java -файлів;

– розробка методу збору інформації про модулі у проекті;

– розробка методу отримання посилань на класи;

– створення генератора діаграм.

1 ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Аналіз альтернативних рішень

Були розглянуті альтернативні програмні рішення для вирішення поставленої проблеми, які можуть бути використані в середовищі розробки Android Studio, серед яких плагіни CodeIris Plugin, SequenceDiagram Plugin, PlantUML Diagram Generator , а також вбудовані в інструменти IDE Structure та Hierarchy.

Source file structure [8] - вбудований інструмент, що відображає класи, їх властивості, поля та залежності. Працює при виборі файлу. Є можливість відсортувати елементи на ім'я, їх видимості (public, protected, private).

Можна налаштувати, наскільки докладно потрібно відобразити інформацію (чи показувати змінні, анонімні внутрішні класи тощо). Він також дає можливість розгортати та згортати методи і переходити до джерела та від нього.

Source code hierarchy [9] - дозволяє вивчити ієрархію класів, методів та викликів та досліджувати структуру вихідних файлів. Є три типи ієрархій:

– type hierarchies показує батьківські та дочірні класи поточного класу;

– метод hierarchies показує підкласи, у яких метод перевизначає обраний метод, і навіть суперкласи чи інтерфейси, у яких обраний метод перевизначається$

– call hierarchies показує методи, що викликають (супертипи) або викликані (підтипи). Якщо викликати ієрархію дзвінків для поля, вона покаже список методів, у яких використовується вибране поле.

CodeIris Plugin [10] - це інтегрована візуалізація вихідного коду проекту в IDE. Спочатку плагін був розроблений для проектів на Java, але в останньому оновленні додано експериментальну підтримка Kotlin. Плагін не адаптований для нових версій Android Studio і має низький рейтинг в офіційному маркетплейсі JetBrains [11], оскільки користувачі скаржаться на різні незручності у використанні.

SequenceDiagram Plugin [12,13] - плагін, що дозволяє згенерувати діаграму простих послідовностей для вибраного файлу, перейти за кодом, клацнувши на частині діаграми (для Kotlin не підтримується), видалити клас з діаграми,

експортувати діаграму як зображення або файл.

PlantUML Diagram Generator [14] - плагін, що дає можливість створити різні UML -діаграми (Call Hierarchy, Class Structure, Control Flow, Data Flow), ґрунтуючись на програмному коді. Багато опцій фільтрації для відображення необхідної інформації, є рекурсивна генерація окремих діаграм або цілих каталогів.

Було ще розглянуто плагін UML class diagrams [15, 16], який за умовчанням вбудований у IDE Intellij Idea Ultimate. Він може бути застосований до Android - проекту, так як є можливість імпортувати код з Android Studio. Плагін раніше підтримував лише Java -класи, але з 2021 року було додано підтримку Kotlin. На діаграмі класів відображаються класи, їх методи та атрибути, а також відносини між ними – залежність, спадкування та агрегація. Можна налаштовувати області видимості за допомогою патернів (наприклад, можна виключити модуль з DI).

За замовчуванням діаграма генерується із залежностями, спрямованими знизу нагору. На основі цього можна швидко визначити компоненти, що порушують Dependency Rule (правило залежності).

Різні вбудовані рішення для візуалізації структури коду зустрічаються і в інших популярних IDE. Visual Studio займає друге місце у рейтингу IDE за версією Stack Overflow [17], тому було прийнято рішення вивчити, які стандартні інструменти в ній використовуються для вирішення поставленої проблеми [18]:

– Solution Explorer (Class View) відображає елементи програми. У верхній панелі відображаються простір імен, типи, інтерфейси, переліки та класи, а в нижній панелі - елементи, що належать типу, вибраному у верхній панелі;

– Вікно Call Hierarchy показує, де викликається цей метод чи властивість. У ньому також перераховані методи, що викликаються із цього методу. Можна переглядати кілька рівнів графа викликів, який показує відносини між методами, що викликають і викликаються в заданій області видимості;

– Object Browser відображає зовнішні об'єкти, які використовуються в проекті (файли керованого коду, збірки бібліотек, бібліотеки типів і файли .ocx).

При порівнянні альтернативних рішень, які можуть бути застосовані до Android -розробки, в першу чергу зверталася увага на наявність підтримки Kotlin -

файлів, так як він визнаний Google як рекомендована мова Android -розробки і більш популярний, ніж Java: згідно з опитуванням, проведеним Stack Overflow, 61,55% програмістів використовують Kotlin для розробки мобільних додатків .

По-друге, враховувалася наявність функціоналу, здатного відображати специфічні для Android -додатків відносини та елементи. Тільки інструменти Structure і Hierarchy задовольняють цей критерій. Також звертали увагу на підтримку найбільш поширених UML -діаграм. Детальний їх аналіз наведено в наступному підрозділі.

Останнім критерієм була можливість вбудувати інструмент в Android Studio, оскільки це офіційна IDE, створена компанією Google, яка також займає перше місце в топі IDE для розробки мобільних програм для ОС Android [20].

Результати проведеного аналізу відображені у табл. 1.1 та 1.2.

Таблиця 1.1 – Порівняння аналогів

Назва	Підтримка Kotlin	Облік специфіки Android - додатків	Підтримка діаграми класів
Structure і Hierarchy Android Studio	Так	Так	Ні
CodeIris	Тільки в експериментальному режимі	Ні	Відображення діаграми класів для всього проекту
Sequence Diagram Plugin	Так	Ні	Ні
PlantUML Diagram Generator	Ні	Ні	Є можливість відобразити діаграму, але її наповнення необхідно вказати самостійно
UML class diagrams із IntelliJ Idea	Так	Ні	Так. Відображаються не тільки класи, а й пакети, модулі та методи

Таблиця 1.2 – Порівняння аналогів

Назва	Підтримка діаграми компонентів	Підтримка діаграми пакетів	Можливість вбудувати інструмент у Android Studio
Structure та Hierarchy Android Studio	Ні	Ні	Є інтегрованим за замовчуванням інструментом IDE
CodeIris	Ні	Ні	Не підтримується у нових версіях IDE
Sequence Diagram Plugin	Ні	Ні	Так
PlantUML Diagram Generator	Ні	Ні	Так
UML class diagrams з IntelliJ Idea	Ні	Ні	Ні

Таким чином, вбудовані інструменти Structure та Hierarchy Android Studio найбільш повно задовольняють поставленим критеріям, проте вони не дозволяють відобразити інформацію про проект у вигляді діаграм та не показують його структуру. UML class diagrams з Intellij Idea Ultimate мають найбільшу кількість налаштувань відображення, на виході ми отримуємо діаграму класів, яку можна експортувати, але підтримка даного плагіна в Android Studio відсутня.

Облік специфіки Android -додатків при побудові діаграм, що відображають структуру проекту, не представлений в жодному з вивчених аналогів.

1.2 Визначення вимог до візуальних моделей

Візуальними моделями в Android-розробці є UML-діаграми [7], оскільки

вони дозволяють відобразити класи та модулі, їх взаємозв'язки та наповнення.

В роботі були вивчені результати опитування про практичне використання UML з метою визначити найбільш використовувані UML -діаграми, що використовуються для візуалізації структури вихідного коду проекту [21].

Результати представлені рис. 1.1.

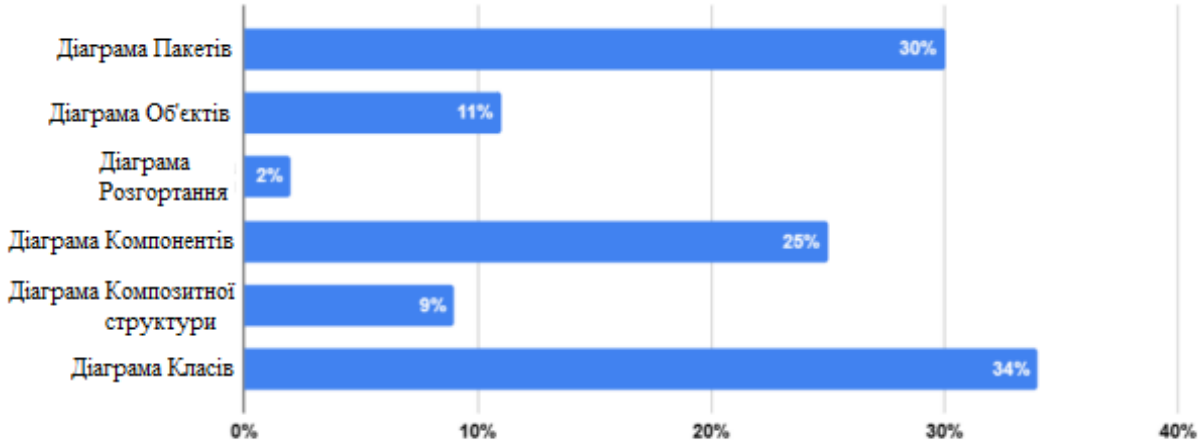

Рисунок 1.1 – Використання UML -діаграм для моделювання структури вихідного коду

Видно, що найбільше використовуються діаграми класів і пакетів (30-34%), потім йдуть діаграми компонентів (25%). Було ухвалено рішення розглядати ці три типи діаграм з урахуванням специфіки Android-додатків. Для цього було сформовано низку вимог.

Для діаграми класів:

– повинні бути представлені як Kotlin, так і Java-класи, оскільки обидві ці мови використовуються для розробки;

– треба відобразити наповнення цих класів, а також зв'язок між ними;

– при бажанні повинна бути можливість також відображати пакети в діаграмі;

– оскільки програма може складатися з кількох gradle-модулів [22], має бути можливість створити діаграму як для всього проекту, так і для конкретного модуля.

Приклад діаграми класів представлений на рис. 1.2.

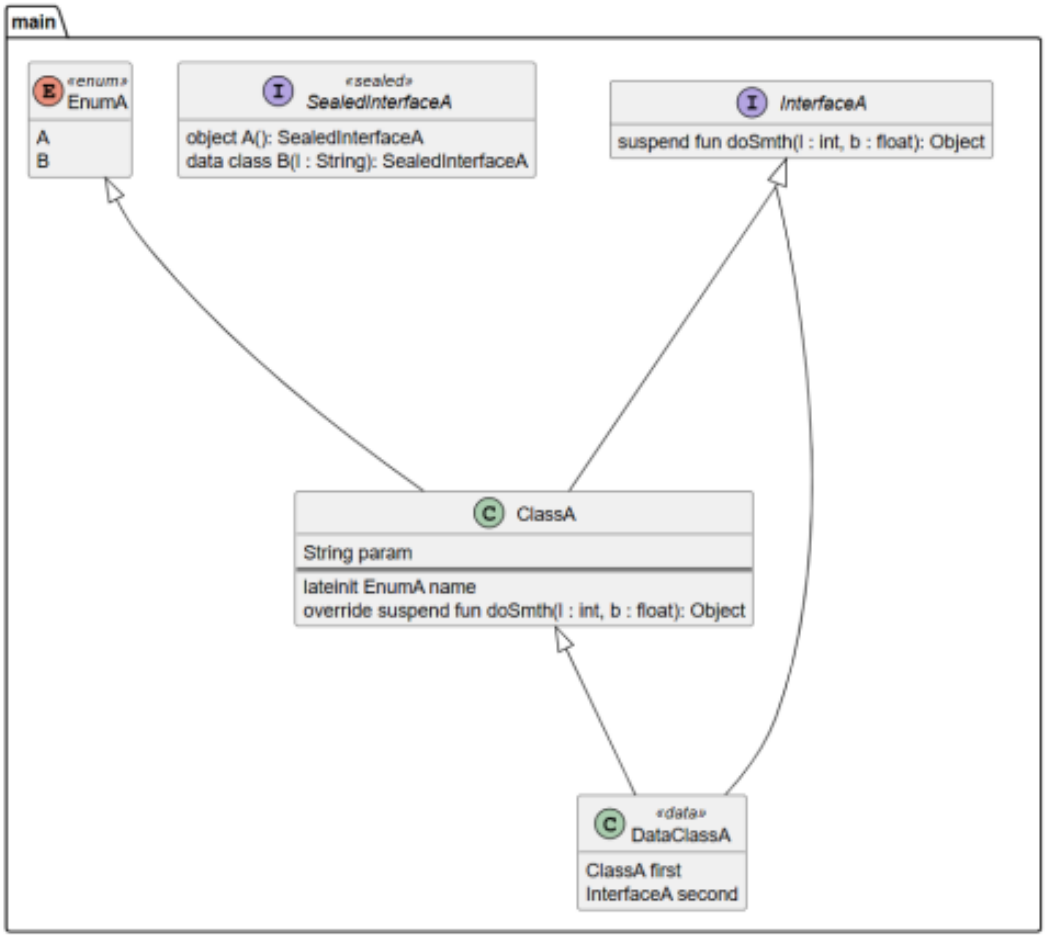

Рисунок 1.2 – Приклад діаграми класів

Для діаграми пакетів:

– як одиниця відображення розглядаються gradle-модулі проекту та їх взаємозв'язку;

– модулі і посилання групуються, ґрунтуючись на схожості зв'язків, для підвищення читаємості.

Приклад діаграми пакетів показано на рис. 1.3,

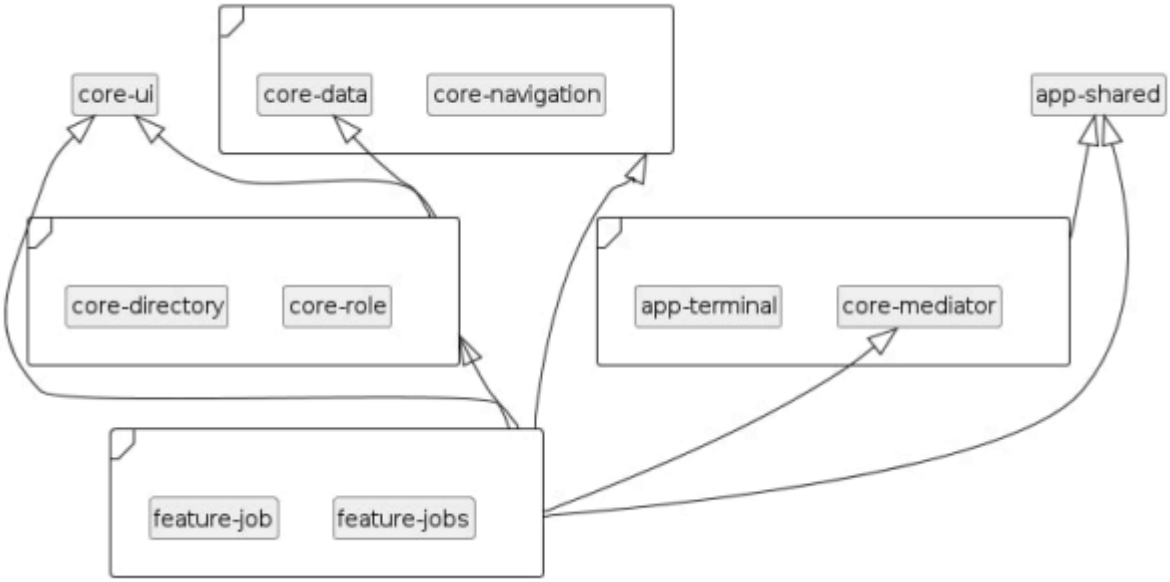

Рисунок 1.3 – Приклад діаграми пакетів

Діаграма компонентів була переосмислена і запропоновано новий вид діаграм - діаграма Android -компонентів, приклад якої представлений на рис. 1.4.

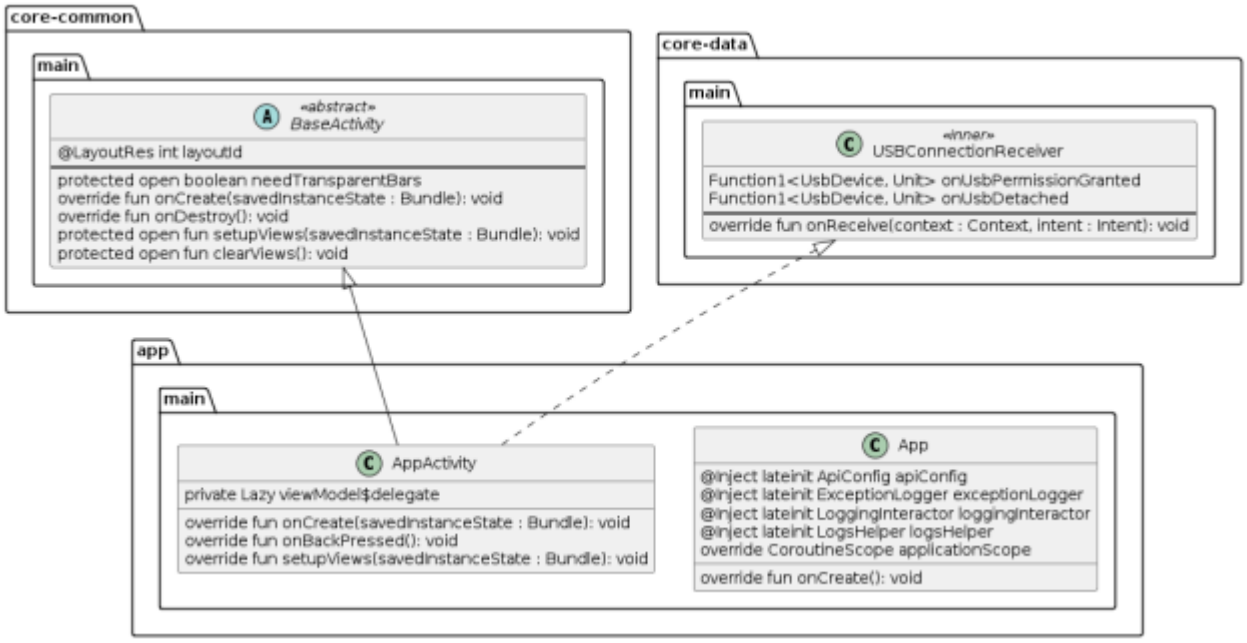

Рисунок 1.4 – Приклад діаграми Android-компонентів

На ній відображені зв'язки між різними спадкоємцями базових компонентів Android -додатків: Activity, Service, Broadcast Receiver та Content Provider [23]. Це

допомагає розробникам зрозуміти, які є залежності та зв'язки між компонентами та виявити потенційні вразливості у безпеці, а також наочно відображає основні вхідні точки програми. Для цього необхідно, щоб діаграма:

– відображала всіх спадкоємців основних компонентів, як прямих, і непрямих, оскільки кожен базовий компонент має багато варіацій;

– показувала наповнення та зв'язки отриманих класів, за аналогією з діаграмою класів, враховуючи те, що це може бути як Java, так і Kotlin -файл;

– враховувала транзитивні посилання між класами-спадкоємцями.

2 ТЕОРЕТИЧНА ЧАСТИНА

2.1 Концепція програмного рішення

В рамках роботи над кваліфікаційною роботою була розроблена концепція програмного рішення, що містить у собі три основні модулі: взаємодії з користувачем, збору інформації за вихідним кодом та генерації діаграм (рис. 2.1).

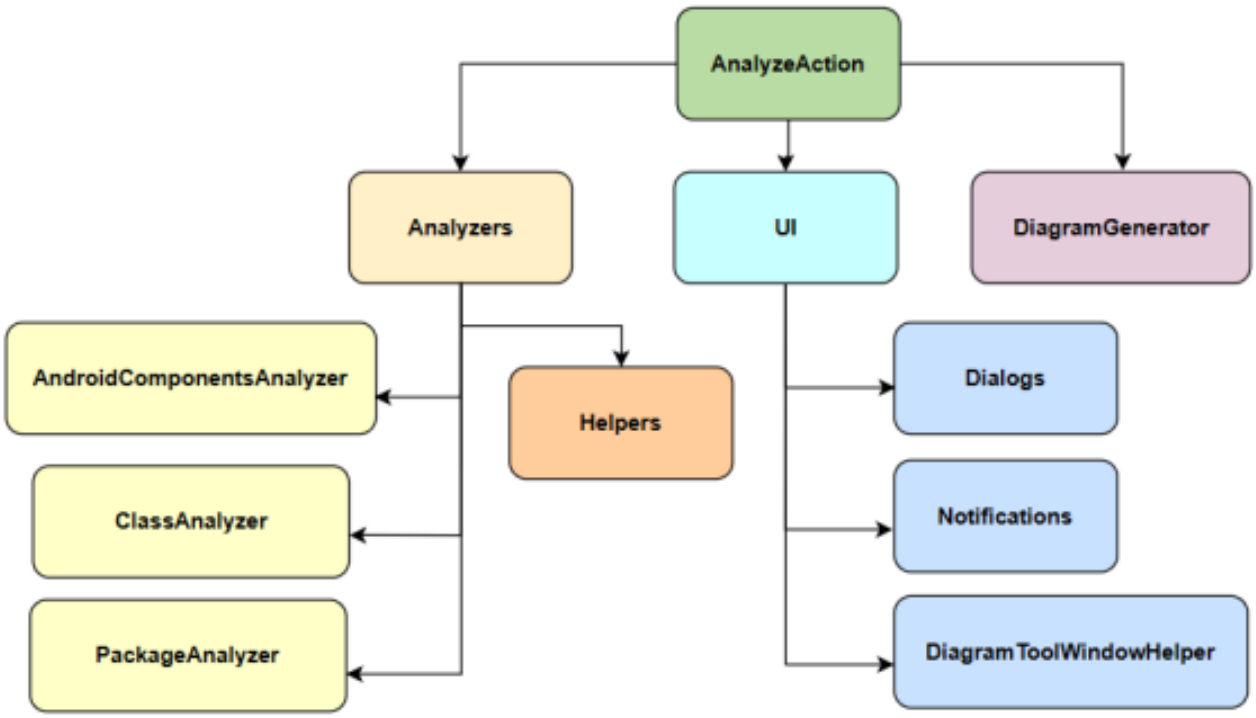

Рисунок 2.1 – Концепція програмного рішення

Після встановлення плагіна з jar або zip -архіву в меню Plugins налаштувань IDE, користувач стає доступний AnalyzeAction в меню Android Studio, який запускає аналіз проекту і генерацію UML -діаграми і є точкою входу. Також було налаштовано поєднання клавіш для запуску цієї дії.

Після завершення роботи плагіна користувачеві з'являється вікно зі структурною діаграмою, що відображає елементи проекту та взаємозв'язку між ними.

У модулі Analyzers були створені аналізатори, що інкапсулюють логіку зі збору інформації для кожного типу діаграм, а також класи із загальними допоміжними функціями.

В рамках модуля UI були опрацьовані діалог для налаштування параметрів діаграм, діалог вибору модуля, механізм показу повідомлень та повідомлень та вікно з результуючою діаграмою.

У модулі Diagram Generator було реалізовано алгоритм створення файлів puml і svg з інформації, зібраної у модулі Analyzer, враховуючи вибраний тип діаграми та налаштування відображення.

При розробці концепції ми зіткнулися з потребою в гнучкій структурі, що представляє інформацію, необхідну відображення конкретних діаграм. Тому був опрацьований клас Info, який відображено на рис. 2.2. Була розроблена впорядкована ієрархія для збільшення гнучкості при додаванні нових типів діаграм, у якій кожен із підкласів відповідає за зберігання інформації для відповідного типу. Також було реалізовано допоміжні класи для посилань, внутрішніх класів, полів та типів класів.

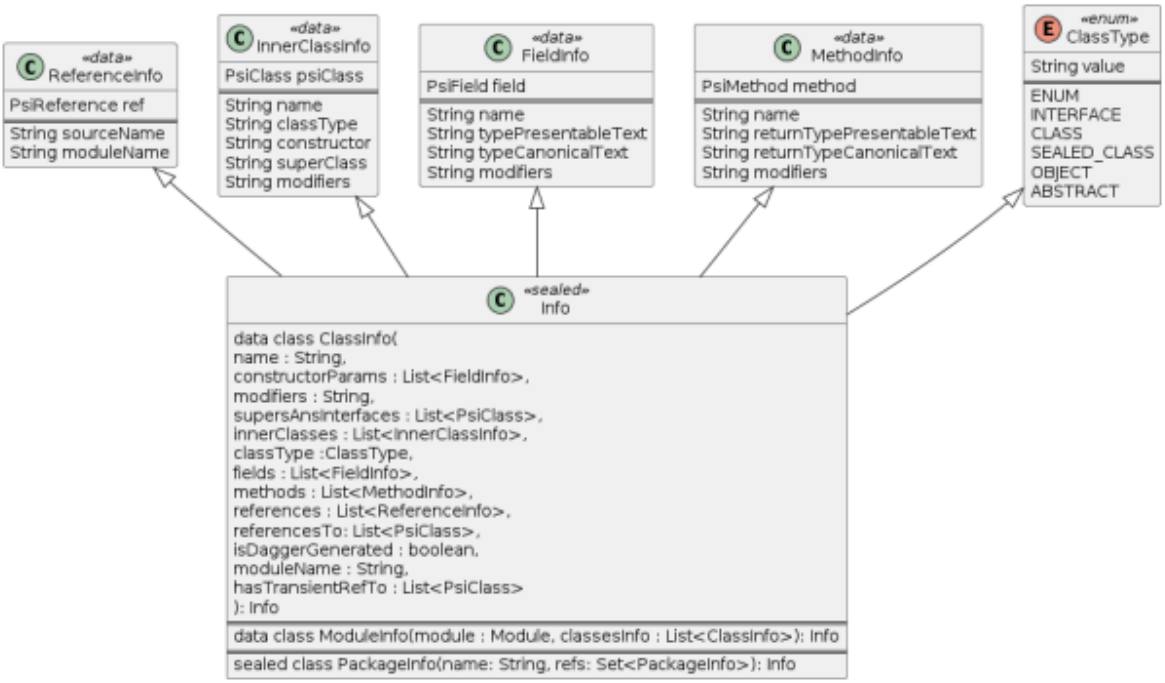

Рисунок 2.2 – Структура класу Info

Як Info для діаграми пакетів можуть бути як кластери, так і одиночні модулі, тому PackageInfo також є sealed -класом. Про його структуру буде докладніше розказано у розділі, присвяченому діаграмі пакетів.

2.2 Робота з PSI

Основним підходом для отримання інформації про вихідний код є обробка PSI дерева.

PSI – це спеціалізований шар платформи IntelliJ [24, 25]. Для кожного файлу IntelliJ будує дерево PSI, приклад якого наведено на рис. 2.3, це структура, що представляє вміст файлу як ієрархію елементів певної мови програмування. Взаємодія з цим деревом відбувається в рамках контексту Project, дозволяючи отримати доступ до структури коду та працювати з ним у межах цього проекту.

Рисунок 2.3 – Приклад частини збудованого PSI дерева

IntelliJ Platform не є самостійним продуктом, а є платформою для створення IDE з відкритим вихідним кодом. Зокрема на ній засновані IntelliJ IDEA та Android Studio. Саме IntelliJ Platform Plugin SDK використовувалося для розробки плагіна, що дозволило взаємодіяти з проектом та PSI безпосередньо.

При виклику action користувач відпрацьовує функцію actionPerformed з параметром event типу AnActionEvent. Таким чином, вхідними даними для роботи плагіна є PsiFile, що отримується з event, з якого виходить клас Project для подальшої роботи.

У процесі реалізації алгоритму роботи плагіна ми зіткнулися з потребою в різній обробці Kotlin та Java -файлів для отримання інформації про класи, що вимагало аналізу їхнього синтаксису. В результаті, відображеному в табл. 2.1, було отримано групи модифікаторів, які необхідно було одержувати для класів.

Таблиця 2.1 – Результати аналізу синтаксису Kotlin та Java

Мова програмування	Вимагають різних підходів під час обробки	Не впливають на обробку, але необхідно відображати	Модифікатори видимості
Kotlin	class, enum, interface, object, sealed	open, abstract, inline, data, inner, value, companion	public, private, protected, internal
Java	class, enum, interface, sealed	inner, final, abstract, static, record	public, private, protected

Алгоритм роботи плагіна, розробленого у роботі, представлений на рис. 2.4.

Отримувати безпосередньо текст модифікаторів ми не можемо, тому що необхідно фільтрувати модифікатори видимості та анотації, які до нього потрапляють.

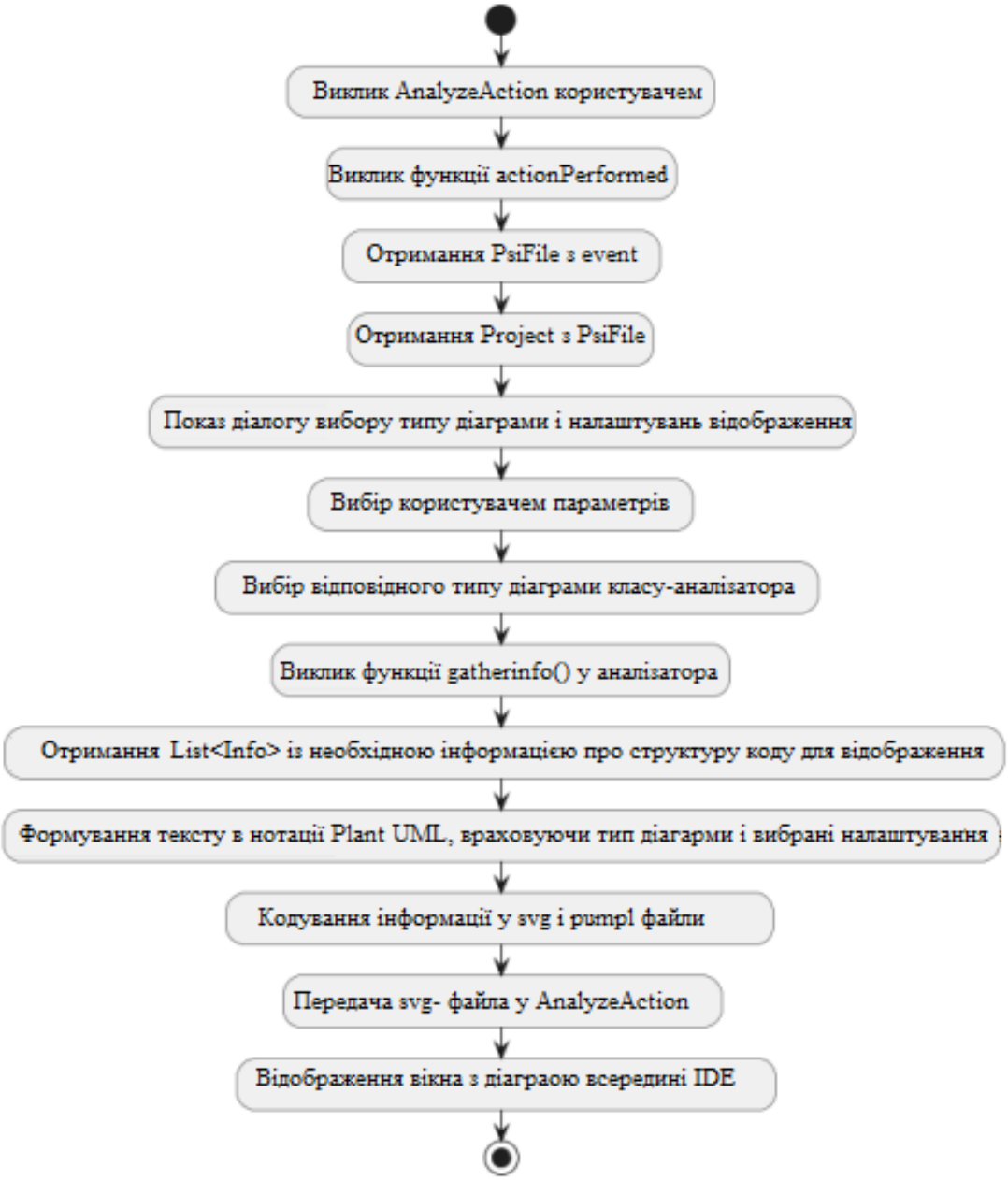

Рисунок 2.4 – Алгоритм роботи плагіна

Можливості безпосередньо отримати список токенів у модифікаторі в API немає, тому був розроблений алгоритм (рис. 2.5), що ґрунтується на обробці дочірніх елементів modifierList:

– для Java -класів було сформовано список допустимих токенів, ґрунтуючись на списку додаткових модифікаторів, які фільтруються з дочірніх вузлів списку модифікаторів класу. Інструкції у своїй ігноруються;

– для Kotlin-класів шукається оригінальний Kotlin -елемент. З нього виходить модифікатор видимості, щоб усунути його з результуючого списку. За

допомогою аналізу кількох Kotlin -класів була виділена наступна залежність: цікавим для нас типом модифікатором є KtModifierKeywordToken. З дочірніх вузлів списку модифікаторів класу фільтруються елементи цього типу і видаляється знайдений раніше модифікатор видимості.

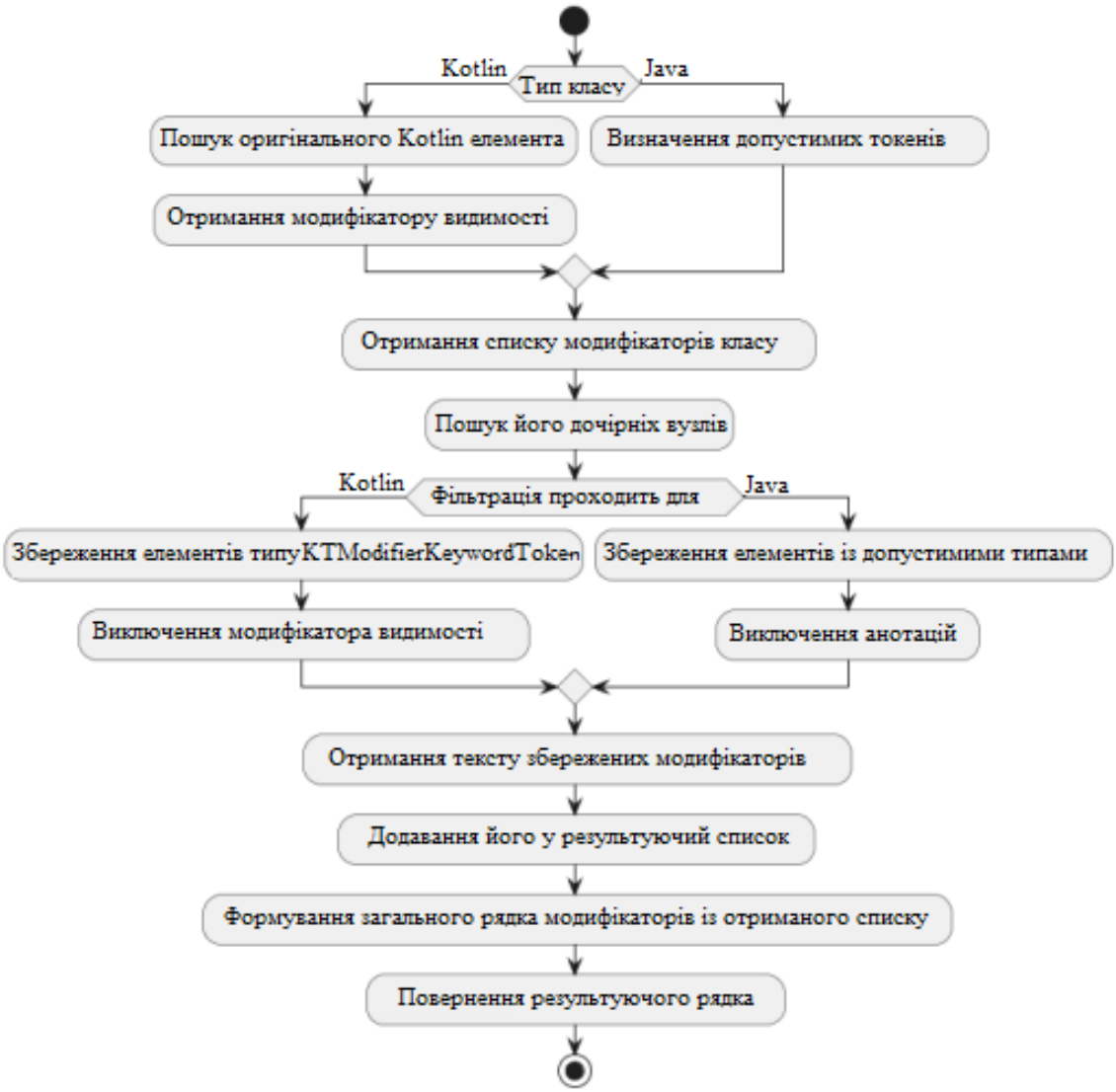

Рисунок 2.5 – Алгоритм отримання модифікаторів класу

Наступнй етап – визначення типу класу, ґрунтуючись на модифікаторах, які потребують різних підходів під час обробки. Однак до class, enum, interface, object, sealed були додані і abstract classes, оскільки вони вимагають окремих правок під час відображення.

Алгоритм визначення типу класу наведений на рис. 2.6. Для Java -класів тип

визначається з допомогою наявності відповідних PsiModifier, для Kotlin необхідно спочатку отримати оригінальний Kotlin- елемент, який може представляти оголошення класу чи об'єкта, для класів здійснюється перевірка наявності спеціальних ключових слів у модифікаторах, якими визначається тип.

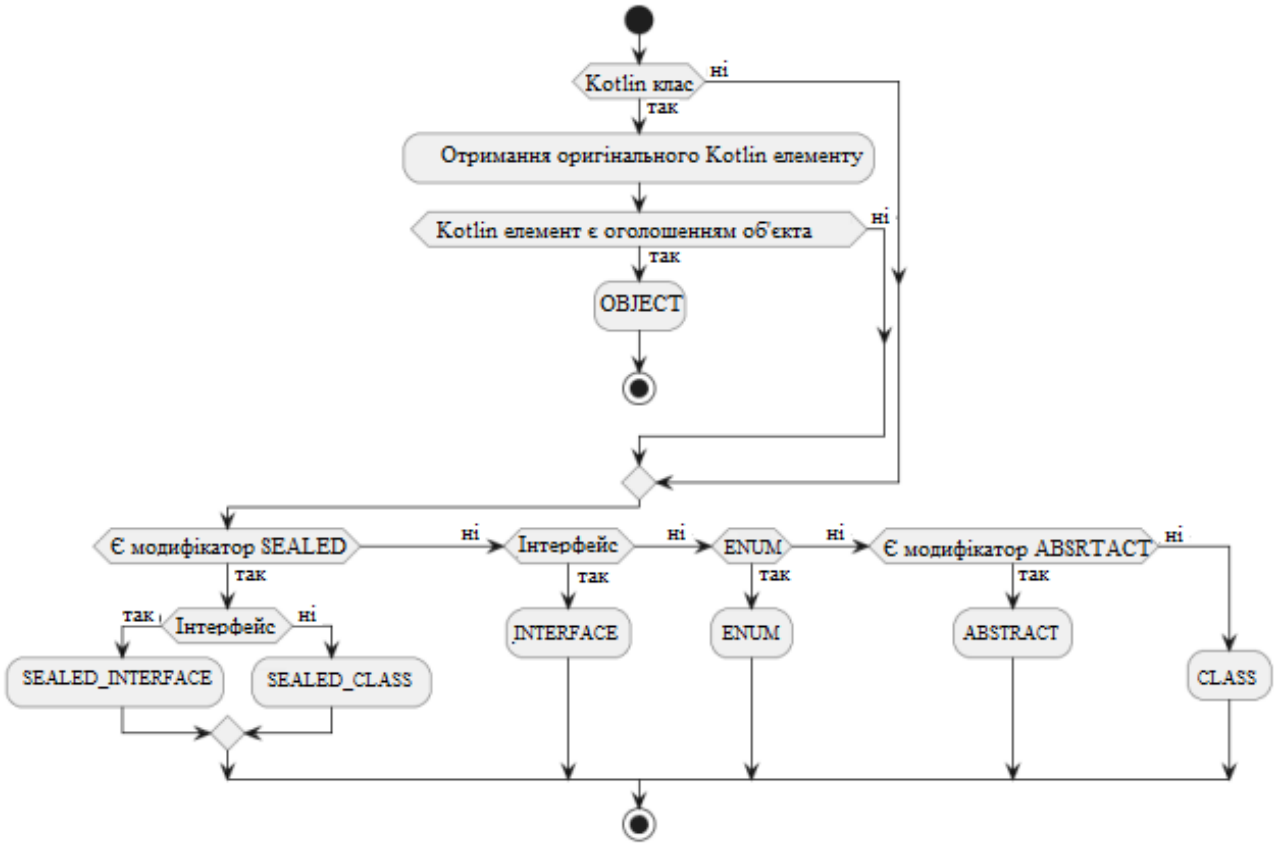

Рисунок 2.6 – Алгоритм визначення типу класу за його модифікаторами

Модуль аналізатора повинен збирати інформацію для різних видів діаграм, для цього були розроблені такі допоміжні методи:

– отримання PSI елементів, які посилаються на клас у області видимості проекту за допомогою пошуку вихідного файлу та модуля, в якому знаходиться цей файл;

– визначення, чи клас згенерованим DI фреймворком Dagger [26], за допомогою пошуку серед анотацій специфічних для даної бібліотеки оголошень Component, Module, Provides і Inject;

– отримання параметрів конструктора класу. Для кращого читання цього

параметри конструктора виділяються окремо від інших полів. Було проведено аналіз Java та Kotlin -класів, щоб зрозуміти, за якими критеріями можна визначити, чи є PsiField параметром з конструктора. Для Kotlin -класів необхідно було фільтрувати KtParameter, для Java -класів треба було шукати поля, які є PsiParameter;

– отримання полів класу, які не є згенерованими. Для цього необхідно було прибрати поля, які є параметрами конструктора або мають анотації згенерованих: "Generated", "AutoGenerated" Для Kotlin object також було прибрано поле INSTANCE, відповідне KtObjectDeclaration, оскільки воно не несе корисної інформації при відображенні. Для класів прибрані згенеровані функції equals, hashCode, toString і функції data dassів, і навіть гетери і сеттери для полів;

– отримання незгенерованих методів. У процесі аналізу було виявлено, що для Kotlin необхідно розглядати функції, чий оригінальний Kotlin -елемент є KtNamedFunction, за винятком цього, алгоритм однаковий для Kotlin і Java -класів: виходять всі методи, що містяться в класі, що виключаються, виключаються методи, що представляють конструктори, і з анотаціями "Generated";

– пошук модуля, що відповідає класу, за допомогою ModuleUtilCore PSI Module [27];

– отримання внутрішніх класів з конструктами та модифікаторами для sealed -класів, тому що це важлива для відображення інформація.

2.3 Збір інформації для діаграм

У цьому підрозділі описані алгоритми збору даних із вихідного коду проекту для подальшої побудови діаграм. Приклади результуючих діаграм класів та пакетів створювалися на основі відкритого проекту architecture-samples, написаного розробниками Google. Розглядалася гілка modularization, в якій представлений приклад багатомодульного Android -додатку.

Для демонстрації діаграми Android -компонентів використовувався проект Birday - додаток з відкритим вихідним кодом для запам'ятовування днів народження та подій, оскільки необхідно було наявність великої кількості взаємозалежних компонентів.

2.3.1 Діаграма класів

Для обробки класів, представлених у вигляді KotlinUltraLightClass та PsiClass, та їх мапінгу в Info з необхідною інформацією для діаграм, що відображають наповнення класів (діаграми класів та діаграми Android - компонентів), був створений клас ClassProcessor**.**

При роботі алгоритму запам'ятовується, які класи вже були оброблені, щоб уникнути повторів. Також виконується перевірка, чи не лежить цей файл у папках, які відповідають згенерованим файлам. Далі за допомогою наших допоміжних методів, описаних раніше, виходять:

- ім'я класу;
- список модифікаторів;
- перелік параметрів конструктора;
- список полів та методів;
- список посилань на цей клас та посилань на інші класи;
- список успадкованих класів та інтерфейсів;
- список внутрішніх класів;
- тип класу;
- ім'я модуля.

ClassAnalyzer відповідає за збирання інформації для діаграми класів. Був розроблений різний функціонал збору даних для Kotlin та Java-файлів за допомогою рефлексії, який використовує ClassProcessor для отримання детальної інформації. Спочатку алгоритм отримував усі Kotlin і Java-файли з проекту, послідовно обробляючи їх, проте постала проблема читання при побудові діаграми

класів для всього проекту, тому було прийнято рішення змінити алгоритм і дати можливість користувачеві фільтрувати відображення конкретного модуля. Для цього було сформовано наступний підхід (рис. 2.7):

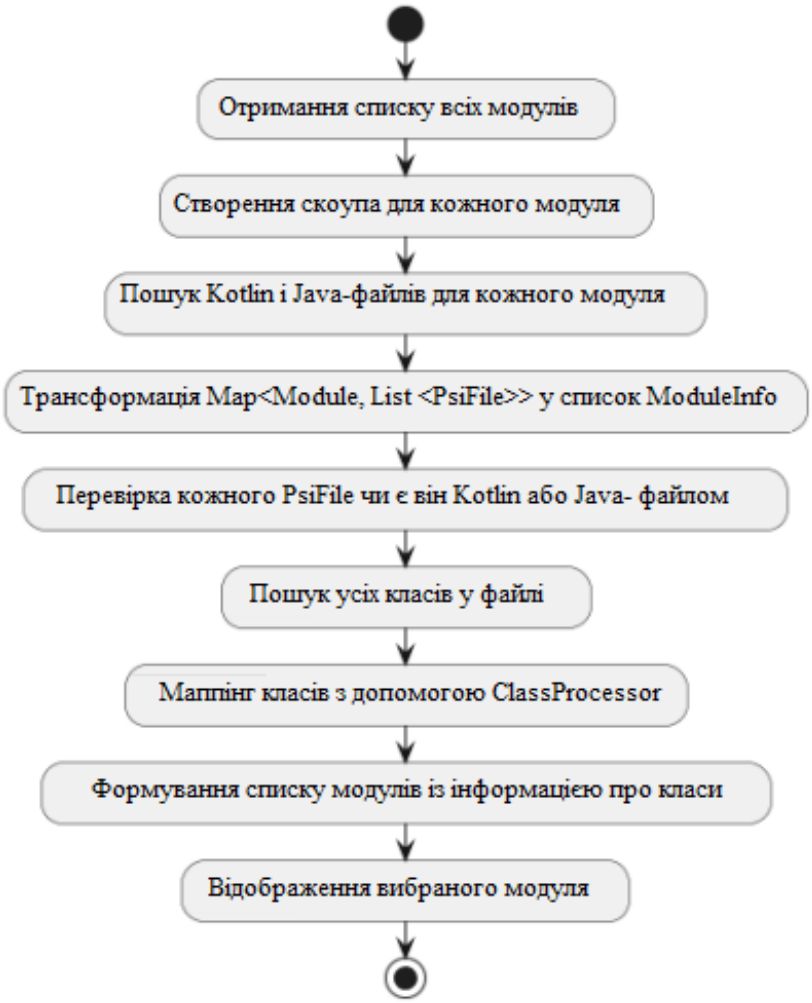

Рисунок 2.7 – Алгоритм роботи з модулями у межах діаграми класів

– отримується список усіх модулів, які у проекті з допомогою PsiModuleManager;

– для кожного модуля створюється відповідна область видимості для пошуку файлів;

– шукаються всі Kotlin та Java-файли для кожного модуля;

– отримана Map<Module, List<PsiFile>> трансформується в список ModuleInfo, за допомогою обробки PsiFile;

– для кожного PsiFile йде перевірка, чи є він Kotlin або Java-файлом, далі

шукаються всі класи, що знаходяться у файлі, та обробляються за допомогою ClassProcessor.

В результаті виходить список модулів зі списком інформації про класи всередині них. Така структура дозволяє при застосуванні вибору користувача модуля не запускати алгоритм ще раз, а просто знаходити потрібний модуль і відображати тільки його.

На рис. 2.8 представлений підсумковий алгоритм збору інформації для діаграми класів, приклад результуючої діаграми відображений на рис. 2.9.

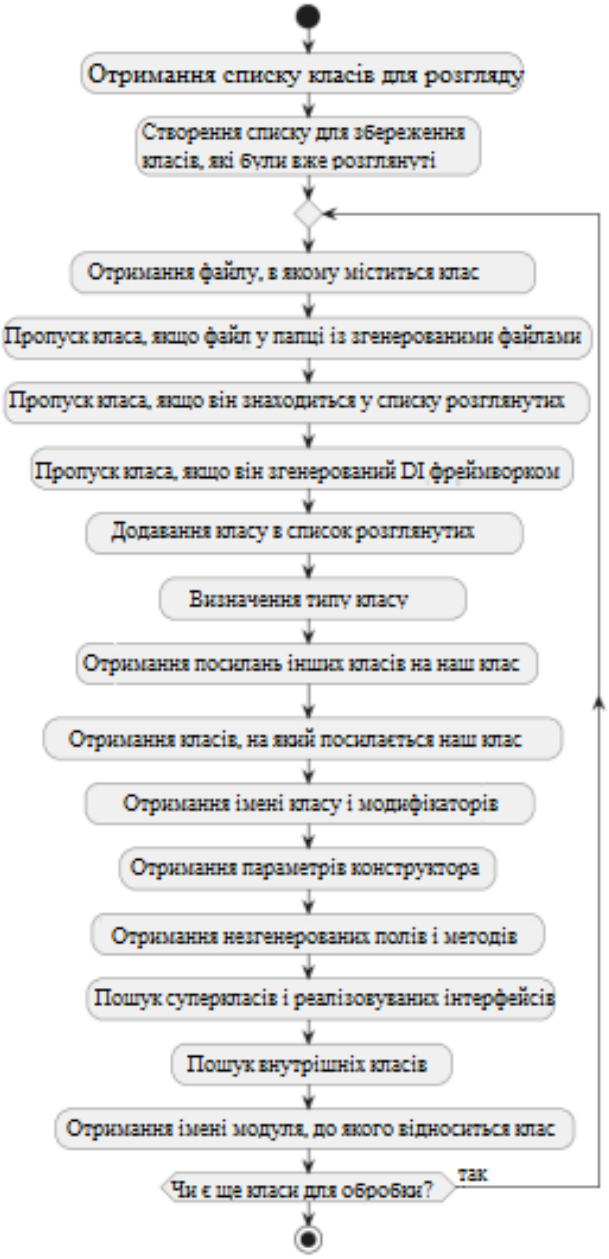

Рисунок 2.8 – Алгоритм збирання інформації для діаграми класів

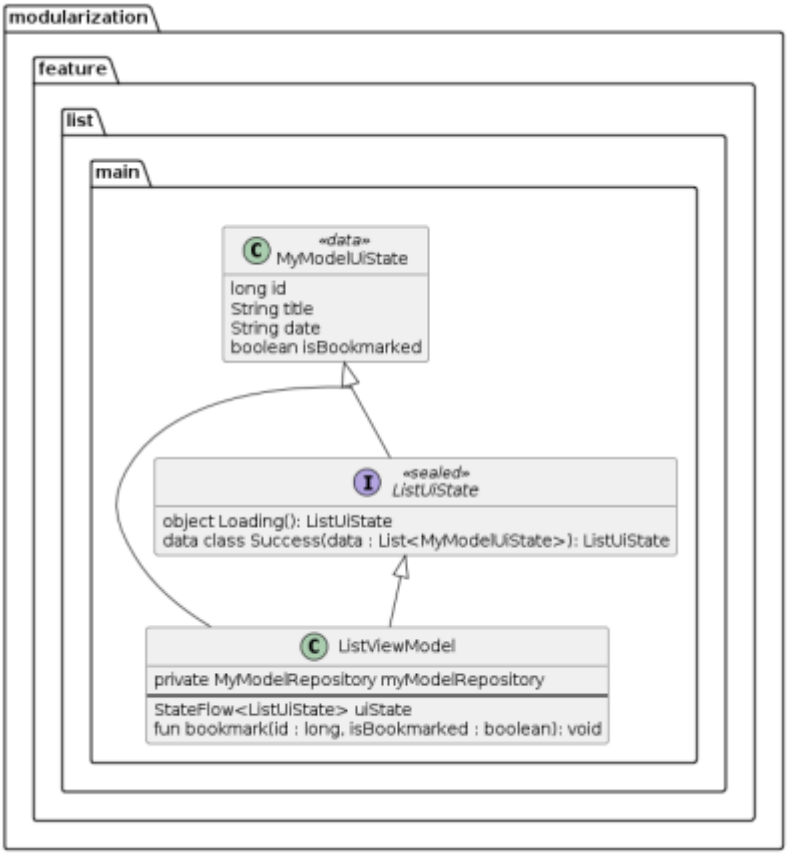

Рисунок 2.9 – Приклад результуючої діаграми класів

2.3.2 Діаграма Android-компонентів

AndroidComponentsAnalyzer відповідає за збирання інформації для діаграми Android-компонентів. Фундаментальною відмінністю від інших аналізаторів є те, що перед обробкою необхідно знайти всіх можливих спадкоємців, не обов'язково прямих, класів "android.app.Activity", "andmid.app. Service", "android.content.BmadcastReceiver", "android.content.ContentPmvider", "android.app.AppHcatюn". Для цього необхідно було знайти серед усіх класів, не тільки в рамках проекту, класи, що відповідають перерахованим базовим класам. Далі відбувається пошук всіх спадкоємців усередині області видимості проекту та отриманий список класів передається для подальшої обробки за допомогою ClassProcessor.

При тестуванні отриманого рішення був помічений випадок, коли спадкоємці класів пов'язані не безпосередньо, наприклад, коли один з них використовує інший клас, внутрішнім класом якого розглядається клас. Ця проблема було позначено як “проблема транзитивних посилань”, тобто. виникла потреба враховувати як прямі посилання між класами, так й посилання у вигляді інших класів, які є базовими компонентами. Іншою частиною проблеми було те, що ReferenceSearch [28], який використовується раніше, шукає лише посилання на аналізований клас, але не навпаки. Першою ітерацією цю проблему було вирішено для Kotlin -класів:

1) Для формування списку транзитивних посилань було створено допоміжний метод для перевірки повного імені класу, чи є спадкоємцем одних із зазначених раніше класів-компонентів, за допомогою PSI класу InheritanceUtil.

2) Було створено метод для рекурсивного збору транзитивних посилань на класи-спадкоємці компонентів із можливістю задати глибину пошуку для оптимізації обходу та уникнення циклів. Глибина пошуку визначається користувачем у стартовому діалозі, мінімальна глибина пошуку 2, так як пошук на першому рівні виявить тільки прямі посилання.

3) На вхід ця рекурсивна функція приймає екземпляр класу, який хочемо розглянути. Після аналізу була сформульована робоча гіпотеза, що ми можемо сказати, що один клас посилається на інший, якщо в ньому згадується тип класу. Отримати всі посилання на інші типи вдалося за допомогою KtTreeVisitorVoid за допомогою обходу всіх KtTypeReference .

4) Далі для кожної KtTypeReference виходить bindingContext, що дозволяє отримати повне ім'я класу, на який це посилання вказує.За допомогою JavaPsiFacade виходить PsiClass із відповідним ім'ям.

5) Йде перевірка, якщо цей клас є спадкоємцем класу-компонента, то він додається до результуючого списку і рекурсія далі не продовжується, інакше знову викликається рекурсивна функція для цього класу, при цьому збільшується лічильник вкладеності. Результат дзвінка додається до підсумкового списку.

6) Отже, рекурсія обривається, якщо досягнуто заданий рівень вкладеності

чи знайдено посилання спадкоємця класу-компонента.

7) Повертається результуючий перелік транзитивних посилань.

Однак при розробці алгоритму для обробки Java -класів вдалося створити рішення, спільне для обох мов, ґрунтуючись на першій ітерації (рис. 2.10):

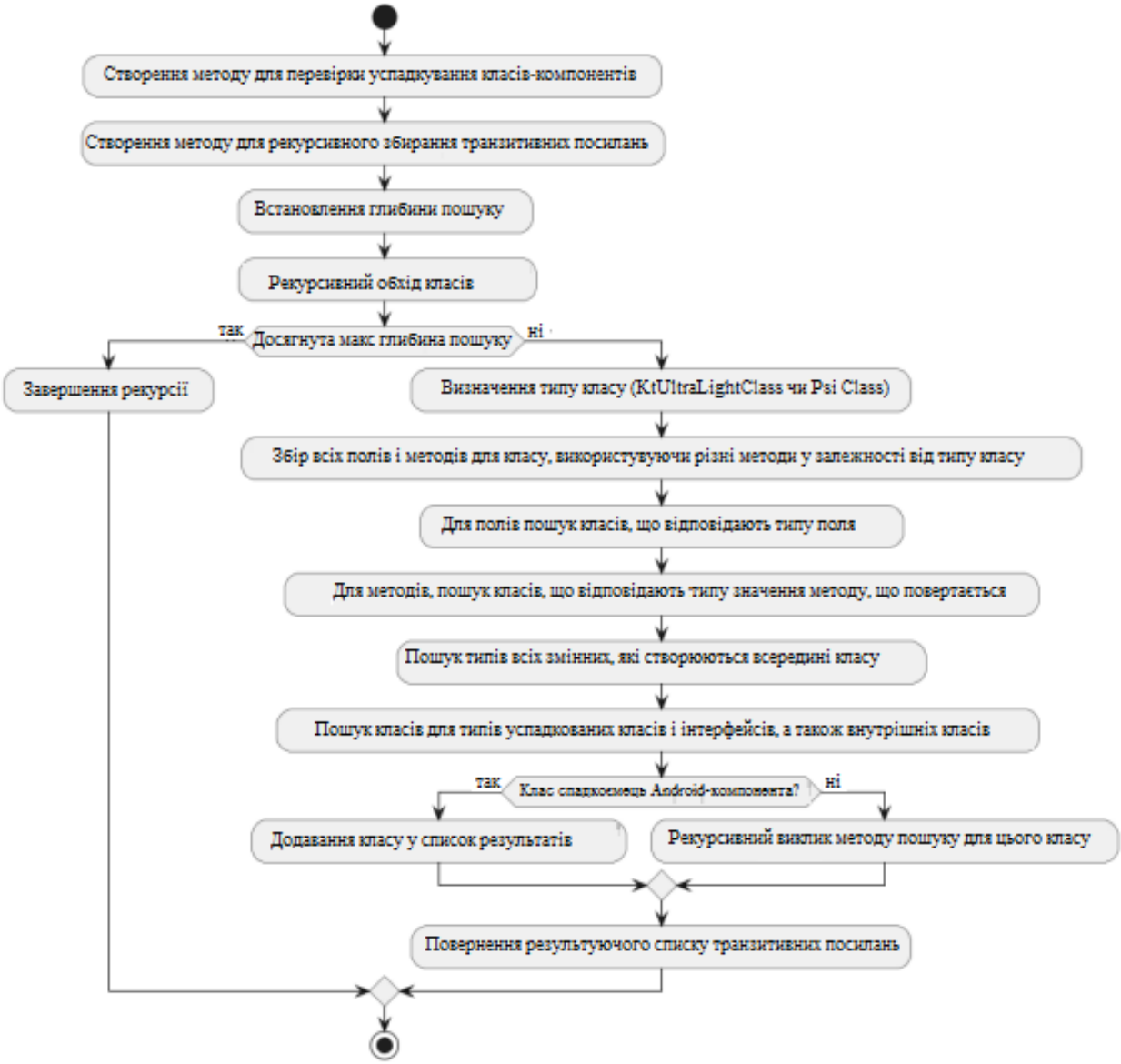

Рисунок 2.10 – Алгоритм збору транзитивних посилань для класів-спадкоємців Android-компонентів

– функція перевірки того, чи є клас спадкоємцем Android компоненти, була збережена;

– змінилася функція збору посилань на інші класи. Тепер збираються всі

поля та методи для класів, використовуючи різні методи залежно від того, чи це KtUltraLightClass або PsiClass. Для полів шукаються класи, відповідні типу поля, для методів йде пошук для типу повертається з методу, типів його змінних, а також типів всіх змінних, які створюються всередині методу. Аналогічно шукаються класи для типів успадкованих класів та інтерфейсів, і навіть внутрішніх класів;
отриманий список PsiClass, знайдений по всіх типах, що зустрічаються в класі, використовується в рекурсивному методі з пошуку транзитивних посилань, який також працює із заданою глибиною пошуку. Далі алгоритм відповідає першій ітерації, рекурсивно викликає сам себе, доки не отримає клас, який є спадкоємцем Andmid-компонента або не досягне заданої глибини.

Отже, отримавши інформацію про кожен клас-спадкоємець за допомогою ClassProcessor і список транзитивних посилань для нього, аналізатор віддає список Info для відображення діаграми (рис. 2.11).

Рисунок 2.11 – Приклад результуючої діаграми Android -компонентів

2.3.3 Діаграма пакетів

PackageAnalyzer відповідає за збір інформації для діаграми пакетів, яка в нотації Android додатків відповідає gradle-модулям. На першій ітерації використовувалася інформація, зібрана за допомогою ClassAnalyzer**,** аналізуючи, куди відносяться класи з посилань з ClassInfo. Особливість такого підходу в тому, що будуть відображені посилання лише за наявності використання класів з іншого модуля, але можуть бути ситуації, коли модуль знає про існування іншого, але не використовує його класи, тому цей алгоритм потребував доопрацювання.

Новий підхід полягає в аналізі модулів безпосередньо, не розглядаючи окремо класи всередині них. За допомогою ModuleManager формується список усіх модулів у проекті. В Android-розробці всі модулі з основним вихідним кодом представлені main -пакетом, тому із загального списку ми фільтруємо лише ті, що містять у собі “.main”. Для кожного такого модуля створюється ModuleRoot Manager для пошуку всіх orderEntries, що є всіма залежностями модуля, з них нас цікавлять тільки залежності на інші модулі. За допомогою налагодження було виявлено, що залежно від інших модулів представлені ModuleOrderEntry, тому вони фільтруються із загального списку і перетворюються на модулі.

Структура PackageInfo (рис. 2.12) є інформацією для побудови діаграми пакетів, разюче відрізняється від інших підтипів Info. Вона також є sealed -класом.

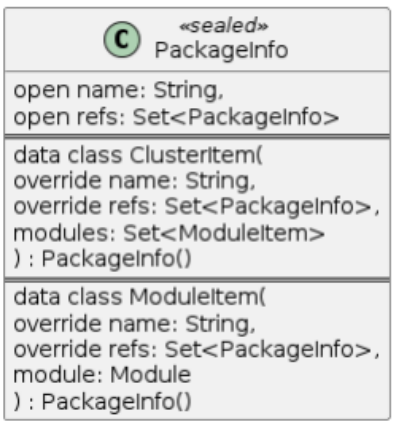

Рисунок 2.12 – Структура PackageInfo

Це рішення було прийнято для можливості кластеризації модулів та групування посилань, оскільки одиницею відображення може бути як одиничний модуль, так і кластер, і посилання може вказувати аналогічно модуль або кластер.

Далі настає етап кластеризації. Для цього було створено допоміжний клас RefsClustering. У ньому спочатку серед усіх модулів виділяються ті, які мають ідентичні посилання, далі модулі, що залишилися, поділяються на кластери за допомогою алгоритму DBSCAN [29] та бібліотеки SMILE [30]. Важливо тут те, що кластери з одним модулем та кластери з 0 загальних посилань ігноруються. В результаті ми отримуємо список кластерів та модулів, які не увійшли до жодного кластеру.

Наступним кроком є групування посилань. Була створена рекурсивна функція groupRefs(), яка приймає на вхід поточний аналізований PackageInfo і список всіх кластерів (без окремих модулів), який був отриманий на попередньому кроці. Далі кожного кластера перевіряється, чи містить список посилань поточного елемента все модулі кластера. Якщо так, ми замінюємо всі посилання на ці модулі одним посиланням на відповідний кластер.

Після того, як перевірка була виконана для кожного кластера, ми дивимося, чи є поточний PackageInfo кластером. Якщо так, то функція рекурсивно викликається для кожного модуля, що міститься в кластері.

Приклад результуючої діаграми відображено на рис. 2.13.

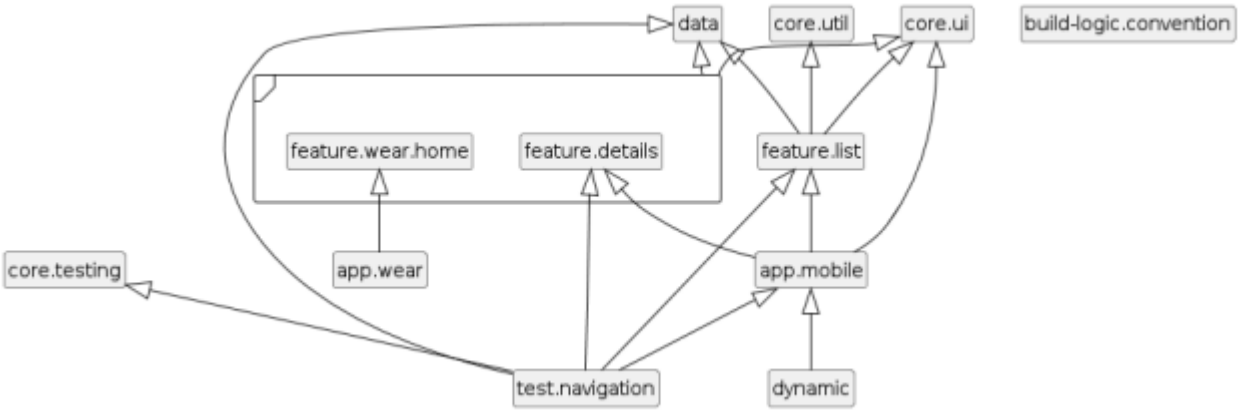

Рисунок 2.13 – Приклад результуючої діаграми пакетів

3 ПРАКТИЧНА ЧАСТИНА

3.1 Генерація діаграм

Для формування puml і svg було створено клас DiagramGenerator, у якому нами реалізовано алгоритм створення текстового представлення діаграм по List<Info>, зібраному аналізаторами, з обраного типу діаграми та налаштувань відображення.

Першим етапом виходить розташування на диску файлу, з якого був запущений плагін, щоб створити там поки що порожні файли для svg -діаграми і для puml -файла, якщо була раніше обрана відповідна опція.

Далі формується текст у нотації Plant Uml, залежно від типу діаграми.

Для діаграм Android -компонентів та діаграми класів викликається генерація діаграми класів за List<ClassInfo>. Відмінність лише в тому, що для діаграми класів необхідно скласти цей список виходячи з того, чи нас цікавить весь проект і потрібно об'єднати списки List<ClassInfo> з усіх модулів, або нас цікавить тільки обраний користувачем модуль і тільки його список.

Спочатку обробляється основна інформація про клас:

– записується інформація про модуль, у якому лежить клас. Так як ми отримуємо список послідовним об'єднанням модулів, є гарантія того, що класи з одного і того ж модуля йдуть поспіль, що дозволяє коректно визначати область пакета;

– записується тип класу, його ім'я та модифікатори, за наявності;

– потім послідовно обробляються всі параметри з конструктора, вказуються їх модифікатори, тип і ім'я;

– параметри конструктора відокремлюються від інших полів за допомогою подвійного рядка. Ігноруються поля з модифікатором “companion”, оскільки companion object відображається як внутрішній клас. Для enum полів не відображається їх тип, тільки ім'я для зручності читання. Відображення інших полів аналогічно до відображення параметрів конструктора;

– далі, після подвійного рядка, відображаються внутрішні класи як запис “innerClass $ {inner.name}". Винятком є лише спадкоємці sealed-класу, для них відображаються модифікатори, тип класу, конструктор з параметрами, що близько до звичного відображення в IDE.

Після інформації про клас обробляються його посилання. Для кожного суперкласу, інтерфейсу, класу, що посилається, і мети транзитивного посилання перевіряється, чи є вони серед усіх оброблених класів, далі визначається повне ім'я разом з модулем, до якого вони відносяться і відображається вірне відношення з класом.

У першій ітерації пакети для класів вказувалися лише за обраної користувачем опції їхнього відображення і за маппінгу посилань використовувалися лише імена самих класів. Але при тестуванні на різних проектах виявилася проблема з помилками відображеннями за наявності класів з однаковими іменами в різних проблемах. Якщо пакети відображалися, лише посилання були некоректні, але без них ми отримували помилку спроби оголосити нібито один і той же клас два рази. Для розв’язання такої проблеми завжди вказували пакети при створенні текстового представлення pumpl та використовували повне ім'я класу з ім'ям модуля у посиланнях. Якщо користувачу не потрібно відображення пакетів, то вони просто ховаються за допомогою стилю із зображення діаграми.

Для діаграми пакетів обробляються елементи списку Package.Info в залежності від їх типу. Якщо це кластер:

– він відображається як пакет без імені;

– усередині перераховуються його модулі;

– далі обробляються всі посилання, загальні для вього кластера

– для кожного внутрішнього модуля визначаються посилання, які не є спільними, та записуються окремо.

Модуль відображається як елемент без пакета, потім перераховуються всі посилання.

Результуючий рядок з усією інформацією кодується у svg та puml- файли.

Шлях до svg передається назад на місце виклику в AnalyzeAction для відображення всередині IDE.

Верхньорівневий алгоритм роботи DiagramGenerator наведено на рис. 3.1.

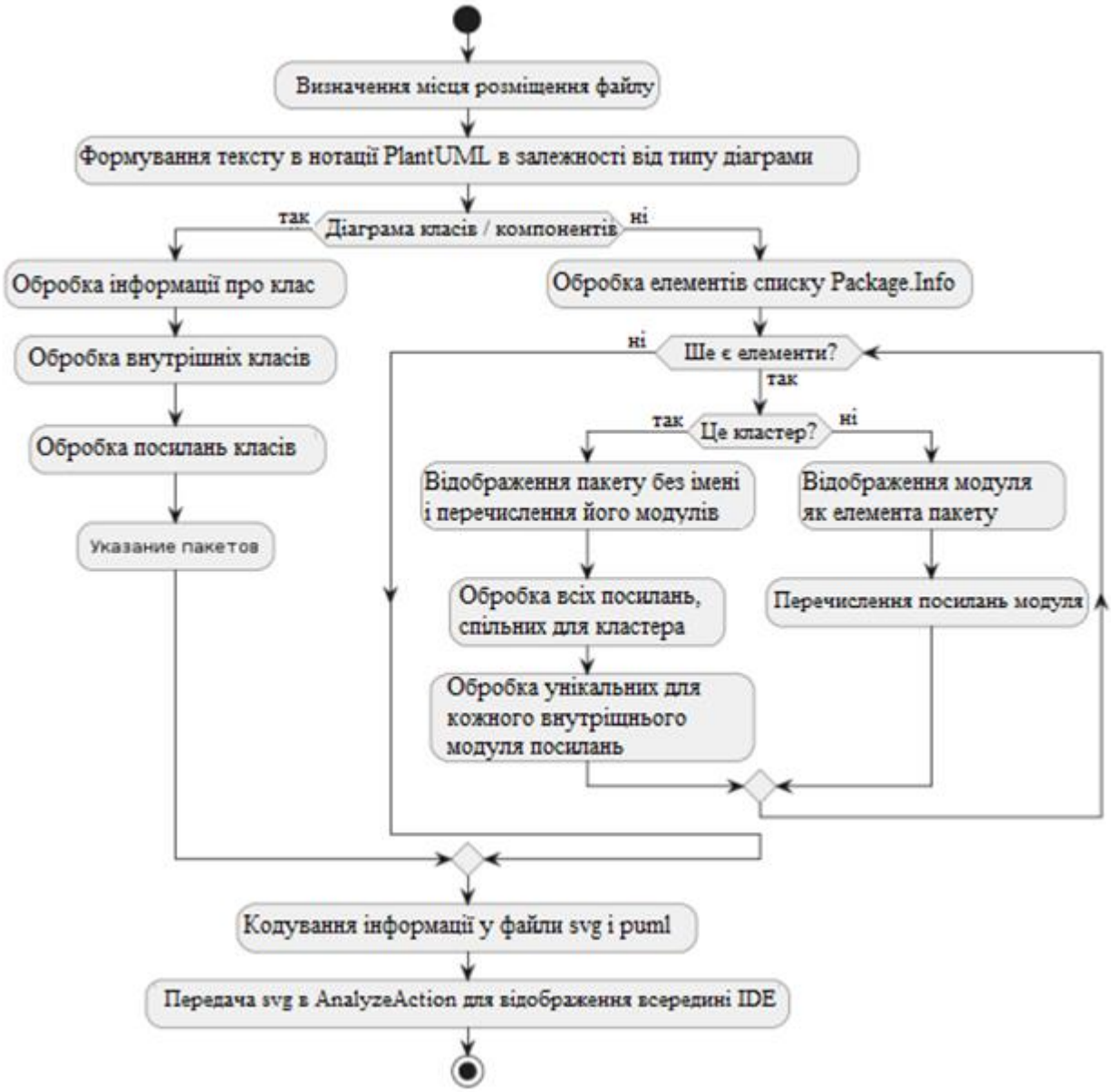

Рисунок 3.1 – Алгоритм роботи DiagramGenerator

3.2 Створення інтерфейсу плагіна

Вся робота плагіна побудована з використанням різних потоків для операцій читання, виконання обчислень та відображення інформації для оптимізації швидкості роботи.

Створено систему оповіщення користувача про помилки під час аналізу вихідного коду за допомогою діалогових повідомлень в IDE, реалізованих за допомогою Messages [31]. Аналогічні повідомлення з'являються під час старту (рис. 3.2) та завершення роботи плагіна з відповідними іконками.

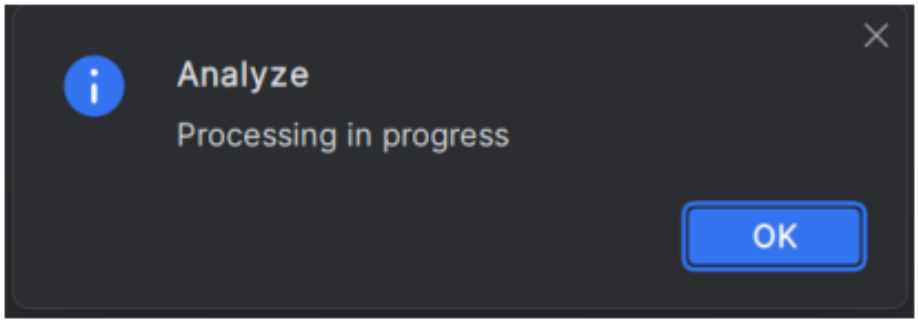

Рисунок 3.2 – Повідомлення, що інформує про початок аналізу

Час роботи плагіна вимірюється та відображається у повідомленні після завершення роботи, створеному за допомогою NotificationManager [32]. Приклад такого повідомлення наведено на рис. 3.3.

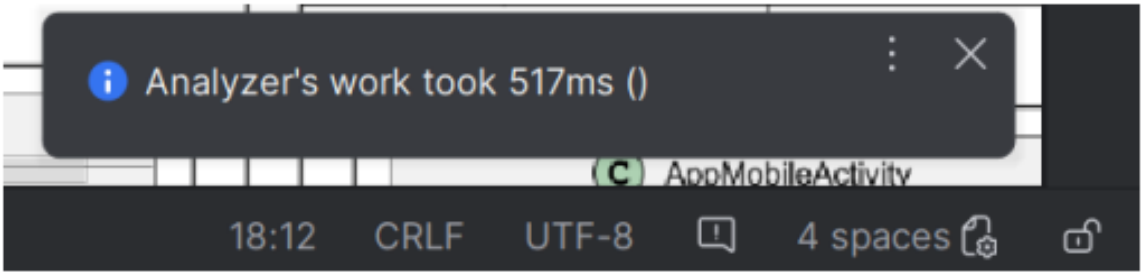

Рисунок 3.3 – Повідомлення з інформацією про час роботи плагіна

Також було розроблено два діалогові вікна. AnalyzeOptionsDialog є стартовим діалогом – першим, що бачить користувач при запуску AnalyzeAction. Воно було створено за допомогою класу DialogWrapper [33] та Kotlin UI DSL

Version 2 [34]. Приклад діалогу зображено на рис. 3.4.

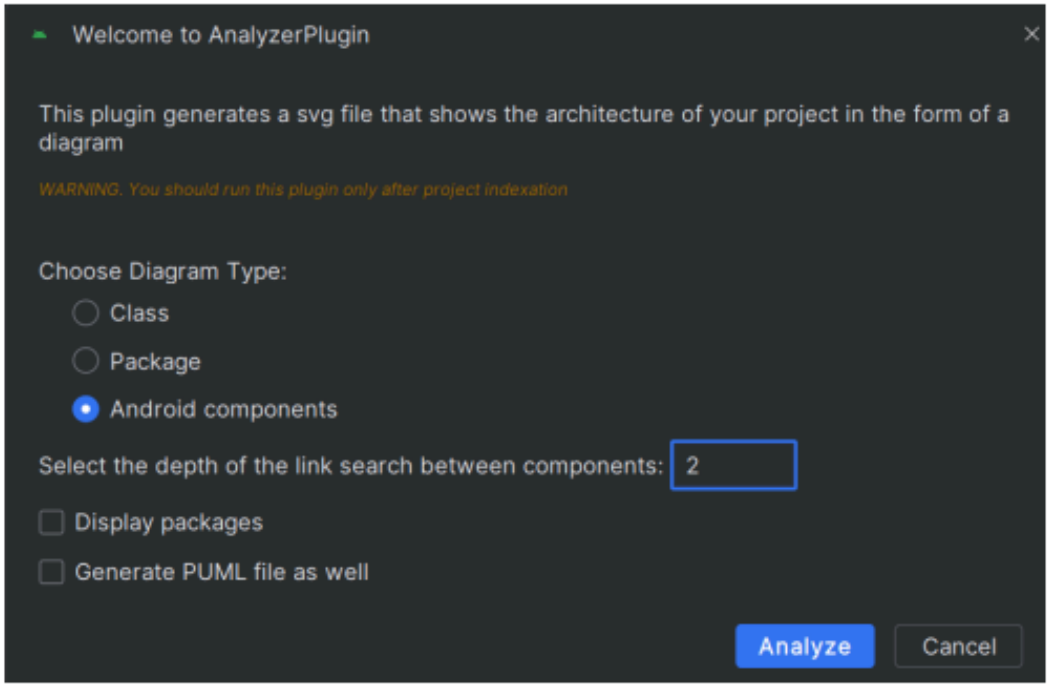

Рисунок 3.4 – Стартове діалогове вікно

У ньому представлені:

– попередження, що плагін варто запускати лише після індексації файлів, оскільки індексація забезпечує дані для PSI;

– група радіокнопок для вибору типу діаграми;

– числове текстове поле для вибору максимальної глибини пошуку для діаграми Android -компонентів;

– група чекбоксів для налаштування необхідності відображати пакети (недоступно для діаграми пакетів) та створювати puml- файл.

При натисканні користувачем Analyze формується DiagramResultState з вибраними параметрами і передається назад в AnalyzeAction для налаштування роботи аналізатора та генератора діаграм.

ChooseModuleDialog (рис. 3.5) необхідний для вибору користувачем модуля, для якого необхідно відобразити діаграму класів. На вхід він приймає список модулів проекту, якими створюється ComboBox [35] зі своїми іменами. При натисканні “Generate” вибраний модуль передається для обробки DiagramGenerator.

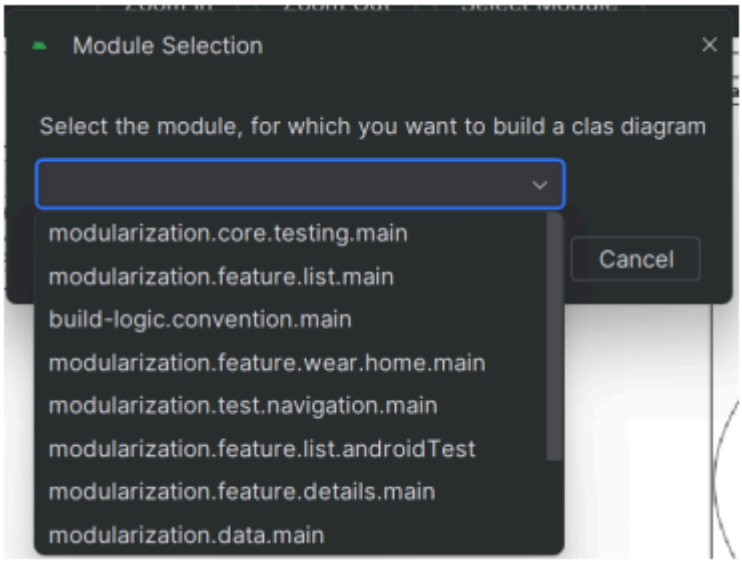

Рисунок 3.5 – Діалогове вікно вибору модуля для фільтрації діаграми класів

Результат роботи плагіна відображається у вбудованому вікні IDE, яке налаштовується за допомогою класу DiagramToolWindowHelper. У ньому є функція ініціалізації, що приймає виклик функції на відкриття діалогу з вибором модулів і ToolWindow [36], раніше зареєстрованому в AnalyzeAction за допомогою ToolWindowManager. При ініціалізації створюється спочатку порожня панель з текстом, що повідомляє користувачеві про те, що тут буде відображена результуюча діаграма, а також створюються інструменти для масштабування зображення та відображення svg всередині вікна.

По готовності svg із DiagramGenerator викликається функція оновлення панелі. У ній створюється панель, що прокручується, з svg за допомогою бібліотеки Apache Batik Swing [37], а також панель з кнопками масштабування зображення, що працюють за допомогою афінних перетворень [38]. Якщо користувач працює з діаграмою класів, то також показує кнопка для виклику діалогу вибору модуля (рис. 3.6), за допомогою переданої раніше функції. Після цього викликається перемальовка панелі для її оновлення, і користувач бачить результат роботи плагіна всередині вікна (рис. 3.7).

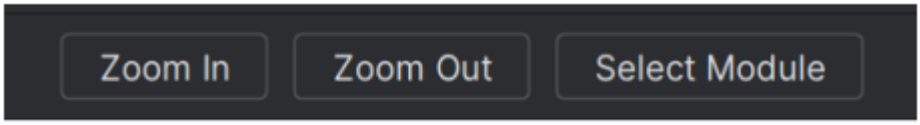

Рисунок 3.6 – Кнопки масштабування та кнопка вибору модуля

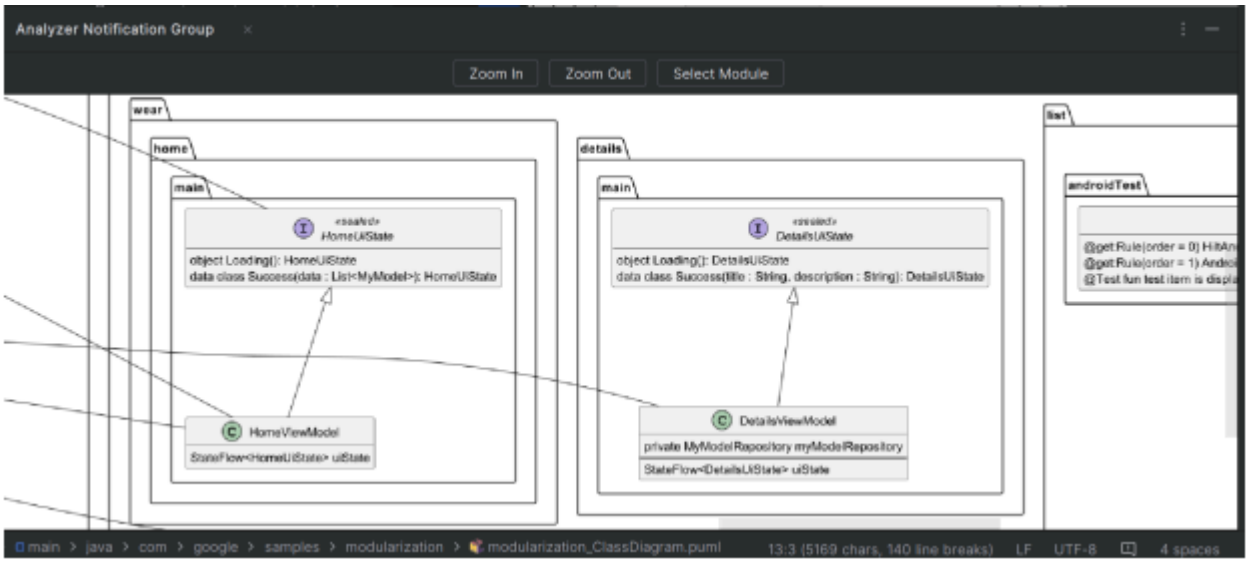

Рисунок 3.7 – Приклад результату роботи плагіна

Упродовж усієї роботи плагіна відображається прогрес усіх етапів за допомогою ProgressBar [39] та текстових повідомлень: процес обробки файлів, класів, пошуку спадкоємців, генерації діаграми тощо. Приклад індикатора прогресу представлений на рис. 3.8.

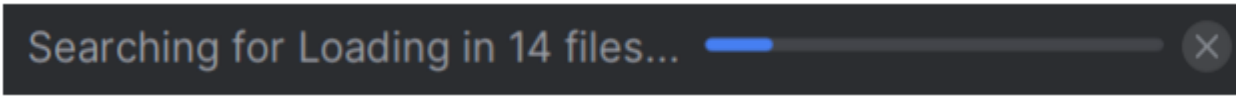

Рисунок 3.8 – Progress Indicator з повідомленням

РОЗДІЛ 4. БЕЗПЕКА ЖИТТЄДІЯЛЬНОСТІ, ОСНОВИ ОХОРОНИ ПРАЦІ

4.1 Класифікація шкідливих та небезпечних виробничих факторів

Шкідливий виробничий фактор – небажане явище, яке супроводжує виробничий процес і вплив якого на працюючого може призвести до погіршення самопочуття, зниження працездатності, захворювання, виробничо зумовленого чи професійного, і навіть смерті, як результату захворювання. Небезпечний виробничий фактор – небажане явище, яке супроводжує виробничий процес і дія якого за певних умов може призвести до травми або іншого раптового погіршення здоров'я працівника (гострого отруєння, гострого захворювання) і навіть до раптової смерті [40].

Поділ несприятливих чинників виробничого середовища на шкідливі та небезпечні зумовлене різним характером їх дії на людський організм, тим, що вони потребують різних заходів та засобів для боротьби з ними та профілактики викликаних ними ушкоджень, а також рядом причин організаційного характеру. В той же час між шкідливими та небезпечними виробничими факторами інколи важко провести чітку межу. Один і той же чинник може викликати травму і профзахворювання (наприклад, високий рівень іонізуючого або теплового випромінювання може викликати опік або навіть призвести до миттєвої смерті, а довготривала дія порівняно невисокого рівня цих же факторів – до хвороби; пилинка, що потрапила в око, спричиняє травму, а пил, що осідає в легенях, – захворювання, що зветься пневмоконіоз). Через це всі несприятливі виробничі чинники часто розглядаються як єдине поняття – небезпечний та шкідливий виробничий фактор (НШВФ) [41]. За своїм походженням та природою дії всі НШВФ можна поділити на 5 груп: фізичні, хімічні, біологічні, психофізіологічні та соціальні До фізичних НШВФ відносяться машини та механізми або їх елементи, а також вироби, матеріали, заготовки тощо, які рухаються або обертаються; конструкції, які руйнуються; системи, устаткування або елементи обладнання, які знаходяться під підвищеним тиском; підвищена запиленість та загазованість

повітря; підвищена або понижена температура повітря, поверхонь приміщення, обладнання, матеріалів; підвищені рівні шуму, вібрації, ультразвуку, інфразвуку; підвищений або понижений барометричний тиск та його різкі коливання; підвищена та понижена вологість; підвищена швидкість руху та підвищена іонізація повітря; підвищений рівень іонізуючих випромінювань; підвищене значення напруги в електричній мережі; підвищені рівні статичної електрики, електромагнітних випромінювань; підвищена напруженість електричного, магнітного полів; відсутність або нестача світла; недостатня освітленість робочої зони; підвищена яскравість світла; понижена контрастність; прямий та віддзеркалений блиск; підвищена пульсація світлового потоку; підвищені рівні ультрафіолетової та інфрачервоної радіації; гострі крайки, зачипки, шершавість на поверхні заготовок, інструментів та обладнання; розташування робочого місця на значній висоті відносно землі (підлоги); слизька підлога; невагомість.

Хімічні НШВФ:

- за характером дії на організм людини поділяються на токсичні, задушливі, наркотичні, подразнюючі, сенсибілізуючі, канцерогенні, мутагенні та такі, що впливають на репродуктивну функцію;

- за шляхами проникнення в організм людини поділяються на такі, що потрапляють через: 1) органи дихання; 2) шлунково-кишковий тракт; 3) шкіряні покриви та слизова оболонка;

- які перебувають у різному агрегатному стані: 1)твердому 2)газоподібному 3)рідкому.

Біологічні НШВФ – це: - патогенні мікроорганізми (бактерії, віруси, рикетсії, спірохети, грибки, найпростіші) та продукти їхньої життєдіяльності; - макроорганізми (тварини та рослини) та продукти їхньої життєдіяльності. До психофізіологічних НШВФ відносяться фізичні (статичні та динамічні) перевантаження і нервово-психічні перевантаження (розумове перенапруження, перенапруження аналізаторів, монотонність праці, емоційні перевантаження). 6 Соціальні НШВФ – це неякісна організація роботи, понаднормова робота, змушеність праці в колективі з поганими відносинами між його членами, соціальна

ізольованість з відривом від сім_ї, зміна біоритмів, незадоволеність роботою, фізична та/або словесна образа та її ризик, насильство та його ризик. Один і той же НШВФ за природою своєї дії може належати водночас до різних груп.

4.2 Вплив вібрації на людину

Вібрація - це механічні коливання пружних тіл або коливальні рухи механічних систем. Для людини вібрація є видом механічного впливу, який має негативні наслідки для організму [40].

Причиною появи вібрації є неврівноважені сили та ударні процеси в діючих механізмах. Створення високопродуктивних потужних машин і швидкісних транспортних засобів при одночасному зниженні їх матеріалоємності неминуче призводить до збільшення інтенсивності і розширення спектру вібраційних та віброакустичних полів. Цьому сприяє також широке використання в промисловості і будівництві високоефективних механізмів вібраційної та віброударної дії.

Дія вібрації може приводити до трансформування внутрішньої структури і поверхневих шарів матеріалів, зміни умов тертя і зносу на контактних поверхнях деталей машин, нагрівання конструкцій. Через вібрацію збільшуються динамічні навантаження в елементах конструкцій, стиках і сполученнях, знижується несуча здатність деталей, ініціюються тріщини, виникає руйнування обладнання. Усе це приводить до зниження строку служби устаткування, зростання імовірності аварійних ситуацій і зростання економічних витрат. Вважають, що 80% аварій в машинах і механізмах здійснюється внаслідок вібрації. Крім того, коливання конструкцій часто є джерелом небажаного шуму. Захист від вібрації є складною і багатоплановою в науково-технічному та важливою у соціальноекономічному відношеннях проблемою нашого суспільства [41].

Вплив вібрації на людину залежить від її спектрального складу, напрямку дії, прикладення, тривалості впливу, а також від індивідуальних особливостей людини.

При оцінці вібраційного впливу потрібно враховувати, що коливальні процеси притаманні живому організму. В основі серцевої діяльності і кровообігу та біострумів мозку лежать ритмічні коливання. Внутрішні органи людини можна розглядати як коливальні системи з пружними зв'язками. Частоти їх власних коливань лежать у діапазоні 3..6 Гц. Частоти власних коливань плечового пояса, стегон і голови щодо опорної поверхні (положення стоячи) складають 4...6 Гц, голови щодо плечей (положення сидячи) 25...30 Гц.

При впливі на людину зовнішніх коливань (хитавиці, струсів, вібрації) відбувається їхня взаємодія з внутрішніми хвильовими процесами, виникнення резонансних явищ. Так, зовнішні коливання частотою менш 0,7 Гц утворюють хитавицю і порушують у людини нормальну діяльність вестибулярного апарата. Інфразвукові коливання (менш 16 Гц), впливаючи на людину, пригнічують центральну нервову систему, викликаючи почуття тривоги, страху. При певній інтенсивності на частоті 6..7 Гц інфразвукові коливання, втягуючи у резонанс внутрішні органи і систему кровообігу, здатні викликати травми, розриви артерій, тощо [41].

Вібрація, що діє на людину, має широкий діапазон – від десятих часток одного до декількох тисяч Гц. Характерними ознаками шкідливого впливу вібрації на людину є можливі зміни у функціональному стані: підвищена втома, збільшення часу моторної реакції, порушення вестибулярної реакції. Медичними дослідженнями встановлено, що вібрація є подразником периферичних нервових закінчень, розташованих на ділянках тіла людини, що сприймають зовнішні коливання. Адекватним фізичним критерієм оцінки її впливу на організм людини є коливальна енергія, що виникає на поверхні контакту, а також енергія, поглинена тканинами і передана опорно-руховому апарату та іншим органам. У результаті впливу вібрації виникають нервовосудинні розлади, ураження кістково-суглобної та інших систем організму. Відзначаються, наприклад, зміни функції щитовидної залози, сечостатевої системи, шлунково-кишкового тракту. Так, медичні дослідження показали , що у працюючих в умовах вібрації відбуваються значні зміни кістковосуглобної системи, які виражаються у функціональній перебудові

кісткової тканини, регіональному остеопорозі, кістовидних утвореннях у кістках, асептичному некрозі кісток, хронічних переломах. Відзначається, що терміни виникнення змін у кістках у працівників вібраційних професій коливається в межах від 6-8 місяців до 2-5 років.

Шкідливість вібрації збільшується при одночасному впливі на людину таких факторів, як знижена температура, підвищений шум, запиленість повітря, тривала статична напруга тощо. Сучасна медицина розглядає виробничу вібрацію як могутній стрес-фактор, що має негативний вплив на психомоторну працездатність, емоційну сферу і розумову діяльність, підвищує ймовірність виникнення різних захворювань і нещасних випадків. Особливо небезпечний тривалий вплив вібрації для жіночого організму. Широкий комплекс патологічних відхилень, викликаний впливом вібрації на організм людини, кваліфікується як віброзахворювання [40].

Вібрація як фізичний чинник виробничого середовища спостерігається в металообробній, гірничодобувній, металобудівній, машинобудівній, авіаційній та інших галузях народного господарства. Джерелом вібрації можуть бути різні механізми, вібраційне устаткування, віброінструменти, акустичні системи, транспортні та сільськогосподарські машини.

Загальна вібрація поділяється на транспортну вібрацію, яка діє на людину на робочих місцях в транспортних засобах (трактори сільськогосподарські та промислові, самохідні сільськогосподарські машини (комбайни), тягачі, грейдери ті інші); транспортно-технологічну вібрацію, яка діє на людину на робочих місцях машин з обмеженою рухливістю (екскаватори, крани промислові та будівельні, гірничі комбайни, транспорт виробничих приміщень та інші) та технологічну вібрацію, яка діє на людину на робочих місцях стаціонарних машин чи передається на робочі, де немає джерел вібрації (верстати та метало-деревообробне, пресувально-ковальське обладнання, ливарні машини, електричні машини, насосні агрегати та вентилятори, обладнання для буріння свердловин, бурові верстати, машини для тваринництва, очищення та сортування зерна (у тому числі сушарні), обладнання промисловості будматеріалів (крім бетоноукладачів), установки хімічної та нафтохімічної промисловості та інші.

Оператори машин, які зазнають у процесі трудової діяльності впливу вібрації, підлягають попереднім та періодичним медичним оглядам відповідно до Порядку проведення медичних оглядів працівників певних категорій, затвердженого Наказом МОЗ України від 21.05.2007 р. №246. Обов'язкові попередні (під час прийняття на роботу) та періодичні (протягом трудової діяльності) медичні огляди дозволять визначити стан здоров'я працівника та можливість виконання без погіршення стану здоров'я професійних обов'язків, своєчасно виявити ранні ознаки хронічного професійного захворювання, забезпечує динамічне спостереження за станом здоров`я в умовах дії шкідливих та небезпечних факторів і трудового процесу, вирішує питання щодо можливості продовжувати роботу в умовах дії шкідливих та небезпечних факторів і трудового процесу [40].

За результатами періодичних медичних оглядів роботодавець забезпечує проведення відповідних оздоровчих заходів Заключного акта у повному обсязі та усуває причини, що призводять до професійних захворювань. Організовує проведення лабораторних досліджень умов праці на робочих місцях та вживає заходів до усунення небезпечних і шкідливих для здоров'я виробничих факторів.

До роботи операторами машин допускаються особи не молодші 18 років, які пройшли попередній медичний огляд, мають відповідну кваліфікацію та ознайомлені з характером впливу вібрації на організм.

ВИСНОВОК

В результаті виконання роботи було створено програмний засіб візуалізації структури Android -проекту.

Для досягнення поставленої мети було вирішено такі завдання:

– було проведено аналіз найчастіше використовуваних діаграм для моделювання структури вихідного коду. Для реалізації у програмному рішенні були обрані діаграми класів, пакетів та компонентів;

– розроблено тип діаграм, що відображає взаємозв'язки основних компонентів Android -додатків. На ній представлені взаємозв'язки між різними спадкоємцями базових компонентів Android -додатків, що полегшує завдання розробників у розумінні залежностей та взаємодій між компонентами, допомагає виявляти потенційні вразливості у системі безпеки;

– розроблено метод збору інформації про класи для Kotlin і Java - файлів, для цього був проведений аналіз синтаксису обох мов і реалізовані методи для роботи з PSI, що дозволяють отримувати інформацію про модифікаторів класу, його тип, посилання, параметри конструктора, полях і методах, а також про модуль, до якого відноситься клас;

– розроблено метод збору інформації про модулі у проекті за допомогою аналізу зв'язків gradle -модулів. Також реалізовано механізм кластеризації модулів та їх посилань шляхом виявлення модулів з ідентичними посиланнями та використання алгоритму DBSCAN;

– розроблено метод отримання посилань на класи за допомогою ReferenceSearch також під час розробки діаграми компонентів Android був реалізований метод рекурсивного збору транзитивних посилань на класи-спадкоємці, заснований на гіпотезі, що клас посилається на інший, якщо в ньому згадується тип цього класу;

– створено генератор діаграм, що дозволяє відображати діаграми класів, пакетів та Android -компонентів за допомогою обробки інформації від класів

аналізаторів. При створенні діаграм для кожного класу записується інформація про модуль, тип, ім'я та модифікатори класу, обробляються параметри конструктора та внутрішні класи. Для кожного суперкласу, інтерфейсу та посилань перевіряється їх наявність серед усіх оброблених класів. Для створення діаграми пакетів обробляються посилання модулів.

Розроблене в рамках даної роботи рішення дозволить створювати ряд діаграм за кодом Android -проекту, ілюструючи його структуру.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Perry, Dewayne E., Wolf Alexander L. Foundations for the study of software architecture // ACM SIGSOFT Software engineering notes. 1992. Vol. 17, № 4. P. 40-52.

2. Garlan D., Shaw M. An introduction to software architecture // Advances in software engineering and knowledge engineering. 1993. P. 1-39.

3. Clements P., Garlan D., Little R., Nord R., Stafford J. Documenting software architectures: views and beyond // Proceedings of the 25th International Conference on Software Engineering. 2003, May. P. 740-741.

4. Guide to Android app modularization. – URL: https://developer.android.com/topic/modularization (дата звертання: 10.04.2025).

5. Minelli R., Mocci A., Lanza M. I Know What You Did Last Summer - An Investigation of How Developers Spend Their Time // Proceedings of the 23rd IEEE International Conference on Program Comprehension. Florence, Italy, 2015. P. 25-35.

6. Schreiber A., Nafeie L., Baranowski A., Seipel P., Misiak M. Visualization of Software Architectures in Virtual Reality and Augmented Reality // Proceedings of the IEEE Aerospace Conference. Big Sky, MT, USA, 2019. P. 1-12.

7. Посібник із побудови схем UML і моделювання баз даних. – URL: https://www.microsoft.com/uk-ua/microsoft-365/business-insights-ideas/resources/guide-to-uml-diagramming-and-database-modeling (дата звертання: 10.04.2025).

8. Source file structure. JetBrains: Essential tools for software developers and teams. – URL: https://www.jetbrains.com/help/idea/viewing-structure-of-a-source-file.html (дата звертання: 12.04.2025).

9. Source code hierarchy. JetBrains: Essential tools for software developers and teams. – URL: https://www.jetbrains.com/help/idea/2021.3/viewing-structure-and-hierarchy -of-the-source-code.html (дата звертання: 12.04.2025).

10. CodeIris. Visualize Your Software. – URL: http://plugin.codeiris.com/ (дата звертання: 14.04.2025).

11. CodeIris - IntelliJ IDEs Plugin | Marketplace. – URL: https://plugins.jetbrains.com/plugin/7324-code-iris (дата звертання: 14.04.2025).

12. SequenceDiagram - IntelliJ IDEs Plugin | Marketplace. – URL: https://plugins.jetbrains.com/plugin/8286-sequencediagram (дата звертання: 15.04.2025).

13. SequencePlugin for IntelliJ IDEA. – URL: https://vanco.github.io/SequencePlugin/ (дата звертання: 16.04.2025).

14. PlantUML Diagram Generator - IntelliJ IDEs Plugin. — URL: https://plugins.jetbrains.com/plugin/15991-plantuml-diagram-generator (дата звертання: 16.04.2025).

15. UML class diagrams | IntelliJ IDEA Documentation. — URL: https://www.jetbrains.com/help/idea/2022.2/class-diagram.html (дата звертання: 16.04.2025).

16. Rebrof D. Class diagrams in android projects. — URL: http12.04.2025://medium.com/@denisrebrof/class-diagrams-in-android-projects-ee34ff986fcd (дата звертання: 18.04.2025).

17. Stack Overflow Developer Survey. – URL: https://survey.stackoverflow.co/2023/#technology-most-popular-technologie s (дата звертання: 19.04.2025).

18. View the structure of code by using different tool windows. — URL: https://learn.microsoft.com/en-gb/visualstudio/ide/viewing-the-structure-of-code?view=vs-2022 (дата звертання: 19.04.2025).

19. Kotlin vs Java: the 12 differences you should know. — URL: https://www.imaginarycloud.com/blog/kotlin-vs-java (дата звертання: 19.04.2025).

20. 7 TOP IDES FOR ANDROID DEVELOPMENT IN 2024. — URL: https://lanars.com/blog/ide-for-android-development (дата звертання: 19.04.2025).

21. Ozkaya M., Erata F. A survey on the practical use of UML for different software architecture viewpoints // Information and Software Technology. 2020. Vol. 121. — URL: https://www.sciencedirect.com/science/article/pii/ S0950584920300252.

22. Structuring Projects with Gradle. — URL: https://docs.gradle.org/current/userguide/multi_project_builds.html (дата звертання: 20.04.2025).

23. Application fundamentals. – URL: https://developer.android.com/guide/components/fundamentals#Components (дата звертання: 20.04.2025).

24. The IntelliJ Platform | IntelliJ Platform Plugin SDK (jetbrains.com). — URL: https://plugins.jetbrains.com/docs/intellij/intellij-platform.html#plugins (дата звертання: 24.04.2025).

25. Program Structure Interface (PSI) | IntelliJ Platform Plugin SDK (jetbrains.com). — URL: https://plugins.jetbrains.com/docs/intellij/psi.html (дата звертання: 25.04.2025).

26. Dagger. — URL: https://dagger.dev/ (дата звертання: 25.04.2025).

27. Module | IntelliJ Platform Plugin SDK (jetbrains.com). — URL: https://plugins.jetbrains.com/docs/intellij/module.html (дата звертання: 26.04.2025).

28. PSI References | IntelliJ Platform Plugin SDK (jetbrains.com). — URL: https://plugins.jetbrains.com/docs/intellij/psi-references.html (дата звертання: 26.04.2025).

29. DBSCAN Clustering in ML | Density based clustering. — URL: https://www.geeksforgeeks.org/dbscan-clustering-in-ml-density-based-cluste ring/ (дата звертання: 27.04.2025).

30. haifengl/smile: Statistical Machine Intelligence & Learning Engine. — URL: https://github.com/haifengl/smile (дата звертання: 27.04.2025).

31. Miscellaneous Swing Components, Messages | IntelliJ Platform Plugin SDK. — URL: https://plugins.jetbrains.com/docs/intellij/misc-swing-components.html#messages (дата звертання: 28.04.2025).

32. Notifications | IntelliJ IDEA Documentation. — URL: https://www.jetbrains.com/help/idea/notifications.html (дата звертання: 29.04.2025).

33. Dialogs | IntelliJ Platform Plugin SDK. — URL: https://plugins.jetbrains.com/docs/intellij/dialog-wrapper.html (дата звертання: 02.05.2025).

34. Kotlin UI DSL Version 2 | IntelliJ Platform Plugin SDK. — URL: https://plugins.jetbrains.com/docs/ intellij/kotlin-ui-dsl-version-2.html (дата звертання: 02.05.2025).

35. Combo Box | IntelliJ Platform Plugin SDK. — URL: https://plugins.jetbrains.com/docs/intellij/ combo-box.html (дата звертання: 03.05.2025).

36. Tool Windows | IntelliJ Platform Plugin SDK. — URL: https://plugins.jetbrains.com/docs/intellij/tool-windows.html (дата звертання: 03.05.2025).

37. Batik Swing components. — URL: https://xmlgraphics.apache.org/ batik/using/swing.html (дата звертання: 04.05.2025).

38. Boreskov A., Shikin E. Computer Graphics: from Pixels to Programmable Graphics Hardware. Boca Raton : CRC Press, 2014. 568 p.

39. Execution Contexts, Progress Indicator | IntelliJ Platform Plugin SDK. — URL: https://plugins.jetbrains.com/docs/intellij/coroutine-execution-contexts.html #progress-indicator (дата звертання: 08.05.2025).

40. Яремко З. М. Безпека життєдіяльності: Навч. посіб. Львів., 2005. 301 с.

41. Желібо Є. П. Заверуха Н.М., Зацарний В.В. Безпека життєдіяльності. Навчальний посібник. К.: Каравела, 2004. -328 с.