МАГІСТЕРСЬКА РОБОТА

МР. ІПм - 12.00.00.000 ПЗ

Група ІПм-23-2

Качанюк Василь

2024

Івано-Франківський національний технічний університет нафти і газу

Інститут інформаційних технологій

Кафедра інженерії програмного забезпечення

Качанюк Василь Сергійович

(прізвище, ім'я, по батькові)

УДК 004.942

(індекс)

МАГІСТЕРСЬКА РОБОТА

Методи автоматизації процесів класифікації для виявлення помилок програмного забезпечення

(назва роботи)

Інженерія програмного забезпечення

(назва освітньої програми)

121 - Інженерія програмного забезпечення

(шифр і назва спеціальності)

Качанюк В.С.

(підпис, ініціали та прізвище здобувача освітнього ступеня)

Науковий керівник Вовк Роман Богданович, к.т.н., доцент

(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Допущено до захисту

Завідувач кафедри

доц. Бандура В.В.

(посада) (підпис) (дата) (ініціали та прізвище)

Нормоконтроль

доц. Вовк Р.Б.

(посада) (підпис) (дата) (ініціали та прізвище)

Робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

Івано-Франківськ – 2024

Івано-Франківський національний технічний університет нафти і газу

Інститут інформаційних технологій

Кафедра інженерії програмного забезпечення

Освітній рівень магістр

Спеціальність 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Зав. кафедрою **ІПЗ**
доц. В.В. Бандура
“ 04 ” вересня 2024 р.

ЗАВДАННЯ

НА МАГІСТЕРСЬКУ РОБОТУ СТУДЕНТУ

Качанюку Василю Сергійовичу

(прізвище, ім’я, по-батькові)

1. Тема магістерської роботи “ Методи автоматизації процесів класифікації для виявлення помилок програмного забезпечення ”

керівник проекту (роботи) Вовк Роман Богданович, к.т.н., доцент

затверджені наказом закладу вищої освіти від “ 22 ” листопада 2024 р. № 781/7

2. Строк подання студентом проекту (роботи) 15 грудня 2024 р.

3. Вихідні дані до проекту (роботи) Теоретичні концепції та формальні моделі побудови та функціонування інформаційних та програмних виявлення помилок

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

1. Дослідження та аналіз предметної області виявлення помилок програмного забезпечення
2. Моделі та методи процесів класифікації для виявлення помилок в архітектурі ПЗ
3. Дослідження методів машинного навчання з точки зору використання пошуку помилок в ПЗ
4. Імплементація методів автоматизації процесів класифікації для виявлення помилок ПЗ

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

1. Рольові стереотипи (рис. 1.1)
2. Інтерфейс аналізатора коду CodeScene (рис. 2.1)
3. Об’єктно-орієнтована програма, її спрощене абстрактне синтаксичне дерево (рис. 2.2)
4. Решітка типів в мові метапрограмування RASCAL (рис. 2.3)
5. Структура розв’язувача (рис. 2.4)

6. Консультанти розділів проекту (роботи)

Розділ	Консультант	Підпис, дата
Перевірка на плагіат	доц., к.т.н. Вовк Р.Б.	

7. Дата видачі завдання 04 вересня 2024 р.

Керівник _____
(підпис)

Завдання прийняв до виконання_____
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назви етапів магістерської роботи	Строк виконання етапів роботи	Примітка
1	Підбір і вивчення літератури по темі магістерської роботи	15.09.2024	виконано
2	Аналіз концепцій та алгоритмів предметної області	29.09.2024	виконано
3	Дослідження та аналіз предметної області виявлення помилок програмного забезпечення	15.10.2024	виконано
4	Моделі та методи процесів класифікації для виявлення помилок в архітектурі ПЗ	08.11.2024	виконано
5	Дослідження методів машинного навчання з точки зору використання пошуку помилок в ПЗ	20.11.2024	виконано
6	Імплементація методів автоматизації процесів класифікації для виявлення помилок ПЗ	01.12.2024	виконано
7	Затвердження пояснювальної записки роботи завідувачем кафедри	15.12.2024	виконано

Студент – магістр _____
(підпис)

Керівник роботи _____
(підпис)

АНОТАЦІЯ

Магістерська робота: 81 с., 15 рис., 13 табл., 53 джерела.

Тема: Методи автоматизації процесів класифікації для виявлення помилок програмного забезпечення

Об'єкт дослідження: процеси аналізу та виявлення помилок в архітектурі програмного забезпечення.

Мета роботи: дослідження методів автоматизації класифікації для виявлення помилок у програмному забезпеченні шляхом використання машинного навчання для ідентифікації стереотипів класових ролей та перевірки на порушення архітектурних правил.

Предмет дослідження: моделі та методи автоматизованої класифікації стереотипів класових ролей та виявлення порушень архітектурних правил у програмному забезпеченні на основі машинного навчання.

Результати дослідження:

В роботі виконано розробку автоматизованого підходу до ідентифікації стереотипів класових ролей та виявлення архітектурних помилок у програмному забезпеченні на основі машинного навчання.

Висновок

Виконано розробку методології, що дозволяє автоматизувати процес виявлення помилок у програмному забезпеченні, особливо в частині архітектурних помилок та порушень стереотипів класових ролей. Інструмент може бути застосований для підвищення якості програмних продуктів, зменшення витрат на їх тестування та обслуговування, а також для полегшення роботи розробників під час проведення ревізії та рефакторингу коду.

ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ, АВТОМАТИЗАЦІЯ, ВИЯВЛЕННЯ ПОМИЛОК, СТЕРЕОТИПИ КЛАСОВИХ РОЛЕЙ, АРХІТЕКТУРНІ ПРАВИЛА, МЕТАПРОГРАМУВАННЯ, МАШИННЕ НАВЧАННЯ, КЛАСИФІКАЦІЯ

ABSTRACT

Master Thesis: 81 pp., 15 fig., 13 tab., 53 sources.

Thesis Subject: Methods of automation of classification processes for detecting software errors

Research object: analysis and error processes in software architecture.

The purpose of the work: the study of classification automation methods for detecting errors in software by using machine learning to identify stereotypes of class roles and for violations of architectural rules.

Research subject: models and methods of automated classification of stereotypes of class roles and detection of constructed architectural rules in software based on machine learning.

Research results

The paper developed an automated approach to the identification of class role stereotypes and the detection of architectural errors in software based on machine learning.

Conclusion

A methodology has been developed that allows you to automate the process of detecting errors in architecture software, especially in the area of errors and to break the stereotypes of class roles. The tool can be used to improve the quality of software products, reduce the costs of their testing and maintenance, as well as to facilitate the work of developers during code revision and refactoring.

SOFTWARE, AUTOMATION, ERROR DETECTION, CLASS ROLE STEREOTYPES, ARCHITECTURAL RULES, METAPROGRAMMING, MACHINE LEARNING, CLASSIFICATION

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

IGT - Image Guided Therapy

AST - Abstract Syntax Trees

DSL - Domain-Specific Language

GBDT - Gradient Boosting Decision Trees

MCC - Matthews correlation coefficient.

ML - Machine Learning.

RF - Random Forest.

ST - Structurer.

SP - Service Provider.

SrcRI - Source-code Role Identifier.

SMOTE - Synthetic Minority Over-sampling Technique.

ВСТУП

Актуальність теми.

Сучасні програмні системи стають дедалі складнішими, що підвищує вимоги до їх надійності та якості. Помилки в архітектурі програмного забезпечення можуть призвести до серйозних проблем у функціонуванні програм, що знижує продуктивність систем та підвищує витрати на їх обслуговування. Тому автоматизація процесів виявлення помилок, зокрема ідентифікація стереотипів класових ролей та перевірка на порушення архітектурних правил, є важливим напрямком для забезпечення високої якості програмного забезпечення. Розвиток методів машинного навчання надає нові можливості для створення ефективних інструментів автоматизації виявлення помилок.

З кожним роком складність сучасного програмного забезпечення зростає, що призводить до підвищення вимог щодо його надійності та якості. Помилки в програмному коді можуть не тільки знижувати продуктивність програмних систем, але й викликати критичні збої, що впливають на бізнес-процеси, фінансові втрати та безпеку. Особливо це стосується архітектурних помилок, які можуть залишатися непоміченими під час традиційного тестування, оскільки вони не завжди викликають явні збої, але суттєво впливають на довгострокову стабільність та можливість масштабування систем.

Автоматизація процесів виявлення помилок є важливим напрямком розвитку програмної інженерії, оскільки вона дозволяє значно знизити витрати на тестування та забезпечити виявлення помилок на ранніх етапах розробки. Водночас, застосування методів машинного навчання для автоматизованої класифікації помилок та ідентифікації стереотипів класових ролей відкриває нові можливості для покращення якості коду. Це особливо важливо для великих проектів, де перевірка архітектурних рішень вручну є надзвичайно складною та трудомісткою задачею.

Актуальність дослідження підсилюється тим, що традиційні методи аналізу та виявлення помилок не завжди можуть ефективно вирішувати проблеми, пов'язані з архітектурними недоліками або порушеннями у структурі класових ролей. Розвиток інструментів, таких як RASCAL та ClaiR, у поєднанні з методами машинного навчання дозволяє створювати нові автоматизовані рішення для виявлення цих помилок на основі аналізу вихідного коду. Таким чином, дослідження є актуальним не тільки для підвищення якості та надійності програмних систем, але й для поліпшення процесу розробки, зокрема в частині оптимізації архітектури програмного забезпечення.

Така автоматизація виявлення помилок має особливе значення для індустрії програмного забезпечення, де будь-які покращення виявлення помилок та оптимізації процесів розробки можуть мати суттєвий економічний ефект. Зокрема, зменшення витрат на тестування, скорочення часу на пошук і виправлення помилок, а також підвищення якості кінцевого продукту.

Мета дослідження - дослідження методів автоматизації класифікації для виявлення помилок у програмному забезпеченні шляхом використання машинного навчання для ідентифікації стереотипів класових ролей та перевірки на порушення архітектурних правил.

Об’єкт дослідження - процеси аналізу та виявлення помилок в архітектурі програмного забезпечення.

Предмет дослідження – моделі та методи автоматизованої класифікації стереотипів класових ролей та виявлення порушень архітектурних правил у програмному забезпеченні на основі машинного навчання.

Відповідно до мети роботи було сформовано наступні **задачі:**

- Дослідити існуючі підходи до виявлення помилок в архітектурі програмного забезпечення та ідентифікації стереотипів класових ролей.
- Проаналізувати особливості процесів метапрограмування та використання інструментів для розбору вихідного коду.
- Вивчити методи машинного навчання для класифікації помилок у програмному забезпеченні.
- Розробити та впровадити методологію автоматизованої класифікації стереотипів класових ролей на основі машинного навчання.
- Оцінити точність і зручність використання запропонованого інструменту та виявити можливі проблеми з його точністю.
- Провести оптимізацію класифікатора та дослідити його ефективність для виявлення помилок у програмному забезпеченні.

Методи дослідження.

У процесі дослідження використовувалися методи статичного аналізу коду, метапрограмування, а також машинного навчання, зокрема методи класифікації: випадковий ліс, XGBoost та мультинаївний Байєс. Для обробки та аналізу вихідного коду було застосовано інструменти RASCAL і ClaiR.

Наукова новизна отриманих результатів полягає в розробці автоматизованого підходу до ідентифікації стереотипів класових ролей та виявлення архітектурних помилок у програмному забезпеченні на основі машинного навчання. Запропонована методологія використовує машинне навчання для аналізу програмного коду, що дозволяє автоматизувати процеси виявлення помилок архітектурних правил та підвищити ефективність пошуку помилок у складних програмних системах.

Практичне значення магістерської роботи полягає у розробці інструменту, що дозволяє автоматизувати процес виявлення помилок у програмному забезпеченні, особливо в частині архітектурних помилок та порушень стереотипів класових ролей. Інструмент може бути застосований для підвищення якості програмних продуктів, зменшення витрат на їх тестування та обслуговування, а також для полегшення роботи розробників під час проведення ревізії та рефакторингу коду.

Структура магістерської роботи. Робота складається зі вступу, трьох розділів та висновків. Загальний обсяг роботи становить 81 сторінку, і містить 15 рисунків, 13 таблиць, список використаних джерел із 53 найменувань.

РОЗДІЛ 1. ДОСЛІДЖЕННЯ ТА АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ВИЯВЛЕННЯ ПОМИЛОК ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1. Опис предметної області дослідження

Програмні системи та проекти продовжують зростати у складності та розмірах з моменту появи перших комп'ютерів. Щоб підтримувати організованість та якість програмного забезпечення, перед написанням будь-якого коду створюється архітектура програмного забезпечення. Архітектура програмного забезпечення слугує кресленням, визначаючи високорівневу структуру, компоненти та взаємодії всередині системи. Це фундаментальна організація системи, втілена в її компонентах, їхніх взаєминах один з одним та з навколишнім середовищем, а також принципах, що керують її розробкою та еволюцією.

Архітектура вимагає чіткого розуміння кожного компонента та його функціональності або ролі в загальній картині. Кожен компонент має свою мету та відповідальності, які можуть залежати від інших компонентів або частини всієї системи. Компонент реалізовується як клас з методами, функціями, змінними та іншим, дотримуючись обмежень та шаблонів проектування, описаних в архітектурі. Ці характеристики класів повинні відповідати чітко визначеній архітектурі та є ключовим кроком в об'єктному проектуванні. Однак це не завжди так, і програмне забезпечення може не відповідати архітектурі або архітектура може бути погано розроблена.

Під час безперервного розробки та невеликих оновлень ролі компонентів можуть змінюватися або мати більше відповідальностей, ніж спочатку передбачено в характеристиках класів архітектури. Крім того, ці характеристики можуть не бути визначені всередині архітектури, оскільки на практиці часто трапляється, що характеристики класів залишаються поза етапами проектування або не зазначаються всередині архітектури.

Характеристики цих класів потім необхідно реверс-інжинірити, щоб перевірити їх відповідність та будь-які порушення архітектурних правил.

Не існує чіткої залежності між архітектурою та стереотипами класових ролей та їх впливом на реалізацію. Однак стереотипи класових ролей безпосередньо пов'язані з основними відповідальностями та поведінкою класу. Які, по суті, є основними властивостями класу. Архітектура програмного забезпечення програми або обчислювальної системи також включає елементи програмного забезпечення, зовнішні видимі властивості цих елементів та їх взаємозв'язки. При цьому поведінка кожного елемента є частиною архітектури [5]. Це показує, що повністю виправдано включати стереотип ролі кожного класу або компонента в архітектуру. Це може підтримати реалізацію системи, описаної архітектурою, а також допомогти розробникам програмного забезпечення зрозуміти архітектуру.

Ідентифікація цих характеристик класів є ключовою інформацією для розуміння системами та прийнятих рішень щодо проектування розробниками програмного забезпечення. Вона виконує дві функції: пояснює з першого погляду деякі важливі аспекти очікуваної поведінки компонента та повідомляє про свої наміри щодо проектування іншим [3]. Реверс-інжиніринг характеристик класів вимагає аналізу вихідного коду всередині системи та позначення його характеристиками.

Ця робота має на меті класифікувати характеристики класів всередині системи та перевіряти наявність помилок та порушень архітектурних правил і сприяє автоматизованій класифікації класів, яка може виявляти проблеми та порушення на основі класифікації класу.

Досліджуване програмне забезпечення складається з кількох компонентів, які розділені на каталоги та підкаталоги, які називаються підкомпонентами. Кожен підомпонент складається з тисяч файлів, написаних на мові програмування C++, деякі з яких старі, а деякі нові. Розробники перебувають у процесі рефакторингу старих файлів. З такою кількістю рівнів компонентів стає все важче пояснити та підтримувати код, гарантуючи, що

його відповідальність не змінюється або нові функції не додаються до оригіналу. Керувати функціональністю цих мінливих компонентів і розуміти систему стає все складніше. Відповідальність класу має залишатися чіткою протягом усього процесу розробки і не повинна часто змінюватися або закриватися змінами в класі. Однак наразі відсутня система контролю за виконанням обов'язків кожного класу. Інструмент необхідний для визначення цих обов'язків шляхом аналізу вихідного коду, щоб виявити його відповідальність. Крім того, інструмент повинен повідомляти про будь-які порушення визначених архітектурних правил.

Підводячи підсумок, мета цієї роботи полягає в тому, щоб розробити інструмент, який може автоматично розпізнавати рольові стереотипи кожного класу та виявляти будь-які помилки архітектурних правил. Інформацію, зібрану за допомогою цього інструменту, можна використати для аналізу та вивчення стереотипної архітектури ролі класу в системі.

1.2. Основні питання магістерського дослідження

Метою роботи є можливість оцінити якість класів та їх дизайн виключно за допомогою стереотипів класових ролей. Використання стереотипів класових ролей як інструменту оцінки досі не є поширеним у літературі, оскільки він зазвичай використовується як концептуальна допомога при проектуванні та особливо визначенні масштабу компонентів [16]. Можливо встановити набір правил для архітектури системи, які називаються архітектурними правилами, що визначають, що є хорошим дизайном, та можуть оцінювати систему. Необхідно дослідити моделі та алгоритми і розробити інструмент, який дозволить досягти мети та далі вивчати його результати.

В роботі пропонується одне основне дослідницьке питання та три додаткові дослідницькі питання, які випливають з основного дослідницького питання. Відповідаючи на ці дослідницькі питання, можна було б знайти

потенційні приховані проблеми в архітектурі ПЗ. Дослідницькі питання для цього завдання такі:

1. Як можна визначити якість дизайну за допомогою стереотипів класових ролей ?

Це основне дослідницьке питання, яке також випливає з мети цього завдання. Для оцінки класу необхідний набір архітектурних правил, пов'язаних зі стереотипами ролей, які потім можна порівняти з іншими метриками якості коду.

1.1. Які ознаки є значущими для точної класифікації стереотипів ролей ?

З вихідного коду можна отримати багато ознак. Попередня існуюча література вже досліджувала ці ознаки та їх значущість. Однак це може відрізнятися для інших мов програмування, оскільки вони дотримуються іншої структури. На це питання можна відповісти за результатами класифікатора.

1.2. Як можна візуалізувати стереотипи класових ролей, щоб зрозуміти структуру всередині архітектури?

Візуалізація може допомогти зрозуміти комунікацію між різними стереотипами ролей та потенційні проблеми з архітектурою. Для відповіді необхідно реалізувати інструмент візуалізації, який показує класи та їх директорії, включаючи їх стереотипізацію.

1.3. Які проблеми можна виявити за допомогою стереотипів класових ролей ?

Для відповіді на це питання необхідно знайти помилки та потенційні проблеми всередині вихідного коду. Жодна література не викладала чітко будь-які прямі проблеми, які можна було б знайти через стереотипізацію класів ролей. Це питання тісно пов'язане з результатами класифікатора та перевіркою на наявність будь-яких порушень у визначеному наборі архітектурних правил.

1.3. Особливості поняття стереотипів класових ролей в архітектурі програмного забезпечення

Розробка програмного забезпечення може бути включена в архітектуру, але більше зосереджується на програмному забезпеченні всередині компонентів і самого класу. Процес проектування описує, як вимога повинна бути виконана дизайнерським продуктом, тоді як форми його представлення забезпечують засоби моделювання ідей про дизайн [7]. Дизайн системи показує, як об’єкти взаємодіють, обмінюючись інформацією, функціями або інтерфейсом. Процес проектування часто починається з винаходу об’єктів, призначення відповідальності за них і виконання роботи програми [3].

Відповідальність – це вже термін, який широко використовується в розробці та архітектурі програмного забезпечення. Це одна з перших речей, які дизайнер повинен визначити, як об’єкт виконує свої обов’язки [33] і якими вони є. Обов’язки компонента всередині архітектури зазвичай являють собою список функціональних можливостей і завдань, які компонент повинен виконувати щодо вимог загальної системи. Ці функціональні можливості та функції відомі загальній системі для використання іншими компонентами, спільної роботи чи успадкування. Якщо систему знову розкласти на логічні частини, має бути можливим ідентифікувати об’єкти або ролі та класи проектування, які реалізують конкретні ролі [33]. Тут можна розділити обов’язки та ролі системи на класи та надати кожному з них мету чи роль для загальної системи чи підкомпонента.

У цій роботі основна увага буде зосереджена на ролях цих класів і з’ясовано, чи можливо їх реверсивне проектування та дотримання цієї ідеї щодо обов’язків і ролей об’єктів. Весь контекст щодо ролей і обов’язків об’єкта посилатиметься на рольовий стереотип об’єкта.

Рольовий стереотип класу був представлений у [34], щоб ідентифікувати та описати ідеальні типи обов'язків класів. Деякі з цих стереотипів – контролер, координатор або постачальник послуг. Ці

стереотипи є загальними типами та обов'язками класів і мають бути визначені на етапі проектування системи. Автор дотримується підходу проектування, орієнтованого на відповідальність, щоб створити основу для міркувань про об’єкти. Цей підхід означає, що чітко визначений об’єкт підтримує чітко визначену роль.

Було класифіковано ролі та обов’язки на шість рольових стереотипів, по суті, над спрощенням фокусу об’єкта. Об’єкт можна помістити в один із цих шести стереотипів ролі класу :

• Власник інформації: як випливає з назви, ці класи містять інформацію. Носій інформації може бути простим власником даних, який зберігає інформацію про об’єкт, або більш складним об’єктом, який містить інформацію з певною логікою, застосованою до неї. Власник інформації інкапсулює дані та надає такі методи, як геттери та сетери для маніпулювання цими даними.

• Постачальник послуг: ці типи класів виконують роботу та пропонують послуги іншим об’єктам. Цей клас зазвичай має методи з багаторазовою функціональністю, які можуть використовувати різні частини системи.

• Структуратор: цей об’єкт відповідає за підтримку зв’язків між іншими об’єктами та групами об’єктів. Ці класи можуть мати структури даних, як-от бази даних, які зберігають і підтримують зв’язки між сутностями, або абстрактні структури, як-от архітектура програмної системи.

• Контролер: цей об’єкт головним чином відповідає за прийняття рішень і керування потоком програми. Однією з функцій контролера є прийняття вхідних даних і перетворення їх на команди для системи.

• Координатор: ці об’єкти використовуються для розподілу завдань і керування іншими об’єктами. Ці об’єкти не приймають багато незалежних рішень, але допомагають керувати та контролювати, як частини системи взаємодіють одна з одною на основі попередньо визначених правил або системних процедур.

• Інтерфейс: об’єкти, які відповідають за зв’язок між частинами системи. Інтерфейси несуть відповідальність за те, щоб дані були у правильному форматі та що будь-які перетворення, необхідні для зв’язку, виконуються. Це може бути інтерфейс користувача, який взаємодіє з користувачами та передає внутрішнім підсистемам будь-які дії, що виконуються в GUI.

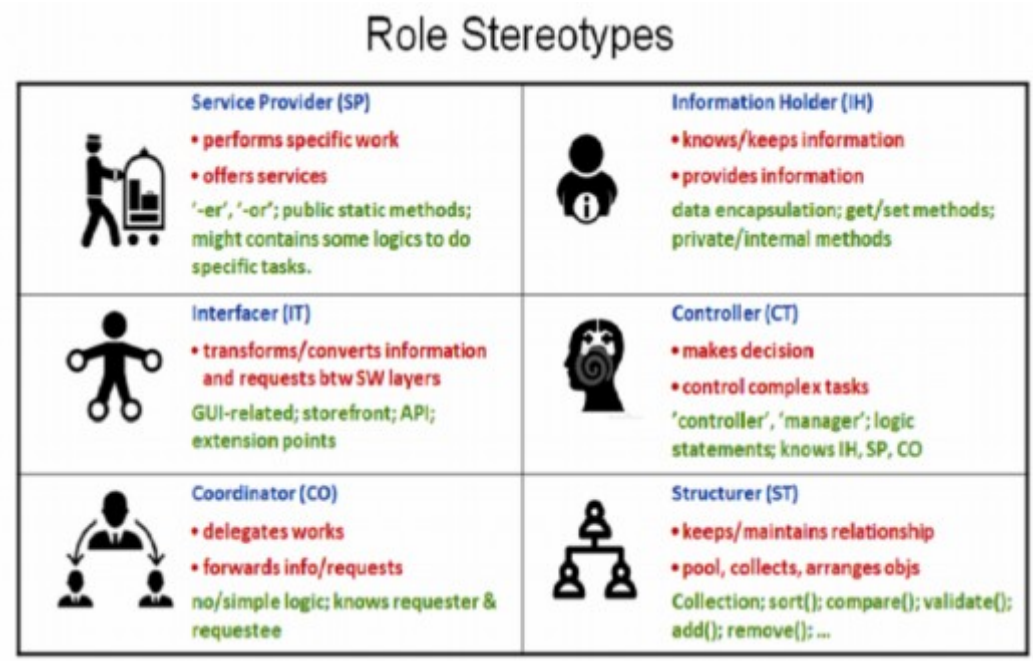

Рис. 1.1. Рольові стереотипи

Виявлення цих рольових стереотипів може допомогти в різних завданнях розробки та обслуговування програмного забезпечення, таких як розуміння програми, узагальнення програми та забезпечення якості [4, 14, 15, 30]. Деякі дослідження також показують, що рольові стереотипи допомагають діаграмам класів UML покращити зрозумілість діаграм [4, 14]. Проте на практиці стереотипи про роль класу рідко чітко документуються.

Визначити рольовий стереотип класу можна шляхом побудови списку обов’язків і завдань класу та порівняння їх із шістьма рольовими стереотипами. У вихідному коді достатньо великого класу також достатньо інформації, щоб визначити стереотип ролі, перевіривши його функції та

властивості. Однак точні функції та властивості можуть змінюватися залежно від мови програмування та умов найменування. Прикладом Structurer у Java може бути те, що він може розширювати структуру колекції Java або еквівалент, має імена методів, які маніпулюють колекціями, наприклад sort(), compare(), validate(), remove(), updates(), add(). Клас може містити тип «визначений користувачем об'єкт» як атрибути [15]. Аналогічно, для Structurer у C++ схема іменування методів дещо відрізняється, або вона не розширює жодну структуру колекції, але може використовувати змінні з типами структур. Не всі класи легко стереотипні. Існують можливості, коли клас надає дві або більше ролей. Приклад, використаний автором це те, як постачальник послуг часто може зберігати інформацію, необхідну для надання послуг. Це робить клас постачальником послуг і власником інформації. Однак уся відповідальність класу полягає в наданні послуг, оскільки інформація використовується виключно для підтримки цього. Можуть бути випадки, коли клас може мати кілька рольових стереотипів. Це випадок, коли відповідний клас має багато різних типів клієнтів, яким потрібні послуги класу або інформація, яку він містить.

Ще одна проблема, з якою можна зіткнутися, створюючи стереотип класу, полягає в тому, як його інтерпретують. Стереотип класової ролі не завжди чіткий і може змінюватися залежно від того, на чому рецензент робить акцент. Клас із функціями координатора та контролера може призвести до плутанини. Цей сценарій може статися, коли клас приймає рішення за допомогою операторів if, що призводить до вихідного виклику іншого класу. У цьому сценарії один рецензент може сказати, що в центрі уваги — прийняття рішень, а інший — що в центрі уваги вихідні дзвінки.

Висновки до розділу

Перший розділ присвячено дослідженню та аналізу предметної області виявлення помилок програмного забезпечення, дозволяє сформувати чітке

уявлення про основні аспекти, які є критичними для розуміння контексту даної проблематики.

У процесі дослідження предметної області виявлено, що програмне забезпечення залишається надзвичайно складним та багаторівневим продуктом, розробка і тестування якого вимагають не тільки високого рівня технічної експертизи, але й використання систематизованих підходів для виявлення та усунення помилок. Встановлено, що помилки у програмному забезпеченні можуть виникати на різних етапах життєвого циклу розробки, що підкреслює важливість побудови ефективної системи контролю якості.

Ключовими питаннями магістерського дослідження стали визначення типології помилок, що можуть виникати у програмних системах, а також аналіз методів та інструментів для їх виявлення. Окрема увага приділена стереотипам класових ролей в архітектурі програмного забезпечення, які значною мірою впливають на ефективність процесу виявлення помилок та забезпечують структурований підхід до організації архітектурних елементів програмної системи.

Таким чином, перший розділ закладає теоретичну базу для подальшого дослідження, систематизуючи основні концепції, пов'язані з виявленням помилок у програмному забезпеченні, та окреслюючи напрямки для розробки ефективних методів їх виявлення і аналізу.

РОЗДІЛ 2. МОДЕЛІ ТА МЕТОДИ ПРОЦЕСІВ КЛАСИФІКАЦІЇ ДЛЯ ВИЯВЛЕННЯ ПОМИЛОК В АРХІТЕКТУРІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1. Особливості процесів аналізу коду та метапрограмування

Аналіз коду — це процес вивчення компонентів класу та їх порівняння з набором правил. Цей процес включає аналіз вихідного коду, каталогу, форматування та імен. Часто використовується інструмент аналізу коду під назвою CodeScene, щоб оцінити якість вихідного коду. Інструмент призначає оцінку працездатності коду кожному класу в діапазоні від 1 до 10 на основі набору правил. Крім того, CodeScene надає короткий опис оцінювання. Оцінка занять використовується для порівняння їх з результатами даної роботи.

CodeScene — це інструмент для аналізу коду, який застосовує методи штучного інтелекту та пов'язані з поведінкою дані для аналізу якості програмного коду, продуктивності команди, та можливих ризиків у програмному забезпеченні. Його основна мета—допомогти командам розробників зосередитися на проблемних ділянках коду та покращити процес розробки.

Основні можливості CodeScene:

- Аналіз технічного боргу. CodeScene автоматично виявляє області коду з високим рівнем складності та технічного боргу. Він може визначити модулі, які потребують покращення, або фрагменти коду, що є складними для розуміння та супроводу.

- Аналіз еволюції коду. Інструмент відстежує зміни в коді з часом, дозволяючи побачити, як розвивається проєкт. Це корисно для розуміння частоти змін у певних файлах та виявлення "гарячих точок", тобто файлів, які часто змінюються.

- Інтеграція з репозиторіями. CodeScene інтегрується з популярними системами контролю версій, такими як GitHub, GitLab, Bitbucket, що дозволяє автоматично аналізувати код з репозиторіїв та створювати звіти без необхідності ручного втручання.

- Аналіз продуктивності команд. Інструмент аналізує активність команди розробників, допомагаючи виявити вузькі місця у процесі розробки, такі як перевантаженість розробників, нерівномірний розподіл роботи, або проблеми з комунікацією в команді.

- Візуалізація залежностей та структури коду. CodeScene створює інтерактивні діаграми, які показують взаємозв'язки між різними частинами коду. Це допомагає зрозуміти структуру проєкту та вплив змін у коді на інші компоненти.

- Прогнозування ризиків у коді. Інструмент визначає потенційно ризиковані зміни в коді та модулі, які можуть бути джерелом помилок у майбутньому. Це дозволяє завчасно виправляти проблеми та знижувати ризик виникнення дефектів у продуктивному коді.

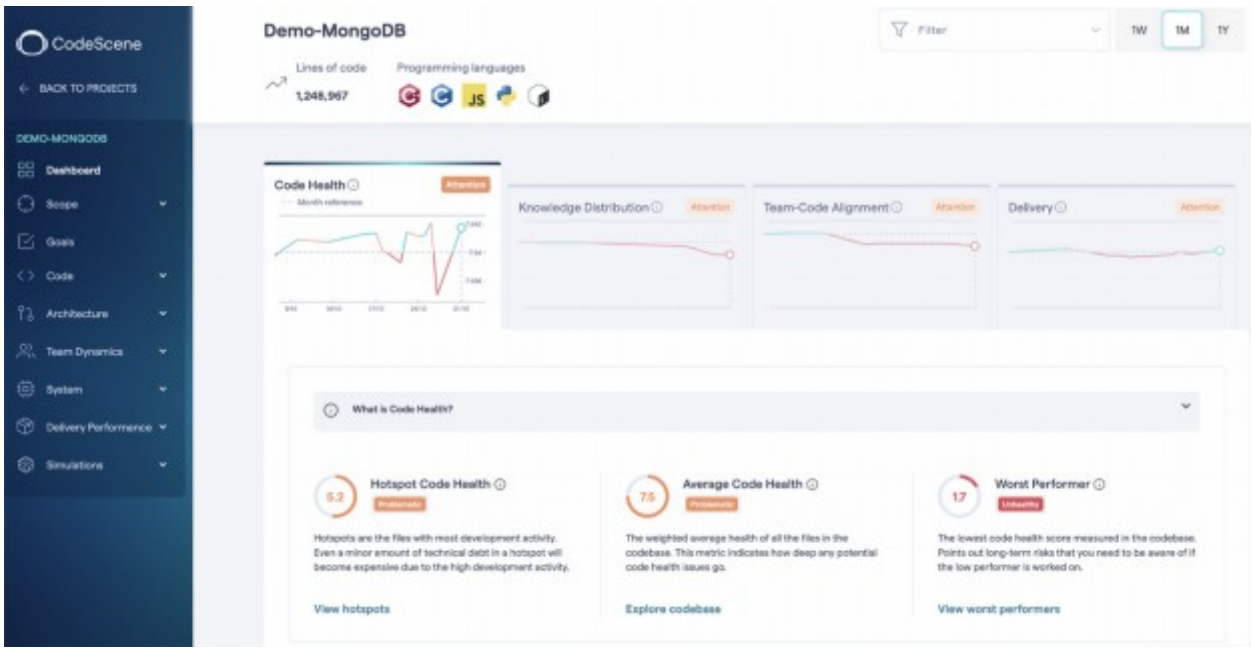

Рис. 2.1. Інтерфейс аналізатора коду CodeScene

CodeScene широко використовується в Agile та DevOps командах, які прагнуть вдосконалити свою практику розробки та підтримувати високу якість коду. Завдяки своєму зосередженню на поведінкових аспектах та еволюції коду, він допомагає краще зрозуміти не тільки технічний стан проєкту, але й те, як працює команда, що дозволяє робити процеси більш ефективними та мінімізувати ризики.

Метапрограмування — це процес визначення типових вихідних шаблонів програмного забезпечення, з яких класи компонентів програмного забезпечення або їх частини можуть бути автоматично створені для створення нових компонентів програмного забезпечення [8]. По суті, це дозволяє розробникам розглядати програмне забезпечення як дані. Це дозволяє їм створювати новий код, абстрактні синтаксичні дерева (AST) і створювати предметно-орієнтовані мови (DSL). Хоча концепція метапрограмування існує вже кілька десятиліть, діяльність фокусується на метапрограмування, таке як створення AST, DSL або інших застосувань, стрімко зростає протягом останніх кількох років, причому більшість мов пропонують певну підтримку метапрограмування, а кількість розробленого метакоду зростає експоненціально [22].

Одним із методів вилучення функцій є перетворення вихідного коду в абстрактне синтаксичне дерево (AST) або подібні моделі. AST — це дерево, яке містить дані вхідного вихідного коду. Однак він містить лише необхідну інформацію та формується шляхом видалення непотрібних форм із дерева синтаксичного аналізу в цілому [19]. AST є оптимальним методом вилучення лише релевантних і необхідних функцій із вхідного вихідного коду, який використовується в цій дипломній роботі. Ці дерева та інші типи моделей можна створити за допомогою метапрограмування.

Abstract Syntax Tree (AST) — це структуроване дерево, що представляє синтаксис програмного коду в абстрактній формі. AST відображає структуру коду таким чином, що кожен вузол дерева відповідає певному синтаксичному елементу програми (наприклад, операції, вирази, змінні,

блоки умов тощо). AST широко використовується в метапрограмуванні для аналізу та трансформації коду.

Особливості Abstract Syntax Tree:

- Абстрактність. На відміну від конкретного синтаксичного дерева (Concrete Syntax Tree, CST), яке відображає кожен символ і синтаксичну деталь, AST є більш абстрактним. Він опускає такі елементи, як дужки або коми, зосереджуючись на логічній структурі коду. Це робить його більш компактним і зручним для аналізу.

- Структура. AST складається з вузлів, де кожен вузол представляє певний синтаксичний елемент. Наприклад:

- Вузол може представляти оператор (наприклад, if або while).

- Вузли можуть містити підвузли, які представляють аргументи функції, умови, оператори та інше.

- Ієрархія. Дерево AST має ієрархічну структуру, де верхні вузли відповідають більш загальним конструкціям (наприклад, функції або класу), а нижчі вузли — окремим виразам або операціям всередині цих конструкцій.

AST є основним інструментом у метапрограмуванні, оскільки він дозволяє розробникам:

- Аналізувати код. AST можна використовувати для аналізу коду, виявлення синтаксичних конструкцій або перевірки наявності певних патернів у коді. Це корисно для інструментів статичного аналізу, таких як лінтери та аналізатори коду.

- Трансформація та рефакторинг коду. Розробники можуть використовувати AST для автоматичного рефакторингу коду, змінюючи певні вузли дерева. Наприклад, можна замінити всі виклики однієї функції на іншу або модифікувати структуру виразів.

Це також використовується для генерації коду — наприклад, з AST можна створити новий код на основі існуючих шаблонів.

- Компілери та інтерпретатори. AST є ключовим компонентом в роботі компіляторів та інтерпретаторів мов програмування. Після того, як код

синтаксично розбирається, він перетворюється на AST, що потім використовується для генерації машинного коду або виконання інтерпретатором.

- Створення DSL (Domain-Specific Languages). AST є основою для побудови мов, специфічних до певної предметної області (DSL). Розробники можуть створювати інструменти для перетворення коду цієї мови в AST та виконувати подальшу обробку або трансляцію в інші мови.

Спільність усіх підходів до перетворення на основі логіки полягає в представленні програми набором логічних пунктів. Можливе представлення речення для невеликої програми на Java показано на рисунку 2.2.

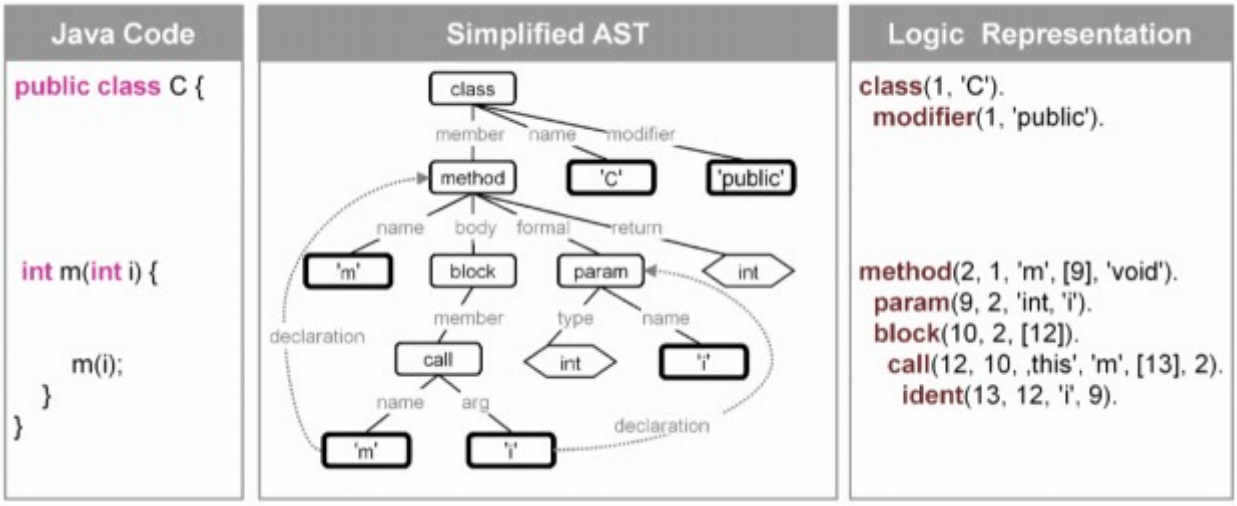

Рис. 2.2. Об’єктно-орієнтована програма, її спрощене абстрактне синтаксичне дерево (AST) і представлення AST у вигляді набору речень

Кожне речення представляє вузол абстрактного синтаксичного дерева програми (AST). Перший аргумент пропозиції є унікальним ідентифікатором для вузла. Інші аргументи — це вбудовані кінцеві значення або ідентифікатори інших вузлів. Кожен «чужий» ідентифікатор представляє посилання на вузол із цим ідентифікатором. Наприклад, другий аргумент кожного вузла не верхнього рівня є посиланням на його охоплюючий вузол.

Метапрограмування логіки являє собою потужний парадигму програмування, яка передбачає використання логічних мов програмування

для маніпулювання та аналізу власних програмних конструкцій. Цей підхід дозволяє розглядати програми як дані, що підлягають обробці іншими програмами, написаними на тій самій мові.

2.2. Дослідження мови метапрограмування та інструментів розбору вихідного коду

RASCAL є мовою метапрограмування, яка забезпечує високорівневу інтеграцію аналізу та маніпуляції вихідного коду на концептуальному, синтаксичному, семантичному та технічному рівнях [20]. Поряд із цим, вона надає ті ж самі функціональні можливості, що й інші мови програмування високого рівня, такі як маніпуляція структурами даних. RASCAL також підтримує функціональності для пошуку шаблонів, компоновки, перемикання та обходу, що спрощує процес програмування [17, 20]. Rascal базується на статичній типізації, це означає, що перед виконанням програми виявляється якомога більше помилок і невідповідностей. Типи впорядковані в так звану решітку типів, показану на рисунку 2.3.

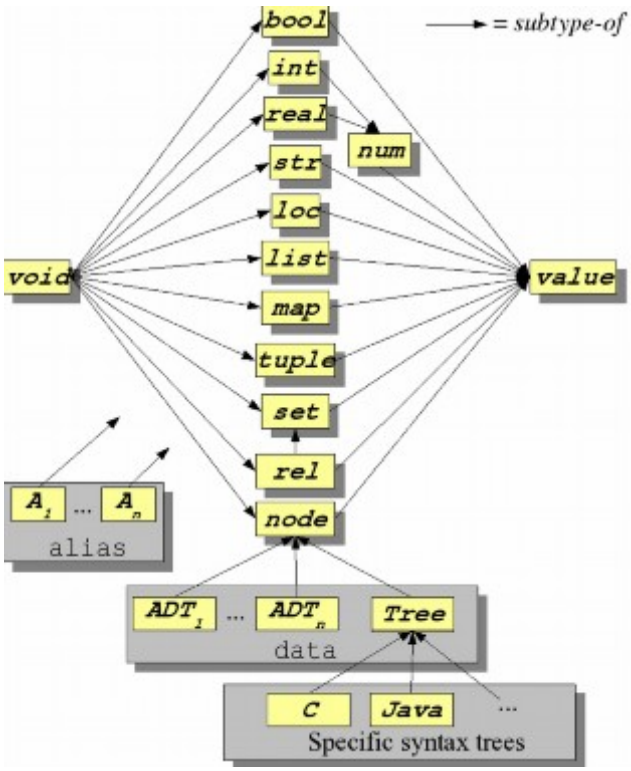

Рис. 2.3. Решітка типів в мові метапрограмування RASCAL

RASCAL дає змогу обхідно відвідувати кожен вузол у абстрактному синтаксичному дереві (AST) та шукати відповідні шаблони через використання кількох варіантів перемикання. Коли шаблон збігається з конкретним випадком, можна відвідати відповідний вузол для отримання додаткової інформації або видобування характеристик, таких як розташування у вихідному коді, типи та інші атрибути. Це дозволяє отримати повне уявлення про синтаксис вихідного коду завдяки чітко визначеній структурі AST та методам, що обходять релевантні вузли. Однак стандартна бібліотека RASCAL не підтримує код на мовах C та C++, тому для ефективного використання RASCAL потрібна додаткова бібліотека.

Розв’язувач намагається розв’язати обмеження в TModel; невирішені обмеження створюють повідомлення про помилки. Метою Solver є вирішення обмежень, які були зібрані Collector, і створення TModel. Функції, які надає Solver показано на рисунку 2.4.

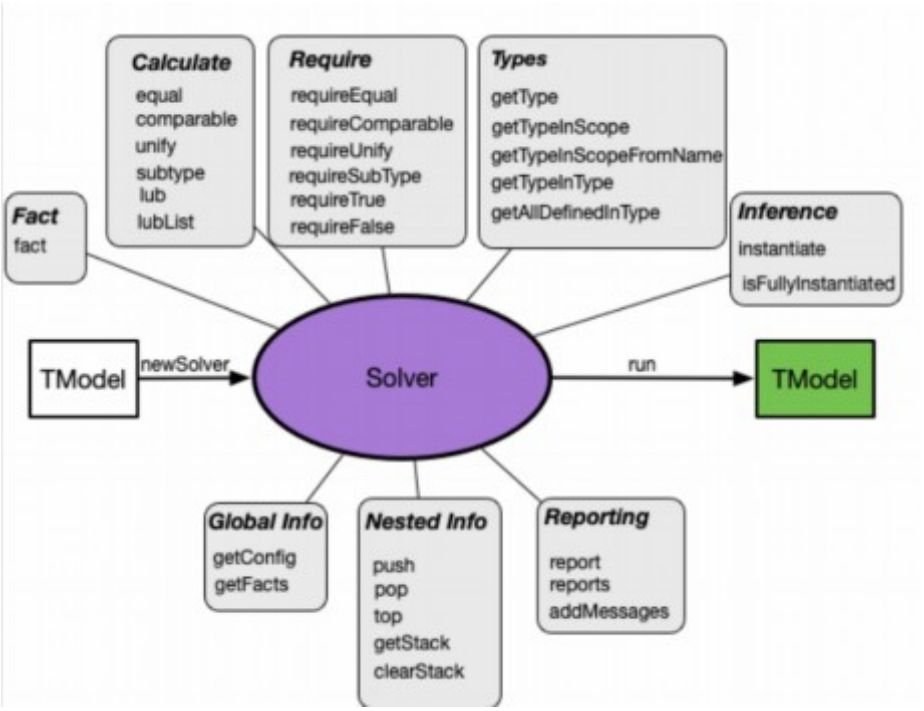

Рис. 2.4. Структура розв’язувача

CLAIR — це відкрите розширення для RASCAL, яке реалізує синтаксичний аналіз вихідного коду мов C та C++ [2, 3]. Це розширення

підтримує перетворення вихідного коду C та C++ у абстрактні синтаксичні дерева (AST) та M3-моделі, що раніше було неможливим, дозволяючи фахівцям з програмування аналізувати, трансформувати або видобувати характеристики з коду на мовах C та C++.

ClaiR надає відображення відкритого інтерфейсу Eclipse CDT C і C++ на модель Rascal M3 для подальшої обробки. Він може працювати з і без роботи всередині IDE Eclipse.

AST у CLAIR зберігає деревоподібне представлення коду у текстовому форматі, використовуючи дужки для позначення різних рівнів дерева. У RASCAL AST зберігається як тип Declaration в CLAIR, який містить декларації, що здійснюються у вихідному коді, такі як визначення функцій, одиниці трансляції або параметри. Всередині кожної декларації можуть міститися додаткові типи, такі як Declarators, Expressions та інші. Для навігації по AST зазвичай використовується метод visit() з визначенням випадків для захоплення потрібних характеристик. Повний перелік характеристик та шаблонів, які можна відвідати, міститься у документації CLAIR.

Іншою функціональністю, яку надає CLAIR, є вилучення моделі M3 з AST. Модель M3 — це тип даних RASCAL, який складається з набору бінарних відносин, отриманих з AST [6, 17]. Модель M3 складається з кількох полів, кожне зі своєю структурою даних зв’язку, яка фіксує розташування та назву функції з вихідного коду. На відміну від AST, модель M3 не вимагає відвідування кожного поля. Щоб отримати список розташувань і повернутих імен, потрібна лише назва поля. Потім можна використовувати відповідний фільтр для відфільтрування похідних функцій або специфічних функцій, таких як змінні, методи або параметри. Однак моделі M3 не зберігають порядок функцій, які були присутні в AST, через що моделі M3 не можуть отримати функції з певних частин коду.

2.3. Дослідження методів машинного навчання з точки зору використання пошуку помилок в програмному забезпеченні

Алгоритм машинного навчання, по суті, є типом алгоритму або набором кількох алгоритмів, які вчаться від себе та свого оточення, запускаючи його ітеративно зі знаннями з попередніх ітерацій [12]. Існує багато типів алгоритмів машинного навчання, які мають досвід у певній галузі, як-от контрольоване навчання, у якому алгоритм відображає набір вхідних даних на вихід, або навчання з підкріпленням, у якому алгоритм встановлює політику для наступної ітерації та спостерігає за результати, спостерігаючи за тим, який вплив мала дія на навколишнє середовище. в у розділі 1 була представлена проблема класифікації. Алгоритм навчання під наглядом може розв’язати проблему класифікації, коли алгоритму можна надати набір ознак і наказати зіставити їх із набором результатів, у даному випадку, шістьма рольовими стереотипами.

Машинне навчання дійсно має певні проблеми та проблеми, які потребують постійного моніторингу та мінімізації ризиків їх виникнення. Поширеною проблемою, яка може виникнути в більшості, якщо не в усіх, алгоритмах навчання є перевиконання та недостатня адаптація моделі. До кожного алгоритму можна застосувати методи, щоб зменшити ймовірність цього, налаштувавши такі гіперпараметри, як кількість ітерацій, використовуючи набір даних перевірки або налаштувавши набір даних навчання.

Для визначення класифікації рольового стереотипу можна використовувати кілька алгоритмів навчання під наглядом. Python є хорошою мовою програмування, яка надає такі алгоритми через загальнодоступні бібліотеки та розширення. Scikit-learn — це бібліотека машинного навчання з відкритим кодом, написана мовою Python [21] і пропонує різноманітний список алгоритмів машинного навчання. Бібліотека легко інтегрує кілька

бібліотек інших алгоритмів машинного навчання, а також підтримує кілька проблемних просторів.

2.3.1. Випадковий ліс

Випадкові ліси поєднують кілька дерев рішень, які розділяють вхідні дані на менші підмножини на основі рішень щодо функцій. Потім листи дерев рішень класифікують вхідні дані, які вони спочатку отримали. Потім випадковий ліс повертає класифікацію на основі того, яка класифікація траплялася найчастіше з усіх дерев рішень.

А Випадковий ліс по суті поєднує вихід багатьох дерев рішень в один вихід, який є класифікацією. Алгоритм можна налаштувати для зміни кількості дерев, які генеруються. Збільшення може призвести до кращої точності, але також до ризику переобладнання. Дерева також можна конфігурувати і дозволяють змінювати їх максимальну глибину або максимальну кількість розділень даних.

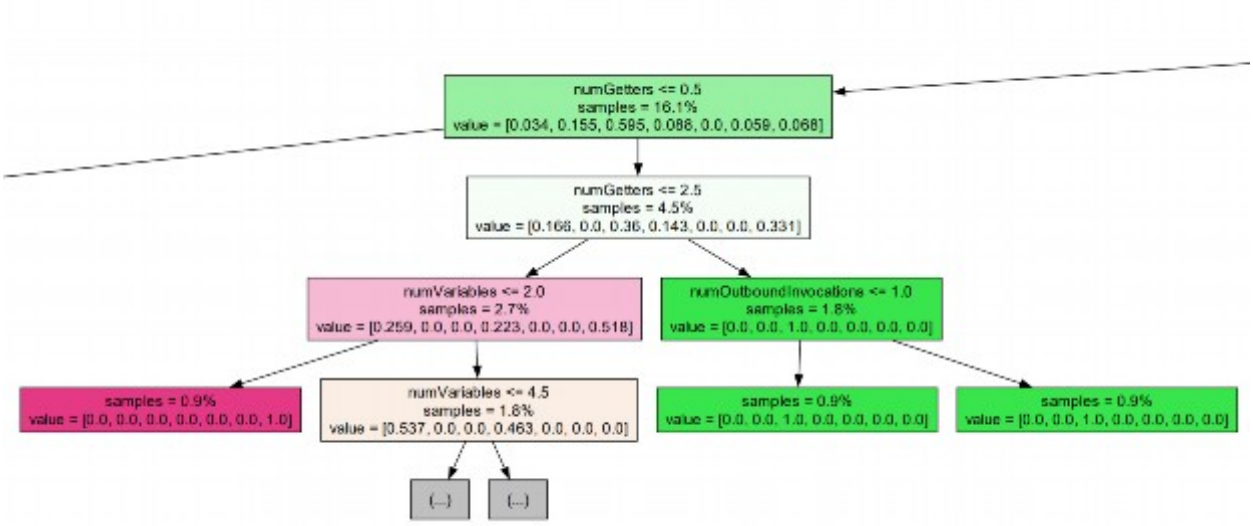

Рис. 2.5. Частина дерева рішень

На рисунку 2.5 показано приклад одного з багатьох дерев рішень, які можуть бути всередині випадкового лісу. Масив чисел є ймовірністю для кожної класифікації, і рішення мають істинний і хибний край, який йде вниз до нового вузла або залишає. На малюнку показано два додаткові листи з

крапками, які вказують на те, що дерево рішень можна згорнути ще більше, щоб показати більше вузлів і листків у дереві.

2.3.2. Модель XGBoost

XGBoost є скороченням від eXtreme Gradient Boosting. Це реалізація фреймворку, що посилює градієнт, зробленого Фрідманом. Подібно до випадкових лісів, XGBoost створив декілька дерев рішень із підвищенням градієнта (GBDT). Вони відрізняються способом побудови дерев рішень. Випадковий ліс бере середнє значення дерев рішень, тоді як XGBoost посилює кілька моделей дерева рішень. Ідея «підсилення» або вдосконалення однієї слабкої моделі шляхом поєднання її з декількома іншими слабкими моделями для створення спільної сильної моделі [28]. Кожна ітерація GBDT використовує залишки помилок попередньої моделі, щоб відповідати поточній моделі. Остаточний прогноз є зваженою сумою всіх прогнозів дерева [28]. Цей метод побудови дерев мінімізує недообладнання та зміщення від класифікатора.

XGBoost (eXtreme Gradient Boosting) - це потужна бібліотека з відкритим кодом, яка використовується в машинному навчанні для вирішення широкого спектра задач, таких як регресія, класифікація та ранжування. Вона базується на методі градієнтного підсилення, який є одним з найефективніших алгоритмів машинного навчання.

XGBoost створює модель шляхом послідовного додавання простих моделей (часто це дерева рішень), причому кожна наступна модель намагається виправити помилки попередніх. Цей процес називається градієнтним підсиленням, оскільки кожна нова модель "підсилює" загальну модель, рухаючись у напрямку градієнта помилки.

Ключові особливості XGBoost:

- Швидкість: XGBoost відомий своєю високою швидкістю навчання завдяки оптимізаціям, паралельним обчисленням та ефективному використанню пам'яті.

- Точність: Моделі, створені за допомогою XGBoost, часто демонструють високу точність на різних типах даних.

- Гнучкість: XGBoost підтримує різні типи цілей (регресія, класифікація, ранжування) та дозволяє налаштовувати гіперпараметри для досягнення оптимальних результатів.

- Регуляризація: Вбудовані механізми регуляризації допомагають уникнути перенавчання моделі.

- Розподілені обчислення: XGBoost може працювати на великих даних за допомогою розподілених обчислень.

Базова структура моделі дерева екстремального градієнтного підсилення показана на рисунку 2.6.

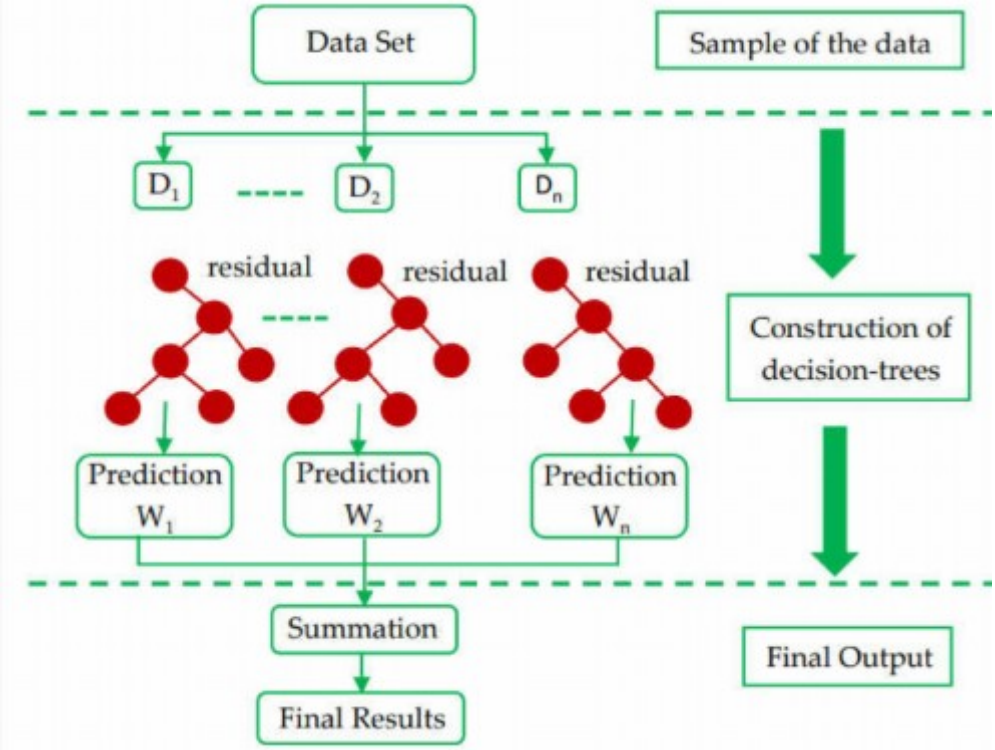

Рис. 2.6. Структура алгоритму XGBoost

Переваги XGBoost перед іншими алгоритмами:

- Висока ефективність: XGBoost часто перевершує інші алгоритми за точністю та швидкістю.

- Гнучкість: Може працювати з різними типами даних та задач.

- Легкість використання: Наявність бібліотек для популярних мов програмування (Python, R, Java тощо) спрощує використання XGBoost.

2.3.3. Метод мультинаївного Байєса

Існує багато різних версій наївних класифікаторів Байєса. Як правило, усі вони припускають, що функції є незалежними, що дозволяє використовувати масштабований та ефективний метод навчання. Багаточленний наївний метод Байєса широко використовується для віднесення документів до класів на основі статистичного аналізу їх вмісту [29]. Multi Naive Bayes є розширенням класичного алгоритму Naive Bayes, який використовується в машинному навчанні для задач класифікації. Класичний Naive Bayes припускає, що ознаки (атрибути) даних є умовно незалежними один від одного, якщо відома їхня класність. Це означає, що ймовірність того, що об'єкт належить до певного класу, залежить тільки від ймовірностей його ознак, а не від їхніх взаємозв'язків.

Multi Naive Bayes розширює цю припущення, дозволяючи враховувати залежності між ознаками. Він робить це шляхом розбиття набору ознак на підмножини, які вважаються незалежними один від одного. Кожна підмножина ознак використовується для побудови окремого Naive Bayes класифікатора, а потім результати цих класифікаторів об'єднуються для отримання кінцевого прогнозу.

Класифікатор визначає ймовірність належності кожної ознаки до класу, а потім призначає їм класифікацію як результат. Він використовує свої попередні знання, щоб визначити ймовірність майбутніх прогонів.

Висновки до розділу

Другий розділ присвячено моделям та методам процесів класифікації для виявлення помилок в архітектурі програмного забезпечення, і

узагальнює результати дослідження сучасних підходів до аналізу коду та використання машинного навчання у виявленні помилок.

У результаті аналізу процесів аналізу коду та метапрограмування виявлено, що ці технології дозволяють не лише здійснювати статичний аналіз вихідного коду, але й динамічно адаптувати його структуру для спрощення процесу виявлення помилок. Метапрограмування виступає важливим інструментом у автоматизації перевірки коду, оскільки дозволяє створювати абстракції та шаблони, що підвищують ефективність роботи з великими обсягами програмного коду.

Дослідження мов метапрограмування та інструментів розбору вихідного коду продемонструвало важливість використання сучасних інструментів для автоматизованого аналізу структури програмного забезпечення. Використання таких інструментів дозволяє зменшити трудовитрати та ризик людських помилок під час перевірки, а також прискорює виявлення потенційних дефектів у коді.

Методи машинного навчання, розглянуті в роботі, включаючи випадковий ліс, XGBoost та метод мультинаївного Байєса, підтвердили свою ефективність у задачах класифікації помилок у програмному забезпеченні. Випадковий ліс забезпечує високу точність та стійкість до переобучення, модель XGBoost вирізняється високою продуктивністю та швидкістю обробки даних, а метод мультинаївного Байєса демонструє простоту впровадження та хороші результати у випадках з невеликою кількістю навчальних даних.

Таким чином, другий розділ закладає практичну основу для застосування сучасних методів аналізу коду та класифікації помилок у програмному забезпеченні, показуючи перспективність використання метапрограмування та машинного навчання для підвищення ефективності процесу виявлення помилок

РОЗДІЛ 3. ІМПЛЕМЕНТАЦІЯ МЕТОДІВ ТА МОДЕЛЕЙ АВТОМАТИЗАЦІЇ ПРОЦЕСІВ КЛАСИФІКАЦІЇ ДЛЯ ВИЯВЛЕННЯ ПОМИЛОК ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1. Особливості стереотипізації програмного забезпечення

Дослідження, спрямовані на ідентифікацію та класифікацію стереотипів, проводилися раніше. Проте, що є несподіваним, у науковій літературі мало уваги приділено стереотипам ролей класів. Різні автори опублікували три статті, у яких спочатку ідентифікуються стереотипи методів [9], що можуть бути використані для побудови сигнатури програмної системи [10], а потім ця сигнатура використовується для визначення стереотипу класу [11]. Інший підхід до стереотипізації класів полягає у застосуванні машинного навчання, що було реалізовано для Java.

3.1.1 Ідентифікація стереотипів ролей методів

Робота [9] пропонує класифікацію стереотипів методів у мові C++, яка використовує іншу таксономію порівняно з стереотипами класів. Ідентифікація таких стереотипів методів має свої переваги, як зазначають автори. Вона сприяє визначенню стереотипів класів і забезпечує більш точний розрахунок метрик. Такі метрики оцінюють, наскільки об’єктно-орієнтованим є клас або система, а також прогнозують зміни, які можуть відбутися при зміні стереотипу методу. Це допомагає зрозуміти контекст класу та характер його взаємодії з іншими класами [10].

Для цього спершу надається визначення та пояснення кожного стереотипу методу, після чого представлено набір правил для їх ідентифікації. Розроблено інструмент, що перевіряє ці правила та, на їх основі, визначає стереотип методу. Також інструмент автоматично документує кожен метод у вихідному коді з вказівкою на його рольовий стереотип. В результаті роботи близько 10% методів залишилися

неідентифікованими, а деякі були класифіковані некоректно через низьку якість стилю програмування [9].

3.1.2. Стереотипізація за допомогою сигнатур класів

Наступні роботи [10, 11] є продовженням ідентифікації стереотипів методів з метою класифікації стереотипів ролей класів. Розподіл раніше класифікованих стереотипів методів формує сигнатуру системи. Сигнатура описує поширеність статичних структур, таких як співпраця між компонентами та зміни стану [10]. Часто досліджують методи за допомогою свого інструмента в проєктах з відкритим кодом на C++/C та аналізують частоту і розподіл стереотипів методів у межах системи.

Дана робота зосереджена на синтаксичних характеристиках вихідного коду C++ для визначення стереотипу ролі класу. Хоча це здається схожим на дослідження, запропоноване в [9], але використовує зовсім іншу таксономію класових стереотипів, яка не збігається з шістьма визначеними класовими рольовими стереотипами. Даний підхід використовує частоту та розподіл методичних стереотипів у класі для визначення стереотипу ролі класу. Стаття називає розподіл класів сигнатурами класів, які використовуються як вхідні дані для автоматичної класифікації стереотипу ролі класу за допомогою визначеної схеми. Стаття продовжується прикладом системи, яка візуалізує розподіл стереотипів методів для кожного стереотипу ролі класу, а потім встановлює правила для кожного стереотипу, які можна використовувати для автоматичної класифікації стереотипу до класу.

Документ оцінив їхній підхід, опитавши думки експертів щодо класового стереотипу та порівнявши їх із класифікацією інструменту. У випадках, коли інструмент не погоджувався з експертами, він вказував на певну форму проблеми з підходом або таксономією рольових стереотипів. З емпіричного дослідження видно, що приблизно 94%-99% класів підходять під один класовий рольовий стереотип. В роботі робиться висновок, що автоматична ідентифікація класових стереотипів може підтримувати краще

розуміння програми та відновлення дизайну в цілому. Це також може допомогти виявити погані класи для рефакторинга. Його навіть можна використовувати для оцінки та покращення дизайну або використовувати як індикатор поганого дизайну [11].

3.1.3 Створення стереотипів за допомогою машинного навчання

В [26] представляється підхід машинного навчання для класифікації стереотипів ролей класу програм Java з відкритим кодом. Цей підхід вимагає певної форми статичного аналізу вихідного коду для класифікації класів.

У документі використовується екстрактор функцій, щоб отримати якомога більше інформації та даних, подібно до статичного аналізу коду. Базовий набір даних створено для навчання кількох моделей машинного навчання. Їхній класифікатор ML покращує існуючий метод класифікації на основі правил, оскільки підхід ML класифікує всі класи. Підходи на основі правил залишають багато класів некласифікованими. Витягнуті функції були класифіковані на п’ять категорій, які мають клас, а саме доступність, складність, функціональність, співпраця та правила іменування [26]. Потім у документі створюється базовий набір даних правдивості, вручну позначаючи стереотипи ролей класів у класах Java за схемою критеріїв. Отримані мітки базової істинності потім додаються до набору даних з усіма характеристиками, які були витягнуті з їх інструменту. Потім набір даних, що містить основні мітки істини та ознаки, використовується як вхідні дані для трьох різних класифікаторів: випадкового лісу, мультиноміального простого Байєса та опорного векторного механізму. Показано, що алгоритм Random Forest має найкращу класифікаційну продуктивність.

3.2. Представлення пропонованої методології

Ми обираємо три тематичні дослідження та збираємо їх вихідний код (Крок 1). По-друге, ми визначаємо та встановлюємо еталонну істину (Кроки

2 і 3). Далі, ми виділяємо характеристики з вихідного коду, які будуть використовуватися алгоритмами машинного навчання (Крок 4). Після цього ми проводимо експерименти з різними алгоритмами машинного навчання (Крок 5). Нарешті, ми оцінюємо ефективність алгоритмів машинного навчання (Крок 6) та досліджуємо еволюцію структури програмного забезпечення у контексті стереотипів ролей (Крок 7). У наступних підрозділах ми детально розглядаємо ці кроки.

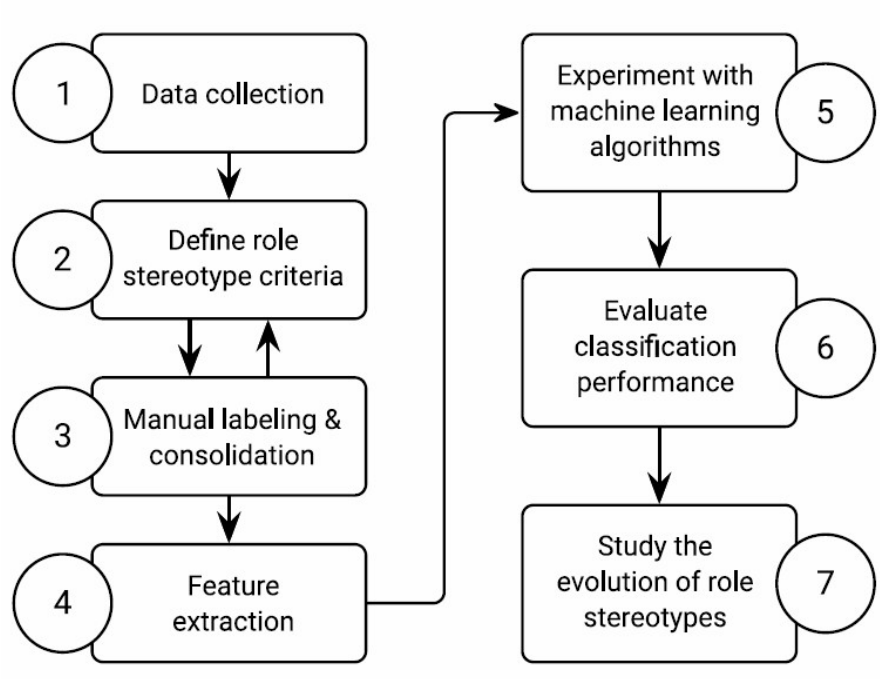

Рис. 3.1. Етапи методології

Збір даних

У цьому дослідженні використовується три програмні системи з відкритим кодом як тематичні дослідження: K-9 Mail, Bitcoin Walle та SweetHome3D. Таблиця 3.1 містить описову інформацію про ці проєкти.

Таблиця 3.1.

Опис проектів використаних для дослідження

#	OSS project	Version	#Class	#Release	#Contributors	#Stars	Type
1	K-9 Mail	v5.304	779	367	202	4276	Mobile/Android
2	Bitcoin Wallet	v6.31	222	274	26	1705	Mobile/Android
3	SweetHome3D	v5.6	546	46	n/a	4.7/5.0	Desktop

K-9 Mail та Bitcoin Wallet є Android-застосунками, що розміщені на GitHub, тоді як SweetHome3D — це Java-застосунок, розміщений на SourceForge.

Ці проєкти були обрані для дослідження з таких причин:

- Вони використовують Java як основну мову програмування.

- Це активні проєкти з відкритим кодом: вони існують багато років, і для них доступні стабільні версії. Ці проєкти не є студентськими.

- Вони відрізняються за розміром (кількість класів), доменом та технологією (Android та чиста Java), а їхні розміри виходять за межі «іграшкових» проєктів.

- Вони належать до доменів, які можуть бути зрозумілі нефахівцям.

Ми завантажили вихідний код цих проєктів з відповідних посилань на GitHub та SourceForge. Під час аналізу було виявлено невелику кількість вкладених класів, які могли б впливати на процес виділення характеристик зовнішніх класів. Тому наступним кроком стало вилучення цих вкладених класів в окремі незалежні класи. Процес вилучення проводився наступним чином. Спочатку вихідний код було розібрано за допомогою інструмента srcML. Для заданого вихідного коду srcML створює список класів, включаючи вкладені класи та їхні деталі, у стандартизованому XML-поданні. Далі, за допомогою XPath-запитів ми згенерували всі класи з збереженого XML-файлу. У результаті кожен вкладений клас було вилучено в окремий Java-файл. Після цього ми використовували Checkstyle для видалення невикористаних імпортів у кожному класі, щоб отримати точну кількість імпортів та кількість рядків коду в класі. Скрипти для автоматизації цього процесу вилучення включені до пакета реплікації цієї статті (Replication package, 2018). Після виконання цих кроків ми отримали загалом 1547 Java-класів у трьох проєктах.

Критерії для стереотипів ролей

Для створення еталонної істини для алгоритмів машинного навчання ми спершу встановлюємо критерії, які використовуватимуться експертами

для ручної класифікації класів за стереотипами ролей. Автори отримали початкові критерії, вивчаючи описи. Далі ці критерії були уточнені та скориговані на наступних зустрічах, під час яких автори оцінювали додаткові набори класів. Критерії поділяються на три категорії:

1) Критерії, що стосуються характеристик класів,

2) Критерії, що стосуються взаємозв'язків між стереотипами ролей,

3) Інші критерії.

Далі ми розглянемо кожну з цих категорій.

Критерії, що стосуються характеристик класів

Ці критерії зосереджуються на внутрішніх (статичних) властивостях класів. Розглянемо стереотип Structurer як перший приклад: рисунок 3.2 показує критерії, що використовуються для характеристики класів типу Structurer.

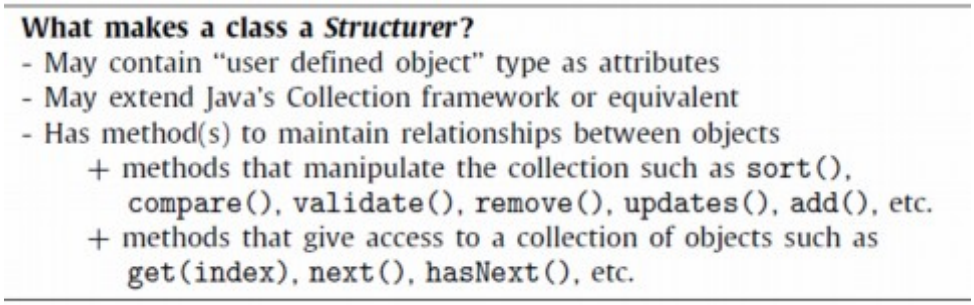
What makes a class a *Structurer*?
- May contain "user defined object" type as attributes
- May extend Java's Collection framework or equivalent
- Has method(s) to maintain relationships between objects
  + methods that manipulate the collection such as `sort()`, `compare()`, `validate()`, `remove()`, `updates()`, `add()`, etc.
  + methods that give access to a collection of objects such as `get(index)`, `next()`, `hasNext()`, etc.

Рис. 3.2. Властивості класу Structurer

У цьому випадку ми звертаємо увагу на типи даних атрибутів, використання бібліотек та вміст методів у класі, щоб визначити, чи здатний клас організовувати або маніпулювати колекцією об'єктів. Інший приклад — Information Holder, який може включати механізми збереження даних (файли або бази даних). Інші властивості класів, такі як назва класу та методи отримання/зміни значень (getter/setter), використовуються для інших стереотипів ролей. Повний список критеріїв для всіх стереотипів можна знайти в пакеті реплікації цього дослідження [25].

Деякі з критеріїв частково схожі з існуючими правилами. Наприклад, ми обидва розглядаємо методи отримання та встановлення значень (getter і setter) як показники стереотипу Information Holder (у нашій класифікації) та Entity/Data Provider. Проте, наші критерії також враховують інші характеристики, такі як функціональність збереження даних.

3.2.1. Критерії щодо взаємозв'язків між ролями

Вірфс-Брок зазначає, що "ролі, які відіграє об'єкт, передбачають певні типи співпраці". Спираючись на його концепцію, ми розробили набір критеріїв, що аналізують співпрацю стереотипів з іншими класами. На основі теорії стереотипів ролей і співпраці було створено графік (рис. 3.3).

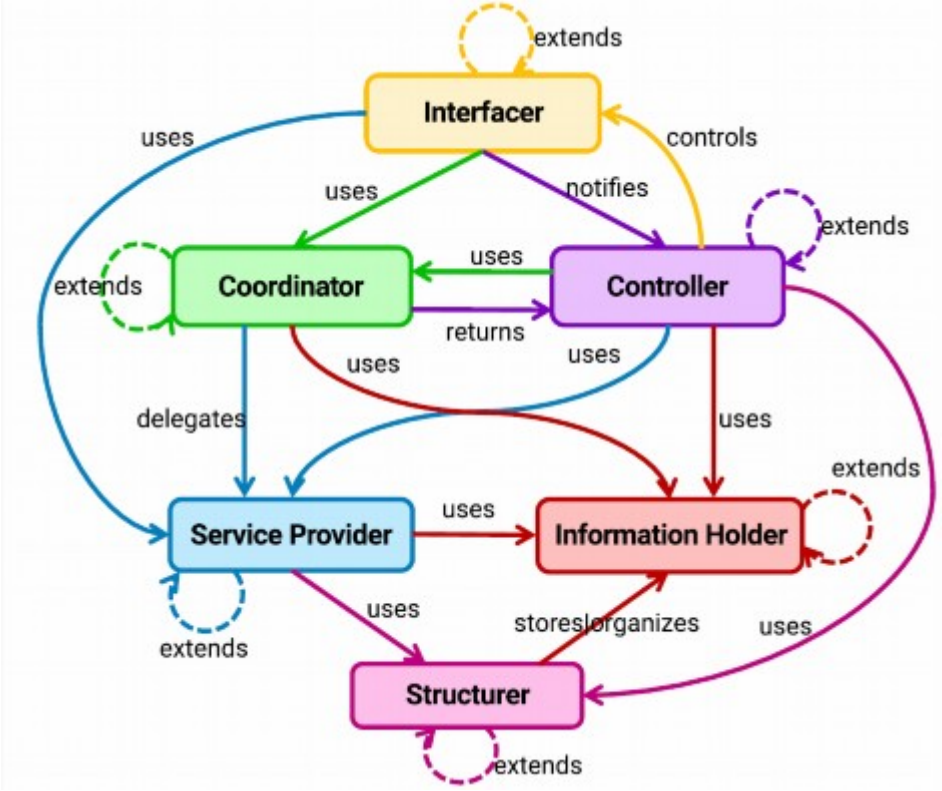

Рис. 3.3. Взаємозв’язки між ролями стереотипів

Графік демонструє типові взаємозв'язки між різними стереотипами ролей. Наприклад, Information Holders зазвичай використовуються Controllers, Service Providers або Structurers, але не Coordinators чи Interfacers.

Таким чином, наявність або відсутність зв'язків може слугувати критерієм для визначення можливої ролі класу.

Нижче наведено додаткові приклади критеріїв, що стосуються взаємозв'язків між ролями: Structurer може бути пов'язаний з декількома Information Holders; Service Provider може зберігати інформацію шляхом співпраці з класами Information Holder та Structurer. Як посередник між різними рівнями системи, Interfacer може співпрацювати з Coordinators та Service Providers на кожному рівні для виконання міжрівневих завдань; при цьому клас Controller часто керує такими завданнями.

Ми визначили додаткові критерії, спрямовані на відмінності в поведінці між різними стереотипами ролей. Наприклад, і Service Provider, і Controller можуть включати логіку управління потоком виконання. Проте задум дизайну цих класів вказує на те, що рішення, прийняті Controller, мають впливати на ширший контрольний потік системи, тоді як рішення, прийняті Service Provider, повинні переважно впливати на потік всередині самого класу.

По-друге, ми звернули особливу увагу на проєкти, пов'язані з Android. Зокрема, Android-застосунки будуються на базі Android-фреймворків, які інкапсулюють низькорівневу функціональність операційної системи Android. Таким чином, деякі класи Controller, Structurer та Interfacer на рівні інтерфейсу користувача та управління активностями, а також їх співпраця можуть залишатися прихованими. Ролі та відповідальність цих класів, що розширюють або реалізують базову функціональність, можуть бути недооціненими. Експерти приділяють особливу увагу, переглядаючи ролі та співпрацю їхніх предків (Android-класів), переважно використовуючи посилання на API Android.

Іноді клас може виконувати більше однієї ролі. Можливість наявності кількох ролей. У такому випадку експерти обговорюють та обирають найбільш виразну роль для цього класу, а також фіксують додаткову роль, якщо така є.

Ручне маркування та узгодження

Для встановлення узгодженості критеріїв і методів їх застосування ми використовували ітеративний процес. Спочатку ми випадково обрали по 20 класів для кожного проєкту. Потім виконується ручне маркування цих класів. Ці кроки повторювалися ще два рази для проєкту K-9 Mail, оскільки в ньому найбільша кількість класів, доки критерії не стали достатніми або насиченими. Далі маркували всі інші класи. У результаті ми створили еталонну базу з 1547 промаркованих класів, що охоплюють усі три випадки. У таблиці 3.1 подано розподіл усіх класів в еталонній базі за проєктами та за прототипами ролей.

Таблиця 3.1.

Розподіл ролей-стереотипів у кожному досліджуваному проекті

Project	CO	CT	IH	IT	SP	ST	Total
K-9 Mail	79	20	231	77	323	49	779
Bitcoin Wallet	2	5	83	62	57	13	222
SweetHome3D	21	38	227	63	159	38	546
Total	102	63	541	202	539	100	1547

У порівнянні з класичним підходом, деякі їхні ознаки схожі на отримані, наприклад, numMethod, setters, getters, і numOutboundInv. Однак вони враховували меншу кількість ознак, тоді як більшість з наших ознак не були розглянуті в їхніх роботах.

Витягування ознак

Наша класифікація базується на статичному аналізі Java-коду системи. Спочатку ми витягуємо ознаки за допомогою інструменту srcML. Для заданого вихідного коду srcML створює список класів та їхніх деталей у стандартизованому XML-представленні. Далі значення ознак розраховуються за допомогою XPath-запитів. Ми обрали 23 ознаки вихідного коду, що відповідають критеріям, використаним у ручній класифікації. У Таблицях 3.2 - 3.6 наведено перелік ознак (F) та коротку суб’єктивну думку про їхню

доцільність для класифікації різних стереотипів ролей. Ознаки згруповані у п’ять категорій (C), а саме: Доступність (Accessibility), Складність (Complexity), Функціональність (Functionality), Іменування (Naming Convention) та Співпраця (Collaboration).

Таблиця 3.2.

Функції доступності, які представляють, наскільки доступним є клас та його вміст

№	Feature name	Data type	Explanation
F01	*classPublicity*	string ('default', 'private', 'protected', or 'public')	The access modifier of the class. We assume that *Service Provider* and *Information Holder* classes might offer public access so that other classes can collaborate with them.
F02	*numPublicMethods*	integer	The number of public methods inside the class. We assume that an *Information Holder*, an *Interfacer*, a *Service Provider*, or an *Interfacer* class might have many public methods.
F03	*numPrivateMethods*	integer	The number of private methods inside a class. We assume that a *Controller*, *Coordinator*, and *Service Provider* might distribute the job on separate methods inside the class.
F04	*numProtectedMethods*	integer	The number of protected methods inside a class. We assume that a *Controller*, *Coordinator*, and *Service Provider* might distribute the job to separate methods inside the class. These methods might still be used or overridden by any sub-classes.

Таблиця 3.3.

Ознаки складності, що представляють складність класу

№	Feature name	Data type	Explanation
F05	*loc*	integer	The number of lines in the class' source code. We assume that the *Controller* and *Service Provider* stereotype will have more lines of code than the other.
F06	*numIfs*	integer	The number of conditional statements in the class body, i.e., the `if`/`if-else` and `switch` statements. *Controller* classes might use lots of conditional statements to make decisions and to control workflows.
F07	*numParameters*	integer	The total number of parameters in all methods in the class. We assume that methods in a *Service Provider* or a *Coordinator* class might have many parameters.
F08	*numAttr*	integer	The number of attributes declared in the class. We assume that an *Information Holder* class might have many attributes.
F09	*numMethod*	integer	The number of methods declared in the class (constructors are excluded). We assume that *Service Provider* and *Coordinator* classes have many methods.
F10	*setters*	integer	The number of methods which names start with `set-`. We assume that this method is a setter method, i.e., the method that modifies a variable in an *Information Holder*.
F11	*getters*	integer	The number of methods which names start with `get-`. We assume that this method is a getter method, i.e., the method that accesses variable values in an *Information Holder* class.

Таблиця 3.4.

Функції угоди про іменування для виявлення певних умов іменування в назві класу

№	Feature name	Data type	Explanation
F19	*numWordName*	integer	The number of words in the class name. We assume that *Information Holder* and *Structurer* role stereotypes have a simple short name, while the others might have a longer name.
F20	*isOrEr*	boolean	Indicates whether the class name ends with `-or` or `-er`. We believe a *Controller* or a *Service Provider* class is more likely to have a name that ends with `-or` or `-er`.
F21	*isController*	boolean	Indicates whether the class name ends with `-Controller`. We believe that those classes are more likely to be *Controller* classes.

Таблиця 3.5.

Функціональні особливості, спрямовані на виявлення конкретних функцій, які може мати клас

№	Feature name	Data type	Explanation
F12	*isPersist*	boolean	Indicates whether a class has persistence features, i.e., implements a `Serialisable` interface or importing database connectivity libraries. We assume that *Information Holder* classes are more likely to employ persistence features.
F13	*isCollection*	boolean	Indicates whether a class is a subclass of Java's collection library. We assume that a *Structurer* might need it to maintain relations between objects.
F14	*isClass*	boolean	Indicates whether the source code file is a class. We assume that a class can represent all of the role stereotypes.
F15	*isEnum*	boolean	Indicates whether the source code is a Java enum. We assume that the enum type can represent an *Information Holder*.
F16	*isAbstractClass*	boolean	Indicates whether a class is an abstract class.
F17	*isStaticClass*	boolean	Indicates whether a class is a static class. We assume that a static class can represent a *Service Provider* or an *Information Holder*.
F18	*isInterface*	boolean	Indicates whether the source code file is a Java interface. We assume that an Interface can provide methods that a *Service Provider* or a *Structurer* must implement.

Таблиця 3.6.

Особливості співпраці, що вказує на рівень співпраці між класом та іншими класами

№	Feature name	Data type	Explanation
F22	*numImports*	integer	The number of `import` statements in the class. Classes carrying the roles like *Controller*, *Coordinator*, *Interfacer*, and *Service Provider* might need to collaborate with many other classes. Thus, we assume that they might have many `import` statements.
F23	*numOutboundInv*	integer	The number of invocations to methods outside of itself. We assume that the *Coordinator*, *Controller*, and *Interfacer* invoke many methods outside of itself.

3.2.2. Виконання класифікації за допомогою машинного навчання

Ми проводимо експерименти з трьома алгоритмами машинного навчання: Random Forest (RF), Multinomial Naïve Bayes (MNB) та Support Vector Machine (SVM). Ці алгоритми широко використовуються в дослідженнях у сфері машинного навчання та демонструють хороші результати в різних застосунках. Ми не проводимо експерименти з нейронними мережами, оскільки вони потребують дуже великих навчальних вибірок (десятки тисяч об'єктів), які на даний момент у нас відсутні. Для оцінки продуктивності кожного алгоритму ми використовуємо стратифіковану перехресну перевірку на 10 вибірках (stratified 10-fold cross-validation), вимірюючи результати за показниками точності (Precision),

повноти (Recall), F1-міри (F1-Score) та коефіцієнта кореляції Меттьюса (Matthews Correlation Coefficient, MCC).

Ми проводимо експерименти з алгоритмами машинного навчання. У першому експерименті ми аналізуємо, який алгоритм забезпечує найкращі результати при класифікації всіх стереотипів ролей. Для цього кожен стереотип ролей розглядається як окрема категорія класифікації, що становить багатокласову класифікацію. Також ми досліджуємо використання техніки повторного вибору SMOTE для вирішення проблеми незбалансованого розподілу стереотипів ролей. Останні дослідження в галузі програмної інженерії використовували SMOTE для роботи з незбалансованими наборами даних. При використанні, техніка надлишкового вибору перевибирає всі стереотипи ролей, окрім найпоширенішого. Ми застосовуємо SMOTE виключно до навчальних наборів для всіх ознак. Нарешті, ми порівнюємо продуктивність при використанні звичайної техніки та техніки повторного вибору SMOTE.

У другому експерименті ми досліджуємо, які ознаки є найважливішими для класифікації кожного стереотипу ролей. Для цього ми використовуємо лише один алгоритм машинного навчання — той, що показав найкращі результати в першому експерименті. У цьому випадку ми виконуємо бінарні класифікації для кожного стереотипу, тобто використовуємо дві категорії:

1) конкретний стереотип;

2) всі інші стереотипи разом.

Ми оцінюємо важливість ознак у цій класифікації, використовуючи Scikit-toolkit для машинного навчання та вбудований метод для обчислення важливості ознак на основі оцінок Gini. Цю оцінку можна отримати, розраховуючи значущість вузла для кожного розділення ознаки та нормалізуючи її відносно суми всіх значень важливості ознак.

Ми досліджуємо узагальненість навченого класифікатора машинного навчання, оцінюючи використання різних програмних проєктів як навчальних даних для класифікації інших програмних проєктів. Для цього

використовуємо найпродуктивніший алгоритм машинного навчання з попереднього експерименту та аналізуємо його продуктивність з різними наборами навчальних даних.

Дослідження еволюції стереотипів ролей

Щоб зрозуміти еволюцію структури програмного забезпечення щодо стереотипів ролей, ми обираємо значну кількість версій з наших трьох випадків. Ми застосовуємо класифікатор до вихідного коду цих версій та проводимо кількісний аналіз і вимірювання на отриманих даних. Крім того, ми використовуємо методи дослідницької візуалізації даних для виявлення закономірностей та динаміки в цих програмних проєктах.

3.3. Методика навчання класифікатора на основі маркування класів

У цьому розділі описано процес створення основного набору істинних даних для навчання класифікатора та наведено ідеї, отримані в результаті ручного маркування класів. Крім того, він пояснює процес вилучення функцій і визначає конкретні функції, які вибираються.

Збір даних

Автоматизований процес визначення рольових стереотипів потрібен для обробки понад тисячі файлів і виявлення будь-яких потенційних порушень і проблем. Жодна література раніше не класифікувала рольові стереотипи в C++ або C через класифікатор. Існує потреба у достатньо великому наборі даних, що містить основні істини щодо рольового стереотипу кожного класу . У цьому розділі пояснюються етапи створення такого набору даних і вибір класів у ньому.

Вибір класів всередині основного набору даних істинності здійснюється на основі списку правил, які повинні дотримуватися для кожного класу:

1. Клас не повинен бути тестовим.

2. Клас не повинен містити згенерований код.

Можна відфільтрувати будь-який з тестових класів, оскільки всі класи мають належні назви. Фільтр також можна використовувати для видалення будь-яких класів, які мають показник нижче 9,0. Вибір класів також не повинен бути згенерованим кодом, який було відфільтровано вручну під час позначення основних істин. Рівень PosC системи позиціонування має багато класів і найрізноманітніші класи порівняно з іншими рівнями. Таким чином, базовий набір даних міститиме лише класи з рівня poscore .

Досить великий базовий набір істинних даних довелося створювати вручну, оскільки жодна інша література не мала набору даних для стереотипів ролей класу в C++ або налаштованого для архітектури Philips. Зазвичай клас має стандартизовані угоди про найменування та шаблони, узгоджені для всієї системи. Однак інколи може статися так, що шаблони або домовленості про найменування відрізняються між підкомпонентами системи, як це буде видно пізніше під час вилучення функцій.

3.3.1. Визначення критеріїв рольового стереотипу

Набір критеріїв будується на основі визначення кожного рольового стереотипу, критерії попередніх статей для кожного стереотипу ролі та скориговані для мови програмування C++. Нижче наведено приклад набору критеріїв для власника інформації.

• Клас зберігає інформацію всередині об'єкта або змінної.

• У класі є об’єкти або змінні, які зберігаються.

• Клас використовує або визначає типи, поля або будь-які інші типи об’єктів, які використовуються виключно для зберігання інформації. (друки , журнали, JSON)

• Клас не має прямого зв’язку з іншими класами, за винятком випадків обміну або збору інформації, наприклад, через методи get/set.

• Клас має кілька визначених методів get/set.

• Клас може перетворювати інформацію в іншу форму під час її зберігання.

• Клас може містити « визначення» в назві класу або файлу.

• Клас має численні публічні методи.

Ці критерії ґрунтуються на загальних обов’язках власника інформації. Кожна точка критерію в цьому списку має однакову вагу. Якщо клас задовольняє більшість із цих вимог, його можна класифікувати як власника інформації. Однак це також залежить від того, наскільки добре клас задовольняє інші критерії для решти п'яти рольових стереотипів. Можуть бути випадки, коли клас задовольняє більшості, якщо не всім критеріям для Носія інформації, а також відповідає критеріям іншого рольового стереотипу. У цих випадках клас повинен бути позначений як рольовий стереотип, який задовольняє найвищий відсоток балів для кожного рольового стереотипу.

Ручне маркування рольового стереотипу для кожного класу було ретельно виконано відповідно до набору критеріїв. Класи були розмічені на основі статичного аналізу коду за допомогою Clair. Рання версія екстрактора функцій була вже створена на етапі підготовки дисертації для підтримки ручного маркування класів. Критерії та мітки заднім числом оновлювалися одночасно щоразу, коли виявлялися нові відомості, специфічні для системи позиціонування.

Загалом 294 класи всередині PosC були переглянуті вручну та позначені рольовим стереотипом Розподіл кожного рольового стереотипу можна побачити в таблиці 3.7.

Таблиця 3.7.

Розподіл набору даних

Role Stereotype	Count
Nothing	81
Controller	31
Coordinator	36
Information Holder	22
Interfacer	52
Service Provider	60
Structurer	12
Total	294

Кількість класів, які були позначені як «нульові», є великою, оскільки була присутня висока частота шаблону проектування команд, що дозволяло класам складатися лише з кількох рядків. Вони були позначені як ніщо, оскільки жодна інформація з цих класів не могла бути отримана для впевненого призначення рольового стереотипу. Такий самий ярлик присвоюється класам, які є надто обширними та довгими. Ці класи зазвичай мають понад тисячу рядків коду і іноді, завдяки цьому, задовольняють кілька критеріїв. Це також дозволяє класифікатору класифікувати ці випадки та потенційно знаходити порушення AR2, коли немає жодного з двох випадків. Таким чином, позначення Nothing тут по суті не означає порушення, оскільки малі класи також були позначені Nothing тут.

Розподіл шести рольових стереотипів у таблиці 3.7 здається нормальним для системи позиціонування. Зазвичай постачальники послуг є найпоширенішим рольовим стереотипом у системах. Велика кількість інтерфейсів є незвичною, але на це можна відповісти через характер програмного забезпечення позиціонування, яке ближче до введення даних користувача та перетворює вхідні дані на різні компоненти.

3.3.2. Процес екстракції функцій

Функція Extractor — це інструмент, який буде реалізовано для цього призначення. Основна функція цього інструменту полягає, як випливає з назви, у вилученні функцій і фактів із класів C++ і C. Це стало можливим завдяки створенню моделей, вилученню функцій і їх збереженню. У цьому розділі пояснюються всі ці дії та їхні негайні результати.

Усі моделі M3 можна створити за допомогою інструменту, розробленого в [17]. Крім того, також можна генерувати моделі AST шляхом коригування кількох рядків коду. Замість використання CreateM3FromCppFile під час генерації моделі M3 можна використати createMSAndAstPromCppFile для генерації як моделі M3, так і моделі AST,

використовуючи однакові аргументи. Кожен файл буде виводити дві моделі, які будуть передані в екстрактор функцій.

У лістингу 3.1 показано доповнення до генерації моделі перевірки ерозії. Після цього фрагмента коду обидві моделі зберігаються у файлі двійкових значень за допомогою методу WriteBinaryValueFile () із бібліотеки Rascal Util.

Лістинг 3.1. Покоління моделі M3 і AST

```
try {
    model = generateModel(source, stdLib, includeDir)
} catch Java(RunTimeException) {
    print error
}
if (generateM3AndAST) {
    try {
        ast = generateAST(source, stdLib, includeDir)
    } catch Java(RunTimeException) {
        print error
    }
}
```

Після генерації моделей можна переглянути кожну модель M3 і AST окремо та знайти шаблони та особливості всередині. Залишається питання, яку ознаку виділити та яка може бути корисною для класифікатора для визначення рольового стереотипу? На це питання частково відповідає інша робота, яка також класифікує рольові стереотипи для іншої мови програмування або за допомогою інших засобів [16, 27]. Однак деякі функції, представлені в цих документах, несумісні з середовищем C++. Функції, які можна отримати, також обмежені якістю та структурою моделей M3 і AST. Таблиця 3.8 показує характеристики, які можна отримати за допомогою згенерованих моделей M3 і AST.

Функції тут здебільшого взяті з моделі M3, оскільки їх легко знайти в моделі M3. У лістингу 3.2 показано, як функції витягуються шляхом доступу до відношення в m.declarations. Це список зв’язків, який відображає абонента на викликаного.

Функції були дещо змінені, наприклад numControls , який спочатку був numControllers, який підраховує кількість методів і функцій, які закінчуються

на 'controller'. Цю функцію було змінено, щоб відповідати правилам найменування кодової бази Philips. Більшість функцій у таблиці 3.8 можна відразу взяти з моделі M3. Однак функції numSetters, numGetters, numoutbound - Inv і numIfs вимагають обробки даних моделі M3 і відвідування згенерованого AST. Єдиною функцією, яка вимагала прямого доступу до файлу, у якому була створена модель, був numLines.

Таблиця 3.8.

Безпосередньо витягнуті функції

Feature	Model	Explanation
numFunctions	M3	The number of functions
numMethods	M3	The number of methods.
numVariables	M3	The number of variables.
numParameters	M3	The number of parameters in method definitions.
numFields	M3	The number of fields defined.
numStructures	M3	The number of structs defined
numTypeDef	M3	The number of type definitions defined.
numEnums	M3	The number of enums defined.
numIncludes	M3	The number of includes that is required by the class.
numLines	File	The number of lines of the file. (loc)
hasNameSpace	AST	True when a namespace is defined
numPublicMethods	M3	The number of public methods
numPrivateMethods	M3	The number of private methods
numProtectedMethods	M3	The number of protected methods

Лістинг 3.2. Вилучення функцій з моделі M3

```
bool isFunction(loc entity) = entity.scheme == "cpp+function" or entity.scheme
    == "c+function";
list[loc] functions(M3 m) = [e | <e, x> <- m.declarations, isInSource(x, src)
    and isFunction(e))];
```

Кількість рядків витягується за допомогою простого методу Util, який зчитує надане розташування файлу аргументів і список, де кожен елемент у списку є одним рядком у вихідному коді. Розмір списку дорівнює кількості рядків.

Простір імен можна знайти в обох моделях, M3 і AST. Однак, завдяки деяким тестам, простір імен було швидше та надійніше отримати з AST, оскільки простір імен не можна було знайти в m.declarations. Це робиться, відвідавши AST і перевіривши наявність оголошення простору імен, як

показано в лістингу 3.3. Clair визначає два можливі випадки в просторі імен, який можна знайти в AST. Обидва ці випадки видимі тут і використовуються для вилучення назви простору імен.

Лістинг 3.3. Вилучення простору імен з AST

```
visit(ast) {
  case \namespaceDefinition(Name name, list[Declaration] declarations): {
    if(checkSource(name.src, ast.src) == 1) {
      visit(name) {
        case \name(str \value): return \value;
      }
    }
  }
  case \namespaceDefinitionInline(Name name, list[Declaration] declarations):
       {
    if(checkSource(name.src, ast.src) == 1 && name.src.extension == "cpp" &&
       name.isMacroExpansion == false) {
      visit(name) {
        case \name(str \value): return \value;
      }
    }
  }
}
return ""; // No name space defined in class
```

Характеристики, наведені в таблиці 3.9, показують усі характеристики, які потребують виведення або обчислення. Для цих функцій потрібно знайти відповідні шаблони в AST або отримати дані моделі M3 для конкретної функції.

Таблиця 3.9.

Специфічні та похідні функції

Feature	Model	Explanation
numSetters	M3	The number setters defined.
numGetters	M3	The number getters defined.
numOutboundInv	AST	The number of calls that go outwards.
numGettersInvoked	M3 & AST	The number of outward get calls.
numSettersInvoked	M3 & AST	The number of outward set calls.
numIfs	AST	The number of if, if else, and switch cases.
numLoops	AST	The number of for, while, and do loops.
numNamedTypes	AST	The number of non-primitive types defined.
numWordsClass	M3	The number of words in the class name.
numControl	M3	The number of "control" in names.
numOrEr	M3	The number of "er" and "er" in names.
kinStubAndProxyInName	M3	True when "Stub" or "Proxy" in name.
definitionInName	M3	True when "definition" in name.
numUIInName	M3	The number of "UI" in names.
utilityInName	M3	True when "utility" in name.
atlInName	M3	True when "ATL" in name.
methodsHasSameName	M3	The number of methods with same name.

Для визначення кількості геттерів (numGetters) та сеттерів (numSetters) використовується регулярний вираз, який дозволяє знаходити будь-які випадки "get" та "set" у назві методу. Це суттєво залежить від того, яку конвенцію іменування використовують розробники Philips. У методах можуть використовуватися як стиль camelCase, так і підкреслення, що вимагає розробки регулярного виразу, здатного врахувати обидва стилі. Для знаходження всіх випадків "set" у методах, що використовують стиль camelCase, застосовується наступний регулярний вираз: /(\WSet[A-Z].+|\Wset[A-Z].+|.+Set\W.+)/. Цей вираз знаходить методи з такими назвами, як C

```
/class/setScore() та C
/class/scoreSet(), але не C
/class/setupSettings().
```

Інший регулярний вираз необхідний для випадків, коли використовується стиль іменування з підкресленнями: (\Wset .+|.+ set(.+). Цей регулярний вираз розпізнає назви методів, такі як C

```
/class/set score() та C
/class/score set(), але не C
/class/data set tester().
```

Аналогічно, кількість геттерів (numGetters) можна визначити, замінивши "set" та "Set" на "get" та "Get" відповідно.

Для обчислення numOutBoundInv проводиться перевірка кожного виклику функції в абстрактному синтаксичному дереві (AST) та його джерела. Якщо виклик функції знаходиться всередині класу, але не є методом, визначеним у цьому ж класі, то він вважається вихідним викликом (outbound invocation). Розроблений інструмент бере назви викликів функцій та згодом перетворює їх у числове значення. Лістинг 4.4 демонструє код, який витягує виклики функцій. Спочатку кожен вузол у AST проходиться до тих пір, поки не буде знайдено виклик функції. Після цього викликається

інший метод, який знаходить поле та назву виклику функції за допомогою getCppName(). Метод повертає функцію у форматі "/field.method()", яка додається до списку всіх викликів.

Лістинг 3.4. Вилучення зовнішніх викликів

```
// Extract a list of outbound invocations
list[str] astOutboundInvocations(Declaration ast) {
  list[str] functionCalls = [];
  visit(ast) {
    case \functionCall(Expression functionName, list[Expression] arguments): {
      if (isSameSource() && functionName.src.extension == "cpp" && !
          functionName.isMacroExpansion) {
        functionCalls += ("/" + functionCallName(functionName) + "()");
      }
    }
  }
  return functionCalls;
}

// Extract the functions name and its field owner
str functionCallName(Expression functionName) {
  switch(functionName) {
    case \idExpression(Name name): {
      return getCppName(name);
    }
    case \fieldReference(Expression fieldOwner, Name name): {
      str fieldOwnerString = functionCallName(fieldOwner);
      return fieldOwnerString + "." + getCppName(name);
    }
    case \fieldReferencePointerDeref(Expression fieldOwner, Name name): {
      str fieldOwnerString = functionCallName(fieldOwner);
      return fieldOwnerString + "." + getCppName(name);
    }
    default: {
      return "";
    }
  }
}
```

Визначення кількості циклів (numLoops)

У лістингу 3.5 показано процес збору інформації про кількість циклів (numLoops). Цей фрагмент коду підраховує кількість відвідувань AST (абстрактного синтаксичного дерева) різних типів циклів. Всього існує шість різних типів циклів for, do та while, визначених як шаблони AST у системі Clair.

Інший набір характеристик, представлений у таблиці 3.9, перевіряє певні патерни іменування в коді. Ці характеристики були отримані під час

інтерв’ю після узгодження з експертами щодо їхніх патернів та зв’язку з рольовими стереотипами.

Лістинг 3.5. Вилучення циклів з AST

```
int count = 0;

visit(ast) {
    case \forWithDecl(Statement sInitializer, Declaration conditionDeclaration,
         Expression iteration, Statement body): count += checkSource(
        sInitializer.src, srcName);
    case \rangeBasedFor(Declaration declaration, Expression initializer,
        Statement body): count += checkSource(declaration.src, srcName);
    case \for(Statement sInitializer, Expression condition, Expression
        iteration, Statement body): count += checkSource(sInitializer.src,
        srcName);
    case \while(Expression condition, Statement body): count += checkSource(
        body.src, srcName);
    case \whileWithDecl(Declaration conditionDeclaration, Statement body):
        count += checkSource(conditionDeclaration.src, srcName);
    case \do(Statement body, Expression condition): count += checkSource(body.
        src, srcName);
}

return count;
```

Наведені нижче характеристики були виявлені в процесі цих інтерв'ю та їх зв'язок із шістьма рольовими стереотипами класів:

- kinStubAndProxyInName.

Деякі вихідні файли всередині будь-якого підкаталогу kin мають у назві класу слова "Stub" або "Proxy". Це патерн, що часто використовується та є послідовним у підкомпонентах kin. Такий патерн іменування вказує на те, що клас є різновидом Interfacer.

- definitionInName

Вихідні файли або назви класів, які містять слово "definition" у назві, коли всередині них є визначення змінних або об'єктів. Вони також можуть містити набори даних. Це може свідчити про те, що клас є Information Holder.

- numUIInName

Деякі класи тісно пов'язані з елементами або компонентами інтерфейсу користувача (UI). У таких випадках клас часто використовує слово "UI" для визначення методів, функцій або змінних. Висока частота використання "UI" може вказувати на Interfacer.

- utilityInName

Клас з назвою, що містить слово "utility", може вказувати на те, що цей клас використовується іншим класом як помічник. Це може свідчити про те, що клас надає певну функціональність іншим класам, як, наприклад, Service Provider.

- atlInName

Клас, у назві якого є "ATL", зазвичай є Interfacer. Подібно до kinStubAndProxyInName, цей патерн використовується більш загально в інших компонентах.

Усі характеристики, наведені в таблицях 3.8 і 3.9, використовуються класифікатором як вхідні дані.

3.4. Виконання процесу класифікації

Класифікатор побудовано на Python з бібліотекою scikit-learn. Для класифікації рольових стереотипів було обрано наступні три класифікатори: Random Forest, Multinomial Naive Bayes (MNB) і XGBoost. Перш ніж вводити згенерований набір даних об’єктів, необхідна певна попередня обробка набору даних. Попередня обробка даних передбачає спочатку видалення та перетворення ознак у цілі числа, щоб вони стали сумісними з класифікаторами. Потім дані розбиваються на ланцюжок і тестовий набір із розділенням у 25%.

Перша ітерація MNB дала точність близько 20%. Ця ітерація мала лише функції, згадані в таблиці 3.8, мала лише близько 150 точок даних у наборі даних, і не було зроблено жодної оптимізації класифікатора чи набору даних. Оцінка перехресної перевірки також була розрахована за допомогою методу перехресної оцінки з бібліотеки scikit-learn. Встановлення кількості згорток на 5 дає результат перехресної перевірки 24%. Випадковий ліс і XGBoost мали дещо вищу точність і показник перехресної перевірки, обидва становили близько 30%. Тепер мета полягає в тому, щоб збільшити ці два показники за допомогою оптимізації набору даних і класифікатора, а також

уникнути поширених проблем машинного навчання, таких як переобладнання. Три параметри були змінені, щоб більше відповідати даним за допомогою XGBoost, tree_method було встановлено значення «hist», категоричні дані було ввімкнено, а ціль класифікатора встановлено на «multi:softprob», оскільки це проблема кількох класифікацій. Таблиця 3.10 показує порівняння кожного класифікатора у вигляді порівняння.

Таблиця 3.10.

Порівняння ефективності кожного класифікатора

Classifier	Accuracy	Cross-Validation
Random Forest	30%	28%
MNB	20%	24%
XGBoost	33%	34%

Лістинг 3.6. Класифікатор XGBoost і звіт про його оцінку

```
preprocess = PreProcessing(csv_directory)
X_train, X_test, y_train, y_test, X_resampled, y_resampled = prepare_data()

# Start XGBoost classifier and calculate their weights
self.clf = XGBClassifier(tree_method="hist", enable_categorical=True, objective
    ='multi:softprob')
classes_weights = class_weight.compute_sample_weight(
    class_weight='balanced',
    y=y_train
)

clf.fit(X_train, y_train, sample_weight=classes_weights)
predictions = clf.predict(X_test)
report = classification_report(y_test, predictions)
```

З першої ітерації було очевидно, що XGBoost перевершує класифікатори випадкового лісу та MNB у всіх випадках. Тому етапи оптимізації зосереджені на оптимізації класифікатора XGBoost. У лістингу 3.6 показано, як застосовуються гіперпараметри та методи Python scikit-learn, які використовуються для створення класифікатора. Тут також розраховуються ваги класів, щоб бути більш збалансованими. Про це йтиметься в наступних підрозділах. Крім того, звіт про класифікацію складається з результатів, який є звітом про роботу класифікатора.

Наразі точність не є ідеальною, тому мета полягає в подальшій оптимізації шляхом зміни параметрів або попередньої обробки даних. У наступному розділі обговорюються кроки оптимізації для підвищення точності та продуктивності класифікатора.

3.4.1. Виконання оптимізації

Перші дві проблеми, визначені з набором даних, — це дисбаланс рольових стереотипів, як видно з таблиці 3.7 , і низька кількість точок даних у наборі даних. Дисбаланс у частоті рольових стереотипів є нормальним для більшості систем, оскільки деякі рольові стереотипи використовуються рідше, ніж інші стереотипи, залежно від контексту системи. Однак результати показують, що класифікатор віддає перевагу класифікації рольових стереотипів, які частіше зустрічаються в наборі даних. Як бачимо в таблиці 3.11 “Service Provider” і “Nothing” є більш поширеними ярликами, ніж будь-який інший рольовий стереотип. Це змушує класифікатор класифікувати ці два стереотипи частіше, оскільки він має найвищу ймовірність того, що клас є будь-яким із цих двох рольових стереотипів і, отже, класифікатор є правильним.

Таблиця 3.11.

Частота ролей стереотипу на основі 150 точок даних

Role Stereotype	Frequency
Nothing	26%
Controller	7%
Coordinator	18%
Information Holder	11%
Interfacer	10%
Service Provider	23%
Structurer	4%

Цей дисбаланс рольових стереотипів можна вирішити шляхом реалізації ваг для кожного рольового стереотипу, надаючи менш частим рольовим стереотипам у наборі даних вище покарання за неправильну

класифікацію під час навчання. Цей підхід дав трохи вищий бал близько 5%, що не був достатньо значним, щоб зробити класифікатор більш точним. Потрібен інший метод для вирішення дисбалансу всередині набору даних. Ваги для кожного рольового стереотипу знаходяться в таблиці 3.12. Ці значення не є безпосередньо засновані на повному наборі даних у таблиці 4.5, а на 25% розподілі, зробленому для тестування.

Таблиця 3.12.

Ваги для кожної ролі стереотипу

Role Stereotype	**Weight**
Controller	1.33766
Coordinator	1.17714
Information Holder	1.96190
Interfacer	0.81746
Service Provider	0.70068
Structurer	3.26984
Nothing	0.51629

Іншим способом боротьби як з дисбалансом даних, так і з низькою кількістю точок даних є застосування методу, який часто використовується для алгоритмів машинного навчання в розпізнаванні образів, званого наддискретизацією. Зазвичай для класифікатора розпізнавання образів можна створити більше даних шляхом обертання зображення або додавання шуму до зображення, оскільки зображення залишається тим самим. Подібна техніка може бути використана для створення додаткових точок даних всередині набору даних основної правди. Однак це створює певний шум у ознаках, що може ускладнити класифікатору знаходження певних закономірностей. Ефективність класифікатора слід ретельно спостерігати, щоб побачити, як він реагує на додатковий шум, який створює наддискретизація. Підходящою технікою наддискретизації для цієї ситуації є SMOTE.

SMOTE є технікою наддискретизації, яка створює додаткові дані шляхом вибору двох класів, позначених тим самим рольовим стереотипом, а потім малювання ребра між їхніми ознаками. Алгоритм SMOTE потім випадково вибирає точку на одному з ребер як нове значення і створює новий клас, позначений як той самий рольовий стереотип. Простий приклад можна побачити на рисунку 3.4.

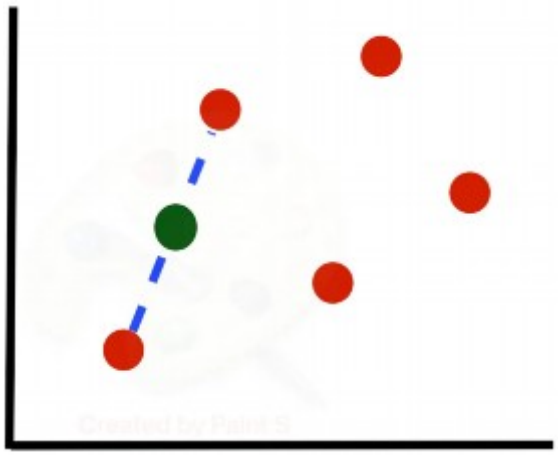

Рис. 3.4. Приклад роботи техніки SMOTE

Використання SMOTE також допоможе з дисбалансом рольового стереотипу, оскільки він наддискретизує лише міноритарні мітки в наборі даних. Це також видно з результатів класифікатора. Загалом до набору даних було додано близько 50 нових класів, що підвищило точність класифікатора до 63% та оцінку перехресної перевірки до 64% зі стандартним відхиленням 7%.

Останнім кроком оптимізації класифікатора є додавання додаткових функцій. Ці функції походять від моделей M3 і AST, які також згадуються в таблиці 3.9. Ці функції не були реалізовані відразу, оскільки в той час тестувалися та розроблялися кілька ітерацій класифікатора, а також одночасно позначалося більше базових істин. Однак важливість ознаки на рисунку 3.5 показує вплив, який кожна ознака має на класифікатор. Функції, які стосуються шаблонів у назві класів і методів, таких як kinStubOrProxylnName , utilityInName та atlInName мають більший вплив на

продуктивність класифікатора, оскільки це шаблони, які визначаються та дотримуються розробниками всередині кодової бази позицій. Властивості самого вихідного коду також важливі для класифікатора, такі як кількість методів або кількість використовуваних TypeDef.

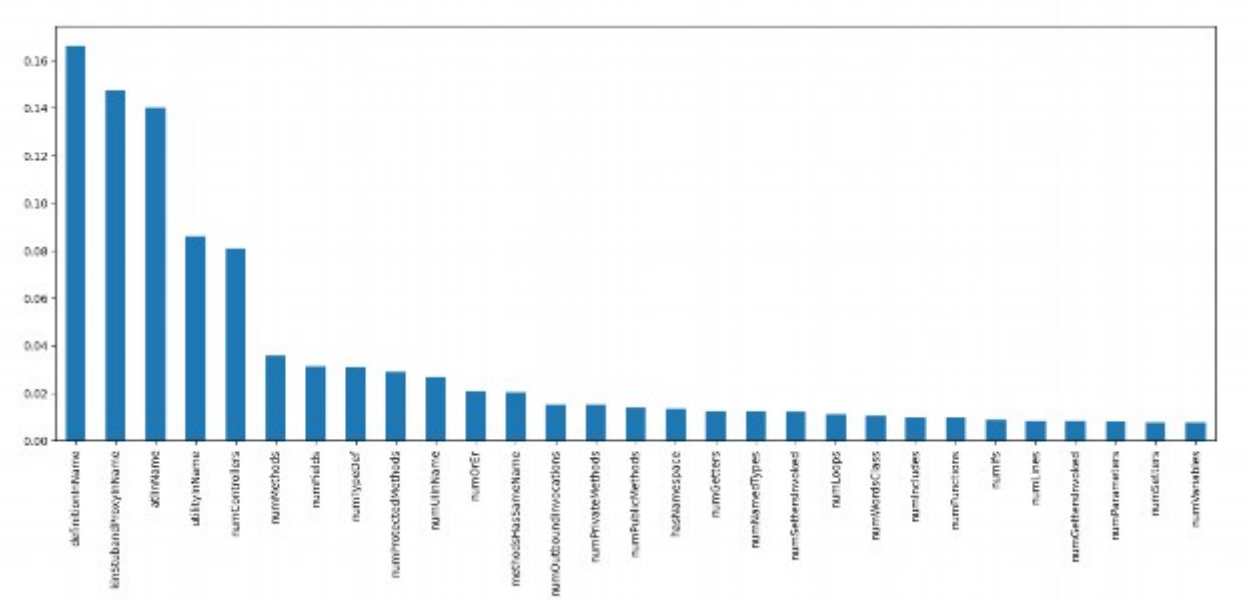

Рис. 3.5. Важливість ознаки класифікатора

Усі ці кроки підвищили точність класифікатора з 30% до 86%, а показник перехресної перевірки – з 24% до 84% зі стандартним відхиленням 10%.

3.5. Виконання візуалізації результатів для виявлення помилок

3.5.1. Генерація JSON

Інструмент візуалізації вимагає файл JSON як вхідні дані у форматі, указаному на сторінці GitHub. Приклад такого вхідного файлу показано в лістингу 3.7. Для створення такого файлу JSON потрібен сценарій Python, який перетворює набір даних для кожного класу у вузол із краями, які можуть включати клас або вихідні виклики функції. Оскільки більшість

функцій, необхідних для візуалізації, уже видобуто, легко створити сумісний файл JSON.

Лістинг 3.7. Приклад виведення файлу JSON


```
{
  "elements": {
    "nodes": [
      {
        "data": {
          "id": "|file:///C:/software/src/code1.cpp",
          "properties": {
            "shortname": "Class 1",
            "kind": "class",
            "rs": "Coordinator",
          }
        }
      },
      {
        "data": {
          "id": "|file:///C:/software/src/code2.cpp",
          "properties": {
            "shortname": "Class 2",
            "kind": "class",
            "rs": "Information Holder",
          }
        }
      }
    ],
    "edges": [
      {
        "data": {
          "id": 1,
          "source": "|file:///C:/software/src/code1.cpp",
          "label": "calls",
          "target": "|file:///C:/software/src/code2.cpp"
        }
      },
```


Для візуалізації країв для кожного класу була потрібна одна додаткова функція. numOutboundInv витягує лише виклики функцій, але не їх призначення. Але, можна отримати місце визначення функції, коли ми створимо всі згенеровані моделі M3. Метод compose від Clair виведе одну складену модель M3, де можна отримати список імен функцій та їх розташування. Однак існує невелика ймовірність отримати неправильне розташування функції, якщо ім’я функції знаходиться поза межами класу . Це трапляється лише в двох випадках, коли деякі функції мають однакові назви або коли функція в межах класу не входить до складеної моделі.

Алгоритм, показаний у лістингу 3.8 перевіряє кожну функцію всередині numOutBoundInv і шукає їх розташування всередині складеної моделі M3.

Лістинг 3.8. Вилучення розташування функції

```
set[str] getFunctionLocation(list[str] outboundCalls, rel[loc,loc]
    functionDefinitions) {
    set[str] callLocations = {};
    set[loc] allDefinitions = {};
    for (str functionCall <- toSet(outboundCalls)) {
        removePartBeforeDot(functionCall) // Changes "field.function" to "
            function"
        removePartBeforeDoublePoints(functionCall) // Changes "str:function" to
             "function"
        removeBrackets(functionCall) // Changes "function()" to "function"
        removeSlashes(funtionCall) // Changes "/function" to "function"
        allDefinitions = {};
        try
            allDefinitions = functionDefinitions[[x | x <- functionDefinitions
                <0>, contains(toLowerCase(x.path), toLowerCase(functionCall))
                ][0]]; // Get all function location from functionDefinitions
                that have the same name
        catch NoSuchElement(0) :
            iprintln("No function definition found for: " + functionCall);

        for (loc def <- allDefinitions) {
            // Only take function that has same name as functionCall
            if (filterTestsAndIncludes(def)) {
                callLocations += def.path;
                break; // We dont want duplicate function calls with same name
            }
        }
    }
    return callLocations;
}
```

Ця частина інструменту, навіть оптимізована, працює повільно, тому її рекомендується запускати лише тоді, коли користувач хоче візуалізувати результати, викликавши екстрактор функцій із додатковим логічним прапорцем getFunctionCalls як аргумент. Існує проблема з цим алгоритмом, оскільки методи можуть мати однакові назви та параметри, коли обидва не знаходяться в одній області. Цей метод не може перевірити область перевіреного класу, що може призвести до неточних країв у візуалізації через створення країв поза межами методу.

3.5.2. Структура ролі стереотипу

Можна швидко зрозуміти архітектуру системи, візуалізуючи комунікацію між класами та вказуючи їхній стереотип класової ролі.

Візуалізація на рисунку 3.6 показує, як Координатор (зелений) викликає Контролер (фіолетовий), який взаємодіє з кількома постачальниками послуг (синій). Деякі виклики відбуваються в обох напрямках, оскільки Клас 20 і Клас 21 викликають один одного. Це може вказувати на проблему з дизайном одного з цих класів.

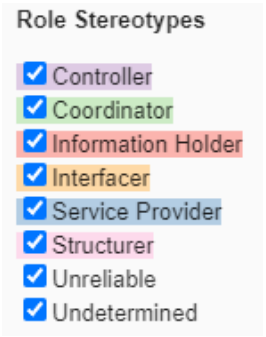

Рис. 3.6. Легенда ролі стереотипу

Рисунок 3.7 показує комунікації невеликого каталогу. Кожен блок є каталогом. Якщо один блок міститься всередині іншого блоку, це означає, що цей блок є підкаталогом більшого блоку. За допомогою цього подання можна побачити, як рольові стереотипи взаємодіють з іншими каталогами.

Наразі ця візуалізація є інструментом для допомоги користувачам зрозуміти структуру рольових стереотипів.

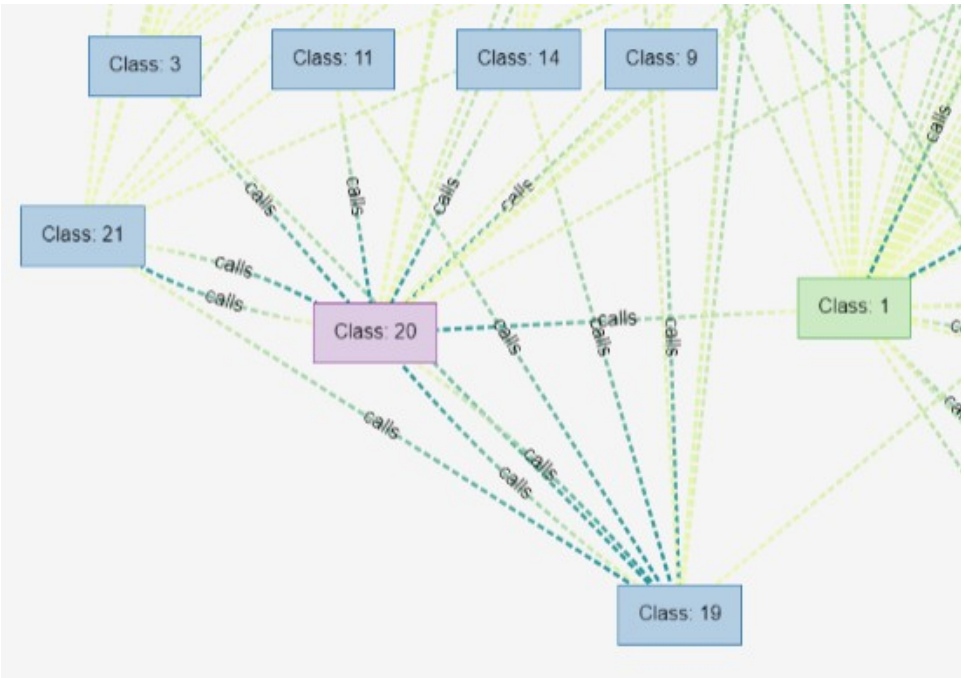

Рис. 3.7. Структура стереотипу внутрішньої ролі

На рисунку 3.7 показано кілька ящиків, які знаходяться в окремих каталогах. Ці менші блоки не є центром цієї візуалізації, тому, щоб візуалізація була організованою, показано лише класи, які взаємодіють із найбільшим блоком. Однак він повинен давати достатній контекст про взаємодію між класами та їхній стереотип про роль у класі. Однією з таких структур є невеликий ящик, усередині якого є два класи власників інформації (червоний), які використовуються для надання інформації класам постачальника послуг (синій) у великому каталозі. Візуалізація також показує вузли, які не забарвлені. Це вважається порушенням класифікатора; це дозволяє розробникам більше переглядати контекстний клас і може пояснити причину порушення.

Візуалізація також працює для інших систем, оскільки її легко адаптувати за допомогою екстрактора функцій. На рисунку 3.8 показано частину системи.

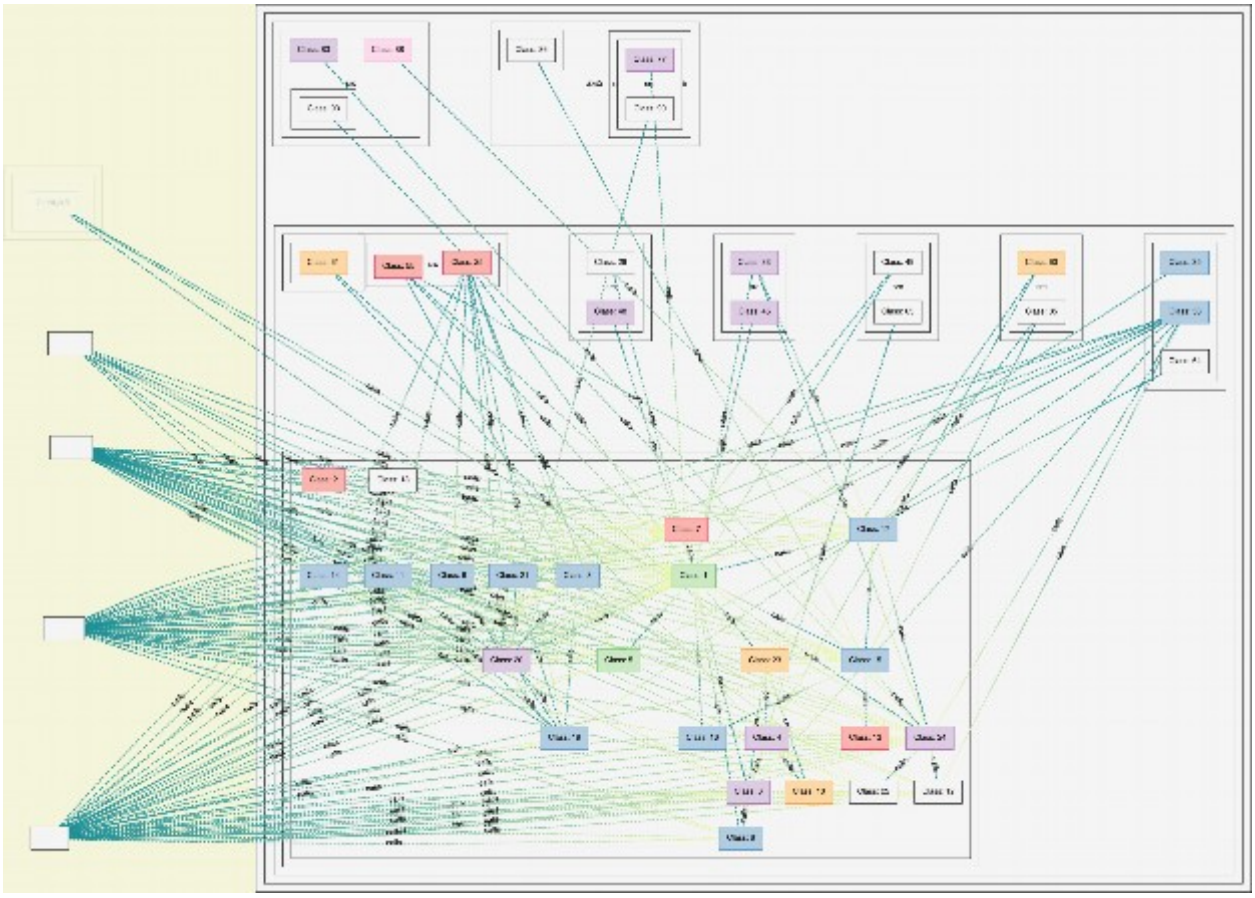

Рис. 3.8. Структура стереотипу зовнішньої ролі

П’ять класів класифікуються як Постачальник послуг (синій), один клас – Координатор (зелений) і один клас – Інтерфейс (жовтий). Незабарвлені класи класифікуються як порушення. Ця візуалізація показує структуру рольових стереотипних класів, зокрема те, як взаємодіють постачальники послуг. Класи, позначені як Постачальники послуг, не взаємодіють з іншими Постачальниками послуг, тоді як це відбувається всередині програмного забезпечення, як показано на рисунку 3.6. Крім того, постачальники послуг підключаються безпосередньо до класу координатора, а не спочатку через контролер. Інша частина, яка візуалізується на рисунку 3.8 це дві помилки (порушення). StringUtils.cpp — це клас, який порушував AR1, що означає, що для цього класу не існувало єдиного стереотипу основної ролі. Однак візуалізація показує багато викликів до класів за межами цього крупного плану, що наводить на думку про Координатора. Внутрішня частина класу може свідчити про інше, оскільки в його вихідному коді є ознаки постачальника послуг, що спричинило порушення.

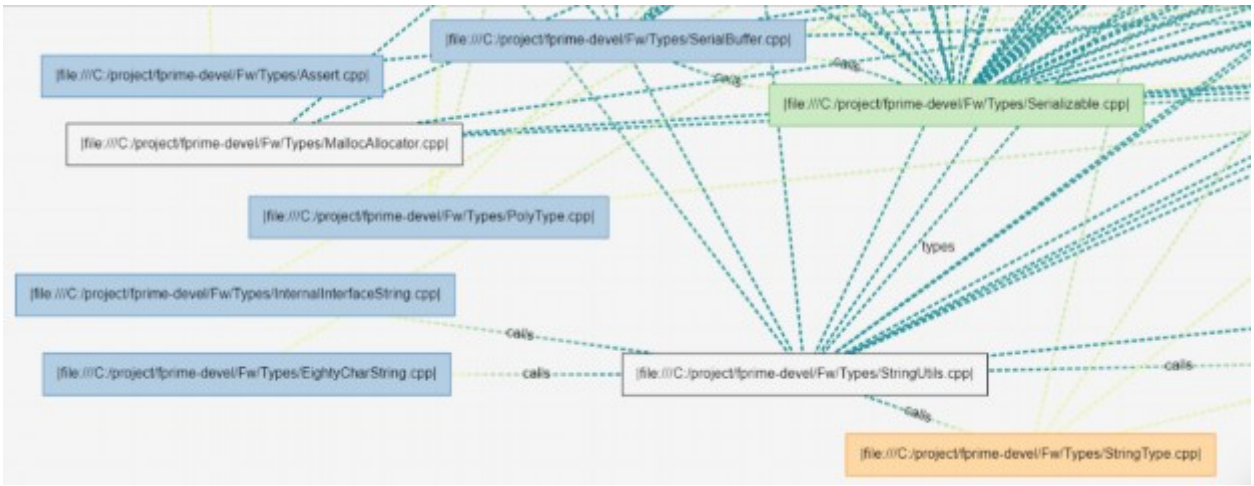

Рис. 3.8. Деталізація помилок (порушень) в класах

Маючи додатковий час і зосередившись на візуалізації, можна розглянути взаємодію з кожним стереотипом ролі класу для виявлення порушень або визначення нових архітектурних правил взаємодії між стереотипами ролі класу. Прикладом може бути те, що клас погано

спроектований, коли він є власником інформації та здійснює виклик до іншого власника інформації. Цей приклад є лише припущенням. Наразі потрібно більше літератури та досліджень для оцінки взаємодії між стереотипами класової ролі, де ця візуалізація може бути використана для пошуку закономірностей.

3.5.3. Виявлення помилок

Виявлення помилок виконується після того, як класифікатор зробив свою класифікацію. Спочатку перевіряється відсоток імовірності кожного рольового стереотипу. Якщо найвища ймовірність рольового стереотипу нижча за 90% або занадто близька до іншого рольового стереотипу, клас позначається як порушення AR1. Наприклад, якщо клас на 50% класифікується як контролер, а на 40% — як координатор, тоді це буде позначено як помилка. Це обмеження встановлено на 20%, як зазначено у визначенні AR1. Помилка типу AR2 відбувається після цієї перевірки класифікатором. Класифікатор був навчений виявляти класи, які занадто малі або занадто складні, щоб класифікувати їх як Нічого. Виявлення помилки потім перевіряє, чи правильно визначений клас чи ні, перевіряючи розмір класу. Якщо клас містить більше 100 рядків коду, його не слід позначати як Nothing, оскільки клас достатньо великий, щоб отримати значущі функції та призначити йому рольовий стереотип.

3.5.4. Причина появи помилок

Помилка позначається класифікатором, коли порушується правило. Розглянутий клас повертається з імовірністю класифікації кожного стереотипу. Дві найвищі ймовірності з шести виділяються розробником. Два виділені стереотипи вказують на структуру класу. Клас зі стереотипом Контролера та Координатора означає, що клас приймає кілька рішень за допомогою операторів if і case, які вказують на Контролера, у той же час клас може мати багато вихідних викликів із кількома параметрами та багатьма

включеннями, які вказують на Координатор. При такому підході загалом можлива 21 різна комбінація, коли відбувається помилка. Кожна комбінація сигналізує про іншу проблему, яка спричинила помилку. Однак це можна спростити, згадавши лише сім можливих стереотипів і визначивши причину хибної класифікації. Наступні сім причин змінено з огляду на те, чому класифікатор позначає їх як такі на основі ознак, і тому вони не завжди безпосередньо пов’язані з критеріями для кожного рольового стереотипу.

• **Controller.** Клас хибно класифікується як контролер, коли він приймає багато рішень, що призводить до високої цикломатичної складності. Зазвичай це трапляється, коли в класі є велика кількість циклів if і for. Клас також може мати деякі вихідні виклики з деякими параметрами та включеннями. Щоб вирішити цю помилкову класифікацію, розробник повинен зменшити навантаження на клас, знизивши його високу цикломатичну складність або перемістивши деякі з його гілок до іншого класу.

• **Coordinator**. Клас помилково класифікується як Coordinator, якщо він має багато залежностей з іншими класами з багатьма вихідними викликами та великою кількістю включень до інших класів. Виклики функцій всередині класу також можуть мати деякі параметри. Розробник може змінити клас, зменшивши кількість залежностей або створивши окремий клас обробника, який може бути координатором для цього класу.

• **Structurer.** Коли клас хибно класифікується як Structurer, тоді клас може містити певну форму геттерів і сеттерів, але зазвичай має багато визначених спеціальних типізованих об’єктів. Також можливо, що клас має певні приватні або захищені методи, які можуть спричинити помилкову класифікацію. Спосіб розв’язати цю хибну класифікацію — перевірити об’єкти всередині класу та їхні зв’язки з окремим класом, який міг би діяти як структуратор цих об’єктів.

• **Information Holder.** Клас помилково класифікується як Власник інформації, якщо клас має багато визначених змінних, але не обов’язково

абстрактні типізовані об’єкти. Визначені методи та функції зазвичай є геттерами та/або сеттерами. Розробнику потрібно переглянути змінні та методи класу та визначити, чи зберігається інформація в них. Якщо це так, то змінну або метод можна перемістити або змінити в інший клас.

• **Interfacer.** Помилкова класифікація інтерфейсу може бути тому, що клас тісно пов’язаний з компонентом інтерфейсу користувача. Клас має деякі або багато назв із «UI » в них. Також може статися, що клас є інтерфейсом для іншого об’єкта або інтерфейсом для сервера, і тоді ім’я класу може містити «проксі » або «ATL», що вказує на інтерфейс або сервер. Якщо клас не є інтерфейсом , тоді ім’я класу або методу може містити такі ключові слова, як «UI», «ATL» або «Proxy». Зазвичай розробник може легко виправити це, змінивши схему іменування класу та методів на щось більш підходяще.

• **Service Provider.** Клас хибно класифікується як Постачальник послуг через велику кількість методів і функцій. Ці методи та функції також можуть мати однакові назви, але визначені різними параметрами. Він також може містити вихідні виклики сеттерів і/або геттерів. Помилкова класифікація також може бути викликана багатьма публічними методами. Розробник може повторно оцінити клас, перевіривши його функціональність і перевірити, чи є методи причиною помилкової класифікації. Якщо це справді хибна класифікація, тоді може бути важко змінити клас, не впливаючи на функціональність усього класу. Хорошим підходом може бути переміщення методів в окремий клас, який має одну функціональність або однакову назву з різними параметрами.

• **Nothing.** Помилкова

класифікація класу може бути тому, що клас не має чітких ознак, які можуть вказувати на будь-який із шести рольових стереотипів. Можливо, клас занадто малий, неповний або просто занадто великий, щоб його зрозуміти машинний учень. Зазвичай викиди функцій також позначаються таким чином. Якщо клас невеликий або абстрактний з невеликою кількістю

визначень, це не повинно викликати негайного занепокоєння. Однак, якщо клас чітко визначений, розробник повинен оцінити, чи є клас викидом порівняно з іншими класами. У такому випадку розробнику може бути важко скерувати рішення.

Класифікатор позначатиме класи, які порушують AR1, зірочкою ” *”, а потім ймовірність стереотипу кожної ролі. Порушення AR2 позначається класифікатором знаком « -».

Висновки до розділу

Отже, в цьому розділі представлено імплементацію методів та моделей автоматизації процесів класифікації для виявлення помилок програмного забезпечення. Запропоновано інструмент для візуалізації помилок складається з трьох компонентів: екстрактора ознак, який приймає згенеровані моделі з READ як вхідні дані, класифікатора, який приймає набір даних і базові істини як вхідні дані та виводить класифікації, і засобу перевірки порушень, який виявляє порушення двох визначених архітектурних правил про класові рольові стереотипи. Інструмент також містить функцію візуалізації, яка в кольорі кодує стереотипи ролей і показує зв’язок між класами за допомогою діаграми типу UML.

ВИСНОВКИ

У магістерській роботі досліджено та проаналізовано методи автоматизації процесів класифікації для виявлення помилок програмного забезпечення. Основну увагу приділено автоматизованим методам, які дозволяють ідентифікувати шість стереотипів ролей класу та порушення архітектурних правил у програмному забезпеченні на мові C++. В ході дослідження використано інструмент RASCAL та його розширення ClaiR, які дозволяють здійснювати екстракцію властивостей та функцій з вихідного коду, а також проводити класифікацію для виявлення помилок архітектури.

Дослідження предметної області виявило важливість чіткого розуміння концепцій стереотипів класових ролей для підвищення ефективності виявлення помилок. Були визначені основні питання, що стосуються автоматизації цього процесу, а також структуровано підхід до архітектурної організації програмних систем.

Другий розділ присвячено дослідженню моделей та методів класифікації. Проведено аналіз особливостей метапрограмування, яке сприяє підвищенню ефективності обробки коду. Досліджено різні методи машинного навчання, такі як випадковий ліс, XGBoost та мультинаївний Байєс, які демонструють хороші результати в класифікації помилок.

Впровадження автоматизованої методики показало, що машинне навчання може ефективно використовуватися для ідентифікації стереотипів ролей у класах і виявлення порушень архітектурних правил. Проведено експерименти з екстракцією функцій та автоматичною класифікацією на основі даних з вихідного коду. Інструмент забезпечує можливість візуалізації результатів, що дозволяє краще розуміти стереотипи ролей класу та виявляти архітектурні помилки.

Загалом, результати роботи можуть зробити хороший внесок у розвиток автоматизації процесів виявлення помилок у програмному забезпеченні. Виявлено перспективні підходи до використання машинного

навчання для покращення точності ідентифікації стереотипів класових ролей та автоматизації перевірки архітектурних правил, що сприяє підвищенню якості програмних систем на етапі їх розробки та тестування.

Запропонована методологія використовує машинне навчання для аналізу програмного коду, що дозволяє автоматизувати процеси виявлення порушень архітектурних правил та підвищити ефективність пошуку помилок у складних програмних системах.

ПЕРЕЛІК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Rodin Aarssen, Jurgen J. Vinju, ruichen ing, Davy Landman, Jouke Stoel, and Omar Duhaiby. usethesource/clair: v0.9.2, August 2023. URL https://doi.org/10.5281/zenodo.8261944.8
2. O. Andriyevska, N. Dragan, B. Simoes, and J.I. Maletic. Evaluating uml class diagram layout based on architectural importance. In 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis, pages 1–6, 2005. doi: 10.1109/VISSOF.2005.1684296.
3. Len Bass, Paul Clements, and Rick Kazman. Software architecture in practice. Addison-Wesley Professional, 2003. 1
4. Bas Basten, Mark Hills, Paul Klint, Davy Landman, Ashim Shahi, Michael J Steindorfer, and Jurgen J Vinju. M3: A general model for code analytics in rascal. In 2015 IEEE 1st international workshop on software analytics (SWAN), pages 25–28. IEEE, 2015.
5. James R Cordy and Medha Shukla. Practical metaprogramming. Queen's University of Kingston. Department of Computing and Information Science, 1992.
6. Natalia Dragan, Michael L Collard, and Jonathan I Maletic. Reverse engineering method stereotypes. In 2006 22nd IEEE International Conference on Software Maintenance, pages 24–34. IEEE, 2006. 11
7. Natalia Dragan, Michael L. Collard, and Jonathan I. Maletic. Using method stereotype distribution as a signature descriptor for software systems. In 2009 IEEE International Conference on Software Maintenance, pages 567–570, 2009. doi: 10.1109/ICSM.2009.5306394. 11
8. Natalia Dragan, Michael Collard, and Jonathan Maletic. Automatic identification of class stereotypes. pages 1–10, 09 2010. doi: 10.1109/ICSM.2010.5609703. 11, 12, 13, 34
9. Issam El Naqa and Martin J Murphy. What is machine learning? Springer, 2015.

10. Jerome H Friedman. Greedy function approximation: a gradient boosting machine. Annals of statistics, pages 1189–1232, 2001. 10
11. Marcela Genero, Jos´e A. Cruz-Lemus, Danilo Caivano, Silvia Abrah˜ao, Emilio Insfran, and Jos´e Angel Cars´ı. Does the use of stereotypes improve the comprehension of uml sequence diagrams? In Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM ’08, page 300–302, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781595939715. doi: 10.1145/1414004.1414059. URL https://doi.org/10.1145/1414004.1414059.
12. Truong Ho-Quang, Arif Nurwidyantoro, Satrio Rukmono, Michel Chaudron, Fabian Fr¨oding, and Duy Ngoc. Role stereotypes in software designs and their evolution. Journal of Systems and Software, 189:111296, 03 2022. doi: 10.1016/j.jss.2022.111296.
13. Truong Ho-Quang, Arif Nurwidyantoro, Satrio Adi Rukmono, Michel R.V. Chaudron, Fabian Fr¨oding, and Duy Nguyen Ngoc. Role stereotypes in software designs and their evolution. Journal of Systems and Software, 189, July 2022. ISSN 0164-1212. doi: 10.1016/j.jss.2022. 111296. Publisher Copyright: © 2022 Elsevier Inc.
14. Ruichen Hu. An automated approach to check software architecture erosion. Master’s thesis, Eindhoven University of Technology, Eindhoven, 8 2023.
15. IEEE Architecture Working Group et al. Ieee recommended practice for architectural description for software-intensive systems. std 1471, IEEE, 2000.
16. Jaehyun Kim and Yangsun Lee. A study on abstract syntax tree for development of a javascript compiler. International Journal of Grid and Distributed Computing, 11(6):37–47, 2018.
17. Paul Klint, Tijs Van Der Storm, and Jurgen Vinju. Rascal: A domain specific language for source code analysis and manipulation. In 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, pages 168–177. IEEE, 2009.

18. Oliver Kramer and Oliver Kramer. Scikit-learn. Machine learning for evolution strategies, pages 45–53, 2016.
19. Yannis Lilis and Anthony Savidis. A survey of metaprogramming languages. ACM Comput. Surv., 52(6), oct 2019. ISSN 0360-0300. doi: 10.1145/3354584. URL https://doi.org/10.1145/3354584.
20. Lyu, M. R. Handbook on Software Reliability Engineering. McGraw-Hill, USA and IEEE Computer Society Press, Los Alamitos, California, USA, 1996.
21. Cory Maklin. Synthetic minority over-sampling technique (smote), 2022. URL https://medium.com/@corymaklin/synthetic-minority-over-sampling-technique-smote-7d419696b88c. ix,
22. NASA. A flight-proven, multi-platform, open-source flight software framework, 2023. URL https://github.com/nasa/fprime.
23. Arif Nurwidyantoro, Truong Ho-Quang, and Michel R. V. Chaudron. Automated classification of class role-stereotypes via machine learning. In Proceedings of the 23rd International Conference on Evaluation and Assessment in Software Engineering, EASE '19, page 79–88, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450371452. doi: 10.1145/3319008.3319016. URL https://doi.org/10.1145/3319008.3319016.
24. Arif Nurwidyantoro, Truong Ho-Quang, and Michel R. V. Chaudron. Automated classification of class role-stereotypes via machine learning. In Proceedings of the 23rd International Conference on Evaluation and Assessment in Software Engineering, EASE '19, page 79–88, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450371452. doi: 10.1145/3319008.3319016. URL https://doi.org/10.1145/3319008.3319016
25. Arthur V. Ratz. Multinomial naive bayes' for documents classification and nlp, 2021. URL https://towardsdatascience.com.

26. Filippo Ricca, Massimiliano Di Penta, Marco Torchiano, Paolo Tonella, and Mariano Ceccato. How developers' experience and ability influence web application comprehension tasks supported by uml stereotypes: A series of four experiments. IEEE Transactions on Software Engineering, 36(1):96–118, 2010. doi: 10.1109/TSE.2009.69.
27. Satrio Adi Rukmono and Michel R.V. Chaudron. Enabling analysis and reasoning on software systems through knowledge graph representation. In 2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR), pages 120–124, 2023. doi: 10.1109/MSR59073.2023.00029.
28. Tianyu Liu. Automatic code modernization with rascal. Master's thesis, Eindhoven University of Technology, Eindhoven, 2018.
29. Rebecca Wirfs-Brock, Alan McKean, Ivar Jacobson, and John Vlissides. Object Design: Roles, Responsibilities, and Collaborations. Pearson Education, 2002. ISBN 0201379430
30. R.J. Wirfs-Brock. Characterizing classes. IEEE Software, 23(2):9–11, 2006. doi: 10.1109/MS.2006.43.
31. Li, Z., Shan, Z., Zhu, W., Zhang, H. (2020). "Automated Bug Triage with Deep Learning" IEEE Transactions on Software Engineering.
32. Kim, S., & Ernst, M. D. (2007). "Which Warnings Should I Fix First?" ACM SIGSOFT Software Engineering Notes.
33. Malhotra, R., & Khanna, M. (2013). "Investigation of Machine Learning Algorithms for the Prediction of Software Fault Proneness" International Journal of Computer Applications.
34. Hall, T., Beecham, S., Bowes, D., Gray, D., & Counsell, S. (2012). "A Systematic Literature Review on Fault Prediction Performance in Software Engineering" IEEE Transactions on Software Engineering.
35. Lamkanfi, A., Demeyer, S., Serebrenik, A., & Ven, T. (2013). "Predicting the Severity of a Reported Bug" Proceedings of the IEEE Working Conference on Mining Software Repositories.

36. Menzies, T., Greenwald, J., & Frank, A. (2007). "Data Mining Static Code Attributes to Learn Defect Predictors" IEEE Transactions on Software Engineering.
37. Arora, A., Sharma, D. K., & Kour, D. (2018). "A Survey of Machine Learning Techniques for Bug Prediction" International Journal of Software Engineering and Knowledge Engineering.
38. Bird, C., Barr, E. T., Nash, A., & Nagappan, N. (2011). "Understanding Software Code Changes Using Similarity Clustering" Proceedings of the IEEE/ACM International Conference on Automated Software Engineering.
39. Guo, P. J., & Engler, D. (2011). "Using Automatic Bug Triaging to Improve Software Quality" Proceedings of the International Conference on Software Engineering (ICSE).
40. Le, T., & Zhang, Z. (2019). "Deep Learning-Based Bug Report Classification for Bug Triage Improvement" Journal of Systems and Software.
41. Bowes, D., Hall, T., & Christianson, B. (2017). "Software Defect Prediction: Do Different Classifiers Find the Same Defects?" Software Quality Journal.
42. Zhang, F., Zou, Y., & Hassan, A. E. (2014). "Cross-Project Defect Prediction Using a Model-Agnostic Approach" IEEE Transactions on Software Engineering.
43. Shivaji, S., Whitehead, E. J., Akella, R., & Kim, S. (2009). "Reducing Features to Improve Bug Prediction" Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering.
44. Chakraborty, P., & Choudhury, T. (2019). "Software Bug Prediction Using Ensemble Learning Techniques" Journal of Software: Evolution and Process.
45. Ni, X., Li, L., & Wang, H. (2019). "Bug Localization with Semantic and Structural Features" IEEE Transactions on Reliability.

46. Nam, J., Kim, S., Lo, D., & Pan, S. J. (2013). "Transfer Defect Learning" Proceedings of the International Conference on Software Engineering (ICSE).
47. Rahman, F., & Devanbu, P. (2011). "How, and Why, Process Metrics Are Better" Proceedings of the International Conference on Software Engineering.
48. Gondra, I. (2008). "Applying Machine Learning to Software Fault-Proneness Prediction" Journal of Systems and Software.
49. Just, R., Jalali, D., & Ernst, M. D. (2014). "Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs" Proceedings of the International Symposium on Software Testing and Analysis (ISSTA).
50. Thung, F., Lo, D., Jiang, L., & Zhang, Z. (2012). "Automatic Defect Categorization" Proceedings of the IEEE/ACM International Conference on Automated Software Engineering.
51. Xuan, J., & Monperrus, M. (2014). "Learning to Combine Multiple Ranking Metrics for Fault Localization" Proceedings of the International Conference on Software Maintenance and Evolution.
52. Ostrand, T. J., Weyuker, E. J., & Bell, R. M. (2005). "Predicting the Location and Number of Faults in Large Software Systems" IEEE Transactions on Software Engineering.
53. Zhong, H., Zhang, L., Wang, H., & Mei, H. (2015). "An Effective Bug Triage Approach by Incorporating Bug Tossing and Severity Information" Proceedings of the IEEE International Conference on Software Maintenance and Evolution.