БАКАЛАВРСЬКА РОБОТА

БР. ІП - 60.00.00.000 ПЗ

Група ІП-22-3

Бачкур Артур

2026

Івано-Франківський національний технічний університет нафти і газу

Факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Бачкур Артур Іванович

(прізвище, ім'я, по батькові)

УДК 004.942
(індекс)

БАКАЛАВРСЬКА РОБОТА

Методи зменшення дублювання коду під час розробки

Інженерія програмного забезпечення

(назва освітньої програми)

121 - Інженерія програмного забезпечення

(шифр і назва спеціальності)

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

Здобувач освітнього ступеня _____Бачкур А.І._____
(підпис, ініціали та прізвище здобувача)

Науковий керівник _____Храбатин Роман Ігорович, к.ф.м.н, доцент_____
(підпис, прізвище, ім'я, по батькові, науковий ступінь, вчене звання керівника)

Допущено до захисту
Завідувач кафедри

_доц._____Бандура В.В.__
(посада) (підпис) (дата) (ініціали та прізвище)

Івано-Франківськ – 2026

Івано-Франківський національний технічний університет нафти і газу

Інститут, факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Ступінь вищої освіти бакалавр

Спеціальність 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Зав. кафедрою **ІПЗ**
доцент. В.В. Бандура
“ ____ ” _____2026 р.

ЗАВДАННЯ
НА БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТОВІ

Бачкур Артур Іванович

(прізвище, ім’я, по-батькові)

1. Тема проекту (роботи) "Методи зменшення дублювання коду під час розробки ”
керівник проекту (роботи) Храбатин Роман Ігорович, к.ф.м.н, доцент
затверджені наказом вищого навчального закладу від “ 21_ ” _квітня 2026 р. № 179/7

2. Строк подання студентом проекту (роботи) 10 червня 2026 р.

3. Вихідні дані до проекту (роботи) Результати і матеріали отримані під час проходження переддипломної практики

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

1. Аналіз вимог до програмного забезпечення
2. Аналіз вимог і постановка задачі
3. Проектування системи
4. Реалізація проекту
5. Тестування та впровадження

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

1. Блок-схема процесу рефакторингу Janus (рис.2.1, ст.27)
2. Блок-схема етапу попередньої обробки (рис.2.3, ст.30)
3. Діаграма кроків рефекторингу (рис.2.8, ст.39)
4. Діаграма пакета ре факторингу (рис.3.5, ст.53)
5. Послідовність кроків, пов'язаних з перевірками (рис.3.7, ст.55)

6. Консультанти розділів проекту (роботи)

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

2. Дата видачі завдання **2026 р.**

Керівник _____
(підпис)

Завдання прийняв до виконання_____
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту (роботи)	Строк виконання етапів проекту	Примітка
1	Визначення та обгрунтування теми роботи	17.01.2026	виконано
2	Огляд існуючих концепцій, рішень та сервісів в даній області	21.02.2026	виконано
3	Побудова моделі або алгоритму власного рішення	01.03.2026	виконано
4	Документування реалізації власного оригінального рішення вибраними засобами	21.04.2026	виконано
5	Оформлення пояснювальної записки бакалаврської роботи	02.05.2026	виконано

Студент _____Бачкур А.І.

Керівник роботи _____Храбатин Р.І.
(підпис)

АНОТАЦІЯ

Бакалаврська робота містить 77 сторінок, 20 рисунків, 5 таблиць, список використаних джерел із 40 найменувань.

Метою роботи є дослідження методів зменшення дублювання коду під час розробки програмного забезпечення та аналіз підходів до підвищення якості, підтримуваності та ефективності програмних систем.

Об'єкт дослідження: процеси розробки програмного забезпечення з використанням сучасних підходів до організації коду.

Предмет дослідження: методи виявлення клонів коду, підходи до рефакторингу та інструменти автоматизації зменшення дублювання коду.

Результати дослідження: проведено аналіз причин виникнення дублювання коду, досліджено основні типи клонів коду, розглянуто сучасні методи їх виявлення та усунення, а також оцінено ефективність застосування рефакторингу та автоматизованих інструментів.

У першому розділі розглянуто теоретичні основи дублювання коду, його вплив на якість програмного забезпечення, а також методи виявлення та усунення клонів коду.

У другому розділі досліджено методи та інструменти зменшення дублювання коду, включаючи автоматизовані підходи до аналізу вихідного коду, попередню обробку та практики впровадження відповідних рішень.

У третьому розділі розглянуто реалізацію методів виявлення та усунення дублювання коду, проведено оцінку їх ефективності та валідацію отриманих результатів.

Висновок: застосування сучасних методів виявлення клонів коду та рефакторингу дозволяє значно зменшити дублювання, підвищити якість програмного забезпечення, спростити його підтримку та знизити витрати.

КЛЮЧОВІ СЛОВА: ДУБЛЮВАННЯ КОДУ, КЛОНИ КОДУ, РЕФАКТОРИНГ, ЯКІСТЬ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, АВТОМАТИЗАЦІЯ, АНАЛІЗ КОДУ, ОПТИМІЗАЦІЯ.

ANNOTATION

The bachelor's thesis contains 77 pages, 20 figures, 5 tables, a list of sources used with 40 names.

The aim of the work is to study methods for reducing code duplication during software development and to analyze approaches to improving the quality, maintainability, and efficiency of software systems.

Research object: software development processes using modern approaches to code organization.

Research subject: methods for detecting code clones, refactoring techniques, and tools for automating code duplication reduction.

Research results: an analysis of the causes of code duplication was conducted, main types of code clones were studied, modern detection and elimination methods were reviewed, and the effectiveness of refactoring and automated tools was evaluated.

The first section describes the theoretical foundations of code duplication, its impact on software quality, and methods for detecting and eliminating code clones.

The second section analyzes methods and tools for reducing code duplication, including automated source code analysis, preprocessing techniques, and practical implementation approaches.

The third section covers the implementation of code duplication detection and elimination methods, as well as the evaluation and validation of their effectiveness.

Conclusion: the application of modern code clone detection methods and refactoring techniques significantly reduces duplication, improves software quality, simplifies maintenance, and reduces development costs.

KEY WORDS: CODE DUPLICATION, CODE CLONES, REFACTORING, SOFTWARE QUALITY, AUTOMATION, CODE ANALYSIS, OPTIMIZATION.

ЗМІСТ

ВСТУП

У сучасному світі програмне забезпечення є ключовим елементом функціонування інформаційних систем, веб-додатків та цифрових сервісів. Зростання складності програмних продуктів, збільшення обсягів коду та необхідність швидкої розробки нових функціональних можливостей створюють нові виклики для розробників. Однією з найбільш поширених проблем у процесі розробки є дублювання коду, яке негативно впливає на якість програмного забезпечення, ускладнює його підтримку та знижує ефективність командної роботи.

Актуальність теми зумовлена тим, що дублювання коду призводить до зростання технічного боргу, підвищує ризик виникнення помилок та ускладнює внесення змін у програмний продукт. У сучасних умовах розробки, де широко застосовуються гнучкі методології та швидкі цикли оновлення, особливо важливо забезпечити чистоту, структурованість і повторне використання коду. Існуючі підходи до розробки не завжди ефективно вирішують проблему дублювання, що потребує дослідження сучасних методів його виявлення та усунення.

Метою роботи є дослідження методів зменшення дублювання коду під час розробки програмного забезпечення, а також аналіз підходів до підвищення якості, підтримуваності та ефективності програмних систем.

Завданнями дослідження є аналіз причин виникнення дублювання коду та його впливу на якість програмного забезпечення, дослідження методів виявлення клонів коду, вивчення підходів до рефакторингу як засобу усунення дублювання, аналіз сучасних інструментів автоматизації обробки вихідного коду, реалізація підходів до зменшення дублювання та оцінка їх ефективності.

Об’єктом дослідження є процеси розробки програмного забезпечення, що включають створення, модифікацію та підтримку програмного коду.

Предметом дослідження є методи та інструменти виявлення і усунення дублювання коду, зокрема аналіз клонів коду, рефакторинг та автоматизовані підходи до оптимізації структури програмного забезпечення.

Методи дослідження включають аналіз наукових джерел для вивчення теоретичних основ дублювання коду, порівняльний аналіз існуючих методів та інструментів, моделювання процесів обробки коду, експериментальний підхід для реалізації та тестування запропонованих рішень, а також оцінку ефективності отриманих результатів.

У роботі розглядаються підходи до виявлення та усунення клонів коду, методи рефакторингу, а також сучасні інструменти автоматизації аналізу програмного коду. Особлива увага приділяється підвищенню якості програмного забезпечення, зменшенню технічного боргу та оптимізації процесу розробки. Очікується, що застосування запропонованих методів дозволить покращити структуру коду, знизити витрати на його підтримку та підвищити ефективність розробки.

Бакалаврська робота містить 74 сторінок, 20 рисунків, 5 таблиць, 3 розділи, висновки та список використаних джерел із 40 найменувань.

РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ ЗМЕНШЕННЯ ДУБЛЮВАННЯ КОДУ

1.1 Поняття дублювання коду та його вплив на якість програмного забезпечення

Дизайн – це складно. Дизайн програмного забезпечення багаторазового використання є особливо складним. Програмне забезпечення багаторазового використання зазвичай є результатом багатьох ітерацій дизайну. Деякі з цих ітерацій відбуваються після того, як програмне забезпечення було повторно використано, і результуючі зміни впливають не лише на дизайн програмного забезпечення багаторазового використання, але й на дизайн іншого програмного забезпечення, яке його використовує.

Під час написання коду для великих проектів існує ймовірність того, що існуючі фрагменти коду будуть переписуватися знову і знову. Це може робитися несвідомо, але також шляхом копіювання та вставки, що, на жаль, є поширеною практикою [1]. У деяких тематичних дослідженнях комерційного програмного забезпечення повідомлялося, що 5% коду були клонами (1) у проекті з відкритим кодом, ця кількість, як повідомлялося, була вищою: 12% (2). Новіші дослідження вказують на подібні цифри для широкого спектру типів програмного забезпечення з відкритим кодом (3). Загальновідомо, що дублювання коду швидко знижує зручність його підтримки. Бажано видалити ці клони, об'єднавши їх в один екземпляр коду.

Щоб мати змогу видалити клони, їх спочатку потрібно знайти. Під час роботи з великою базою коду, що містить сотні тисяч рядків коду або більше, яка містить багато дублікатів коду, досить важко знайти весь цей дублікат. Хоча існують інструменти виявлення дублікатів коду, вони здебільшого базуються на текстовому (2) або токеновому (1) (4) аналізі коду. Текстовий аналіз знайде лише код, який текстово абсолютно ідентичний, якщо, наприклад, хтось трохи змінить ім'я змінної, подібність коду більше не розпізнається. Також немає гарантії, що

два фрагменти коду, які є текстово однаковими, є семантично однаковими. Аналіз на основі токенів знайде більше дублікатів коду, оскільки він порівнює типи токенів, наприклад, ідентифікатор відповідатиме ідентифікатору [2]. Це також означає, що буде знайдено більше клонів коду, які семантично не однакові.

Навіть якщо припустити наявність сучасних детекторів дублікатів коду, які можуть виявляти семантично ідентичні, але потенційно синтаксично різні дублікати, усунення дублікатів зазвичай виконується вручну. Цей процес включає трудомістку діяльність з перевірки екземплярів дубліката одного за одним та редагування коду в різних місцях, щоб замінити всі дублікати одним екземпляром. Отже, другою проблемою створення підтримуваного коду після виявлення дублікатів є пропонування (або виконання) автоматичного рефакторингу, тобто заміна всіх екземплярів дубліката (також званого клоном) одним екземпляром. Переваги рефакторованого коду численні: база коду, яку потрібно підтримувати, стає меншою; загальна структура може покращитися, витрати на тестування та розуміння зменшуються, оскільки тепер потрібно тестувати або розуміти лише один екземпляр коду, а не всі його клони.

Метою цього дипломного проєкту є створення інструменту, який знаходить фрагменти коду, що є семантично однаковими або схожими, незалежно від того, чи є вони текстово однаковими чи ні, та, де це доречно, пропонує та виконує операції рефакторингу коду для усунення цього дублювання [3]. Ці операції рефакторингу коду будуть здійснюватися у формі об'єднання методів вилучення таким чином, щоб виклик результуючого методу з правильними аргументами міг замінити обидва семантично схожі фрагменти коду, не змінюючи нічого в зовнішній поведінці коду. Враховуючи, що цей проєкт виконується в контексті діяльності з розробки програмного забезпечення в IT-компанії, що спеціалізується на розробці на Visual Basic, ми обмежимося нашою сферою виявлення клонів та рефакторингу кодом Visual Basic.

Існують інструменти, які пропонують видалення клонів коду з вихідного коду Java за допомогою так званого рефакторингу "Метод вилучення" або "Метод підтягування" (5) з використанням лише синтаксичної інформації. Однак

синтаксичної інформації недостатньо для фактичного виконання рефакторингу коду. Отже, нам потрібен механізм для вилучення як синтаксичної, так і семантичної інформації з вихідного коду Visual Basic, щоб задовольнити нашу вимогу автоматичного видалення клонів.

Оскільки потрібно знайти клони коду, які можна автоматично видалити за допомогою методу вилучення без зміни зовнішньої поведінки коду, клони коду неможливо знайти, використовуючи лише текстове або токен-виявлення дублікатів. Необхідно точно знати, які змінні та члени використовуються в коді та що з ними робиться. Це, знову ж таки, можна зробити лише шляхом глибокого семантичного аналізу коду.

Крім того, щоб програмна система залишалася корисною та використовувалася, її необхідно постійно оновлювати та підтримувати [4]. Отже, технічне обслуговування програмного забезпечення передбачає впровадження нових необхідних функцій та усунення проблем із вже наявними функціональними можливостями, а також вдосконалення низки інших аспектів, які не обов’язково пов’язані з наданою функціональністю.

Фактично, супровід програмного забезпечення передбачає покращення таких аспектів якості програмного забезпечення, як безпека, продуктивність, надійність, зручність використання та багато інших [4]. Одним із аспектів, який слід враховувати, є супровідність, що означає, що будь-які зміни, внесені до коду, не повинні перешкоджати подальшим модифікаціям.

Коли питання підтримки програмного забезпечення належним чином не враховується під час постійних змін, що вносяться з часом до дизайну (або, що ще гірше, до архітектури програмного забезпечення), якість програмної системи буде поступово знижуватися. Симптоми деградації системи можуть проявлятися на різних рівнях: на рівні коду як запахи коду [23], на архітектурному рівні як запахи архітектури [22] або на рівні дизайну як запахи дизайну [26].

Запах дизайну зазвичай вказує на проблему, пов'язану з основними принципами об'єктно-орієнтованого програмування, таку як неправильне використання успадкування або поліморфізму. З іншого боку, архітектурні запахи

більше пов'язані зі способом, яким кожна структура коду (яка може бути класом або пакетом у Java) взаємодіє з усіма іншими; наприклад, циклічна залежність передбачає відсутність розділення завдань та виділяє набір структур коду, які не можна використовувати незалежно.

Однак жодна з вищезгаданих проблем насправді не стає проблемою, доки вона не має прямого впливу на спосіб написання коду, і коли це відбувається, зазвичай на реалізований код впливає один або кілька *«запахів» коду.*

Запах коду визначається як поверхнева ознака, яка зазвичай відповідає глибшій проблемі в системі [23]. Прикладом може бути фрагмент коду зі значною кількістю *if-else операторів.* оператори, що може означати, що успадкування та поліморфізм не використовуються належним чином; такий запах коду зазвичай називають оператором перемикання [23]. Додатковим прикладом може бути пакет, у якому невелика функціональна зміна призводить до великої кількості змін у різних класах, спричинених циклічною природою залежності між ними; цей запах коду зазвичай називають у літературі «дробовиковою хірургією» [23].

Основна увага в цій роботі приділяється іншому запаху коду, який називається дублікат коду [23], що вважається однією з найсерйозніших і найпоширеніших проблем, що може суттєво вплинути на якість програмної системи [23, 2]. Він часто виникає шляхом копіювання та вставки фрагментів коду в програмній системі з незначними модифікаціями або адаптаціями. Цей метод повторного використання коду називається *клонуванням коду* (або дублюванням коду), а вставлений фрагмент коду називається *клоном* (або дублікатом).

Клонування коду вважається шкідливим з двох причин: *(i)* численні, можливо непотрібні, дублікати коду збільшують вартість обслуговування та *(ii)* невідповідні зміни в клонованому коді можуть призвести до дефектів і, отже, до неправильної поведінки програми.

Тому особливо важливо виявляти клони шляхом постійного моніторингу вихідного коду під час його еволюції, розуміти причини їх появи та видаляти їх, коли це можливо та зручно.

Фактично, важливо підкреслити, що видалення клонів може також призвести до появи нових залежностей, які можуть погіршити архітектуру або дизайн програмного забезпечення [5]. Тому серед дослідників та практиків існує консенсус щодо того, що не всі клони коду слід рефакторувати.

Метою цієї роботи є розробка інструменту, який зможе реалізувати цю ідею, дозволяючи користувачам керувати виявленням клонів коду на основі їхніх потреб та автоматизувати рефакторинг клонів коду, максимально впливаючи на архітектуру. Цей інструмент названо *Янусом,* на честь давньоримського бога дуальності, переходів, початків та закінчень.

Зокрема, автоматизація рефакторингу включає не лише впровадження методів рефакторингу, але й автоматизацію всього процесу рефакторингу , включаючи виконання тестів, корисне для забезпечення збереження функціональності, та версійність змін, корисну для відстеження та управління кроками рефакторингу.

Усі ці функціональні можливості розглядаються в рамках цієї роботи, оскільки основна ідея полягає в тому, щоб не розробляти нічого з нуля, а повторно використовувати вже реалізовану бібліотеку, яка частково або повністю забезпечує необхідну функціональність [6]. Тому основні зусилля пов'язані з адаптацією функціональних можливостей, які вже надаються існуючими бібліотеками, за допомогою яких розробляється змістовний та ефективний робочий процес.

Класифікація клонів коду

Клони коду вводяться через дуже поширену практику копіювання фрагментів коду, їх вставки та застосування незначних або суттєвих модифікацій чи адаптацій на основі нового контексту.

Цей процес дає кілька можливих типів клонів коду, які можна розділити на дві основні групи залежно від виду подібності, а саме: *текстову подібність* і *функціональна схожість* . Крім того, що стосується текстової схожості, можна виділити кілька можливих відповідних модифікацій. Клони коду зазвичай організовані за чотирма типами, представленими в цьому розділі; приклад для

кожного типу наведено на рисунку 1.1

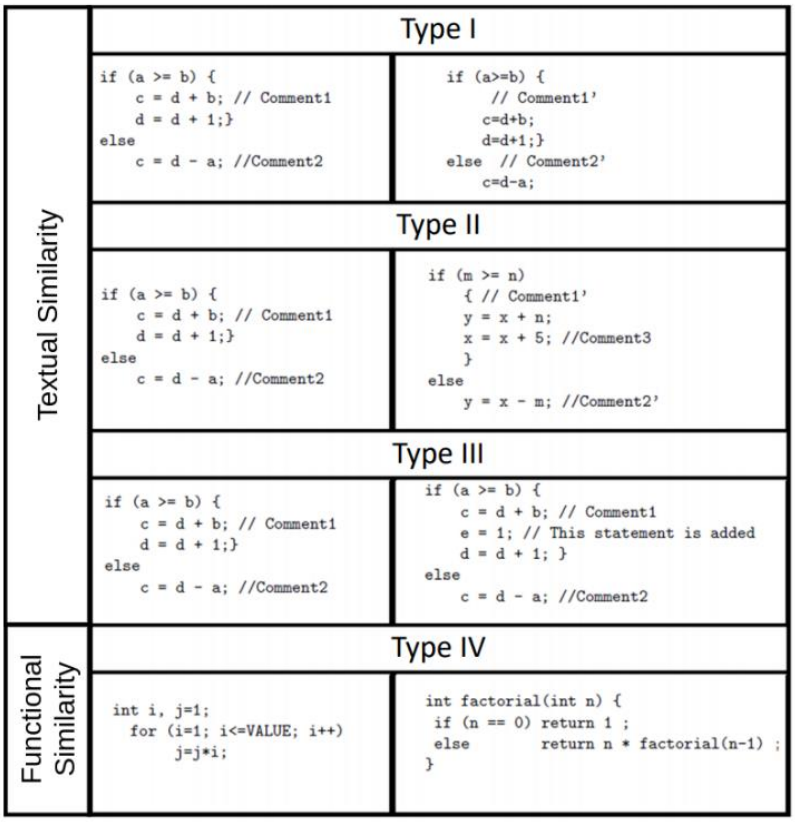

Рисунок 1.1 - Типи клонів коду

Текстова схожість

Два фрагменти коду можна вважати клонованими на основі подібності їхнього програмного тексту. Можна виділити такі типи текстових клонів [9, 10]:

- Тип I: *Ідентичні фрагменти коду* за винятком варіацій у пробілах (включаючи варіації у макеті) та коментарів.
- Тип II: *Структурно/синтаксично ідентичні фрагменти* за винятком варіацій в ідентифікаторах, літералах, типах, макеті та коментарях.
- Тип III: *Скопійовані фрагменти з подальшими модифікаціями* . Оператори можна змінювати, додавати або видаляти, а також змінювати ідентифікатори, літерали, типи, макет та коментарі.

Функціональна подібність

Два фрагменти коду можна вважати клонованими на основі подібності

їхньої функціональності, не будучи текстово схожими, тобто вони мають подібні попередні та наступні умови. Це

зазвичай називають *семантичними клонами* та позначаються як клони IV типу.

• Тип IV: Два або більше фрагментів коду, які виконують одне й те саме обчислення, але реалізовані за допомогою різних синтаксичних варіантів.

Пошуково-орієнтована розробка програмного забезпечення

Програмна інженерія (ПІ) часто розглядає проблеми, що передбачають пошук відповідного балансу між конкуруючими та потенційно суперечливими цілями. Часто існує вражаюче великий набір варіантів, і знайти хороші рішення може бути важко [7]. У ПІТ термін «пошук» використовується для позначення метаевристичних методів оптимізації на основі пошуку, які використовуються. У контексті ПІТ оптимальні або майже оптимальні рішення шукаються в просторі пошуку рішень-кандидатів, керуючись функцією придатності, яка розрізняє кращі та гірші рішення [31]. Для цієї мети може бути використаний і використовувався широкий спектр різних методів оптимізації та пошуку, наприклад, імітація відпалу, генетичні алгоритми, генетичне програмування та сходження на пагорби [31].

SBSE для обслуговування програмного забезпечення

Більшість робіт із застосування SBSE, пов'язаних з обслуговуванням програмного забезпечення [31], зосереджені навколо трьох основних тем: *(i)* модуляризація програмного забезпечення на основі пошуку, *(ii)* підходи до автоматизації рефакторингу на основі пошуку [17] та *(iii)* підходи до створення оптимальної послідовності рефакторингів на основі пошуку. Однак, варто зазначити, що дослідження в рамках SBSE також розглядають інші аспекти, пов'язані з обслуговуванням програмного забезпечення [31], такі як прогнозування якості та міграція застарілих систем.

З огляду на значну кількість тем, що стосуються цієї роботи, її презентація поділена на три різні категорії. Спочатку представлено існуючі підходи до виявлення клонів коду. Після цього наведено короткий опис поширених методів

рефакторингу, що використовуються для боротьби з запахом дублікатів коду. Нарешті, представлено короткий огляд використання методів SBSE для виявлення клонів коду та рефакторингу клонів коду.

1.2 Методи виявлення клонів коду

У літературі існує консенсус щодо важливості виявлення та усунення дублікатів коду. Фактично, ручне виявлення клонів неможливе для великих систем, тому необхідна автоматична підтримка.

Автоматизація виявлення клонів коду є активною галуззю досліджень і можна кластеризувати різні підходи на основі того, як здійснюється виявлення. Щодо *текстових клонів,* тобто клонів коду типів I, II та III, існують три основні стратегії, що використовуються для виявлення клонів: текстова, абстрактне синтаксичне дерево та на основі токенів [8]. Крім того, нещодавно було запропоновано деякі нові підходи, що використовують алгоритми машинного навчання. Нарешті, було запропоновано кілька методологій для виявлення семантичних клонів (тобто клонів коду типу IV). У цьому розділі всі ці методи коротко представлені.

Текстове порівняння

Базується на текстовому порівнянні цілих рядків коду. Для підвищення продуктивності рядки розбиваються за допомогою хеш-функції для рядків, і порівнюються лише рядки в одному розділі. Результат візуалізується у вигляді точкової діаграми, де кожна точка вказує на пару клонованих рядків. Клони можна візуально знайти як певні шаблони на цих точкових діаграмах. Послідовні рядки можна автоматично звести до більших клонованих послідовностей як неперервні діагоналі або зміщені діагоналі на точковій діаграмі.

Використовується ефективне зіставлення рядків, розроблене на основі відбитків пальців, для виявлення кластерів файлів зі значною кількістю спільного тексту.

Порівнюють певні фрагменти тексту, а саме ідентифікатори, використов

уючи латентне семантичне індексування, яке є методом, що належить до галузі інформаційного пошуку. Їхня ідея полягає в тому, щоб ідентифікувати фрагменти, в яких зустрічаються подібні імена, як потенційні клони.

Порівняння токенів

Методика також базується на порівнянні на основі рядків. Замість порівняння рядків, послідовності токенів рядків ефективно порівнюються за допомогою суфіксного дерева. Спочатку кожна послідовність токенів для цілого рядка підсумовується так званим функтором, який абстрагує конкретні значення ідентифікаторів та літералів [3]. Функтор унікально характеризує цю послідовність токенів. Функтори присвоєння можна розглядати як ідеальну хеш-функцію. Конкретні значення ідентифікаторів та літералів фіксуються як параметри цього функтора [9]. Два рядки вважаються клонами, якщо вони відповідають своїм функторам та кодуванню параметрів.

Збільшується повнота зайвих різних, але еквівалентних послідовностей шляхом нормалізації послідовностей токенів. Оскільки синтаксис не враховується, ідентифіковані клони можуть перетинати різні синтаксичні одиниці, які неможливо замінити за допомогою функціональної абстракції. Як на етапі попередньої обробки [28], так і на етапі постобробки [34] можна знайти клони, які повністю потрапляють у синтаксичні блоки, якщо відомі роздільники блоків. Попередня та постобробка вимагають певної синтаксичної інформації, зібраної або шляхом підрахунку токенів, що відкривають та закривають синтаксичні області видимості, або граматик островів, або повноцінного синтаксичного аналізу [28].

Порівняння абстрактних синтаксичних дерев (AST)

Розбивають піддерева абстрактного синтаксичного дерева програми на основі хеш-функції, а потім порівнюють піддерева в тому ж розділі за допомогою зіставлення дерев (з урахуванням деяких розбіжностей).

Подібний підхід раніше запропонували, використовуючи динамічне програмування для пошуку відмінностей між двома версіями одного й того ж файлу. Суфіксні дерева (центральні для методів на основі токенів) також можна використовувати для виявлення послідовностей ідентичних вузлів AST.

У підході вузли AST серіалізуються в попередньому порядку обходу, для цих серіалізованих вузлів AST створюється суфіксне дерево, а отримані максимально довгі послідовності вузлів AST потім обрізаються відповідно до їх синтаксичної області, так що залишаються лише синтаксично замкнені послідовності.

Методи на основі машинного навчання

Пропонується методи виявлення на основі навчання, де все для представлення термінів та фрагментів у вихідному коді видобується з репозиторію [10]. Їхній аналіз коду підтримує фреймворк, який спирається на глибоке навчання, для автоматичного зв'язування шаблонів, виявлених на лексичному рівні , з шаблонами, виявленими на синтаксичному рівні.

Пропонується модель класифікації, яка застосовує машинне навчання до суджень кожного окремого користувача щодо клонів коду.

Пропонується підхід, у якому набори ознак генеруються після розбору заданої програми на C на наявність фрагментів коду та подальшого зіставлення їхньої схожості. На основі наборів ознак класифікація алгоритмів виконується за допомогою методу опорних векторів (SVM) як інструменту машинного навчання.

Представлено CCLEARNER, перший підхід до виявлення клонів, що базується виключно на токенах та використовує глибоке навчання [11]. CCLEARNER витягує токени з відомих клонів коду на рівні методів та неклонів для навчання класифікатора, а потім використовує класифікатор для виявлення клонів у заданій кодовій базі.

Розглядають виявлення клонів коду як проблему рекомендацій. Відповідно до цієї ідеї вони представляють CREC, підхід, заснований на навчанні, який рекомендує клони шляхом вилучення ознак із поточного стану та минулої історії програмних проектів.

Виявлення семантичних клонів

Витягується ознаки з абстрактних синтаксичних дерев (AST) та графів залежностей програм (PDG) і представляють пару фрагментів коду як вектор.

Потім витягнуті ознаки були використані для навчання моделі у виявленні синтаксичних та семантичних клонів за допомогою великої кількості алгоритмів.

Визначається функціональну подібність з точки зору поведінки вхідних даних та загального виходу. Вхідні дані визначають два фактори: значення параметрів, що передаються методу, та стан купи під час виклику методу. Загальна поведінка виходу визначається як з точки зору повернених значень методу, так і його ефектів. Ефекти методу – це їхні можливі постійні зміни в купі, включаючи мутації статичних полів.

Пропонується наближений алгоритм для виявлення семантичних клонів, який масштабується до мільйонів рядків коду, що базується на редукції навмисно вибраних підграфів PDG до абстрактних лісів синтаксичних дерев.

Зрештою, пропонується алгоритм, який знаходить клони семантичного коду на основі дерев розбору та формальних граматик.

1.3 Рефакторинг як засіб усунення дублювання коду

Однією з головних цілей виявлення клонів є їх видалення з програмної системи за допомогою автоматичного або ручного рефакторингу. За допомогою рефакторингу клонів ми можемо зменшити складність, зменшити потенційні джерела помилок, що виникають через дублювання коду, та підвищити зрозумілість програмних систем. У цьому підрозділі описано найпоширеніші методи рефакторингу [23], спрямовані на рефакторинг клонів.

Метод вилучення, в якому клонований код замінюється викликом функції новоствореного методу, що містить спільний код фрагментів клонів.

Метод вилучення та підтягування, за допомогою якого клонований код, визначений у дочірніх класах, замінюється викликом функції новоствореного методу, що містить спільний код, і цей метод підтягується до їхнього найнижчого спільного батьківського класу [32, 33].

Метод вилучення утиліти. Застосовується, коли клони розташовані в методах непов'язаних класів і не мають доступу до жодних змінних екземпляра чи

методів. У цьому випадку дубльований код вилучається в статичний метод і поміщається в новостворений утилітарний клас [19].

Вилучення суперкласу та вилучення класу, в яких дубльований код, спільний для двох або більше класів, переміщується до новоствореного класу. Ці методи підходять, коли дубльований код знаходиться у двох різних класах, які не входять до однієї ієрархії, і обсяг спільного коду достатній для втілення відповідальності [23, 1].

Метод шаблону форми та заміна коду типу за допомогою стратегії. Балазінська та ін. реалізують автоматичний процес рефакторингу, який: *(i)* факторизує спільні частини клонів, *(ii)* параметризує їхні відмінності та *(iii)* роз'єднує їх конкуренцію, створюючи налаштовуваний та повторно використовуваний компонент [6, 5, 4]. Цей компонент матиме структуру одного з двох відомих шаблонів проектування: методу шаблону або стратегії [25].

Алгоритм заміщення. Використовується для рефакторингу семантичних клонів шляхом вибору одного з реалізованих алгоритмів [23] та заміни інших викликами функцій.

SBSE для виявлення дублікатів коду та рефакторингу

Наскільки нам відомо, лише кілька робіт у літературі з SBSE розглядають проблему виявлення дублікатів коду та рефакторингу.

Виявлення: Пропонується метод пошуку на основі еволюційного алгоритму (EA) для ідентифікації клонів у вихідному коді. Їхньою метою було знайти набір груп клонів за допомогою пошуку, натхненного EA, з динамічним розміром популяції [12]. Кожна особина представляє кандидатну групу клонів, і таким чином, якість рішення визначається на рівні популяції (тобто мінімізуючи кількість особин, одночасно максимізуючи подібність клонів, представлених у кожній особині).

Рефакторинг: описують проблему планування рефакторингу клонів коду як проблему обмеженого рюкзака за допомогою наступної цілі оптимізації.

Нехай S - програмна система, а F - набір з n проблемних функцій в S (наприклад, набір дубльованих функцій). Кожна функція в F описується набором

метрик, що відображають властивості коду (наприклад, розмір, цикломатична складність, зв'язність, зв'язність) та рівнем пріоритету. Чим вищий пріоритет, тим вища ймовірність того, що функція буде обрана та фактично рефакторована. Мета полягає в тому, щоб знайти найціннішу підмножину функцій у *F,* тобто підмножину, яка, якщо її рефакторувати, гарантує максимальне покращення якості та споживає мінімальну кількість ресурсів, тобто орієнтовні зусилля, необхідні для рефакторингу певного клону коду. Тому їхньою метою було максимізувати покращення якості за заданий період часу, при цьому зусилля не перевищують максимально доступну величину, встановлену користувачем.

Рефакторинг коду

Небагато написано про автоматичний рефакторинг клонів коду, і практично немає записів про те, як такі методи реалізуються реальними інструментами. Усі знайдені підходи, такі як ті, що обговорювалися вище, абстрактні імена змінних та літеральні значення, і дуже мало підходів (якщо такі взагалі є) виконують глибокий семантичний аналіз. Отже, такі підходи менш придатні (якщо взагалі придатні) для виконання автоматичного рефакторингу клонів коду з причин, обговорених раніше в цьому розділі.

Метод вилучення

Фаулер представив у своїй книзі (11) детальний список рефакторингів. У цій роботі основна увага приділяється «Методу вилучення». Рефакторинг, описаний у книзі, виконується вручну та використовується для розбиття великих методів на менші методи [13]. Однією з мотивацій є те, що менші методи збільшують ймовірність того, що інші методи також зможуть їх використовувати, що саме і робиться в цьому проєкті : фрагменти коду, які використовуються повторно, вилучаються. Однак ми робимо це автоматично, а не вручну.

Пропозиції щодо рефакторингу на основі метрик

Інструмент із поширеними цілями, як і ця робота, — це Aries (5). Можливості рефакторингу коду пропонуються користувачеві у вигляді інструменту Aries у вигляді шаблонів рефакторингу «Метод вилучення» та «Метод підтягування», також описаних у (11). Aries використовує вивід CCFinder (4) та виконує

на ньому метрики коду, щоб запропонувати, які знайдені клони коду підходять для рефакторингу. CCFinder використовує лише лексичний аналіз, в Aries синтаксична інформація додається шляхом розбору вихідних файлів, де знайдені клони коду. Однак Aries не розглядає семантику та типи. Таким чином, він може базувати свої пропозиції лише на синтаксисі, тому не можна гарантувати, що запропоновані рефакторинги можливі, особливо якщо вони будуть застосовані до такої мови, як Visual Basic, зі структурами як властивостями та модулями. Отже, Aries підходить як допомога для подальшого ручного рефакторингу, але не вирішує проблему глибокого автоматичного рефакторингу.

Рефакторинг клонів коду на основі зрізів

У (12) дублікований код знаходиться за допомогою підходу на основі програмного зрізу [14]. Надаються алгоритми для вилучення методів для дублікатного коду. Вони були реалізовані лише частково, залежності між операторами все ще потрібно було визначати вручну. Два вузли вважаються співпадаючими, якщо їхні внутрішні вирази ідентичні, ігноруючи всі імена змінних та літеральні значення [15]. Отже, визначення параметрів вилученого методу було б нетривіальним, що робить цей метод менш придатним для автоматичного рефакторингу клонів на основі методів.

1.4 Висновок по розділу

У першому розділі було розглянуто теоретичні основи зменшення дублювання коду в процесі розробки програмного забезпечення. Проаналізовано поняття дублювання коду та визначено його негативний вплив на якість, підтримуваність і масштабованість програмних систем. Досліджено методи виявлення клонів коду, що дозволяють знаходити повторювані фрагменти програмного коду та оцінювати рівень їх поширення. Також розглянуто рефакторинг як один із основних засобів усунення дублювання та покращення структури програмного забезпечення. Отримані результати формують теоретичну основу для подальшого дослідження методів автоматизованого аналізу та оптимізації коду.

РОЗДІЛ 2. МЕТОДИ ТА ІНСТРУМЕНТИ ЗМЕНШЕННЯ ДУБЛЮВАННЯ КОДУ

2.1 Автоматизовані підходи до виявлення та усунення клонів коду

Основним внеском цієї роботи є проектування та розробка Janus, плагіна Eclipse, метою якого є автоматизація виявлення клонів коду та рефакторингу для проектів Java за допомогою ітеративного робочого процесу, зведеного на рисунку 2.1.

Цей робочий процес поділено на чотири основні кроки: *(i)* попередня обробка вихідного коду, *(ii)* виявлення клонів коду з урахуванням рефакторингу, *(iii)* автоматизований рефакторинг клонів коду та *(iv)* перевірки поведінки.

Першим кроком є *попередня обробка вихідного коду*, під час якої вихідний код завантажується та обробляється для полегшення доступу та розробки на наступних кроках потоку. Крім того, на цьому етапі завершується нормалізація коду, що означає, що деякі відмінності, які можуть впливати на виявлення клонів коду, усуваються. Деякими прикладами таких відмінностей є значення констант та назви змінних.

Завдяки етапу попередньої обробки можна обрати текстове порівняння як основний метод пошуку дублікатів коду в аналізованому проекті. Цей вибір мотивований тим, що текстове порівняння є найефективнішим підходом серед представлених через значний обсяг роботи, виконаної в контексті алгоритмів зіставлення зі зразком [16]. Наразі етап попередньої обробки дозволяє використовувати алгоритм точного зіставлення для виконання порівняння; однак цей підхід можна розширити, використовуючи більш дозвільні алгоритми. Деякі приклади, які були представлені, - це ймовірнісні алгоритми зіставлення рядків, які є особливо ефективними, або алгоритми вирівнювання рядків, які збільшують відмінності, що ігноруються, та збільшують тип клонів коду, придатних для виявлення.

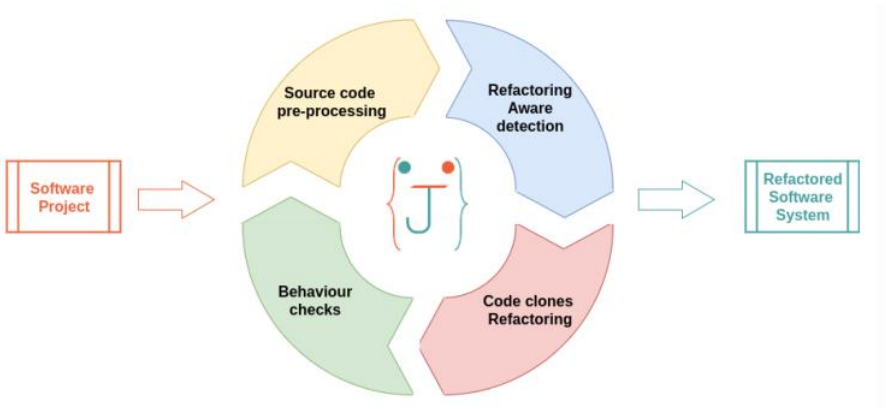

Рисунок 2.1 - Блок-схема процесу рефакторингу Janus

Дійсно, другий крок стосується вибору набору клонованих операторів, які будуть піддані процесу рефакторингу. Цей вибір виконується за допомогою алгоритму під назвою *Refactoring-Aware Detection (RAD)*. RAD працює виходячи з припущення, що не всі клони коду варті рефакторингу; зокрема, він враховує серйозність дублювання та зусилля, необхідні для видалення клонів коду, головним чином з точки зору впливу на загальну архітектуру та тип клону коду. Тому до проблеми виявлення дублікатів коду підійшли як до задачі оптимізації, яка вирішується за допомогою методів SBSE.

Щодо пов'язаних робіт, Janus обробляє дублікати послідовно, тобто по одному за раз, замість того, щоб розглядати групи дублікатів коду, оскільки рефакторинг групи клонів коду може впливати на рефакторинг (а іноді й на виявлення) інших екземплярів клонів коду. Цю проблему також розглядали, і формалізували її як обмеження в задачі оптимізації. Цей підхід безпосередньо не стикається з цією проблемою, оскільки вибирається лише один екземпляр дубліката коду [17]. Однак проблема все ще може виникнути опосередковано, оскільки рефакторинг одного набору операторів clones все ще може зробити недійсним рефакторинг інших.

Перший важливий аспект, який слід підкреслити, полягає в тому, що виявлення здійснюється на рівні деталізації операторів першого рівня (див. рисунок 2.2), що суттєво знижує ймовірність виникнення цієї проблеми. Цей аспект буде детальніше описано в наспипних розділах.

Крім того, екземпляр вибраного дубліката коду вважається найціннішим для цієї конкретної ітерації, що має означати, що недійсний екземпляр, який не можна рефакторувати, є менш цінним або становить вищий ризик.

Цей підхід можна розглядати як жадібний алгоритм m, де ми сподіваємося досягти «найкращого» результату шляхом рефакторингу «найкращого» дубліката екземпляра коду на кожному кроці.

Загальновідомо, що жадібні алгоритми не гарантують оптимального рішення. Однак цікаво відзначити, що цей підхід насправді близький до стандартного способу, за допомогою якого виконується ручний рефакторинг. Таким чином, розробник може краще зрозуміти кроки, що виконуються алгоритмом [18]. Більше того, розробник зможе впливати на спосіб вибору дубльованих операторів і, таким чином, змусити RAD імітувати спосіб, у який вони самі вибирали б клони коду.

Після вибору дублікованих операторів, які потрібно рефакторувати, запускається *автоматизований рефакторинг клонування коду* , де Janus аналізує оператори та вирішує, чи може він обробляти їх автоматично. Щодо відповідної роботи, Janus може автоматично застосовувати три методи рефакторингу: метод вилучення, метод вилучення та підтягування та метод вилучення суперкласу. Крім того, він частково може застосовувати утилітний метод вилучення, якщо дублікований код знаходиться у статичному методі.

Вибір впровадження лише цих методів рефакторингу ґрунтується на тому факті, що їхній вплив на архітектуру проекту мінімальний; наприклад, вони не додають жодної залежності між пакетами чи навіть об'єктами, оскільки використовують ту, яка вже існує. З цієї причини вони особливо добре підходять для автоматичного застосування.

Тим не менш, існують деякі структури коду, такі як лямбда-функції, які можуть призвести до внесення функціональних змін у Janus. З цієї причини поведінкові перевірки інтегровані в робочий процес Janus як крок верифікації. Цей четвертий і останній крок детально описано в наступному розділі. На цьому етапі

користувач може вибрати класи, що містять «основні методи», та тестові випадки JUnit, які будуть виконуватися після кожного кроку рефакторингу. Крім того, проект буде версіонований, використовуючи Git як систему контролю версій, протягом усього процесу рефакторингу, щоб забезпечити автоматичне відкат змін у разі невдачі поведінкових перевірок. Більше того, версіонування дозволить розробнику легко відстежувати всі зміни, застосовані Janus, та досить легко обробляти їх завдяки значній кількості функцій, які Git вже надає. До всіх згаданих вище функцій можна отримати доступ через графічний інтерфейс. Крім того, валідація RAD та всього робочого процесу Janus наведена в розділі.

Як згадувалося, ці функції не розробляються з нуля, а всі вони будуються на основі вже доступних бібліотек, які частково або повністю реалізують необхідні функції [19]. Цей аспект підкреслюється діаграмою стека, зображеною на рисунку 2.2, на якій кожен з основних пакетів, що складають Janus, спирається принаймні на одну зовнішню бібліотеку.

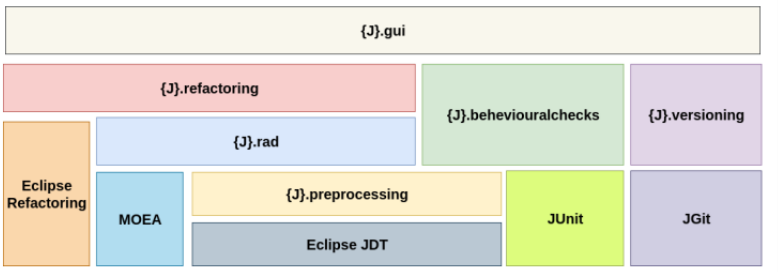

Рисунок 2.2 - Діаграма стека архітектури Janus.

Бібліотеки, наведені вище, потребують бібліотек, наведених нижче, для виконання своїх завдань. Блоки, що починаються з {J}, представляють пакети Janus.

2.2 Попередня обробка та аналіз вихідного коду

У цьому підрозділі міститься опис першого кроку робочого процесу. Цей крок стосується операції попередньої обробки, яка застосовується для

полегшення доступу до коду та його опрацювання на наступних кроках потоку.

Цей розділ поділено на два розділи: перший описує функціональність, що надається Janus стосовно цього кроку, тоді як другий розділ надає деякі відомості про реалізацію, а також короткий опис архітектури пакета, пов'язаного з цим кроком.

Дизайн та функціональність

Перший крок стосується завантаження та обробки вихідного коду для отримання набору методів, з якими працюватиме Janus. Цей крок поділено на три операції, показані на рисунку 2.3: (i) вилучення методів та операторів, (ii) нормалізація коду та (iii) вибір методів.

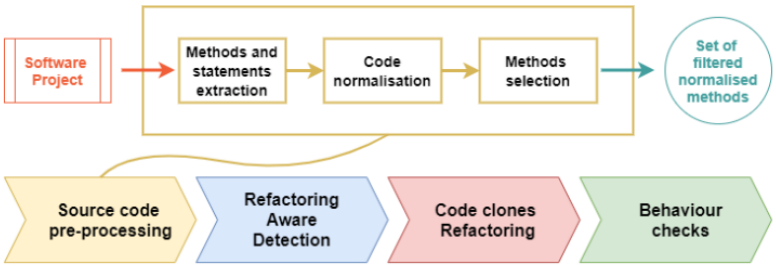

Рисунок 2.3 - Блок-схема етапу попередньої обробки

Методи та видобування операторів

Вихідний код перетворюється на власне абстрактне синтаксичне дерево, описане на рисунку 2.4, яке має оператори першого рівня як листя, а папку проекту як корінь [20].

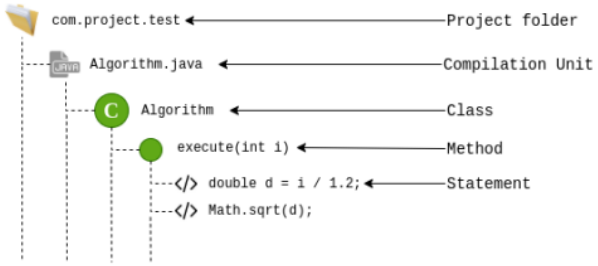

Рисунок 2.4 - Користувацьке абстрактне синтаксичне дерево, створене під час етапу попередньої обробки

Оператор першого рівня – це оператор Java, який не вкладений в інший оператор. Приклад представлено на рисунку 2.5. Оператор першого рівня – це оператор **for** та всі оператори, що входять до нього, тоді як кожен вкладений оператор є оператором другого рівня.

Ця структура даних використовується лише для розбору вихідного коду та вилучення операторів першого рівня, і вона не враховується під час виявлення дублікатів коду [21]. Фактично, клони коду виявляються на рівні деталізації операторів першого рівня.

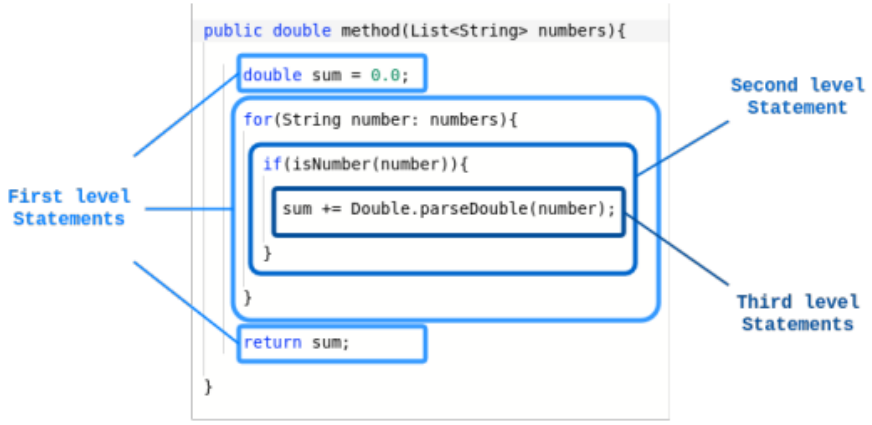

Рисунок 2.5 - Приклад рівнів операторів у фрагменті коду

Нормалізація коду

Деякі елементи коду замінюються токеном за замовчуванням. Зокрема, назви змінних замінюються значенням за замовчуванням **var** , тоді як також можна запитувати заміну значень констант (значення за замовчуванням залежить від типу, як зазначено в таблиці 2.1). Цей варіант залишається за користувачем, оскільки Janus ще не може автоматично рефакторувати клони коду, які використовують різні значення констант [22]. Тому, запитуючи цей додатковий крок попередньої підготовки, кількість клонів, які необхідно рефакторувати вручну, може збільшитися.

Крім того, з операторів першого рівня видаляються всі зайві символи, включаючи нові рядки, пробіли та крапку з комою.

Таблиця 2.1

Постійні значення за замовчуванням після нормалізації

Тип даних (Type)	Значення за замовчуванням (Default Value)
number (число)	0
char (символ)	'x'
boolean (логічний)	false
String (рядок)	"const"

Вибір методу

Методи можна відфільтрувати на основі кількох атрибутів, кожен з яких пов'язаний з різними випадками:

- довжина методу з точки зору кількості рядків коду та кількості символів. Фактично, можна вказати мінімальну довжину, необхідну для включення методу до простору пошуку. Ця функція дозволяє уникнути включення методів, які недостатньо довгі для включення клонів коду, що потребують рефакторингу, таких як методи *отримання* та *встановлення* ;
- тип коду, що міститься, насправді можна запросити виключення всіх тестових випадків. Ця функція мотивована тим, що код, що міститься в тестових класах, часто сильно дублюється, але користувач може захотіти зберегти це дублювання, щоб переконатися, що кожен тестовий випадок є незалежним та зрозумілим сам по собі.
- потреби користувача, фактично також можна вказати набір пакетів, класів або методів, які слід ігнорувати під час етапу попередньої обробки, використовуючи файл конфігурації. Також можливо не включати оператори, що містять певне ключове слово, через той самий файл конфігурації. Приклад цього файлу конфігурації наведено на рисунку.

Результат цього першого кроку ми називатимемо набором *відфільтрованих нормалізованих методів,* де кожен метод можна формально визначити як набір усіх нормалізованих операторів, що містяться в ньому. Крім того, варто зазначити, що оператори також піддаються процесу відбору через те, що

рефакторинг певних операторів змінить поведінку проекту або навіть призведе до помилок компіляції. Конкретний приклад наведено на прикладі всіх операторів першого рівня, які містять оператор «return».

2.3 Інструментальні засоби та практики впровадження рішень

Пакет попередньої обробки, зображений на рисунку 2.6, реалізує шаблон проектування «Фасад» [25], що забезпечує єдину та зрозумілу точку входу, через яку можна отримати доступ до всієї наданої функціональності.

Крім того, обробка всіх методів реалізована за допомогою шаблону проектування *пулу об'єктів* [25], який дозволяє мати один екземпляр одного класу, який повинен створювати, містити та обробляти всі екземпляри іншого класу [23]. У цьому випадку InstanceHandler є інтерфейсом для такого класу, а MethodHandler – фактичною реалізацією. Method – це клас, який має оброблятися, і він представляє один член набору відфільтрованих нормалізованих методів.

Інтерфейси InstanceHandler та Instance – це єдині два файли, які повинні імпортуватися будь-яким клієнтським класом, оскільки вони забезпечують абстракцію всього, що обробляється цим пакетом.

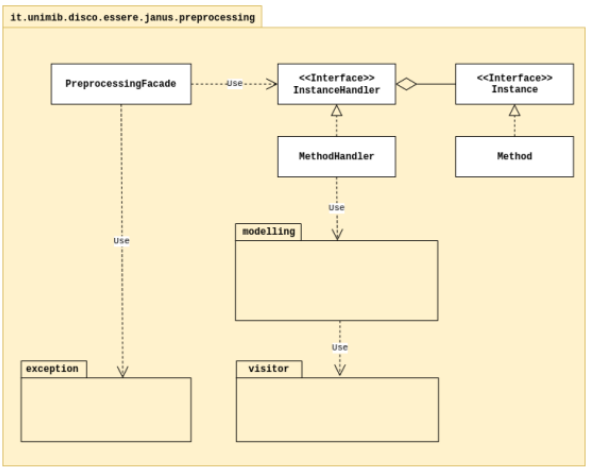

Рисунок 2.6 - Схема пакета попередньої обробки

Зрештою, є два основні підпакети, де реалізована фактична бізнес-логіка,

а саме:

- Пакет моделювання , який містить усі класи, що представляють кожен ключовий елемент проекту Java, починаючи від JavaDirectory до JavaStatement, як показано на рисунку 2.4. Усі класи, що належать до цього пакету, були реалізовані відповідно до шаблону проектування Composite [25], який спеціально використовується для моделювання об'єктів, що мають деревоподібну структуру, тому він був використаний для моделювання кожної нотатки Cutom AST (див. рисунок 2.4).
- Пакет Visitor , що містить класи Visitor, пов'язані зі шаблоном проектування *Visitor* [25]. Цей шаблон проектування дозволяє здійснювати навігацію та пошук у складній структурі шляхом створення екземплярів класів, які повинні дотримуватися певного інтерфейсу. У цьому випадку складна структура – це абстрактне синтаксичне дерево, що надається Eclipse JDT. Ці класи дозволять, наприклад, шукати всі константні значення або всі імена змінних у кожному класі Java у проекті.

Рефакторинг коду

У попередніх розділах ми показали, як вихідний код перетворюється на абстрактні синтаксичні дерева (AST), як будується система типів і як роздільна здатність типів може бути використана для визначення типу та походження значень, представлених виразами в коді. У цьому розділі ми покажемо, як цю функціональність можна використовувати для пошуку семантично однакових фрагментів коду та, за бажанням, для їх видалення шляхом переписування коду за допомогою уніфікованого рефакторингу вилучення методів. Рефакторинг – це спосіб перетворення коду зі збереженням поведінки (15). Одним з них є шаблон вилучення методів. Шаблон вилучення методів описує, як фрагмент коду можна вилучити та помістити в новостворений метод, а також як оригінальний фрагмент можна замінити викликом вилученого методу.

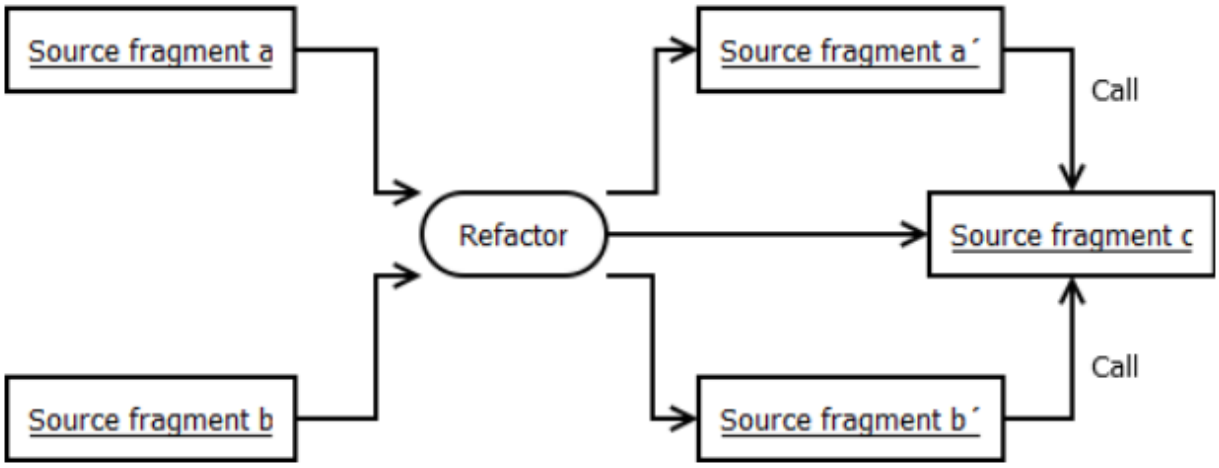

Рисунок 2.7 - Система рефакторингу виглядає, як показано на діаграмі.

«Просте» вилучення методу базується на одному фрагменті коду. На противагу цьому, вилучення методу, яке потрібно виконати в нашому контексті, бере два фрагменти коду та створює метод, виклик якого може замінити обидва фрагменти. За браком кращого терміна ми називаємо це вилученням *об'єднуючого* методу [23]. Щоб виконати корисне вилучення об'єднуючого методу в контексті рефакторингу клонів, алгоритм зіставлення має бути побудований таким чином, щоб його результатами були клони коду, придатні для вилучення.

Вхід системи рефакторингу формується двома фрагментами вихідного коду: вихідним фрагментом a та вихідним фрагментом b. Якщо вони містять семантично рівний підфрагмент, то на виході буде три фрагменти коду. Один фрагмент (вихідний фрагмент c) буде новоствореним методом, що містить заміну семантично рівних підфрагментів з вхідних даних. Два інші фрагменти (вихідний фрагмент a' та вихідний фрагмент b') - це вихідний фрагмент a та вихідний фрагмент b, перетворені таким чином, що семантично рівні підфрагменти замінюються викликом новоствореного методу (c), а зовнішня поведінка вихідного фрагмента a' та вихідного фрагмента b' ідентична вихідному фрагменту a та вихідному фрагменту b. Розглянемо наступний приклад, у якому наступні два фрагменти коду є тілами деяких членів.

Лістинг 2.1 - Два фрагменти вихідного коду: вихідним фрагментом a та вихідним фрагментом b.

```
1   Dim c As Integer
2   c = 2
3   c += 1
4   For a As Integer = 1 To 10
5       c += a
6   Next
```

```
1   Dim c As Integer = 0
2   c += 1
3   For b As Integer = 1 To 100
4       c += b
5   Next
6   c += 1
7   c += 1
```

Якщо ми хочемо рефакторувати ці фрагменти вручну, створивши новий метод, виклик якого замінює відповідний виділений фрагмент без зміни зовнішньої поведінки програми, спочатку нам потрібно ідентифікувати відповідні під фрагменти [24]. Коли підфрагменти ідентифіковані, створюється новий метод. У новому методі ми ігноруємо той факт, що керуючі змінні циклів 'a' та 'b' називаються по-різному, і просто вибираємо ім'я, скажімо, 'a'. Вирази верхньої межі '10' та '100' відрізняються, тому їх слід замінити змінною. Результат може бути наступним.

Лістинг 2.2 - Створюється новий метод

```
1   Function NewFunc(ByVal c As Integer, ByVal newVar As
Integer) As Integer
2       c += 1
3       For a As Integer = 1 To newVar
4           c += a
5       Next
6       Return c
7   End Function
```

Код у прикладі можна змінити наступним чином, не змінюючи зовнішньої поведінки програми.

Лістинг 2.3 - Змінили вирази верхньої межі змінною

```
1   Dim c As Integer
2   c = 2
3   c = NewFunc(c, 10)
```

```
1   Dim c As Integer = 0
2   c = NewFunc(c, 100)
3   c += 1
4   c += 1
```

Тут ми ігноруємо той факт, що змінні 'a' та 'b' мають різні назви; оскільки обидві змінні 'a' та 'b' мають однаковий тип і використовуються абсолютно однаково в обох фрагментах коду, їх можна безпечно замінити новою змінною з іменем 'a'. Ми вирішили замінити вирази верхньої межі змінною, оскільки обидві змінні '10' та '100' мають однаковий тип у фрагментах, але їх значення відрізняються. Неявно змінна 'c' також замінюється новою змінною. Отже, новостворений метод містить три змінні: 'a', 'c' та 'newVar', дві з яких ('c' та 'newVar') стали параметрами методу, а одна з них ('c') повертається методом. Значення 'c' та 'newVar' потрібно передавати як аргументи, оскільки їхні значення відрізняються в обох вихідних фрагментах. Значення 'c' повертається, оскільки воно використовується знову після виклику методу.

Існує ймовірність, що програміст, який виконує ручний рефакторинг, захоче потім ще більше перетворити ці два фрагменти таким чином.

Лістинг 2.4 - Перетворення; кілька операторів об'єднуються в один інший оператор

```
1   Dim c = NewFunc(2, 10)
```

```
1   Dim c = NewFunc(0, 100)
2   c += 1
3   c += 1
```

На першому кроці цього прикладу, окрім самого вилучення методу, виконуються лише підстановки, оператори не змінюються. Тіло вилученого методу

починається зі складеного присвоєння, а після цього йде оператор циклу for, що містить складене присвоєння в своєму тілі, точно таке ж, як і в замінених

фрагментах. Другий крок включає перетворення; кілька операторів об'єднуються в один інший оператор [25].

Для спрощення реалізації в цій роботі виконуються лише підстановки та жодних перетворень. Отже, виконується вилучення об'єднуючого методу, як це було зроблено на першому кроці прикладу. Очищувальні перетворення, як показано на другому кроці прикладу, не виконуються, але їх можна додати в майбутній роботі, якщо це необхідно.

Те, як дубліковані фрагменти пов'язані один з одним, впливає на те, як їх можна видалити. У (16) визначено три випадки дублювання коду:

- Дублювання всередині одного класу.
- Дублювання між братніми класами.
- Дублювання між непов'язаними класами.

Кожен із цих випадків потребує різного підходу під час рефакторингу. Перший випадок є найпростішим; не потрібно особливо звертати увагу на межі класів. У другому випадку, коли класи мають один і той самий суперклас, можливо, можна буде видалити клон коду з рефакторингу, розмістивши об'єднуючий видобутий метод у суперкласі. Найскладніший випадок — це коли є клони коду в непов'язаних класах; це може свідчити про нову відповідальність, яку можна винести в окремий клас.

Область видимості методу впливає на те, як можна отримати доступ до змінних та членів. Якщо, наприклад, метод, що витягується, потребує доступу до членів екземпляра типу 'A', спосіб побудови методу залежить від того, де знаходиться цей метод.

Наприклад, якщо витягнутий метод поміщається в сам «A», то інші методи «A» будуть безпосередньо доступні з цього методу. Якщо метод поміщається в супертип «A», то безпосередньо доступні лише методи в супертипі. Якщо метод поміщається в зовсім інший тип, екземпляр «A» повинен бути доступний

якимось іншим способом для доступу до методів на ньому.

Якщо потрібно вилучити об'єднувальний метод з фрагментів непов'язаних типів, спочатку слід визначити, де найкраще розмістити вилучений метод. Після визначення місця, залежно від випадку, наслідки можуть бути численними. Найпростіший випадок - вилучити об'єднувальний метод з двох фрагментів, якщо ці два фрагменти містяться в одному оголошенні типу. У цій роботі розглядається лише найпростіший випадок.

Щоб виконати вилучення об'єднавчого методу, спочатку потрібно визначити семантично рівноцінні фрагменти коду, які підходять для вилучення. З цих фрагментів потрібно зробити вибірку. Після цього вибраний збіг потрібно переписати.

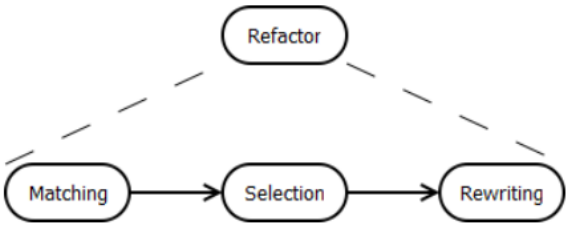

Рисунок 2.8 – Діаграма кроків рефекторингу

2.4 Висновок по розділу

У другому розділі було досліджено методи та інструменти зменшення дублювання коду. Розглянуто автоматизовані підходи до виявлення та усунення клонів коду, що дозволяють підвищити ефективність аналізу програмних систем. Проаналізовано методи попередньої обробки та аналізу вихідного коду для підготовки даних до процесів пошуку дублювань. Також досліджено сучасні інструментальні засоби та практики впровадження рішень для автоматизації процесів рефакторингу та оптимізації структури програмного забезпечення. У результаті визначено ефективні підходи до зниження рівня дублювання коду у програмних системах.

-РОЗДІЛ 3. РЕАЛІЗАЦІЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ МЕТОДІВ

3.1 Аналіз та виявлення клонів коду з урахуванням рефакторингу

У цьому розділі міститься опис другого кроку робочого процесу, представленого в попередньому розділі. Цей крок стосується виявлення набору клонованих операторів, які будуть піддані процесу рефакторингу, що здійснюється за допомогою алгоритму під назвою «Виявлення з урахуванням рефакторингу».

Потреба в алгоритмі виявлення, який знає, що виявлений клон коду буде автоматично рефакторовано, випливає з того факту, що не всі клони коду повинні бути рефакторовані [26]. Таким чином, метод виявлення з урахуванням рефакторингу має на меті бути алгоритмом виявлення, здатним оцінити фактори ризику, пов'язані з рефакторингом заданого набору клонованих операторів, і порівняти їх з покращенням супроводжуваності, що виникає в результаті рефакторингу.

Цей розділ поділено на два розділи: перший описує функціональність, що надається Janus стосовно цього кроку, тоді як інший розділ надає деякі відомості про реалізацію та короткий опис архітектури пакета, пов'язаного з цим кроком.

Дизайн та функціональність

Головною метою виявлення з урахуванням рефакторингу (RAD) є вибір екземпляра клонів текстового коду. Зокрема, RAD здатний виявляти клони коду Типу I, II та лише підмножину Типу III, оскільки виявлення виконується на рівні гранулярності операторів першого рівня [27]. У цьому контексті термін «екземпляр клонів коду» означає набір операторів, продубльованих у всій аналізованій кодовій базі. Однак важливо підкреслити, що набір операторів, вибраних RAD, може бути лише підмножиною всіх операторів, які є клонами вибраних. Це пов'язано з тим, що RAD також враховує ризик рефакторингу, який буде описано.

Зокрема, до цієї проблеми було підійдено як до задачі оптимізації шляхом визначення двох цільових функцій, які представляють основні аспекти, що мають враховуватися під час процесу рефакторингу: (i) подібність коду, яка

вимірює обсяг коду, спільного для різних компонентів набір методів, та (ii) ризик рефакторингу, що відображає як очікувані *зусилля на рефакторинг,* так і *ризик появи інших дефектів* під час процесу рефакторингу. Таким чином, результат є компромісом між цими двома цілями, які часто можуть суперечити одна одній.

Одна з головних переваг RAD, яка відрізняє його від традиційних підходів, що використовуються для виявлення клонів коду, полягає в тому, що розробники мають простий спосіб налаштувати виявлення відповідно до своїх потреб [28]. Фактично, належне використання ваг функцій дозволить RAD віддавати перевагу одному типу клонів коду над іншими. Наприклад, вибираючи високі ваги для ризику рефакторингу, перевага буде надаватися клонам коду, які легше рефакторувати.

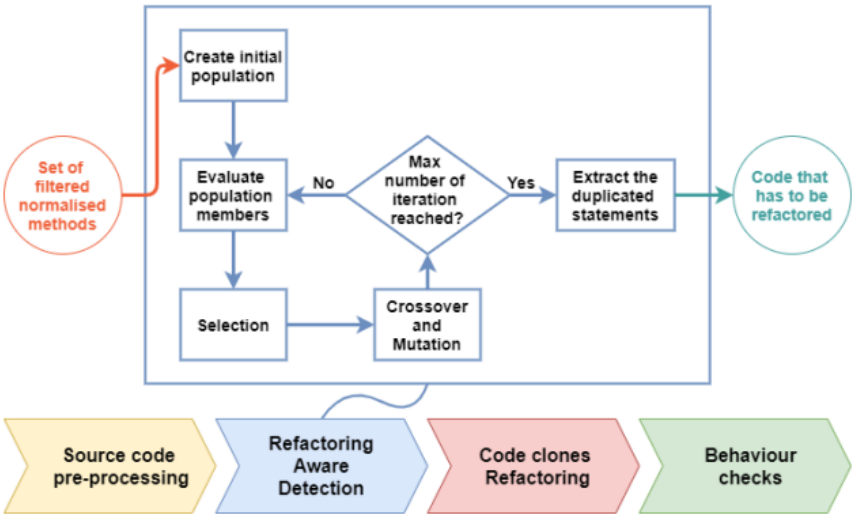

Рисунок 3.1 - Блок-схема етапу виявлення, заснована на використанні генетичного алгоритму для вирішення задачі оптимізації

Ця функція може допомогти у визначенні пріоритетів кроків рефакторингу на основі доступних ресурсів, тобто головним чином часу. Крім того, розробники можуть використовувати цей параметр, щоб RAD імітував спосіб, яким вони самі вибирали б клони коду.

Опис RAD організовано наступним чином: формально вводиться простір пошуку, потім визначається задача шляхом надання математичних функцій, що

складають задачу оптимізації, та опису значення та призначення кожної функції. Нарешті, аналізується задача оптимізації та вводяться алгоритми її розв'язання.

Пошуковий простір

Нехай M = {m i ,m $_2$, ...,*m $_n$ }* буде *множиною відфільтрованих нормалізованих методів,* тобто методів, що пройшли етап попередньої обробки, описаний у розділі . Простір пошуку можна визначити як:

$$S = 2^M(M \cup \{\emptyset\}) \quad (3.1)$$

що означає, що будь-яка підмножина, яка містить принаймні два методи, є можливим рішенням. Крім того, розглядаються лише групи з потужністю, меншою або рівною *max-card* (де *max-card* означає максимальну потужність; це параметр, який налаштовується користувачем), щоб зменшити розмір простору пошуку. Таким чином, можливі рішення належать до множини екземплярів *S′,* визначеної як:

$$S' = \{s \in S \mid |s| \leq \max\_card\} \subset S \quad (3.2)$$

Опис проблеми

Як зазначалося раніше, наш підхід базується на двох основних цілях. Ці дві цілі покликані знайти в *S′* підмножину методів з найбільшою схожістю та найнижчим ризиком ре факторингу [29]. У наступних розділах описано, як оцінюється кожен із цих аспектів, шляхом зазначення пов'язаної функції та детального опису її значення та призначення.

Функція подібності

Функцію подібності (рівняння 3.3) оцінюють шляхом: (i) створення набору, що містить усі твердження всіх методів, вибраних цією групою, (ii) порівняння кожного твердження між собою та (iii) обчислення кількості клонованих тверджень (NoCS), середньої кількості клонів для кожного дублікату тверджень (ADCL) та середньої кількості клонів для кожного дублікату тверджень (ANoC).

$$\text{Similarity}(s_i) = w_{s_1} \cdot \text{NoCS}(\text{stmts}_{s_i}) + w_{s_2} \cdot \text{ADCL}(\text{cloned}_{s_i}) + w_{s_2} \cdot \text{ANoC}(\text{cloned}_{s_i}) \quad (3.3)$$

Кожна з цих функцій, пов'язана з виявленням клонів коду, дозволяє впливати на пошук таким чином, щоб RAD віддавав перевагу одному типу клону коду над іншими типами. Це робиться шляхом вибору вагових коефіцієнтів, які є параметрами, що встановлюються користувачем.

Важливо підкреслити, що вид клонів коду не пов'язаний з його типом, тип клонів коду враховується за допомогою функції ризику рефакторингу (рівняння 3.4). Вид клонів стосується відмінностей, пов'язаних з такими аспектами, як довжина клонів коду або їх частота. Причина врахування цих факторів полягає в тому, що глибоке знання коду може призвести до розуміння того, які типи клонів коду більш ймовірні, і цю інформацію можна використовувати для ефективнішого процесу виявлення [30].

Наприклад, якщо більша ймовірність копіювання та вставки невеликих фрагментів коду, тоді також можна призначити вищі ваги, пов'язані з функцією ANoC, та нижчі значення ваг, пов'язаних з ADCL. Таким чином, RAD буде змушений включати якомога більше клонів, і такі клони коду будуть переважнішими за набір довгих, але дуже рідкісних клонів коду.

Кількість клонованих операторів (NoCS)**:** ця функція обчислює, скільки операторів клоновано у вибраних методах. Саме за допомогою цієї функції фактично здійснюється виявлення клонів коду. Використовується простий алгоритм точного збігу, як видно з функції Check (рівняння 3.1), що означає, що лише ідентичні оператори вважаються клонами. Це пов'язано з кроком нормалізації, що виконується під час фази попередньої обробки, який створює представлення коду, що відображає виключно його структуру, абстрагуючись від контекстуальних та змінних елементів.

$$\text{NoCS}(\text{stmts}_{s_i}) = \sum_{j=1}^{|\text{stmts}_{s_i}|} \sum_{r=j}^{|\text{stmts}_{s_i}|} \text{Check}(\text{stmts}_{s_i[j]}, \text{stmts}_{s_i[r]}) \quad (3.4)$$

де:

$$\text{Check}(x, y) = \begin{cases} 1 & \text{if } x = y \text{ (exact match)} \\ 0 & \text{otherwise} \end{cases} \quad (3.5)$$

Середня довжина клонованих операторів (ADCL): ця функція обчислює, скільки символів у середньому складають клоновані оператори у вибраних методах. Її можна використовувати, щоб RAD віддавав перевагу довшим клонам коду під час задачі оптимізації.

$$\text{ADCL}(\text{cloned}_{s_i}) = \frac{1}{|\text{cloned}_{s_i}|} \cdot \backslash\text{sum}_{j=1}^{|\text{cloned}_{s_i}|} \text{length}(\text{cloned}_{s_i[j]}) \quad (3.6)$$

Середня кількість клонів для кожного дубліката оператора (ANoC): ця функція обчислює, скільки разів у середньому дублікат оператора клонується в межах вибраних методів. Її можна використовувати, щоб RAD віддавав перевагу частим клонам коду під час задачі оптимізації.

$$ANoC(cloned_{s_i}) = 1/|cloned_{s_i}| \cdot \backslash sum_{j=1}^{|cloned_{s_i}|} |\{c\_y \in cloned_{s_i} \mid c\_y = cloned_{s_i}[j] \ (exact\ match)\}| \quad (3.7)$$

Рефакторинг функції ризику/ Функція Ризик рефакторингу повинна відображати як очікувані зусилля на рефакторинг, ¬так і ймовірність погіршення (появи дефектів або зниження якості) стану програмного проекту в результаті процесу рефакторингу.

Існує дві причини цього подвійного значення; більш теоретична полягає в тому, що дуже часто ці два аспекти тісно пов'язані один з одним. Наприклад, якщо два клони коду містять кілька константних значень, людині буде складніше виконати ефективний рефакторинг порівняно з операцією над клонами коду з

точним збігом. Хоча, якщо рефакторинг виконує машина, ймовірно, константи будуть передані як параметри, але вони генеруватимуть метод, на який впливає запах коду з довгим списком параметрів [23], що може означати зниження якості. Друга, більш практична причина полягає в тому, що не всі клони коду, які можна ідентифікувати, можуть бути автоматично рефакторовані [31]. У таких випадках Janus працює лише як детектор, і важливо враховувати зусилля, які можуть знадобитися людині. Хоча якщо клон коду можна рефакторувати автоматично, RAD є механізмом прийняття рішень Janus, тому важливо, щоб менш ризиковані екземпляри клонів коду були пріоритетними.

$$\mathrm{RefRisk}(s_i) = w_{rr_1} \cdot \mathrm{CodePosition}(s_i) - w_{rr_2} \cdot \mathrm{VarSim}(s_i) \qquad (3.8)$$

Цю мету можна реалізувати різними способами; досі розглянуті конкретні способи реалізації включають подібність змінних та розташування клонів коду.

Подібність змінних: ця функція перераховує кількість змінних, спільних для методів, що входять до вибраного набору. Вона була розроблена з неявним припущенням, що кількість змінних, спільних для методів, негативно корелює з кількістю конфліктів, які виникнуть під час процесу рефакторингу. Якщо кількість конфліктів низька, то ймовірність зниження якості програмного проекту також зменшується.

$$\mathrm{VarSim}(s_i) = \left|\bigcap_{m_i \in s_i} \mathrm{Variables}(m_i)\right| \qquad (3.9)$$

де *Variables(m **i**)* – це множина, що містить:

- параметр, необхідний для m*;
- поле класу, в якому визначено m*;
- усі змінні, оголошені в методі.

Позиції клонованого коду: ця функція оцінює відстань між методами в

наборі. Ця метрика базується на концепції відстані методів. Відстань між двома методами – це попередньо задане значення, яке відображає, чи знаходяться вони в одному класі, в одній ієрархії, в одному пакеті чи не пов'язані між собою.

Ці значення є параметрами, що встановлюються користувачем, але для правильного встановлення значень відстані важливо пам'ятати про головну мету функції Refactoring Risk: мінімізувати ризик, пов'язаний з рефакторингом клонів коду в межах вибраних методів [32]. Єдиним фактором, який повинен впливати на значення відстані в цьому контексті, є ймовірність появи дефектів або зниження якості в програмному проекті шляхом застосування методів рефакторингу, можливих для цієї конкретної відстані.

Вони ініціалізуються такими значеннями o=0, u=10, v=20 та z=50, адаптованими для доступних на даний момент методів рефакторингу, описаних у розділі. Тому, наприклад, метод рефакторингу, який включає клони коду в одній ієрархії, все ще може вимагати застосування методу рефакторингу Introduce Local Extension [23], що призведе до введення нового файлу Java в кодову базу. Таким чином, вплив цього методу значно вищий порівняно з рефакторингом клонів в тому ж класі. Ця функція дозволяє користувачеві точно вказати, наскільки відрізняється вплив одного методу рефакторингу порівняно з іншими, і яким методам рефакторингу слід надавати перевагу.

$$\mathrm{CodePosition}(s_i) = \sum_{j=1}^{|s_i|} \sum_{r=j}^{|s_i|} \mathrm{MethodsDist}(m_j, m_r) \qquad (3.10)$$

де:

$$\mathrm{MethodsDist}(m_i, m_j) = \begin{cases} o & \text{if } m_i \text{and } m_j \text{are in the same class} \\ u & \text{if } m_i \text{and } m_j \text{are in the same hierarchy} \\ v & \text{if } m_i \text{and } m_j \text{are in the same package} \\ z & \text{otherwise} \end{cases} \qquad (3.11)$$

$$\text{with: } o < u < v < z \text{ and } o, u, v, z \in \backslash\mathrm{mathbb}\ R^{+}$$

Аналіз задачі оптимізації

З огляду на дані та формалізми, представлені в попередніх розділах, а саме:

• si ∈ S, тобто група методів;

• stmtssi, тобто список усіх нормалізованих виразів кожного методу, включеного до si;

• clonedsi, тобто список усіх клонованих нормалізованих виразів у si;

Визначено таку задачу оптимізації:

$$\left(\text{maximize}_{S'=\{s_1,s_2,\ldots,s_t\}}\right)\text{FitnessFunction}(s_i) = w_s \cdot \text{Similarity}(s_i) - w_{rr} \cdot \text{RefRisk}(s_i) \quad (3.12)$$

Можна побачити, як ця функція придатності формалізує багатоцільову оптимізаційну задачу, що складається з двох основних функцій: подібності, яку необхідно максимізувати, та ризику рефакторингу, який необхідно мінімізувати [33]. Можна спостерігати, що кожна з цих двох функцій об'єднується як сукупність функцій , які тісно пов'язані одна з одною та дуже незалежні від усіх інших (тобто функції корелюють одна з одною та не залежать від функцій (3.4) та (3.5), і навпаки). Нарешті, цікаво відзначити, як дві основні функції, якщо розглядати їх окремо, досить легко оптимізувати, оскільки вони майже не підлягають жодним обмеженням. Фактично, враховуючи простір пошуку S', функція подібності максимізується одним з в $_i \in$ S', що містить максимально можливу кількість методів. Хоча функція ризику рефакторингу мінімізується однією S $_·$, що містить максимально можливу кількість методів. Однак, оскільки ці дві цілі мають бути оптимізовані разом, завдання оптимізації стає значно складнішим, тому потрібен розширений алгоритм оптимізації.

Сімейство алгоритмів, що використовувалися в цій роботі, — це генетичний алгоритм, як зображено на рисунку 3.1. Зокрема, для оцінки RAD були обрані такі алгоритми:

• Простий одноцільовий генетичний алгоритм, який вирішує саме

проблему, зазначену в рівнянні 3.5.

- NGSA-II [16] – це алгоритм оптимізації, розроблений для багатоцільової задачі оптимізації, що базується на використанні недомінованого підходу сортування, також відомого як оптимізація Парето. У цьому випадку цілями є функції подібності (рівняння 3.3), функція VarSim (рівняння 3.5) та функція CodePosition (рівняння 3.4).
- NGSA-III [14] – це алгоритм оптимізації, розроблений для багатоцільової задачі оптимізації, який, по суті, є розширенням NGSA-II, що використовує набір опорних точок для підтримки різноманітності точок Парето під час операції пошуку. Цілі оптимізації такі ж, як і в NGSA-II, тобто функції подібності (рівняння 3.7), функція VarSim (рівняння 3.8) та функція CodePosition (рівняння 3.9).

Причиною використання цих трьох алгоритмів є можливість здійснити більш надійне порівняння результатів, отриманих шляхом підходу до цієї проблеми як до одно-, багато- та багатоцільової задачі оптимізації. Фактично, у надзвичайно великому наборі еволюційних алгоритмів, розроблених протягом багатьох років, рідко можна знайти набір з них, узгоджений з точки зору операції, що виконується над членами популяції, яка відрізняється лише кількістю оброблюваних цілей [34]. Однак ці три алгоритми мають спільний чотириетапний процес генетичних алгоритмів (тобто оцінка, відбір, схрещування та мутація). Крім того, схрещування та мутація досить узгоджені в усіх трьох алгоритмах, тоді як основна відмінність полягає в оцінці та відборі, які є мінімальними відмінностями, які необхідно прийняти для здійснення такого порівняння.

Однак, важливо підкреслити, що технічно кожен алгоритм оптимізації чорної скриньки може бути використаний для вирішення цієї проблеми. З цієї причини Janus підтримує досить великий набір алгоритмів, які надаються MOEA Framework [W2], перелічених у таблиці 3.1.

Таблиця 3.1

Алгоритм, який можна використовувати в Janus для вирішення задачі оптимізації

Назва (Name)	Тип об'єктива	Опис (Description)
Genetic Algorithm (Генетичний алгоритм)	Одноцільовий	Найпростіша версія, що базується на чотирьох кроках: оцінка, відбір, кросовер (схрещування) та мутація.
ϵ-MOEA	Багатоцільовий	MOEA сталого стану, що використовує архівування за ϵ-домінуванням для запису оптимальних рішень за Парето.
ϵ-NSGAII	Багатоцільовий	Розширення NSGA-II з використанням ϵ-домінування та рандомізованого перезапуску для покращення пошуку.
MOEA\D	Багатоцільовий	Базується на декомпозиції проблеми на багато окремих одноцільових формулювань.
NSGAII	Багатоцільовий	Використовує підхід некерованого сортування (оптимізація за Парето).
NSGAIII	Багатоцільовий	Розширення NSGA-II, що використовує набір опорних точок для підтримки різноманітності точок Парето.
PAES	Багатоцільовий	Багатоцільова версія стратегії еволюції.
PESA-II	Багатоцільовий	Ще одна багатоцільова версія стратегії еволюції.
SPEA2	Багатоцільовий	Використовує метод «сили» (strength-based) для ранжування рішень.

Більше того, еволюційний підхід був особливо розглянутий завдяки його багатому досвіду успішного застосування до складних проблем, включаючи ті, що розглядаються у SBSE. Хоча генетичні алгоритми виконують евристичний пошук і тому можуть сходитися до локальних оптимумів або навіть до довільних точок (а не до глобального оптимуму), для цього сценарію використання не обов'язково знаходити оптимальне рішення; зазвичай достатньо «достатньо хорошого» кандидата на рефакторинг.

Впровадження

Пакет rad , зображений на рисунку 3.2, містить клас точки входу MethodSelector , який дозволяє запускати RAD через метод *selectInstances.*

Для коректного налаштування MethodSelector необхідно використовувати класи, що містяться в пакеті moea . Ці класи використовують переваги реалізації методу шаблонів проектування [25], реалізованого бібліотекою MOEA.

MOEA Framework [W2] — це безкоштовна бібліотека Java з відкритим кодом для розробки та експериментування з багатоцільовими еволюційними алгоритмами (MOEA) та іншими універсальними одно- та багатоцільовими алгоритмами оптимізації.

Реалізація RAD значною мірою спирається на цю бібліотеку, зокрема, класи, оголошені локально, розширюють абстрактний клас *AbstractProblem* [W3], що представляє ціль багатоцільової оптимізації. Основний метод, який перевизначається локально, - це той, який оцінює рішення, знайдені під час ітераційного процесу. Потім MethodSelector подбає про введення цих класів у робочий процес MOEA, щоб вони могли бути використані вибраним алгоритмом.

Для оцінки рішень пакет moea спирається на пакет оцінки , який включає класи, що реалізують функції. Як видно на рисунку 3.2, цей пакет поділено на два підпакети.

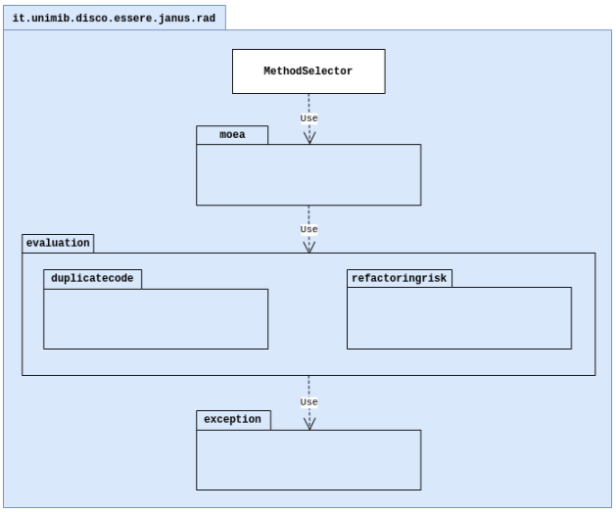

Рисунок 3.2 - Схема корпусу RAD

Перший, який називається dupliatecode , дозволяє обчислювати функцію подібності (рівняння 3.4), де здійснюється фактичне виявлення , і кожен дублікат коду аналізується для вилучення інформативної ознаки (рівняння 3.6 та 3.7). Другий, який називається refactoringrisk , обчислює функцію ризику рефакторингу (рівняння 3.5) шляхом вилучення інформації, пов'язаної з типом клону коду

(рівняння 3.3) та способом його розподілу в архітектурі проекту (рівняння 3.4).

Весь пакет організовано за допомогою методу шаблонів проектування [25]. Зокрема , всі класи, що виконують оцінку, розширюють AbstractEvaluator , абстрактний клас, що реалізує методи evaluate , що виконують ітерацію по популяції, та computeFitnessFunction , абстрактний метод, перевизначений будь-яким конкретним підкласом [35].

Зрештою, кожен із цих підпакетів організовано за допомогою стратегії шаблонів проектування [25], яка забезпечує спільний інтерфейс для кожної з двох основних груп класів, а саме AbstractDuplicateCodeEvaluator та AbstractRefRisksEvaluator .

3.2 Автоматизований рефакторинг та оптимізація структури коду

Цей підрозділ містить опис третього кроку робочого процесу, представленого в попередніх розділах. Метою автоматизованого рефакторингу клонів коду є визначення того, чи можна виконати рефакторинг автоматично, і в такому разі вибирається та застосовується відповідний метод рефакторингу.

Цей підрозділ поділено на два розділи, перший має на меті описати функціональність, що надається Janus стосовно цього кроку. Другий розділ надає деякі відомості щодо реалізації та короткий опис архітектури пакета.

Дизайн та функціональність

На цьому етапі вибрано екземпляр клонів коду, тобто набір дублікатів операторів, які потрібно рефакторувати. Тому метою цього кроку є вибір та застосування підходящих та здійсненних методів рефакторингу. Процес вибору зведено на рисунку 3.3.

На даний момент реалізовано лише підмножину можливих методів рефакторингу для виправлення клонів коду, а саме: метод вилучення, метод вилучення та підтягування, вилучення суперкласу та частково метод вилучення утиліти. Усі методи рефакторингу базуються на стратегії, яка застосовує метод вилучення [23] до одного з клонів коду, а потім замінює решту входжень клону

викликом вилученого методу.

Отже, перша проблема, яку має вирішити метод автоматизованого рефакторингу, полягає в наступному:

• Як можна зробити витягнуті методи доступними з інших методів, включаючи досліджуваний клон?

У підрозділі описано автоматизовані методи рефакторингу шляхом перерахування *(i)* попередніх умов, які необхідно виконати для використання кожного конкретного методу, та *(ii)* кроків, які

дотримуються для застосування цієї техніки рефакторингу.

Друга проблема, яку необхідно вирішити, полягає в наступному:

• Як врахувати всі відмінності між клонами коду, пов'язані з іменами змінних та константними змінними?

У розділі ця проблема описана детальніше. Крім того, обговорюється, якою мірою Janus здатний вирішувати ці конфлікти.

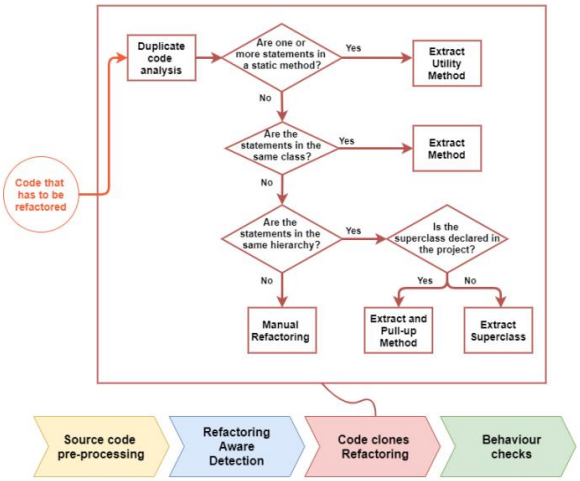

Рисунок 3.3 - Блок-схема етапу рефакторингу, яка підсумовує вибір рефакторингу серед доступних наразі варіантів

Метод вилучення

- Передумова:

— вибрані клони коду належать до одного класу.

Кроки рефакторингу:

1. Витягніть метод, який міститиме один із клонів коду;
2. Замініть клони коду other/s викликом видобутого методу.

Метод вилучення та підтягування

- Передумови:

— вибрані клони коду належать до різних класів;

— і всі залучені класи знаходяться в одній ієрархії;

— а найнижчий спільний суперклас задіяних класів — це клас, оголошений у файлі Java в каталозі вихідного коду.

- Кроки рефакторингу:

1. Витягніть метод, який міститиме один із клонів коду;
2. Підтягніть витягнутий метод до найнижчого спільного надкласу;
3. Встановіть видимість видобутого методу на захищений;
4. Замініть клони коду other/s викликом видобутого методу.

Витягти суперклас

- Передумова:

— вибрані клони коду розміщуються в різних класах;

— і всі залучені класи знаходяться в одній ієрархії;

— а найнижчий спільний суперклас задіяних класів не є класом, визначеним у файлі Java, присутньому в каталозі вихідного коду; отже, це клас, оголошений в одній із зовнішніх бібліотек, від яких залежить проєкт;

— а найнижчий спільний суперклас серед задіяних класів не є java.lang.Object .

- Кроки рефакторингу:

1. Застосуйте рефакторинг Introduction Local Extension [23], створивши новий клас, який розширює найнижчий спільний суперклас задіяних класів.
2. Застосування рефакторингу методом вилучення та підтягування

Однак, техніка вилучення суперкласу значною мірою залежить від дерева ієрархії. Поточна доступна реалізація не здатна реконструювати все дерево ієрархії всіх бібліотек, від яких залежить проект. З цієї причини вона включає лише

класи, які розширені принаймні одним класом з проекту, і, отже, це єдині, які можна використовувати для застосування рефакторингу вилучення суперкласу.

Крім того, якщо будь-який з клонів коду включено до статичного методу, Janus може переконатися, що видобутий метод також є статичним. Це необхідно, щоб не генерувати помилку під час компіляції, і саме тому було зазначено, що метод утиліти вилучення реалізовано частково.

Обробка конфліктів змінних та констант

Під час процесу рефакторингу клонів коду типів II та III можуть виникнути деякі конфлікти, що стосуються констант, значень та змінних. Приклади таких конфліктів представлені на рисунку 3.4.

У цьому розділі перераховані ці конфлікти, а також обговорюється, якою мірою Янус здатний з ними справлятися.

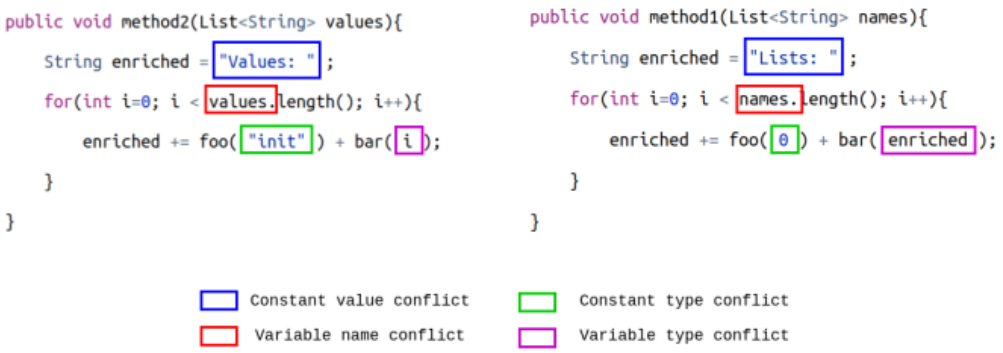

Рисунок 3.4 - Приклади клонів коду, можливі типи конфліктів

Конфлікти змінних

Такий конфлікт виникає, коли дві різні змінні використовуються в одній і тій самій позиції. Існує дві основні причини цього конфлікту:

- Конфлікти імен : дві змінні є екземплярами одного типу (класу), тому єдина відмінність полягає в назві змінної;
- Конфлікти типів : дві змінні є екземплярами різних типів;

Постійні конфлікти

Такий тип конфлікту виникає, коли два різні константні значення використовуються в одній і тій самій позиції в клонах коду. Існує два основних випадки,

коли може виникнути такий тип конфлікту:

- Конфлікти значень : дві константи різні, але одного й того ж примітивного типу;
- Конфлікти типів : дві константи належать до різних примітивних типів;

Під час етапу попередньої обробки, створюється представлення коду, яке дозволяє виконувати RAD. Очікується виявлення клонів з конфліктами в іменах змінних, значеннях констант, а також типах змінних. Наразі неможливо виявити дубліковані оператори, на які впливають конфлікти типів констант, як клони, оскільки значення констант за замовчуванням базуються на примітивному типі констант (як зазначено в таблиці 3.1).

Єдиний конфлікт, який Janus наразі може автоматично рефакторувати, це конфлікт імен змінних. Це пов'язано з тим, що рефакторинг методом вилучення завжди застосовується у всіх автоматизованих методах, тому імена, що використовуються у вилученому клоні, є тими, що й будуть використані.

Однак, конфлікт типів змінних та констант не планується вирішувати. Наявність таких конфліктів означає, що методи, які викликаються в кожному з клонів коду, не ідентичні, і поведінка зміниться після застосування рефакторингу в g. Janus здатний перевірити, чи клони коду зазнали впливу конфлікту імен змінних або конфлікту типів змінних [36]. У першому випадку він автоматично виконує рефакторинг, зберігаючи імена витягнутого клону коду. У другому випадку він викидає виняток, щоб повідомити про проблему. Таким чином, єдиним конфліктом, який Janus поки що не може вирішити, є проблема константних значень.

Впровадження

Пакет рефакторингу , зображений на рисунку 3.3, було реалізовано на основі шаблону проектування Factory Method [25]. Фактично, він має абстрактний клас CCRefactoring як точку входу, яка реалізує статичний метод selectRefactoringTechniques .

Цей метод виконує три основні кроки, зведені на рисунку 3.4:

1. він вирішує, який метод рефакторингу є можливим для дублікатів операторів, переданих на вхід;
2. він оголошує екземпляр вибраної методики рефакторингу;
3. він перетворює конкретний екземпляр на CCRefactoring та повертає його.

Реалізація рішення таким чином видаляє залежності від усіх конкретних класів і залишає лише ту, яка стосується абстрактного класу.

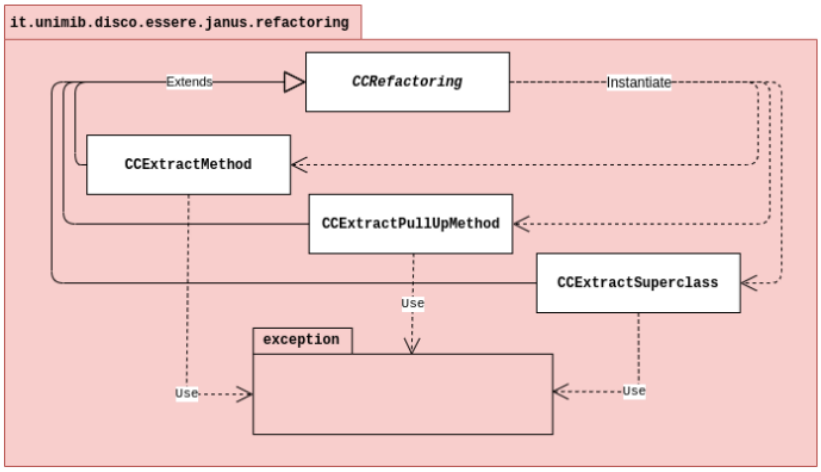

Рисунок 3.5 - Діаграма пакета ре факторингу

Поведінкові перевірки та керування версіями

У цьому розділі наведено опис четвертого та останнього кроку робочого процесу. Цей крок стосується виконання класів, що містять метод main, та тестових випадків JUnit, вибраних користувачем, щоб переконатися, що виконаний рефакторинг (див. розділ 3.6) не вплинув на функціональну поведінку програми. Крім того, зміни версійні, щоб забезпечити можливість відкату у випадку порушення поведінки, тобто коли хоча б один тест не пройде.

Цей розділ поділено на два розділи: перший описує, як виконуються перевірки поведінки, тоді як другий зосереджений на версійному управлінні. Крім того, обидва вони надають технічні деталі щодо реалізації функцій [36].

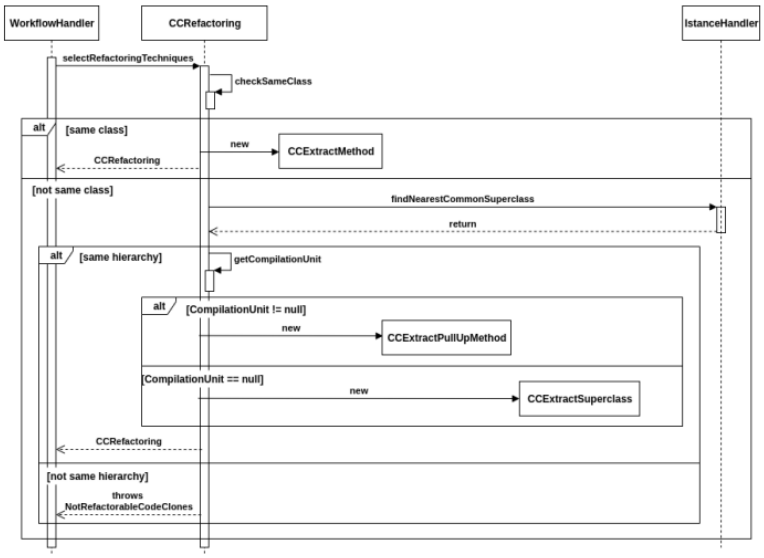

Рисунок 3.6 - Діаграма послідовності вибору методу рефакторингу

Поведінкові перевірки

Наразі підтримуються три типи перевірок: тест компіляції, тести основного класу та тести JUnit. У цьому розділі описано всі три типи тестів та наведено деякі деталі щодо їх реалізації.

Компіляційний тест

Проєкт компілюється після рефакторингу, щоб переконатися у відсутності помилок компіляції.

Цей конкретний тест виконується за допомогою плагіна Eclipse JDT, який забезпечує спосіб компіляції всього проекту за допомогою інтерфейсу iProject.

Тести основного класу

Усі класи з «головним методом», що належать до проекту та вибрані користувачем, мають бути виконані успішно, тобто без повідомлення про будь-які винятки чи помилки.Такий метод повинен відповідати трьом типовим умовам: *(i)* він має назву «main», *(ii*) він повертає «void», *(iii)* він є статичним та *(iv)* мати лише один параметр — масив рядків.

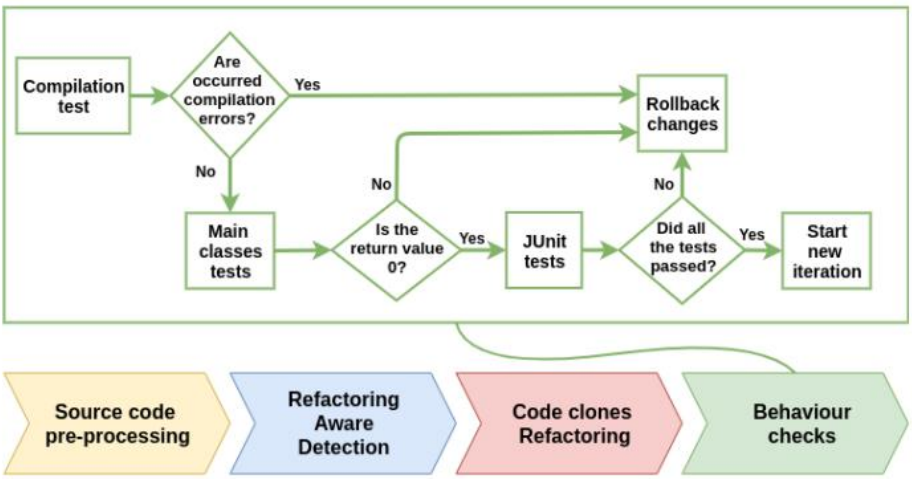

Рисунок 3.7 - Послідовність кроків, пов'язаних з перевірками поведінки

Пошук класів, що містять такі методи, виконується на етапі попередньої обробки, зокрема під час вилучення методу та оператора.

Далі, ці класи запускаються завдяки бібліотеці Apache Commons [W4], яка надає клас DefaultExecutor [W5], що дозволяє виконувати оператори командного рядка.

JUnit-тести

Усі тестові випадки JUnit [W 6], що містяться в проекті та вибрані користувачем, повинні пройти перевірку, тобто всі їхні твердження мають бути задоволені.

Список усіх класів, що містять тестові випадки JUnit, витягується за допомогою класу ITestFinder , який надається плагіном JUnit [W7] для Eclipse.

Версіонування

Версіонування всіх змін, застосованих Janus під час процесу рефакторингу, є надзвичайно важливою функцією. Вона дозволяє перевіряти та (можливо) відкатувати оновлення, зроблені під час та після процесу рефакторингу.

Janus інтегровано із системою контролю версій Git [W1]. Цей вибір зумовлений *(i)* широким використанням цього фреймворку як для проектів з відкритим кодом, так і для приватних проектів, а також *(ii)* великою кількістю наданих функцій.

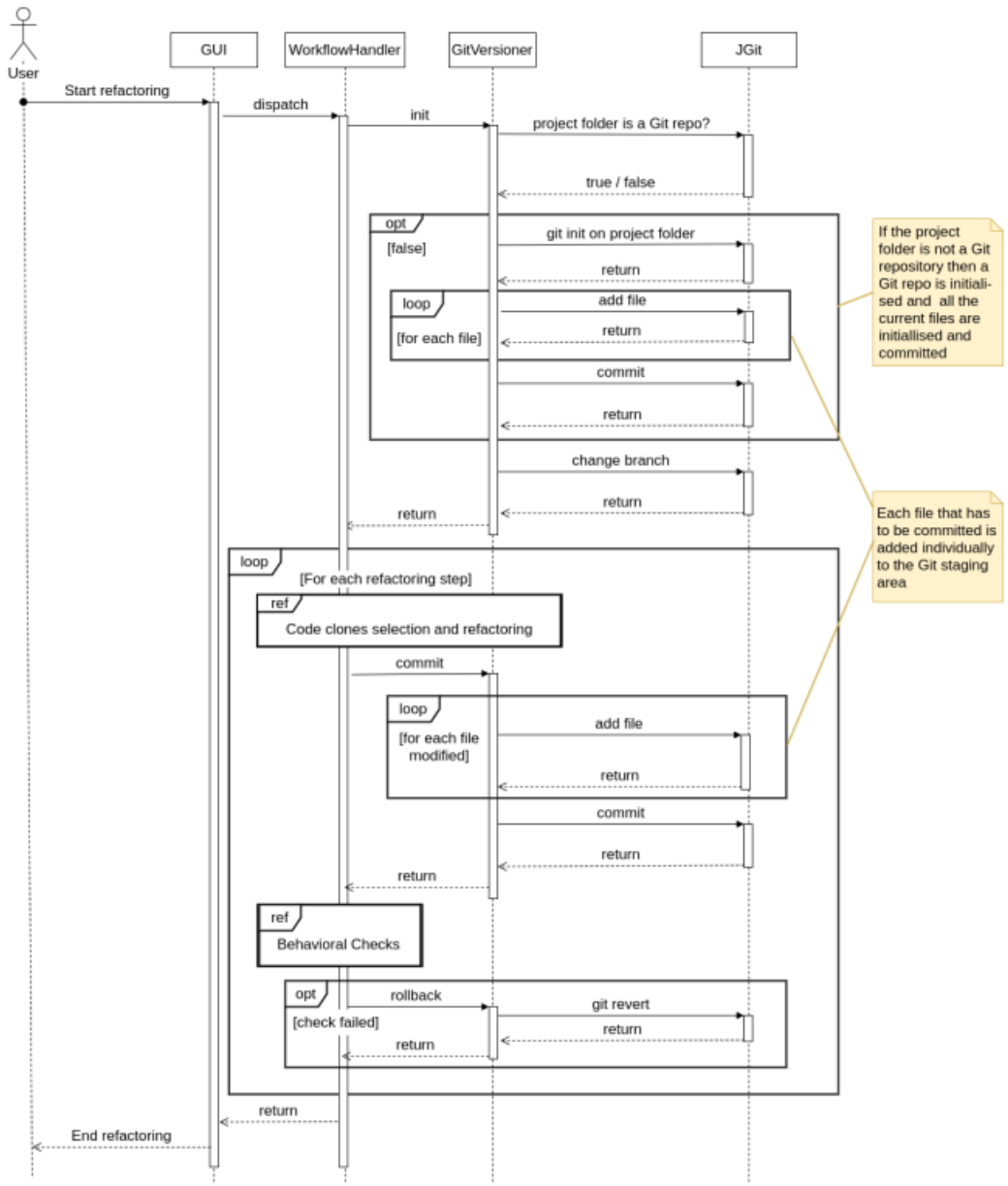

Рисунок 3.8 - Діаграма послідовності кроків версіонування протягом усього процесу рефакторингу

Інтеграція значною мірою спирається на JGit [W 8], бібліотеку Java, яка реалізує більшість функцій системи контролю версій Git, такі як процедури доступу до репозиторію, мережеві протоколи та алгоритми контролю версій.

Кроки версіонування, що виконуються під час процесу рефакторингу, зведені на рисунку 3.8. Спочатку необхідно перевірити, чи є папка проєкту репозиторієм Git. Якщо ні, то Janus ініціалізує її та виконає початковий коміт.

Однак, Janus не шукатиме репозиторій у жодній іншій папці. З цієї

причини також можна вручну вказати шлях до репозиторію Git, який користувач бажає використовувати для керування версіями [37].

Щойно папка проєкту стане ініціалізованим репозиторієм Git, Janus створить нову гілку (або візьме гілку, якщо вона вже існує) під назвою «CCRefactoring».

Згодом, для кожного кроку рефакторингу зміни, досягнуті за допомогою методів рефакторингу, фіксуються. Якщо хоча б одна перевірка поведінки не пройшла, застосовується відкат.

3.3 Оцінка результатів та валідація ефективності підходів

У цьому підрозділі обговорюється підтвердження ефективності Janus; зокрема, представлено два різні експерименти.

Перший з них представлено в розділі і зосереджено на оцінці здатності RAD сходитися до рішення: порівнюються різні алгоритми, а також обговорюються переваги та недоліки багатоцільового підходу шляхом порівняння його з одноцільовим. Крім того, способи вирішення задач оптимізації використовуються для аргументації того, якою мірою кожен алгоритм здатний забезпечити детермінований результат.

Другий експеримент, спрямований на оцінку ефективності всього робочого процесу. Тому основною темою обговорення є здатність Janus правильно рефакторувати клони коду без внесення дефектів або змін у поведінку. Нарешті, повнота Janus оцінюється шляхом порівняння кількості автоматично рефакторованих клонів коду зі списком усіх клонів коду, фактично присутніх у програмному проекті.

Конвергенція RAD

У цьому розділі оцінюється здатність RAD сходитися до змістовного рішення на основі алгоритму оптимізації, що використовується для вирішення задачі. Цей експеримент було проведено шляхом кількаразового виконання RAD на 16 проектах, перелічених у таблиці 3.2, з використанням трьох різних

алгоритмів.

Зокрема, три порівняні алгоритми представлені в розділі: Одноцільовий ГА, NSGAII та NSGAIII.

RAD було виконано п'ять разів для одноцільового генетичного алгоритму (ГА) та десять разів для двох інших алгоритмів. Різниця в кількості ітерацій зумовлена припущенням, що очікується, що одноцільовий ГА демонструватиме меншу дисперсію як у запропонованому рішенні, так і в загальному процесі оптимізації, порівняно з багатоцільовими алгоритмами.

Щоразу, коли виконувався RAD, незалежно від використаного алгоритму, записувалися такі дані:

• вибрані дубліковані твердження;

• значення найпристосованішого члена популяції протягом усіх ітерацій генетичного алгоритму (ГА);

• час, необхідний для виконання шістдесяти ітерацій.

Що стосується першого та другого пунктів, отримані результати узагальнено в Додатку В. Зокрема, еволюція найвідповіднішого члена представлена для кожного алгоритму та для кожного аналізованого програмного проекту.

Фактично, кожен ряд даних був зібраний шляхом обчислення середнього значення та стандартного відхилення значень найпристосованішого члена протягом кожного виконання для кожної з шістдесяти ітерацій алгоритму.

Аналізуючи криві, наведені на рисунках, можна зробити два цікаві висновки.

По-перше, графіки, пов'язані зі стандартним відхиленням, показують, що одноцільовий генетичний алгоритм (ГА) завжди здатний сходитися до рішення, на відміну від NSGAII та NSGAIII, які зазвичай намагаються мінімізувати дисперсію під час ітерацій. Ця особливість пов'язана з методом оптимізації, оскільки багатоцільові алгоритми використовують фронт Парето для цього експерименту [38]. Фактично, у випадках багатоцільового алгоритму єдиного оптимального рішення не існує, а існує набір недомінованих рішень; отже, результуюча дисперсія значення придатності буде вищою. Однак це не означає, що недоміновані

рішення менш цінні порівняно з тими, що вибрані одноцільовим ГА, оскільки кожне з них має бути спеціально підібране на основі принаймні однієї мети.

Крім того, це перше розуміння підтверджує припущення, зроблене під час розробки експерименту, для якого одноцільовий генетичний алгоритм (ГА) проводився лише п'ять разів замість десяти, і справді виконується.

По-друге, багатоцільові алгоритми здатні знаходити рішення, які значно краще підходять порівняно з одноцільовим генетичним алгоритмом (ГА). Це можна легко помітити, аналізуючи графіки, пов'язані із середнім значенням придатності. Це тісно пов'язано з відмінностями, що спостерігаються у стандартному відхиленні значення придатності, про які йшлося раніше. Фактично, оскільки ітерації слідують одна за одною, дисперсія в одноцільовому ГА зменшуватиметься, і буде все важче й важче покращувати рішення, навіть якщо крок мутації гарантуватиме, що деякі нові будуть враховані. З іншого боку, недоміновані рішення, природно, забезпечать вищу дисперсію, тому нащадки з великою ймовірністю досліджуватимуть нову область простору пошуку. З цього випливає той факт, що кожен з батьків особливо підібраний на основі принаймні однієї мети, ймовірно, що нащадкові рішення також будуть підібрані на основі принаймні однієї мети.

Враховуючи цю інформацію, можна стверджувати, що одноцільовий генетичний алгоритм (ГА) здатний швидше сходитися до рішення, і тому його рекомендується використовувати, коли користувач не хоче встановлювати малу кількість ітерацій. Навпаки, якщо користувач хоче встановити велику кількість ітерацій для алгоритму оптимізації, тоді рекомендується використовувати багатоцільовий підхід, оскільки це призведе до більш оптимальних рішень.

Більше того, дубльовані оператори, які були обрані в більшості випадків для різних виконань RAD для кожного алгоритму. Оскільки ваги різних функцій були встановлені одноманітним чином можна побачити, що були випадки, коли перевагу надавали довгому (але рідкісному) екземпляру клонів коду, та деякі інші, коли перевагу надавали короткому, але дуже частому екземпляру клону коду.

Нарешті, останній набір записаних даних – це час, необхідний для виконання шістдесяти ітерацій. Цю інформацію узагальнено на рисунку 3.9, де можна помітити, що генетичний алгоритм (ГА) зазвичай вимагає більше часу порівняно з багатоцільовими алгоритмами.

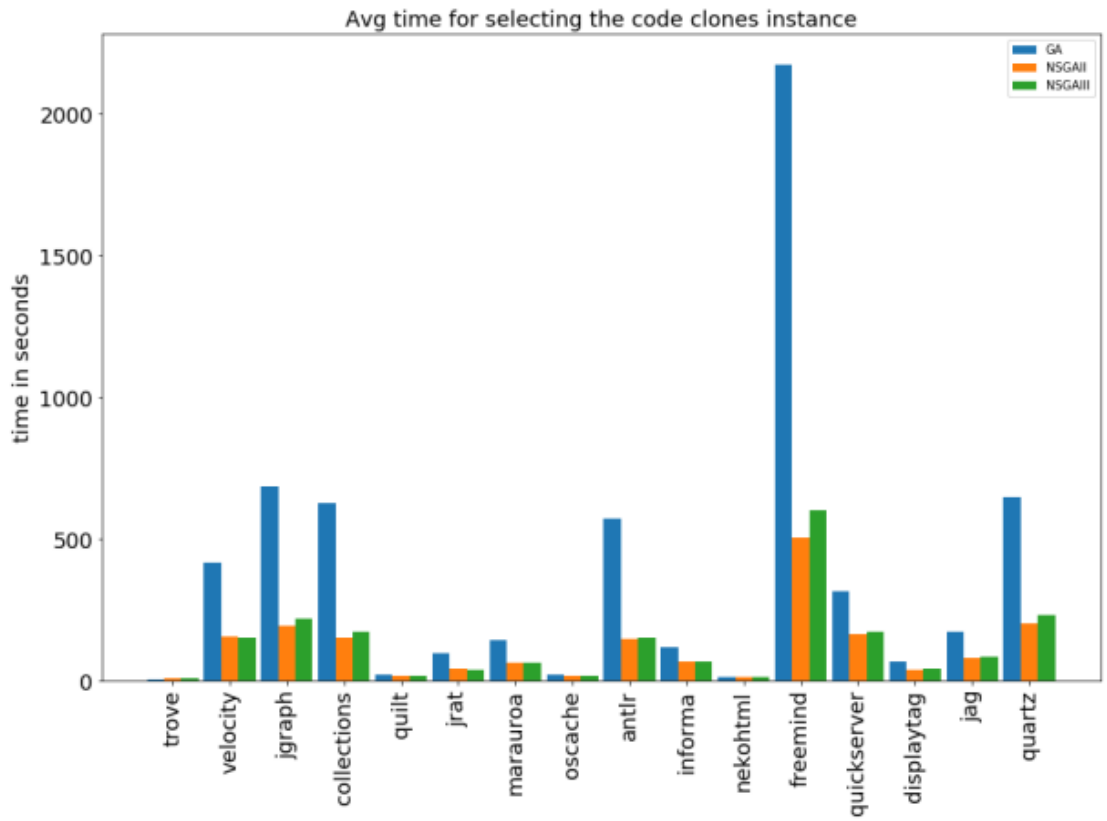

Рисунок 3.9 - Середній час, необхідний для вибору одного екземпляра клонів коду для кожного проаналізованого проекту

Тому той факт, що одноцільовий ГА здатний швидше сходитися до рішення, має бути зважений тим фактом, що час, необхідний для сходження до такого рішення, може бути високим. Навпаки, багатоцільові алгоритми потребують менше часу для виконання ітерації порівняно з одноцільовим ГА, але їм також важко сходитися до рішення. Тому знову ж таки, користувач повинен вирішити, який компроміс є найціннішим у конкретному випадку.

Автоматизований процес рефакторингу

У цьому розділі представлено оцінку ефективності всього потоку Janus. Зокрема, оцінку було виконано з використанням двох різних метрик оцінки, а

саме: виявлених/рефакторованих кластерів-клонів та дублікатів рядків коду. Крім того, можливості Janus були оцінені шляхом застосування автоматизованого рефакторингу на двох різних програмних системах.

Таблиця 3.2

Програмний проект, що використовується для перевірки збіжності RAD

Назва (Name)	**Версія**	**Домен (Domain)**	**Рядків коду (Lines of Code)**	**Кількість класів**
Apache Collection	1.4.2_19	інструменти (tool)	27 635	458
FreeMind	1.4.2_19	генератори діаграм, візуалізація	50 198	912
JGraphT	1.6.0	інструменти (tool)	11 931	255
JRat	1.4.2_19	тестування (testing)	14 146	250
Marauroa	1.5.0_22	ігри (games)	13 823	204
NekoHTML	1.2.2_017	парсери, генератори	6 625	55
OSCache	1.4.2_19	проміжне ПЗ (middleware)	6 198	75
QuickServer	1.4.2_19	проміжне ПЗ (middleware)	10 885	111
Quilt	1.4.2_19	тестування (testing)	5 683	77
Apache Velocity	1.4.2_19	генератори діаграм, візуалізація	26 854	261

Виявлено та рефакторовано клоновані кластери

Термін *«клонований кластер»* використовується для визначення *вичерпного* набір дублікатних операторів, захоплених в аналізованій кодовій базі. Отже, існує різниця між кластером клонів та екземпляром клонів коду: останній є менш обмежувальним, оскільки не вимагає, щоб набір містив усі дубліковані оператори, включені до кодової бази. Більш формально, можна стверджувати, що екземпляр клонів коду завжди включено до кластера клонів, враховуючи певний оператор.

Мета цієї метрики оцінювання - зрозуміти, скільки клонів коду, що містяться в проєкті, може виявити Janus і скільки з них він може автоматично рефакторувати. Однак RAD, тобто алгоритм виявлення клонів коду, який

використовується Janus, виявляє екземпляри клонів коду, а не кластери клонів, тому кластер клонів вважається виявленим, якщо виявлено принаймні два клони коду, включені до кластера клонів, і, отже, вважаються екземплярами клонів коду.

Базові дані, використані для розрахунку цієї метрики, були надані *колекцією клонів корпусів Qualitas*, що являє собою набір даних, що містить перелік та опис клонів коду, знайдених у більшості систем, включених до *корпусу Qualitas***.** Qualitas Corpus -це кураторська колекція програмних систем, призначених для емпіричних досліджень артефактів коду.

Дубльовані рядки коду

Дубльовані рядки коду представляють обсяг коду, який фактично дублюється в аналізованому проєкті. Метою цього показника оцінки є фактичне розуміння впливу автоматизованого рефакторингу на проєкт.

Цей показник було оцінено за допомогою SonarQube, автоматичного інструменту перевірки коду, який здатний виявляти помилки, вразливості та «запахи» коду в проекті. Тому рішення використовувати цей інструмент також дає додаткову перевагу, оскільки гарантує, що автоматизований рефакторинг не вносить дефектів коду (тобто помилок), моментів у коді, які можна використати для атак (тобто вразливостей), та інших проблем, які заслуговують на увагу.

Процес оцінювання

Послідовність дій, що використовуються для оцінки ефективності Janus, зведена на рисунку 3.10, поділена на чотири основні кроки:

1. Оцінити проєкт за допомогою SonarQube, щоб обчислити кількість помилок, вразливостей, запахів коду та дублікатів рядків коду перед процесом рефакторингу;
2. Потім виконується одна ітерація всього робочого процесу Janus, що означає, що чотири основні кроки: попередня обробка вихідного коду, виявлення клонів коду з урахуванням рефакторингу, автоматизований рефакторинг клонів коду та поведінкові перевірки, виконуються рівно один раз.
3. Екземпляр клону коду, який підлягає рефакторингу на цьому кроці,

перевіряється та зіставляється з кластером клонів, який його містить;

4. Якщо протягом п'яти послідовних разів Janus не зміг знайти інші клони коду для рефакторингу, то процес рефакторингу вважається завершеним, і проєкт знову оцінюється за допомогою SonarQube.

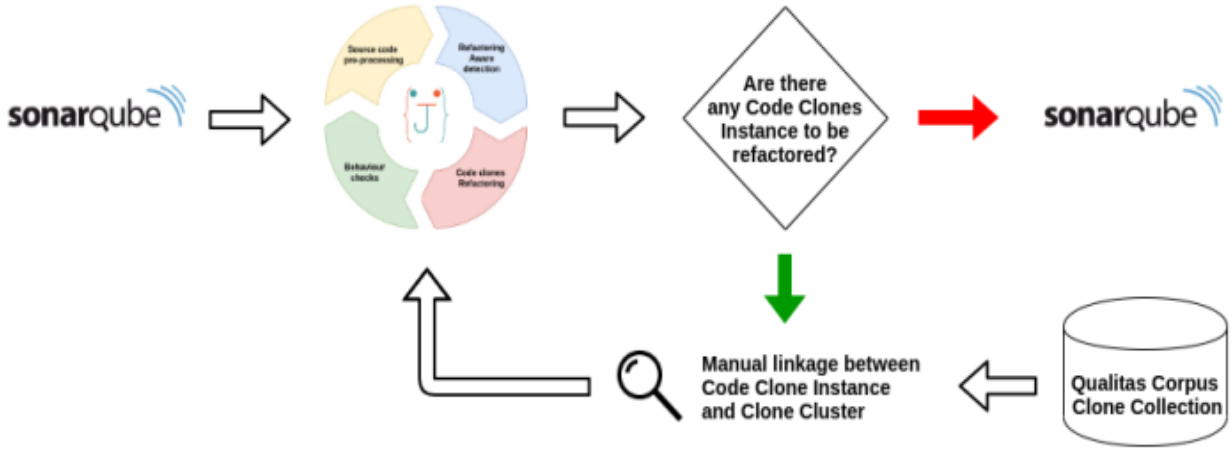

Рисунок 3.10 - Потік, що використовується для оцінки ефективності Janus

Параметри, що використовуються під час процесу рефакторингу, наведено в таблиці 3.3.

Таблиця 3.3

Налаштування Януса, що використовувалися для експерименту

Категорія	Параметр	Значення
Janus Flow	Кількість ітерацій (Iteration of Janus Flow)	50
Preprocessing	Включити тестування (Include Test)	false
RAD	Кількість ітерацій (Iteration)	50
	NoCS	33
	ADCL	33
	ANoC	33
	Позиція коду (CodePosition)	80

Обговорення результатів

Процес оцінювання, було виконано для двох різних програмних систем, а саме JPF та JFimDataMath. Отримані результати наведено в таблиці 3.4.

Таблиця 3.4

Результати оцінки всього потоку Janus

Проєкт (Project)	Версія	Виявлено кластерів клонів (%)	Автомати-чно рефакто-рено (%)	Дублікати LoC (до)	Дублікати LoC (після)
JPF	1.5.1	72%	39,5%	434	386
JFin_DataMath	1.0.1	76,9%	53,8%	236	65

Перший аспект, який слід підкреслити, полягає в тому, що не всі кластери клонів були виявлені. Це пов'язано зі способом, яким здійснюється виявлення: фактично, кластер клонів, включений до колекції клонів Qualitas Corpus, був виявлений за допомогою підходу на основі AST [12]. Зокрема, AST кожного методу порівнювалися з іншими, і всі піддерева, нормалізована різниця яких була нижчою за заздалегідь визначений поріг, вважалися клонами. Цей підхід справді дуже ефективний, але він не дозволяє контролювати, які типи відмінностей приймаються. Наприклад, деякі клони в колекції містять оператор **return** або відрізняються для класу змінної, що використовується в певній точці, і їх не можна легко рефакторувати.

```
public int sumFoo(List<Foo> foos) {
    int sum = 0;
    for(Foo foo: foos){
        sum = sum + foo.getValue();
    }
    return sum
}
```

```
public int sumBar(List<Bar> bars) {
    int sum = 0;
    for(Bar bar: bars){
        sum = sum + bar.getValue();
    }
    return sum
}
```

Рисунок 3.11 - Клон коду, який відрізняється для класу змінної

Приклад показано на рисунку 3.11, де можна побачити, що фрагменти коду справді є клонами коду, а відмінності стосуються класів, якими вони маніпулюють.

RAD навмисно не здатний виявляти ці клони коду, і ця різниця враховується у функції подібності змінних. RAD не нормалізує такі відмінності, оскільки «вилучити» клони коду не є тривіальним, оскільки вони залежать від різних класів. Фактично, існує лише два способи виконання вилучення:

• перевірити, чи мають класи спільний суперклас, який вже реалізує метод “getValue()” . Однак, Janus ще не може виконати таку операцію;

• або використовувати клас java.lang.Object як загальний суперклас, а потім виконувати перетворення на нижчі типи, яке необхідно захистити за допомогою ключового слова instanceof . Однак ця операція перешкоджає читабельності та зручності обслуговування коду.

Таким чином, використання колекції клонів корпусів Qualitas як базової інформації дозволило краще оцінити RAD, висвітливши поточні обмеження. Водночас RAD вже здатний отримувати конкурентоспроможні результати при застосуванні до реальних проектів.

Такий самий висновок можна зробити і для кластера-клону, який пройшов автоматичний ре факторинг [39]. Фактично, навіть якщо автоматизовано лише три методи рефакторингу, вони дають переконливі результати при застосуванні до реальних проектів.

Крім того, цікаво відзначити, що, враховуючи всі автоматично застосовані методи рефакторингу, метод вилучення використовувався приблизно у 80% випадків, тоді як метод вилучення та підтягування та рефакторинг суперкласу вилучення складають лише по 10% випадків кожен.

Більше того, завдяки безпеці впроваджених на даний момент методів рефакторингу, Janus зміг виконати рефакторинг без будь-яких змін у поведінці та без появи дефектів, вразливостей чи неприємних запахів коду.

Зрештою, різниця в дублікатах рядків коду до та після рефакторингу підкреслює, що Janus вже здатний впливати на дублювання в рамках проєкту.

Однак помітна відсутність кореляції між відсотком рефакторованих кластерів клонів та видаленими рядками коду. Це головним чином пов'язано з тим, що частота та довжина клонів коду вважаються однаково важливими, як

показано в таблиці 3.4. Таким чином, значна кількість дуже малих клонів коду може мати менший вплив, ніж довгий фрагмент коду, продубльований лише двічі за весь проект, що стосується метрики дубльованих рядків коду.

Тим не менш, можна стверджувати, що рефакторинг значної кількості дуже малих клонів коду може насправді забезпечити ефективніший спосіб покращення зручності підтримки, оскільки чим більша кількість дублікатів екземплярів, тим важче підтримувати їх узгодженість.

Саме тому для оцінки ефективності Janus було використано два показники, і це також підтверджує рішення забезпечити зручний спосіб впливу на таке рішення під час процесу виявлення.

Головною метою цієї роботи була розробка інструменту для автоматизованого рефакторингу клонів коду. Однак у літературі існує консенсус щодо того, що не всі клони коду слід рефакторувати. Тому інструмент, спрямований на автоматизацію рефакторингу клонів коду, потребує алгоритму виявлення, здатного оцінити ризики, пов'язані з рефакторингом заданого набору клонованих операторів, та порівняти їх з очікуваним покращенням з точки зору зручності обслуговування, яке буде досягнуто за допомогою рефакторингу.

Перша пропозиція таких алгоритмів виявлення, що отримала назву «Виявлення з урахуванням рефакторингу» (RAD). Зокрема, проблему виявлення було розглянуто як проблему оптимізації, де враховуються дві основні цілі: функція подібності та функція ризику рефакторингу. Функція подібності має на меті розрізняти різні види клонів коду, дозволяючи користувачеві вказувати найцінніші клони для рефакторингу (з його точки зору).

Крім того, мета полягала не лише у впровадженні автоматизованих методів рефакторингу, а й у автоматизації всього процесу рефакторингу, який включає автоматизоване виконання тестів після кожного кроку рефакторингу та версіонування змін.

Валідація цього робочого процесу, була розділена на дві частини. Перша має на меті довести, що користувач може фактично впливати на виявлення, та перевірити здатність RAD сходитися до змістовного рішення. Такий спосіб

оцінки Janus було проілюстровано, а більшість результатів.

Друга частина стосується оцінки ефективності всього робочого процесу, тобто оцінки фактичного впливу Janus на підтримку реальної кодової бази..

Обидві стратегії валідації виявили деякі обмеження алгоритмів , що використовуються на даний момент, які вже обговорювалися у відповідних розділах і узагальнені. Однак, досягнуті на сьогодні результати підкреслюють, що основні цілі були досягнуті: зокрема, користувач вже може впливати на виявлення, навіть якщо властивості клонів, що надаються на даний момент для керування виявленням, є лише невеликою підмножиною тих, які розробник хотів би використовувати [40]. Крім того, Janus вже може впливати на якість програмної системи з точки зору зручності обслуговування, навіть якщо наразі реалізовано лише три методи рефакторингу.

Загрози валідності

Слід зазначити кілька аспектів, які становлять загрозу валідності цієї роботи. Ці аспекти поділяються на три категорії: *(i)* загрози внутрішній валідності, в яких окреслюються можливі покращення, яких можна досягти, *(ii)* загрози зовнішній валідності, в яких обговорюється узагальнюваність отриманих результатів, і, нарешті, *(iii)* обмеження експериментів.

Загрози внутрішній валідності

Перша загроза валідності RAD випливає з обмежень еволюційних алгоритмів, які не гарантують збіжності до глобального оптимуму. Однак, вже було зазначено, що для цього застосування можна прийняти збіжність до локального оптимуму. Крім того, наведені досі результати підкреслюють, що RAD вже здатний сходитися до повного рішення, тобто до набору методів, які фактично мають значну кількість спільного коду, що вартий рефракторингу.

Крім того, як функцію подібності, так і функцію ризику рефакторингу можна покращити в кількох напрямках. Фактично, що стосується останньої функції, існує кілька аспектів, пов'язаних, наприклад, з «релевантністю» методу, яку можна визначити з точки зору кількості методів, що викликають цей метод, або кількості значень констант, що використовуються в методах. Всю цю

інформацію можна використовувати для кращої оцінки впливу рефакторингу на програмний проект. Також функціональну подібність можна покращити, враховуючи семантичну інформацію, корисну для виявлення клонів коду типу IV.

Більше того, поточні обмеження автоматизованих методів рефакторингу здебільшого стосуються невеликої кількості вже реалізованих методів та поточних обмежень, пов'язаних з відмінностями в клонах коду, які Janus здатний вирішувати автоматично. Однак було підкреслено, що навіть такі обмежені методи рефакторингу вже здатні суттєво впливати на підтримку програмного забезпечення в реальній кодовій базі.

Зрештою, що стосується етапу попередньої обробки, то існують два основні обмеження. Перше – це ступінь деталізації виявлення клонів коду. Фактично, наразі ступінь деталізації виявлення фіксована на рівнях операторів першого рівня. Це пов'язано з тим, що розширення рівня деталізації на глибші рівні викличе кілька додаткових проблем, пов'язаних з продуктивністю, а також з обмеженнями щодо етапу рефакторингу. Наприклад, рефакторинг набору операторів другого рівня може призвести до неможливості рефакторингу набору операторів першого рівня. Тому для вирішення таких проблем слід запровадити стратегію пріоритезації.

Друге обмеження слід розглядати як загрозу ефективності, а не валідності цієї роботи, і воно пов'язане з відсутністю диференціального навантаження. Це означає, що на кожній ітерації весь програмний проект завантажується з нуля. Причиною такого вибору дизайну є можливість взаємодії користувача з кодовою базою *під час* процесу рефакторингу. Це означає, що теоретично кожен клас може бути змінений між одним кроком рефакторингу та наступним. Ця функція дуже важлива, оскільки дозволяє користувачеві застосовувати складніші методи рефакторингу, які може лише спростити Janus, але не повністю впоратися з ними.

Загрози зовнішній валідності

Основна загроза узагальнюваності результатів пов'язана з високою гетерогенністю вихідного коду. Це стосується як здатності генетичного алгоритму сходитися до рішення, так і можливості рефакторингу вибраних фрагментів

коду.

Фактично, на здатність генетичного алгоритму до збіжності впливає наявність спеціальних типів клонів коду, наприклад, пов'язаних з тестовими випадками та інтерфейсом користувача. Саме тому під час етапу попередньої обробки тестові випадки можуть бути виключені з аналізу. Крім того, навіть якщо Janus не в змозі розрізнити код, який реалізує графічний інтерфейс користувача, користувач все одно може вказати кілька фільтрів за допомогою файлу *.janusignore*.

З іншого боку, етап рефакторингу зіткнеться з проблемами через особливі або нещодавно введені структури коду, такі як лямбда-функції, які нелегко рефакторувати.

Вищезгадані проблеми можуть мати більший чи менший вплив, залежно від сфери розробки програмного проекту та способу розробки коду.

Обмеження експериментів

Перше важливе експериментальне обмеження стосується використання лише програмного забезпечення з відкритим вихідним кодом для валідації, яке зазвичай має високу якість. Крім того, наголошено на поширеності практики «копіювання та вставки» в промисловому програмному забезпеченні, ймовірно, для того, щоб полегшити роботу новачкам, які можуть бути не знайомі з архітектурою та використовуваними бібліотеками. Ця практика дійсно генерує дуже значну кількість клонів коду, які Janus може рефакторувати, що рідко можна знайти у програмному забезпеченні з відкритим вихідним кодом.

Крім того, проекти, що використовуються для оцінки всього робочого процесу Janus, є відносно невеликими. Цю властивість необхідно забезпечити, щоб зробити ручне зв'язування між клонами коду.

Екземпляри та кластери клонів доцільні, проте Janus, швидше за все, досяг би кращих результатів у великих проектах головним чином з двох причин. По-перше, тому що стає все складніше вручну відстежувати клони коду зі збільшенням його обсягу, а по-друге, тому що більше коду означає більше функціональності, яку можна було б корисно клонувати для реалізації нових функцій.

Майбутній розвиток

Основні майбутні розробки в основному пов'язані з вирішенням поточних загроз внутрішній валідності. Тому основним удосконаленням, пов'язаним з етапом попередньої обробки, є реалізація диференціального навантаження, яке включатиме постійний моніторинг усіх файлів, включених до аналізованих проектів, що може бути полегшено етапом версіонування. Крім того, має бути забезпечена можливість рефакторингу клонованих операторів на довільному рівні, навіть якщо це може призвести до пошуку додаткового компромісу між кількістю клонів, які можна рефакторувати, та часом, необхідним для виконання ітерації.

Що стосується RAD, то найкориснішим покращенням була б інтеграція кількох додаткових властивостей клонів, які користувач може використовувати для керування виявленням відповідно до своїх потреб. Однак ці аспекти можуть відрізнятися залежно від програмної сфери та способу роботи команди розробників. Тому найкращим способом визначити пріоритети властивостей клонів, які слід реалізувати, є звернення безпосередньо до них. Отже, одним дуже корисним майбутнім розвитком було б створення опитування, пов'язаного з аспектами, які розробники найбільше враховують під час рефакторингу клонів коду, яке слід надіслати компаніям, основним продуктом яких є програмні проекти.

Крім того, може бути корисним підійти до виявлення з урахуванням рефакторингу зовсім іншим чином, ніж запропоновано в цій роботі. Прикладом може бути використання алгоритму машинного навчання, навченого на наборі даних, в якому кожен екземпляр є клоном коду, і користувач може оцінити серйозність кожного екземпляра.

Більше того, можна було б впровадити більше методів рефакторингу, щоб збільшити вплив Janus на зручність підтримки програмного забезпечення. З цієї причини Janus повинен мати змогу обробляти конфлікти константних значень, проте автоматизований рефакторинг цих клонів коду може одночасно вплинути на якість програмного забезпечення з іншої точки зору, наприклад, створення вилучених методів, на які впливає запах коду Long Paramenter List [23], оскільки константи слід передавати як параметри.

Більше того, з Janus можна інтегрувати кілька додаткових систем керування версіями. Інші розширення включають можливість запуску користувацьких скриптів, які компанія може використовувати для тестування коду. Цей другий напрямок майбутнього розвитку може включати інтеграцію Janus з

процесами безперервної інтеграції, як для перевірки того, що рефакторована програмна система не порушує інтеграцію, так і для автоматичного рефакторингу кодової бази як кроку процесу.

3.4 Висновок по розділу

У третьому розділі було розглянуто практичну реалізацію та оцінку ефективності методів зменшення дублювання коду. Проведено аналіз і виявлення клонів коду з урахуванням застосування методів рефакторингу. Досліджено підходи до автоматизованого рефакторингу та оптимізації структури програмного коду, що дозволяють підвищити його зрозумілість і підтримуваність. Також виконано оцінку результатів та валідацію ефективності запропонованих підходів. Отримані результати підтверджують доцільність використання автоматизованих методів виявлення та усунення дублювання коду для покращення якості програмного забезпечення.

ВИСНОВКИ

У ході виконання дипломної роботи було досліджено методи зменшення дублювання коду під час розробки програмного забезпечення, що є важливим аспектом підвищення якості, підтримуваності та ефективності сучасних програмних систем. Проведене дослідження охоплює аналіз причин виникнення дублювання коду, вивчення підходів до його виявлення, а також реалізацію методів усунення за допомогою рефакторингу та автоматизованих інструментів. У результаті виконано комплексну роботу, спрямовану на оптимізацію структури програмного коду та зниження технічного боргу.

На першому етапі було розглянуто теоретичні основи дублювання коду та визначено його негативний вплив на якість програмного забезпечення. Зокрема, встановлено, що наявність клонів коду ускладнює підтримку системи, підвищує ймовірність виникнення помилок і призводить до неузгодженості змін. Досліджено основні типи клонів коду та методи їх виявлення, що дозволило сформувати базу для подальшого аналізу та практичної реалізації.

У другому розділі було проаналізовано сучасні методи та інструменти зменшення дублювання коду. Особливу увагу приділено автоматизованим підходам до виявлення клонів, попередній обробці вихідного коду та використанню спеціалізованих інструментів аналізу. Розглянуто практики впровадження рішень, що дозволяють зменшити дублювання на ранніх етапах розробки, а також підвищити ефективність командної роботи.

У третьому розділі реалізовано підходи до виявлення та усунення дублювання коду, зокрема із застосуванням методів рефакторингу. Проведено оцінку ефективності запропонованих рішень, яка показала, що використання автоматизованих інструментів у поєднанні з рефакторингом дозволяє значно зменшити кількість клонів коду, покращити читабельність та структуру програмного забезпечення. Також було здійснено валідацію отриманих результатів, що підтвердила доцільність застосування досліджених методів.

Загалом, результати роботи свідчать про те, що систематичне

використання методів виявлення та усунення дублювання коду є необхідною умовою для створення якісного програмного забезпечення. Застосування рефакторингу, принципів повторного використання коду та сучасних інструментів аналізу дозволяє знизити технічний борг, підвищити гнучкість системи та спростити її подальший розвиток.

Разом із тим, слід зазначити, що ефективність зменшення дублювання коду залежить від правильної організації процесу розробки, рівня кваліфікації розробників та вибору інструментів. У майбутньому доцільно розширити дослідження шляхом інтеграції методів машинного навчання для автоматичного виявлення складних структурних клонів, а також впровадження більш глибокого аналізу якості коду в процеси безперервної інтеграції та розгортання.

Таким чином, поставлена мета роботи досягнута, а отримані результати можуть бути використані для підвищення ефективності розробки програмного забезпечення та покращення якості програмних продуктів.

СПИСОК ПОСИЛАНЬ НА ДЖЕРЕЛА

1. Fowler, M. (2018). *Refactoring: Improving the Design of Existing Code (2nd ed.)*. martinfowler.com. https://martinfowler.com/books/refactoring.html
2. Martin, R. C. (2017). *Clean Architecture*. pearson.com. https://www.pearson.com/store/p/clean-architecture
3. Martin, R. C. (2008). *Clean Code*. pearson.com. https://www.pearson.com/store/p/clean-code
4. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns*. pearson.com. https://www.pearson.com/store/p/design-patterns-elements-of-reusable-object-oriented
5. Freeman, E., & Robson, E. (2021). *Head First Design Patterns*. oreilly.com. https://www.oreilly.com/library/view/head-first-design/9781492077992/
6. Hunt, A., & Thomas, D. (2019). *The Pragmatic Programmer*. pragmaticprogrammer.com. https://pragprog.com/titles/tpp20/
7. Kerievsky, J. (2004). *Refactoring to Patterns*. pearson.com. https://www.pearson.com/store/p/refactoring-to-patterns
8. Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. martinfowler.com. https://martinfowler.com/eaaCatalog/
9. Spinellis, D. (2006). *Code Quality: The Open Source Perspective*. addison-wesley.com
10. McConnell, S. (2004). *Code Complete*. microsoftpressstore.com
11. Roy, C. K., & Cordy, J. R. (2007). A Survey on Software Clone Detection Research. queensu.ca
12. Kamiya, T., Kusumoto, S., & Inoue, K. (2002). CCFinder: A Multilinguistic Token-Based Code Clone Detection System. IEEE
13. Baxter, I., et al. (1998). Clone Detection Using Abstract Syntax Trees. ICSM
14. Koschke, R. (2007). Survey of Research on Software Clones. dagstuhl.de

15. Bellon, S., et al. (2007). Comparison and Evaluation of Clone Detection Tools. IEEE

16. SonarQube Documentation. (2025). sonarsource.com. https://docs.sonarsource.com/

17. PMD Source Code Analyzer. (2025). pmd.github.io. https://pmd.github.io/

18. Checkstyle Documentation. (2025). checkstyle.org. https://checkstyle.org/

19. ESLint Documentation. (2025). eslint.org. https://eslint.org/docs/latest/

20. ReSharper Documentation. (2025). jetbrains.com. https://www.jetbrains.com/resharper/documentation/

21. DRY Principle. (2024). deviq.com. https://deviq.com/principles/dont-repeat-yourself

22. SOLID Principles. (2024). deviq.com. https://deviq.com/principles/solid

23. Refactoring Guru. (2025). refactoring.guru. https://refactoring.guru/

24. Code Smells Guide. (2025). refactoring.guru/smells

25. Microsoft .NET Code Analysis. (2025). learn.microsoft.com. https://learn.microsoft.com/en-us/dotnet/fundamentals/code-analysis

26. GitHub Code Review Guidelines. (2025). docs.github.com. https://docs.github.com/en/pull-requests

27. Git Documentation. (2025). git-scm.com. https://git-scm.com/docs

28. Continuous Integration (CI/CD). (2025). learn.microsoft.com. https://learn.microsoft.com/en-us/devops

29. DevOps Practices Guide. (2024). atlassian.com. https://www.atlassian.com/devops

30. Fowler, M. (2014). Microservices Architecture. martinfowler.com. https://martinfowler.com/microservices/

31. Newman, S. (2021). *Building Microservices*. oreilly.com

32. Evans, E. (2003). *Domain-Driven Design*. addison-wesley.com

33. Stack Overflow Developer Survey. (2024). stackoverflow.co

34. IEEE Software Engineering Standards Overview. (2024). ieee.org

35. ISO/IEC 25010 Software Quality Model. (2023). iso.org

36. Google Engineering Practices Documentation. (2025). google.github.io/eng-practices/

37. Clean Code JavaScript. (2024). github.com/ryanmcdermott/clean-code-javascript

38. Code Duplication Metrics Study. (2023). ACM Digital Library. https://dl.acm.org

39. Static Code Analysis Techniques. (2024). IEEE Xplore

40. Software Maintainability and Reuse. (2023). SpringerLink.

БІБЛІОГРАФІЧНА ДОВІДКА

Тема бакалаврської роботи: "Методи зменшення дублювання коду під час розробки"

Обсяг пояснювальної записки: 74 аркушів

Дата закінчення бакалаврської роботи 10 червня 2026р.

Підпис студента _____