Міністерство освіти і науки України

Харківський національний університет радіоелектроніки

Факультет ________________________Комп'ютерних наук______________________

(повна назва)

Кафедра ________________________Програмної інженерії_______________________

(повна назва)

АТЕСТАЦІЙНА РОБОТА
Пояснювальна записка

_____________другий (магістерський)_____________

(рівень вищої освіти)

Дослідження моделей та засобів генерації діаграм об'єктів та класів для оцінювання якості розробки

(тема)

Виконав: студент 2 курсу, групи ІПЗм–17–2___
спеціальності 121– Інженерія програмного забезпечення
(код і повна назва спеціальності)

Освітньо–професійної програми

_Інженерія програмного забезпечення______________

________________________Остапенко Д.С.__________

(прізвище, ініціали)

Керівник ___________проф. Єрохін А,Л.___________

(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри, проф. _____________________ З.В.Дудар

2019 р.

Харківський національний університет радіоелектроніки

Факультет Комп’ютерних наук

Кафедра Програмної інженерії

Рівень вищої освіти другий (магістерський)

Спеціальність 121–Інженерія програмного забезпечення
(код і повна назва)

освітньо–наукова програма Інженерія програмного забезпечення
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри ______________
(підпис)
«_____»________________ 20 ___ р.

ЗАВДАННЯ
НА АТЕСТАЦІЙНУ РОБОТУ

студентові Остапенко Денису Сергійовичу
(прізвище, ім’я, по батькові)

1. Тема роботи Дослідження моделей та засобів генерації діаграм об'єктів та класів для оцінювання якості розробки
затверджена наказом по університету від “___” _______20 ___ р № _____
заповнюється вручну після отримання наказу

2. Термін подання студентом роботи до екзаменаційної комісії
08 червня 2019 р.

3. Вихідні дані до роботи *алгоритми машинного навчання, абстрактні синтаксичні дерева. Використовувати ОС X середовище, діаграма класів та об’єктів, дерева прийняття рішень, метрики якості програмного коду*

4. Перелік питань, що потрібно опрацювати в роботі *мета роботи, аналіз проблемної галузі і постановка задачі характеристика математичних метрик дослідження якості програмного коду , алгоритми реорганізації програмного коду, існуючі рішення автоматичного реорганізування коду, використання дерев прийняття рішень для оцінки якості*

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп’ютерних ілюстрацій (слайдів) *Мета завдання, обґрунтування доцільності розроблення, постановка задачі, об'єктна модель системи, базові моделі, інтерфейс програмної системи, результати тестування програмної системи, демонстраційні матеріали*

Зворотний бік бланку завдання

6. Консультанти розділів роботи:

Найменування розділу	Консультант (посада, прізвище, ім’я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата
Спецчастина	проф. Єрохін А.Л.		

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка*
1.	Аналіз предметної галузі		
2.	Огляд існуючих рішень		
3.	Методи дослідження якості програмного коду		
4.	Підготовка пояснювальної записки		
5.	Спецчастина		
6.	Підготовка презентації та доповіді		
7.	Попередній захист		
8.	Нормоконтроль, рецензування		
9.	Занесення роботи в електронний архів		
10.	Допуск до захисту у зав. кафедри		
* заповнюється вручну після виконання чергового пункту			

Дата видачі завдання _____ ______________20___ р.

Студент ____

(підпис)

Керівник роботи __________________________ ______проф. Єрохін А.Л.________ _

(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ / ABSTRACT

Пояснювальна записка до атестаційної роботи: ___ с., ___ рис., ___ джерел, ___ таблиць.

Метою роботи є дослідження моделей та засобів генерації діаграм об'єктів та класів для оцінювання якості розробки у середовищі JavaScript.

Методи розробки базуються на технології JavaScript, та методах і принципах машинного навчання.

У результаті роботи розроблено програмну реалізацію, ціллю якої є оцінка якості коду та надання візуального відображення у вигляді UML діаграм, які описують порушення архітектурних принципів та надають поради для покращення якості кодової бази.

ПРОГРАМНИЙ КОД, АБСТРАКТНЕ СИНТАКСИЧНЕ ДЕРЕВО, МАШИНЕ НАВЧАННЯ, ДЕРЕВО ПРИЙНЯТТЯ РІШЕНЬ, JAVA SCRIPT, ЯКІСТЬ КОДУ.

The goal of this project is to explore models and means of generation of class and objects diagrams for assessing the quality of software development in JavaScript environment.

The developing methods and based on JavaScript programming language and Machine learning rules, methods and principles.

The result of developing is CLI utility aimed on assessing the quality of the code and providing visual representation which consists of UML diagrams, which illustrate violation of architectural principles and give pieces of advice for improving the quality of codebase.

SOFTWARE CODE, ABSTRACT SYNTAX TREE, MACHINE LEARNING, DECISIONS TREE, JAVA SCRIPT, CODE QUALITY.

ЗМІСТ

ВСТУП 6

1 Аналіз предметної галузі 8

1.1 Загальна характеристика якості програмного коду 8

1.2 Цикломатична складність коду 9

1.3 Індекс підтримки 13

1.4 Аналіз існуючих рішень 16

1.4.1 Утиліта ESlint 16

1.4.2 Ресурс deepcode.ai 18

1.5 Постановка задачі 20

2 Методи дослідження та реорганізування коду 22

2.1 Виявлення патернів проектування в об’єктно–орієнтованому середовищі 22

2.2 Виявлення помилок проектування в об’єктно–орієнтованому середовищі 24

2.3 Дерева прийняття рішень 27

3 Описання програмної реалізаціії 31

3.1 Парсинг вхідного програмного коду 31

3.2 Пошук оптимальних моделей 34

3.3 Реорганізація коду за допомогою системи машинного навчання 36

3.4 Побудова діаграм класів та об’єктів 39

4 Автоматичний рефакторинг та оцінка якості 41

4.1 Використання архітектурного шаблону проектування «Будівельник» 42

4.2 Пошук дуплікацій у програмному коді 49

4.3 Рефакторинг програмного коду на основі власного дерева прийняття рішень від користувача. 54

Висновки 58

Перелік посилань 59

Додаток А Слайди презентації 61

Додаток Б Лістинг коду дерева прийняття рішень 75

Додаток В Наукові публікації 85

Додаток Г Електронні матеріали (CD) 92

ВСТУП

Методи машинного навчання активно вдосконалюються протягом останніх десятиліть і знайшли своє застосування в різних технологічних областях, наприклад, в медицині, економіці, комп'ютерному зорі, біоінформатиці, важкій промисловості, інформаційному пошуку і т.д. Однак, на сьогоднішній день все ще існує безліч таких завдання та викликів, з якими жива людина справляється набагато краще навченого алгоритму. Одне з таких напрямків є – аналіз програмного коду.

Говорячи про аналіз програмного коду, слід зазначити що, це є трудомісткий, затратний з боку часових ресурсів, процес. У даній роботі під «аналізом програмного коду» розуміється дослідження його якості, а саме використання певних метрик, що дають змогу математично оцінити наданий програмний код, а також дослідження доцільності використання деяких шаблонів проектування у проектування, заміняючи менш якісні ділянки коду.

На жаль, в наш час машина не може думати та роботи глибокі, зважені рішення, які зазвичай робить розробник програмного забезпечення, тому не слід очікувати принаймні в найближчому майбутньому, що натреновані моделі зможуть писати самостійні модулі або навіть цілі працюючі програмні додатки, тим самим практично повністю замінюючи реальних розробників, це підтверджує те, що написання програмного коду та дизайн програмних додатків є в якомусь плані творчим процесом, тому в деяких рішеннях машинному алгоритму буде дуже, а іноді неможливо знайти закономірність. Проте, технології правила та методи машинного навчання достатньо розвинуті в наші часи, щоб забезпечети розробників усім необхідним, для створення інтегрованої системи дослідження якості коду та його автоматичної реорганізації для підвищення метрик якості та слідкування загальноприйнятим архітектурним шаблонам, правилам та практикам написання якісного, тестованого, підтримуваного, розширюваного та зрозумілого іншим програмного коду.

Можна виділити кілька конкретних завдань, пов'язаних з аналізом коду, в яких може бути застосовано машинне навчання:

– виявлення та усунення однакових частин (дуплікацій) логіки в програмному коді;

– виявлення та виправлення глибоких та внутрішніх помилок архітектури та проектування.

– виявлення та дослідження використання тих чи інших шаблонів та усунення анті–патернів проектування;

– виявлення та виправлення загальних, специфічних до мови програмування помилок у коді;

– надання можливості створювати свої власні стандарти коду та автоматично реорганізовувати код згідно них;

– надання порад щодо покращення якості програмного коду.

Об'єктно–орієнтований код є найцікавішим для досліджень, так як цей підхід розробки зараз є найбільш масово розповсюдженим. Крім того, об'єктно–орієнтований код має відносно нескладну структуру, і його компоненти легко піддаються аналізу. Помилки проектування при розробці програмних продуктів ускладнюють повторне використання і подальший супровід. На виправлення таких помилок може бути витрачено багато часу і сил, тому можна говорити про наявність ніші для автоматизованих алгоритмів та моделей дослідження і покращення якості розробки.

Ще слід зазначити, що об'єктно–орієнтований код є гарним ресурсом для візуальної обробки, тому дослідження якості, виявлення порушень правил проектування та архітектурних принципів, надання порад щодо покращення кодової бази може бути з легкістю представлено у вигляді зручних та детально описаних діаграм класів та об’єктів.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Загальна характеристика якості програмного коду

Постійна зростаюча конкуренція на ринку програмного забезпечення, нестійке положення самого ринку і необхідність постійно впроваджувати нові продукти, послуги та усувати існуючі дефекти для клієнтів, спонукають компанії пришвидшувати темпи автоматизації та використовувати різноманітне за походженням і якістю виконання ПЗ [1]. Водночас все актуальнішою постає проблема ризиків, які виникають унаслідок неякісного ПЗ. Мова йде як про цілеспрямовані атаки злочинного походження, так і збої в роботі ПЗ, які виникають внаслідок ненавмисних помилок у програмному коді чи є наслідком інших недоліків при розробці ПЗ [2]. Таким чином, існує актуальна задача оцінки якості програмного коду для подальшого його вдосконалення чи заміни альтернативними рішеннями вищої якості [3]. 49 Незважаючи на активну роботу, яка проводиться вченими у напрямку вирішення проблеми оцінки якості програмного коду, задача в цілому не є вирішеною. Існуючі методики часто не дають цілісної картини про реальний стан коду, різні показники нерідко показують протилежні та неузгоджені результати [1]. Все зазначене вище обумовлює актуальність удосконалення підходів до оцінки якості програмного коду, формування цілісної картини по певному проекту, його придатності для використання практичних задач, а також напрямків для підвищення якості. Особливої актуальності в даному контексті набуває проблема моніторингу та аналізу дефектів у програмному коді. А також необхідний засіб, який реалізую даний функціонал.

Якість програмного коду залежить від декількох аспектів:

– першим критерієм якості як вже було зазначено є так звана функціональна якість, яка дає характеристику того, наскільки повно написаний програмний код відповідає описаним функціональним вимогам та розробленим специфікаціям;

– другим та більш важливим критерієм для цієї роботи є структурна якість, яка відноситься до нефункціональних вимог, таких як: надійність, легкість у

підтримці, перспектива до розширення та швидкодія, більшість цих характеристики можуть буть обчислені через внутрішній аналіз структури кодової бази та її компонентів.

Також ще однією з важливих проблем у процесі написання якісного коду та рефакторингу існуючого є відносний показник який показує наскільки легко сторонній розробник в змозі читати та розуміти написаний програмний код, тому слід вводити таке поняття, як «складність коду». Одним з найперших вимірів складності коду був розроблений Томасом Маккейбом у 1976 році, який носить ім'я цикломатичної складності.

При обчисленні цикломатичної складності використовується граф потоку керування програми. Вузли графа відповідають неподільним групам команд програми, вони з'єднані орієнтованими ребрами, якщо група команд, відповідна другого вузла, може бути виконана безпосередньо після групи команд першого вузла. Ще важливо зазначити, що цикломатична складність може бути також обчислена для окремих функцій, модулів, методів або класів в межах деякої програми.

1.2 Цикломатична складність коду

Цикломатична складність програмного коду – це кількість лінійно незалежних маршрутів через програмний код. Наприклад, якщо вихідний код не містить ніяких точок розгалуження або циклів, то складність дорівнює одиниці, оскільки є тільки єдиний маршрут через код. Якщо код має єдиний оператор IF, що містить просте умова, то існує два шляхи через код: один якщо умова оператора IF має значення TRUE і один – якщо FALSE. Математично цикломатична складність структурованої програми визначається за допомогою орієнтованого графа, вузлами якого є блоки програми, з'єднані ребрами, якщо управління може

переходити з одного блоку на інший. Тоді графа цикломатична складність може бути розрахована згідно приведеної формулі 1.1.

$$M = E - N + 2P \tag{1.1}$$

де M – цикломатична складність;

E – кількість ребер у графі;

N – кількість узлів у графі;

P – кількість компонент зв’язності.

Одне з спочатку запропонованих Маккейбом застосувань формули полягає в тому, що необхідно обмежувати складність програм під час їх розробки. Він рекомендує, те щоб програмістів зобов'язували обчислювати складність розроблюваних ними модулів і розділяти модулі на більш дрібні щоразу, коли цикломатична складність цих модулів перевищить десять. [4] Ця практика була включена Національному Інституті Стандартів і Технологій США в методику структурного тестування з зауваженням, що з часу виходу публікації Маккейба, вибір значення складності у розмірі десять отримав вагомі підтвердження, проте в деяких випадках може бути доцільно послабити обмеження і дозволити модулі зі складністю до п’ятнадцяти одиниць, але при цьому сам Маккейб наполягав у письмовому обґрунтуванні данного рішення.

Ще одним з цікавих фактів є те, що вимірювання цикломатичної складності коду має місце навіть у сьогоднішньому світі розробки програмного забезпечення, про це свідчить використання метрики цикломатичної складності коду у інтегрованому середовищі розробки Microsoft Visual Studio Code, а також у такій системі контролю якості програмного коду як Sonar Qube і слід зазначати, що існує чимало плагінів, які додають цю метрику в інших системах.

Слід зазначити, що чим меншим є цикломатична складність програмного коду тим краще, тому під час розробки слід ретельно стежити за зростанням цього числа та своєчасно виконувати реорганізацію програмних модулей, класів, окремих функцій та блоків, тощо. Короткий аналіз кореляції цикломатичної

складності і аналіз загальних характеристик програмного коду наведено у таблиці 1.1.

Таблиця 1.1 – Кореляція цикломатичної складності програмного коду та його якості.

Цикломатичне число	**Опис значення**
1 – 10	– Гарно – структований код; – Висока тестованість ; – Мала ціна внесення змін до існуючого коду.
10 – 20	– Досить складний код; – Середня тестованість; – Середня ціна внесення змін до існуючого коду.
20 – 40	– Дуже складний код; – Проблемний до тестування; – Висока ціна внесення змін до існуючого коду.
> 40	– Взагалі нетестований та складний для читання розробником код; – Дуже висока ціна внесення змін до існуючого коду.

Згідно наведеної вище таблиці можна повністю запевнитися в переконаннях Маккейба щодо вибору 10 – 15 одиниць для цикломатичної складності як верхньої границі програмних компонентів. Далі на рисунку 1.1 приведений приклад

простого графу, який може представляти потік виконання деякої програми, та підрахунок для нього значення цикломатичного числа.

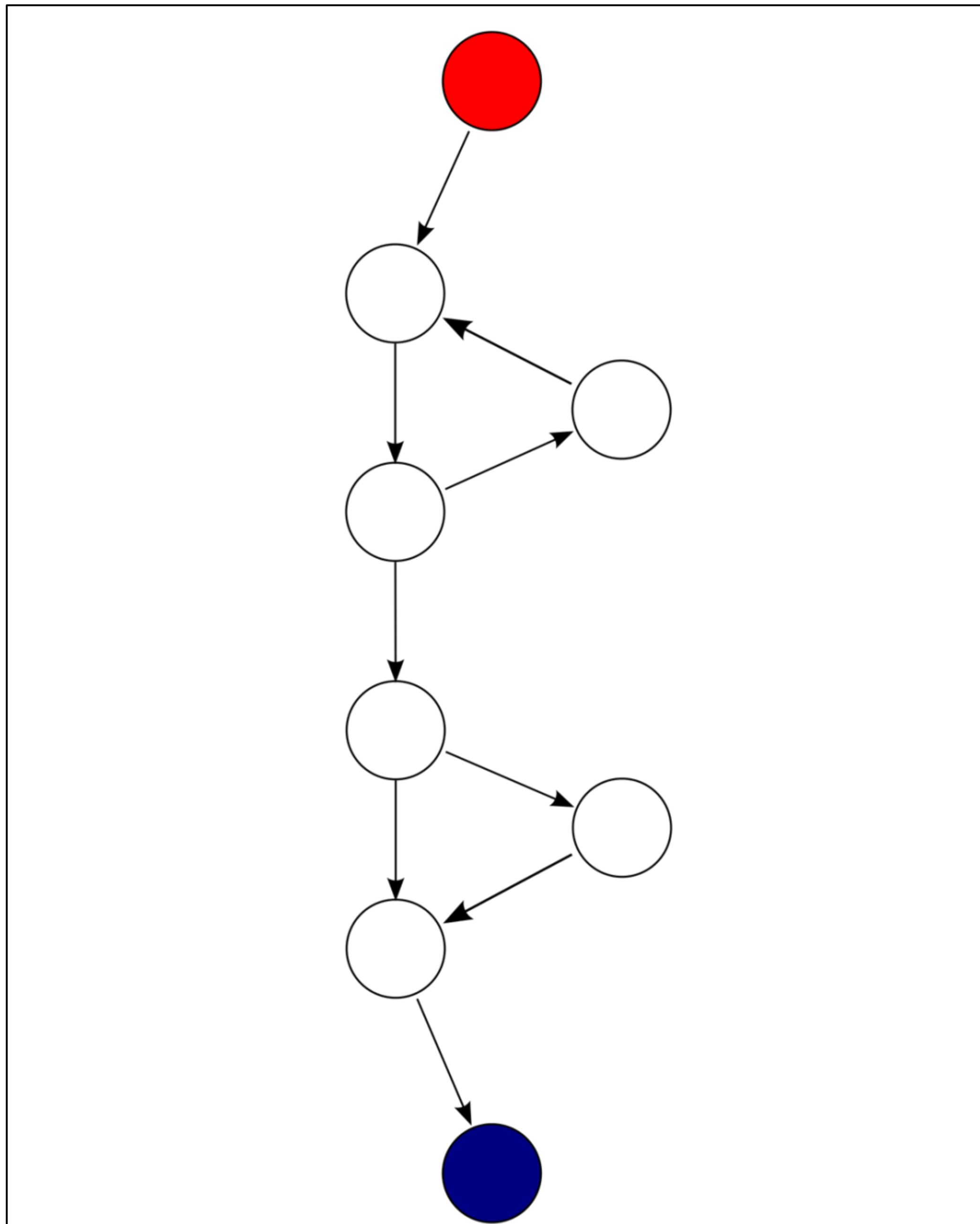

Рисунок 1.1 – Граф управління потоком простої програми

Програма починає свою роботу з червоного вузла, потім йдуть цикли (після червоного вузла йдуть дві групи по три вузла). Вихід з циклу здійснюється через умовний оператор (нижня група вузлів) і кінцевий вихід з програми в синьому вузлу. Для цього графа можна підрахувати, що $E = 9$, $N = 8$ і $P = 1$, тому цикломатична складність даної програми буде дорівнювати $9 - 8 + 2 \times 1 = 3$.

1.3 Індекс підтримки

Супровід (підтримка) програмного забезпечення – це процес поліпшення, оптимізації та усунення дефектів програмного забезпечення після передачі в експлуатацію. Супровід ПО – це одна з важливих фаз життєвого циклу програмного забезпечення, наступна за фазою передачі ПО в експлуатацію. В ході супроводу в програму вносяться відповідні зміни, з тим, щоб виправити виявлені в процесі використання та тесування дефекти і недоробки, а також для додавання нових функцій, з метою зробити програмне запезпечення зручнішим для використання (юзабіліті) і застосовність ПО.

Індекс підтримки – це метрика програмного забезпечення яка обчислює наскільки підтримуваним є програмний код. Індекс підтримки рахується за допомогою складеної формули, яка містить загальну кількість рядків програмного коду, цикломатичну складність и об’эм Халстеда. Ця метрика використовується у декількох автоматизованих інструментах обчислення метрик якості програмного забезпечення, наприклад таких як інтегроване середовище розробки Microsoft Visaul Studio Code.

Існує два види метрики індексу підтримки:

– індекс підтримки з урахуванням коментарів – коли в під рахунок індексу включені коментарі залишені розробниками до деяких частин програмного коду. Цей вид є більш строгим з приводу обчислення, тому що зазвичай розробники залишають багато коментарів, але на жаль, далеко не всі коментарі підтримуються, тому складність та читабельність кінцевого програмного коду значно та неочікувано зростає;

– індекс підтримки без урахування коментарів – є більш поблажливим, тому що фізичне значення кількості рядків коду значно зменшується без урахування рядків з коментарями, але ж слід розуміти, цей метод може бути не таким надійним особливо в старих та складних проектах, тому слід ретельно ставитись до вибору методу.

Важливо зазначити, що чим більше значення індексу підтримки має вхідний програмний код, то можна казати, що більш тестованим, підтримуваним, розширюваним, зрозумілим для інших розробників та взагалі якіснішим є даний код.

Тому індекс підтримки (*MI*) може бути вирахуваний як приведено на формулі 1.2:

$$MI = 171 - 5.2 * \ln(V) - 0.23 * (G) - 16.2 * \ln(LOC) \quad (1.2)$$

де MI – індекс підтримки;

V – величина об'єму Хальстеда;

G – цикломатичну складність деякого модулю / класу / функції;

LOC – фізична кількість рядків в деякому модулі / класі / функції;

CM – відсоток ліній з коментарями у деякому модулі / класі / функції.

Також є ще одна формула, яка використовується в інтегрованому середовищі розробки Microsoft Visual Studio Code, ця формула базується на приведеній вище, але вона є більш досконалою. Visual Studio позначає методи / класи зеленим кольором, якщо значення метрики знаходиться в межах від 20 до 100 одиниць, жовтим кольором, якщо значення знаходиться в межах від 10 до 20 одиниць, і нарешті червоним кольором, коли значення становить менше 10 одиниць. Оптимізоване значення метрики індексу підтримки може бути вирахувано згідно формули 1.3.

$$MI = MAX(0, (171 - 5.2 \times \ln(V) - 0.23 \times (G) - 16.2 \times \times ln(LOC)) \times 100 / 171) \quad (1.3)$$

де MI – індекс підтримки;

V – величина об'єму Хальстеда;

G – цикломатичну складність деякого модулю / класу / функції;

LOC – фізична кількість рядків в деякому модулі / класі / функції;

CM – відсоток ліній з коментарями у деякому модулі / класі / функції.

Далі в таблиці 1.2. наведена кореляція значення індексу підтримки та короткої характеристики властивостей програмного коду, відповідно до отриманого значення метрики індексу підтримки.

Таблиця 1.1 – Кореляція індексу підтримки програмного коду та його якості.

Значення індексу підтримки	**Опис значення**
85 >	– Гарно структурований код; – Висока підтримуваность; – Мала ціна внесення змін до існуючого коду.
65 – 85	– Досить складний код; – Середня підтримуваность; – Середня ціна внесення змін до існуючого коду.
65 <	– Складний, неструктурований код, який місить великі, погано задокументовані функції, та безліч операторів розгалуження.

Коротка кореляція значення індексу підтримки та якості коду дає зрозуміти, що індекс підтримки слід тримати хоча б на рівні 75 одиниць, та чим більше значення індексу підтримки має досліджуваний клас чи модуль – тим більш якісним є код.

Завдяки цьому інструменту можна досить легко провести ревью великого проекту, знайти місця, які необхідно переписати. Також досить корисно стежити за процесом зміни вищеописаних метрик. Це може показати керівнику про ставлення програмістів до розробки того чи іншого проекту, а також динаміку зміни якості коду по кожному програмісту, що важливо в процесі розробки. Іншою причиною

стеження за метриками, є, певні порогові значення, при досягненні яких, необхідно провести рефакторинг.

1.4 Аналіз існуючих рішень

На даний момент проблема оцінювання якості коду є досить вагомою у галузі розробки та підтримування програмного забезпечення і повністю не має універсального, але існує досить багато додатків, які у деякій мірі поліпшують життя розробникам, та наближають до вирішення проблеми автоматизації оцінки та рефакторингу програмного коду. Так як ця дослідницька робота використовує мову програмування JavaScript для розробки та оцінювання якості, то на далі будуть приведено та проаналізовано набір найбільш відомих та популярних інструментів і ресурсів, які першочергово націлені на оцінку програмного коду JavaScript середовища та його компонентів.

1.4.1 Утиліта ESlint

ESlint – це лінтинг утиліта з відкритим вихідним кодом, створена Ніколасом Закасом у червні 2013 року. Лінтування коду – це один з типів статистичного аналізу програмного коду, який повторюється с разу на раз, для того щоб виявити проблематичні паттерни, або програмний код, який не відповідає або порушує деякі визначені заздалегідь стандарти чи практики кодування. Існує дуже багато код лінтеров, за звичай кожна популярна мова програмування має свій лінтер і навіть інколи вони вбудовуються в компілятори, щоб виконувати перевірки під час компіляції.

Мова програмування JavaScript є динамічна та слабо типізована, тому вона є особливо схильна до помилок розробників. Без переваг процесу компіляції

помилки у JavaScript коді за звичай шукаються під час запуску програми (синтаксичний аналіз). ESLint дозволяє шукати помилки, не запускаючи JavaScript додатки

Першочерговою причиною створення утиліти ESlint було надання можливості розробникам створювати свої власні правила для аналізу та рефакторингу коду.

У середині ESlint аналізує абстрактне синтаксичне дерево для виявлення проблем у коді.

Підведемо підсумки щодо використання ESlint. Сильні сторонами даної утиліти є:

– швидкодія. Утиліта Eslint має дуже велику швидкою, що дозволяє аналізувати та проводити реорганізацію великого обсягу програмного коду за лічені секунді;

– можливість до розширення. Використовуючи ESlint розробник має можливість створювати свої власні стандарти кодування, ділитися ними та імпортувати вже існуючі;

– модульність. ESlint може бути дуже легко інтегрований у будь які середовища розробки, або використаний окремо за допомогою Command Line Interface;

– рефакторинг. ESlint дозволяє зразу ж усувати знайдені помилки та порушення правил кодування.

Недоліки:

– за звичай правила ESlint націлені на виправлення маленьких та незначних помилок у програмному коді. Вона не призначена на пошук архітектурних помилок, або загальновідомих стандартів написання чистого, зрозумілого усім коду;

– не має ніяких засобів щодо візуалізації знайдених помилок та зміненого програмного коду.

На рисунку 1.2 наведений приклад коду функцій від одного аргументу «до» и і «після» рефакторингу за допомогою утиліти Eslint.

```
1   function TestFn      (arg1) {
2     var a = arg1 + 1;
3
4     if (a > 5) {
5       if (a % 2 === 0) {
6           return 0;
7       }
8     }
9
10    return 1;
11  }
```

```
1   function TestFn(arg1) {
2     var a = arg1 + 1;
3
4     if (a > 5 && a % 2 === 0) {
5       return 0;
6     }
7
8     return 1;
9   }
10
11
```

Рисунок 1.2 – До і після рефакторингу за допомогою утиліти ESlint

Згідно рисунку можна помітити, що ESlint зміг досить непогано покращити зовнішній вигляд коду, а саме:

– були прибрані непотрібні знаки пробілу у назві функції;

– зміненні береги, щоб відповідати стандартам, які вказані у правилах.

– декларація змінної *var* була замінена на більш нову, яка відповідає сучасному стандарту – *const;*

– оптимізовано оператор розгалуження до більш простого та зрозумілого для інших розробників;

1.4.2 Ресурс deepcode.ai

«Deepcode AI система огляду коду» – є платформою, яка вчиться з мільйонів інших доступних програмних продуктів з відкритою кодової базою. Цей сервіс використовує вже накопиченні знання платформи, для пропонування ключових варіантів, щодо покращення програмного коду кінцевих користувачей. Слід зазначити, що ця система є досить новою розробкою, вона з'явилась у кінці літа 2018 року, та ще знаходиться у активній розробці та модернізації.

Розберемо сильні та слабкі сторони цієї системи. До сильных сторін можна віднести:

– використання бази знань та системи машинного навчання. Завдяки використанню машинного навчання – ця система стає дуже сильним конкурентом будь якому існуючому рішенню у галузі оцінки якості та рефакторингу програмного забезпечення, так як поради щодо рефакторингу коду будуть мати набагато більшу цінність ніж ті, що пропонуються існуючими системи аналізу коду, тому що вони були зібрані на основі практик, запроваджених реальними людьми;

– підтримка багатьох мов програмування. Ця платформа не є націленою тільки на мову програмування JavaScript, вона також підтримує інші популярні мови, такі як Java, C#, GoLang, PHP, тощо.

– інтеграція з системами контроля версій. Deepcode AI надає можливість щодо інтегрування з такими популярними системами контроля версій як Microsoft Github, та Atlassian BitBucket.

Слабкі сторони:

– молода платформа. Ця платформа тільки но з'явилась, тому не слід очікувати дуже багато вдалих та корисних порад щодо рефакторингу;

– платна підписка. Використання платформи можливо тільки після оформлення платної підписки, в залежності від розміру проекту змінюється і ціна, можливо у майбутньому буде надано деякий базовий функціонал безкоштовно;

– не модульне рішення. Платформа Deepcode AI не є модульним рішенням, тому доступ до її можливостей може бути отриманий тільки через використання веб додатку та зв'язаних аккаунтів Atlassian BitBucket та Microsoft Github, це суттєво зменшує кількість кінцевих користувачей, тому що більшість розробників надають перевагу інструментам, які можуть бути вбудовані в їхні найулюбленіші середовища розробки;

– швидкодія. Глибокий аналіз та тренування займає велику кількість часу (за звичай за велику для очікування бізнесом), але на жаль з цим нічого на даний момент зробити не можливо, бо такими системними вимогами є використання нейронних мереж.

На рисунку 1.3 приведено приклад рефакторингу за допомогою Deepcode AI.

```
    private static byte[] macAddress() {
        try {
-           byte[] mac =
    NetworkInterface.getNetworkInterfaces().nextElement().getHa
    rdwareAddress();
+           Enumeration<NetworkInterface> interfaces =
    NetworkInterface.getNetworkInterfaces();
+           byte[] mac = null;
+           while (interfaces.hasMoreElements() && mac != null
    && mac.length != 6) {
+               NetworkInterface netInterface =
    interfaces.nextElement();
+               if (netInterface.isLoopback() ||
    netInterface.isVirtual() ) { continue; }
+               mac = netInterface.getHardwareAddress();
```

Рисунок 1.3 – Рефактороинг коду за допомогою платформи Deepcode AI

Згідно рисунку, описується робота з мережевими адресами, потрібно зробити більш точну перевірку, щоб дізнатися що мережевий інтерфейс, який ми отримали має 6–ти бітний MAC адрес.

1.5 Постановка задачі

Метою даної роботи є застосування методів машинного навчання в області автоматичного рефакторинга об'єктно–орієнтованого коду і проведення відповідних досліджень. Були поставлені наступні завдання:

– провести аналіз предметної області, охарактеризувати існуючі методи дослідження якості коду, навести приклади;

– дослідити вже існуючі рішення та засоби на ринку аналізу та рефакторингу програмного забезпечення, виявити сильні сторони та недоліки;

– вибрати алгоритми і метрики, на основі яких буде здійснюватися реструктуризація;

– реалізувати прототип модульної системи машинного навчання, яка буде виконувати автоматичну реструктуризації та надавати поради щодо покращення загальної якості коду та уникнення серйозних архітектурних проблем при майбутній розробці;

– провести тестування системи використовуючи різноманітні навчальні приклади, які наближені до реальних ситуацій.

– розробити функціонал генерації діаграм класів та об’єктів для підвищення ефективності процесі огляду коду, та надання розробникам програмного забезпечення більше інформації та доказів з приводу виконаного рефакторингу.

2 МЕТОДИ ДОСЛІДЖЕННЯ ТА РЕОРГАНІЗУВАННЯ КОДУ

Існує досить багато різноманітних методів, спрямованих на розбір написаного програмного коду, більшість з них так чи інакше використовує графи чи дерева для проведення аналізу, виявленню порушених зв'язків між компонентами, використання чи навпаки не слідкування усталеним практиками, шаблонам чи правилам проектування. Далі будуть розглянуті декілька основних з них.

2.1 Виявлення патернів проектування в об'єктно–орієнтованому середовищі

За допомогою алгоритмів роботи з нейронними мережами [5], в програмах, написаних на об'єктно–орієнтованих мовах, можуть бути знайдені поширені шаблони проектування [6, 7]. Для реалізації конкретного архітектурного шаблона проектування зазвичай потрібна взаємодія кількох класів, тому процес розпізнавання шаблонів ділиться на два етапи:

– для кожного класу в програмному коді спробувати визначити, яку функцію в кожному з розглянутих архітектурних шаблонів проектування цей клас міг би виконувати;

– аналізуючи найбільш ймовірні комбінації ролей класів, спробувати визначити, чи не утворюють вони в сукупності один з розглянутих архітектурних патернів.

В якості ознак, на основі яких здійснюється навчання нейронної мережі, використовуються наступні характеристики:

– NSF (Number of Static Fields) – описує кількість статичних полів та методів класу;

– NOM (Number of Methods) – описує кількість методів класу;

– NOAM (Number of Abstract Methods) – описує кількість абстрактних методів класу;

– NOPC (Number of Private Constructors) – описує кількість приватних конструкторів;

– NOF (Number of Fields) – кількість оголошених полів в досліджуваному класі;

Також для навчання можуть бути використані більш складні метрики, наприклад виклики класу (Response for Class) [6] або афферентний зв'язок (Afferent Couplings).

Даючи визначення метриці виклику класів простою мовою – це значення можна виразити числом методів класу плюс кількість методів інших класів, що викликають даний клас.

Метрика виклику класу являє собою мірою потенціальної взаємодії цього класу з іншими класами, що дозволяє говорити о динаміці використання цього об'єкту в середовищі. Дана метрика характеризує динамічну складову зовнішніх зв'язків класів.

Якщо у відповідь на повідомлення, може бути викликане велике число методів, то можна казати, що буде суттєво ускладнене тестування і відлагодження класу, так як це вимагає більшого рівня розуміння класу від розробника, а також зростає довжина тестової послідовності.

З ростом значення виклику класу явно збільшується складність програмного класу. Найгірша величина викликів може бути використана при визначенні часу тестування. На формулі 2.1 приведено порядок обчислювання метрики виклику класів.

$$RS = \{M\} \cup_{all\ i} \{R_i\} \quad (2.1)$$

де RS – метрика виклику класу;

$\{R_i\}$ – представляє собою набір сторонніх методів чи функцій, які визиваються за допомогою метода i;

$\{M\}$ – є набором усіх методів у досліджуваному класі.

2.2 Виявлення помилок проектування в об’єктно–орієнтованому середовищі

Ще одне важливе завдання в області аналізу програмного коду об'єктно додатків – пророкування помилок без тестування. Тут можуть бути застосовані різні алгоритми машинного навчання, зокрема, Alternating Decision Tree [8] і LogitBoost [9], який показав найкращі результати [10] у порівнянні з Artificial Neural Networks , Random Forest, Naive Bayes і K–Star [11].

Автоматична реструктуризація програм може здійснюватися на рівні методів, класів або пакетів. У першому випадку зміни відбуваються в структурі операторів та блоків методу, у другому – в атрибутах і методах деякого класу, або ж зв’язки класах реорганізовуються між собою. У разі реструктуризації на рівні пакетів деякі класи можуть бути переміщені в інші пакети, переіменовані або ж зовсім видаленні за не більше потрібністю в використанні.

Один з механізмів автоматичного рефакторингу методів в об'єктно-орієнтованих програмних додатках використовує зважений граф залежностей, оснований на історії модифікацій методів [15]. Також були опубліковані роботи [17], присвячені виявленню прийому рефаторингу "Витяг методу" (Extract Method Refactoring [12]). Одна з тих робот базується на аналізі побудованого графа залежностей між операторами вихідного коду, при побудові якого враховується значення щільності зв'язку між ними [13].

До типу алгоритмів реструктуризації на уровні класів належать алгоритм Automatic Refactorings Identification (ARI) [14] та алгоритм Hierarchical Agglomerative Clustering (HAC) [16], в основі 8 якого лежить принцип векторизації сутностей (класів і методів).

Ієрархічна кластеризація (також графові алгоритми кластеризації і ієрархічний кластерний аналіз) – сукупність алгоритмів упорядкування та групування даних, спрямованих на створення ієрархії (дерева) вкладених кластерів.

Агломеративні методи (англ. Agglomerative): нові кластери створюються

шляхом об'єднання дрібніших кластерів і, таким чином, дерево створюється від листя до кореню.

Алгоритми ієрархічної кластеризації припускають, що аналізована множина об'єктів характеризується певним ступенем зв'язності. За кількістю ознак іноді виділяють монотетичні і політетичні методи класифікації. Як і більшість візуальних способів подання залежностей, графи швидко втрачають наочність при збільшенні числа кластерів.

Стандартний алгоритм ієрархічної кластеризації має часову складність $O(n^3)$ та потребує пам'яті $O(n^2)$ що занадто повільно навіть для наборів даних середнього розміру. Однак, для деяких випадків, є агломератові методи, які виконуються за $O(n^2)$ час . Це методи SLINK при однозв'язній та CLINK при повнозв'язній кластеризації. Використання структури даних як купа дозволяє у загальному випадку скоротити час виконання до $O(n^2 logn)$ ціною збільшення вимог до об'єму використаної оперативної пам'яті. Нажаль, Такі накладні витрати на пам'ять для багатьох мов програмування роблять цей підхід неможливим для реалізації.

В вектор кластеру за звичай входять чотири метрики програмного забезпечення:

– глибина вибраного класу в дереві спадкоємства (Depth in Inheritance Tree, DIT);

– метрика Fan–Out (для класів – це кількість стороніх класів, на які посилається даний клас, для методів – це кількість інших методів, які викликаються з даного).

– метрика Fan–In (для класів: кількість інших класів, що мають посилання на даний, для методів: кількість сторонніх методів, що міcть у собі виклик данного метода);

– кількість прямих класів–спадкоємців (Number of Children, NOC);

Також для класів і векторів будуються множини пов'язаних сутностей (Relevant properties, RP). Далі вводиться функція відстаней між векторизованнимм сутностями приведена на формулі 2.2.

$$d(X,Y)=\begin{cases}\sqrt{\frac{1}{m}*((1-\frac{|RP_X\cap RP_Y|}{|RP_X\cup RP_Y|})+\sum_{i=1}^{4}(v_{X_i}-v_{Y_i})^2)}, RP_X\cap RP_Y\neq\emptyset\\ \infty, otherwise\end{cases} \quad (2.2)$$

де *d(X, Y)* – є відстаннь між векторизованими сутностями;

RP – пов’язані сутності;

m – кількість методів, в яких використовуються досліджувані сутності;

Далі на основі цієї метрики виконуються різні алгоритми кластеризації, в тому числі, алгоритм ієрархічної кластеризації [16]. Загальна схема алгоритму кластерізації наведена на рисунку 2.1.

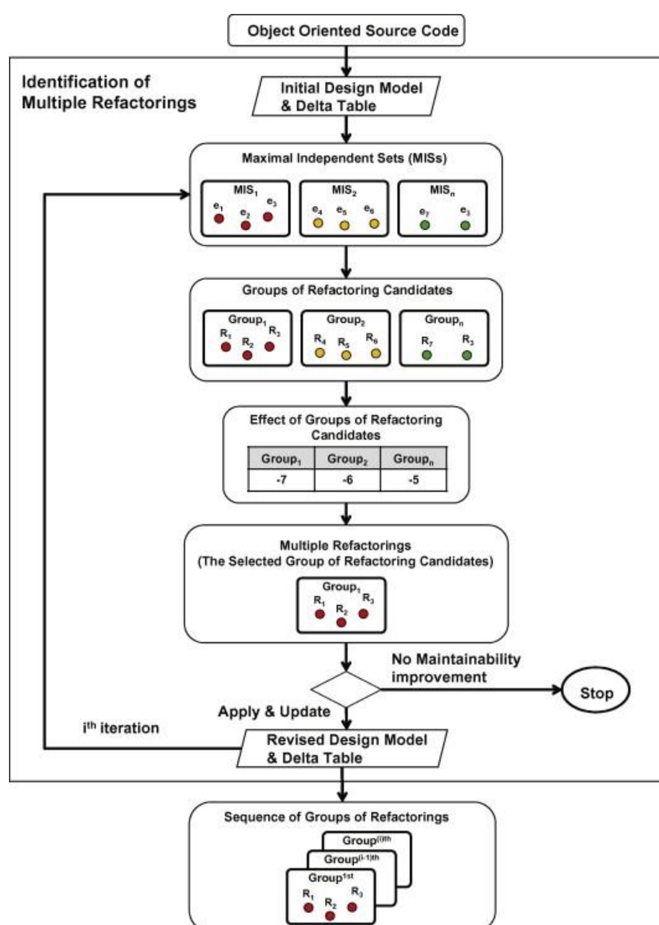

Рисунок 2.1 – Загальна схема алгоритмів кластерізації

Для кількісної оцінки структури пакетів в додатку була введена спеціальна метрика аналізу програмного забезпечення (Package Structure Analysis metric [8]), а

також був реалізований допоміжний інструмент, призначений спеціально для розробників програмного забезпечення, з ціллю аналізу та візуалізації структури пакеті проектів. Також було запропоновано адаптований алгоритм виділення спільнот в графі – залежностей між класами вихідного коду (Constrained Community Detection Algorithm, CCDA). Зважений граф залежностей між класами влаштований таким чином, що кожній вершині відповідає деякий клас, а вага ребра визначається як кількість пов'язаних між собою методів або атрибутів двох класів. Подальший алгоритм заснований на так званому показнику якості (Quality Index) Qw, адаптованому для використання у зважених графах.

2.3 Дерева прийняття рішень

Дерево прийняття рішень (також може називатися деревом класифікації або регресійний деревом) – це засіб підтримки прийняття рішень, що використовується в машинному навчанні, аналізі даних і статистичним аналізом. Структура дерева є максимально схожою з класичною структурою даних – «листя» та «гілки». На ребрах («гілках») дерева рішення записані атрибути, від яких залежить цільова функція, в «листі» записані значення цільової функції, а в інших вузлах – атрибути, за якими розрізняються випадки. Для того щоб почати класифікувати новий випадок, треба спуститися вниз по дереву до листа і видати відповідне значення. Подібні дерева рішень широко використовуються в інтелектуальному аналізі даних.

Дерева рішень, використовувані бувають двох основних типів:

- аналіз дерева класифікації, коли прогнозований результат є класом, до якого належать дані;
- регресійний аналіз дерева, коли прогнозований результат можна розглядати як дійсне число (наприклад, ціна на будинок, або тривалість перебування пацієнта в лікарні).

Мета полягає в тому, щоб створити модель, яка передбачає значення цільової змінної на основі декількох змінних на вході.

На рисунку 2.2 приведено просте дерево прийняття рішень, яке на основі заздалегідь підготовлених умов обчислює колір координати

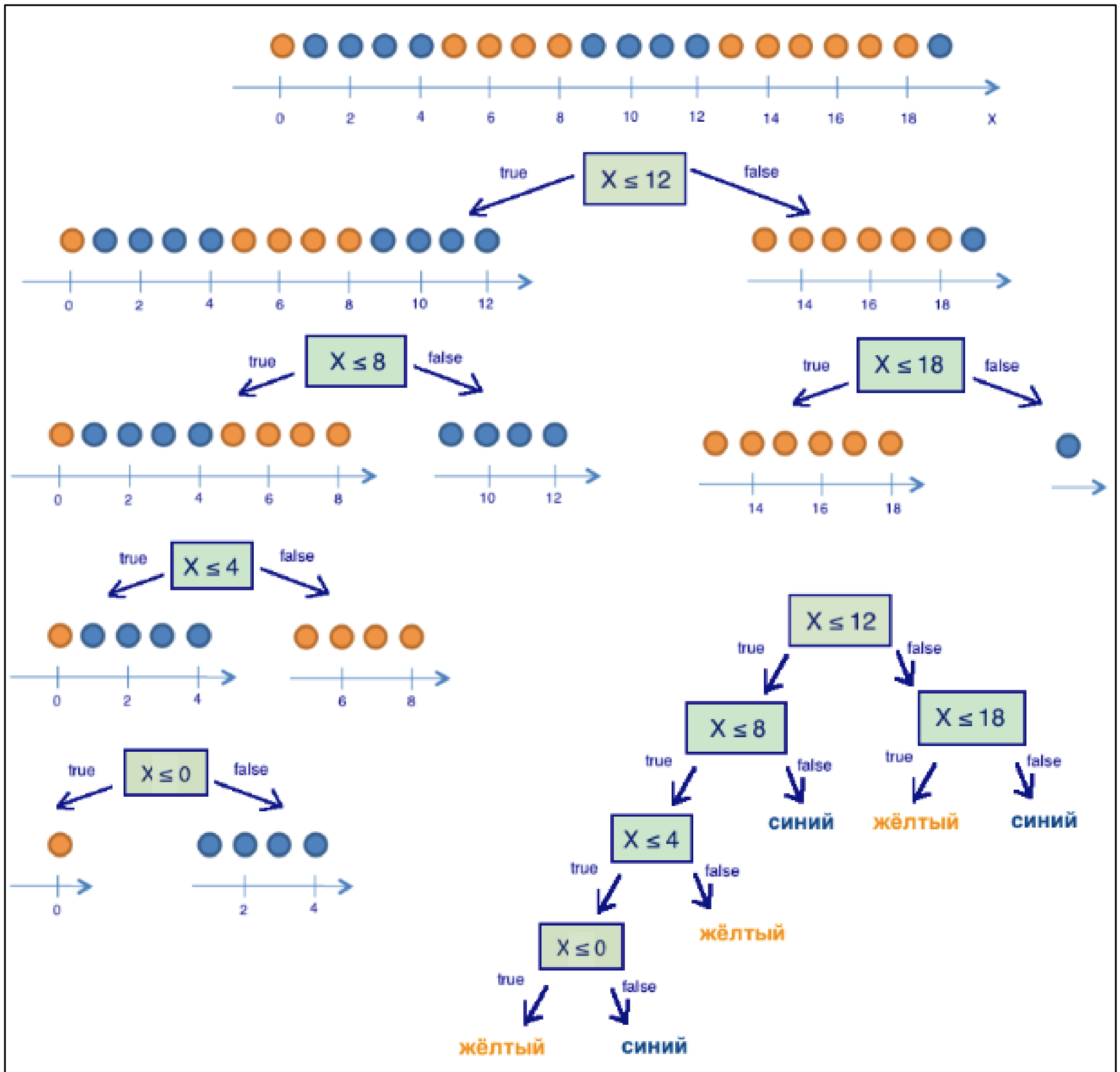

Рисунок 2.2 – Просте дерево прийняття рішень

Листям дерева прийняття рішень є класи. Щоб класифікувати об'єкт за допомогою дерева прийняття рішень – потрібно послідовно спускатися по дереву (вибираючи напрямок ґрунтуючись на значеннях предикатів застосовуються до об'єктів, що класифікуються). Шлях від кореня дерева до листя можна трактувати

як пояснення того, чому той чи інший об'єкт віднесений до якого–небудь класу.

У розглянутому прикладі, для спрощення, всі об'єкти характеризуються тільки одним атрибутом – координатою х, але точно такий же підхід можна застосувати і до об'єктів з множиною різних за значенням атрибутів.

Також, що не накладається обмежень на значення атрибутів об'єкта – вони можуть мати як категоріальну, так і числову або логічну природу. Потрібно тільки визначити предикати, які вміють правильно обробляти значення атрибутів (наприклад, навряд чи є сенс використовувати предикати «більше» або «менше» для атрибутів з логічними значеннями).

Сильні сторони методу дерев прийняття рішень є:

– простий в розумінні та інтерпретації. Люди здатні інтерпретувати результати моделі дерева прийняття рішень після короткого пояснення;

– не потребує підготовки даних. Інші техніки вимагають нормалізації даних, додавання фіктивних змінних, а також видалення пропущених даних;

– здатний працювати як з категоріальним, так і з інтервальними змінними. Інші методи працюють лише з тими даними, де присутній лише один тип змінних. Наприклад, метод відносин може бути застосований тільки на номінальних змінних, а метод нейронних мереж тільки на змінних, виміряних по інтервального шкалою;

– використовує модель «білого ящика». Якщо певна ситуація спостерігається в моделі, то її можна пояснити за допомогою булевої логіки. Прикладом «чорного ящика» може бути штучна нейронна мережа, так як результати даної моделі важко піддаються поясненню;

– дозволяє оцінити модель за допомогою статистичних тестів. Це дає можливість оцінити надійність моделі;

– є надійним методом. Метод добре працює навіть в тому випадку, якщо були порушені початкові припущення, включені в модель;

– дозволяє працювати з великим об'ємом інформації без спеціальних підготовчих процедур. Даний метод не вимагає спеціального обладнання для роботи з великими базами даних.

Недоліки методу дерев прийняття рішень:

– проблема отримання оптимального дерева рішень є NP–повною з точки зору деяких аспектів оптимальності навіть для простих завдань [18]. Таким чином, практичне застосування алгоритму дерев рішень засноване на евристичних алгоритмах, таких як алгоритм «жадібності», де єдино оптимальне рішення вибирається локально в кожному вузлі. Такі алгоритми не можуть забезпечити оптимальність всього дерева в цілому;

– в процесі побудови дерева рішень можуть створюватися занадто складні конструкції, які недостатньо повно представляють дані. Дана проблема називається перенавчанням [18]. Для того, щоб її уникнути, необхідно використовувати метод «регулювання глибини дерева».

– для даних, які включають категоріальні змінні з великим набором рівнів, більша інформаційна вага присвоюється тим атрибутам, які мають більшу кількість рівнів.

У цій роботі також будуть використовуватись дерева прийняття рішень які використовують абстрактні синтаксичні дерева для виконання тих чи інших рішень, що згодом призведуть до оптимізації програмного коду.

3 ОПИСАННЯ ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

Принцип роботи прототипу програмної реалізації проілюстровано на рисунку 2.1.

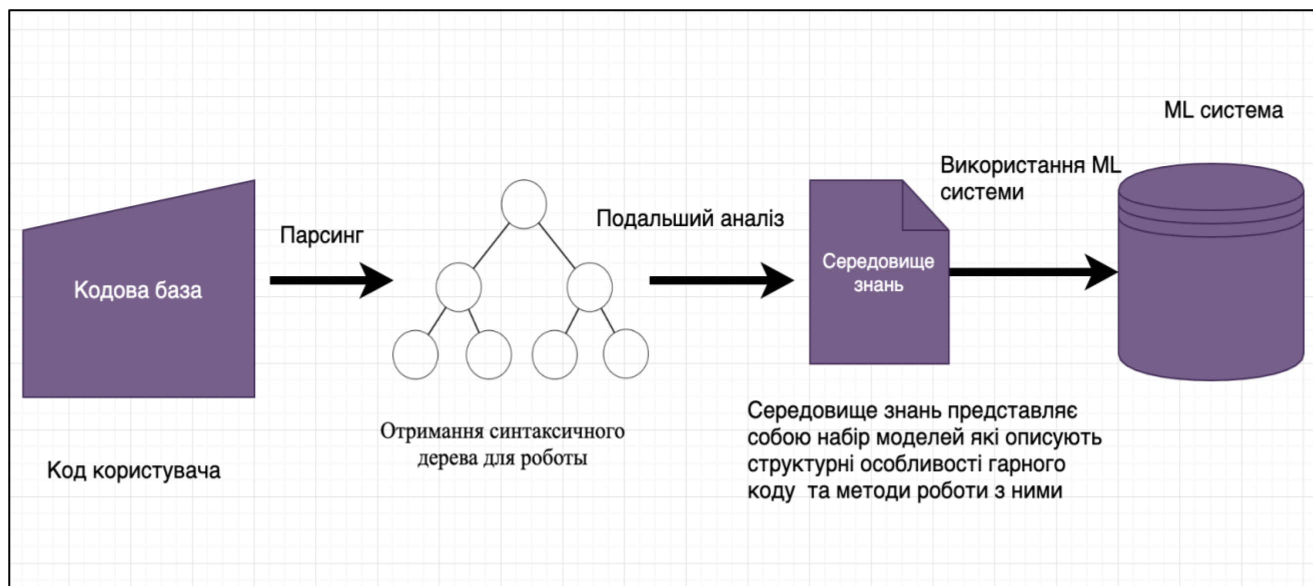

Рисунок 3.1 – Принци роботи прототипу системи

Розберемо кожен крок и компонент окремо. Починається усе з загрузки користувачем його програмного коду, або частини у додаток. На даному етапі цієї дослідницької роботи забезпечена підтримка тільки скриптової мови програмування Java Script, у подальшому планується провести розширення і надати підтримку основним мовам програмування, таким як: C#, C++, Java, Go, тощо.

3.1 Парсинг вхідного програмного коду

Первинним результатом роботи цього додатку є множина оброблених та розпарсених абстрактних синтаксичних дерев вхідного програмного коду, які у

подальшому будуть використані при пошуку помилок, порушень правил написання стандартизованого коду та архітектурних шаблонів проектування і виконанні автоматичного рефакторингу. Для проведення парсингу використовується JavaScript бібліотека *Acorn,* яка від самого початку була розроблена та спрямована на аналізування Java Script файлів. На рисунку 3.2 приведено приклад абстрактного синтаксичного дерева для функції від одного аргументу.

```
- Program {
    type: "Program"
    start: 0
    end: 109
  - body: [
    - FunctionDeclaration = $node {
        type: "FunctionDeclaration"
        start: 0
        end: 109
      + id: Identifier {type, start, end, name}
        expression: false
        generator: false
      + params: [1 element]
      - body: BlockStatement {
          type: "BlockStatement"
          start: 22
          end: 109
        - body: [
          + VariableDeclaration {type, start, end, declarations, kind}
          + IfStatement {type, start, end, test, consequent, ... +1}
          + ReturnStatement {type, start, end, argument}
          ]
        }
      }
    ]
    sourceType: "module"
  }
```

Рисунок 3.2 – Абстрактне синтаксичне дерево яке описує просту функцію від одного аргументу

Саме така структура абстрактного дерева (у форматі Java Script Notation Object) використовується при аналізі вхідного програмного коду, а також в моделях, які описуюсь ти чи інші архітектурні шаблони проектування, стандарти та правила написання чистого та підтримуваного програмного коду. Також, слід зазначити, що дерево містить безліч інших допоміжних властивостей, які дають більш детальний опис, щодо аналізованого коду і саме допомагає роботи більш глибокий и точний аналіз якості коду. Так як для людини дуже складно розуміти на аналізувати таку рекурсивну структура даних на рисунку 3.3 приведене теж саме за внутрішньою структурою, але трохи спрощене абстрактне синтаксичне дерево.

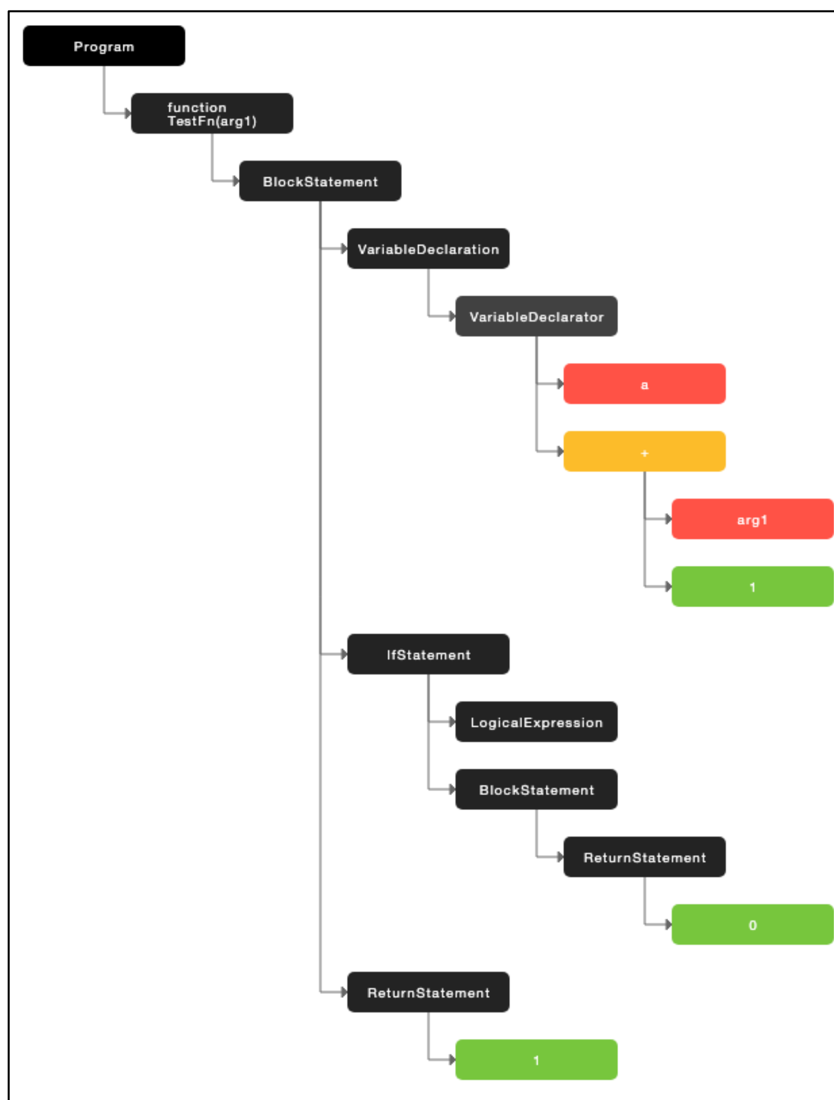

Рисунок 3.3 – Схематичне зображення абстрактного синтаксичного дерево, яке описує функцію від одного аргументу

А вже на рисунку 3.4 приведена та сама, зазначена вище JavaScript функція від одного аргументу, на основі якої було побудоване дане абстрактне синтаксичне дерево.

```
1    function TestFn(arg1) {
2      const a = arg1 + 1;
3
4      if (a > 5 && a % 2 === 0) {
5        return 0;
6      }
7
8      return 1;
9    }
```

Рисунок 3.4 – Код функції від одного аргументу

Як можна побачити, що абстрактне синтаксичне дерево є досить важкою в плані вимог до оперативної пам’яті структурою даних, так як на розбір такої маленької і простої функції було відведено аж вісім рівнів дерева.

3.2 Пошук оптимальних моделей

Далі виконується підбір найбільш підходящих моделей які описують практики написання чистого підтримуваного коду або навпаки моделі чи дерева, що описують порушення тих чи інших архітектурних патернів проектування. Усі ці моделі знаходяться у так званому «Середовище знань». Середовище знань представляє собою заздалегідь згруповані по певним критеріям чи типам абстрактні синтаксичні дерева, набір моделей, які виявлять ти чі інші патерни та

правила допомагають ML системі робити найбільш оптимальні та зважені рішення з приводу рефакторингу.

На рисунку 3.5 зображена діаграма класів, яка описує тип моделі рефакторингу програмного коду.

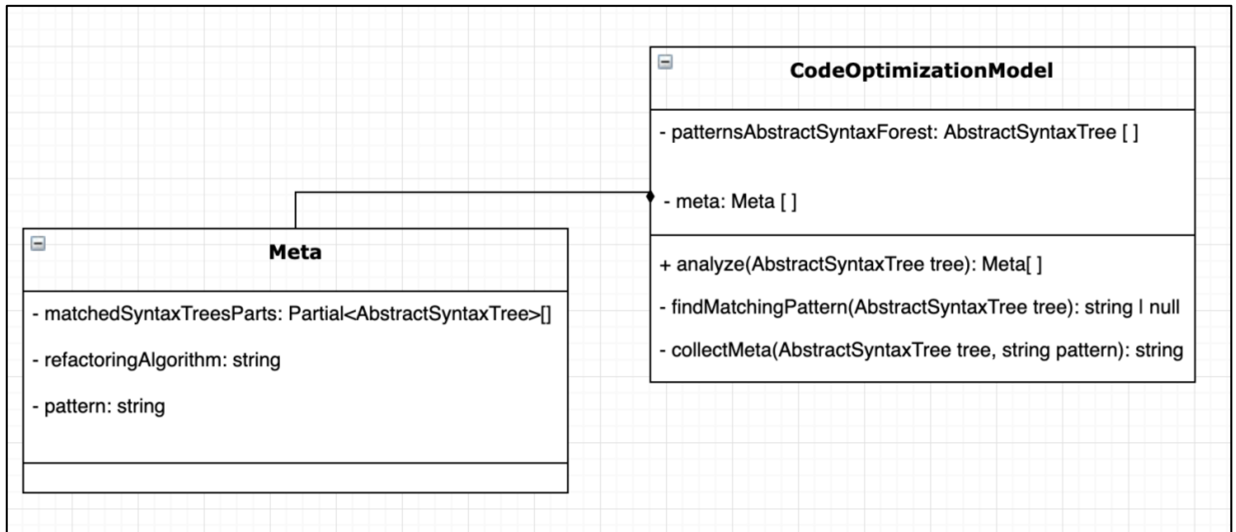

Рисунок 3.5 – Діаграма класів моделі рефакторингу

Згідно цієї діаграми, клас *CodeOptimizationModel* виконує первинний аналіз вхідного абстрактного синтаксичного дерева, для того щоб дізнатись чи підходить вхідний код під ситуації які описуються деякими архітектурними шаблонами проектування чи іншими правилами та практиками написання якісного та стандартизованого програмного коду. Якщо вхідне синтаксичне дерево не підходить під умови шаблону, то метод *findMatchingPattern* поверне значення *null*, в іншому випадку, цей метод поверне значення рядку, що представляє собою назву деякого шаблону і після цього почне збирати мета дані (див. метод *collectMeta*) для передачі їх в систему машинного навчання і виконання автоматичного рефакторингу.

Об’єкт мета даних на даному етапі складається з частин синтаксичного дерева за допомогою яких була розпізнана ситуація щодо використання деякого архітектурного патерна чи правила написання якісного коду, далі, з алгоритму рефакторингу (це може бути HAC або ARI, які були раніше вже описані в даній

дослідницькій роботі), і останньою властивістю мета даних є безпосередньо назва шаблону.

У наступному кроці, масив (може бути декілька випадків, які описує патерн) мета даних передається у систему машинного навчання та відбувається рефакаторинг і реорганізація програмного коду.

3.3 Реорганізація коду за допомогою системи машинного навчання

Система машинного навчання складається з лісу дерев рішення, згрупованих по певними архітектурним патернам, правилам та стандартам написання якісного та гарного коду, тощо. Кожне дерево приймає набір конкретних кроків для того щоб зібрати необхідну допоміжну інформацію та провести реорганізацію програмного коду.

Як вже було зазначено, збір інформації виконується в листках дерева, а так як основним ресурсом з, яким працюють дерева вирішення у цій дослідницькій роботі є абстрактні синтаксичні дерева, то збір атрибутів складається в знаходженні деяких компонентів класів та функцій

Після того як інформація була зібрана, і будь яке дерево було пройдене з кореня до низу, тобто останнього листка і було повернено значення TRUE, (це означає, що є можливість для виконання рефакторингу програмного коду), то клас чи функція будуть реорганізовані за допомогою алгоритмів HAC чи ARI.

Якщо ж на деякому з шагів (мається на увазі деякий лист дерева) повертається значення, яке є FALSE, то це означає, що на дане дерево прийняття рішень не може прийняти остаточне рішення щодо потреби в реорганізації програмного коду, тому ніякі подальші дії не виконуються. Якщо ж увесь ліс дерев вирішення повертає значення FALSE, то ми маємо ситуацію коли дана система машинного навчання ще не має достатнього досвіду вирішення задачі, або

реорганізація не може бути виконана, так як алгоритм рефакторингу не має достатніх даних, але це вже є обмеженнями цих алгоритмів.

На даному етапі існування цього програмного додатку надана підтримка щодо розпізнавання декількох архітектурних шаблонів проектування та правил написання гарного коду, далі будуть наведені та описані основні з них, а також підтримується можливість запровадження та використання власних стандартів написання коду.

Архітектурні шаблони проектування які підтримуються на даному етапі системою:

– патерн «Будівельник». Шаблон дозволяє вам створювати різні види об'єкту, уникаючи засмічення конструктора. Він корисний, коли може бути декілька видів об'єкту або коли потрібна безліч кроків, пов'язаних з його створенням. Розпізнавання цього шаблону проектування за звичай відбувається за допомогою аналізу конструктора, його структури та логіки (якщо вона є), та кількості властивостей класу;

– патерн «Фасад». Це простий інтерфейс для роботи зі складною підсистемою, яка містить безліч класів. Фасад може бути спрощеним відображенням системи, що не має 100% тієї функціональності, якої можна було б досягти, використовуючи складну підсистему безпосередньо. Разом з тим, він надає саме ті «фічі», які потрібні клієнтові, і приховує все інше. Фасад корисний у тому випадку, якщо використовується якась складна бібліотеку з безліччю рухомих частин, з яких вам потрібна тільки частина. Розпізнавання та реорганізація коду згідно цього шаблону проектування є дуже складним та у деякому плані «творчим» завданням, тому що можна знайти безліч ситуацій та ознак щодо доцільності використання цього шаблону. На даному етапі аналізує логіку коду, та робить припущення щодо винесення деяких частин коду у більш загальний та простий для розуміння інтерфейс;

– патерн «Ітератор». Це поведінковий патерн проектування, що дає змогу послідовно обходити елементи складових об'єктів, не розкриваючи їхньої внутрішньої організації. Ідея патерна Ітератор полягає в тому, щоб винести

поведінку обходу колекції з самої колекції в окремий об'єкт. Об'єкт–ітератор відстежуватиме стан обходу, поточну позицію в колекції та кількість елементів, які ще залишилося обійти. Одну і ту саму колекцію зможуть одночасно обходити різні ітератори, а сама колекція навіть не знатиме про це. Розпізнавання патерну «Ітератор» відбувається під час аналізу складних конструкцій *for in, for of, while* у програмному коду. Тому, якщо були помічені повторювані ітерації через складні об'єкти, то припускається, що патерн «Ітератор» може мати місце у викиненій ситуації;

– усунення дублювання коду. У більшості випадків дублювання виникає тоді, коли в проекті працюють декілька авторів, причому над різними його частинами. Вони працюють над схожими задачами, але не знають, що колега вже написав схожий код, який можна використати замість написання свого. Зустрічається і непряме дублювання, коли конкретні ділянки коду відрізняються зовні, хоча і виконують одну і ту ж задачу. Таке дублювання буває досить складно виявити і виправити. В окремих випадках дублювання створюється навмисно. Найчастіше в поспіху, коли строки здачі проекту горять. Програміст–початківець бачить у вже написаному коді «майже такий, як треба» фрагмент, і не може встояти від спокуси просто скопіювати його як є і вставити десь в іншому місці (і може повторюватися десяток разів). Розпізнавання цієї ситуації досить просте – треба проаналізувати логіку програмного коду та внутрішню структуру компонентів, якщо будуть знайдено достатню кількість збігів то слід використати прийоми реорганізації коду «Винесення класу» або ж «Винесення методу класу»;

– використання користувальницьких практик щодо написання гарного та підтримуваного програмного коду. Система надає можливість користувачу описати та запровадити свої власні стандарти щодо програмного написання коду. Наприклад, використання прийому композиції функцій там, де це буде доцільним рішенням (більш детальний приклад з композицією функцій буде наведено в наступному розділі), або ж надання переваги композиції програмних компонентів, замість використання широко відомого механізму наслідування класів. Також, можуть бути виконані маніпуляції з директоріями, розміщенням в них файлів та їх

власних назв. Нема дуже серйозних обмежень щодо власних практик написання коду.

3.4 Побудова діаграм класів та об'єктів

В даному програмному додатку надана підтримка побудови діаграм класів та об'єктів для візуальної оцінки якості програмних компонентів та зазначення виявлених порушень на малюнках. Візуальний підхід до проектування з використанням раціонального уніфікованого підходу та уніфікованої мови моделювання (Unified Modeling Language, UML) дозволяє ефективно вести боротьбу із постійно зростаючою складністю ПЗ, здійснювати їх аналіз, будувати стабільну архітектуру складних програмних систем різного призначення. Система є складною, якщо розробники для складання деякого цілісного уявлення про неї розглядають її не з однією, а з багатьох різних точок зору: з позиції об'єктів і відносин між ними, бізнес– та інших процесів. Крім того, програмне забезпечення розглядається програмними інженерами з позицій глобальних і локальних змінних, однозначно ідентифікованих імен змінних, інкапсуляції частин програмних кодів і багатьох інших точок зору.

Основою створення якісного ПЗ є моделювання, що дозволяє:

– наочно продемонструвати деталі, які відносяться до внутрішньої та зовнішньої структури системи, яка розробляється, описати поведінку програмних компонентів;

– здійснювати візуалізації та керування розвитком архітектури програмної системи;

– забезпечити краще розуміння створюваної системи, що найчастіше призводить до її спрощення й забезпечує можливість повторного використання вже існуючих програмних компонентів, а також більш простого та швидкого розуміння

логіки програмних компонентів та прийнятих рішень новими у команді розробниками;

– мінімізувати ризики. Візуалізація допомагає краще побачити картину того що відбувається зараз з системою, тому деякі неявні помилки архітектурного проектування можуть бути знайдені та виправленні заздалегідь.

На рисунку 3.6 приведено діаграму класів для програмного механізму, який у цьому додатку виконує побудову діаграм об'єктів та класів для користувача

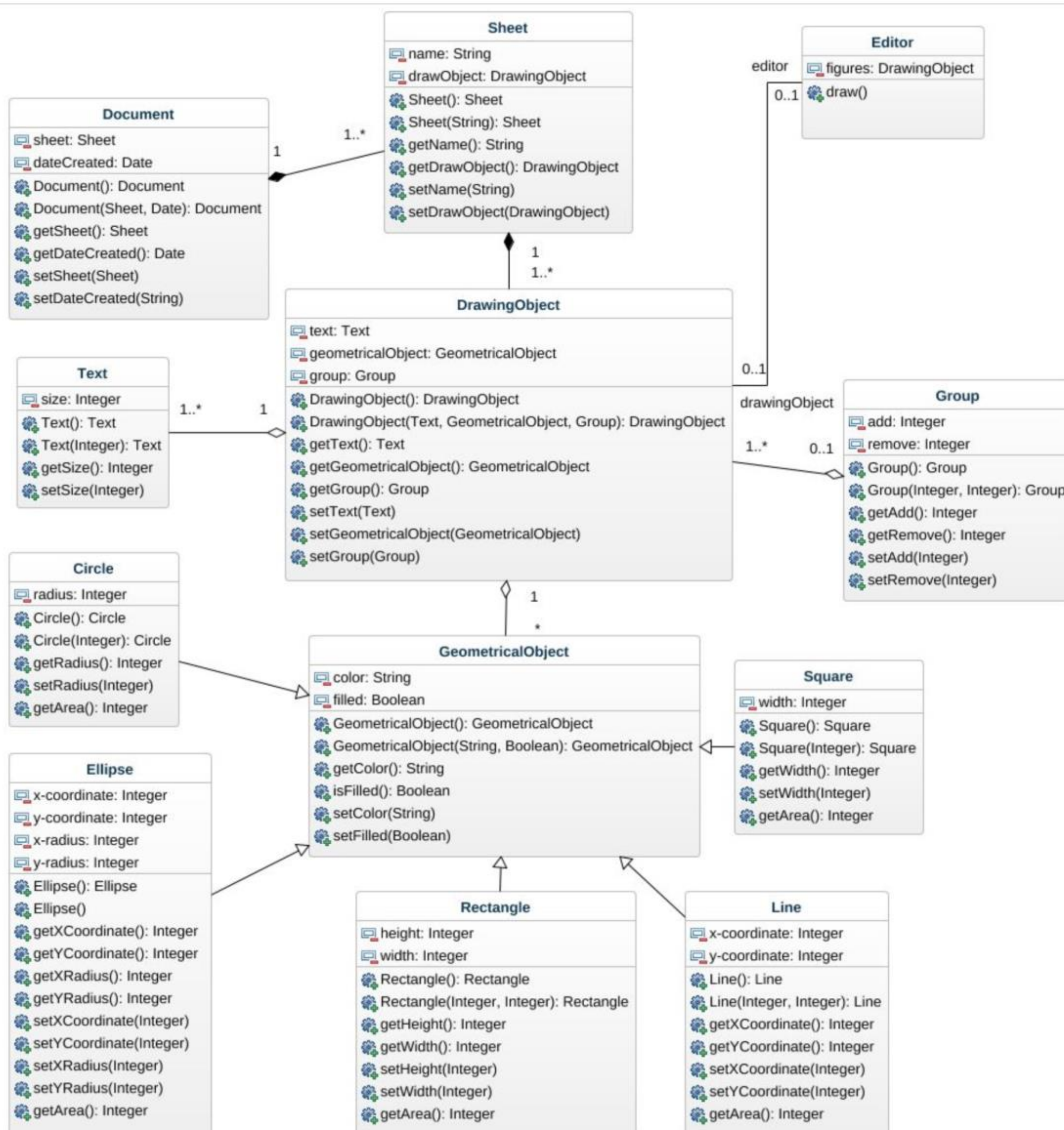

Рисунок 3.6 – Діаграма класів для механізму побудови діаграм об'єктів та класів

Згенеровані діаграми мають розширення *.svg, що дозволяє з легкістю переглянути їх в будь якому браузері.

4 АВТОМАТИЧНИЙ РЕФАКТОРИНГ ТА ОЦІНКА ЯКОСТІ

Для того щоб аргументувати доцільність використання методів машинного навчання, а саме дерев рішення, як автоматизований та розширюваний інструмент для розпізнавання та реорганізації не стандартизованого та погано підтримуваного програмного коду будуть виконані наступні кроки:

– безпосереднє приведення прикладу неякісно написаного програмного коду;

– наведена низка вагомих аргументів, щодо того чому приведений у попередньому пункті приклад програмного коду потребує реорганізації, або які правила написання коду, проблеми описані архітектурними шаблонами проектування він порушив;

– наведені значення основних метрик оцінки якості коду, а саме: значення цикломатичної складності коду, значення індексу підтримки та значення об’єму Хальстеда. Найбільш вагомим є значення індексу підтримки;

– спрощена схема дерева рішень, на основі якого буде прийматись рішення щодо того чи потрібно виконувати реорганізацію приведного програмного коду чи такої потреби нема, а також описані кроки, які виконує дерево, та його компоненти;

– отримання та детальний розбір реорганізованого за допомогою дерева рішень програмного коду, приведена порівняльна характеристика з станом коду до виконання рефакторингу;

– наведені нові значення основних метрик оцінки якості коду, для того щоб емпірично запевнитися в тому, що метрики були покращені за допомогою порівняння зі старими значеннями;

– наведення рисунків з UML діаграми класів чи об’єктів для візуалізації та кращого розуміння картини того, що було змінено та реорганізовано і описано чому саме;

Далі буде розглянуто декілька ситуацій щодо реорганізації програмного коду, а саме:

– використання архітектурного шаблону проектування «Будівельник» для спрощення складного конструктора;

– виявлення та усунення дуплікацій програмного коду у функціях / класах / методах;

– рефакторинг програмного коду на основі власного дерева прийняття рішень від користувача.

4.1 Використання архітектурного шаблону проектування «Будівельник»

На рисунку 4.1 зображено приклад створення екземпляру простого типу даних *DummyClass*, який має лише один метод – це його власний конструктор.

```
1   class DummyClass {
2     constructor(prop1, prop2) {
3       if (prop1.length > 10) {
4         this.prop1 = prop1;
5       }
6
7       if (prop2 === true) {
8         this.prop2 = true;
9       }
10    }
11  }
12
13  const object = new DummyClass("test", true);
```

Рисунок 4.1 – Приклад програмного коду, до якого слід використати шаблон проектування «Будівельник»

Розберемо чому приведений на рисунку програмний код вважається не оптимальним. Клас *DummyClass* є досить простим, але має відносно складний

метод конструктору, який володіє занадто великим об'ємом знань щодо вхідних аргументів, тим самим ускладняються логіка і рівень безпроблемної підтримки цього класу людьми стрімко знижується.

Тому є доцільним використання архітектурного шаблону проектування «Будівельник», який вирішує дану проблему введенням допоміжного класу будівельника, спрощуючи складний конструктор до процесу по–крокового створення об'єкту. Цей шаблон проектування пропонує розбити процес конструювання об'єктів на низку окремих кроків (наприклад, *побудуватиВластивість1*, *встановитиВластивість2,* і т. д.). Щоб створити об'єкт, потрібно по черзі (або в довільному порядку) викликати методи будівельника. До того ж не потрібно викликати всі кроки, а лише ті, що необхідні для виробництва об'єкта певної конфігурації. Зазвичай, один і той самий крок будівництва може відрізнятися для різних варіацій виготовлених об'єктів. Також, при необхідності, побудований об'єкт може бути частково перероблений, оскільки будівельник зберігає екземпляр об'єкту що будується у свої полях тим надає гнучкиий доступ до нього.

Розглянемо на рисунку 4.2, на якому зображено основні значення метрик якості програмного коду а саме: значення індексу підтримки, об'єму Хальстеда та цикломатичної складності даного програмного коду.

```
{
    "maintainabilityIndex": 78.23276402335317,
    "cyclomaticComplexity": 1,
    "halsteadVolume": 29.47722668160484
}
```

Рисунок 4.2 – Значення основних метрик якості програмного коду для прикладу до використання шаблону «Будівельник»

На рисунку 4.3. приведено спрощене дерево прийняття рішень, яке використовується для розпізнавання проблеми, описаної вище архітектурним

шаблоном проектування «Будівельник», та підбору оптимальної моделі рефакторингу початкового програмного коду.

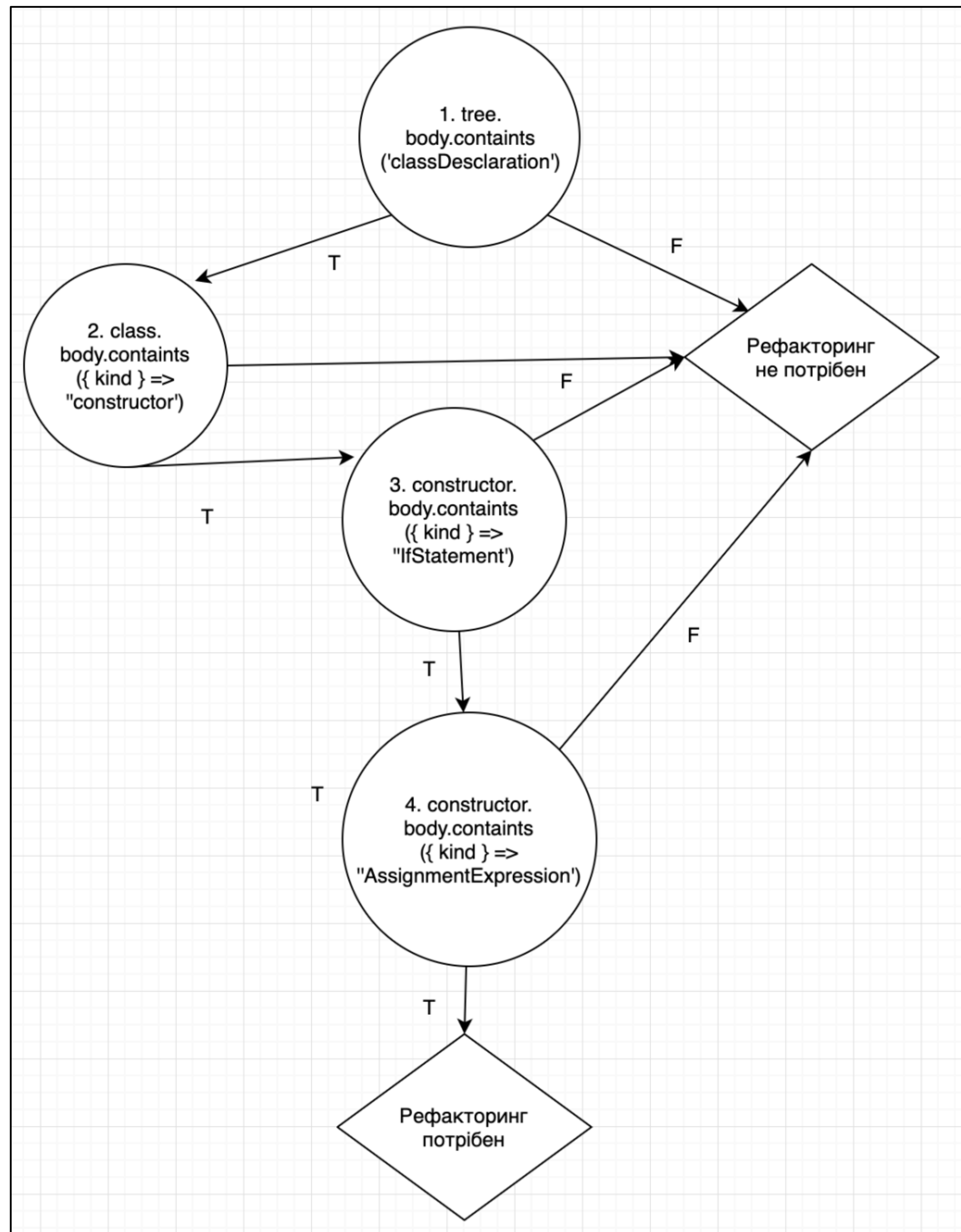

Рисунок 4.3 – Дерево рішень для оптимізації коду згідно патерну проектування «Будівельник»

Розберемо приведено дерево крок за кроком на прикладі вже відомого класу *DummyClass*, а також розглянемо більш детальніше його структуру, проміжний стан та предикати листів.

– по–перш за все, потрібно визначити чи є присутнім у даному файлі декларація будь якого класу, якщо ж вхідний файл не має жодного класу, то рефакторинг не буде порозводитись. Згідно рисунку 4.1 у приведеному лістингу програмного коду є декларація класу *DummyClass*, тому алгоритм має змогу рухатись далі по дереву;

– якщо файл має декларацію класу, то наступним кроком є перевірка того, чи є у оголошеному класі метод конструктору, якщо даного методу клас не має, то рефакторинг і рух далі по дереву не буде продовжено. У класі *DummyClass* є оголошений конструктор який залежить від двох вхідних параметрів, тому предикат на даному етапі поверне значення, яке відповідає дійсності і рух по дереву буде продовжений;

– третім кроком є пошук конструкцій розгалуження (блоки if–else, тернарні оператори), вважаємо, якщо конструктор має такі конструкції, то це є складний конструктор, який має бути реорганізований до більш простого. Якщо в конструктор не містить команд розгалуження , то рух далі по дереву і рефакторинг коду не буду порозводитися. У класі *DummyClass* знаходиться цілих дві конструкції IF, тому предикат цього листа повертає дійсне значення і розбір дерева продовжиться;

– останнім кроком є пошук команд присвоєння у конструкторі усередині конструкцій розгалуження (ускладнення логіки створювання). Якщо таких команд нема, то рух далі по дереву і рефакторинг не виконуються, тому що дерево вже не має листів предикатів, що означає що ознаки даного класа не підходять під проблеми, які виправляються архітектурним шаблоном проектування «Будівельник». В приведеному прикладі програмного коду є цілих дві команди присвоєння властивостей класу усередині IF конструкцій, тому вважаємо, що цей клас має складний конструктор, який ускладнює читабельність та легкість в розуміння та підтримуванні для інших розробників, тому слід використати шаблон «Будівельник» і реорганізувати код.

Після того як дерево закінчило свою роботу, для виконання рефакторингу програмного коду будуть використані алгоритми, які вже були описані в першій

частині, а саме – Automatic Refactorings Identification та алгоритм Hierarchical Agglomerative Clustering.

На рисунку 4.4. приведено приклад реорганізованого коду для класу *DummyClass*.

```
1   class RefactoredDummyClass {
2     constructor() {
3       this.prop1 = null;
4       this.prop2 = false;
5     }
6   }
7
8   class DummyClassBuilder {
9     constructor() {
10      this.instance = new RefactoredDummyClass();
11    }
12
13    buildProp1(prop1) {
14      if (prop1.length > 10) {
15        this.instance = prop1;
16      }
17      return this;
18    }
19
20    buildProp2(prop2) {
21      this.prop2 = prop2;
22      return this;
23    }
24
25    done() {
26      return this.instance;
27    }
28  }
29
30  const object = new DummyClassBuilder()
31    .buildProp1("test")
32    .buildProp2(true)
33    .done();
34
```

Рисунок 4.4 – Рефакторинг за допомогою класу «Будівельник»

Після рефакторингу, як вже було зазначено, коду стало більше, але код став більш чистим, так як клас *DummyClassBuilder* взяв на себе складні частини

конструктора від не реорганізованого класу *DummyClassю.* За такої структури класів, від клієнтського коду повністю приховується процес конструювання об’єктів. Клієнту залишиться лише прив’язати бажаного будівельника до класу який має бути створений, а потім вже отримати від цього будівельника готовий результат. Також слід зазначити, що завдяки використанню цього архітектурного шаблону проектування з’являється можливість до дуже легкого розширення класу *DummyClass,* а також впровадженню інших будівельників, які матимуть різну логіку побудови.

Обчислимо основні значення метрик якості для вже реорганізованого програмного коду, щоб емпірично довести те, що рефакторинг який був проведений не завдав шкоди якості коду, а навпаки приніс користь та підвищив значення. На рисунку 4.5 приведені значення основних метрик для реорганізованого коду.

```
{
    "maintainabilityIndex": 90.06874506387814,
    "cyclomaticComplexity": 1,
    "halsteadVolume": 40.10455336520968
}
```

Рисунок 4.5 – Значення основних метрик якості для реорганізованого коду

Після виконання рефакторингу значення індексу підтримки зросло від приблизно 78 одиниць, до приблизно 90 одиниць – це є свідченням того, що отриманий реорганізований код є більш підтримуваним і якісним та згідно шкалі наведеною у розділі 1.3 є дуже гарно структурованим та тестованим програмним кодом. Значення цикломатичної складності в обох випадках дорівнює нулю, це пояснюється тим що, обидва приклади коду є досить простими прикладами та не мають складних логічних переходів у своєму програмному потоку виконання. Щодо значення об’єму Хальстеда, то воно незначно зросло – приблизно 29

одиниць в порівнянні з приблизно 40–ка одиницями, зростання цього значення обґрунтовується значно більшою кількістю фізичних рядків програмного коду в реорганізованому прикладі, але ж найбільш ваговим значенням є індекс підтримки. Виходячи з цього можна покласти, що проведений експеримент завершився успіхом.

Користуючись можливістю прототипа системи до генерації UML діаграм, далі на рисунку 4.6 приведена діаграма класів, яка ілюструє виконані зміни (на прикладі обох станів) та надає необхідні коментарі щодо структури класів та виконаних змін у програмного коді

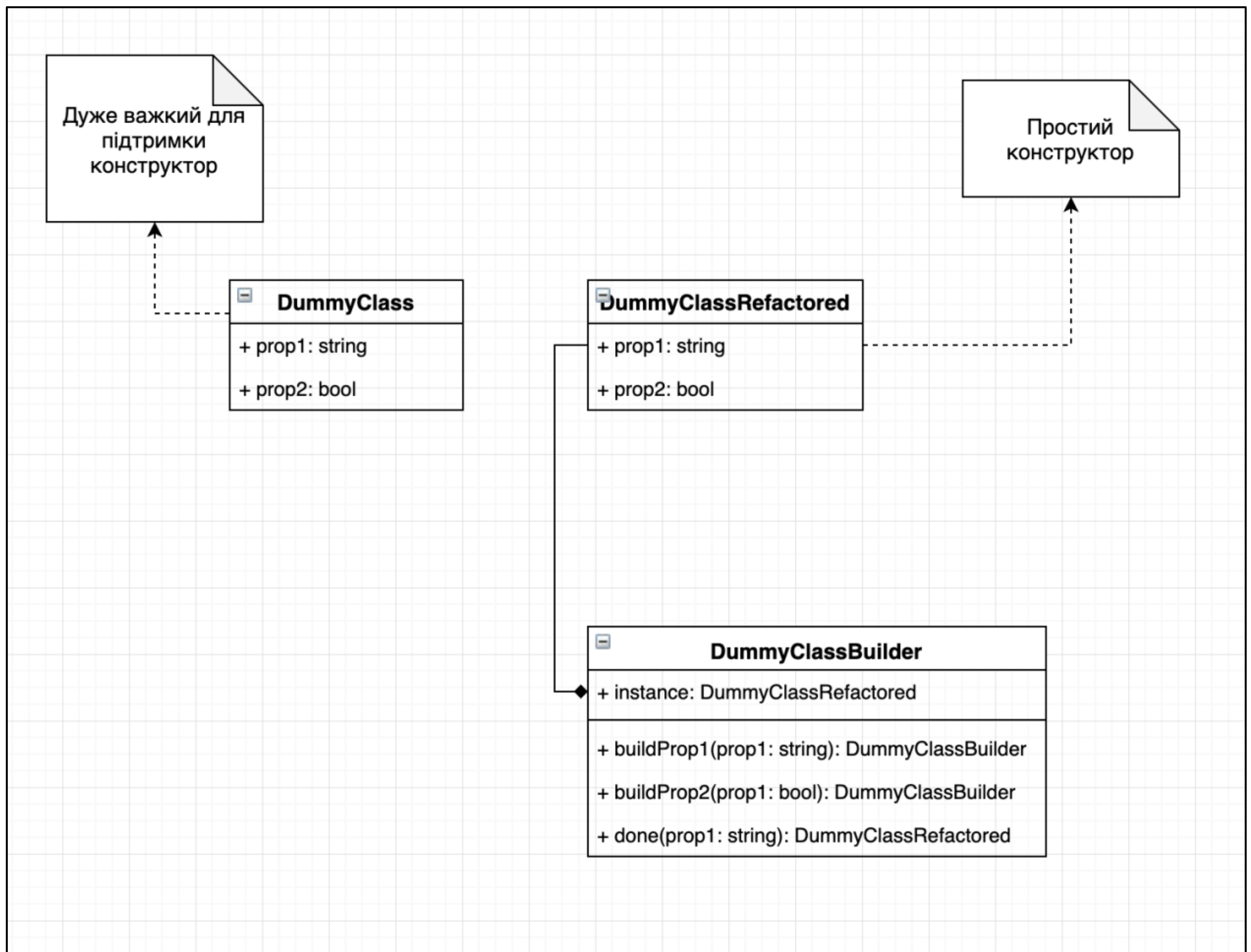

Рисунок 4.6 – Діаграма оптимізованого коду з використанням патерну «Будівельник»

Діаграма показує, що було створено два нових класи та надає інформацію, що конструктор був реорганізований за для поліпшення підтримки.

4.2 Пошук дуплікацій у програмному коді

Дублювання коду є ознакою низького стилю програмування. Гарний стиль програмування звичайно заснований на повторному використанні коду. Може здаватися, що використання дублікатів дозволить дещо прискорити процес створення програми, так як команді розробників не потрібно буде думати над тим, як код використовується і як він може використовуватися надалі. Однак проблема полягає в тому, що написання коду це лише невелика частина життєвого циклу продукту, і подальший супровід програмного коду з дублікатами буде занадто ускладненим.

На рисунку 4.7 наведено приклад програмного коду який містить повторювані частини.

```
1   const array1 = [];
2   const array2 = [];
3
4   let sum1 = 0;
5   let sum2 = 0;
6   let average1;
7   let average2;
8
9   for (let i = 0; i < 4; ++i)  {
10      sum1 += array1[i];
11  }
12
13  average1 = sum1 / 4;
14
15  for (let i = 0; i < 4; ++i)  {
16      sum2 += array2[i];
17  }
18
19  average2 = sum2 / 4;
20
```

Рисунок 4.7 – Приклад коду який містить повторювальну логіку

У цьому прикладі два цикли мають однакову логіку і можуть бути виділені в окрему функцію, тому використання цієї функції позбавить наведений програмний код від дублікатів.

Перед тим як приступати до рефакторингу коду та розбору дерева прийняття рішень – проаналізуємо основні метрики якості програмного коду, які приведені на рисунку 4.8.

```
{
    "maintainabilityIndex": 58.12378822414473,
    "cyclomaticComplexity": 3,
    "halsteadVolume": 4287.3526701282635
}
```

Рисунок 4.8 – Основні метрики якості програмного коду для прикладу з дублікатами

Наведений програмний код є досить простим та коротким фрагментом, але навіть у такому вигляді метрики дають явно зрозуміти, що він має деякі серйозні проблеми, а саме:

– значення індексу підтримки в розмірі приблизно 58 – ми одиниць доводить факт того, що приведений приклад програмного коду є дуже сильно заплутаним, з ускладненим розумінням для інших розробників та важко тестований;

– значення цикломатичною складності дорівнює 3–ьох – ця величина знаходиться у межах допустимої норми, але беручи до уваги приведений фрагмент і його логіку роботи, то значення цикломатичної складності для приведеного коду має становити 1;

– значення об’єму Хальстеда є також досить великим – це логічно обумовлено отриманою поганою оцінкою метрики індексу підтримки та цикломатичної складності програмного коду.

Проаналізувавши отримані метрики, робимо висновок, що даний програмний код має дуже низьку якість та потребує реорганізації.

Розберемо дерево прийняття рішень для цього прикладу, яке наведено на рисунку 4.9.

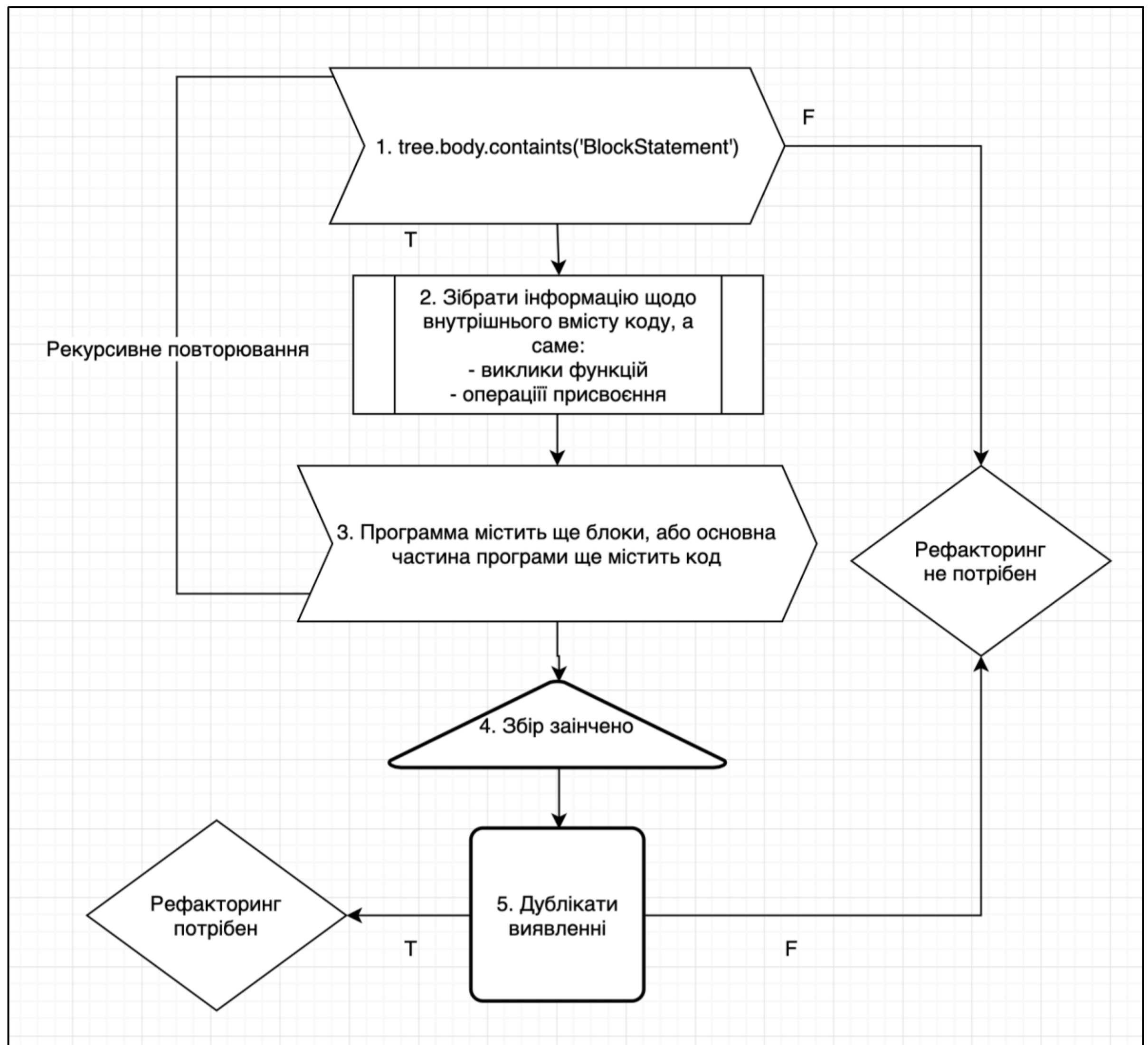

Рисунок 4.9 – Дерева прийняття рішення для виявлення та усунення дублікатів коду

Дерево прийняття рішень має 5 кроків:

– спочатку треба знайти хоча б один не пустий блок з будь яким програмним кодом. В наведеному прикладі ми маємо блок *for,* який починається на рядку номер 9;

– у наступному кроці необхідно проаналізувати та запам’ятати схему внутрішньої частини блоку для подальшого зрівняння з іншими частинами

програмного коду. На даному етапі ми маємо схему присвоєння з додаванням частини масиву;

– рекурсивно продовжити досліджувати інші блоки, або частину програмного коду, що залишилася. Маємо схожий блок *for* циклу, який починається на рядку номер 15;

– після завершення кроків номер 2 та 3, вже маються у наявності два досліджених блоки;

– на цьому кроці треба виконати аналіз зібраних даних щодо повторювальних блоків та спробувати знайти частини, які мають дублікати чи повторювальну логіку. У наведеному прикладі дублікатами є рядки 9 – 13 та рядки 15–19 відповідно;

– дублікати були знайдені, тому приведеному програмному коду потрібен рефакторинг.

Виконаємо рефакторинг для того, щоб усунути знайдені проблеми. Вигляд попереднього коду після реорганізації наведено на рисунку 4.10.

```
1   const calcAverage = array => {
2     let sum = 0;
3
4     for (let i = 0; i < 4; ++i) sum += array[i];
5
6     return sum / 4;
7   };
8
9   const array1 = [];
10  const array2 = [];
11
12  const average1 = calcAverage(array1);
13  const average2 = calcAverage(array2);
14
```

Рисунок 4.10 – Вигляд реорганізованого коду з усуненими дублікатами

Далі провидимо обчислення основних метрик якості програмного коду для реорганізованого варіанту, для того щоб запевнитись емпірично, що усунення

дублікатів покращило код. На рисунку 4.11 приведені нові метрики для коду без дублікатів.

```
{
    "maintainabilityIndex": 73.31528511149446,
    "cyclomaticComplexity": 1,
    "halsteadVolume": 304.5100753646749
}
```

Рисунок 4.11 – Нові метрики якості програмного коду для реорганізованого прикладу, без дублікатів

Згідно рисунку 4.11 можна з впевненістю казати, що виконаний рефакторинг значно покращив якість програмного коду, про свідчать покращені значення метрик:

– індекс підтримки виріс з 58 до 73 одиниць, що свідчить про суттєво покращення у розумінні коду іншими розробниками та підвищення рівня тестованості;

– значення цикломатичної складності коду було знижено з 3 одиниць до 1, що є очікуваним явищем, бо код який було наведено до реорганізації має дуже просту логіку, тому там не повинно спостерігатись високих значень для цієї метрики;

– значення об'єму Хальстеда також було суттєво знижено, це відбулось завдяки покращенню оцінок цикломатичної складності коду, індексу підтримки а також фізичному зниженні кількості операндів та операторів в даному програмному файлі.

Беручи до уваги нові метрики реорганізованого коду, можна зробити висновок, що реорганізація програмного коду з ціллю усунення дублікатів при використанні дерев прийняття рішення дала очікувані результати і має місце для використання.

4.3 Рефакторинг програмного коду на основі власного дерева прийняття рішень від користувача.

Як вже було заявлено раніше, дана система дозволяє створювати та використовувати власні дерева прийняття рішень, для того щоб дати змогу розробникам слідкувати своїм власним стандартам написання гарного та підтримуваного коду. Розглянемо приклад користувальницького стандарту написання програмного коду на основі використання композиції функцій для підвищення читабельності коду. Композиція функцій – це перетворення послідовності функцій (значень функціонального типу: функцій, анонімних функцій, лямбда–виразів, методів) в одну функцію (значення функціонального типу). При композиції результат попередньої функції стає аргументом наступної, яка, в свою чергу, повертає результат, який передається в якості а аргументу третьої і так далі). Закон асоціативності справедливий для будь–якого числа функцій. Та єдина функція, яка є результат композиції функцій *f1, f2, f3, … , fn* (в зазначеному порядку),

Розглянемо приклад програмного коду, який згодом буде перетворений з використанням прийому композиції функцій, даний код приведено на рисунку 4.12.

```
1    const incr = num => num++;
2    const multiplyByTwo = num => num * 2;
3    const subtractTen = num => num - 10;
4
5    const x = 50;
6    const incremented = incr(x);
7    const multiplied = multiplyByTwo(incremented);
8    const subtracted = subtractTen(multiplied);
9
```

Рисунок 4.12 – Приклад коду для використання композиції функцій

Даний програмний код є досить простим прикладом математичних операцій.

Розберемо покроково функціонал приведеного програмного коду: спочатку змінна *x* інкрементує своє значення за допомогою виклику функції *incr(num),* передаючи себе, в якості вхідного аргументу, далі інкрементований результат як аргумент передається у функцію *multiplyByTwo(num)*, що виконує множення вхідного значення числа на два, а в кінці даного фрагменту програмного коду, від помноженого значення віднімається десять за допомогою виклику функції *subtractTen(num),* та передачі попереднього значення як аргументу викликаємої функції. Усі ці операції можуть бути скомпоновані.

Розглянемо та опишемо принцип роботи дерева прийняття рішень для реорганізації програмного коду з використання методу композиції функцій на рисунку 4.13.

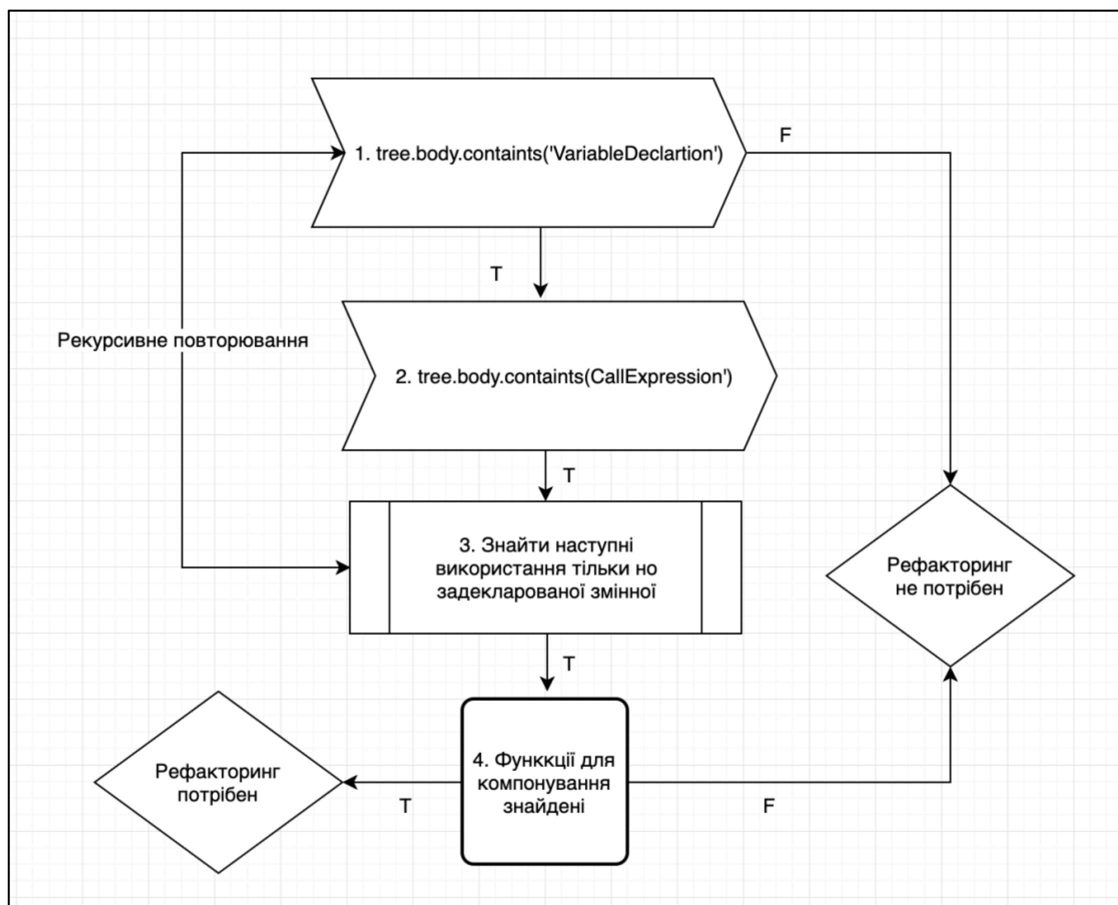

Рисунок 4.13 – Дерево прийняття рішень для реорганізації коду з використанням композиції функцій

Це дерево містить чотири основних предикати і одну рекурсивну частину.

По кроковий опис принципу роботи дерева рішення, приведеного на рисунку 4.13:

– перш за все потрібно знайти бодай одну декларацію змінною. Якщо абстрактне синтаксичне дерево вхідного програмного коду містить *VariableDeclaration* блок, то маємо підставу казати, що наданий код має оголошення змінної, тому рух далі по дереву може бути продовжений, в іншому випадку – реорганізація програмного виконана не буде. У наведеному програмному коді є оголошення змінної на 5–ому рядку, тому подальший аналіз буде проведено;

– у цьому кроці треба виявити, що знайдена зміна була оголошена в результаті виклику деякої функції, тому в абстрактному синтаксичному під–дереві блоку *VariableDeclaration* розпочинається пошук *CallExpression* блоку, що свідчить про те що, змінна була оголошена як результат виклику деякої функції, якщо ж блок функції було не знайдено, то рух далі по дереву не виконується. На рядку номер 5 мається декларація змінною *x*, але ця змінна не була оголошена в результаті роботи деякої функції, тому наступної зупинкою для дерева прийняття рішень буде рядок номер 6, де змінна була декларована як результат роботи функції інкрементування;

– надалі треба проаналізувати наступні використання змінної, яка була отримана в попередньому кроці, а саме – знайти інші декларації змінних подібних до кроку номер 2. В приведеному коді присутні такі рядки це – 7–ий та 8–ий рядки відповідно. При відсутності рядків, які підходять під ознаки компонування функцій рефакторинг виконаний не буде;

– так як були знайдені рядки, які використовують попередні значення викликаних функцій, то робиться припущення що у даному фрагменті коду можна використати прийом композиції функцій, тим самим значно скоротити кількість рядків. В приведеному випадку є 3 виклики функцій, що можуть бути скомпоновані в один єдиний;

Виконаємо рефакторинг програмного коду та розберемо отримані результати. Реорганізований код приведено на рисунку 4.14

```
1    const incr = num => num++;
2    const multiplyByTwo = num => num * 2;
3    const subtractTen = num => num - 10;
4
5    const x = subtractTen(multiplyByTwo(inc(50)));
6
```

Рисунок 4.14 – Приклад реорганізованого програмного коду за допомогою методу композиції функцій

Як можна помітити, кількість рядків була значно скорочена, тому що декілька викликів функції були переміщені в один єдиний, який знаходиться на рядку номер 5.

ВИСНОВКИ

В ході даної дослідницької роботи була проаналізована предметна область, яка стосується якості програмного забезпечення, фокусуючись на програмному коді доведено що оцінка якості програмного коду може бути перенесено в математичну площину, а також були розглянуті та опрацьовані основні алгоритми, які виконують перебудови у абстрактних синтаксичних деревах, розглянуті основні метрики якості програмного коду та проведена коротка кореляція між станом коду та значенням цих метрик, також були досліджені та проаналізовані існуючі рішення, а саме: модульна утиліта для контролю написання стандартизованого програмного коду EsLint і потужна, але ще досить молода платформа машинного навчання: яка базується на нейронних мережах DeepCode.ai.

Було розглянуто та описано доцільність використання методу дерев рішень для розпізнавання порушених архітектурних шаблонів чи правил проектування або написання програмування, дана загальна характеристика роботи методу дерев рішення.

Метрика індексу підтримки була представлена основним предметом дослідження. Під час апробації було виявлено, що реорганізований код має вищі оцінки індексу підтримки в порівнянні з початковим кодом, це є абсолютним доказом того, що дана система працює і покращує програмний код, тим самим спрощуючи процес розробки програмного забезпечення для розробників.

Також дана система має можливість створювати діаграми класів та об’єктів для більш наглядного демонстрування стану програмних компонентів та існуючих зв’яків між ними.

В подальшому планується:

– надати підтримку інтеграції для таких сервісів як: Atlassian BitBucket, Microsoft Github, GitLab для забезпечення перевірки коду у режимі реального часу;

– додати значно більше моделей для розпізнавання нових архітектурних патернів, правил та практик написання коду, тощо.

ПЕРЕЛІК ПОСИЛАНЬ

1. Janusz Laski, William Stanley Software Verification and Analysis. An Integrated, Hands–On Approach – Springer–Verlag London Limited, 2009.–205p.

2. Tharwon Arnuphaptrairong. Top Ten Lists of Software Project Risks : Evidence from the Literature Survey. IMECS International MulityConference of Engineers and Conputer Scientistis. VOL 1, March 2011.–732–737pp.

3. Handbook of Software Quality Assurance. Fourth Edition / editor G. Gordon Schulm;

4. Tom MacCabe, A Complexity Measure, IEEE Transactions on Software Engineering, SE–2, no. 4, 308 pages;

5. Bishop C. M. Neural Networks for Pattern Recognition. — Oxford University Press, 1995;

6. S. Alhusain S. Coupland R. John, Kavanagh M. Detecting Design Patterns in Object–Oriented Program Source Code by Using Metrics and Machine Learning // 13th UK Workshop on Computational Intelligence (UKCI). — 2013. — P. 244–251.

7. Satoru Uchiyama Atsuto Kubo Hironori Washizaki Yoshiaki Fukazawa. Detecting Design Patterns in Object–Oriented Program Source Code by Using Metrics and Machine Learning // Journal of Software Engineering and Applications. — 2014. — P. 983–998.

8. Kannadhasan N., Maheswari B. Uma. Machine Learning based Methodology for Testing Object Oriented Applications // Journal of Engineering and Applied Sciences. — 2015. — Vol. 10, no. 17. — P. 7400–7405.

9. J. Friedman T. Hastie R. Tibshiran. Additive Logistic Regression: a Statistical View of Boosting. — Stanford University, 1998.

10. Ruchika Malhotra Yogesh Singh. On the Applicability of Machine Learning Techniques for Object Oriented Software Fault Prediction // Software Engineering: An International Journal. — Vol. 1, no. 1. — P. 24–37.

11. John G. Cleary Leonard E. Trigg. K*: An Instance–based Learner Using an Entropic Distance Measure // 12h International Conference on Machine Learning. — P. 108–114.

12. Fowler Martin. Refactoring: Improving the Design of Existing Code. — Addison–Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999, 1999.

13. T. Kanemitsu Y. Higo S. Kusumoto. A visualization method of program dependency graph for identifying extract method opportunity // Proceedings of the 4th Workshop on Refactoring Tools. — P. 8–14.

14. Wei–Feng Pan Bo Jiang Bing Li. Refactoring Software Packages via Community Detection in Complex Software Networks // International Journal of Automation and Computing. — 2013. — P. 157–166.

15. Katsuhisa Maruyama Ken–ichi Shima. Automatic Method Refactoring Using Weighted Dependence Graphs // Proceedings of the 1999 International Conference on Software Engineering, 1999. — 1999.

16. Marian Zsuzsanna. A STUDY ON HIERARCHICAL CLUSTERING BASED SOFTWARE RESTRUCTURING // STUDIA UNIV. BABES–BOLYAI, INFORMATICA. — 2012. — Vol. LVII, no. 2.

17. N. Tsantalis A. Chatzigeorgiou. Identification of extract method refactoring opportunities for the decomposition of methods // Journal of Systems and Software. — Vol. 84, no. 10. — P. 1757–1782.

18. Murphy P.M., Pazzani M.J. ID2–of–3: Constructive induction of M–of–N concepts for discriminators in decision trees. In Proceedings of the Eighth International Machine Learning Workshop, Evanston, IL. Morgan Kaufmann.1991.Pp.183–187

19. S. Alhusain S. Coupland R. John, Kavanagh M. Detecting Design Patterns in Object–Oriented Program Source Code by Using Metrics and Machine Learning // 13th UK Workshop on Computational Intelligence (UKCI). — 2013. — P. 244–251.

20. Zsuzsanna Marian Gabriela Czibula Istvan Gergely Czibula. Using Software Metrics for Automatic Software Design Improvement // Stud Inf Control. — Vol. 21. — P. 249–258.