БАКАЛАВРСЬКА РОБОТА

БР.ІП – 43.00.00.000 ПЗ

Група ІП-22-2

Муляк Максим

2026

Івано-Франківський національний технічний університет нафти і газу

Факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Муляк Максим Зіновійович

(прізвище, ім'я, по батькові)

УДК 004.942
(індекс)

БАКАЛАВРСЬКА РОБОТА

Підходи до рефакторингу та підтримки існуючого коду
(назва роботи)

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 _____М.З. Муляк_____
(підпис, ініціали та прізвище здобувача)

Науковий керівник ____ Гобир Лідія Мирославівна, асистент _____
(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

Муляк Максиму Зіновійовичу

(прізвище, ім’я, по-батькові)

1. Тема проекту (роботи) "Підходи до рефакторингу та підтримки існуючого коду"

керівник проекту (роботи) Гобир Лідія Мирославівна, асистент

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 Цикл очищення, що застосовується при інкапсуляції інтерфейсу (рис. 1.1, ст. 17)

2 Різні об’єкти нерухомості під однаковою назвою класу (рис. 2.1, ст. 36)

3 Загальна архітектура CodeSage (рис. 2.5, ст. 40)

4 Архітектура розширення VS Code (рис. 2.6, ст. 42)

5 Результативність пакетного тестування (рис. 3.1, ст. 49)

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 _____Муляк М.З.
(підпис)

Керівник роботи _____Гобир Л.М.
(підпис)

АНОТАЦІЯ

Бакалаврська робота містить 70 сторінок, 28 рисунків, список використаних джерел із 40 найменувань.

Метою роботи є дослідження підходів до рефакторингу та підтримки існуючого програмного коду, а також аналіз методів підвищення його якості, зрозумілості та супроводжуваності.

Об'єкт дослідження: процеси рефакторингу та підтримки існуючих програмних систем.

Предмет дослідження: методи, підходи та інструменти рефакторингу програмного коду, засоби виявлення проблем у коді та покращення структури.

Результати дослідження: проведено аналіз основних підходів до рефакторингу, досліджено типові проблеми існуючого коду (code smells, антипатерни), розглянуто сучасні інструменти підтримки коду та запропоновано методологію ефективного рефакторингу.

У першому розділі розглянуто теоретичні основи рефакторингу програмного коду, визначено його цілі, принципи та роль у процесі підтримки програмних систем.

У другому розділі досліджено методи та інструменти рефакторингу, зокрема підходи до виявлення дефектів у коді, архітектурні рішення та методологію проведення рефакторингу.

У третьому розділі розглянуто практичну реалізацію підходів до рефакторингу, включаючи розробку та інтеграцію інструментів підтримки, а також проведено оцінку ефективності запропонованих рішень.

Висновок: застосування системного підходу до рефакторингу дозволяє підвищити якість програмного коду, зменшити технічний борг, покращити підтримку та подальший розвиток програмних систем.

КЛЮЧОВІ СЛОВА: РЕФАКТОРИНГ, ПРОГРАМНИЙ КОД, ТЕХНІЧНИЙ БОРГ, CODE SMELLS, АНТИПАТЕРНИ, СУПРОВІД ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, VS CODE, ЯКІСТЬ КОДУ.

ANNOTATION

The bachelor's thesis contains of 70 pages, 28 figures, and a reference list containing 40 sources.

The aim of the work is to study approaches to refactoring and maintaining existing software code, as well as to analyze methods for improving its quality, readability, and maintainability.

Research object: processes of refactoring and maintenance of existing software systems.

Research subject: methods, approaches, and tools for code refactoring, as well as techniques for detecting code issues and improving code structure.

Research results: the main approaches to refactoring were analyzed, common problems of existing code (code smells, anti-patterns) were studied, modern code maintenance tools were reviewed, and an effective refactoring methodology was proposed.

The first section describes the theoretical foundations of software refactoring, defines its goals, principles, and role in software maintenance, and analyzes common problems in software evolution.

The second section examines methods and tools for refactoring, including approaches to detecting code defects, architectural solutions, and refactoring methodologies.

The third section presents the practical implementation of refactoring approaches, including the development and integration of support tools, as well as the evaluation of the effectiveness of the proposed solutions.

Conclusion: the application of a systematic approach to refactoring improves code quality, reduces technical debt, and enhances software maintainability and evolution.

KEY WORDS: REFACTORING, SOURCE CODE, TECHNICAL DEBT, CODE SMELLS, ANTI-PATTERNS, SOFTWARE MAINTENANCE, VS CODE, CODE QUALITY.

ЗМІСТ

ВСТУП

У сучасному світі розробка програмного забезпечення є складним і безперервним процесом, який передбачає не лише створення нових систем, але й активну підтримку та вдосконалення вже існуючого коду. Зі зростанням масштабів програмних продуктів, ускладненням архітектур та швидкими змінами вимог до функціональності особливої актуальності набувають питання якості коду, його зрозумілості та підтримуваності. Накопичення технічного боргу, поява неефективних конструкцій та відсутність єдиних стандартів розробки призводять до зниження продуктивності команд і підвищення вартості супроводу програмних систем.

Актуальність теми зумовлена необхідністю застосування ефективних підходів до рефакторингу та підтримки існуючого коду, що дозволяють підвищити якість програмного забезпечення без зміни його функціональності. У сучасних умовах швидкої розробки програмних продуктів часто виникають ситуації, коли код стає складним для розуміння та модифікації. Існуючі підходи до супроводу програмного забезпечення не завжди забезпечують достатній рівень автоматизації виявлення проблем, таких як «code smells» або антипатерни, що ускладнює процес їх усунення. Тому дослідження методів і інструментів рефакторингу є важливим завданням для підвищення ефективності розробки та підтримки програмних систем.

Метою роботи є дослідження підходів до рефакторингу та підтримки існуючого програмного коду, а також розробка ефективних методів і інструментів для підвищення його якості, зрозумілості та супроводжуваності.

Завданнями дослідження є аналіз предметної області та існуючих підходів до рефакторингу програмного коду, визначення основних проблем підтримки програмних систем, дослідження методів виявлення дефектів коду (зокрема code smells та антипатернів), аналіз сучасних інструментів рефакторингу, розробка методології проведення рефакторингу, реалізація інструменту підтримки (на прикладі розширення для VS Code), а також оцінка

ефективності запропонованих рішень.

Об’єктом дослідження є процеси рефакторингу та підтримки існуючих програмних систем, що включають аналіз, модифікацію та оптимізацію програмного коду без зміни його функціональних характеристик.

Предметом дослідження є методи, підходи та інструменти рефакторингу програмного коду, включаючи виявлення проблемних ділянок коду, застосування шаблонів рефакторингу, а також використання сучасних середовищ розробки та допоміжних інструментів.

Методи дослідження включають аналіз наукових джерел для вивчення теоретичних основ рефакторингу, порівняльний аналіз існуючих підходів та інструментів, моделювання структури програмного забезпечення, експериментальний метод для реалізації та тестування інструментів рефакторингу, а також оцінювання результатів із використанням якісних і кількісних показників.

У роботі передбачається дослідження теоретичних основ рефакторингу, аналіз методів і інструментів підтримки коду, а також практична реалізація підходів до покращення якості програмного забезпечення. Особлива увага приділяється автоматизації процесів виявлення проблем у коді та інтеграції інструментів рефакторингу у середовище розробки. Очікується, що результати роботи сприятимуть підвищенню ефективності супроводу програмних систем, зменшенню технічного боргу та покращенню якості програмного продукту.

Бакалаврська робота містить 70 сторінкок, 28 рисунків, , 3 розділи, список використаних джерел із 40 найменувань.

РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ РЕФАКТОРИНГУ ТА ПІДТРИМКИ ПРОГРАМНОГО КОДУ

1.1 Поняття, цілі та принципи рефакторингу програмного коду

Підтримка програмного забезпечення продовжує бути більшою та найдорожчою частиною життєвого циклу програмної системи: підтримка чистого дизайну програмного забезпечення дозволяє розробникам програмного забезпечення швидше впроваджувати нові функції та зменшувати кількість помилок у програмі [1]. Дослідники повідомляли про ці аспекти в різних сценаріях, від великих застарілих систем мейнфреймів і критично важливих для систем до широко поширених систем з відкритим кодом.

З еволюцією, яка охоплює багато років або навіть десятиліть розробки програмного забезпечення, застарілий код часто страждає від тривалого накопичення технічного боргу. Зрештою, таке накопичення різко уповільнює процес інновацій і вимагає нетривіальних зусиль для усунення десятиліть недоліків коду та компромісів в архітектурному дизайні.

Що таке Legacy Code?

Кілька років тому я запитав у друга, як у нього справи з новим клієнтом. Він відповів: «Вони пишуть legacy code». Я одразу зрозумів, що він насправді мав на увазі, і ця думка мене сильно зачепила.

Звичайно, існує емоційно нейтральне визначення «legacy code»: це код, написаний у минулому, який підтримують, бо він працює. Але для людей, які щодня працюють зі старим кодом, «legacy code» - це справжня скринька Пандори.

«Legacy code» означає безсонні ночі та тривожні дні, проведені за розбором поганої структури, коду, який працює якимось незбагненним чином; дні додавання нових функцій, коли неможливо навіть приблизно оцінити, скільки часу це займе.

Вік коду тут абсолютно ні до чого. Люди пишуть legacy code прямо зараз, можливо, навіть у вашому проєкті.

Головна відмінність legacy code від non-legacy code - це відсутність всебічних тестів.

Ми можемо відчути це на простому думковому експерименті. Наскільки легко було б змінювати вашу кодову базу, якби вона могла «відкусити» у відповідь? Якби вона могла сказати вам, коли ви випадково порушили поведінку, яку потрібно зберегти? Було б досить легко, чи не так?

На жаль, у більшості проєктів під час редагування коду ми маємо дуже мало зворотного зв’язку щодо того, чи зміни, які ми внесли, покращили код чи погіршили його [2]. Мало хто з розробників працює повністю в темряві. Деякі мають набір ручних дій, які вони виконують над системою щоразу після перекомпіляції, щоб перевірити, чи змінили вони поведінку так, як очікували. Але що з рештою поведінки системи? Якщо вона теж не тестується, завжди є ймовірність, що ви вводите тонкі (subtle) баги.

У глибині душі ми це знаємо. Що ми робимо? Стаємо обережнішими, намагаємося розмірковувати, чи може зміна, яку ми вносимо, вплинути на щось інше, і вагаємося навіть перед рутинними рефакторингами.

Метод трохи розрісся - чи варто витягнути з нього окремий метод? Ми могли б, але якщо боїмося зламати код, то уникаємо цього.

Саме це уникання є слизьким схилом, через який люди починають вживати слово «legacy code» як лайливе.

Більшість страху при внесенні змін у великі кодові бази - це страх внести тонкі баги, страх ненавмисно щось змінити.

З тестами ви часто можете покращувати код безкарно. Для мене ця різниця настільки критична, що перекриває будь-які інші відмінності. З тестами ви можете робити код кращим. Без тестів ви насправді не знаєте, чи робите ви його кращим, чи гіршим.

Ключ до ефективної роботи з legacy code - привести його в такий стан, коли можна точно знати, що ви вносите зміни «по одній» і вони не впливають на решту системи [3]. Коли ви можете це робити, ви можете зосередитися на необхідній

роботі, отримувати реальний зворотний зв’язок і відразу стикатися з наслідками своєї роботи, а не дізнаватися про них від розлючених користувачів через місяці.

Що відбувається, якщо у вас немає потрібних тестів? Якщо ви дуже розумні, то можете дозволити собі вносити зміни без тестів. Але не варто надто радіти. Мене постійно вражає той факт, що «розумності» недостатньо. Дуже розумні люди покладаються на міркування замість тестів, але, швидше за все, ви просто стаєте надто консервативними.

Стара приказка каже: «Якщо не зламано - не чини». На жаль, така філософія призводить до справді заплутаних систем. Якщо ви не додаєте функцію найзрозумілішим способом лише тому, що хочете уникнути змін у існуючій системі, ви щойно зробили систему трохи хаотичнішою.

Виявляється, уникати тестів завдяки «розумності» може бути досить дурним рішенням.

На мою думку, найскладніша проблема в розробці програмного забезпечення - це розуміння.

Чи дійсно ми знаємо, що робить той чи інший шматок коду? Якщо ми не знаємо, що робить кожен фрагмент коду, то як ми можемо мати хоч якесь уявлення про те, скільки часу займе внесення зміни в систему?

У найгіршому випадку ніхто не може дати оцінку навіть у межах порядку величини, бо зрозуміти, що потрібно змінити, стає схожим на археологічні розкопки [4]. Вартість безпосередньо внесення зміни стає мізерною порівняно з вартістю з’ясування, що саме потрібно змінити. Можна було б просто виконувати роботу, поки збираєш інформацію для оцінки.

Багато систем існують у такому стані роками.

Ми можемо намагатися накопичувати розуміння через міркування про програмне забезпечення, але це не дуже ефективно. Нам доводиться щоразу переосмислювати все заново, коли потрібно знову це зрозуміти.

Тести відразу говорять нам, що робить певний фрагмент програмного забезпечення. Озброївшись цим знанням, ми отримуємо ґрунтовне розуміння нашої системи.

Без такого розуміння немає жодного пристойного способу дізнатися, скільки часу займе внесення зміни; ми просто знаємо, що з часом це займає дедалі більше часу.

Літографія передових напівпровідникових матеріалів (ASML) – це голландська компанія, яка проектує та виробляє літографічні машини, що є важливим компонентом у виробництві мікросхем. Оскільки її експертиза в апаратному забезпеченні є провідною у світі, потреба в логічному, моніторинговому та керуючому програмному забезпеченні зросла в геометричній прогресії. ASML має кілька систем, які дозволяють інженерам контролювати код, прогнозувати вплив змін інтерфейсу та збирати дані на всіх рівнях. Автоматизовані інструменти та робочі процеси забезпечують надзвичайно високий та узгоджений стиль кодування в усій кодовій базі ASML.

Хоча галузь програмної інженерії значно просунулася вперед, загальні зусилля з підтримки та рефакторингу в ASML не встигають за темпами вдосконалення [5]. У їхньому конкретному випадку компанія доклала величезних зусиль для стандартизації своєї кодової бази та збору даних за всіма можливими показниками; вони створили цілу екосистему інструментів, яка дозволяє інженерам візуалізувати та перевіряти систему на різних рівнях деталізації. Однак стандарти та інструменти кодування не запобігли накопиченню архітектурного технічного боргу: повідомлялося, що кілька високорівневих систем ASML страждали від низьких показників якості коду, зокрема, щодо архітектурних залежностей. Це свідчить про те, що, хоча забезпечення дотримання інструментів та стандартів кодування працює на рівні деталей реалізації, робота інженерів ASML все ще призводила до тісних зв'язків, великого розгалуження та незаконних залежностей.

У цій роботі повідомляється про роботи з технічного обслуговування, виконані на одному *функціональному кластері* в рамках кодової бази ASML, яка накопичила значну кількість технічного боргу за останні два десятиліття. Репозиторій складається з 14 компонентів : щоб дати уявлення про розмір проекту,

критично важливий компонент може складатися з понад 1000 інтерфейсів, десятків бібліотек і загалом понад 3000 файлів реалізації.

Для виконання робіт з обслуговування ми дослідили, як практично впровадити принципи проектування SOLID на компонентах ASML, щоб систематично рефакторувати та покращувати кодову базу без небажаних побічних ефектів. SOLID – це набір із п'яти принципів проектування (єдина відповідальність, відкритість/закритість, підстановка Ліскова, сегрегація інтерфейсів та інверсія залежностей), які спрямовані на підвищення зручності обслуговування, масштабованості та гнучкості об'єктно-орієнтованого програмного забезпечення (в цьому випадку C++), а також надають рекомендації щодо створення надійного та адаптивного коду.

Впровадження принципів SOLID (особливо принципу DI) призвело до кількох практичних дій, пов'язаних з архітектурним технічним боргом, накопиченим застарілим кодом ASML. Ми абстрагували ці конкретні дії в *шаблони рефакторингу* , пов'язані з діями з обслуговування та типом коду, в якому виконувалося обслуговування [6]. Ці шаблони є практичною реалізаціями принципу DIP і можуть бути застосовані та повторно використані в будь-якій кодовій базі, де прийнята багаторівнева архітектура, і не обов'язково обмежуються застарілими системами.

Головною метою цієї роботи було *«рефакторинг трьох застарілих компонентів ASML, визначених за допомогою пріоритизації на основі метрик, з впровадженням принципів SOLID-проектування».* Два дослідницькі питання, що розглядалися в цій роботі:

RQ1 Як підхід, заснований на даних, сприяє визначенню пріоритетів та зменшенню архітектурно-технічного боргу в галузевих умовах?

Обґрунтування: рефакторингові заходи – це складні починання, які - потребують значних зусиль. Це опитування має на меті дослідити, як показники процесів та продуктів можуть впливати на підхід до визначення пріоритетів діяльності з обслуговування.

RQ2 Як перевести принципи SOLID-проектування в конкретні шаблони

рефакторингу для даної системи?

Обґрунтування : принципи проектування є абстрактними за своєю природою та не вимагають тієї чи іншої реалізації. У цьому запиті ми досліджуємо, які та скільки конкретних дій було потрібно для реалізації принципу інверсії залежностей SOLID у коді ASML.

Внеском цієї роботи є звіт про досвід, який детально описує ідентифікацію, пріоритезацію та погашення архітектурно-технічного боргу з використанням принципів проектування SOLID та в галузевому контексті.

Методологію, яку ми застосували для виконання рефакторингу застарілого коду ASML, можна по суті розбити на 5 основних кроків:

1. Огляд доступних внутрішніх показників
2. Аналіз впливу
3. Пріоритезація
4. Огляд літератури
5. Впровадження рефакторингу

Коротко кажучи, у наших звітах про досвід детально описано *«* підхід на основі даних для визначення та визначення пріоритетів компонентів для рефакторингу та типу рефакторингу, який слід виконувати на кожному рівні вертикальної архітектури компонента*».* У наступних підрозділах ми детальніше проаналізуємо та опишемо підмножину вищезазначених кроків.

Огляд доступних метрик

Внутрішній процес розробки в ASML автоматично збирає низку вимірювань за певними показниками, пов'язаними з атрибутами продукту або якості. Як зовнішні дослідники, ми не мали права збирати більше показників за допомогою інструментів статичного аналізу [7]. Оскільки ми повинні були дотримуватися існуючого процесу та використовувати лише доступні показники, такий сценарій не є рідкістю та вже повідомлявся в попередніх дослідженнях розробки та підтримки програмного забезпечення.

Для моніторингу якості внутрішній інструмент ASML аналізує компоненти на основі кількох метрик та поділяє їх на метрики рівня коду [C],

архітектурні [A] та якості [Q]. Наступні вважаються «метриками рівня коду» в ASML та є підмножиною:

C1 Цикломатична складність - вимірювання складності програми. Це кількісна міра кількості лінійно незалежних шляхів через вихідний код програми. Ця метрика обчислюється за допомогою графа потоку керування кодом. Основні компоненти, як правило, мають вищу цикломатичну складність через накопичення технічного боргу. Таким чином, їхній бал зазвичай менше 45%.

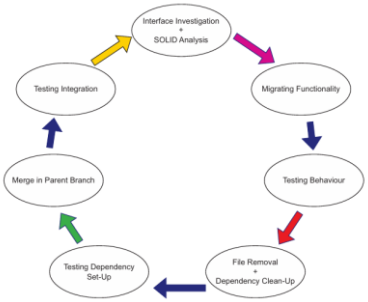


Рисунок 1.1 - Цикл очищення, що застосовується при інкапсуляції інтерфейсу.

C2 Покриття коду - цей показник відповідає кількості тестів, реалізованих для компонента. Більшість компонентів мають відносно низьке покриття тестами, зазвичай менше 50%. Через внутрішню структуру та управління продуктом немає потреби в більшому покритті коду модульними тестами.

C3 Попередження компілятора - Вимірює загальну кількість попереджень компілятора.

У ASML як показники процесу та якості розглядаються такі показники:

Стандарти кодування Q1 - Вимірює стандарти кодування на основі внутрішніх правил ASML для написання вихідних файлів, заголовкових файлів та будь-якого іншого власного формату файлів. Завдяки внутрішнім технологіям та робочим процесам стандарти кодування постійно високі для всіх компонентів. Більшість компонентів та файлів отримують рейтинг значно вище 90%.

Q2 Безпека. Цей показник враховує безпеку коду та відомі вразливості, які можна використовувати для злому програми. Код ASML загалом можна вважати надзвичайно безпечним.

Дублювання коду Q3. Принаймні 100 токенів повинні бути логічно ідентичними, щоб кваліфікуватися як дублювання коду, а рядок вважається дублікатом коду, якщо він містить принаймні один токен, який є частиною дублювання.

Наступні показники вважаються показниками архітектурного рівня:

A1 Розгалуження вентилятора - Вимірювання кількості компонентів, що використовують поточний компонент. Чим більший розгалуження вентилятора, тим більший вплив виходу модуля за межі області видимості.

A2 Абстрактна інтерпретація Абстрактна інтерпретація – це техніка, за якої обчислення програми оцінюються на абстрактних об'єктах з метою отримання результатів без виконання програми. Зазвичай це реалізується шляхом перетворення вихідного коду в модель, а не в виконуваний файл програми. Потім ця модель перевіряється на наявність шаблонів, які виявляють потенційні помилки [8]. Абстрактна інтерпретація абстрагується від явного значення змінної та натомість розглядає всі можливі значення, які вона може відобразити в певній абстрактній області.

A3 Узгоджені інтерфейси. Цей показник показує, скільки інтерфейсів безпосередньо включає певний інтерфейс.

A4 Непрямо укладені інтерфейси. Цей показник показує, скільки інтерфейсів певний інтерфейс включає опосередковано.

A5 Залежності збірки - ця метрика показує, скільки вихідних файлів залежить від одного інтерфейсу.

Аналіз впливу

Оскільки рефакторинг може впливати на численні аспекти поведінки системи (наприклад, мінімізацію складності, зменшення дублювання коду тощо), наша команда мала вирішити, який аспект сприяє найбільшій вигоді для розробників функціонального кластера. Ми проаналізували тенденції зібраних показників, і атрибути, які постійно знаходилися в «небезпечному» діапазоні, були визнані більш релевантними. Для кожного з них ми сформулювали аналіз

впливу запропонованих змін і використали його для етапу визначення пріоритетів.

Автоматичний збір метрик у процесі розробки ASML базується на порогових значеннях та позначається кольорами, як показано на рис. 1.2 нижче. Вони можуть надаватися на різних рівнях агрегації (на рівні файлів, на рівні компонентів, на рівні функціонального кластера).

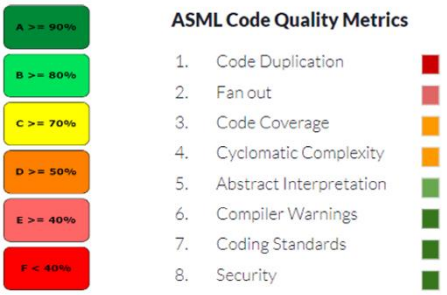


Рисунок 1.2 - Результати аналізу показників для трьох компонентів досліджуваного функціонального кластера.

Беручи до уваги знання зацікавлених сторін та тенденцію спостережуваних значень, метрики «Безпека», «Стандарти кодування», « Попередження компілятора» та «Абстрактна інтерпретація» показали вимірювання, які переважно перевищували 90 відсотків у всіх компонентах функціонального кластера. Для цілей нашої роботи з рефакторингу архітектори ASML вважали ці метрики *безпечними,* і їх було розміщено внизу нашого списку пріоритетів (тобто нерелевантними для інформування про діяльність з рефакторингу).

З іншого боку, 4 інші показники отримали нижче 50% балів за даними інструменту моніторингу ASML. Кожен з них був визнаний важливим та релевантним: було обговорено їх точне місце в нашому списку пріоритетів, особливо враховуючи *вплив* , який вони матимуть на майбутню еволюцію кодової бази та на команди розробників [9]. У звіті нижче наведено опис цих ,
чотирьох показників ризику та ймовірний вплив рефакторингу на кожен з них.

Віялоподібний розворот

Ця властивість, ймовірно, має найбільший вплив на ремонтопридатність системи, особливо у великих компонентах, які створювалися роками і навіть десятиліттями. Компоненти використовуються не лише поза межами свого структурного блоку, але й у різних *функціональних кластерах.*

Вплив. Зменшення розгалуження найбільше покращить підтримку проекту, і це не тільки порушить залежності, але й створить середовище, де важче створювати нові [10]. Розглядаючи проблему з точки зору показників проекту, можна помітити, що цей рефакторинг залучить кілька команд через володіння кодом різних проектів, і з точки зору часу це може зайняти більше місяців. Рефакторинг у такому великому масштабі вимагатиме планування та синхронізації на кількох рівнях в організації. Враховуючи обмеження цього дослідження, це буде неможливо.

Рішення та ранжування. Через важливість проблеми та її можливий вплив, їй було надано *середній* пріоритет впливу.

Дублювання коду

Цей атрибут зазвичай середній або нижче середнього. На відміну від розгалуження, цю проблему можна вирішити на рівні компонентів, вилучивши спільні блоки коду у функцію або ввівши нові типи. Це може зробити одна команда, відповідальна за код, або один розробник.

Вплив. Зменшення дублювання коду може призвести до зменшення кількості місць, де виникають незаконні залежності, що спричинить легший рефакторинг та, можливо, вилучення нового компонента.

Рішення та ранжування. Через високий вплив цієї метрики, її відповідність цілям команди та ізоляцію від інших компонентів, цій метриці було надано *високий* пріоритет, коли мова йшла про критерії рефакторингу.

Покриття коду

Серед більшості компонентів ми спостерігали низьке покриття коду. Інженери зазвичай не пропускали тестування критичного коду, але в іншому випадку вони покладалися на симульовані функціональні тести, щоб перевірити правильність роботи системи. Ще однією причиною такого рішення є те, що

покриття коду не відповідає безпосередньо потребам команди.

Вплив. Обговорення з архітекторами підкреслило той факт, що дублювання коду є проблемою поточного робочого процесу, а не архітектурним недоліком. Додавання більшої кількості тестів, безумовно, було б бажаним, але це не зменшить залежності коду, що є кінцевою метою.

Рішення та ранжування. З вищезазначених причин цьому показнику було надано *нижчий* пріоритет впливу: рішення випливало з внутрішньої комунікації з командами та робочими процесами.

Цикломатична складність

Спільною проблемою для великих компонентів, як правило, є накопичення цикломатичної складності, яка накопичувалася протягом двох десятиліть розробки [11]. Однак цикломатична складність стосується базової структури коду всередині самого компонента та методів. Після перевірки компоненти, як правило, мають нижчі показники цикломатичної складності через численні перевірки операторів if, які виконуються після кожної дії. Однак їх нелегко рефакторувати, оскільки після кожної операції потрібно перевірити позитивний результат, і машина може перейти до наступного завдання. Ці перевірки є критично важливими, тому їх виконання є вимогою ASML. Таким чином, цикломатична складність не надає повністю точних даних через природу специфічної для предметної області кодової бази.

Вплив. Рефакторинг коду з урахуванням цієї проблеми, найімовірніше, покращить читабельність та зручність підтримки компонента, але не зменшить кількість незаконних залежностей, що встановлюються в компонент. З огляду на метрики проекту, можна буде зменшити складність окремого компонента, але це займе більшу частину часу на дослідження. Зрештою, це не відповідає меті команди щодо зменшення залежностей.

Рішення та ранжування. Цей показник не має прямого впливу на накопичення незаконних залежностей, тому йому буде надано *найнижчий можливий* пріоритет впливу.

Пріоритизація

ASML-архітектор був доступний для опису основних архітектурних рішень, прийнятих за останні 20 років на основі кодової бази, що належить до функціонального кластера, разом з результатами зібраних даних. Як аналіз впливу, так і визначення пріоритетів стали цінними завдяки кількісному аналізу та висновкам, які ми змогли отримати від архітектора.

На основі аналізу метрик ми сформулювали список пріоритетів на основі 5 рівнів: *Найвищий, Високий, Середній, Низький* та *Найнижчий.* Критерії були побудовані на основі дослідження метрик компонентів разом з архітектором ASML в рамках дослідження. Зокрема, ми обговорили проблему розгалуження компонентів, проте через масштабність завдання було погоджено, що вона повинна мати нижчий пріоритет впливу в рамках дослідження. Однак, у системі, нею повинні займатися кілька команд. ASML не був особливо зацікавлений у дублюванні коду, хоча це створює серйозну проблему: через внутрішні робочі процеси деякий код обов'язково копіювався кілька разів. Однак архітектори ASML погодилися, що в рамках дослідження найкраще залишити це як метрику з високим пріоритетом.

Після ретельного обговорення та на основі доступних показників, критерій пріоритезації на основі даних наведено в таблиці 1.1 :

Таблиця 1.1

Аналіз впливу на окремі показники.

Назва (Метрика)	**Пріоритетність впливу**
Зменшення незаконних залежностей (Reduce Illegal Dependencies)	Найвищий
Дублювання коду (Code Duplication)	Високий
Розгалуження (Fan out)	Середній
Покриття коду (Code Coverage)	Низький
Цикломатична складність (Cyclomatic Complexity)	Найнижчий

1.2 Проблеми підтримки та еволюції існуючих програмних систем

Під час огляду літератури ми визначили один конкретний принцип

SOLID, якого слід дотримуватися під час рефакторингу, – принцип інверсії залежностей (DI). Враховуючи цей принцип проектування, ми виділили макрокомпоненти, які (а) потребували рефакторингу та (б) мали б максимальний вплив після рефакторингу. Практична реалізація принципу DI була зосереджена на пошуку повторюваних шаблонів, а не на окремих одноразових діях. Довгострокова перспектива полягала в поширенні реалізації DI на інші компоненти коду, які не були включені до нашої роботи [12].

Використовуючи інформацію, надану архітектором, та метрики (з їх пріоритетністю), наступним кроком нашої методології був вибір підмножини компонентів з найвищим пріоритетом впливу, які забезпечили б найбільший вплив після рефакторингу.

У таблиці показано кількість незаконних залежностей (як вхідних , так і вихідних) компонентів у функціональному кластері. Виділено три компоненти, яким було надано пріоритет для робіт з технічного обслуговування в ASML, виходячи з великої кількості незаконних залежностей. Нижче ми описуємо кожен компонент більш детально.

Компонент А

У таблиці показано, що компонент з позначкою "А" має 2 незаконні вихідні залежності (наприклад, розгалуження). З огляду на рис. 2(b) вище, ми знаємо, що ці залежності повільно з'являлися протягом еволюції цього компонента, і вони є вхідними стосовно компонентів з позначками "B" та "C" відповідно, що становить 100% його незаконних залежностей. З доступного набору метрик ми також знаємо, що цей компонент страждає від *дублювання коду* та *розгалуження* : виходячи з двох вищезазначених аспектів (незаконні залежності мають найвищий пріоритет впливу, а дублювання коду - другий за величиною), рефакторинг цього компонента матиме найбільший вплив, якщо його буде рефакторовано [13]. У свою чергу, це також позитивно вплине на
компоненти B та C, які також будуть об'єктом рефакторингу.

Компонент Б

Цей компонент отримує високий бал на основі критеріїв пріоритету

впливу для таких показників: розгалуження, цикломатична складність та покриття коду. Однак як цикломатична складність, так і покриття коду мають низький пріоритет впливу, тому загальна важливість рефакторингу компонента B не буде такою високою, як у компонента A. Зменшення незаконного розгалуження матиме значні наслідки для зменшення залежностей, проте, як обговорювалося раніше, це виходить за рамки дослідження. Рефакторинг незаконних залежностей компонента B відповідає лише за 1 вхідне з'єднання: це становить лише 50% незаконних залежностей у рамках дослідження, що вдвічі менше порівняно з компонентом A.

Компонент C

Компонент, позначений як "C", є поширеним компонентом, призначеним для використання іншими комп'ютерами, потенційно різних типів. З таблиці 2 ми бачимо, що компонент B має 3 незаконні вхідні залежності, але лише 1 з точки зору обсягу дослідження. Це становить 50% незаконних залежностей в області дослідження. Рефакторинг A повністю очистить цю залежність. Це призводить до рішення, що C має низький бал з точки зору метрик, що стосуються незаконних залежностей. Компонент C підтримувався в хорошому стані протягом своєї розробки, що можна побачити за переважно високими балами за метриками якості коду. Компонент не має жодної важливої метрики з балом менше 70%, а загальна якість компонента позначена на рівні 85%. Єдиними пунктами для покращення можуть бути Fan Out з балом близько 70% та покриття коду з балом нижче 40%. Покриття коду має найнижчий пріоритет впливу на основі наших критеріїв, і завдяки якості компонента B порівняно з компонентом B, йому буде присвоєно третє та останнє місце в нашому списку рефакторингу компонентів.

Поточний стан компонентів A, B та C

Спочатку єдиними компонентами, присутніми у функціональному кластері, були B та C (див. рис. 1.3(a)). Кожен блок представляє компонент, і кожен компонент може складатися з кількох інтерфейсів, бібліотек, виконуваних файлів та файлів реалізації.

На рис. 1.3 (b) показано, як виглядає стан системи після двох десятиліть розробки. Ми бачимо, що 2 компоненти: A та B належать до категорії *машинно-залежних компонентів*, а один з них (C) є *спільним компонентом* . Стрілки показують напрямок залежностей. При попередньому візуальному огляді можна побачити кілька недопустимих залежностей, що виходять від компонента A до компонентів B та C. На рис. 1.3 (c) показано бажану архітектуру між цими компонентами після усунення порушень архітектури. На рис. 1.3 (d) показано позначення, які будуть використовуватися для представлення зв'язків для всіх діаграм нижче:

- Суцільна стрілка - відображає зв'язок «Використання». Це означає, що компонент використовує деякі з експортованих ззовні бібліотек з іншого компонента.
- Пунктирна лінія — відображає зв'язок «Експорт/Використання». Це означає, що компонент не тільки експортує функціональність для використання в інших компонентах, але й використовує цю функціональність внутрішньо.
- Білий блок — представляє машинно-залежний компонент типу 1.
- Заштрихована рамка — представляє машинно-залежний компонент - типу 2.
- Пунктирна рамка представляє загальні компоненти. Ці компоненти - призначені для повторного використання на різних типах машин.

Компонент A незаконно залежить від 6 модулів у компоненті B та 3 модулів компонента C. Необхідність обслуговування цих компонентів чітко показана у з'єднаннях, показаних як «безпосередньо оброблювані інтерфейси», «непрямо оброблювані інтерфейси» та «залежності збірки» у таблиці 3 .

Проблеми із залежностями, які ми запропонували усунути для досягнення найбільшого ефекту, ґрунтуються на зв'язках, що порушують принцип залежності у версіях.

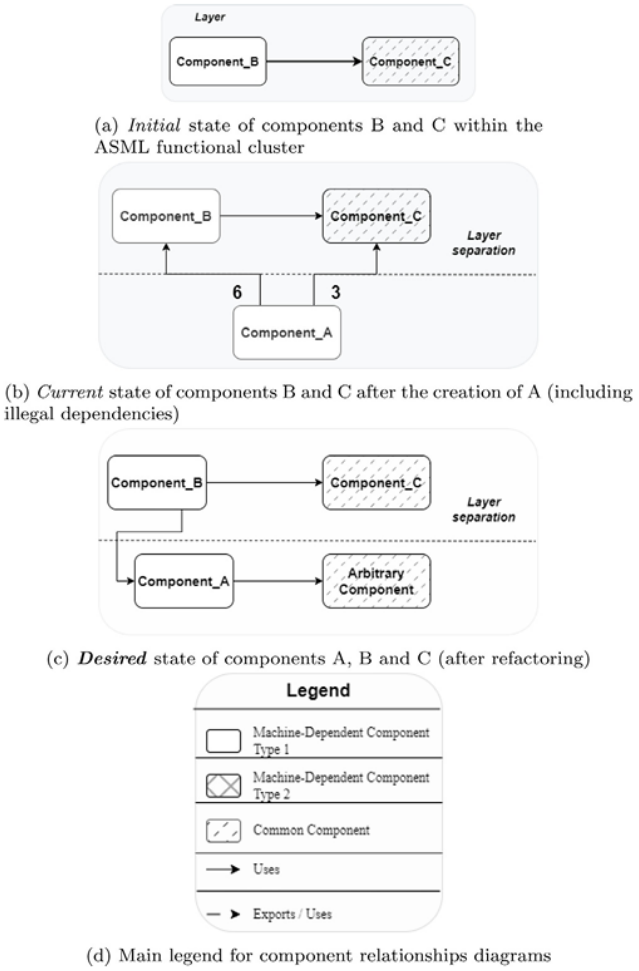
У цьому розділі ми ілюструємо, як ми втілили принцип залежності у версіях у практичні дії, використовуючи такі рекомендації:

1. високорівневі модулі не повинні залежати від низькорівневих

модулів;

2. високорівневі та низькорівневі модулі повинні залежати від абстракцій;

3. Абстракції повинні визначати інтерфейс (не конкретну реалізацію), якого дотримуються високорівневі та низькорівневі модулі.



(a) *Initial* state of components B and C within the ASML functional cluster

(b) *Current* state of components B and C after the creation of A (including illegal dependencies)

(c) ***Desired*** state of components A, B and C (after refactoring)

(d) Main legend for component relationships diagrams

Рисунок 1.3 - Динаміка нелегальних взаємозалежностей між трьома компонентами, що аналізувалися в дослідженні

Подібно до існуючих шаблонів рефакторингу (наприклад, метод вилучення, клас вилучення, метод переміщення тощо), наші шаблони рефакторингу мали на меті розробити систематичний підхід до покращення архітектурної структури існуючого коду без зміни його зовнішньої поведінки. Розроблені нами шаблони поділяються на два основні типи залежно від типу машин, для яких був створений застарілий код: шаблони рефакторингу для

машинно-залежного коду та шаблони рефакторингу для машинно-незалежного коду [14].

Наведені нижче шаблони задокументовані з використанням спільної структури та візуально відображають стан «ЯК Є» (поточний стан компонентів до рефакторингу) та стан «ЯК БУТИ» (бажаний стан компонентів після рефакторингу). Хоча діаграми відносяться до коду ASML, деталі реалізації нечіткі, що робить їх залежними від проекту та придатними для повторного використання в різних типах проектів програмної інженерії.

Машинно-залежні шаблони

Нижче ми наведемо приклад шаблону рефакторингу для рефакторингу коду, який залежить від конкретної машини та не прив'язаний до спільного компонента. Ми використовуємо структурований підхід для опису кожного шаблону та додаємо візуалізацію шаблону «до» та «після» рефакторингу. Для кращої читабельності рукопису два інші машинно-залежні шаблони.

Намір розмістити спільні експортовані символи у відповідному компоненті через порушення залежностей.

Мотивація. Через порушення залежностей лише один компонент використовує символи, і цей компонент знаходиться на логічно нижчому рівні.

Застосовність Компонент нижчого рівня залежить від модуля вищого рівня та використовує підмножину його експортованих символів.

Реалізація Незаконно експортовані символи слід розмістити в модулі нижчого рівня та інкапсулювати. Реалізація цього шаблону рефакторингу полягає у переміщенні експортованих символів у модулі нижчого рівня.

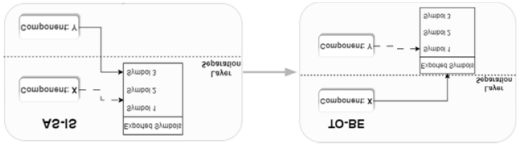


Рисунок 1.4 - Структура та реалізація шаблону рефакторингу: «Експортовані символи»

Переваги та компроміси. Реалізація цього шаблону покращує дизайн системи, забезпечує сильнішу інкапсуляцію, розрив залежностей, а також гарантує, що з цих символів знову не можуть виникнути незаконні залежності.

Наслідки. Розміщенням списку експортованих символів на найнижчому логічному рівні залежність було інвертовано, а отже, вирішено. Список експортованих символів було інкапсульовано, що гарантує відсутність незаконної залежності внаслідок їх використання.

1.3 Основні підходи та шаблони рефакторингу коду

У цьому підрозділі ми наводимо приклад шаблонів рефакторингу, які слід використовувати під час рефакторингу залежностей, що включають спільні компоненти. Подібно до випадку, залежного від машини, інші шаблони, незалежні від машини, були описані нижче в Додатку В.

Ці шаблони були розроблені ще раз протягом дослідження, на основі критеріїв та аналізу впливу, які ми створили, а також у співпраці з архітекторами програмного забезпечення ASML. Для цих типів шаблонів було досліджено внутрішню документацію щодо програмного забезпечення ASML, щоб отримати більше знань про предметну область щодо спільних компонентів та того, як вони повинні використовуватися.

Усі діаграми були показані архітектору ASML та неодноразово переглядалися, щоб гарантувати, що кінцевий результат діаграми є правильним відображенням стану системи [15]. Рефакторинг із спільних компонентів та в спільні компоненти є складним завданням, яке вимагає специфічних знань про внутрішню структуру системи та компонента, щоб прийняти обґрунтоване рішення про те, в який компонент потрібно перемістити фрагмент коду.

Мета Перемістити експортований список символів із спільного компонента, який використовується кількома машинно-залежними компонентами різних типів, до іншого спільного компонента, що знаходиться на рівні залежності найнижчого рівня .

Мотивація: експортований список символів слід перемістити до спільного компонента, який знаходиться на рівні залежності найнижчого рівня. Компонент, в якому буде розміщено код, залежатиме від проекту.

Застосовність: машинно-залежний компонент має недопустиму залежність від спільного компонента, і не тільки це, але й ці символи взаємно спільні для кількох компонентів.

Реалізація повинна переміщувати експортований список символів до спільного компонента на найнижчому можливому рівні дерева залежностей. Компонент, в якому буде розміщено код, залежатиме від проекту та ієрархії.

Переваги та компроміси Список експортованих символів було переміщено до відповідного спільного компонента. Це розділяє компоненти та забезпечує кращу зв'язність.

Наслідки. Список символів має бути загальним компонентом, який служить правильному призначенню. Якщо такого компонента не існує на потрібному рівні, тоді слід створити компонент для зберігання цих символів.

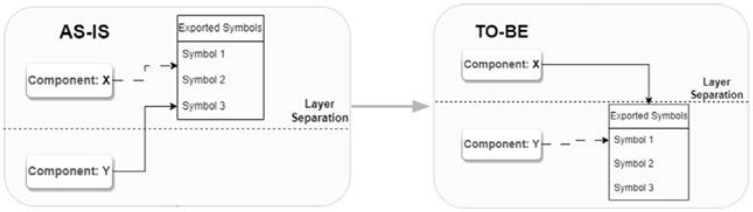


Рисунок 1.5 - Структура та реалізація шаблону рефакторингу: «Експортовані символи із загального компонента»

Протягом процесу рефакторингу кожен рефакторинг слід класифікувати за одним із семи описаних вище шаблонів. Через природу шаблонів, їх у більшості випадків можна застосовувати паралельно: це означає, що розробники не обмежуються іншими та можуть працювати незалежно з кодовою базою, не враховуючи послідовність виконання [16]. Це дозволяє розробникам бути більш продуктивними та розподіляти робоче навантаження з рефакторингу між кількома командами, які працюватимуть одночасно над процесом рефакторингу. Це

показує, що процес пріоритизації на основі даних може не лише стандартизувати рефакторинг у великій компанії, але й забезпечити паралелізацію на рівні завдань.

Застосування шаблонів рефакторингу для зменшення незаконних - залежностей

Рефакторинг залежностей між компонентом A та B

Після рефакторингу 6 інтерфейсів ми ефективно подолали одну вихідну та одну вхідну незаконну залежність. Це зменшило незаконні залежності на 33% в рамках проекту. Тепер важливо класифікувати шаблони рефакторингу, які ми використовували, з точки зору їхньої загальної важливості. Важливість буде розрахована відносно кількості залежностей експортованих символів, вирішених кожним з них. Ці дані вказуватимуть на найпоширеніший тип залежності, а також показуватимуть, як їх можна вирішити.

З таблиці можна зробити висновок, що з 294 залежностей дві третини було вирішено за допомогою першого шаблону, третина – за допомогою другого, і лише одна – за допомогою третього. Важливо зазначити, що понад 85% від загальної кількості залежностей, рефакторованих за допомогою шаблону 2, були знайдені в одному інтерфейсі, що спотворює діаграму та вносить упередженість. Зі зібраних даних можна зробити висновок, що більшість залежностей від експортованих символів у всій системі можна вирішити, дотримуючись діаграми, показаної на рис. 3. Це також вказує на те, що більшу частину рефакторингу можна виконувати паралельно, оскільки реалізація першого шаблону не вимагає послідовних кроків між рефакторингом різних експортованих символів. Це показує, що методології пріоритизації на основі даних можуть масштабуватися в корпоративному середовищі та дозволяти кільком командам працювати разом після збору критеріїв та даних.

Спостерігали, що більшість залежностей спільні для кількох компонентів. З одного боку, це було очікувано, оскільки спільні компоненти призначені для повторного використання, що показує, що внутрішньо вони використовуються так, як передбачає їхня конструкція. З іншого боку, це означає, що рефакторинг залежностей спільних компонентів є складнішим і має виконуватися кількома

командами. Хоча процес рефакторингу можна виконувати паралельно, кільком командам доведеться узгодити нову структуру, що потенційно уповільнить процес.

Видно, що понад 80% від загальної кількості залежностей було вирішено за допомогою другого шаблону рефакторингу спільних компонентів. Це вказує на важливість варіанту використання та на те, що, найімовірніше, більшість залежностей у спільних компонентах можна вирішити, дотримуючись його. Цей шаблон, зокрема, не обертається навколо жодних послідовних дій, що робить процес рефакторингу легко паралелізованим, проте через природу спільних компонентів їх рефакторинг є складнішим через необхідність координації між командами та пріоритетами [17]. Оскільки це виходить за рамки цього дослідження, ці залежності не були рефакторовані, було рефакторовано лише 9 залежностей, які можна було рефакторувати в машинно-залежний компонент з обсягу дослідження. Хоча A все ще має 3 залежності від B, ми зменшили залежності експортованих символів до B на 20%

1.4 Висновок по розділу

У першому розділі було розглянуто теоретичні основи рефакторингу та підтримки існуючого програмного коду. Проаналізовано поняття, основні цілі та принципи рефакторингу, спрямовані на покращення структури та якості програмного забезпечення без зміни його функціональності. Досліджено проблеми підтримки та еволюції існуючих програмних систем, зокрема складність супроводу, накопичення технічного боргу та зниження зрозумілості коду. Також розглянуто основні підходи та шаблони рефакторингу, які використовуються для підвищення підтримуваності та масштабованості програмних рішень. Отримані результати формують теоретичну основу для подальшого дослідження методів і практичних засобів рефакторингу.

РОЗДІЛ 2. МЕТОДИ ТА ІНСТРУМЕНТИ РЕФАКТОРИНГУ І ПІДТРИМКИ КОДУ

2.1 Архітектура програмних модулів та інструментальні засоби рефакторингу

CSS (каскадні таблиці стилів) – це мова кодування, яка надає веб-сайту зовнішнього вигляду та макета. Поряд з HTML, CSS є основоположним для веб-дизайну. Без нього веб-сайти все ще були б простим текстом на білому фоні. CSS дозволив запровадити кілька інновацій у макеті веб-сторінок, такі як можливість:

1. Вкажіть шрифти, відмінні від стандартних для браузера
2. Вкажіть колір і розмір тексту та посилань
3. Застосування кольорів до фону
4. Містити елементи веб-сторінки в блоки та розміщувати ці блоки в певних позиціях на сторінці

Це важливий інструмент для електронної комерції, оскільки він дозволяє розробникам створювати власні веб-сайти в більшості випадків без зміни готових шаблонів HTML. За допомогою CSS можна впливати на елементи веб-сайту глобально, а не змінювати їх по одному.

Bootstrap 4

Bootstrap 4 - це бібліотека HTML, CSS та JavaScript, яка допомагає розробникам спростити процес створення веб-сайтів. Коли ви додаєте bootstrap до проекту, він надає базові стилі елементам веб-сайту за допомогою HTML-класів. В результаті ви отримуєте єдиний вигляд елементів на веб-сайті, наприклад: таблиці, форми, поля введення, прапорці. Розробники також можуть використовувати класи CSS, визначені в Bootstrap, для подальшого стилізації зовнішнього вигляду свого вмісту [18]. Веб-сайт за замовчуванням, розроблений командою eCommerce, був створений за допомогою Bootstrap 4. Ця бібліотека дозволяє розробникам заощаджувати час на стилізації веб-сайтів, оскільки вона має вбудовані правила стилю за замовчуванням.

CSS-коди Sass

Sass (Syntactically Awesome Style Sheets) - це мова сценаріїв препроцесора. Вона компілюється в CSS (Cascading Style Sheets). Існує кілька причин, чому Sass використовується розробниками електронної комерції.

Змінні

Перша причина полягає в тому, що це дозволяє використовувати змінні. Змінні у веб-розробці дозволяють зберігати значення в одному місці, а потім посилатися на нього в кількох інших місцях. Досить важливо мати змінні під час роботи над проектом з великою кількістю CSS, часто з багатьма повторюваними значеннями, оскільки це економить багато часу.

Вкладений CSS

Друга причина полягає в тому, що препроцесор Sass дозволяє розробникам писати вкладений CSS. Вкладений CSS не лише зменшує кількість рядків CSS у файлі, але й дозволяє дотримуватися тієї ж візуальної ієрархії HTML.

Міксини

«Міксини» – це третя причина використання препроцесора Sass. Вони дозволяють розробникам визначати шаблони пар значень властивостей, які потім можна повторно використовувати в інших правилах. Іншими словами, «міксини» допомагають розробникам не повторюватися, багаторазово записуючи ті самі властивості.

Назва «mixin» – це селектор класу, який ідентифікує оголошуваний «mixin». Використання «mixins» може потенційно скоротити час, витрачений на написання правил стилю для елементів.

Команди Gulp

Щоб створити новий проєкт, розробник виконує команду Gulp. Gulp - це кросплатформний раннер, який дозволяє програмістам автоматизувати багато завдань розробки, щоб розробникам не доводилося виконувати їх вручну. Це допомагає автоматизувати процеси веб-розробки [19]. У нашому випадку це допомагає мініфікувати файли та уникнути завантаження CSS-файлів та шаблонів HTML вручну.

Гіт

Репозиторій або просто Repo - це простір для зберігання, де розробники можуть зберігати свої проекти. Репозиторій може бути локальним для папки на комп'ютері або може бути каталогом на GitHub чи іншому онлайн-хостингу. У репозиторії можна зберігати всілякі файли. Крім того, його можна використовувати для відстеження змін у будь-якому наборі файлів. Часто його використовують для координації роботи між розробниками, які спільно працюють над вихідним кодом під час розробки програмного забезпечення. Його основні цілі включають цілісність даних, швидкість та підтримку розподілених, нелінійних робочих процесів (Git-Scm nd).

У стилях веб-сайтів електронної комерції за замовчуванням папка модуля складається з файлів "variables", **"colors"** та "component". У нашому випадку файл **"variables"** містить змінні для точок зупинки, шрифтів та деяких властивостей для кнопок, наприклад, їх висоти. Змінні дозволяють розробникам значно скоротити час написання та зміни стилів веб-сайту. Файл під назвою **"colors"** містить змінні, пов'язані з кольором, що використовується на веб-сайті. Наприклад, змінні для зміни кольору тексту, елементів, фону та меж. Файл **"components.scss"** містить додаткові правила стилю для випадкових компонентів, що використовуються на веб-сайті, **"mixins"** та медіа-запитів. Цей файл був додатково рефакторований у проекті.

Папка модулів

У стилях за замовчуванням для веб-сайтів електронної комерції папка модуля складається з "змінних", **"кольорів"** та файлу "компонент". У нашому випадку файл **"змінні"** містить змінні для точок зупинки, шрифтів та деяких властивостей для кнопок, наприклад, їх висоти. Змінні дозволяють розробникам значно скоротити час написання та зміни стилів веб-сайту. Файл під назвою **"кольори"** містить змінні, пов'язані з кольором, що використовується на веб-сайті. Наприклад, змінні для зміни кольору тексту, елементів, фону та меж. Файл **"components.scss"** містить додаткові правила стилю для випадкових компонентів, що використовуються на веб-сайті, **"міксинів"** та медіа-запитів. Цей файл був

додатково рефакторований у проекті.

Папка компонентів

Папка компонента складається з файлів із CSS-класами, які визначають властивості певного компонента [20]. Компоненти є багаторазово використовуваними, самодостатніми та незалежними елементами інтерфейсу користувача. Наприклад, кошик, нижній колонтитул, заголовок, оформлення замовлення, карусель тощо. Кожен із перелічених компонентів також може бути створений з інших компонентів, наприклад, карусель на веб-сайті може містити кілька карток товарів, які, у свою чергу, складаються із зображення, кнопки покупки, рядка з кодом товару або ціною. Під час роботи над великим проектом гарною практикою є розділення стилів на менші частини, щоб код був більш зрозумілим, читабельним та легшим у роботі.

"styles.scss" - Файл

Головна мета цього файлу - об’єднати всі стилі за замовчуванням в один. Для цього всі компоненти та модулі вставляються в цей файл за допомогою правила @import. Правило CSS «@import» використовується для імпорту правил стилів з інших таблиць стилів. Загальною практикою є зберігання всіх файлів імпорту в одному, щоб об’єднати їх усі та зберегти просту та зрозумілу архітектуру файлів.

2.2 Методи виявлення проблем у програмному коді

Розробка та підтримка стандартного веб-сайту електронної комерції вимагає постійного вдосконалення, наприклад, додавання нових функцій, зміни стилів або виправлення помилок. Цей процес вимагає добре продуманої архітектури файлів та добре структурованих таблиць стилів.

Добре структурована архітектура позбавляє розробників від витрат часу на пошук правильного рядка коду та роздумів, куди додати нову властивість. Погана структура файлів може спричинити конфлікти, перекриття та помилки. CSS не має конфліктів, оскільки до одних і тих самих елементів можна застосовувати кілька

правил, а їхній порядок відображає «важливість». Наприклад, якщо елемент з'являється останнім, він замінить попередні правила. Але для збереження гарної структури рекомендується розділяти стилі на менші компоненти. Також важливо уникати використання глобальних стилів, які можуть впливати на всі елементи веб-сайту.

Існує кілька проблем у стилях за замовчуванням веб-сайтів електронної комерції, через які їх не можна назвати добре структурованими.

Перекриваючі стилі

Кожен елемент на веб-сайті зазвичай має клас або ідентифікатор. Щоб стилізувати елемент, потрібно звернутися до елемента, написавши його клас або ідентифікатор, а потім надавши йому властивості. Зазвичай цей процес не викликає жодних проблем. Однак, якщо є кілька елементів з однаковими класами або ідентифікаторами, які мають різні властивості, це може призвести до перекриття стилів (рисунок 2.1).

```
.price {
    min-height: 35px !important;
    .unit-text {
        font-size: 0.7em !important;
    }
    .price_steps {
        font-size: 10px !important;
        display: block !important;
        margin-top: -3px !important;
    }
}
```

```
.price {
    position: absolute;
    right: 10px;
    bottom: 5px;
    padding-top: 6px;
    padding-bottom: 6px;
    font-weight: bold;
    font-size: 1.4em;
    .text-right{
        font-weight: normal;
    }
}
```

Рисунок 2.1 - Різні об’єкти нерухомості під однаковою назвою класу.

Вирішення проблеми перекриття за допомогою елементів «важливості»

Існує кілька способів уникнути перекриття стилів. Перший – додати тег «!important» після значення, що йде за властивістю (рисунок 2.2). Він використовується для додавання більшої важливості значенню та перевизначення правила стилізації. У деяких випадках необхідно додати правило «important», наприклад, якщо розробник хоче перевизначити вбудовані стилі бібліотеки. У

нашому випадку ця бібліотека – Bootstrap. Вбудовані стилі Bootstrap мають більшу «важливість», ніж стилі з підключеного CSS-файлу, тому розробнику може знадобитися додати властивість «important», щоб перевизначити стилі Bootstrap.

Однак, краще уникати властивості «important», оскільки вона ускладнює налагодження та підтримку стилів для розробників [21]. Щоб змінити стиль елемента, краще знайти рядок коду, який потрібно змінити, та замінити його значення. Таким чином, розробники можуть зменшити кількість рядків у CSS-файлах та зберегти їх чистішими.

```
.product-card_invoice_alert{
    display: block !important;
    padding-left: 7px !important;
    padding-right: 7px !important;
    cursor: help;
    word-break: break-all;
}
```

Рисунок 2.2 - Властивості стилю, перезаписані за допомогою правила «!important».

Вирішення проблеми перекриття шляхом визначення батьківського елемента

Другий спосіб уникнути конфліктів у стилях – це вказати їхній батьківський елемент. Цього можна досягти, просто написавши клас або ідентифікатор батьківського елемента перед назвою фактичного елемента. Однак, цей підхід також можна покращити, вкладаючи одне правило стилю всередину іншого. У нашому випадку цей метод можливий завдяки Sass.

Вкладений CSS підвищує модульність та зручність підтримки таблиць стилів і значно зменшує обсяг написаного CSS. Це також допомагає зберігати всі стилі, пов'язані з елементом, безпосередньо під його батьківським елементом (рисунок 2.3).

```
.right-productcard-box{
    .product-card_invoice_alert{
        display: block ;
        padding-left: 7px ;
        padding-right: 7px ;
        cursor: help;
        word-break: break-all;
    }
}
```

Рисунок 2.3 «product-card-invoice_alert» — клас та його стильові властивості всередині батьківського елемента - «right-productcard-box».

Різні розміри екрану

Сьогодні важливо створювати версії веб-сайтів не лише для настільних комп'ютерів, але й для мобільних телефонів та планшетів. Для цього елементи, що використовуються на веб-сайті, повинні мати різні властивості залежно від розміру екрана.

```
<div class="col-6">
    <picture class="group-image">
        %if has_fallbacks%
        %if has_original_thumbnail_2%
        <source srcset="%original_thumbnail_2%" type="image/%original_type%">
        %/if has_original_thumbnail_2%
        <source srcset="%original_full_path%" type="image/%original_type%">
        %/if has_fallbacks%
        %if has_thumbnail_2%
        <source srcset="%thumbnail_2%" type="image/%type%">
        %/if has_thumbnail_2%
        <source srcset="%full_path%" type="image/%type%">
        <img src="%full_path%" class="img-fluid" alt="%alt_text%"/>
    </picture>
</div>
```

Рисунок 2.4 - Клас Bootstrap «col-6», що використовується у стандартному HTML-шаблоні.

Класи Bootstrap

Існує кілька методів, які електронна комерція використовує для створення станів елементів для різних пристроїв. Перший – це використання бібліотеки Bootstrap. Bootstrap має вбудовані класи, наприклад, « **col-6»** , які дозволяють розробникам встановлювати певну ширину елемента на основі розміру екрана без написання CSS (рисунок 2.4).

Медіа-запити в CSS-коді

Другий метод, який команда електронної комерції використовує для визначення поведінки елемента на основі розміру екрана, – це написання медіа-запитів у файлах CSS. «Медіа-запити – це функція CSS3, яка дозволяє вказувати, коли слід застосовувати певні правила CSS. Це дозволяє застосовувати спеціальний CSS для мобільного перегляду. Перевага цього методу полягає в тому, що завантажується лише дійсний CSS» (CSS Media queries nd).

Медіазапити можна зберігати як в окремому файлі, так і у файлі з іншими правилами CSS, але важливо забезпечити легкість їх підтримки. У нашому випадку медіазапити розташовані випадковим чином у більшості файлів компонентів. Краще зберігати медіазапити в їх батьківському елементі. Таким чином, легше та швидше знайти певне правило медіазапиту та змінити його значення за потреби.

Великі файли

Щоб файл CSS був чистим та добре структурованим, гарною практикою є розділення стилів на менші компоненти. У стилях електронної комерції за замовчуванням є кілька файлів, які містять властивості різних елементів веб-сайту. Це не лише призводить до перекриття стилів, але й робить ці файли дуже великими та важкими для підтримки.

Щоб виправити цю проблему, я рефакторингую ці файли та розподіляю код всередині них на окремі компоненти. Це допоможе зберегти кращу архітектуру файлів стилів та спростить розробникам підтримку CSS-коду.

Існує кілька проблем зі стилями за замовчуванням на веб-сайтах електронної комерції, які заважають роботі розробників і не дозволяють розробникам комфортно підтримувати версію стилів веб-сайту. Великі файли, погана структура коду та стилі, що перекриваються, впливають на продуктивність розробників і збільшують час роботи. Ці проблеми можна вирішити шляхом рефакторингу файлів стилів за замовчуванням та створення кращої архітектури файлів. Для цього мені доведеться створити нову гілку з репозиторію Misc., створити нову папку проекту, рефакторингувати існуючі файли в ній та

реструктуризувати CSS. Головне завдання - зберегти вигляд веб-сайту за замовчуванням.

2.3 Методологія проведення рефакторингу та супроводу коду

Наша запропонована методологія базується на безшовному розширенні VSCode, яке інтегрується з існуючими робочими процесами розробників, забезпечуючи аналіз коду в режимі реального часу та пропозиції щодо можливостей ре факторингу [22]. Після вибору фрагмента коду розширення взаємодіє з бекенд-інфраструктурою, використовуючи семантичні знання Langchain та контекстуальне розуміння LLM для визначення та рекомендацій щодо покращень рефакторингу. Ці рекомендації надаються розробнику. Загальна архітектура:

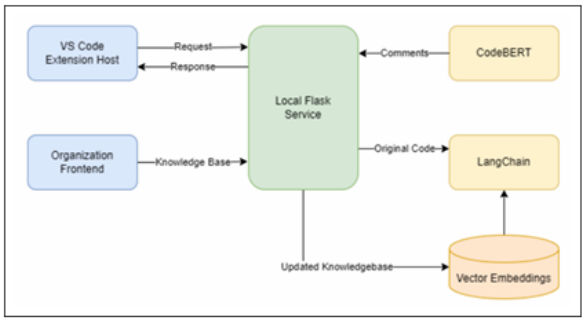


Рисунок 2.5 -Загальна архітектура CodeSage

Структуру можна побачити на рисунку 2.5.

Використовуючи децентралізований граф знань Langchain для мов програмування, наша система отримує глибоке розуміння семантики коду, що дозволяє їй генерувати контекстно-залежні пропозиції щодо рефакторингу, які покращують структуру, дизайн та зручність обслуговування коду. Крім того, спеціалізований LLM, навчений на великому наборі даних коду та прикладах рефакторингу, надає системі можливість навчатися на минулих рішеннях та адаптувати свої рекомендації до конкретних стандартів кодування, вимог проекту

та стилів програмування [23].

Окрім рефакторингу, наше рішення автоматизує створення комплексної документації, включаючи детальні пояснення змін, внесених під час рефакторингу, обґрунтування змін та потенційні наслідки для майбутніх модифікацій. Ця автоматизована документація гарантує, що зміни коду супроводжуються чіткими поясненнями, що покращує розуміння коду та зручність його підтримки для майбутніх розробників.

Поєднуючи автоматизований рефакторинг коду, генерацію документації та потужність Langchain та LLM, наша запропонована методологія має на меті революціонізувати практику підтримки коду в транснаціональних компаніях. Ми уявляємо майбутнє, де кодові бази послідовно рефакторуються, добре документуються та прості в обслуговуванні, що призводить до покращення якості програмного забезпечення, продуктивності розробників та зниження витрат на обслуговування.

Автоматизація рефакторингу коду та документації: запропоноване рішення

Підтримка узгодженості, довговічності та безпомилкової функціональності великих кодових баз є критичним завданням для багатонаціональних корпорацій (ТНК). Ручний рефакторинг коду та процеси документування не тільки займають багато часу, але й схильні до людських помилок, що призводить до невідповідностей, неефективності та підвищеного ризику дефектів програмного забезпечення.

Основні проблеми якості коду, з якими стикаються транснаціональні компанії, включають:

- Забезпечення високої якості коду завдяки послідовним стандартам та практикам кодування.
- Збереження читабельності та простоти коду для полегшення налагодження та майбутніх модифікацій.
- Повторювані фрагменти коду збільшують зусилля на обслуговування та ризик нестабільної поведінки.

• Код, який не є модульним або непридатним для повторного використання, призводить до надлишкового коду та труднощів з ізоляцією та виправленням проблем.

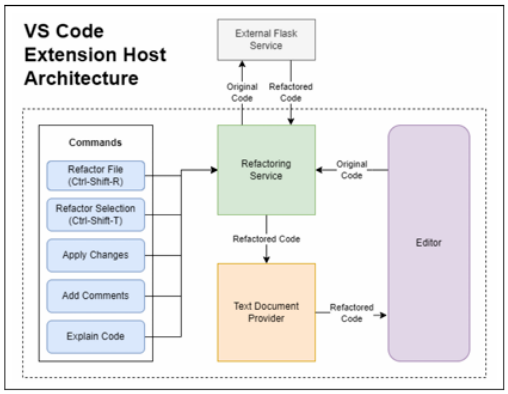


Рисунок 2.6 - Архітектура розширення VS Code

Щоб вирішити ці проблеми, ми пропонуємо інтегровану систему, яка автоматизує рефакторинг коду та створення документації, що працює на основі комбінації фронтенд-розширення VSCode та бекенд-інфраструктури, що використовує Langchain та спеціалізовану, налаштовану мовну модель (LLM) на основі знань.

Чиста, зручна в обслуговуванні кодова база завдяки дотриманню узгоджених стандартів кодування та зменшенню технічного боргу.

• Зворотній зв'язок у режимі реального часу для забезпечення дотримання стандартів кодування та раннього виявлення проблем.

• Аналізувати код, щоб пропонувати інтелектуальні пропозиції щодо рефакторингу та документації, зберігаючи модульність та можливість повторного використання.

Виявлення (Sensing) та розділення (Separating)

В ідеальному випадку вам не потрібно робити нічого особливого з класами системи, щоб почати ефективно з нею працювати. В ідеальній системі ви можете додати in vivo harness (тестовий «упряж» безпосередньо в робочому коді) і почати додавати тести для будь-якої нової роботи, яку виконуєте. Якщо у вас виникає

питання, як щось працює, ви створюєте об’єкти, пишете для них characterization tests (тести, що характеризують поведінку), а потім переходите до інших завдань.

Якби все було так просто, не було б потреби писати про це. На жаль, часто це буває складно.

Залежності між класами можуть сильно ускладнювати розміщення конкретних груп об’єктів під тест [24]. Ви хочете створити об’єкт одного класу і поставити йому питання, але для його створення вам потрібні об’єкти іншого класу, а тим, у свою чергу, потрібні об’єкти ще одного класу, і так далі, і так далі. Зрештою ви отримуєте майже всю систему в harness і один величезний setup-метод.

У деяких мовах програмування це не є великою проблемою. В інших, особливо в C++, час лінкування вашої системи сам по собі може зробити швидке повернення до роботи (rapid turnaround) майже неможливим, якщо ви не розірвете залежності, щоб відокремити частини системи.

У системах, які не розроблялися за допомогою Test Driven Development, вам часто доводиться розривати залежності, щоб помістити класи в тестовий harness. Але це не єдина причина для розриву залежностей.

Іноді клас, який ви хочете тестувати, має ефекти на інші класи, про які вам потрібно знати. Інколи ви можете виявити ці ефекти через інтерфейс іншого класу. В інших випадках це неможливо, тому єдиний варіант - імперсонувати (замінити) інший клас, щоб мати змогу безпосередньо виявляти (sense) ці ефекти.

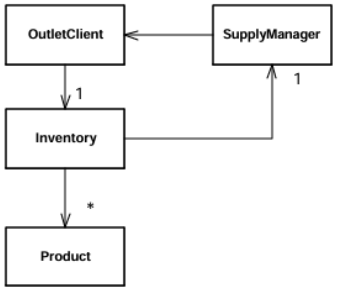


Рисунок 2.7 - Модель простої системи точки продажу (POS).

Давайте розглянемо приклад, який яскраво ілюструє ці проблеми. Ось модель простої системи точок продажу (Point of Sale). У цій системі клас

OutletClient є монолітним: він створює різноманітні об’єкти з бібліотек, наданих постачальниками, і було б особливо складно витягнути його в тестовий harness.

Одна з речей, яку робить OutletClient - оновлює інвентар (inventory) під час продажу. Інвентар повідомляє менеджера постачань (SupplyManager) щоразу, коли він змінюється [25]. Якщо SupplyManager виявляє, що потрібно відправити більше товарів у точку продажу, він надсилає сповіщення точці про очікувану поставку продуктів.

Ми хотіли б внести деякі зміни в SupplyManager, Inventory та деякі інші класи, з якими вони співпрацюють, але було б досить болісно затягувати OutletClient у тестовий harness. На жаль, OutletClient є справжнім центром (focal point) у цьому дизайні. Як тоді написати characterization tests для SupplyManager?

Найскладніше те, що OutletClient - це єдиний клас, який знає про деякі дії, які виконує SupplyManager. Якщо ми зможемо створити інвентар незалежно від OutletClient, ми зможемо досліджувати його інтерфейс і тестувати SupplyManager окремо. З іншого боку, нас може цікавити лише характеристика поведінки, яку можна запустити через публічні методи OutletClient. У цьому додатку OutletClient є єдиним клієнтом Inventory. Оскільки це так, ми можемо бути впевнені, що Inventory не використовується ніяким іншим способом, окрім як побічний ефект викликів методів OutletClient. Це значно звужує обсяг поведінки, яку потрібно характеризувати.

У нас є одночасно проблеми і виявлення (sensing), і розділення (separating). Ми хочемо відокремити ті частини OutletClient, які залежать від речей, що було б проблематично помістити в harness, але ми також хочемо знати те, про що знає лише OutletClient.

Розрив зовнішніх залежностей (Breaking External Dependencies)

Зовнішні залежності розривати досить легко. Розгляньмо приклад.

У цій моделі в нас є два класи: Transaction та TransactionLog. Щоразу, коли створюється транзакція, ми передаємо їй журнал транзакцій (transaction log), до якого вона має писати. Той факт, що саме ми створюємо цей журнал і передаємо його транзакції, дає нам великий контроль.

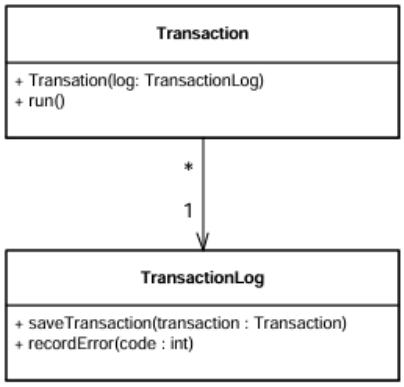


Рисунок 2.8 - Розрив зовнішніх залежностей

Найпростіше, що можна зробити - застосувати один із найбезпечніших рефакторингів із книги Мартіна Фаулера: Extract Interface (Витягти інтерфейс). Після того, як ми це зробимо, ми зможемо створювати об’єкти, які виглядають як журнали транзакцій, і передавати їх транзакціям.

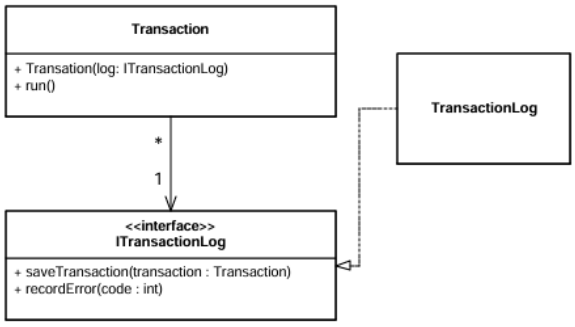


Рисунок 2.9 – приклад Extract Interface (Витягти інтерфейс)

У Java або C++ витягнення інтерфейсу виконується дуже просто. Ви створюєте порожній інтерфейс, змушуєте цільовий клас його реалізовувати, а потім змінюєте клас, який використовує цей об’єкт, щоб він приймав об’єкти типу інтерфейсу, а не конкретного цільового класу.

Після цього спробуйте скомпілювати систему. Щоразу, коли компілятор повідомить, що на інтерфейсі відсутній метод, додайте цей метод до інтерфейсу [26]. Якщо об’єкт утримується іншими посиланнями в класі, який його використовує, вам, можливо, також доведеться змінити їх на тип інтерфейсу.

Extract Interface - це потужний спосіб досягти як виявлення ефектів (sensing), так і розділення (separation). Ви можете використовувати його однаково незалежно від того, чи приходить залежність через аргумент конструктора, чи через звичайний аргумент методу.

2.4 Висновок по розділу

У другому розділі було досліджено методи та інструменти рефакторингу і підтримки програмного коду. Розглянуто особливості архітектури програмних модулів та сучасні інструментальні засоби, що підтримують процеси рефакторингу. Проаналізовано методи виявлення проблем у програмному коді, включаючи аналіз складності, дублювання та потенційних дефектів. Також досліджено методологію проведення рефакторингу та супроводу коду, що дозволяє забезпечити стабільність і покращення якості програмних систем. У результаті визначено ефективні підходи до організації процесів підтримки та модернізації програмного забезпечення.

РОЗДІЛ 3. РЕАЛІЗАЦІЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ РІШЕНЬ

3.1 Розробка та інтеграція інструменту підтримки рефакторингу

В основі нашого рішення лежить фронтенд-розширення Visual Studio Code, як показано на рисунку 3.1. Це розширення бездоганно інтегрується з існуючими робочими процесами розробників, забезпечуючи аналіз коду в режимі реального часу та пропозиції щодо можливостей рефакторингу. Після вибору фрагмента коду для рефакторингу розширення ініціює зв'язок з бекенд-інфраструктурою для обробки коду та створення рекомендацій щодо рефакторингу. Ці рекомендації потім надаються розробнику, що дозволяє йому переглянути та застосувати зміни одним клацанням миші.

Підвищення ефективності та зручності обслуговування за допомогою автоматизованого рефакторингу коду

Автоматизований рефакторинг коду пропонує кілька переконливих переваг для транснаціональних компаній, зокрема: Покращена якість коду: Інструменти автоматизованого рефакторингу можуть ефективно виявляти та виправляти неприємні запахи коду, надлишковий код та неефективні структури, що призводить до чистішого та зручнішого в обслуговуванні коду.

Скорочення часу розробки: Автоматизуючи завдання рефакторингу, розробники можуть перенаправити свої зусилля на проектування та реалізацію вищого рівня, значно скорочуючи час, витрачений на ручне обслуговування коду.

Покращена читабельність коду: Добре рефакторований код легше зрозуміти та змінити, що сприяє співпраці між розробниками та зменшує ймовірність помилок під час внесення змін.

Дотримання стандартів кодування: Запропоноване рішення може бути адаптоване для забезпечення дотримання певних стандартів кодування в межах багатонаціональної мережі (ВС), забезпечуючи узгодженість у всій кодовій базі.

Вичерпні коментарі: Розширення Visual Studio Code спрощує завдання створення вичерпних коментарів, аналізуючи код у режимі реального часу та

надаючи інтелектуальні, контекстно-залежні пропозиції за допомогою інтегрованої мовної моделі (LLM).

Використання Langchain та LLM для інтелектуального рефакторингу коду

Наше запропоноване рішення використовує Langchain, децентралізований граф знань для мов програмування, щоб забезпечити насичений контекст для аналізу та рефакторингу коду. Langchain зберігає зв'язки між програмними конструкціями, дозволяючи системі розуміти семантичне значення коду та генерувати контекстно-залежні пропозиції щодо ре факторингу [27]. Це семантичне розуміння є критично важливим для виявлення можливостей рефакторингу, які не тільки покращують структуру коду, але й підвищують його загальний дизайн та зручність підтримки.

Для подальшого покращення інтелекту системи ми використовуємо спеціалізовану LLM-мастеринг на основі знань, що налаштований на величезному наборі даних коду та пов'язаних з ним прикладах рефакторингу. Цей LLM надає системі можливість навчатися на минулих рішеннях щодо рефакторингу та адаптувати свої рекомендації до конкретних стилів програмування, стандартів кодування та вимог конкретного проекту.

Автоматизація створення комплексної документації

Окрім рефакторингу коду, наше рішення також автоматизує створення вичерпної документації для рефакторованого коду. Ця документація містить детальні пояснення змін рефакторингу, обґрунтування цих змін та будь-які потенційні наслідки для майбутніх модифікацій коду. Автоматизуючи цей процес, ми усуваємо необхідність ручного документування, гарантуючи, що зміни коду супроводжуються чіткими та лаконічними поясненнями.

Ми успішно розробили розширення Visual Studio Code для автоматизованого рефакторингу коду та інтегрували його з сервером Flask. Розширення пропонує два режими роботи: рефакторинг вибраного фрагмента коду або рефакторинг усього файлу. Вибравши потрібний режим рефакторингу, користувач може ввести код. Рефакторований код потім відображається на

спеціальній панелі, де зміни виділені зеленим кольором, а пропущені частини – червоним. Таке візуальне представлення дозволяє користувачам легко ідентифікувати зміни, внесені до коду. Щоб забезпечити контроль користувача над процесом рефакторингу, розширення пропонує користувачеві прийняти або відхилити запропоновані зміни [28]. Цей крок підтвердження гарантує, що рефакторований код відповідає намірам та уподобанням користувача.

Для нашого розуміння ми провели тестування, надавши розширення VS Code 50 учасникам, розділеним на п'ять груп по 10 учасників у кожній. У першій групі 9 з 10 учасників (90%) повідомили, що програмне забезпечення точно передбачило бажаний результат. Вихідність наступних груп показано на рисунку 3. Таким чином, загальна точність нашого рішення для всіх п'яти груп становить 74%.

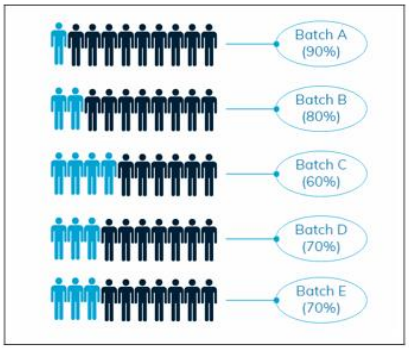


Рисунок 3.1 - Результативність пакетного тестування

Рефакторинг вибраного фрагмента коду

Як видно на рисунку 3.2, користувач може вибрати рефакторинг вибраного фрагмента коду або всього файлу. Під час рефакторингу вибраного фрагмента коду розширення ізолює вибрану частину коду та надсилає її на сервер Flask для обробки. Сервер використовує граф знань Langchain та точно налаштований LLM для аналізу семантики коду, виявлення можливостей рефакторингу та генерації відповідних рекомендацій [29]. Рефакторований код потім надсилається назад до розширення VSCode та представляється користувачеві на спеціальній панелі. Користувач може переглянути зміни, виділені зеленим кольором, та вирішити, чи

прийняти, чи відхилити пропозиції щодо рефакторингу.

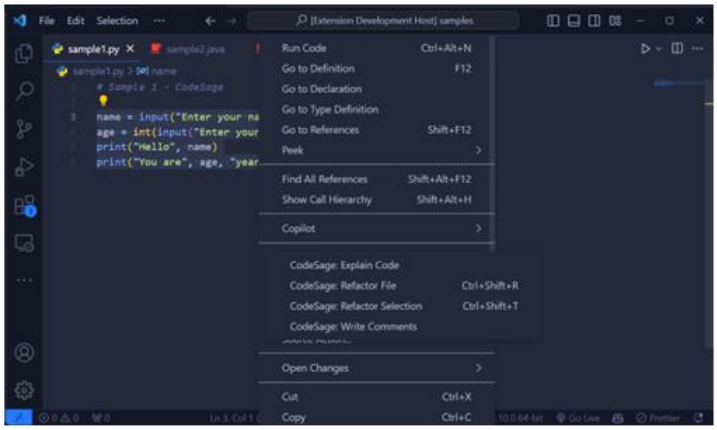


Рисунок 3.2 - Користувач може вибрати для рефакторингу або фрагмент коду, або весь файл

Рефакторинг усього файлу

Рефакторинг усього файлу передбачає надсилання всього файлу коду на сервер Flask для аналізу та рефакторингу. Сервер застосовує той самий процес семантичного аналізу та генерації пропозицій щодо рефакторингу, що й у вибраному режимі фрагмента коду. Рефакторований код потім надсилається назад до розширення VSCode, де він відображається разом з оригінальним кодом. Користувач може порівняти оригінальний та рефакторований код, позначивши зміни зеленим кольором, а пропуски червоним, і вирішити, чи прийняти, чи відхилити загальні зміни рефакторингу.

Контроль користувача над рефакторингом

Як видно на рисунку 5, в обох режимах рефакторингу розширення VS Code пропонує користувачеві прийняти або відхилити запропоновані зміни перед їх застосуванням до коду. Цей користувацький контроль гарантує, що рефакторований код відповідає намірам та уподобанням розробника. Забезпечуючи цей крок підтвердження, ми надаємо розробникам можливість контролювати кодову базу та гарантуємо, що зусилля з рефакторингу покращать загальну якість та зручність обслуговування коду.

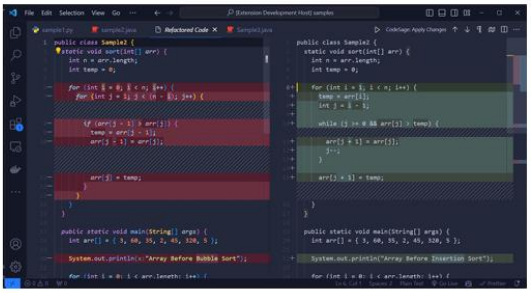


Рисунок 3.3 - Користувач може переглянути рефакторований код і прийняти його

Успішна розробка розширення VSCode для автоматизованого рефакторингу коду демонструє доцільність та ефективність запропонованої нами методології. Здатність розширення рефакторувати як вибрані фрагменти коду, так і цілі файли, у поєднанні з візуальним представленням змін та контролем користувача над рефакторингом, робить його цінним інструментом для розробників у транснаціональних компаніях [30].

3.2 Практичне застосування методів рефакторингу до існуючого коду

Основна причина, чому HTML та CSS не вважаються мовами програмування, полягає в тому, що вони визначають лише структуру та стиль веб-сторінки, яку ви створюєте. Вони не містять жодних інструкцій, як інші мови фронтенду. Однак, CSS-код та HTML-файли все одно повинні бути простими у використанні для розробників. Перед рефакторингом коду важливо визначити «хороший» код.

«Хороший код пишеться так, щоб він був читабельним, зрозумілим, пройшов автоматизовані тести, не був надто складним і добре виконував те, що задумано» . Щоб зробити CSS більш зрозумілим для розробників, існує кілька методологій, які можуть спростити роботу з таблицями стилів.

CSS-методики

Щоб уникнути необхідності вигадувати власні правила написання CSS, розробник може застосувати один із підходів, які спільнота дизайнерів

випробувала в багатьох проектах. Ці підходи називаються методологіями CSS. Ці методології, по суті, є посібниками з кодування, які допомагають розробникам писати організований та структурований код. Зазвичай вони відображають CSS детальніше, ніж розробник, який би написав та оптимізував кожен селектор для власного набору правил для цього проекту. Оскільки багато з цих систем широко використовуються, інші розробники з більшою ймовірністю зрозуміють підхід, який ви використовуєте.

Методологія БЕМ

BEM розшифровується як Модифікатор Елемента Блоку (Block Element Modifier). У цій методології блок є окремою сутністю, наприклад, продуктом, кнопкою або посиланням. Модифікатор – це прапорець на блоці. Він змінює стиль або поведінку. BEM – це методологія CSS для створення CSS-коду повторного використання. Вона надає розробникам шаблони іменування компонентів, які роблять зв'язок між стилізованими класами компонентів та розміткою більш очевидним. У нашому випадку методологію BEM не можна застосувати, оскільки вона вимагала зміни HTML-коду для зміни назв класів.

Методологія OOCSS

Більшість методологій, з якими стикаються розробники, мають певний зв'язок з концепцією об'єктно-орієнтованого CSS (OOCSS). Основна ідея об'єктно-орієнтованого CSS полягає в тому, щоб розділити ваш CSS на кілька елементів багаторазового використання, які можна використовувати будь-де на вашому сайті. Наприклад, якщо у нас є два елементи, які мають більшість спільних стилів, краще уникати написання тих самих правил стилю кілька разів. Найкращим рішенням було б об'єднати стилі елементів в одному правилі, а потім окремо розширити стилі елементів, якщо потрібно [31]. Таким чином, ми можемо уникнути повторень і зберегти таблиці стилів зрозумілішими. Ця методологія частково використовувалася в процесі рефакторингу, оскільки вона не вимагає жодних змін HTML.

Налаштування проекту

Перед рефакторингом коду я створив нову папку для цього проєкту. Я вирішив назвати папку проєкту «test-styles». Після цього я створив окрему гілку з репозиторію Misc. Я назвав цю гілку «thesis-styles» (Рисунок 3.4). Створивши окрему гілку, я переконався, що моя робота не вплине на оригінальні файли.

Рунок 3.4 - Нова гілка «thesis_styles».

Рефакторинг масивних файлів

Коли розробник починає працювати над великими CSS-файлами та великими проектами, він/вона розуміє, що підтримка та робота з великим CSS-файлом може бути складною. У деяких випадках, коли у вас є багато різних правил стилю для елементів на веб-сайті, вам може знадобитися розділити ваші таблиці стилів на менші файли. Це полегшує роботу та упорядкування вашого CSS-файлу.

«_main-contents.scss» – файл

Першим кроком у рефакторингу стилів за замовчуванням було розділення CSS у великих файлах на вже існуючі компоненти. Одним із файлів з найбільшою кількістю CSS був «_main-contents.scss». Він містив 1330 рядків правил стилізації для кількох компонентів. Були правила стилізації для таких класів, як «page-2», «product-list», «ajaxContainer», «share-buttons». Більшість стилів не підходили до жодного з уже існуючих компонентів, тому мені довелося створювати нові файли компонентів [32]. Тільки з файлу рефакторингу «_main-contents.scss» мені довелося створити такі файли компонентів, як:

1. “_product-list-item.scss”
2. “_product-group-list.scss”
3. “_product-list.scss”

4. “_product-related.scss”

5. “_product-variation.scss”

6. “_product-page.scss”

7. “_group-page.scss”

8. “_product-related.scss”

9. “_buttons.scss”

Причина, чому всі назви файлів починаються з символу підкреслення, полягає в тому, що важливо дотримуватися однакової системи іменування протягом усього проєкту. Вже існуючі компоненти мали таку систему іменування, тому я вирішив зберегти її та назвати нові компоненти так само.

Після розділення всіх правил стилю на різні компоненти файл **«_maincontents.scss»** виявився порожнім [33]. У цьому файлі більше не було потреби, тому я його видалив. Під час проекту я видалив більшість вихідних файлів після розділення CSS всередині них на різні компоненти.

«_cart.scss» – файл

Другим за розміром файлом був **«_cart.scss»** . Цей файл містив усі правила стилізації кошиків для покупок на веб-сайті. На веб-сайті є кілька місць, де можна використовувати кошики для покупок. Кошик для покупок – це елемент, який спрощує купівлю товару чи послуги, показуючи клієнту товари, які він збирається купити. Він може відображатися як випадаючий список у верхній частині веб-сайту або як таблиця на окремій сторінці. Існує три різні місця, які електронна комерція використовує для відображення кошика на веб-сайті: випадаючий список у заголовку, таблиця на сторінці кошика для покупок та як таблиця на сторінці оформлення замовлення. Я вирішив створити нові компоненти на основі різних місць, які використовував кошик для покупок. Після рефакторингу файлу **«_cart.scss»** я отримав три компоненти: «_cart - checkout.scss» - містить стилі кошика для покупок, що використовуються лише на сторінці оформлення замовлення.

«_cart - dropdown.scss» – включає стилі доданих товарів у невеликому випадаючому списку.

«_cart - table.scss» - включає стилі доданих товарів у вигляді таблиці.

Загальні компоненти

Майже в кожному оригінальному файлі компонента можна знайти загальне або глобальне правило CSS. Це може бути кнопка, значок, шрифт або правило перевизначення bootstrap. Загальне або глобальне правило стилю, яке впливає на всі елементи веб-сайту, слід зберігати в окремому файлі. У випадку, коли існує багато загальних стилів, гарною практикою є розділити їх на менші компоненти, щоб їх було легше підтримувати [34]. Я вирішив розподілити всі загальні стилі з компоненти відповідно до їхнього призначення. Зрештою, я отримав шістнадцять поширених компонентів

«_alert.scss» – стилі для спливаючих повідомлень на веб-сторінці.

«_amount-box.scss» – стилі для введення сум.

«_body.scss“ – містить загальні правила стилізації для «тіла» кожного використаного HTML-файлу. «_bootstrap-overrides.scss“ – стилі, що замінюють правила стилізації вбудованих бібліотек. «_brochures.scss“ – містить стилі брошур продуктів.

«_buttons.scss» – містить стилі основних кнопок вебсайту.

«_console.scss» – включає стилі елемента та вмісту вкладки в ньому. «_events.scss» – включає стилі елементів подій на головній сторінці.

«_filters.scss» – містить стилі фільтрів списків товарів.

«_font.scss» – містить стилі основних шрифтів та їх стани (при наведенні курсора та активний). «_icons.scss» – містить стилі основних піктограм, що використовуються на вебсайті.

«_label.scss» – містить стилі основних етикеток товарів.

«_main-content.scss» – містить правила стилю індексної сторінки (шаблон, який використовується на кожній сторінці, окрім головної).

«_maintenance.scss» – містить стилі, які не відповідають жодному зі спільних компонентів (лише клас maintenance-wrapper).

«_pagination.scss» – містить стилі стрілок та номерів сторінок, які використовуються для навігації між сторінками.

«_video.scss» – містить стилі відеоконтейнера.

Перевизначення Bootstrap

У шаблонах HTML веб-сайтів електронної комерції за замовчуванням використовується багато класів bootstrap. Кожен із класів Bootstrap має свої власні властивості. У деяких випадках ці властивості слід розширювати, але іноді потрібно зберігати лише деякі з них. Існує кілька випадків у стилях за замовчуванням, коли класи bootstrap потрібно перевизначити або змінити. Наприклад, клас **«form-control»** . Якщо ми перевіримо цей клас у браузері, то побачимо, що за замовчуванням він має кілька стилів, включаючи радіус межі. Ця властивість була перевизначена у стилях електронної комерції за замовчуванням (рисунок 3.5).

```
.form-control {
    border-radius: 0px;
    @media screen and (max-width: $breakpoint-lgs) and (min-width: $breakpoint-sm) {
        font-size: 0.8rem;
    }
}
```

Рисунок 3.5 - Переопределенные свойства класса Bootstrap.

Цей та всі інші перевизначення bootstrap, які глобально впливають на стилі за замовчуванням, я помістив в окремий файл під назвою « **_boostrap-overrides.scss** ». Цей файл вважається загальним компонентом, оскільки вони впливають на кожен елемент веб-сайту за замовчуванням.

Коментарі в CSS

Коли розробник додає рядки коментарів до CSS-коду, це допомагає іншим розробникам працювати зі стилями. Коментарі також допомагають розробникам, коли вони повертаються до проєкту після тривалої перерви. Гарною практикою є додавання коментарів між логічними розділами вашого CSS-коду. Коментарі дуже корисні для навігації у CSS-файлі. Це допомагає швидко знаходити різні розділи під час їх перегляду.

Після розділення всіх оригінальних файлів на менші компоненти, я отримав 43 нових файли компонентів. Для кращої навігації я залишив короткий коментар до кожного файлу, який пояснює призначення стилів у ньому (рисунок 3.6).

```
/* Event dates on event page */
#event_dates_list_event_page {
    .label-default {
        margin-right: 5px !important;
        font-size: 14px !important;
        display: inline-block !important;
    }
}
#event_image_event_page {
    width: 95%;
}
```

Рисунок 3.6 - Короткий коментар у файлі CSS для зручнішої навігації.

Медіа-запити

Під час рефакторингу коду однією з найбільших проблем було розподілити всі медіа-запити під правильним компонентом. У вихідних стилях був великий CSS-файл під назвою « **_media-queries.scss** », який містив більшість медіа-запитів стилів за замовчуванням. Це не найкращий спосіб зберігати всі медіа-запити в окремому файлі, оскільки розробнику може бути складно зрозуміти, з яким компонентом вони пов'язані, оскільки компоненти можуть мати однакові класи та ідентифікатори [35].

Кращий спосіб - розміщувати медіа-запити в компоненті, з яким вони пов’язані (рисунок 3.7). Це заощадить розробникам час на пошук правильного рядка CSS для зміни властивості.

```
/* Main menu */
.content-onscreen {
    @media only screen and (max-width:$breakpoint-md) {
            transform: translate(0);
            transition: transform .4s ease;
            &.offscreen {
                transform: translate(-80%)
            }
    }
}
```

Рисунок 3.7 - Елементи медіа-запитів розміщуються у класі, до якого вони належать.

Оригінальний файл « **_media-queries.scss** » містив 387 рядків CSS. Після розподілу медіа-запитів за правильними класами файл став порожнім, і я його видалив.

Вкладений CSS

Наступним кроком після додавання медіа-запитів під правильними компонентами було зменшення кількості рядків CSS шляхом написання вкладеного CSS. Вкладений CSS у цьому випадку не тільки зменшує обсяг коду, але й робить його зрозумілішим, оскільки немає потреби повторювати імена класів для виклику їхніх дочірніх елементів. На рисунку 9 є повторювані класи, тоді як на рисунку 10 є вкладений CSS.

Перегляд усіх файлів компонентів та вкладення їхньої CSS-структури був найбільш трудомістким процесом під час рефакторингу.

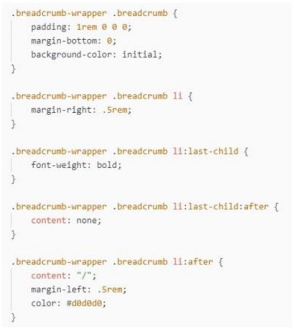


Рисунок 3.8 - Таблиця стилів містить багато повторюваних класів, наприклад — «breadcrumwrapper». Це може ускладнювати навігацію по файлах.

Рисунок 3.9 - Усі елементи та їхні властивості об’єднані в один клас. Вкладеність робить код CSS більш зрозумілим.

Об'єднання компонентів у папках

Наступним кроком було створення добре структурованої архітектури, яка дозволить розробникам легко переміщатися між компонентами. Щоб досягти гарної структури файлів, я створив кілька папок і розподілив усі компоненти між ними. Усі файли компонентів були розміщені в папках відповідно до їх призначення (рисунок 3.10). Наприклад, компоненти, пов'язані з нижнім колонтитулом, були поміщені в папку під назвою «footer», загальні компоненти - в папку «commons», компоненти, пов'язані з продуктом, - в папку «product». Зрештою, я отримав 10 папок із пов'язаними компонентами. Не існує єдиного правильного способу розподілу компонентів між папками. Спосіб розподілу файлів має базуватися на специфікації проекту.

Рисунок 3.10 - Файли компонентів, пов’язані з кошиком, зберігаються у спільній папці під назвою «cart».

Об'єднання файлів

Щоб уникнути великої кількості імпортованих файлів у файлі " **style.scss** ", я вирішив об'єднати всі компоненти з папок в один файл. Для цього я створив новий файл у кожній папці компонентів та імпортував туди всі компоненти (Рисунок 3.11). Така структура дозволила значно зменшити кількість імпортованих файлів у файлі " **style.scss** " та зберегти гарну ієрархію файлів.

```
@import '_cart-checkout.scss'; //Imports checkouts summary-container styles;
@import '_cart-dropdown.scss'; //Imports cart dropdowns styles (header cart);
@import '_cart-table.scss'; //Imports all product-tables styles both on cart- and checkout-page;
```

Рисунок 3.11 - Усі файли компонентів, імпортовані в один об’єднаний файл.

Потенційно це може заощадити час розробника, якщо він/вона захоче додати новий компонент до папки. Вам потрібно лише імпортувати новий компонент до файлу об'єднання внизу файлу компонента. Це автоматично

підключить новий файл до спільного файлу " **style.scss** ", не змінюючи його структуру.

Я вирішив зберегти файли «змінні» та « кольори » в папці modules. Модулі в цьому проєкті – це значення, до яких повинні мати доступ усі компоненти [36]. Наприклад, змінні використовуються в медіа-запитах для позначення розміру екрана в пікселях. Оскільки на цьому етапі медіа-запити є майже в кожному файлі компонента. Кожен компонент повинен мати доступ до файлу « змінних » . Колір також є змінною, яка використовується серед більшості файлів компонентів і повинна бути доступною для всіх компонентів. Ще одними значеннями, які можуть використовувати компоненти, є «міксини» . У вихідній версії стилів за замовчуванням більшість «міксинів» знаходилися у файлі «_components.scss» серед деяких випадкових стилів. Я перейменував цей файл і переніс усі стилі, які не знаходяться всередині «міксинів»**,** до файлів компонентів, до яких вони належать. Зрештою, я отримав три файли в папці «модулі» , які потрібно підключити до компонентів. Щоб надати компонентам доступ до модулів, я створив новий файл «set-up.scss» (рисунок 3.12)

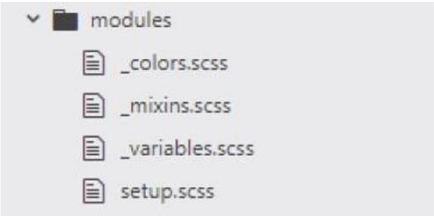


Рисунок 3.12 Нова структура модулів і папок.

У цьому файлі я включив усі модулі за допомогою правила CSS **“@import”** (рисунок 3.13). Після цього я включив файл **“set-up.scss”** до кожного файлу об’єднання в папках компонентів (рисунок 3.14). Останнім етапом було імпортувати компоненти до файлу **“style.scss”,** щоб об’єднати всі компоненти та модулі разом.

```
/* Set up modules */
@import '_variables.scss';
/* Set up colors */
@import "_colors.scss";
/* Set up mixins */
@import "_mixins.scss";
```

Рисунок 3.13 Модулі, імпортовані в один файл «set-up.scss».

Після завершення рефакторингу коду наступним кроком було його тестування та перевірка, чи вплинуло це на оригінальний вигляд вебсайту. Для цього мені довелося встановити нові стилі за замовчуванням для мого проєкту з гілки «thesis-styles» . Щоб автоматично встановлювати нові стилі до папки проєкту, мені довелося написати команду Gulp,

```
@import "./modules/setup.scss"; //Imports styles from cart folder;

/* Import components */
@import "components/cart/cart.scss"; //Imports styles from cart folder;
@import "components/common/common.scss"; //Imports styles from common folder;
@import "components/footer/footer.scss"; //Imports styles from footer folder;
@import 'components/header/header.scss'; //Imports styles from header folder;
@import 'components/images/images.scss'; //Imports styles from images folder;
@import 'components/navigation/navigation.scss'; //Imports navigation from cart folder;
@import "components/pages/pages.scss"; //Imports styles from pages folder;
@import "components/product/product.scss"; //Imports styles from product folder;
@import "components/search/search.scss"; //Imports styles from search folder;
@import "components/widgets/widgets.scss"; //Imports styles from widgets folder;
```

Рисунок 3.14 - Файл налаштувань, імпортований у папку «commons» - файл об’єднання.

Після того, як стилі за замовчуванням були завантажені в нову папку проекту, мені довелося виконати другу команду gulp “watch-customer-styles - customer thesis-styles” [37]. Ця команда мала відстежувати всі зміни в користувацькому SCSS-файлі та мінімізувати або стиснути всі таблиці стилів в один CSS-файл.

Після збереження користувацького файлу, всередині папки «thesis-styles» автоматично була створена папка для збірки. Папка «build» містила мініфікований CSS-файл, готовий до розміщення на сервері. Веб-сайт залишився таким самим після рефакторингу коду, жодних змін не було помічено.

3.3 Оцінка ефективності рефакторингу та аналіз загроз валідності

На завершення, це дослідження представило новий підхід до автоматизованого рефакторингу коду та створення документації для багатонаціональних корпорацій (МНК). Згідно з дослідженням [15], проведеним AlOmar та ін., перевірка коду відіграє вирішальну роль у підтримці якості програмного забезпечення та обміні знаннями між командами розробників. Запропоноване рішення, зосереджене на фронтенд-розширенні VSCode та бекенд-інфраструктурі на базі Langchain та користувацькій мовній моделі (LLM), вирішує проблеми ручного обслуговування коду та підвищує якість коду, продуктивність розробників та загальну зручність обслуговування кодової бази.

Безшовна інтеграція розширення VSCode в існуючі робочі процеси розробників забезпечує аналіз коду в режимі реального часу та пропозиції щодо рефакторингу, що дозволяє розробникам ефективно виявляти та вирішувати проблеми з кодом. Інфраструктура серверної частини, використовуючи семантичні знання Langchain та контекстне розуміння LLM, генерує інтелектуальні рекомендації щодо рефакторингу, які відповідають конкретним стандартам кодування та вимогам проекту.

Запропоноване рішення не лише автоматизує рефакторинг коду, але й генерує вичерпну документацію для рефакторованого коду, гарантуючи, що зміни коду супроводжуються чіткими поясненнями та обґрунтуванням. Така автоматизована документація покращує розуміння коду та зручність його підтримки для майбутніх розробників, сприяючи співпраці та зменшуючи ризик помилок під час внесення змін [38].

Завдяки бездоганній інтеграції автоматизованого рефакторингу коду, генерації документації та використанню вражаючих можливостей Langchain та LLM, це дослідження не лише демонструє практичність, але й підкреслює ефективність інноваційної парадигми, готової переосмислити практику підтримки коду в багатонаціональних корпораціях (МНК).

У цьому розділі ми пропонуємо загальний аналіз спільних результатів та

спостережень нашого дослідження цих трьох застосувань. Ми також перераховуємо загрози валідності, які можуть обмежити застосовність або вплинути на результати.

А. Аналіз результатів

Для всіх трьох програм використання аспектів вимагає часових витрат на розробку аспектів та рефакторинг оригінальної програми. Однак більше половини змін рефакторингу були видаленнями, і загальним ефектом було зменшення розміру вихідного коду [40]. Спостерігалося збільшення часу виконання, вимог до пам'яті та розміру скомпільованого об'єкта. Найбільше збільшення спостерігалося для розміру об'єкта, що вказує на потенційне збільшення обсягу пам'яті та часу виконання. Однак для цих програм сам розмір об'єктного коду не був критичним, оскільки значне збільшення розміру об'єктного коду призвело до невеликого збільшення вимог до пам'яті та часу виконання. Збільшення обсягу пам'яті та часу виконання було спричинено перевірками безпеки в рефакторованій версії, які відсутні в оригінальній програмі. Якби оригінальна програма реалізувала ті самі перевірки, збільшення часу виконання та використання пам'яті було б меншим. Крім того, аспекти, які виконують ці перевірки, можна легко змінити шляхом зміни аспекту, тоді як модифікація еквівалентного коду перевірки в оригінальній програмі вимагатиме багатьох змін. Рефакторинг, якщо його не виконувати ретельно, може призвести до помилок.

Локальність змін була покращена за допомогою аспектів. Загалом, рефакторинг дозволяє уникнути багатьох змін в основних аспектах. Однак у нашому дослідженні ми виявили, що рефакторинг спричинив два види змін в аспектах або допоміжному коді. По-перше, для таких аспектів, як Excepter , чиї точки зрізу складалися зі списку функцій, нам потрібно було оновити точку зрізу в наступних версіях додатковими іменами функцій, щоб повідомити про нові функції. Аспекти, такі як CheckFwArgs , чиї точки зрізу базувалися на тому, що код програми був частиною класу або простору імен, вимагають, щоб основні аспекти дотримувалися цієї угоди про іменування [39]. Таким чином, нові функції, додані до основного аспекту, які потребували перевірки аспектом CheckFwArgs ,

мали бути укладені в клас або простір імен, щоб точка зрізу відповідала їм.

Розповсюдження проблем було зменшено, оскільки аспекти забезпечували кращу модульність, ніж оригінальна об'єктно-орієнтована реалізація. Використання аспектів усувало дефекти, які виникали, коли проблема була непослідовно реалізована в кількох місцях.

Переваги від використання аспектів у трьох застосунках різнилися. Наприклад, ErcChecker мав найбільше зменшення розміру вихідного коду після рефакторингу першої редакції. Натомість, instanceDrivers мав незначну початкову економію коду, але пізніші редакції заощадили більше коду, оскільки в аспектно-орієнтованій реалізації було уникнуто кількох перехресних змін. Як ErcChecker, так і PowerAnalyzer отримали більше половини економії коду під час рефакторингу першої редакції. Потенційна вигода від рефакторингу залежить від обсягу перехресного коду в оригінальному застосунку. Остання редакція PowerAnalyzer містила 16 286 рядків коду, при цьому рефакторинг забезпечив економію 397 рядків (2%). Остання редакція ErcChecker містила 62 608 рядків коду, при цьому рефакторинг забезпечив економію 2546 рядків (4,1%). Остання редакція InstanceDrivers мала найвищий відсоток економії перехресного коду – 348 рядків з 3395 (10%).

Загалом, аспекти, які дозволяють видаляти найбільше рядків коду, роблять це, повідомляючи про багато точок з'єднання. Для командної розробки однією з потенційних проблем може бути повідомлення про ці зв'язки між порадою та точками з'єднання. Як повідомляють. [30], невизначеність, яку забезпечують точки на основі імен, може ускладнити підтримку та паралельну розробку аспектів та основних проблем. Для нашого дослідження, оскільки один розробник виконував рефакторинг та підтримку між ревізіями, це не було проблемою, але це потенційна проблема під час рефакторингу застарілих програм, які підтримуються великою командою.

У нашому дослідженні самі аспекти мало або взагалі не змінилися. Наприклад, клас TimeEvent , який використовується аспектом Timer класу InstanceDrivers , не змінився. Якби змінився тип даних про синхронізацію або

основні аспекти, які збирали ці дані, сам аспект Timer потребував би змін. Однак, у нашому дослідженні таких змін не було.

B. Загрози валідності

Як і в більшості тематичних досліджень, важко узагальнювати результати дослідження трьох застосунків. Таким чином, існують загрози зовнішній валідності. У цьому дослідженні процес рефакторингу застосовувався лише до трьох застарілих застосунків. Ці застосунки не були обрані випадковим чином, що обмежує зовнішню валідність. Крім того, застосунки були з однієї проблемної області. Вибір пов'язаних застосунків, які використовують спільний фреймворк, може призвести до упередження застосунків до певних стилів і характеристик дизайну та кодування. Більше того, ймовірні різні результати при застосуванні процесу до застосунків в інших областях.

Через обмежений характер оцінювання існують загрози внутрішній валідності – чи дійсно аспектна орієнтація відповідає за покращення ремонтопридатності.

Одним із занепокоєнь є те, що один і той самий суб'єкт розробляв аспекти та рефакторингував три програми. Більше того, в рамках своєї роботи цей суб'єкт розробляв одну програму та виконував технічне обслуговування іншої. На аналіз змін між ревізіями міг вплинути попередній досвід роботи з програмами. Визначаючи аспекти з першої ревізії застарілої програми, розробник може бути упередженим до частин програми, які, як відомо, більш схильні до змін, і може бути упередженим до тих частин програми, які не були схильні до змін. Інший суб'єкт може вибрати різні аспекти та реалізувати їх по-різному. Підхід у цьому дослідженні полягав у використанні всіх визначених аспектів. Більше того, ми внесли лише ті зміни, які були необхідні для використання аспектів. Інший підхід полягав би у наданні переваги більш масштабному рефакторингу (наприклад, додаванню більшої об'єктної орієнтації до основного коду) перед використанням аспектів.

Розробник, який створив аспекти, повідомив про дефекти, яких уникнули завдяки аспектам, та дефекти, спричинені аспектами. Інший розробник міг знайти

інші дефекти в оригінальній програмі та вставити інші дефекти під час рефакторингу. Крім того, будь-який розробник аспектів міг не знати про дефекти у своїй рефакторованій програмі, які міг би знайти інший розробник, або якби покриття тестами було вищим.

Використання застарілих програм для проведення дослідження може створювати проблеми. Застаріле програмне забезпечення часто відображає рішення, прийняті на основі мови програмування та інструментів, доступних на момент розробки програмного забезпечення. Застарілий код може бути застарілим через певний стиль дизайну, такий як використання або відсутність шаблонів проектування, що може вплинути на те, які типи рефакторингу можна виконати. Деякі зміни, такі як ті, що пов'язані з проблемою, яка охоплює багато файлів, могли не бути внесені до оригінального коду через вартість; однак, якби оригінальна реалізація використовувала аспекти з самого початку, такі зміни були б менш витратними.

Те, як ми використовуємо репозиторії вихідного коду застарілих програм, створює додаткову загрозу для внутрішньої валідності. Розглянуті нами версії є основними релізами для великого набору файлів. Між двома основними релізами (грубозернистими версіями) програми два файли можуть мати різну історію версій – один може взагалі не змінюватися, тоді як інший може мати кілька версій файлів. Для файлів, які мають кілька змін між версіями, модуль у файлі може зазнати кількох змін, лише одна з яких пов'язана з перехресною проблемою. Коли ця проблема рефакторується як аспект, відбудеться зменшення кількості змін у цьому модулі. Однак, оскільки є інші зміни в модулі та пов'язаному з ним файлі, рефакторинг сам по собі не може зменшити локальність змін файлів та модулів. Більш детальний підхід враховуватиме історію версій кожного файлу окремо. Використання дрібнозернистих версій для кожного файлу, ймовірно, вплине на результати локальності змін, оскільки кожна зміна буде меншою, а деякі зміни будуть пов'язані лише з перехресними проблемами.

Конструкційна валідність зосереджена на тому, чи відображають використані показники мету дослідження. Ми порівняли оригінальні та

рефакторовані програми протягом існуючої історії редакцій, використовуючи показники на основі коду, такі як рядки коду, кількість змінених модулів та файлів, а також поширення проблем коду. Це один із підходів до вивчення впливу аспектно-орієнтованої технології на зручність обслуговування. Інші дослідники створили дві окремі програми [10], [30], одну розроблену з аспектами, а іншу - без аспектів, замість того, щоб модифікувати оригінальну програму для включення аспектів. Обслуговування моделювалося шляхом реалізації функцій, визначених у випадках використання. Ми виявили перехресні проблеми з вихідного коду та не використовували вимоги чи проектну документацію. Інший підхід полягав би у визначенні аспектів з проектної документації або вказівок на наміри розробника.

Наші результати залежать від функцій та можливостей AspectC++, які залежать від функцій C++. Єдиним ефектом, який був прямим результатом використання AspectC++, було роздуття коду, спричинене шаблонами в C++, та відносна недостатня зрілість реалізації AspectC++. AspectC++ базується на дизайні та цілях AspectJ [34], але відмінності між C++ та Java відображаються в AspectC++. Наприклад, оскільки в C++ відсутнє відображення, доступ до інформації про тип параметрів забезпечується через статичний API часу компіляції в AspectC++. Мовні відмінності між C++ та Java, такі як вказівники, управління пам'яттю та перевантаження операторів, впливають на те, як можна реалізувати шаблони проектування, такі як singleton, і призводять до того, що AspectC++ не підтримує деякі функції, такі як доступ до атрибутів класу за допомогою методів отримання та встановлення. Крім того, переплетення під час виконання можливе в AspectJ, але не в AspectC++, що може вплинути на результати продуктивності. Дослідження, що використовують іншу основну мову коду та іншу аспектно-орієнтовану мову, можуть зіткнутися з різними проблемами та перевагами рефакторингу.

Заглядаючи в майбутнє, траєкторія майбутніх дослідницьких зусиль спрямована на всебічне вивчення наслідків автоматизованого рефакторингу коду для продуктивності розробників та різних показників якості коду. Крім того, поставлена амбітна мета – безперешкодно інтегрувати запропоноване рішення у

складну структуру конвеєрів безперервної інтеграції/безперервної доставки (CI/CD), тим самим створюючи динамічну та автоматизовану структуру для рефакторингу коду та оновлення документації. Передбачуване майбутнє передбачає не лише глибоке розуміння теоретичних основ, але й активне прагнення до практичних впроваджень, які обіцяють сформувати майбутній ландшафт практик підтримки коду.

3.4 Висновок по розділу

У третьому розділі було розглянуто практичну реалізацію та оцінку ефективності рішень для рефакторингу й підтримки програмного коду. Проведено розробку та інтеграцію інструменту підтримки рефакторингу, що дозволяє автоматизувати частину процесів аналізу та вдосконалення коду. Досліджено практичне застосування методів рефакторингу до існуючих програмних систем, що сприяло підвищенню якості, зрозумілості та підтримуваності коду. Також виконано оцінку ефективності проведеного рефакторингу та аналіз можливих загроз валідності отриманих результатів. Підсумкові результати підтверджують доцільність використання систематичного рефакторингу для підтримки довготривалої якості програмного забезпечення.

ВИСНОВКИ

У ході виконання дипломної роботи було досліджено підходи до рефакторингу та підтримки існуючого програмного коду, що є важливим етапом життєвого циклу програмного забезпечення. Робота спрямована на підвищення якості коду, його зрозумілості, гнучкості та зручності супроводу. У процесі дослідження було проаналізовано сучасні підходи до рефакторингу, визначено основні проблеми, що виникають у процесі еволюції програмних систем, а також розглянуто інструменти та методи їх вирішення. Отримані результати дозволили сформувати цілісне бачення процесу підтримки коду та розробити практичні рекомендації щодо його вдосконалення.

На початковому етапі було проведено аналіз предметної області, визначено основні поняття рефакторингу, його цілі та принципи. Особливу увагу приділено проблемам, які виникають у процесі підтримки програмного забезпечення, зокрема накопиченню технічного боргу, появі «code smells» та антипатернів. Це дозволило окреслити ключові виклики, пов’язані з підтримкою існуючого коду, та обґрунтувати необхідність застосування системного підходу до його вдосконалення.

У другому розділі було досліджено методи та інструменти рефакторингу, включаючи підходи до виявлення проблем у коді, аналіз структури програмних модулів та застосування шаблонів рефакторингу. Розглянуто сучасні засоби автоматизації, які дозволяють підвищити ефективність процесу супроводу коду, а також запропоновано методологію проведення рефакторингу, що включає послідовність дій від виявлення проблем до оцінки результатів.

Практична частина роботи полягала у реалізації підходів до рефакторингу на прикладі інструменту підтримки, зокрема розширення для середовища розробки. Було продемонстровано можливості інтеграції інструментів аналізу коду у процес розробки, що дозволяє автоматизувати виявлення проблемних ділянок та підвищити продуктивність розробників. Застосування запропонованих підходів до реального програмного коду дало змогу покращити

його структуру, зменшити складність та підвищити читабельність.

У процесі роботи було проведено оцінку ефективності рефакторингу, яка показала, що систематичне застосування відповідних методів дозволяє значно знизити рівень технічного боргу, підвищити якість програмного продукту та спростити його подальший розвиток. Разом з тим, визначено потенційні загрози валідності дослідження, зокрема залежність результатів від обраних інструментів, специфіки програмного коду та суб'єктивності оцінювання якості.

Загалом, результати роботи підтверджують, що використання сучасних підходів до рефакторингу та підтримки існуючого коду є ефективним засобом підвищення якості програмного забезпечення. Запропоновані методи та інструменти можуть бути використані у практиці розробки програмних систем для оптимізації процесу супроводу, зменшення витрат на підтримку та підвищення продуктивності команд розробників.

У подальшому доцільним є розширення функціональності інструментів рефакторингу, впровадження більш глибокого аналізу коду з використанням інтелектуальних методів, а також інтеграція таких рішень у сучасні процеси безперервної інтеграції та доставки (CI/CD), що дозволить ще більше підвищити ефективність розробки програмного забезпечення.

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: A Craftsman's Guide to Software Structure and Design*. pearson.com. https://www.pearson.com/
3. Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. pearson.com. https://www.pearson.com/
4. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns*. pearson.com.
5. Kerievsky, J. (2004). *Refactoring to Patterns*. addison-wesley.com.
6. Feathers, M. (2004). *Working Effectively with Legacy Code*. pearson.com.
7. McConnell, S. (2004). *Code Complete (2nd ed.)*. microsoftpressstore.com.
8. Beck, K. (2003). *Test-Driven Development: By Example*. addison-wesley.com.
9. Fowler, M. (2025). Refactoring Catalog. martinfowler.com. https://refactoring.com/catalog/
10. Microsoft. (2025). .NET Code Analysis and Refactoring. learn.microsoft.com. https://learn.microsoft.com/en-us/dotnet/fundamentals/code-analysis/
11. Microsoft. (2025). Visual Studio Refactoring Tools. learn.microsoft.com. https://learn.microsoft.com/en-us/visualstudio/ide/refactoring/
12. Microsoft. (2025). Roslyn Analyzers Overview. learn.microsoft.com. https://learn.microsoft.com/en-us/dotnet/fundamentals/code-analysis/overview
13. SonarSource. (2025). SonarQube Documentation. sonarsource.com. https://docs.sonarsource.com/
14. ESLint. (2025). Documentation. eslint.org. https://eslint.org/docs/latest/
15. JetBrains. (2025). ReSharper Documentation. jetbrains.com. https://www.jetbrains.com/resharper/documentation/

16. Visual Studio Code. (2025). Extensions API. code.visualstudio.com. https://code.visualstudio.com/api

17. Fowler, M. (2006). Continuous Integration. martinfowler.com. https://martinfowler.com/articles/continuousIntegration.html

18. Humble, J., & Farley, D. (2010). *Continuous Delivery*. addison-wesley.com.

19. Kim, G., Humble, J., Debois, P., & Willis, J. (2021). *The DevOps Handbook*. itrevolution.com.

20. Spinellis, D. (2006). *Code Quality: The Open Source Perspective*. addison-wesley.com.

21. Lanza, M., & Marinescu, R. (2006). *Object-Oriented Metrics in Practice*. springer.com.

22. Brown, N., et al. (2010). Managing Technical Debt. IEEE Software. https://doi.org/10.1109/MS.2010.121

23. Suryanarayana, G., Samarthyam, G., & Sharma, T. (2014). *Refactoring for Software Design Smells*. elsevier.com.

24. Fowler, M. (2019). Code Smells. martinfowler.com. https://martinfowler.com/bliki/CodeSmell.html

25. Parnas, D. (1972). On the Criteria To Be Used in Decomposing Systems into Modules. CACM.

26. Pressman, R. (2014). *Software Engineering: A Practitioner's Approach*. mcgraw-hill.com.

27. Sommerville, I. (2016). *Software Engineering (10th ed.)*. pearson.com.

28. Google. (2025). Engineering Practices Documentation. google.github.io. https://google.github.io/eng-practices/

29. OWASP. (2024). Secure Coding Practices. owasp.org. https://owasp.org/

30. GitHub. (2025). Code Review Best Practices. docs.github.com. https://docs.github.com/en/pull-requests

31. Atlassian. (2025). Code Refactoring Guide. atlassian.com. https://www.atlassian.com/continuous-delivery

32. Microsoft. (2025). .NET Best Practices. learn.microsoft.com. https://learn.microsoft.com/en-us/dotnet/standard/

33. NDepend. (2025). Code Quality Tool Documentation. ndepend.com. https://www.ndepend.com/docs

34. PMD. (2025). Static Code Analyzer. pmd.github.io. https://pmd.github.io/

35. Checkstyle. (2025). Documentation. checkstyle.sourceforge.io. https://checkstyle.sourceforge.io/

36. CodeClimate. (2025). Maintainability Metrics. codeclimate.com. https://codeclimate.com/

37. McCabe, T. (1976). A Complexity Measure. IEEE Transactions on Software Engineering.

38. Chidamber, S., & Kemerer, C. (1994). Metrics Suite for OO Design. IEEE Transactions.

39. Fowler, M. (2013). Microservices. martinfowler.com. https://martinfowler.com/articles/microservices.html

40. Newman, S. (2021). *Building Microservices (2nd ed.)*. oreilly.com.

БІБЛІОГРАФІЧНА ДОВІДКА

Тема бакалаврської роботи: " Підходи до рефакторингу та підтримки існуючого коду"

Обсяг пояснювальної записки: 70 аркушів

Дата закінчення бакалаврської роботи 10 червня 2026р.

Підпис студента _____